Coverity Analysis 2020.12 User and Administrator Guide

Coverity Analysis supports source code analyses and third-party extensions.

Table of Contents

Part 1. Overview

Coverity Analysis is built on top of a set of foundational technologies that support the use of Coverity checkers to detect quality issues (also called quality defects), security issues (potential security vulnerabilities), Test Advisor issues (test issues), and Java runtime defects (Dynamic Analysis issues). Issues found by Coverity Analysis can cause data corruption, unpredictable behavior, application failure, and other problems. Coverity Analysis analyzes code bases to help minimize the number of software issues before your applications reach the customer.

After analyzing source code with Coverity Analysis, developers can view and manage the issues it finds through Coverity Connect or Coverity Desktop. It is also possible to run local analyses of source code using Coverity Desktop. See *Coverity Platform 2020.12 User and Administrator Guide* , *Coverity 2020.12 for Eclipse, Wind River Workbench, and QNX Momentics: User Guide* , *Coverity Desktop 2020.12 for Microsoft Visual Studio: User Guide* , and *Coverity Desktop 2020.12 for IntelliJ IDEA and Android Studio: User Guide* for details.

Scope

This guide covers tasks for setting up and running static quality and security analyses in a centralized (server-based) build system. To set up and run test analyses, see *Test Advisor 2020.12 User and Administrator Guide*. To set up and run Java dynamic analyses, see *Dynamic Analysis 2020.12 Administration Tutorial* .

This guide also provides details on extending the set of compilers that are available to Coverity Analysis (see Part 5, “Using the Compiler Integration Toolkit (CIT)”) and on using Coverity Analysis to commit third-party bugs and issues to Coverity Connect (see Part 6, “Using the Third Party Integration Toolkit ”),

Audience

The audience for this guide is administrators (including build engineers and tools specialists) and power users who set up and run the Coverity analyses in an integrated build environment. For details, see Chapter 1.1, *Roles and Responsibilities*.

For installation instructions and supported platform details, see *Coverity 2020.12 Installation and Deployment Guide* . Coverity Analysis component accessibility varies by license.

To see how Coverity products work together, see Coverity Development Testing.

Chapter 1.1. Roles and Responsibilities

Coverity Analysis tasks and responsibilities vary based on the role you play in your organization:

Coverity Analysis Administrators

Coverity Analysis administrators install, provision, configure, and integrate Coverity Analysis into the software build cycle. The Coverity Analysis administrator role has the following sub-roles:

- IT engineers ensure that Coverity Analysis has the hardware and software resources required to run optimally. They add machines and other hardware, back up data, and install software.
- Build engineers automate and integrate Coverity Analysis into existing build processes. They set up the system to automatically run Coverity Analysis over source code, generate defect reports, and make reports available to the Coverity Analysis consumers. In some organizations, they might also help developers to run Coverity Analysis on their local repositories.

Coverity Analysis Consumers

Coverity Analysis consumers use Coverity Analysis results to assess and improve software. The Coverity Analysis consumer role has the following sub-roles:

- Developers and team leads are the primary consumers of Coverity Analysis analysis results. Both examine and prioritize software issues in Coverity Connect or Coverity Desktop. Developers also fix the issues by using the information that Coverity Connect provides about them. Sometimes these consumers work with the Coverity Analysis administrator to optimize the effectiveness of Coverity Analysis analyses, for example, by changing the set of checkers that is enabled.

  Coverity Connect is a web-based application for examining and handling defect data. Coverity Desktop is a plug-in to the Eclipse or Visual Studio Integrated Development Environment (IDE). Coverity Desktop provides most of the same management capabilities as Coverity Connect and allows you to modify source code to fix defects. A developer can examine, prioritize, triage, and fix defects across related code bases, such as multiple development branches or different target platforms.
- Managers monitor defect reports and trends, to determine the overall software quality and trends. They might also monitor and manage the personnel responsible for fixing defects.

Coverity Analysis Power Users

Coverity Analysis power users are typically developers who understand both your software development environment and Coverity Analysis. These power users help communicate needs and requirements to consumers and administrators. Common tasks for this role include assessing the need for custom models (see Section 1.4.6.2, “Using custom models to improve analysis results”) and determining what checkers to enable (see Section 1.4.6.1, “Enabling Checkers”). Development tasks that pertain to extending the functionality of Coverity Analysis in other ways also fit into this role.

Chapter 1.2. Use Cases

Coverity Analysis supports a number of use cases:

Running analyses with Coverity Analysis

- Each Coverity Analysis analysis uses a set of commands that vary by programming language and analysis type. To get started with Coverity Analysis, you can run Coverity Analysis analyses from the command line or by using the GUI-based Coverity Wizard (see *Coverity Wizard 2020.12 User Guide* for details). For information about common analysis tasks, see Chapter 1.4, *Coverity analyses*.

  Once you are familiar with the basics of Coverity Analysis, you need to create a server-based script that regularly runs the commands needed to analyze your code base (see Part 3, “Setting up Coverity Analysis for use in a production environment”).

Enabling/Disabling Checkers

- Coverity Analysis uses checkers to analyze your code base for specific types of issues. By default, Coverity Analysis enables a certain set of checkers. To control the depth and nature of the analysis, you can work with Coverity Analysis power users (see Chapter 1.1, *Roles and Responsibilities*) to determine whether to change the set of checkers that are enabled. For details, see Section 1.4.6.1, “Enabling Checkers”.

  > **Custom checkers**
  >
  > It is possible to extend the set of checkers that is available to Coverity Analysis. See the document *Learning to Write CodeXM Checkers*.

Using Custom Models of Functions and/or Methods

- A model summarizes the behavior of a function or method. Coverity Analysis allows you to integrate custom models into the analysis. For details, see Section 1.4.6.2, “Using custom models to improve analysis results”.

Extending C/C++ compiler compatibility

- Coverity Analysis supports a number of C/C++ compilers (see *Coverity 2020.12 Installation and Deployment Guide*). To extend the set of C/C++ compilers that are compatible with Coverity Analysis, see Part 5, “Using the Compiler Integration Toolkit (CIT)”.

Using Coverity Analysis to commit third-party issues to the Coverity Connect database

- In addition to supporting the management of software issues found by Coverity Analysis, Coverity Connect supports issues found by third-party tools. The Third Party Integration Toolkit (see Part 6, “Using the Third Party Integration Toolkit ”) relies on Coverity Analysis to commit third-party defects to the Coverity Connect database.

Chapter 1.3. Language Support

Coverity Analysis support can vary by programming language.

Table 1.3.1. Support by Language

Language	Capture Mode	Coverity Desktop Analysis	Coverity Extend SDK	CodeXM	Churn	Coding Standards & Vulnerability Reports	Language Versions	Notes
C/C++	Build capture	Yes	Yes	Yes	<5%	AUTOSAR C++14 R19-03 ISO/IEC TS 17961: 2013, Cor 1 2016 MISRA-C 2004: 1st Ed 2004.10, 2nd Ed with Tech Cor 1 2008.07 MISRA-C 2012: 1st Ed 2013.03, Amendment 1 2016.04, Tech Cor 1 2017.06, Amendment 2 2020.02 MISRA-C++ 2008: 2008.06 SEI CERT C: 2016 Ed, POSIX rules, L1 and some L2 recs SEI CERT C++ 2016 Ed CWE Top 25 CWE On the Cusp	C++98 C++03 C++11 C++14 C++17 C89 C99 C11	
C#	Build capture	Yes	Yes	Yes	<5%	CWE Top 25 OWASP Top 10	Up to C# 8	Less than 5% churn is expected for build capture.
C#	Buildless capture	No	Yes	Yes	No bound	CWE Top 25 OWASP Top 10	Up to C# 8	
CUDA	Build capture	Yes	No	Yes	No bound	AUTOSAR C++14 R19-03 ISO/IEC TS 17961: 2013, Cor 1 2016 MISRA-C 2004: 1st Ed 2004.10, 2nd Ed with Tech Cor 1 2008.07 MISRA-C 2012: 1st Ed 2013.03, Amendment 1 2016.04,		

Language	Capture Mode	Coverity Desktop Analysis	Coverity Extend SDK	CodeXM	Churn	Coding Standards & Vulnerability Reports	Language Versions	Notes
						Tech Cor 1 2017.06, Amendment 2 2020.02 MISRA-C++ 2008: 2008.06 SEI CERT C: 2016 Ed, POSIX rules, L1 and some L2 recs SEI CERT C++ 2016 Ed CWE Top 25 CWE On the Cusp		
Fortran	Standalone	No	No	No	No bound		Fortran 77 Fortran 90 Fortran 95 Fortran 2003 Fortran 2008 Fortran 2018	Fortran Syntax Analysis performs filesystem capture through the `cov-run-fortran` command.
Go	Build capture	Yes	No	Yes	No bound	OWASP Top 10	Go 1.13–1.14.x	Support for Go 1.13 is deprecated as of 2020.12 and will be removed in a future release.
Java	Build capture	Yes	Yes	Yes	<5%	CWE Top 25 CWE On the Cusp OWASP Top 10 OWASP Mobile Top 10 SEI CERT Java: L1 and some L2 rules	Up to Java 14	Less than 5% churn is expected for build capture.
Java	Buildless capture/ Filesystem capture	No	Yes	Yes	No bound	CWE Top 25 CWE On the Cusp OWASP Top 10 OWASP Mobile Top 10	Up to Java 14	
JavaScript	Buildless capture/ Filesystem capture	Yes	Yes	Yes	<5%	OWASP Top 10	ECMAScript 5–11	ECMAScript 11 is also known as ECMAScript 2020.
Kotlin	Build capture	Yes	No	No	No bound	OWASP Mobile Top 10	Kotlin 1.3–1.3.71	
Objective-C/C++	Build capture	Yes	No	No	No bound			

Language	Capture Mode	Coverity Desktop Analysis	Coverity Extend SDK	CodeXM	Churn	Coding Standards & Vulnerability Reports	Language Versions	Notes
PHP	Buildless capture/ Filesystem capture	Yes	No	No	No bound	OWASP Top 10	PHP 5.5.x PHP 5.6.x PHP 7.0.0	Support for PHP 5.x is deprecated as of 2020.12 and will be removed in a future release.
Python	Buildless capture/ Filesystem capture	Yes	No	Yes	No bound	OWASP Top 10	Python 2.7.x and 3.x– 3.8	Support for Python 2.7 is deprecated as of 2020.12 and will be removed in a future release.
Ruby	Buildless capture/ Filesystem capture	Yes	No	No	No bound	OWASP Top 10	Matz's Reference Impl. (MRI) 1.9.2–2.5.1 and equivalents	
Scala	Build capture	Yes	No	No	No bound		Scala 2.12.x	
Swift	Build capture	Yes	No	No	No bound	OWASP Mobile Top 10	Swift 5.3	
TypeScript	Buildless capture/ Filesystem capture	Yes	No	No	No bound	OWASP Top 10	TypeScript 1.0–4.0	
Visual Basic	Build capture	Yes	No	No	No bound	CWE Top 25 OWASP Top 10	Visual Basic .NET	

-

Note

The preceding table combines supported coding standards (AUTOSAR, MISRA, and CERT) and supported vulnerability reports (CWE and OWASP) into one column in order to conserve space.

Not all rules and directives in the listed coding standards are supported. For information on how to run a code analysis using one of the supported coding standards, see Chapter 2.4, *Running coding standard analyses*.

Coverity Analysis automatically finds defects listed in the supported vulnerability reports when you enable all security checkers.

For more information about:

- Coverity Desktop Analysis, see *Coverity Desktop Analysis 2020.12: User Guide* .
- Checker development and the CodeXM language and libraries, see *Learning to Write CodeXM Checkers* .
- Checker development and the Coverity Extend SDK, see *Coverity Extend SDK 2020.12 Checker Development Guide* .
- Build capture mode, see Section 1.4.2.1, “Build capture (for compiled languages)”.
- Buildless capture mode, see Section 1.4.3, “Buildless capture”.
- Filesystem capture, see Section 1.4.2.2, “Filesystem capture”.

For checkers, see "Checkers by Programming Language" in the Coverity 2020.12 Checker Reference. For deployment information, see the following table.

Table 1.3.2. Deployment Considerations

Build and target platforms	Varies by language: See "Supported Platforms" for details.	*Coverity 2020.12 Installation and Deployment Guide*
Memory Requirements	Varies by programming language: See "Coverity Analysis hardware recommendations" for details.	

Chapter 1.4. Coverity analyses

Table of Contents

This section introduces you to the common sequence of tasks needed to complete an analysis that runs in a central (server-based) build environment. For information about that environment, see Chapter 3.1, *The Central Deployment Model*. For detailed analysis procedures, refer to Part 2, “Analyzing source code from the command line ”

Prerequisites to completing an analysis

- To run an analysis, you must have a valid license to Coverity Analysis. If you have not yet set up licensing (for example, during the installation process), you can refer to Setting up Coverity Analysis licensing in the *Coverity 2020.12 Installation and Deployment Guide*.

- You need to have access to a Coverity Connect stream to which you can send your analysis results. You also need to know your Coverity Connect username, password, and the Coverity Connect host or dataport (for example, see Step 6 in Chapter 2.1, *Getting started with Coverity analyses*).

  A Coverity Connect administrator is typically responsible for setting up the stream, giving you permission to commit issues to it, and providing the other information you need. You can refer to Configuring Projects and Streams if you need set up your own stream.

  Note

  Coverity Wizard (see for details) can create a stream in Coverity Connect, rather than through the Coverity Connect UI. However, you still need to know your Coverity Connect login credentials and host. Stream configuration requires administrative permissions to Coverity Connect.

  For information about creating streams through the Coverity Connect UI, see "Configuring Projects and Streams" in the *Coverity Platform 2020.12 User and Administrator Guide*.

- Make sure that you have adequate memory for the analysis. For details, see "Coverity Analysis hardware recommendations" in the *Coverity 2020.12 Installation and Deployment Guide*.

See the *Coverity 2020.12 Command Reference* for descriptions of all the Coverity Analysis commands that are available to you.

1.4.1. The configuration

Before running an analysis, you typically generate a one-time configuration for your native compiler and/or scripting language (such as JavaScript) by using the `cov-configure` command. This configuration is used when you perform the build step of the analysis process (see Section 1.4.2, “The capture”).

Note

You do not need to generate a configuration if you are using buildless capture, as described in the next section.

The configuration provides information about the language of the source files to capture and analyze, and provides settings that are used to emulate your native compiler, its options, definitions, and version. A correct configuration allows the Coverity Analysis to properly translate the arguments of the native compiler and to parse the source code into a form that is optimized for efficient analysis during the build step of the analysis process.

1.4.2. The capture

After generating a configuration, you need to capture a binary representation of your source code to a directory (the intermediate directory) where it can be analyzed. You have three options: build capture, filesystem capture, or buildless capture.

- Use `cov-build` for build and filesystem capture.
- Use `cov-capture` for buildless capture.

Generally speaking, we recommend starting with buildless capture where possible. It requires no setup, no knowledge about how to build your projects, no third party tools installed (in many cases), and will give you satisfactory results quickly. Later, if it makes sense for you to invest more time configuring build capture to get more accurate results, you can move to build capture. The build options you have might also depend on the source language, as shown in the following table.

Language	**Build Options**
C, C++, CUDA, Go, Kotlin, Objective C, Objective C++, Scala, Swift	Use build capture.
PHP, Python, Ruby	Use buildless capture or filesystem capture.
C#, Visual Basic	• Use build capture if you are looking for the most accurate results and you are okay with integrating it into your build. • Use buildless capture if you are looking for the easiest option and your project meets the conditions outlined in "Buildless capture for C#".

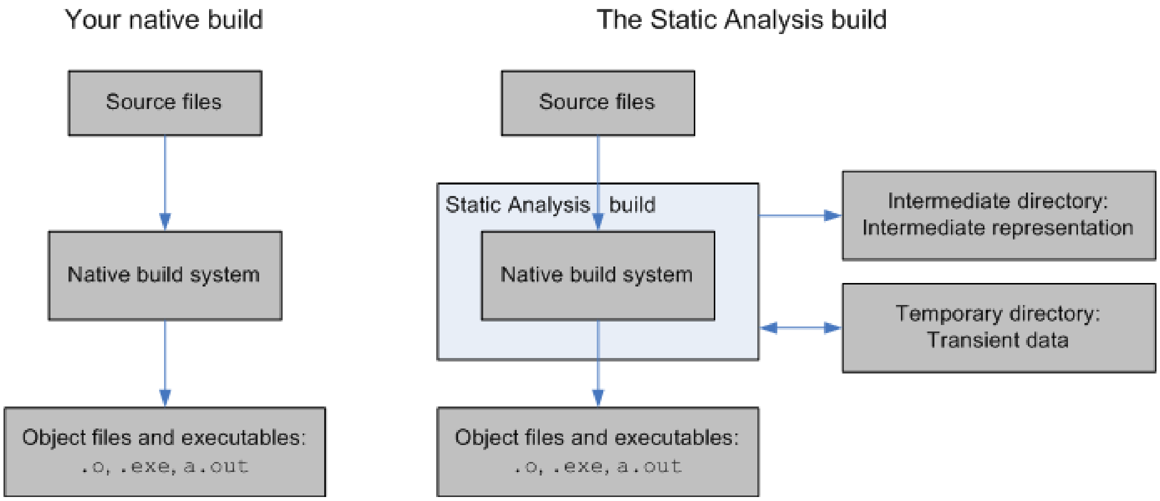
Language	Build Options
Note On macOS, C# support is limited to quality analysis, and Basic analysis is supported only for projects built with the .NET Compiler Platform.	
Java	• Use build capture if you are looking for the most accurate results and you are okay with integrating it into your build. • Use buildless capture if you are looking for the easiest option and your project meets the conditions outlined in "Buildless capture for Java". • Use filesystem capture if buildless capture does not provide satisfactory results.
JavaScript, TypeScript	• Use buildless capture if you are looking for the easiest option and your project meets the conditions outlined in "Buildless capture for JavaScript/TypeScript". • Use filesystem capture if buildless capture does not provide satisfactory results.

1.4.2.1. Build capture (for compiled languages)

Build capture is part of the overall analysis workflow for code that you need to compile, such as C/C++. The Coverity Analysis compiler builds your source code into an intermediate representation of your software. The intermediate representation is a binary form of the source code that provides Coverity Analysis with a view of the operation of your software that is optimized for the efficient analysis of software issues. Compiling all of your source code with the Coverity Analysis compiler is often an iterative process. Though Coverity Analysis makes every effort to parse the compiled code that each supported compiler generates, the nuances of a particular code base sometimes introduce subtle incompatibilities. Nevertheless, analyses of such code bases can still produce useful results.

The `cov-build` command wraps the native build to observe native compiler invocations and operations. It invokes the regular build system, and it runs the `cov-translate` command to translate the native compiler's command-line arguments to the Coverity Analysis compiler command-line arguments. For each observed native compiler call, `cov-translate` runs the Coverity Analysis compiler on the same files with the same arguments, which in turn invokes the `cov-emit` command to compile the file and outputs the intermediate representation into the intermediate directory.

Figure 1.4.1. Example: Building source with the Coverity Analysis compiler (The Coverity Static Analysis build)



The Coverity Analysis compiler requires information about build processes, dependencies, and build-related programs that are only revealed during a native build. Because `cov-build` parses and compiles each source file, first with the native compiler and then with the Coverity Analysis compiler, it takes longer to complete the `cov-build` process than it does to compile your source code with your native compiler alone.

Note that the simplest way to build source code is to use `cov-build` because you do not have to modify your native build. However, the `cov-build` command does not allow you to specify which source files to compile. Instead, it simply compiles everything the native build compiles (except for unchanged source code). If you need to specify which source code to compile, you can invoke the `cov-translate` command directly. This task is more complex and requires you to modify the native build process, but it might also provide greater build efficiency. Running `cov-translate` is also the only supported way to run the Coverity Analysis compiler on AIX systems. For more information, see Section 3.3.3, “Alternative build command: cov-translate”.

1.4.2.2. Filesystem capture

Filesystem capture emits files directly from the disk. It does not require observing a build command.

Filesystem capture uses special `cov-build` options to gather or specify a list of the files that serve as candidates for the analysis. The previous `cov-configure` step(s) determine which files are emitted to the intermediate directory for analysis.

You can combine filesystem capture with build capture into a unified workflow. For an example on how they are combined, see the build step in Chapter 2.1, *Getting started with Coverity analyses* .

1.4.2.2.1. Deployment recommendations

Your goal with filesystem capture is to emit all the files you want to analyze and none of the ones you don't want to analyze. Typically, you'll want to analyze your source code, configuration files, and any library code that your source code needs to compile or run. Typically, you don't want to emit or analyze test cases for library code, or library code your code doesn't need to run. You may not wish to emit or analyze test infrastructure, and test cases for your code.

To control which files to capture for analysis in your deployment, you should create a script or job that emits the set of files that you need to analyze. The script should be robust enough to handle the addition, deletion, and renaming of files or directories in your source repository. To specify the files, you should take one or both of these steps:

- Carefully tune `--fs-capture-search` and `--fs-capture-search-exclude-regex` options to capture the right set of files.
- Adjust your analysis job to generate the list (for example, with help from your SCM system), and use the `--fs-capture-list` option. You should also regenerate the list before every analysis so that changes to your file set are captured. For example:

```
> git ls-files > scm_files.lst
> cov-build --fs-capture-list scm_files.lst <other options>
```

During filesystem capture, the `cov-build` command attempts to emit library code to which your source refers; the `--fs-library-path` option to either `cov-configure` or `cov-build` can help it find this library code. See the sections below for language-specific advice.

1.4.2.2.2. Filesystem capture for JavaScript

The `cov-build` and `cov-analyze` steps are substantially faster if you don't emit library code that is irrelevant to your source code. In particular, do not include your entire node_modules directories in the scope of `--fs-capture-search` or `--fs-capture-list`. The cov-build command attempts to emit any library code that your source code requires (Node.js), imports (ECMAScript 6, HANA XS JS), or includes (HTML) by searching the filesystem relative to the source file; therefore, you should include only source code for which you want analysis results in the scope of `--fs-capture-search` or `--fs-capture-list` and let `cov-build` find the relevant third party library code. If your required/imported/included library code is elsewhere, point the `--fs-library-path` option at the directory containing this library code, for example specify `--fs-library-path /usr/local/node_modules`.

Emit non-minified versions of libraries. If you have both minified and non-minified versions of the same library in your source, emit only the non-minified version. However, if only the minified version is available, you should still emit it, but Coverity analysis will not report defects in it. If you concatenate source files together or otherwise package them for deployment, emit the original source code and not the packaged source code.

1.4.2.2.3. Filesystem capture for PHP

The `cov-build` command attempts to emit any library code that your source code includes or requires by searching the filesystem relative to the source file; therefore, you should include only source code for which you want analysis results in the scope of `--fs-capture-search` or `--fs-capture-list`, and let `cov-build` find the relevant third party library code. If your imported library code is elsewhere on the filesystem (for example, because you change the `include_path`), point the `--fs-library-`

`path` option at the directory containing this library code such that the path in your `include` or `require` statement resolves relative to the `--fs-library-path`).

1.4.2.2.4. Filesystem capture for Python

The `cov-build` command attempts to emit any library code that your source code imports by searching the filesystem relative to the source file; therefore, you should include only source code for which you want analysis results in the scope of `--fs-capture-search` or `--fs-capture-list` and let `cov-build` find the relevant third party library code. If your imported library code is elsewhere on the filesystem (for example, because you use a `setup.py` file or an environment variable such as `PYTHONPATH` when deploying your code), point the `--fs-library-path` option at the package root directory containing this library code.

The `cov-build` command determines the character encoding of a Python source file as follows:

- If the source file starts with a UTF-8 encoded Unicode Byte Order Mark (BOM), the source file will be parsed, analyzed, and presented in Coverity Connect as a UTF-8 encoded source file.
- If the source file contains a Python encoding declaration, the source file will be parsed and analyzed using the encoding specified by the declaration. If the encoding is supported by Coverity Connect, it will be used to present the source file in Coverity Connect.
- If the source file contains neither a UTF-8 encoded Unicode BOM nor an encoding declaration, then different character encoding types are used for parsing and analysis:

  - For Python 2 source files, ISO-8859-1 is used.
  - For Python 3 source files, UTF-8 character encoding is used.

  If the `cov-build --encoding` option is specified, the encoding that it names is used to present the source file in Coverity Connect. Otherwise, a Python 2 file will be presented as a US-ASCII encoded source file, and a Python 3 source file is presented as a UTF-8 encoded source file.

Note

Source files that contain an encoding declaration that names the MBCS or DBCS encoding are not supported.

1.4.2.2.5. Filesystem capture for Ruby

The `cov-build` command determines the character encoding of a Ruby source file as follows:

- If the source file starts with a UTF-8 encoded Unicode Byte Order Mark (BOM), then the source file will be parsed, analyzed, and presented in Coverity Connect as a UTF-8 encoded source file.
- If the source file contains a Ruby encoding declaration, the source file will be parsed and analyzed using the encoding specified by the declaration. If the encoding is one that is supported by Coverity Connect, it will be used to present the source file in Coverity Connect.

  If the `cov-build --encoding` option was specified, the encoding name will be used to present the source file in Coverity Connect.

- If the source file contains neither a UTF-8 encoded Unicode BOM nor an encoding declaration, the UTF-8 character encoding is used to parse and analyze the source file.

1.4.2.2.6. Filesystem capture for configuration and template files

The security analysis of web and mobile applications reads configuration data to understand the application architecture. It also reports insecure configuration settings and cross-sites scripting vulnerabilities that can occur with the supported template languages.

For Android and iOS applications, filesystem capture is the prescribed and recommended approach to emit all configuration files. When using filesystem capture, it should point to (and be run on) the project root directory. This is true even where the bulk of the application source code is emitted using build capture.

For Java web applications, filesystem capture is an option to emit configuration files and JavaServer Pages (JSPs). Alternatively, these files may be emitted as part of a web-application archive (WAR file) by using `cov-emit-java --webapp-archive`. For more information about this command, see the `--webapp-archive` option listed under `cov-emit-java` ↗.

For ASP.NET web applications, filesystem capture is of limited use. See Section 2.2.2, “Running a security analysis on an ASP.NET Web application”.

1.4.2.2.7. Filesystem capture for Java source

Filesystem capture is supported for Java in situations where build capture is not feasible. During filesystem capture, the build command attempts to recover from any errors or warnings caused by incomplete information. This ensures that most of the code is analyzed. Note that without build information, however, a loss in analysis fidelity is to be expected with higher rates of false positives and false negatives.

When you use filesystem capture mode, it prompts the `cov-build` command to search recursively for Java source files in a specified directory. During this process, the `cov-build` command specifies and retrieves dependencies for compiling the source code. The Java source files are then emitted to the intermediate directory for later analysis.

Note

In general, build processes may specify and retrieve dependencies for compiling the source code. They may make changes to the code base by generating or downloading new files. They may also move or transform existing files. These changes are not visible to filesystem capture, since this mode only performs a basic search for files that have pre-configured patterns. It also attempts to resolve dependencies without considering build configurations. For this reason, differences in analysis results may occur when running both build capture and filesystem capture on the same code base.

There are three steps to analyzing code using Java filesystem capture:

1. Use the `cov-configure` command, which prompts the `cov-build` command to perform the Java filesystem capture.

2. Once the Java filesystem capture is configured, run the `cov-build` command for the directory that contains the Java source files.

3. Run the `cov-analyze` command to analyze the emitted files. This step is unchanged from analysis using a regular build capture.

For more information on how to configure your build using the `cov-build` command, see *Coverity 2020.12 Command Reference*. .

1.4.2.2.8. Moving from filesystem capture to buildless capture

Now that filesystem capture is deprecated, you will want to start moving to using buildless capture, which is the new preferred method. The primary difference is that for buildless capture we use the tool `cov-capture` instead of `cov-build`. Buildless capture provides the equivalent functionality to filesystem capture, but is easier to use. This section explains how to make the switch.

To take the simplest possible example, a filesystem capture invocation would look like this:

```
% cov-configure --javafs
% cov-build --dir idir --no-command --fs-capture-search src
```

The equivalent invocation using buildless capture looks like this:

```
% cov-capture --dir idir --source-dir src
```

As you can see, with buildless capture, you don't need to configure any compilers: this is all taken care of automatically by `cov-capture`.

The following sections provide additional examples of differences between filesystem capture and buildless capture. For detailed information, see the `cov-capture --help` output

1.4.2.2.8.1. Specifying custom file patterns

If you previously specified custom file matching patterns with `--file-glob` or `--file-regex`, these are now specified to `cov-capture`. For example if you previously ran the following command:

```
% cov-configure --comptype javascript --file-glob '*.js'
```

Using buildless capture you would instead add a similar argument to your `cov-capture` invocation:

```
% cov-capture --dir idir --source-dir src --file-glob javascript='*.js'
```

1.4.2.2.8.2. Specifying additional library paths

If you previously configured your compiler with additional library paths as shown next:

```
% cov-configure --javafs --fs-library-path path/to/classes
                      --fs-library-path path/to/my.jar
```

Using buildless capture, these additional library paths are simply added to your `cov-capture` invocation:

```
% cov-capture --dir idir --source-dir src --library-dir java=path/to/classes
                     --library-file java=path/to/my.jar
```

1.4.2.2.8.3. Specifying the search directory

Using filesystem capture, you specified the search directory with `--fs-capture-search`:

```
% cov-build --dir idir --no-command --fs-capture-search <dir>
```

Using buildless capture, you use the better named `--source-dir` argument:

```
% cov-capture --dir idir --source-dir <dir>
```

1.4.2.2.8.4. Specifying a list of source files

Using filesystem capture, you could specify a list of source files with `--fs-capture-list`:

```
% cov-build --dir idir --no-command --fs-capture-list <file-list>
```

Using buildless capture, there is a similar argument `--source-list`:

```
% cov-capture --dir idir --source-list <file-list>
```

1.4.2.2.8.5. Listing matched files

Using filesystem capture, you could display what files would be captured without actually capturing them with this command:

```
% cov-build --dir idir --no-command --fs-capture-search <src> --fs-capture-just-print-
matches
```

You can also do this with buildless capture:

```
% cov-capture --dir idir --source-dir <src> --just-print-matches
```

1.4.2.2.8.6. Excluding files

Using filesystem capture, you could exclude files from the capture as follows:

```
% cov-build --dir idir --no-command --fs-capture-search <src>
                         --fs-capture-search-exclude-regex <regex>
```

Using buildless capture, you would do it like this:

```
% cov-capture --dir idir --source-dir <src> --exclude-regex java="<regex>"
```

The main differences for buildless capture are the following:

- Regexes are per language. The above example applies to java.
- You can use globs with `--exclude-glob`.
- The regex/glob is matched relative to the specified `--source-dir`.

1.4.2.2.8.7. Controlling what languages are captured

Filesystem capture would only capture the languages you had configured. So if you only configured `--javafs`, it would only capture Java.

By default, buildless capture tries to capture all languages. You can control what languages are captured by explicitly specifying them. For example, the following invocation captures only Java and JavaScript files.:

```
% cov-capture --dir idir --source-dir src --language java --language javascript
```

Additionally, many languages have other languages automatically associated with them. For example, configuring `--javafs` also selects `--jsp`. To disable some of these associated languages, use the `--no-friend-language` argument. This prevents buildless capture from automatically configuring the capture of JSP files along with Java.

```
% cov-capture --dir idir --source-dir src --language java --no-friend-language jsp
```

1.4.2.2.8.8. Emit failures and parse error threshold

The `--return-emit-failures` and `--parse-error-threshold` options are the same for filesystem and buildless capture. The following command for filesystem capture:

```
% cov-build --dir idir --no-command --fs-capture-search src
                        --return-emit-failures --parse-error-threshold 50
```

Looks like this for buildless capture:

```
% cov-capture --dir idir --source-dir src --return-emit-failures --parse-error-
threshold 50
```

1.4.2.2.8.9. Build and filesystem capture in a single invocation

Using filesystem capture it was possible to run a build capture and a filesystem capture in a single step like so:

```
% cov-build --dir idir --fs-capture-search <dir> ./mybuild.sh
```

This is not currently possible with buildless capture. Instead, you will need to invoke `cov-build` then `cov-capture`:

```

# Build Capture
% cov-configure --swift
% cov-build --dir idir ./mybuild.sh

# Buildless Capture
% cov-capture --dir idir --config-dir config-dir
```

1.4.3. Buildless capture

Buildless capture is the easiest to use of the three capture mechanisms (build, buildless and filesystem).

- It imposes minimal user and environment requirements:
  - No build is required or executed.
  - Third party tools are provided in many cases (maven, npm, etc.) so you don't have to install them.

- It requires minimal user knowledge.
  - You need to supply only the location of the project on disk or in a git repository.
  - You do not need to know how to build your project, how to check a project out, how to download project dependencies, which source files to include or exclude from the analysis, or which language the project uses.
- It eliminates steps and automates processes: source code is automatically checked out, Coverity configuration is not needed, project dependencies are automatically calculated and downloaded, source file inclusion/exclusion lists are automatically calculated, and source code for supported languages is detected and processed.

Buildless capture has four primary modes of operation: Project Mode, SCM Mode, Source Mode, and Config Mode.

- Project Mode and SCM Mode

  In these modes, buildless capture works by recursively searching for project files under a specific project directory (after optionally checking out your repository). It recognizes Java (Maven/Gradle), JavaScript/TypeScript (yarn/npm/bower) and C# .NET Core (.NET Core Command-Line Interface) projects. Information is gathered from each project file. This information is then used to capture each project.
- Source Mode and Config Mode

  In these modes buildless capture works by recursively searching for source files based on the file extension.

Follow these steps to analyze code using buildless capture:

1. Run the `cov-capture` command for the repository/directory that contains your project(s).
2. Run the `cov-analyze` command to analyze the emitted files. This step is unchanged from analysis using a regular build or filesystem capture.

The following sections discuss requirements for each language analysis that supports buildless capture.

1.4.3.1. Buildless capture for C#

For C#, buildless capture is only supported on Windows and for projects that make use of the .NET Core SDK.

C# projects can be configured to be built using two different SDK types: either .NET Core SDK, or the .NET Framework SDK. To tell whether a given project uses the .NET Core SDK, open one of the project’s `.csproj` files, and at or near the top of the file, look for lines similar to the following:

Table 1.4.1. Supported frameworks for C# projects

Supported	`<Project Sdk="Microsoft.NET.Sdk”>`
	`<Project Sdk="Microsoft.NET.Sdk.Web">`

Not supported	`<Project DefaultTargets="Build" ToolsVersion="4.0" xmlns="http://schemas.microsoft.com/[...]">`

Important

A project can consist of multiple `.csproj` files (C# projects). If this is the case, only those C# projects identified as supported will be captured.

In addition, for C# buildless capture requires .NET Core SDK version 2.0, 2.1, or 2.2.

To make sure that a compatible .NET Core SDK is available, either you can install a system .NET Core SDK yourself, or you can run the Coverity installer and choose the *.NET Core SDKs* component.

Caution

Some C# projects specify that a particular SDK version be used, so the .NET Core SDKs component installed by the Coverity installer might not be suitable for capturing such a project. In these cases, you should manually install a .NET Core SDK of the version that the project specifies.

A C# project can also target one or more target frameworks that indicate where the project author intends the project to run. There are two different types of target frameworks: NuGet package-based frameworks, and non-NuGet package-based frameworks. The .NET Framework is a non NuGet package-based framework. This means that you need to install the appropriate .NET Framework targeting pack before you run buildless capture. For NuGet package-based frameworks, the relevant framework is downloaded automatically, just like any other NuGet package.

Note

Buildless capture for C# downloads the dependencies for each project being captured. Because of this, typically an Internet connection that allows access to the relevant `NuGet` repositories is required. As an alternative, you can download the dependencies in advance.

Caution

Build processes might make changes to the code base by generating or downloading new files. They might also move or transform existing files. These changes are not visible to buildless capture. For this reason, differences in analysis results might occur when running both build capture and buildless capture on the same code base.

Caution

Microsoft regularly releases new versions of the .NET Core SDK. Buildless capture might work with these versions, but no testing will have been done to guarantee support.

For documentation on .NET Core from Microsoft see: https://docs.microsoft.com/en-us/dotnet/core/.

1.4.3.2. Buildless capture for Java

The following conditions must be met for buildless capture for Java:

- Your Java code is part of a Maven or Gradle project with the following properties:
  - Maven: Can be built with Maven 2.x or newer
  - Gradle: Can be built with Gradle 1.2 or newer

If your project does not meet these requirements you may not use Project or SCM mode, however you can use Source mode to capture such projects.

Buildless capture for Java also captures JSPs.

Note

Build processes might make changes to the code base by generating or downloading new files. They might also move or transform existing files. These changes are not visible to buildless capture. For this reason, differences in analysis results might occur when running both build capture and buildless capture on the same code base.

1.4.3.3. Buildless capture for JavaScript/TypeScript

The following conditions must be met for buildless capture for JavaScript/TypeScript:

- Your JavaScript/TypeScript code is part of a yarn, npm or bower project.
- You have git installed to capture a bower project (bower, npm, and yarn don’t need to be installed because they are bundled with Coverity).

If your project does not meet these requirements you may not use Project or SCM mode, however you can use Source mode to capture such projects.

Note

When you capture JavaScript, TypeScript source code will also be captured.

1.4.3.4. Buildless capture for PHP, Ruby, and Python

You can use project mode or source mode to capture these languages. Both modes work the same for these languages. Buildless capture simply searches for any source files for these languages; nothing is done with project files or dependency downloading.

1.4.4. The analysis

After building your source with or without `cov-build` (for the former build process, see Section 1.4.2, “The capture”), you run Coverity analysis on your source code to find software issues. You use `cov-analyze` to analyze code and scripts written in all supported programming languages.

1.4.5. The commit

After completing the analysis (see Section 1.4.4, “The analysis”), you commit (or push) the analysis results to Coverity Connect so that developers and team leads can view, manage, and fix software issues they own in an efficient way.

You send the results to a stream in the Coverity Connect database using the `cov-commit-defects` command. Before you can commit analysis results to Coverity Connect, some Coverity Connect configuration tasks must take place (see Prerequisites to completing an analysis) so that developers can view the issues they own in Coverity Connect (and, if set up, Coverity Desktop). See Understanding the primary Coverity Connect workflows for information about using Coverity Connect to view and manage issues.

1.4.6. Tasks that support Coverity analyses

To support the procedures described in Chapter 1.4, *Coverity analyses*, Coverity Analysis administrators sometimes decide to perform one or more of the tasks described in this section.

1.4.6.1. Enabling 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.

For details on checker enablement, refer to the *Enabling Checkers* section of the *Coverity Checker Reference*

1.4.6.2. Using custom models to improve analysis results

A custom model is a piece of source code that is written by a developer to replace the actual implementation of a function or method. Custom models can lead to a more accurate analysis by helping Coverity Analysis find more issues and eliminate false positive results. Candidates for modeling include functions and methods in your source code that the analysis interprets in an unexpected way (see Section 1.4.6.2.2, “Using models to tune the interprocedural analysis”) and/or functions and methods in third-party libraries that Coverity does not model (Section 1.4.6.2.1, “Using models to mimic functions or methods that lack source code”).

After a developer writes a custom model (for details, see Models, Annotations, and Primitives in the *Coverity 2020.12 Checker Reference*), you (the administrator) need to include it in the analysis configuration by running the `cov-make-library` command (see Section 2.6.3.2, “Adding custom models with cov-make-library” and/or Section 2.6.3.3.2, “Using cov-make-library”). For C#, see Models and Annotations in C# in the *Coverity 2020.12 Checker Reference*. For Swift, see Models and Annotations in Swift in the *Coverity 2020.12 Checker Reference*. The `cov-make-library` command creates a file called `user_db` that you need to include in the script that runs the Coverity Analysis analysis.

1.4.6.2.1. Using models to mimic functions or methods that lack source code

Most project code bases include a number of library functions or methods without including their source code. Coverity Analysis cannot always analyze a function or method if its source code is not in the code base.

For example, Coverity Analysis cannot analyze certain third-party binary libraries that are linked to a program. Coverity Analysis *can* analyze Java Virtual Machine (JVM) and .NET bytecode. In either case, it does not report defects in binary libraries, but analyzing or modeling libraries improves the accuracy of the source code analysis.

Coverity Analysis ships with models for most standard libraries that pertain to the languages and platforms that Coverity Analysis supports. You do not have to model these libraries and in general you should not alter the models provided. (You can the examine source code for these models to learn more about writing your own custom models.) Due to the limitations of interprocedural analysis, you might need to perform some tuning: See Section 1.4.6.2.2, “Using models to tune the interprocedural analysis”.

To improve the analysis of code that uses functions or methods from nonstandard libraries, you can add custom models that emulate these functions.

Tip

The most common and useful custom models are allocators/deallocators and *killpaths* (functions that terminate execution). Resource leaks cannot be found without allocation models. If Coverity Analysis does not find any resource leaks, you probably need to create allocation models for every function that behaves as an allocator and deallocator.

If Coverity Analysis generates many false positives, it might mean that there are missing killpath models. For more information, see Adding a Killpath to a Function to Abort Execution in the *Coverity 2020.12 Checker Reference*.

1.4.6.2.2. Using models to tune the interprocedural analysis

For its cross-procedure source code analysis, Coverity Analysis infers a model of each function, whether from the actual source code or from a handwritten model. The engine automatically detects those cases where a constant is returned, a variable holding a constant is returned, or a comparison between the return code and a constant indicates the return value.

If the contextual behavior involves operations that are more complex than assignments to constants, comparisons with constants, and simple arithmetic, Coverity Analysis might not correctly infer the abstract behavior of the function or method without additional assistance. In such a case, it is necessary to create models that provide directions to Coverity Analysis.

Examples: C/C++ interprocedural contexts detected by the Coverity Analysis analysis

- In the following example, Coverity Analysis automatically infers that this function returns 0 when memory is not allocated.

```
// Basic return value dependence:
void* malloc(size_t sz)
{
    void* allocated_ptr;
    if (<detect out of memory condition>) {
        return 0;
    }
    allocated_ptr = <get pointer from Operating System>;
    return allocated_ptr;
}
```

- In the following function, `ptr` is only dereferenced when `flag` is equal to 9. In general, whenever Coverity Analysis sees a constant directly or can, through assignments or comparisons, determine that a variable is compared against a constant, it will note the constant and the comparison type (equal to, not equal to, less than, or greater than) in the function's behavioral model.

```
// Basic argument dependency
void dereference_pointer(int* ptr, int flag)
{
    if (flag == 9)
        *ptr = 9;
        return;
}
```

Coverity Analysis does not track context based on the value of global or static, file-scope variables. It makes very conservative assumptions about when those variables can be modified, rendering their analysis relatively ineffective. If the behavior of a function contextually depends on a global variable's value, it is best to conservatively model that function. For example, if you're modeling a deallocation function, then make that function always deallocate the supplied pointer regardless of the global variable's value. This eliminates the numerous false positives that function may produce. While it will also eliminate bugs due to incorrect usage of that function, the tradeoff between bugs and false positives favors the conservative solution.

Note

To avoid unexpected results, do not move derived model files from one platform to another.

1.4.6.3. Using Coverity Analysis configuration files in the analysis

As discussed in Section 1.4.1, “The configuration”, the `cov-configure` command generates a compiler configuration file. Though not typically recommended without the help of Coverity support (`software-integrity-support@synopsys.com`), you can modify the file in following ways to support your analyses:

Using the `<include>` tag set to include additional configuration files

- You might do so to partition your configuration by organization, project, individual, or other classification. For information about this tag set, see Section 1.4.1, “The configuration”.

Specifying command line options

- You can specify Coverity Analysis command line options explicitly on the command line or through one or more XML-based configuration files. See Section 1.4.6.3.2, “Specifying command line options in the configuration file”. Coverity Analysis searches for `coverity_config.xml` in all configuration directories that are specified in the `<include>` tags. See Section 1.4.6.3.2.2, “Using the prevent tag to specify directories and emit options”.

Specifying Coverity Analysis directories and emit options

- You can specify the temporary and intermediate directory and emit options (for example, to `cov-emit` and `cov-emit-java`)) within the `<prevent>` tag set. For details, see Section 1.4.6.3.2.2, “Using the prevent tag to specify directories and emit options”.

Specifying options used to commit analysis results to Coverity Connect

- You can specify options to `cov-commit-defects` within the `<commit>` tag set. The tags go in the master configuration file. For details, see Section 1.4.6.3.2.1, “Using the <cim> tag to specify commit options”.

Changing the name and/or location of `coverity_config.xml`

- By default, Coverity Analysis creates `coverity_config.xml` file in the following location: `<install_dir>/config`. If you need to change the file name or location, see Section 1.4.6.3.1, “Using alternative configuration file names and directories”.

Note that if you modify configuration files in ways that violate the DTD description (found in `coverity_config.dtd`), most Coverity Analysis commands will issue a warning.

Also note that **COVLKTMPDIR** and environment variable names starting with **COV_** or **COVERITY** are reserved. Users should not set them unless directed to do so by Coverity support staff.

1.4.6.3.1. Using alternative configuration file names and directories

If you install Coverity Analysis in a read-only location, Coverity Analysis will not be able to use the `<install_dir>/config` directory. To specify an alternative configuration file directory (or configuration file name), you can use the `--config` option to the `cov-configure` command. For example:

```
> cov-configure --config /var/coverity/config/coverity_config.xml \
   --comptype gcc --compiler gcc
```

You need to be able to create sub-directories relative to the directory that contains `coverity_config.xml`.

To use an alternative name for the configuration file (for example, `coverity_foobar.xml`) and then use that file name for each step in the analysis, you need to complete *one* of the following tasks:

- Use the `--config` option when running Coverity Analysis commands.

- If recommended by Coverity support, set the `COVERITY_CONFIG` environment variable to point to the directory that contains the configuration file.

- Move `coverity_config.xml` and any other directories generated by the configuration to `~/.coverity`. (Note that this is a local configuration that applies only to you.)

Note

If you need to move the configuration after it is generated, you must move the entire configuration file directory and all of its sub-directories.

1.4.6.3.2. Specifying command line options in the configuration file

Instead of passing command line options to Coverity Analysis on the command line, it is possible to pass command line options to Coverity Analysis through a Coverity Analysis configuration file. However, note that the options that you specify on the command line take precedence over those that you specify in a configuration file.

Coverity Analysis commands point to a Coverity Analysis configuration file by using the following command line option:

```
--config <path/to/XML/config/file>
```

A number of Coverity Analysis commands (including `cov-configure`, `cov-build`, `cov-analyze`, `cov-commit`, and `cov-make-library`) support this option. For others, check *Coverity 2020.12 Command Reference* .

To construct the option tags in the configuration file, refer to the following general rules:

1.4.6.3.2.1. Using the <cim> tag to specify commit options

You use the `cov-commit-defects` command to send analysis results to Coverity Connect. You can use its `--config` option to pass many Coverity Connect-specific options that are specified in the master configuration file (typically, `coverity_config.xml`). Note, however, that if the same option is specified both on the command line and in an XML configuration file, the command line takes precedence.

Note

For an example of the master configuration file, see Section 1.4.1, “The configuration”.

The following <cim> tags are available. The <cim> tag is nested under the <coverity> and <config> elements (see example [p. 26]).

Table 1.4.2. Options under the <cim> tag in coverity_config.xml

Tag	Description	Equivalent `cov-commit-defects` option
`<certs/>`	Set of CA certificates specified in the given `filename` that are in addition to CA certificates obtained from other trust stores. A child of the `<client_security>` tag.	`--certs <filename>`
`<host/>`	Name of the Coverity Connect server host to which you are sending the results of the analysis. Used along with the `<port>` tag. A child of the `<cim>` tag.	`--host <server_name>`
`<password/>`	The password for the user specified with the `<user>` tag. A child of the `<client_security>` tag.	`--password <password>`

Tag	Description	Equivalent `cov-commit-defects` option
	If you put your password into this file, consider taking precautions to set the file permissions so that it is readable only by you.	
`<port/>`	The HTTP port on the Coverity Connect host. Used along with the <host> tag. A child of the `<cim>` tag. • `<cim>/<port>` is equivalent to `--port` • `<cim>/<commit>/<port>` is equivalent to `--dataport`	`--port <port_number>`
`<user/>`	The user name that is shown in Coverity Connect as having committed the analysis results (in a Coverity Connect snapshot). A child of the `<client_security>` tag.	`--user <user_name>`
`<source-stream/>`	The Coverity Connect stream name into which you intend to commit these defects. A child of the `<commit>` tag. The stream must exist in Coverity Connect before you can commit data to it.	`--stream <stream_name>`
`<ssl/>`	Indicator that SSL is to be used for both HTTPS port and dataport connections. A child of the `<client_security>` tag.	`--ssl`

The following example shows how to use the tags described in Table 1.4.2, “Options under the cim tag in coverity_config.xml”:

```
<?xml version="1.0" encoding="UTF-8"?>
<!DOCTYPE coverity SYSTEM "coverity_config.dtd">
<coverity>
  <cit_version>1</cit_version>
  <config>
    <cim>
      <host>cim.company.com</host>
      <port>8443</port>
      <client_security>
         <user>admin</user>
         <password>1256</password>
         <ssl>yes</ssl>
         <certs>/path/to/.certs</certs>
      </client_security>
      <commit>
        <source-stream>myStream</source-stream>
      </commit>
    </cim>
  </config>
</coverity>
```

1.4.6.3.2.2. Using the `<prevent>` tag to specify directories and emit options

You can use the `<coverity><config><prevent>` tag to specify directories and certain build options in the configuration file instead of specifying them on the command line. Options that are specific to the build process reside under the `<emit_options>` tag. Table 1.4.3, “Options under the prevent tag in coverity_config.xml” describes some common configuration options and the equivalent command line option for each. If an option is specified both on the command line to a command and also in `coverity_config.xml`, the command line takes precedence.

Table 1.4.3. Options under the `<prevent>` tag in coverity_config.xml

Tag	Description	Overriding command line option
`<tmp_dir>`	The directory in which to place temporary files.	tmpdir `<dir>` -t `<dir>`
`<dir>`	The top-level directory that Coverity Analysis uses to determine the emit and output directories.	dir `<dir>`

The tags described in Table 1.4.3, “Options under the prevent tag in coverity_config.xml” require a matching closing tag.

Note

For `cov-analyze`, only the following XML tags are supported:

- `<tmp_dir>`
- `<dir>`

Note

It is important to keep tags and their values on the same line. For example:

```
<tmp_dir>/home/user/tmp/data</tmp_dir>
```

Using line breaks (as shown in the following example) can create pathnames with unintended characters (such as carriage returns) or cause other problems.

```
<tmp_dir>/home/user/tmp/
data</tmp_dir>
```

1.4.6.4. Changing a configuration for a compiler

If you have already configured a particular compiler, you cannot create a new configuration for that compiler by re-invoking `cov-configure`. When you invoke `cov-configure`, Coverity Analysis simply inserts the `<include>` directive that references a new compiler configuration file below any other `<include>` directives that are already in the file. When you invoke `cov-build`, Coverity Analysis uses the first configuration it finds that matches the compiler you specify. So the existing configuration (which precedes the new configuration) always takes precedence over the new configuration of the same compiler.

The following example shows a master configuration file. The file includes other `coverity_config.xml` files that are configured for the compilers that belong to the gcc and g++ compiler types:

```
<?xml version="1.0" encoding="UTF-8"?>
<!DOCTYPE coverity SYSTEM "coverity_config.dtd">
<coverity>
<!-- Coverity Version Information -->
<cit_version>1</cit_version>
<config>
<include>$CONFIGDIR$/template-gcc-config-0/coverity_config.xml</include>
<include>$CONFIGDIR$/template-g++-config-0/coverity_config.xml</include>
</config>
</coverity>
```

In the example, the configuration file references compiler-specific configuration files through relative paths of the following form: `$CONFIGDIR$/<comptype>-config-<number>/coverity_config.xml`, where `$CONFIGDIR$` is expanded to the absolute path name of the directory that contains the top-level configuration file, `<comptype>` is the compiler type specified by `cov-configure`, and `<number>` is a numerical designation used to separate multiple compilers of the same type.

If you need to change an existing compiler configuration (for example, because the current one does not work), you can delete it. For example, if you ran `cov-configure --compiler cl --comptype gcc` and wanted to remove the erroneous `cl as GCC` configuration, you could run one of the following to remove those configurations:

```
cov-configure --delete-compiler-config gcc-config-0
```

```
cov-configure --delete-compiler-config g++-config-0
```

Once the configuration works correctly in a local directory, you can run `cov-configure` once more without `--config` to create the configuration. Be sure to save the exact `cov-configure` command that worked and any additional customization, just as you would save any essential source code.

Sample commands using a test configuration file:

```
> cov-configure --config cfg/test-cfg.xml --gcc
> cov-build --config cfg/test-cfg.xml --dir intdir gcc hello.c
```

Part 2. Analyzing source code from the command line

Coverity Analysis can analyze applications that are written in a number of programming languages (see Chapter 1.3, *Language Support*).

After installing and deploying Coverity Analysis, you can run an analysis and commit a snapshot of the resulting defects to Coverity Connect. This section covers post-installation procedures. For more information about deployment and analysis options for a production environment, see Part 1, “Overview”.

Chapter 2.1. [1]Getting started with Coverity analyses

This section provides common steps (configure, build, analyze, commit) for running analyses from the command line. Alternatively, you can run analyses with the GUI-based Coverity Wizard application (see *Coverity Wizard 2020.12 User Guide* for details), which uses commands described in this section.

Note

For information about compatibility between different versions of Coverity Connect and Coverity Analysis, see the section, "Compatibility between Coverity product components" in the *Coverity 2020.12 Installation and Deployment Guide*.

For more information about the steps presented here, see Chapter 1.4, *Coverity analyses*. For analysis tasks that are outside the scope of this section, see Chapter 2.6, *Using advanced analysis techniques*.

To get started from the command line:

1. **Complete the prerequisites to this procedure.**

   See Prerequisites to completing an analysis.

2. **[Onetime step [p. 33]] Generate a configuration for your compiler or your scripting language.**

   Note

   This step is not required if you are using buildless capture.

   Run the `cov-configure` command with the appropriate option.

   Note that you need to run the command separately for each compiler or scripting language that applies.

   Table 2.1.1. Commonly used options to cov-configure[a]

Language	Configuration Option
C/C++ (for GNU GCC and G++)[b]	`--gcc`
CUDA	`--cuda` (for all CUDA compilers)
Go	`--go`
Java build capture (for `java`, `javac`, `javaw`, `apt`) and JSPs	`--java`
Java filesystem capture and JSPs	`--javafs`
JavaScript	`--javascript`
Kotlin	`--kotlin`
Microsoft C and C++ (for `cl.exe`)	`--msvc`

```
For Java analyses of Android applications, see
```

Language	Configuration Option
Microsoft C# (for `csc.exe`)[c]	`--cs`
C, C++, Objective-C, and Objective-C++ (for `clang` and `clang++`)[d]	`--clang`
PHP	`--php`
Python	`--python`
Ruby	`--ruby`
Scala	`--scala`
Swift	`--swift` `--swiftc`[e]

[a] To create configuration for compilers that are not listed here and to understand configuration for a production environment, see Chapter 2.7, *Configuring compilers for Coverity Analysis*.

[b] To perform a configuration for `gcc` and `g++`, you can use the `--gcc` command option.

The console output for a successful configuration looks something like the following:

```
Generated coverity_config.xml at location
/my_install_dir/config/coverity_config.xml
Successfully generated configuration for the compilers: g++ gcc
```

[c] The console output for a successful configuration for C# looks something like the following:

```
Generated coverity_config.xml at location
/my_install_dir/config/coverity_config.xml
Successfully generated configuration for the compilers: csc
```

[d] See also, Section 2.7.3.19, “Clang compilers”

[e] When using `xcodebuild`, the `-UseModernBuildSystem=NO` option must be added to use Swiftc Driver. For additional details, refer to the section "Building projects that use Xcode 10's new build system".

Note

On some Windows platforms, you might need to use Windows administrative privileges when you run `cov-configure`.

Typically, you can set the administrative permission through an option in the right-click menu of the executable for the command interpreter (for example, `Cmd.exe` or Cygwin) or Windows Explorer.

Important!

- You must run `cov-configure` in exactly the same environment that you run your native compiler. If you run the command without configuring the operating environment exactly as it is in your native build environment, the configuration will be inaccurate. (Note that for C/C++ compilers, this command invokes the native compiler to determine its built-in macro definitions and the system include directories.)

- Note that you typically run `cov-configure` only once per installation (or upgrade) and compiler type or scripting language because the configuration process stores the configuration values for the compiler in `coverity_config.xml` file. However, if you need to perform a reconfiguration (for example, because the native compiler, build environment, or hardware changes), see Section 1.4.6.4, “Changing a configuration for a compiler”.

3. **Capture your source code into the intermediate directory.**

   From the source code directory, run the `cov-build` command:

   - **For compiled languages (build capture), including Java build capture:**

     ```
     > cov-build --dir <intermediate_directory> <BUILD_COMMAND>
     ```

   - **For scripts and interpreted code (filesystem capture), and Java filesystem capture:**

     ```
     > cov-build --dir <intermediate_directory> --no-command \
       --fs-capture-search <path/to/source/code>
     ```

     If you are only performing a filesystem capture and not also performing a build capture, you need to pass the `--no-command` command.

   - **For JavaScript filesystem capture:** In addition to the guidelines for scripts and interpreted code, Coverity also recommends that you create a list of files to capture, including the JavaScript libraries (preferably non-minified versions) that you use, and pass that list to `cov-build`.

     If you have both minified and non-minified versions of the same library in your source, you should capture only the non-minified version. However, if only the minified version is available, you should capture it. You can use `--fs-capture` options for this purpose. Although defects will be reported in third party code when it is not minified, and those defects might not be actionable, supporting evidence for defects in your code can only be shown in non-minified code. See also, Deployment recommendations.

     This example creates a list of files in a git repository, then passes it through `--fs-capture-list`:

     ```
     > git ls-files > scm_files.lst
     > cov-build --dir <intermediate_directory> --no-command \
     --fs-capture-list scm_files.lst
     ```

   - **For a combined source code capture (build and filesystem capture):**

     ```
     > cov-build --dir <intermediate_directory> \
       --fs-capture-search <path/to/source/code> <BUILD_COMMAND>
     ```

     Builds of Android applications that are written in Java require the combined capture command if the Java build capture is used.

     Here is an example that uses `--fs-capture-list`:

     ```
     > git ls-files > scm_files.lst
     ```

```
> cov-build --dir <intermediate_directory> --no-command \
  --fs-capture-list scm_files.lst <BUILD_COMMAND>
```

Note that the build command must be specified last.

- **For buildless capture of C#, Java, JavaScript, TypeScript, PHP, Python, and Ruby:**

  This example captures all C#, Java, or JavaScript/TypeScript projects under a directory

  ```
  > cov-capture --project-dir /path/to/my/projects
  > cov-capture --project-dir C:\path\to\my\projects
  ```

  This example captures all C#, Java, or JavaScript/TypeScript projects in a git repository:

  ```

  > cov-capture --scm-url git@mygit.internal.company.com:myrepo.git
  ```

The examples use the following notation for values to command options:

- <BUILD_COMMAND> is the command you use to invoke your compiler (for example, `gcc` or `javac`) on source code, such as Java and C++. For important recommendations, see the build notes.

- `<intermediate_directory>` specifies a name that you designate for the directory that will store an intermediate representation of the code base that you are building or emitting. The command will create an intermediate directory with the name you specify if it does not exist already. This directory can contain data for all the supported programming languages.

Note

A single invocation of `cov-build` can perform filesystem capture and build capture for all applicable languages, including interpreted languages, such as JavaScript, PHP, Python, and Ruby.

Table 2.1.2. Build and filesystem capture notes

Topic	Notes
Java code capture (including for Android and Web application security analyses)	• For Java, if the Java build capture is used, the build command should compile all class files. If you plan to analyze a Java EE, servlet-based application, the build command should also package the Web Archive (WAR, `.war`) file (alternatively, you can simply specify a directory with the unpacked contents of the WAR file when you reach the next step in this procedure). • For Java, if the filesystem capture is used, JSPs are (by default) also captured. Java filesystem capture also enables the filesystem capture of Java Android files that are needed by the analysis, including the manifest (`AndroidManifest.xml`) and layout resource files. • For Java analyses of Android applications, see Section 2.3.1, “Running a security analysis on an Android mobile application”.

Topic	Notes
	• Example that uses Ant: `> cov-build --dir /foo/xalan_j_2_7_0_analysis ant` This UNIX-based example assumes you have previously changed the directory to a Xalan build directory that uses the standard Ant `build.xml` file. • Using the `cov-build` command with Java requires a supported Sun/Oracle JDK. For information about supported JDKs, see the *Coverity 2020.12 Installation and Deployment Guide*. • If you cannot use `cov-build`, see the procedure described in Section 2.6.3.1, “Running an analysis without cov-build”.
ASP.NET Web application capture	See Build capture (cov-build) notes for ASP.NET 4 and earlier.
ASP.NET Core Web application capture	See Build capture (cov-build) notes for ASP.NET Core 2.0 and later.
Interpreted code base capture (including JavaScript, Python, PHP, and Ruby files)	For interpreted languages, see "Filesystem capture for interpreted languages" in the `cov-build` documentation.
Non-ASCII code capture	For C and C++ builds, if you are building non-ASCII code, you need to add the `--encoding <character_encoding>` option to the `cov-build` command.
Builds on the IBM AIX operating system	AIX installations do not include the `cov-build` or commands. To complete the AIX build and analysis tasks, see Section 3.3.2.8.5, “AIX”.
Cygwin	If you intend to use Cygwin, see Section A.2, “Using Cygwin to invoke cov-build”.

Example: Building sample C code

- To analyze the sample C file (`<install_dir>/doc/examples/test.c`), type the following command from the `<install_dir>/doc/examples` directory:

  ```
  > cov-build --dir analysis_dir make test
  ```

  For a successful build, the command-line output should look like the following:

  ```
  gcc -c test.c -o test.out
  1 C/C++ compilation units (100%) are ready for analysis
  ```

```
The cov-build utility completed successfully.
```

The sample makefile uses the gcc compiler and works on UNIX-like systems or on Windows with Cygwin. The intermediate data is stored in the `<install_dir>/doc/examples/analysis_dir` directory.

Build log file: The log file that contains the `cov-build` command results is `<intermediate_directory>/build-log.txt`.

4. **For Java Web application security analyses with build capture only, emit all Web application archives or directories.**

Before analyzing a Java EE servlet-based Web application emitted using build capture, it is first necessary to emit any non-source files, such as JavaServer Pages (JSPs) or those in a packaged Web application archive. This is in addition to any compiled Java source that might have been captured already. Additional defects can be reported based on these files.

Note

Unlike build capture, Java filesystem capture enables the capture of JSPs (by default), therefore it would be redundant to manually emit any JSPs or archives.

Packaged and deployable Java Web applications might take the form of WAR or EAR archive files, directories containing a `WEB-INF/web.xml` file (equivalent to an unpacked WAR file), or directories containing a `META-INF/application.xml` file (equivalent to an unpacked EAR file).

If the build does not generate any WAR or EAR file but the project does contain JavaServer Pages (JSPs), these can also be emitted using filesystem capture. See "Filesystem capture for interpreted languages" in the `cov-build` documentation. If the JSPs are emitted as part of a Web application archive, it is not necessary (and is redundant) to emit them using filesystem capture.

a. [Recommended] Prior to emitting the JSP files, you can pre-compile JSP files to ensure that the JSP files will compile and that the classpath is specified appropriately.

b. If the previous step is successful, emit the JSPs in preparation for the analysis.

To capture a single Web application archive, use the following command:

```
> cov-emit-java --dir <intermediate_directory> \
  --webapp-archive path/to/archive_file_or_dir
```

For details about this command line, see the `--webapp-archive` option documentation.

Important!

You need to emit the JSP files so that the analysis can find and report issues it finds in them. If these files are not present in the WAR file, false negatives will occur, particularly in XSS defect reports.

It is also important to emit the original JSP source, even in cases where the build normally pre-compiles the JSP files into classes and packages those into the WAR file.

The Web application archive or directory should not contain obfuscated classes.

To emit multiple WAR or EAR files, you can run `cov-emit-java` multiple times, use multiple instances of the `--webapp-archive` command option, or use one of the following command options: `--findwars`, `--findwars-unpacked`, `--findears`, or `--findears-unpacked`.

If you run into issues at this stage, see the JSP-related troubleshooting information in Section 2.2.1, “Running a security analysis on a Java Web application”.

5. **Analyze the source code.**

- Run the `cov-analyze` command to analyze the intermediate data:

```
> cov-analyze --dir <intermediate_directory> --strip-path <path/to/source/code>
```

- For Web application analyses, pass Web application options:

```
> cov-analyze --dir <intermediate_directory> --strip-path <path/to/source/code>
 \
  --webapp-security
```

For troubleshooting information and important details about Web application security analyses for Java and ASP.NET, see Chapter 2.2, *Running Web application security analyses*.

- For Android applications that are written in Java, use the `--android-security` option. See Section 2.3.1, “Running a security analysis on an Android mobile application”.

- If you intend to run JSHint analysis, see `--enable-jshint` in the *Coverity 2020.12 Command Reference*. Otherwise, the JavaScript analysis does not require additional command options.

- If you intend to run a MISRA analysis, see Chapter 2.4, *Running coding standard analyses*.

By default, the `cov-analyze` command analyzes all code in the specified intermediate directory through a single invocation of the `cov-analyze` command. The command runs a series of default checkers. You can add or remove checkers from the analysis. For information about which checkers are enabled by default and how to enable additional checkers, see *Enabling and Disabling Checkers* in the *Coverity 2020.12 Checker Reference*.

Recommended

Coverity highly recommends using the `--strip-path` option with `cov-analyze` to specify the root directory of the source code tree. This shortens paths that Coverity Connect displays. It also allows your deployment to be more portable if you need to move it to a new machine in the future.

Using this option with `cov-analyze` (instead of `cov-commit-defects`, when you commit the analysis results to Coverity Connect) can enhance end-to-end performance of the path stripping process.

Example: Analyzing a sample build

- To analyze the sample intermediate data, enter the following command:

  ```
  > cov-analyze --dir <install_dir>/doc/examples/analysis_dir --strip-path <path>
  ```

  The `cov-analyze` command puts analysis results into the intermediate directory.

  If you get a fatal `No license found` error when you attempt to run this command, you need to make sure that `license.dat` was copied correctly to `<install_dir>/bin`.

  To correct this issue, see .Setting up Coverity Analysis licensing.

Understanding the Analysis Summary Report

- The output of the `cov-analyze` command is an analysis summary report. Depending on your platform, the following output might differ:

  ```
  [Looking for translation units
  |0----------25-----------50----------75---------100|
  ****************************************************
  [STATUS] Computing links for 1 translation unit
  |0----------25-----------50----------75---------100|
  ****************************************************
  [STATUS] Computing virtual overrides
  |0----------25-----------50----------75---------100|
  ****************************************************
  [STATUS] Computing callgraph
  |0----------25-----------50----------75---------100|
  ****************************************************
  [STATUS] Topologically sorting 12 functions
  |0----------25-----------50----------75---------100|
  ****************************************************
  [STATUS] Computing node costs
  |0----------25-----------50----------75---------100|
  ****************************************************
  [STATUS] Starting analysis run
  |0----------25-----------50----------75---------100|
  ****************************************************
  2013-10-11 21:29:02 UTC - Calculating 34 cross-reference bundles...
  |0----------25-----------50----------75---------100|
  ****************************************************
  Analysis summary report:
  ------------------------
  Files analyzed                 : 1
  Total LoC input to cov-analyze : 8584
  Functions analyzed             : 12
  ```

```
Paths analyzed                 : 49
Time taken by analysis         : 00:00:02
Defect occurrences found       : 18 Total
                                  1 DEADCODE
                                  1 FORWARD_NULL
                                  1 NEGATIVE_RETURNS
                                  1 OVERRUN
                                  7 RESOURCE_LEAK
                                  1 REVERSE_INULL
                                  1 REVERSE_NEGATIVE
                                  1 SIZECHECK
                                  1 SIZEOF_MISMATCH
                                  1 UNINIT
                                  1 UNUSED_VALUE
                                  1 USE_AFTER_FREE
```

The analysis summary report contains the following information:

- Files analyzed: The total number of files having classes/structs or functions requiring analysis on this run of `cov-analyze`. Files that do not contain classes/structs or functions (such as a header file with only macro definitions) are not reflected in this total.
- Total LoC input to cov-analyze: The total number of lines of code analyzed.
- Functions analyzed: The total number of functions actually requiring analysis or re-analysis. If the count is 0, this output field is not displayed.
- Paths analyzed: The sum of paths traversed for all analyzed functions. (Note that there is some complexity to the calculation that is used to produce this sum.)
- Time taken by analysis: The amount of time taken (in hours, minutes, and seconds) for the analysis to complete.
- Defect occurrences found: The number of defect occurrences found by the analysis, followed by a breakdown by checker. When the snapshot is committed to Coverity Connect, it merges similar defects from a given stream into a single CID, so the number of CIDs will likely differ from the number shown in the analysis summary.

A log file with information about the analysis is created in `<intermediate_directory>/output/analysis-log.txt`

6. **Commit the analysis results to a Coverity Connect stream.**

   Run `cov-commit-defects` to add the defect reports to the Coverity Connect database.

```
> cov-commit-defects  --host <server_hostname> \
    --dataport <port_number> \
    --stream <stream_name> \
    --user admin --dir <intermediate_directory>
```

   Example:

```
> cov-commit-defects --host coverity_server1 \
    --dataport 9090 --stream apache --dir apache_int_dir \
    --user admin --password 1256
```

A successful commit looks something like the following:

```
Connecting to server 10.9.9.99:9090
2013-10-11 23:47:49 UTC - Committing 34 file descriptions...
|0----------25-----------50----------75---------100|
****************************************************
2013-10-11 23:47:49 UTC - Committing 34 source files...
|0----------25-----------50----------75---------100|
****************************************************
2013-10-11 23:47:49 UTC - Calculating 34 cross-reference bundles...
|0----------25-----------50----------75---------100|
****************************************************
2013-10-11 23:47:50 UTC - Committing 34 cross-reference bundles...
|0----------25-----------50----------75---------100|
****************************************************
2013-10-11 23:47:51 UTC - Committing 12 functions...
|0----------25-----------50----------75---------100|
****************************************************
2013-10-11 23:47:51 UTC - Committing 12 defect files...
|0----------25-----------50----------75---------100|
****************************************************
2013-10-11 23:47:53 UTC - Committing 3 output files...
|0----------25-----------50----------75---------100|
****************************************************
New snapshot ID 10001 added.
Elapsed time: 00:00:05
```

- `--dataport` and `--port`: When you run `cov-commit-defects`, you have a choice of specifying `--dataport` (the default is `9090`) or `--port` (the default is `8080`). The Coverity Connect installer refers to the dataport as the commit port.

  Note

  If you are committing to a TLS/SSL-enabled instance of Coverity Connect, use the `--https-port` option instead of `--port`. For more information, see *Coverity 2020.12 Command Reference*.

- `--host`: On a Linux OS, you must enter the full host and domain name or IP address for the `--host` option, for example:

  ```
  --host <server_hostname.domain.com>
  ```

- `--stream`: The <stream_name> value specifies an existing Coverity Connect stream (see analysis prerequisites).

- `<intermediate_directory>` is the directory that contains the defect reports.

7. **View the resulting issues in Coverity Connect.**

   For an example, see "Managing Issues." ↗

Chapter 2.2. Running Web application security analyses

Table of Contents

Coverity Analysis can find security vulnerabilities such as cross-site scripting (XSS) and SQL injection in a wide variety of web applications. To enable web application security analysis, pass the `--webapp-security` option to `cov-analyze`. See Software Issues and Impacts by Security Checker ↗ in the *Coverity 2020.12 Checker Reference* for details on which kinds of vulnerabilities Coverity can find for each programming language. For memory requirements and other prerequisites to the analysis, see Prerequisites to completing an analysis.

This section adds recommendations and troubleshooting information to supplement the basic analysis steps in Chapter 2.1, *Getting started with Coverity analyses* for some kinds of web applications. Languages or environments that are not mentioned in this section do not require additional steps.

2.2.1. Running a security analysis on a Java Web application

You can perform a security analysis on Java Enterprise Edition (Java EE), servlet-based Web applications. The workflow for the Java Web application security analysis mostly follows the typical pattern. The main differences (detailed in Chapter 2.1, *Getting started with Coverity analyses*) follow:

- If you use Java build capture, you should use `cov-emit-java` to make the contents of any WAR or EAR files available for analysis. If the project contains JSP files that are not packaged into a WAR or EAR, filesystem capture can be used to emit the JSPs. Note that it is necessary to emit JSP files so that Coverity Analysis can analyze them and report issues on them.

- Unlike Java build capture, both Java filesystem and buildless capture enable the capture of JSPs by default, therefore manually emitting JSPs or archives would be redundant.

Troubleshooting failures to compile or emit JSP files

- Failure can occur because the dependencies are not available on the classpath. This issue can be resolved by finding the dependencies and adding them to the classpath.

- Failure can occur because the JSP file is not valid and would never compile in any reasonable application server. Such JSP files should be fixed, removed, or ignored depending on whether they are needed.

  Note that you can pre-compile JSP files as part of the build step to ensure that the JSP files will compile and that the classpath is specified appropriately.

- Failure to emit can occur because the JSP file is a fragment and is only meant to be included as part of another JSP. Coverity Analysis attempts to identify such JSPs and log the files it is unable to emit.

- If you encounter another sort of issue related to compiling or emitting JSP files, report it to `software-integrity-support@synopsys.com`.

Troubleshooting Web application security analyses for Java: Using COVERITY_DA_BLACKLIST to prevent certain fatal JRE errors

> When Web application security checkers analyze Java code, `cov-analyze` runs a sanitizer fuzzer to execute string manipulation code in your application. The sanitizer fuzzer runs in a JRE. If the JRE crashes at the analysis step called `Running dynamic analysis for Java Webapp Security`, followed by messages such as `A fatal error has been detected by the Java Runtime Environment` or `[WARNING] Failure in security DA`, you can set the COVERITY_DA_BLACKLIST environment variable to prevent the Coverity analysis from executing the string manipulation code in your application that caused the problem.
>
> The value of this variable should be a comma-separated list of prefixes, which you set to prevent the direct loading of classes that start with any of those prefixes. For example, you might set the following:

```
COVERITY_DA_BLACKLIST=com.acme.util,com.acme.text.util
```

2.2.2. Running a security analysis on an ASP.NET Web application

The security analysis of ASP.NET Web applications is capable of reporting defects in Razor view templates (such as `*.cshtml` files) and WebForms (such as `*.aspx` files). In most cases, these files are captured automatically by `cov-build`. If Web application template files are captured, the following message will be displayed in your console output and in `build-log.txt` (see “Build log file”):

```
140 compiled C# template files captured:
   42 ascx files
   33 aspx files
   65 cshtml files
```

In cases where `cov-build` fails to capture the expected number of template files, please consult the following sections.

2.2.2.1. Capturing an ASP.NET Core (2.0 or later) Web application

To include Web application template files in the analysis, `cov-build` must capture a compilation of your Web applications that has View Precompilation enabled. Typically, View Precompilation is enabled by default. However, if `cov-build` is failing to capture Razor template files, take care to ensure that View Precompilation has not been disabled. This is most commonly controlled by the following MSBuild properties:

- `RazorCompileOnBuild`
- `RazorCompileOnPublish`
- `MvcRazorCompileOnBuild`

- `MvcRazorCompileOnPublish`

2.2.2.2. Capturing an ASP.NET (4.0 or earlier) Web application

By default, `cov-build` attempts to find and emit any Web application template file (Razor and WebForms) contained in the same folder or sub-folder of every project file (`*.csproj` or `*.vbproj` file) that is being compiled. To properly emit template files, `cov-build` also relies on `web.config` files being present in the directory that contains the project file. If the Web application being compiled is not structured this way, the automatic capture of the template files might not capture all of the Web application's template files.

To manually include these files in the analysis, you should first disable the automatic capture of Web application template files by passing the `--disable-aspnetcompiler` option to `cov-build`. Then, `cov-build` must capture an invocation of `Aspnet_compiler.exe` on the published Web application. Publishing the Web application and running `Aspnet_compiler.exe` might not be part of your source build process. Following is an example command line for running `Aspnet_compiler.exe`:

```
Aspnet_compiler.exe -p C:\path\to\MyWebApplicationRoot -v root -d -f -c C:\path\to
\TargetDir
```

The physical path, specified with the `-p` option, should point to the Web-application root path—that is, the directory the Web application was published to.

The virtual path, specified with the `-v` option, is required but does not affect the analysis.

The final command-line option, `-c`, names the output directory for the compiler outputs.

Note

Buildless capture (`cov-capture`) *does not support* ASP.NET Web applications. Buildless capture only supports ASP.NET Core Web applications. Buildless capture only guarantees support for ASP.NET Core Web applications built with .NET Core SDK version 2.0 through 2.2, though it might also work with newer versions of the .NET Core SDK.

Chapter 2.3. Running mobile application security analyses

Table of Contents

2.3.1. Running a security analysis on an Android mobile application

You can perform a security analysis on Android applications that are written in Java and Kotlin. This analysis runs the Android application security checkers. For details, see Security Checkers ↗ in the *Coverity 2020.12 Checker Reference*.

The workflow for the Android application security analysis mostly follows the typical pattern (see Chapter 2.1, *Getting started with Coverity analyses* for that pattern). The main differences are the following:

2.3.1.1. For Java Android applications

- Run the `cov-configure --java` command to enable the build capture of Java Android source and the filesystem capture of Android configuration files. (Note that Java filesystem capture can be used as a fallback.)

- You need to pass `--fs-capture-search` to the `cov-build` command. Coverity captures Java Android files that are needed by the analysis, including the manifest (`AndroidManifest.xml`) and the layout resource files.

- You need to pass `--android-security` to the `cov-analyze` command.

2.3.1.2. For Kotlin Android applications

- Run the `cov-configure --kotlin` command to enable the build capture of Kotlin Android source and the filesystem capture of Android configuration files.

- You need to pass `--fs-capture-search` to the `cov-build` command. Coverity captures Kotlin Android files that are needed by the analysis, including the manifest (`AndroidManifest.xml`) and the layout resource files.

- No special flags are needed for the cov-analyze command, because Kotlin checkers are enabled by default.

2.3.1.3. For hybrid (Java and Kotlin) Android applications

- Run the `cov-configure --java` command and the `cov-configure --kotlin` command to enable the build capture of both Java and Kotlin Android source and the filesystem capture of Android configuration files. (Note that filesystem capture can be used as a fallback only for Java.)

- You need to pass `--fs-capture-search` to the `cov-build` command. Coverity captures Java and Kotlin Android files that are needed by the analysis, including the manifest (`AndroidManifest.xml`) and the layout resource files.
- You need to pass `--android-security` to the `cov-analyze` command.

Note

Java and Kotlin source code are analyzed separately.

2.3.2. Running a security analysis on an iOS mobile application (written in Swift)

Coverity can perform a security analysis of iOS applications that are written in Swift.

Use the `cov-configure --swift` command to enable the build capture of Swift source code, as well as the filesystem capture of important configuration files. Filesystem capture is strongly recommended when emitting any important configuration files (such as property list files and project files). For more information about Swift build capture, see the `cov-build` command.

The `cov-analyze` command does not require any additional command line options to enable Swift security analyses. The Swift iOS security checkers are enabled by default. For details on the checkers and defects that are found, see the Swift checkers in the *Coverity 2020.12 Checker Reference*

The security analysis workflow follows the typical Coverity analyses workflow. (See Chapter 2.1, *Getting started with Coverity analyses* .) .

Chapter 2.4. Running coding standard analyses

Table of Contents

Coverity Analysis enables you to run code analyses using the supported coding standards listed in Table 1.3.1, “Support by Language”.

You can run multiple coding standards analyses in a single run.

HIS metric analysis is available when running any of the supported MISRA C or C++ coding standards. However, it is not possible to specify multiple HIS configurations in a single analysis run. If multiple configurations are specified, the last one on the command line is used.

For each analysis run, only one configuration can be specified per coding standard. You can use the same intermediate directory for coding standards analyses as you do for regular builds, commits, and other types of analysis.

The coding standard analysis works on files with the following file extensions:

```
.C, .c, .cc, .cpp, .cu, .cxx, .h, .hh, .hpp, .hxx, .ipp, .java
```

The coding standard analysis workflow follows the typical pattern described in Chapter 2.1, *Getting started with Coverity analyses* . The main difference is the use of the `--coding-standard-config` option:

To run the coding standard analysis for C/C++, you must pass the `--coding-standard-config` option to `cov-analyze`.

`--coding-standard-config` [Required]

This option is described in the *Coverity 2020.12 Command Reference* .

2.4.1. Coding standard analysis guidelines

To avoid problems, follow these guidelines in running coding standard analyses:

- If you want to run both coding-standard and non-coding-standard analysis on the same source, we recommend that you perform separate analyses that you commit to separate streams. Follow these steps:

  1. Run `cov-build` once.

  2. Run `cov-analyze` and `cov-commit-defects` for the non-coding-standard analysis.

  3. Run `cov-analyze` again with the `--force` option for the coding standard analysis, followed by `cov-commit-defects` to the coding standard stream.

- For coding standard analysis only, use the `--emit-complementary-info` option to the `cov-build` command. Othewise, `cov-analyze` will silently run the build again to capture the additional information it needs.
- Coding standard analysis can report an overwhelming number of issues. We recommend you apply Mandatory rules first, then apply Required rules, and then Advisory rules. These categories and the rules are listed in one of the following directories:
  - `<install_dir>/config/coding-standards/misrac2004`
  - `<install_dir>/config/coding-standards/misrac2012`
  - `<install_dir>/config/coding-standards/misracpp2008`
  - `<install_dir>/config/coding-standards/autosarcpp14`
  - `<install_dir>/config/coding-standards/cert-c`
  - `<install_dir>/config/coding-standards/cert-cpp`
  - `<install_dir>/config/coding-standards/cert-c-recommendation`
  - `<install_dir>/config/coding-standards/cert-java`
  - `<install_dir>/config/coding-standards/iso-ts17961`

Chapter 2.5. Running Fortran Syntax Analysis

Coverity Fortran Syntax Analysis provides Fortran syntax analysis and portability checking. It can emulate the parsing behavior of many current and legacy compilers. In addition, it can verify compliance of the source code with supported Fortran language standards.

Fortran syntax analysis is a standalone tool that performs the file capture and analysis in one step. Source files to be analyzed, include paths and symbol definitions must be listed on the `cov-run-fortran` command line. The analysis expands `INCLUDE` statements and performs limited C preprocessor `cpp` emulation to interpret the sources.

Note:

Fortran Syntax Analysis will attempt to analyze all of the files listed as sources on the command line. Only valid Fortran sources should be listed.

Note:

Fortran Syntax Analysis is sensitive to the compiler emulation and language level chosen. Make sure to select these appropriately. Valid syntax may be flagged as invalid if either the free-form `-ff` option or fixed-form `-nff` option is specified incorrectly.

Once the Fortran Syntax Analysis has completed, the results can be uploaded to Coverity Connect, using the `cov-commit-defects` command.

For more information, see the `cov-run-fortran` command entry in the *Coverity 2020.12 Command Reference*.

Fortran analysis workflow example

- Analyze all files in the current working directory:

```
> cov-run-fortran --dir=idir --vendor=intel --version=14 -- *.f
> cov-commit-defects --dir=idir ...
```

Chapter 2.6. Using advanced analysis techniques

Table of Contents

This section describes general analysis techniques for improving performance.

2.6.1. Incremental, parallel, and desktop analysis

This section describes three analysis modes that you can use to improve performance. For best results, you should be familiar with how these modes work and how they affect Coverity performance:

- **Incremental analysis** speeds up analysis by relying on data stored in the intermediate directory from previous analyses.
- **Parallel analysis** spawns a number of analysis worker processes to carry out the analyses in parallel. The number of workers you can use is related to the number of CPUs and available RAM.
- **Desktop analysis** produces a fast desktop or IDE-based analysis. Results might differ slightly from a full analysis because only changed files are analyzed.

2.6.1.1. Incremental analysis

By default, `cov-analyze` caches build and analysis results in the intermediate directory. Once you create an intermediate directory with `cov-build` and analyze it with `cov-analyze`, you can speed up subsequent build and analysis using *incremental analysis* mode.

To use incremental analysis mode:

1. Pass your analyzed intermediate directory to subsequent `cov-build` runs on the same code base.
2. Run `cov-analyze` on this same intermediate directory with the same command-line options as your previous `cov-analyze` run.

As long as you reuse your intermediate directory in this way, `cov-build` and `cov-analyze` automatically run in incremental mode. In this mode, `cov-build` only re-emits files that have changed since your last `cov-build`. Likewise, `cov-analyze` is able to reuse some of its work on the files and functions that have not changed since the previous `cov-analyze` run. Incremental mode is often faster than a build and analysis using a fresh intermediate directory.

Incremental analysis always produces the same analysis results as building and analyzing the same code from scratch with a fresh intermediate directory.

Note

Not all Coverity checkers speed up in incremental mode. Many Java, C#, and Visual Basic checkers enabled by `--webapp-security` or `--android-security` do not run any faster in incremental mode.

Caution

Be aware that when you run in incremental mode, the build and analysis cannot tell when you've deleted a file or removed it from your build. Rather, it appears to `cov-analyze` as if you haven't recompiled or re-emitted the file and that it should use the version it has cached. Therefore, if you do use incremental mode, it's a good idea to periodically start from scratch with a fresh intermediate directory—particularly if you've moved or deleted files from your source tree or build.

2.6.1.2. Parallel analysis

By default, `cov-analyze` takes advantage of extra CPU cores to speed up analysis. It spawns a number of analysis worker processes according to the number of CPU cores and the amount of physical memory on the machine. Because each worker requires a certain amount of RAM, `cov-analyze` only spawns workers when there is enough RAM to support them.

Important

Prior to running a parallel analysis, make sure that you have the appropriate hardware and enough free memory for each worker that you start. For details, see the Coverity Analysis memory requirements listed in the *Coverity 2020.12 Installation and Deployment Guide*.

There might be times when you need to adjust the number of workers: for example, because `cov-analyze` runs on a shared machine that also runs other jobs. You can use the `cov-analyze` command with the `--jobs` option set to the number of workers that you want to run.

For example, the following command starts six workers:

```
> cov-analyze --dir my_intermediate_dir --jobs 6
```

The following guidelines provide scalability recommendations for different languages and platforms:

- Scalability of a combined C and C++ analysis on Linux (64-bit) and Windows (64-bit) operating systems:
  - Typically, running eight workers yields about a 4x increase in speed over running one worker.
  - Typically, running three workers yields a 2.5x increase in the overall speed of the analysis.
  - Running more than eight workers might not decrease the overall analysis time significantly.
- Scalability of C# analysis on Windows (64-bit) operating systems:
  - Typically, running four workers on C# code yields about a 2.5x increase in speed over running one worker.
  - Typically, running two workers on C# code yields a 1.75x increase in the overall speed of the analysis.
  - Running more than four workers on C# code might not decrease the overall analysis time significantly.

- Scalability of combined C, C++, and C# analysis

  The time for a combined analysis of C, C++, and C# code is close to the time to analyze one after the other with the same settings, but combined analysis usually shows a small advantage when four or more workers are used.

2.6.1.3. Desktop analysis

Desktop analysis is a mode of operation that relies on the results of a previous full analysis. You run desktop analysis using the `cov-run-desktop` command or your IDE rather than using `cov-analyze`; desktop analysis runs very quickly because it only re-analyzes the set of files that have changed since the last central analysis rather than your entire code base.

Unlike incremental analysis, desktop analysis is not guaranteed to produce exactly the same results as a full analysis from scratch. Many checkers report close to the same results. But some checkers are disabled because they only work well with full analysis. Therefore do not assume that a defect that does not show up in desktop analysis was actually fixed.

Fast desktop mode is especially useful for CI/CD deployments, as described in the next section.

2.6.1.4. Running analysis as part of a CI/CD pipeline

Deploying Coverity analysis to a continuous integration / continuous delivery (CI/CD) pipeline involves many trade-offs that must balance analysis speed versus thoroughness of coverage, and which issues you do or do not want to stop your pipeline.

These are complex topics, and they are largely beyond the scope of this document. However, as a starting point, consider a deployment that includes at least the first two of the following types of jobs:

1. **An incremental build and analysis job** that runs continuously in the background, and that uploads scan summaries to Coverity Platform. This setup matches the Desktop use case.

   If this job finds any defects that the following job type (#2) misses, you can deal with those defects without blocking delivery.

2. **A continuous integration job** that runs a `cov-run-desktop` analysis on whatever changed since the last run. This job breaks the CI pipeline if any new defects are reported.

   This job will run quickly because it relies on summaries from the previous job.

3. **An incremental full build and analysis** that doesn’t block your pipeline but that runs a comprehensive set of checkers.

   As with the incremental job, #1, you can deal with the results from job #3 without blocking delivery.

The incremental job, #1, enables the continuous integration job, #2. As a rule, job #2 should run at an early stage of the CI/CD pipeline—for example, during the build phase or the early test phase.

Both job #1 and job #2 should run *exactly* the same analysis configuration: that is, the same set of checkers using the same options. The client might choose this configuration so that analysis runs more quickly, or so that the logic of when a pipeline might break is easy to follow.

For example, the following analysis hones in on code that is liable to be attacked:

```
cov-analyze --disable-default -en SQLI -en XSS
```

Job #3 is optional. A job of this kind does not have to run quickly. Instead, it can run a comprehensive set of checkers in order to detect as many defects as possible. Unlike job #2, the comprehensive job #3 is likely to run at a later point in the pipeline: for example, as a last step before deployment to the staging location.

The following sample invocation, in contrast to the previous one, scans for a wide range of security issues:

```
cov-analyze --webapp-security --distrust-database -en SQL_NOT_CONSTANT
```

2.6.2. Using `cov-analyze` options to tune the analysis

The `cov-analyze` command accepts several options that tune the analysis, affecting both the results and the speed.

Table 2.6.1. Tuning options to `cov-analyze`

Option	Effect on Speed	Effect on Results
`--enable-virtual`	The analysis can take significantly longer for C++ code. This option does not affect C or C# code.	This option enables virtual call resolution to more precisely analyze calls to virtual functions, which can increase the number of defects reported.
`--enable-constraint-fpp`	The analysis can take 10% to 20% longer.	This option uses a false-path pruner (FPP) to perform additional defect filtering. It can increase the analysis time but decrease the number of false positives occurring along infeasible paths. Because this FPP uses an approximate method for pruning false positives, it is possible that a very small number of true positives will also be pruned.

2.6.3. Using advanced analysis techniques

This section concerns advanced build and analysis techniques.

2.6.3.1. Running an analysis without `cov-build`

Sometimes it is difficult or impossible to build a complete code project. In this case, you can still analyze source code: As an alternative to using the `cov-build` command, you can first compile your source in debug mode, using the standard compiler for the language in question (for example, `javac` or `vbc`). After you compile, use the language-specific version of `cov-emit` (for example, `cov-emit-java` or `cov-emit-vb`) to parse the source. Finally, run `cov-analyze` to complete the analysis.

1. Compile your code base using debug information.

   Building in debug mode (for example, with the `-g` option to `javac`, or with the `debug="true"` Ant compile task) allows Coverity Analysis to analyze the compiled code. In the standard analysis flow (see Chapter 2.1, *Getting started with Coverity analyses*), the `cov-build` command automatically runs the compiler in debug mode.

2. For each time you invoke the compiler, run the appropriate version of `emit` as well, using the `--compiler-outputs` option. This captures a build of your source code to the intermediate directory.

   > **Important**
   >
   > You must run the `emit` command on the same source and class files (those class files specified in the classpath) on which you ran the compiler. The `--compiler-outputs` must point either to all of the possible parent directories of the compiler outputs, or to a common parent directory for all of the compiler outputs.

   For example:

   ```
   > cov-emit-java --findsource src \
     --findjars lib;build-lib/ --dir my/intermediate/dir \
     --compiler-outputs build/classes/;build/junitclasses/
   ```

   > **Note**
   >
   > On Windows systems, the semicolons (`;`) shown in the example serve as path separators. On Unix-style platforms (including macOS and Linux), the path separators should be colons (`:`).

   For more detailed information about this command, see the `cov-emit-java` section.

3. Run `cov-analyze`.

   For guidance, see Step 5 in Chapter 2.1, *Getting started with Coverity analyses* .

4. Commit the defect data to the Coverity Connect database.

   For guidance, see Step 6 in Chapter 2.1, *Getting started with Coverity analyses* .

2.6.3.2. Adding custom models with `cov-make-library`

You can use the `cov-make-library` command to create custom models for methods. Adding models has two benefits: finding more bugs and eliminating false positives.

For example, to analyze Java source you might write a method that models a third-party resource allocator, allowing Coverity Analysis to detect and report defects if that allocator is called incorrectly or is misused. To add models, you would begin by coding the methods in Java. A model for analysis can use library methods that are already defined; it can also use methods that are custom-coded by invoking primitives (see Models and Annotations in Java for details) or using the library.

When analyzing C or C++ source code, you can take advantage of interprocedural analysis. This is described in the section that follows, “Adding custom C and C++ models with the `cov-make-library` command”.

2.6.3.3. Adding custom C and C++ models with the `cov-make-library` command

Coverity Analysis can run interprocedural analyses on the code base.

2.6.3.3.1. Prerequisites for interprocedural analysis

To run a thorough interprocedural analysis, one of the following must be present:

- Code in the intermediate directory that is undergoing analysis.
- A user-defined model. See Section 1.4.6.2, “Using custom models to improve analysis results”.
- Models derived from a preceding analysis. See Section 2.6.3.4, “Running analyses that use derived models for C and C++ source code”.

2.6.3.3.2. Using `cov-make-library`

You can use the `cov-make-library` command to add custom models to Coverity Analysis. See "Writing a Model: Finding New Defects" more information about modeling a specific behavior.

The `cov-make-library` command behaves as follows:

- Uses mangled function names for C++ models, but not for C models.
- Uses a C++ extension for C++ models: `.cpp`, `.cc`, or `.cxx`.

Note

Typically, `cov-make-library` expects header (`.h`) files to reside in same directory as the source files that use them. If headers or other files to include reside in a different directory, use the `cov-make-library --compiler-opt` flag to specify additional include directories.

C++ programs use `extern "C"` semantics to declare C functions, and those semantics apply to the library models as well. Because library files are parsed only (rather than compiled and built into executables), you can use external references to undefined functions or types. Rather than relying on linkage, the analysis uses function names to determine which models to use.

To create the model files, specify the following arguments to the `cov-make-library` command:

- (Required) The list of source files that contain the stub library functions
- (Optional) The computed model’s output file name from the `-of <modulefile>` option of the `cov-make-library` command

Note

If you do not specify the `-of <modulefile>` option, the output file goes into the default location.

- (Optional) The path to the XML configuration file (`-c <path/to/coverity_config.xml>`)

Note

In most cases, the default specifications for configuration and output files work correctly.

For more examples, see the following sections that describe how to override and add specific models for allocators and panic functions.

The `cov-make-library` command creates either the file `user_models.xmldb` or the output file that you specified with the `-of` option to the `cov-make-library` command. If the output file already exists, the most current output is appended to it. The following order of precedence determines the directory where the model file gets created:

1. The `-of` option, if you specified it
2. `<install_dir_sa>/config`

The `coverity_config.xml` file contains an encoded version of the models. The analysis reads these models and gives them precedence over other models. If there is a conflict, the models the user explicitly specifies are always used. To indicate to `cov-analyze` that it should read the `<user_file>.xmldb` file, specify it on the command line by using the `--user-model-file` option.

2.6.3.3.2.1. Determining which functions are analyzed and called

Knowing which functions are unimplemented is useful for determining which functions to model. The file `<intermediate_directory>/output/callgraph-metrics.txt` lists which functions are implemented and unimplemented, and how many callers each function has. This file is generated when you add the `--enable-callgraph-metrics` option to the `cov-analyze` command.

Coverity Analysis uses the build process to model and analyze functions, in one of the two following ways:

- If the build process captures a function’s definition, the function is treated as though it is implemented, and Coverity Analysis analyzes the function to build a model for it.
- Otherwise, the function is treated as though it is unimplemented, and does not not have an explicit model (as specified by the `--model-file` ↗ option). In this case, Coverity Analysis makes assumptions about the function in a way that avoids reporting false positives in callers—unless Coverity Analysis has been configured not to do so; for example, by using the option `-co RESOURCE_LEAK:allow_unimpl`.

Coverity Analysis provides a model library of common unimplemented functions such as `malloc()`.

Coverity Analysis also tracks how many times functions, both implemented and unimplemented, are called. This number is the total number of callers that call a function, both directly and indirectly (through one or more other functions). The number of callers for an unimplemented function is useful for determining which functions are a high priority to model. Looking at the number of callers of implemented functions can be useful as well for understanding the code base's architecture.

To find out which functions are analyzed and called:

1. When you run the `cov-analyze` command, add the `--enable-callgraph-metrics` option.
2. When the analysis completes, open the file `<intermediate_directory>/output/callgraph-metrics.txt`. This file lists each function as implemented or unimplemented. The number next to each function is the total number of direct and indirect callers for that function.
3. To see which functions might be good candidates for modeling, look for unimplemented functions that have a high number of callers.

The following table describes columns in the file that can help you determine which functions are analyzed and called.

Table 2.6.2. Important Data in Callgraph Metrics Files (CSV format)

CSV Column	Details
`call_count`	Number of calls of the function (see `unmangled_name` for the name of the function). Count is important for recursive (R) functions.
`TU`	Indicates whether the function has been implemented. TU = -1 Function is not implemented. TU ≠ -1 Function is implemented. For additional detail about the values, you can run the following command: `cov-manage-emit --dir <dir> -tu N list`
`qualifiers`	C Compiler-generated function V Virtual function R Recursive function T Templated function
`cycle_id`	Important for recursive (R) functions.
`module`	Helps identify the source of the model information.
`model_type`	Indicates whether a model for the function was found and whether it is a built-in or user-created model.

CSV Column	Details
	No_Model The function is not modeled. User_Model The function was modeled by a user. Builtin_Model The function was modeled by Coverity developers. Collected_Model The function model was specified through `--derived-model-file`.
`model_file`	Provides the path to a model file if the function is modeled. • For unmodeled function (where `model_type` is `No_Model`): None • For function modeled by a user (where `model_type` is `User_Model`): Filepath to `model.xmldb` • For a built-in model (where `model_type` is `Builtin_Model`): Filepath to `builtin-models.db`

2.6.3.3.2.2. Suppressing macro expansion to improve modeling

Complex macros sometimes cause Coverity Analysis to misinterpret parts of the code. Most commonly, this issue occurs when a model of a library function, such as `strcpy`, is incorrectly defined as a macro by the native compiler. In this case, it is necessary to suppress macro expansion so that Coverity Analysis can identify the model as a function call.

Macro expansion can be suppressed by using the `#nodef` syntax of the Coverity compiler:

- `#nodef macroname`

  This form can be used, for example, to convert a macro implementation of a standard C library function into a function call:

  ```
  #nodef strcpy
  ```

  For a more complete example, see Figure 2.6.1, “A sample ‘user_nodefs.h’ file”.

- `#nodef macroname value`

  This form is useful if you need to model a macro using a function name that differs from the name of the macro, thereby preventing your model function from conflicting with another function that might exist in your code base. For example:

  ```
  #nodef strcpy model_strcpy
  char *model_strcpy(char *, const char *);
  ```

  Note that the function declaration can appear in this file or elsewhere.

- `#nodef macroname(x,...) value`

  In addition to allowing for a different function name, this form allows you to model a macro (such as `#define my_assert(int) { ... } `). For example:

  ```
  #nodef my_assert(x) my_assert_model(x);
  void my_assert_model(int x);
  ```

  Then you can provide a model for `my_assert_model`.

The last two examples suppress the definition of a macro, while providing an alternative definition of the macro. The alternative overrides all future definitions of the macro.

Note

A commented, but otherwise empty template is provided at:

```
<install_dir_sa>/config/user_nodefs.h
```

If you insert company-specific `#nodef` directives in this file, the `cov-configure` command ensures that compilations with the Coverity Analysis compiler (which is invoked when you run `cov-build`) will include the configuration directives in `user_nodefs.h`.

Figure 2.6.1. A sample ‘user_nodefs.h’ file

```
#nodef strpbrk
#nodef memset
#nodef strstr
#nodef free
#nodef snprintf
#nodef memcpy
#nodef gets
#nodef fgets
#nodef strcpy
#nodef setjmp
#nodef strdup
#nodef memcmp
#nodef strrchr
#nodef sigsetjmp
#nodef strcmp
#nodef vsprintf
#nodef puts
#nodef vprintf
#nodef strcpy
#nodef freopen
#nodef printf
#nodef vfprintf
#nodef fread
#nodef realloc
#nodef fclose
#nodef fopen
#nodef sprintf
#nodef vsnprintf
#nodef fprintf
#nodef strncmp
#nodef fwrite
#nodef malloc
#nodef strchr
#nodef calloc
#nodef KASSERT
#nodef assert
#nodef BUG
#nodef BUG_ON
```

2.6.3.3.2.3. Adding a prototype for a function

Suppressing macro expansion (see Section 2.6.3.3.2.2, “Suppressing macro expansion to improve modeling”) might require an additional step of adding a prototype for the function if a function of the same name is not declared; otherwise, the function cannot be called in C++, and in C will cause PW.IMPLICIT_FUNC_DECL warnings. The prototype can be placed in `user_nodefs.h` so that only Coverity Analysis builds will see the prototype instead of the macro.

To increase the accuracy of the analysis, you might want to create a model for a prototype and register it with Coverity Analysis. For example, if you have a macro assertion such as:

```
#nodef my_assert
    void my_assert(int x);
```

... then you can create a model in a separate source file, such as:

```
void my_assert(int x) {
    if (!x)
    __coverity_panic__();
}
```

... and use the `cov-make-library` command to build a model from this source. For more information about models, see chapter 5 of the *Coverity 2020.12 Checker Reference*. ↗.

2.6.3.4. Running analyses that use derived models for C and C++ source code

Coverity Analysis performs interprocedural analyses that generate models of all the source code that is analyzed. Because the source code that you are developing often calls functions in libraries and other peripheral source that are unlikely to change much (if at all), it can be time-consuming and unnecessary to reanalyze them. To help address this issue, Coverity Analysis allows you to use the models that were derived from the original analysis of such code.

After building and completing an analysis of all source code (which includes source code that is undergoing development and the libraries and other peripheral code that it uses), you can continue to re-run the analysis on the source code that is undergoing development. However, instead of always running the analysis directly on the source for the libraries and other peripheral code, you can make analysis use the models that were derived from the analysis of the full code base.

To use derived models:

1. Generate a `derived_models.xmldb` file through the `--output-file` option to the `cov-collect-models` command.

   This is a one-time step to perform only after running an analysis of the full code base, including the libraries and other peripheral code. You will need to repeat the remaining steps according to your internal build and analysis schedule.

2. Pass the file to `cov-analyze` through the `--derived-model-file` option.

3. Rebuild only the portion of the code base that is undergoing development, omitting the peripheral code bases.

   You typically use `cov-build` for this step.

4. Reanalyze the build along with the derived models in `derived_models.xmldb`.

   The `derived_models.xmldb` file is not read by default. When you invoke `cov-analyze`, specify the `.xmldb` file by using the `--derived_model_file` option.

   For each analyzed function call, the model in the `derived_models.xmldb` file for that function is used only if there are no other matching user models (or any other models) that are undergoing analysis in the current intermediate directory. When developers modify their source files, models will be automatically generated for the functions in that code, and any models in the `derived_models.xmldb` file for those functions will be ignored because they are outdated.

There will be no links into the details for derived models.

Chapter 2.7. Configuring compilers for Coverity Analysis

Table of Contents

Before configuring compilers for your production environment, answer several basic questions to determine your system's configuration:

1. Do I know which compilers and their versions I am using in my build and does Coverity support them?

   Unsupported compilers can cause incompatibilities when the Coverity compiler attempts to parse your code. Support for additional compilers is based on a variety of factors including customer need, the compiler's availability, and how many customers are using it. To request that Coverity extend support to your compiler you can send an email request to `software-integrity-support@synopsys.com`.

   Use the following command to list the supported compiler types and the values that are used for identifying them for compiler configurations:

   ```
   > cov-configure --list-compiler-types
   ```

   The following example shows a small portion of the output:

   ```
   csc,csc,C#,FAMILY HEAD,Microsoft C# Compiler
   g++,g++,CXX,SINGLE,GNU C++ compiler
   gcc,gcc,C,FAMILY HEAD,GNU C compiler
   java,java,JAVA,SINGLE,Oracle Java compiler (java)
   javac,javac,JAVA,FAMILY HEAD,Oracle Java compiler (javac)
   msvc,cl,C,FAMILY HEAD,Microsoft Visual Studio
   ```

   In the example, `csc` is the value used to identify the compiler, and `Microsoft C# Compiler` is the name of the supported compiler. More generally, the output contains compiler configuration values for the `--comptype` and `--compiler` options and related information. Note that FAMILY HEAD values are used to configure a related family of compilers (for example, `gcc` for GNU gcc and g++ compilers), while SINGLE values are for single-compiler configurations (for example, `g++` for the GNU g++ compiler only).

   For support documentation, see supported compiler information in the *Coverity 2020.12 Installation and Deployment Guide*.

2. Do I use the same compilers and the same versions each time I do a build?

   If all of your builds are done on the same machine and in a controlled environment, the likely answer is yes.

If your builds are done on many different machines with different compiler versions, then the answer might be no.

If you use different versions of the same compiler that have different binary names (such as different versions of GCC with binary names such as `mips-gcc` or `arm-gcc`, or `gcc32` or `gcc29`), the answer is no.

If you use different compilers each time, such as GCC 4 one time, armcc 3 the next time, the answer is no.

3. How many different machines do I intend to install Coverity Analysis on? If more than one, are my compilers installed at the same hard disk location on all of them? Do multiple machines use the same set of configuration files?

4. Am I using ccache or distcc?

   If you are using either of these tools, you sometimes need to use the `--comptype prefix` setting when configuring Coverity Analysis for your compiler, as shown in the examples below. This setting can help avoid unexpected defect reports.

ccache configuration

- If you use `ccache` (for example, with `gcc`), your `cov-configure` command line should specify the following:

  ```
  > cov-configure --comptype prefix --compiler ccache
  ```

  ```
  > cov-configure --comptype gcc --compiler gcc
  ```

`distcc` configuration

- If `ccache` is set up to execute `distcc` (for example, through the CCACHE_PREFIX variable), it is only necessary to configure the prefix for `ccache`.

- If your `distcc` installation uses the name of the underlying compiler (for example, `gcc -c foo.c`, where `gcc` is really `distcc`), your `cov-configure` command line should specify the following:

  ```
  > cov-configure --comptype <comptype_of_real_compiler> \
     --compiler <distcc_executable_name>
  ```

- If you are prepending `distcc` to compiler command lines (for example, `distcc gcc -c foo.c`), your `cov-configure` command line should specify the following:

  ```
  > cov-configure --comptype prefix --compiler distcc
  ```

  ```
  > cov-configure --comptype <comptype_of_real_compiler> \
     --compiler <first_argument_to_distcc>
  ```

  The first argument to `distcc` is the name of executable for the real compiler, for example, `gcc`.

- If `distcc` is used directly as a compiler (for example, `distcc -c foo.c`), your command line should specify the following:

  ```
  > cov-configure --comptype <comptype_of_real_compiler> \
     --compiler distcc
  ```

The answers to questions 2 and 3 help to determine whether you should generate a *template* configuration or a standard configuration. A template configuration is a general configuration file that specifies the name of a compiler executable that is used in the build. The compiler's location and version are determined during the build. A standard configuration file specifies the compiler's full path, thus hard-coding it to a specific version. Before deciding which configuration type to use, consider each type's costs and benefits.

Table 2.7.1. Comparing template and standard configuration types

Template configuration	**Standard configuration**
You specify the compiler name, without the compiler's full path. Only requires one command to configure per compiler executable name, for example, all `gcc` compiler versions.	You must run `cov-configure` for each build compiler to configure its full path name.
Benefit— Makes the Coverity Analysis installation faster and easier. You can move the configuration across machines, without re-running the `cov-configure` command, even when the compilers are in different locations on different machines.	*Cost*— If compilers are in different locations on different machines, you must use the template configuration.
Cost— Configuring the compilers at build time incurs a cost each time a new compiler is encountered during the build. When using `cov-build`, that cost is only incurred once for each unique compiler in the build. When using `cov-translate` without `cov-build`, that cost is incurred on every single invocation of the Coverity compiler.	*Benefit*— All of the configuration is done once before any builds are invoked. There is no configuration cost each time a build is invoked.
Cost— In a build that has multiple versions of a single compiler (for example, multiple versions of `gcc` used in a single build), if one of those versions of `gcc` allows a source code construct that is incompatible with `cov-emit`, it is much more complex to add a configuration targeted to a single compiler version to resolve the issue.	*Benefit*— Each unique compiler configuration is generated in a separate configuration file. If a modification is required to improve compatibility with a single compiler version, there is a unique file and location available for making the required modifications.

We recommend template configurations for the following compilers: gcc, g++, qnx, tmcc, Tensilica Xtensa, Green Hills, and MetaWare. For information about creating template configurations, see Section 2.7.2, “Generating a template configuration”.

Note

A compiler configuration might be platform-specific. For example, if you configure a gcc or g++ compiler on a 32-bit system, you cannot use it for a build on a 64-bit system. Also, if you change a compiler's default options after configuring it, or install a different version of the compiler, its behavior might change and invalidate the configuration that you created earlier. Make sure that the compiler that you configure exactly matches the compiler that your build uses.

2.7.1. Generating a standard configuration

Each standard configuration that is generated configures one specific compiler installation on the system. Unlike a template configuration, which specifies the executables configured at build time, a standard configuration is a completed configuration file that specifies exactly how `cov-translate` and `cov-emit` are fully compatible with your native build's compilers. Since most common compiler types are hard-coded into the `cov-configure` command, specifying the compiler executable name usually provides enough information for `cov-configure` to determine the compiler's vendor, and whether the vendor's installation package includes other compiler executables.

The following table shows sample `cov-configure` commands for known compilers where the correspondence between executable name, vendor, and installation package is understood. You do not need to specify the full path if the compiler executable's location is included in the `PATH` environment variable. Before running these commands, see Section 2.7.3, “Compiler-specific configurations” to make sure your compiler does not require additional instructions to successfully generate its configuration.

Table 2.7.2. Configuration commands with standard compiler names

Issued command	Compiler description
`cov-configure --compiler <full/path/to>/gcc`	GNU compilers
`cov-configure --compiler <full/path/to>/armcc`	ARM/Thumb compilers
`cov-configure --compiler <full/path/to>/dcc`	Wind River compiler (formerly Diab)
`cov-configure --compiler <full/path/to>/icc`	Intel compiler for x86
`cov-configure --compiler <full/path/to>/cl`	Microsoft Visual C and C++ compilers
`cov-configure --compiler <full/path/to>/cc`	Sun Forte C and C++ compilers
`cov-configure --compiler <full/path/to>/cl470`	TI Code Composer Studio C and C++ compiler[a]
`cov-configure --compiler <full/path/to>/mcc`	Synopsys MetaWare C and C++ compilers

Issued command	Compiler description
`cov-configure --compiler <full/path/ to>/hcac`	

[a] Note that TI compilers require an environment variable to be set in order for `cov-configure` to properly probe compiler behavior. The environment variable should point to the include directories, and is specific to the compiler (for example, `C6X_C_DIR` for the C6000 compiler).

Because a standard configuration applies to a compiler installation, not a single compiler executable, a single invocation of `cov-configure` attempts to configure both the C and C++ compilers in the specified installation if the compiler names are not different than a standard installation.

Many C compilers can compile both C and C++ code depending on the compiler file's extension. The `cov-configure` command creates a different configuration file for each combination of compiler executable and language. Thus, `> cov-configure --compiler gcc` creates a configuration file for each of the following compiler and language combinations:

- `gcc` as a C compiler
- `gcc` as a C++ compiler
- `g++` as a C++ compiler

Additional usage instructions

- If you configure an ARM compiler, you must also configure its Thumb counterpart. Similarly, configuring javac configures any `java`, `apt`, and `javaw` (Windows systems only) commands found in the same `JAVA_HOME` directory tree.
- In the following cases, you must specify the `--comptype <type>` option to `cov-configure`:
  - The compiler has a non-standard name (for example, `i686-powerpc-gcc`).
  - The `cov-configure` command does not recognize the compiler name.

  For example:

  ```
  > cov-configure --compiler i686-powerpc-gcc --comptype gcc
  ```

  All compilers that are not listed in Table 2.7.2, “Configuration commands with standard compiler names” require the `--comptype` option.
- Some compilers require additional options. For example, GNU compiler installations that use a non-standard preprocessor (`cpp0`) path require the GNU `-B` option that specifies it:

  ```
  > cov-configure --compiler gcc -- -B/home/coverity/gcc-cpp0-location/bin
  ```

  The double-hyphen (--) indicates the end of the `cov-configure` options.

2.7.2. Generating a template configuration

You can invoke `cov-configure` with the `--template` argument to generate a template configuration.

Full template configuration for the gcc C/C++ compiler:

```
> cov-configure --template --compiler gcc --comptype gcc
```

Note that the full template configuration for Java and C# is not recommended.

The following alternatives generate a template configuration for the GNU GCC and G++ compilers (using `gcc`), Microsoft C and C++ compilers (using `msvc`), Java compilers (using `java`, not `javac`), and C# compilers (using).

Alternative template configuration for the gcc C/C++ compiler:

```
> cov-configure --gcc
```

[Recommended for C#] Alternative template configuration for the Microsoft C# compiler:

```
> cov-configure --cs
```

[Recommended for Java] Alternative template configuration for build capture with the Java compiler:

```
> cov-configure --java
```

[Recommended for Java] Alternative template configuration for Java filesystem capture:

```
> cov-configure --javafs
```

For more information about creating a template configuration, see the `--template` option in the `cov-configure` documentation.

The previous commands generate:

- The `<install_dir_sa>/config/coverity_config.xml` configuration file.
- The `<install_dir_sa>/config/template-gcc-config-0` sub-directory with its own `coverity_config.xml` file.

The configuration file specifies that `cov-build` configure `gcc` executables as compilers and that `cov-translate` treat them as compilers.

For Java programs, `cov-build` configures the executable and treats it as a Java compiler.

Creating a template configuration for one compiler also creates templates for any related compiler, just as in a standard configuration.

For example:

- gcc implies g++ (cc links to gcc as well on some platforms).
- javac implies `java`, `apt`, and `javaw` (on Windows systems).

To see a full list of supported compiler types, run the `cov-configure --list-compiler-types` option.

2.7.3. Compiler-specific configurations

Some compilers have unique compilation environments that Coverity Analysis simulates to properly parse the source code. Especially important are the predefined macros and include directories built into the compiler. Predefined macros can be configured into `nodefs.h`, and pre-included directories into `coverity_config.xml`. For more information about how to get `cov-translate` to add and remove command-line arguments to pass to `cov-emit`, see Section 1.4.6.3, “Using Coverity Analysis configuration files in the analysis”.

2.7.3.1. gcc/g++

Coverity Analysis is compatible with most gcc compiled code. This includes support for gcc-specific extensions. For example, Coverity Analysis can compile virtually all of the Linux kernel, which heavily uses many gcc extensions. Some known gcc incompatibilities include:

- Nested functions are not supported.
- Abbreviated function template syntax is not supported.
- Computed goto's are handled in a very approximate fashion.
- The `-fpermissive` compiler mode is not supported.
- The `__fp16` builtin type is not supported.

For Mac OS X

Mac OS X users, see Chapter 4.2, *Building with Xcode*.

Coverity Analysis compatibility with modern g++ versions is also good. Older g++ versions (before 3.0) are far more relaxed in their type checking and syntax, and their incompatibilities might be difficult to solve. The `--old_g++` option loosens Coverity Analysis's parsing and type checking enough to let many older code bases compile. If you specify the compiler version when you run `cov-configure`, this option is in `coverity_config.xml`.

Because `cov-configure` invokes the native compiler to determine built-in include paths and built-in preprocessor defines, the GNU C and C++ compiler might require additional steps to configure correctly.

To invoke it properly from the command line, the GNU compiler might require additional `cov-configure` options. In particular, GNU compiler installations that use a non-standard preprocessor (`cpp0`) path require the GNU `-B` option that specifies it:

```
> cov-configure --compiler gcc -- -B/home/coverity/gcc-cpp0-location/bin
```

If your build explicitly uses the GNU compiler on the command line with either the `-m32` or `-64` option, also supply the option to the `cov-configure` command. For example:

```
> cov-configure --compiler gcc -- -m32
```

On some platforms, gcc allows multiple '`-arch <architecture>`' options to be specified in a single compiler invocation. `cov-analysis` will only compile and analyze the source once, as though only the last `-arch` option specified on the command line was present. If all compiler invocations are not consistent regarding the last architecture specified on the command line, `cov-analysis` may produce false positive or false negative results.

2.7.3.2. CEVA compilers

Use a template configuration for the CEVA-XC12 compiler:

```
cov-configure --template --compiler cevaxccc --comptype ceva:xc12
```

2.7.3.3. Freescale Codewarrior compiler

Some Codewarrior compilers require subtypes when you configure them. Use the following `--comptype` values:

- Codewarrior for Starcore and SDMA: `cw:sdma`
- Codewarrior for Starcore DSP: `cw:dsp`
- Codewarrior for Starcore: `codewarriorcc:starcore` (for all targets of Codewarrior Starcore version 10.9 and later)
- Codewarrior for MPC55xx: `cw:55xx`
- Codewarrior for EPPC 5xx: `cw:5xx`

All other Codewarrior compilers require only the `cw` value for `--comptype`.

Note

Codewarrior HC12 beeps when it fails. While configuring this compiler, `cov-configure` will likely cause several compilation failures while probing, resulting in the beeping sound. This is expected behavior.

2.7.3.4. Green Hills compiler

Use a template configuration for the Green Hills C and C++ compiler. This is necessary because some native compiler options like `-bsp <my_hardwar_config>` and `-os_dir <dir>` change the behavior of the compiler and require different analysis configurations.

In this compiler's standard installation, the compiler executable names are `cc<target name>` (for C code) and `cx<target name>` (for C++ code). For example, the C compiler for the Power PC target is called `ccppc`. The compilers are located in an architecture-specific sub-directory of the Green Hills installation, such as `Linux-i86`. Additionally, there are compilers named `ccint<target name>`, and these should be configured as well if used.

Lastly, there is a binary called `ecom<target name>`. This is an undocumented internal binary that is used by some tools. This should e configured using `green_hills_ecom`.

For example:

```
cov-configure --template --compiler ccppc --comptype green_hills
cov-configure --template --compiler ccintppc --comptype green_hills
cov-configure --template compiler ecomppc --comptype green_hills_ecom
```

2.7.3.5. HighTec compiler

Use a template configuration for the HighTec Tricore compiler:

```
cov-configure --template --compiler tricore-gcc --comptype hightec:tricore
```

2.7.3.6. Keil compilers

The Keil compiler for the ARM target platform requires the device argument, and so you must pass the device argument to the compiler when configuring it with the `cov-configure` command. After the `cov-configure` options, specify the characters `--` and then the `--device` option. For example:

```
> cov-configure --comptype keilcc --compiler armcc -- --device=<device_name>
```

Use a template configuration for the Keil MDK for ARM Compiler:

```
cov-configure --template --compiler armcc --comptype armcc
cov-configure --template --compiler armclang --comptype armcc
```

2.7.3.7. Microchip compilers

The following Microchip MPLAB compilers are supported:

- For 8-bit devices, use compile name `xc8` and compile type `microchip:xc8`, or compile name `xc8-cc` and compile type `microchip:xc8cc`

  Note

  Before version 2.00, the XC8 C compiler supports only PIC MCUs, and the documented driver name is `xc8`. Starting with version 2.00, the XC8 C compiler supports both PIC and AVR MCUs, and the documented driver name is `xc8-cc`.

- For 16-bit devices, use compile name `xc16-gcc` and compile type `microchip:xc16`

- For 32-bit devices, use compile name `xc32-gcc` and compile type `microchipcc:xc32`

Use a template configuration for the Microchip Compilers:

```
cov-configure --template --compiler xc8 --comptype microchip:xc8
cov-configure --template --compiler xc8-cc --comptype microchip:xc8cc
cov-configure --template --compiler xc16-gcc --comptype microchip:xc16
cov-configure --template --compiler xc32-gcc --comptype microchipcc:xc32
```

2.7.3.8. Microsoft Visual C and C++

Because `cov-configure` invokes the native compiler to determine built-in include paths and built-in preprocessor defines, the Microsoft Visual C and C++ compiler might require additional steps to configure correctly.

The Microsoft Visual C and C++ compiler executable is named `cl.exe`. Generally, `cl.exe` requires that the path settings include the location of all required DLLs.

Coverity Analysis can simulate parsing bugs that occur in some versions of Microsoft Visual C and C++. Supply the correct version of MSVC to the `cov-configure` command to get the correct `cov-emit` arguments automatically. The `--typeinfo_nostd` option allows some codebases, which rely on the typeinfo structure to not be in the `std` namespace, to compile.

The Coverity compiler supports cross compiling to 64-bit MSVC platforms.

2.7.3.9. PICC compiler

The compiler executable name is `pic1` and the ID is `picc`. Note the following:

- Coverity cannot compile PICC programs in which "@" occurs in either comments or quoted strings.
- PICC allows an extension of binary literals specified by a leading 0b, for example 0b00011111. This is supported by passing the `--allow_0b_binary_literals` flag to `cov-emit` whenever `cov-configure` is given `--comptype picc` or `--compiler picl`.

2.7.3.10. QNX compiler

Use a template configuration for the QNX compiler. The native compiler options `-V` and `-Y` change the behavior of the compiler and require different Coverity Analysis configurations. For example:

```
cov-configure --template --compiler qcc --comptype qnxcc
```

2.7.3.11. Qualcomm Kalimba C compilers

Use a template configuration for the Qualcomm Kalimba C compilers:

```
cov-configure --template --compiler kcc --comptype kalimba:kcc
cov-configure --template --compiler kalcc --comptype kalimba:kalcc
cov-configure --template --compiler kalcc32 --comptype kalimba:kalcc32
```

2.7.3.12. Renesas compilers

Use a template configuration for the Renesas compilers:

```
cov-configure --template --compiler ch38 --comptype renesascc
```

2.7.3.13. STMicroelectronics compilers

Use a template configuration for the STMicroelectronics compilers:

```
cov-configure --template --compiler st20cc --comptype st20cc
```

2.7.3.14. Sun (Oracle) compilers

Use a template configuration for the Sun (Oracle) compilers:

```
cov-configure --template --compiler cc --comptype suncc
```

2.7.3.15. Synopsys MetaWare compilers

Use a template configuration for the Synopsys MetaWare C and C++ compilers:

```
> cov-configure --template --compiler ccac --comptype metawarecc:ccac
```

```
> cov-configure --template --compiler hcac --comptype metawarecc:mcc
```

```
> cov-configure --template --compiler mcc --comptype metawarecc:mcc
```

Language Limitations

The following language extensions are not supported for the specified compilers:

- `long long` variants of the ISO/IEC TR 18037 fixed point `_Accum` and `_Fract` types are not supported for the hcac and mcc compilers.
- Use of the ISO/IEC TR 18037 fixed point `_Accum` and `_Fract` types as the element type of vector types is not supported for the ccac, hcac, and mcc compilers.
- Use of the ISO/IEC TR 18037 fixed point `_Accum` and `_Fract` types and fixed point literal expressions in C++ code is not supported for the hcac and mcc compilers.

Functions and variable initializers that use these features will not be analyzed. However, other functions and variable initializers within the same translation unit will still be analyzed.

2.7.3.16. Texas Instruments C and C++ compilers

Coverity supports 2.53 and later of a number of C and C++ TMS compilers. Use `cov-configure list-compiler-types` for a complete list. The compiler's executable name in a TMS470R1x installation, for example, is generally `cl470.exe`. To configure this compiler, you might specify the command line as follows:

```
> cov-configure --compiler <TMS Installation>\cgtools\bin\cl470.exe \
  --comptype ti
```

Change the preceding example to match the version and installation path of the TMS compiler tools that you are using.

When `cov-build` is launched for a project that uses the TMS compiler, all of the invocations of the compiler will be accompanied with a call to `cov-emit` unless one of the following command-line arguments is present:

1. `-ppd` (generate dependencies only)
2. `-ppc` (preprocess only)
3. `-ppi` (file inclusion only)
4. `-ppo` (preprocess only)

There are currently a small number of unsupported options and keywords to the TMS compilers. These keywords can be translated into nothing, when appropriate, or into a supported ANSI C and C++ equivalent using user_nodefs.h. Contact Coverity support regarding any parse errors that you see with this compiler.

Use a template configuration for the Texas Instruments C7000 compiler:

```
cov-configure --template --compiler cl7x --comptype ti:cl7x
```

Language Limitations

The following language extension is not supported for the specified compiler:

- Operators and functions for vector data types are not supported for the Texas Instruments C7000 compiler.

Functions and variable initializers that use these features will not be analyzed. However, other functions and variable initializers within the same translation unit will still be analyzed.

2.7.3.17. Trimedia C and C++ compilers

Use a template configuration for the Trimedia compilers:

```
cov-configure --template --compiler tmcc --comptype tmcc
```

2.7.3.18. Tensilica Xtensa C and C++ compiler

Use a template configuration for the Xtensa compiler:

```
cov-configure --template --compiler xt-xcc --comptype xtensacc
```

2.7.3.19. Clang compilers

Use a template configuration for the Clang compilers:

```
cov-configure --template --compiler clang --comptype clangcc
```

For Mac OS X

Mac OS X users, see Chapter 4.2, *Building with Xcode*.

2.7.3.19.1. Supported language extensions

Apple Blocks

Support for the Apple Blocks extensions is provided for C and C++ code, and is automatically enabled when enabled in native compiler invocations. Interprocedural analysis of Block invocations requires that `cov-analyze` be invoked with one of the `--enable-single-virtual` or `--enable-virtual` options.

2.7.3.19.2. Supported compliance standards for Clang compilers

Coverity Analysis supports the following compliance standards for Clang compilers:

- MISRA C 2004
- MISRA C 2012
- MISRA C++ 2008
- AUTOSAR C++14
- SEI CERT C
- SEI CERT C++ (only the following subset of rules):

  CERT DCL50-CPP
  CERT DCL52-CPP
  CERT DCL53-CPP
  CERT DCL54-CPP
  CERT DCL55-CPP
  CERT DCL56-CPP
  CERT DCL57-CPP
  CERT DCL58-CPP
  CERT DCL59-CPP
  CERT EXP52-CPP
  CERT EXP57-CPP
  CERT EXP59-CPP
  CERT EXP61-CPP
  CERT EXP62-CPP
  CERT EXP63-CPP
  CERT CTR51-CPP
  CERT CTR56-CPP
  CERT CTR58-CPP
  CERT STR52-CPP
  CERT STR53-CPP
  CERT MEM50-CPP
  CERT MEM52-CPP
  CERT MEM53-CPP
  CERT MEM54-CPP
  CERT MEM56-CPP
  CERT FIO50-CPP
  CERT FIO51-CPP
  CERT ERR52-CPP
  CERT ERR53-CPP
  CERT ERR54-CPP
  CERT ERR55-CPP
  CERT ERR56-CPP
  CERT ERR57-CPP
  CERT ERR58-CPP
  CERT ERR61-CPP
  CERT OOP50-CPP
  CERT OOP53-CPP

CERT OOP54-CPP
CERT OOP55-CPP
CERT OOP56-CPP
CERT OOP57-CPP
CERT OOP58-CPP
CERT CON50-CPP
CERT CON51-CPP
CERT CON52-CPP
CERT CON53-CPP
CERT CON54-CPP
CERT CON55-CPP
CERT CON56-CPP
CERT MSC50-CPP
CERT MSC51-CPP
CERT MSC53-CPP

For more information, refer to the *Coverity Checker Reference*.

2.7.3.19.3. Clang limitations

Clang compilers have various use limitations with Coverity products. These limitations are described in this section.

Coverity features that are not supported (when using a Clang compiler)

- Coverity Architecture Analysis
- Test Advisor
- SEI CERT C++ rules not listed in the preceding section
- ISO TS 17961:2013
- The `#pragma` Coverity-compliance preprocessing directive and `_Pragma` Coverity-compliance preprocessing operator
- Coverity parse warning checkers
- The `--preprocess-next` option for `cov-build`
- The `--record-with-source` (`-rws`) option for `cov-build`

Language limitations

The following language extensions are not supported. Functions and variable initializers that use these features will not be analyzed. However, other functions and variable initializers within the same translation unit will still be analyzed.

- Altivec vector types and expressions
- C++ coroutines TS language extensions

- CUDA language extensions
- The Microsoft `dependent exists` statement
- The Microsoft `__interface` user defined type specifier
- Microsoft structured exception handling statements
- OpenMP language extensions
- The `char8_t` builtin type
- OpenCL language extensions

Compiler driver limitations

- Clang driver invocations that specify the '`-cc1`' option are not supported.
- Clang driver invocations that specify multiple '`-arch <architecture>`' options are not supported.
- The Pre-Tokenized Header (PTH) feature available in Clang 7 and earlier is not supported.

`cov-emit` features that are not supported (with Clang)

- Macro expansion suppression (#nodef). See *Coverity Analysis 2020.12 User and Administrator Guide* for more information.

2.7.4. Using predefined macros for Coverity Analysis-specific compilations

The Coverity compiler defines several special preprocessor macros that you can use to conditionally compile code. These are described in the table below. For example, a macro that does not normally terminate execution can be redefined to act as though it does for the purpose of static analysis

```
#ifdef __COVERITY__
#define logical_assert(x) (assert(x);)
#else
#define logical_assert(x) (if (!x) printf("Variable is null!");)
#endif
```

Macro	Meaning
`__COVERITY__`	Conditionally compile code.
`__COVERITY_HOTFIX__`	The current hotfix release level. For example, for release 1.2.3.4, the hotfix is 4.
`__COVERITY_MAINTENANCE`	The current maintenance release level. For example, for release 1.2.3.4 the maintenance level is 3.
`__COVERITY_MAJOR__`	The current major release level. For example, for release 1.2.3.4 the major level is 1.

Macro	Meaning
`__COVERITY_MINOR__`	The current minor release level. For example, for release 1.2.3.4 the minor level is 3.
`__COVERITY_VERSION__`	The current version encoded as a single integer, with each two digits representing a single component of the version. For example, 1.2.3.4 yields 1020304.

2.7.5. Modifying preprocessor behavior to improve compatibility

Native compilers usually define some macros, built-in functions, and predefined data types. The Coverity compiler does not automatically define all of these by default. Instead, it relies on the following configuration files, which are generated by `cov-configure`:

- `<install_dir_sa>/config/<comp_type>-config-<replaceable>number</replaceable>/coverity-compiler-compat.h`
- `<install_dir_sa>/config/<comp_type-config-<number/coverity-macro-compat.h`

These files are pre-included on the `cov-emit` command line before any other files. Once `cov-emit` parses these files, the definitions should match the native compiler definitions.

Additionally, each time `cov-emit` runs a compilation process, it pre-includes a file called `user_nodefs.h`, which is optional and might not exist. You can place directives in this file to correct problems with the generated compiler compatibility headers. Because this file is shared by all compiler configurations, the definitions that apply to a single compiler should be sectioned off using `#if/#ifdef` and compiler specific macros.

One common cause of incompatibilities is an incomplete deduction of the list of built-in preprocessor definitions by `cov-configure`. Adding these definitions to the `user_nodefs.h` can correct this issue. See Part 5, “Using the Compiler Integration Toolkit (CIT)” for more information about working around compiler incompatibilities.

Note

In general, because using `user_nodefs.h` improperly in C++ can cause parse errors in every file name in a build, Coverity recommends that you *do not* modify `user_nodefs.h` without help from Coverity Support (`software-integrity-support@synopsys.com`). Given the correct directions, this enhancement to the preprocessor of the Coverity Analysis compiler can be a powerful aid in following tasks:

- Finding additional software issues by removing macro expansions that obscure the semantics of the code (see Section 2.6.3.3.2.2, “Suppressing macro expansion to improve modeling”).
- Providing workarounds for compiler incompatibilities that might otherwise require comprehensive changes to `cov-emit` or `cov-translate` and take some time to resolve.

Part 3. Setting up Coverity Analysis for use in a production environment

You can deploy Coverity Analysis alone in a centralized (server-based) build system or in combination with Coverity Desktop Analysis, which allows developers to run local analyses of source code from their desktops or through their IDEs. This section covers the server-based deployment model. For information about the combined deployment model, see *Coverity Desktop Analysis 2020.12: User Guide* .

Chapter 3.1. The Central Deployment Model

The central deployment model separates administrative tasks from the tasks that developers perform.

- As an administrator, you check out the latest source to a platform that supports Coverity Analysis, analyze the source code, and commit the analysis results to Coverity Connect. To deploy Coverity Analysis based on this model, you need to write a script that automatically runs the Coverity Analysis commands needed to analyze a given code base (see Chapter 1.4, *Coverity analyses*).

  Tip

  Completing an analysis of the code base in Coverity Wizard can help because the Coverity Wizard console output (which you can save in a text file) lists all the Coverity Analysis commands and options that it runs in a given analysis. See *Coverity Wizard 2020.12 User Guide* for details.

  You can integrate Coverity Analysis with the build process to provide Coverity Analysis consumers with analysis results from snapshots of the latest source code (for details, see Chapter 3.3, *Integrating Coverity Analysis into a build system*).

  As mentioned in Part 3, “Setting up Coverity Analysis for use in a production environment”, you can also combine this model with an IDE-based deployment model if your developers are using Coverity Desktop for Eclipse or Visual Studio.

- After using Coverity Connect to discover, prioritize, and understand the software issues that they own, developers check out the affected source code files from the source repository, fix one or more of the issues locally, and then check in their fixes to the source repository. Coverity Connect will reflect the fixes in the next round of analysis results (the next snapshot) of the code base that contained the issues.

Chapter 3.2. Coverity Analysis Deployment Considerations

Software organizations often produce several products, each of which typically consists of a number of related code branches and targets for supported platforms, product versions, trunks, and development branches. The Coverity Analysis deployment needs to analyze each code base on a regular basis so that the issues that developers see in Coverity Connect reflect their changes to the code bases.

To plan for your deployment:

1. **Determine which types of analyses to run:**

   - Code base analyses

   - Incremental analyses, parallel analyses, or some other type of analysis process

     For details about these topics, see Part 2, "Analyzing source code from the command line ".

   As part of this process, you also need to perform the following tasks:

   a. **Determine which checkers to run.**

      By default, Coverity Analysis enables a set of checkers that are covered by your Coverity Analysis license. You can work with development team leads and power users to determine whether to enable additional checkers or disable other checkers (see Enabling/Disabling Checkers), and, if necessary, to create custom checkers (see *Learning to Write CodeXM Checkers* ↗.).

   b. **Consider whether to model any functions or methods.**

      Modeling functions in third-party libraries, for example, can improve analysis results. For more information, see Using Custom Models of Functions and/or Methods.

2. **Plan Coverity Connect projects and streams for your analysis results:**

   To allow developers to view and manage their issues, administrators use Coverity Connect to define streams and group them into projects. For example, a technical lead might define a project that is composed of all the streams for a single software product. Such a project might include Linux, MacOS, and Windows target builds, along with multiple versions of each. A manager might need to see a project that consists of all the code streams in a given department.

   For additional information about this topic, see Prerequisites to completing an analysis.

3. **Consider whether to push third-party issues to Coverity Connect so that developers and team leads can view and manage them along with their Coverity Analysis analysis issues.**

   For more information, see Using Coverity Analysis to commit third-party issues to the Coverity Connect database.

4. **Consider whether to use Coverity Desktop in conjunction with Coverity Analysis:**

   For details, see *Coverity 2020.12 for Eclipse, Wind River Workbench, and QNX Momentics: User Guide* and *Coverity Desktop 2020.12 for Microsoft Visual Studio: User Guide*.

5. **Think about how to integrate Coverity Analysis into your build system:**

   See Chapter 3.3, *Integrating Coverity Analysis into a build system*.

   As part of this process, you also need to complete the following tasks:

   a. **Check Coverity Analysis platform and compiler support:**

      Refer to "Supported Platforms" in the *Coverity 2020.12 Installation and Deployment Guide*. If you are using a C/C++ compiler that is not supported, it is possible to extend the compatibility of compilers with Coverity Analysis. For details, see Part 5, “Using the Compiler Integration Toolkit (CIT)”.

      Note

      For performance reasons, the following directories should not reside on a network drive:

      - The Coverity Analysis installation directory.
      - The intermediate directory. Instead, to maximize the performance of the analysis, this directory should reside on the build host.
      - The analyzed code.

      It is possible to run the analysis on a machine that is different from the one used for the build, even one with a different operating system or architecture, so long as the same version of Coverity Analysis is installed on both systems. This setup supports the specialization of machines, distributed builds, and the AIX platform, which does not have the `cov-analyze` command. To run an analysis on a different machine, you need to copy the self-contained intermediate directory to a local disk on the chosen host.

      Reminder: C# security analyses should run on Windows. Analyzing C# Web applications on Linux is not supported.

   b. **Determine memory requirements for the analyses you intend to perform:**

      For details, "Coverity Analysis Hardware Requirements" in the *Coverity 2020.12 Installation and Deployment Guide*.

   c. **Determine the analysis interval:**

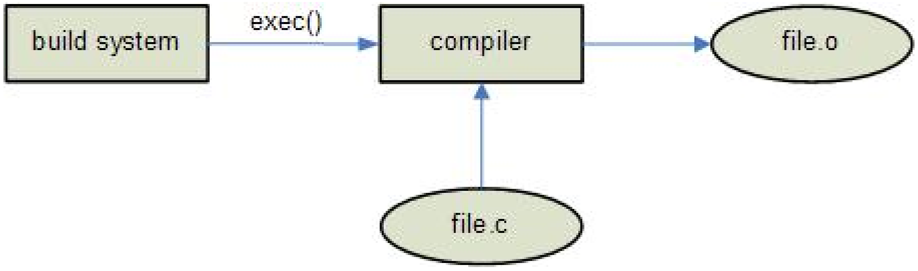
      Because developers continually modify the code base, regularly scheduled Coverity Analysis analyses are necessary to provide information about the introduction of new issues and the elimination of existing ones. For example, you might run the analysis on a nightly basis.

Chapter 3.3. Integrating Coverity Analysis into a build system

Table of Contents

Using a C/C++ code base as an example, Figure 3.3.1, “A typical build system” shows how many build systems interact with the compiler. The build system calls an `exec`-type function to run the compiler process. The compiler process reads in the source files and produces binary object files.

Figure 3.3.1. A typical build system



There are two standard ways of integrating Coverity Analysis into this kind of build system. One way, shown in Figure 3.3.2, “Coverity Analysis integration using the cov-build command”, uses the `cov-build` command (described in Section 3.3.2, “Integrating Coverity Analysis into the build environment — cov-build”) to automatically detect invocations of the compiler. This method usually requires no changes to the build system itself. Instead, it relies on "wrapping" the build system so that Coverity Analysis can piggyback on the compiler invocations. The regular build system is invoked by the `cov-build` command, which sets up the operating environment such that calls to `exec`-type functions made by the dynamically-linked build process are intercepted by the Coverity Analysis `capture` stub library. The capture library calls the `cov-translate` command to translate the compiler command-line arguments to the command line arguments of the Coverity analysis engine (also called the Coverity compiler). The Coverity compiler then parses the file and outputs a binary form of the source file into the intermediate directory, where it is read later by the analysis step. After the Coverity compiler finishes, the capture library continues to run the normal compiler that generates the `.o` files. You must run the actual compiler in addition to the Coverity compiler because many build processes build dependencies and build-related programs during the build itself. The disadvantage of this method is that it requires a longer compile time because each source file is parsed and compiled once with the regular compiler, and a second time by the Coverity compiler. But, the build system itself does not change, and no Coverity Analysis related changes need be maintained.

Figure 3.3.2. Coverity Analysis integration using the `cov-build` command

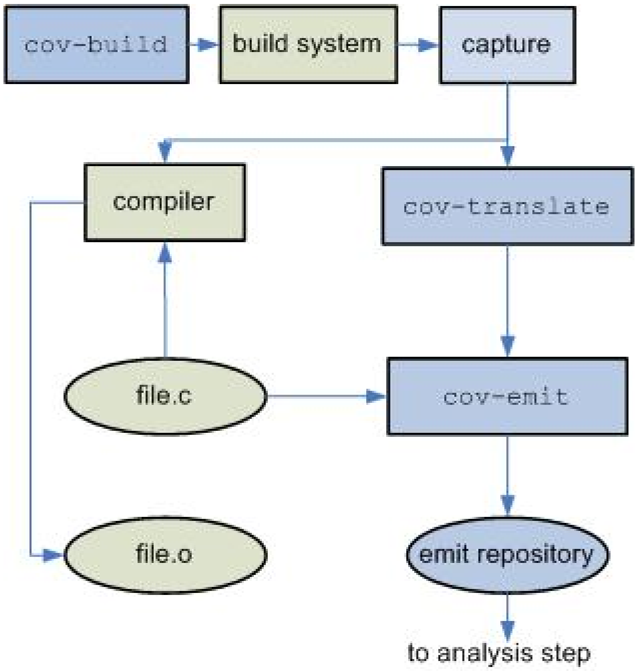
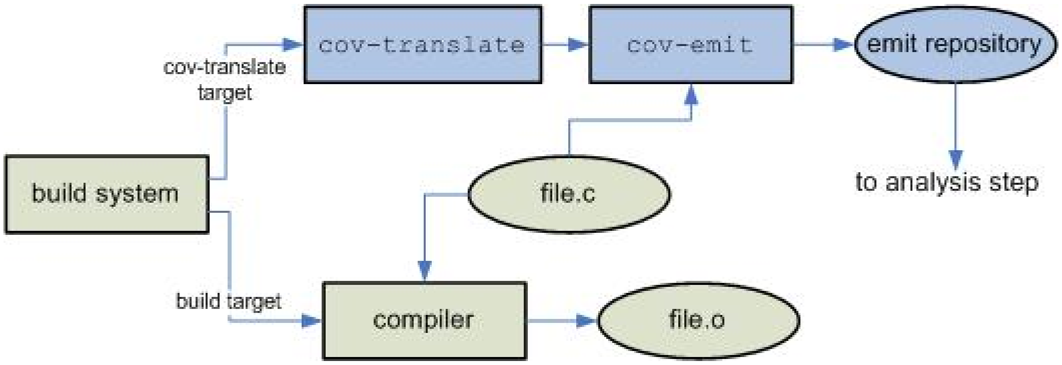


Figure 3.3.3, “Coverity Analysis integration by modifying build targets” shows an alternative Coverity Analysis integration method that relies on modifications to the build targets of the build system itself. Most build systems have the notion of a debug build target and a production build target. Similarly, another build target can be added to invoke the `cov-translate` command, or even the Coverity compiler directly (with the `cov-emit` command), to parse the code and generate the intermediate data. This method requires the build administrator to maintain the changes to ensure that they continue to work when the build steps change. The common `make` utility makes it possible to perform this form of integration by changing a single variable, such as `CC`. The Coverity Analysis translator can be configured to understand the command-line arguments for a variety of compilers, so the arguments to the compiler usually do not need to be changed. For more information about this integration method, see Section 3.3.3, “Alternative build command: cov-translate”.

Figure 3.3.3. Coverity Analysis integration by modifying build targets



The rest of this chapter describes how to use Coverity Analysis to perform these two types of integration.

3.3.1. The intermediate directory

The intermediate directory stores data produced by the Coverity compiler, before the data is committed to a Coverity Connect database.

Caution

The intermediate directory might use a significant amount of space for large code bases.

The intermediate directory cannot be under Rational ClearCase dynamic views.

On Windows, the intermediate directory cannot be on a network drive, neither as a mapped drive nor as a UNC path.

The intermediate directory is intended to be modified only by Coverity programs. Unless directed by Coverity support, do not create or remove files anywhere in the intermediate directory.

You cannot use a VMware shared folder as a location to store the intermediate directory.

3.3.2. Integrating Coverity Analysis into the build environment — `cov-build`

The `cov-build` command integrates Coverity Analysis with a build system, usually without any modifications to the build system itself. Using `cov-build` is the preferred method of build integration. Figure 3.3.2, “Coverity Analysis integration using the cov-build command” shows the basic process that `cov-build` uses to piggyback on a build system to produce the intermediate data. This intermediate data can then be analyzed to produce defect reports. For information about alternative build integration commands, see Section 3.3.3, “Alternative build command: cov-translate”.

After the `cov-config.xml` file is created, you can run the `cov-build` command by placing it in front of your usual build command. The required `--dir` option specifies the intermediate directory.

If the build command depends on features of the command shell that usually invoke it, such as certain shell variables or non-alphanumeric arguments, invoke the build command with a wrapper script. This method preserves the original behavior, since the build command is directly invoked by the type of shell on which it depends.

For example, if the normal invocation of a Windows build is:

```
> build.bat Release"C:\Release Build Path\"
```

use:

```
> cov-build --dir <intermediate_directory> <wrapper.bat>
```

where `<wrapper.bat>` is an executable command script that contains the original and unmodified build command.

On Windows systems, specify both the file name and extension for the build command when using `cov-build`.

For example:

```
> cov-build --dir <intermediate_directory> custombuild.cmd
```

Because `cov-build` uses the native Windows API to launch the build command, the appropriate interpreter must be specified with any script that is not directly executable by the operating system. For example, if the normal invocation of a build within Msys or Cygwin is:

```
> build.sh
```

prefix it with the name of the shell:

```
> cov-build --dir <intermediate_directory> sh build.sh
```

Similarly, if a Windows command file does not have Read and Execute permissions, invoke it as:

```
> cov-build --dir <intermediate_directory> cmd /c build.bat
```

The time that it takes to complete a build increases when you use `cov-build` because after the normal build runs, the Coverity compiler parses the same files again to produce the intermediate data. Consider the following factors that can increase build times with `cov-build`:

- The intermediate data directory is on a network mounted drive. Coverity Analysis creates many files and subdirectories in the intermediate directory, and these operations can be slow on network file systems. Using an intermediate directory on a local disk can eliminate this bottleneck. On Windows, you must use a local drive for the intermediate directory (Windows shared network drives are not supported for the intermediate directory).

- `cov-emit` does not take advantage of pre-compiled headers.

If the speed of `cov-build` is prohibitively slow when compared with your normal build time, one possible solution is to use more processes to parallelize the build. To see how to do so without altering your build scripts, see the section describing record/replay.

3.3.2.1. The output of `cov-build`: the `build-log.txt` log file

The `cov-build` command generates the log file in `<intermediate_directory>/build-log.txt` that contains a line for every command executed by the build process. The contents of `build-log.txt` are similar to:

```
EXECUTING 'make all '
EXECUTING '/bin/sh -c cd qmake && make '
EXECUTING 'make '
CWD = /export/home/acc/test-packages/qt-x11-free-3.3.2/qmake
COMPILING '/export/home/acc/prevent/bin/cov-translate g++ -c -o property.o \
-I. -Igenerators -Igenerators/unix \
-Igenerators/win32 -Igenerators/mac -I/export/home/acc/test-packages/qt-x11 \
-free-3.3.2/include/qmake \
-I/export/home/acc/test-packages/qt-x11-free-3.3.2/include \
-I/export/home/acc/test-packages/qt-x11-free-3.3.2/include \
-DQT_NO_TEXTCODEC -DQT_NO_UNICODETABLES -DQT_NO_COMPONENT \
-DQT_NO_STL -DQT_NO_COMPRESS \
-I/export/home/acc/test-packages/qt-x11-free-3.3.2/mkspecs/solaris-g++ \
-DHAVE_QCONFIG_CPP property.cpp ' \
 /export/home/acc/prevent/bin/cov-emit --g++ -I. \
-Igenerators -Igenerators/unix \
 -Igenerators/win32 -Igenerators/mac \
-I/export/home/acc/test-packages/qt-x11-free-3.3.2/include/qmake \
-I/export/home/acc/test-packages/qt-x11 \
-free-3.3.2/include -I/export/home/acc/test-packages/qt-x11-free-3.3.2/ \
include \
-DQT_NO_TEXTCODEC -DQT_NO_UNICODETABLES -DQT_NO_COMPONENT \
-DQT_NO_STL -DQT_NO_COMPRESS \
-I/export/home/acc/test-packages/qt-x11-free-3.3.2/mkspecs/solaris-g++ \
-DHAVE_QCONFIG_CPP \
--emit=/export/home/acc/prevent/emit -w \
--preinclude /export/home/acc/prevent/config/nodefs.h \
--preinclude /export/home/acc/prevent/config/solaris-x86/nodefs-g++.h \
--sys_include /usr/local/include/c++/3.3.2 \
--sys_include /usr/local/include/c++/3.3.2/i386-pc-solaris2.9 \
--sys_include /usr/local/include/c++/3.3.2/backward \
--sys_include /usr/local/include \
--sys_include /usr/local/lib/gcc-lib/i386-pc-solaris2.9/3.3.2/include \
--sys_include /usr/include property.cpp \
Emit for file '/export/home/acc/test-packages/qt-x11 \
-free-3.3.2/qmake/property.cpp' complete.
Emit for file '/export/home/acc/test-packages/qt-x11-free-3.3.2 \
/src/tools/qsettings.h' complete.
EXECUTING '/usr/local/lib/gcc-lib/i386-pc-solaris2.9/3.3.2/cc1 \
plus -quiet -I. \
-Igenerators -Igenerators/unix \
-Igenerators/win32 -Igenerators/mac \
-I/export/home/acc/test-packages/qt-x11-free-3.3.2/include/qmake \
-I/export/home/acc/test-packages/qt-x11-free-3.3.2/include \
-I/export/home/acc/ \
test-packages/qt-x11 \
-free-3.3.2/include -I/export/home/acc/test-packages/qt-x11-free-3.3.2/ \
mkspecs/solaris-g++ \
-D__GNUC__=3 -D__GNUC_MINOR__=3 -D__GNUC_PATCHLEVEL__=2 \
-DQT_NO_TEXTCODEC \
-DQT_NO_UNICODETABLES -DQT_NO_COMPONENT \
-DQT_NO_STL -DQT_NO_COMPRESS \
-DHAVE_QCONFIG_CPP \
property.cpp -D__GNUG__=3 -quiet -dumpbase property.cpp \
-auxbase-strip property.o \
-o /var/tmp//cc2Wo7sG.s '
EXECUTING '/usr/ccs/bin/as -Qy -s -o property.o /var/tmp//cc2Wo7sG.s '
```

The lines beginning with EXECUTING are commands that are executed by your build system but do not have any relation to compiling source code. For example, the commands executed by the build system to recursively descend into subdirectories in the source tree should show up as EXECUTING. When a compile line is encountered, three lines are printed. The first line begins with CWD, and shows the current working directory for the subsequent compile lines. The subsequent lines beginning with COMPILING are lines that are recognized as compiler invocations. The `cov-translate` program is called with the compiler command line arguments. The `cov-translate` command reads `.xml` and transforms the command line into the following line, which invokes the Coverity front-end program (`cov-emit`) to parse and emit the source file. The command line arguments to `cov-emit` are described in Chapter 2.7, *Configuring compilers for Coverity Analysis*.

For each source file that contains at least one function, the Coverity compiler prints a message "`Emit for file '/path/to/file.c' complete.`" The presence of this message confirms that the file exists in the intermediate directory and will be analyzed in the analysis step. The compiler can decide to skip emitting a file if it decides that it cannot have changed since the last emit. This will only happen if the timestamp for the file and all of the files included by it are the same as the previous emit.

If `cov-emit` produces error messages, it might be because of a misconfiguration or parsing compatibility issue. For more information on how to resolve compilation issues, see Chapter 2.7, *Configuring compilers for Coverity Analysis*. After `cov-emit` completes the emit, the compiler for the regular build runs. This results in additional EXECUTING lines for the compiler proper (`cc1plus` in the example) and the assembler (`as` in the example).

3.3.2.2. Building non-ASCII source code

Coverity Analysis supports non-ASCII encoding of source files. To use the `cov-build` command for non-ASCII-encoded source code, add the `--encoding <enc>` option with the appropriate encoding name. This option enables the following support:

- Appropriate display of the Unicode source code in Coverity Connect.
- Improved parsing of the source code, and reducing parse errors and warnings.

For example, the following command specifies that the source code is in Japanese:

```
cov-build --dir <intermediate_directory> --encoding Shift_JIS make my_build
```

The `--encoding <enc>` option is also available for the `cov-translate` and `cov-emit` commands.

3.3.2.3. Detecting parse warnings, parse errors, and build failures

Different incompatibilities can occur between differing dialects of C and especially C++, which result in parse errors and the `cov-build` command compiling less than all of the source code. You do not need all of the source code compiled to analyze the code for defects. However, the `cov-analyze` command analyzes only the files that `cov-build` was able to compile successfully.

The `cov-build` command, by default, is considered to be successful if it compiles 95% or more of the compilation units. You can change this percentage with the `--parse-error-threshold` option. For example, if you want `cov-build` to return without a warning only if 100% of the code compiles, add the following option to the `cov-build` command:

```
cov-build --dir <intermediate_directory> --parse-error-threshold 100
```

The more compilation units that you can compile without parse errors, the more code you can analyze. To improve the analysis, you can fix or work around many or all of the parse errors.

Sometimes the compiler can recover from a parse error. When the compiler recovers from an error, the compilation unit is compiled successfully, but the function that has the parse error cannot be analyzed. You can see these warnings (as RW.* checkers) in the Coverity Connect when you use the `--enable-parse-warnings` option to the `cov-analyze` command. To see cases when the compiler could not recover from errors, you should also specify the `--enable PARSE_ERROR` option to `cov-analyze`.

A variety of problems found by the Coverity compiler are called parse warnings, which you can see in the Coverity Connect (as PW.* checkers) when parse warnings are enabled. Parse warnings can show simple problems in the code, or can be signs of deeper defects. You can change which parse warnings are exposed as defects by creating a configuration file. A sample file is provided at `<install_dir>/config/parse_warnings.conf.sample`. For more information, see *Coverity 2020.12 Checker Reference*.

If the compiler finds non-standard code, and it can infer what is intended by that code, the compiler generates a semantic warning, which you an see in the Coverity Connect (as SW.* checkers) when parse warnings are enabled.

The `cov-build` command returns a non-zero exit code when either there is a fatal error while attempting to initialize the `cov-build` state before launching the command, or when there is a non-zero exit code from the build command specified on the command line. In the case that there are build failures due to incompatibilities between the Coverity Analysis compiler and the source code being analyzed, if the error does not cause the native compiler to fail and the build to exit, `cov-build` will not exit with a non-zero status code. You can change this behavior by using the option `--return-emit-failures`.

For details about how to handle and resolve parsing incompatibilities, see Section 2.7.3, “Compiler-specific configurations”.

3.3.2.3.1. Viewing parse errors

You can see parse errors in the `build-log.txt` log file, and through Coverity Connect.

The `build-log.txt` file is the log of the `cov-build` command. It is in `<intermediate_directory>/build-log.txt`. The `build-log.txt` file contains other error messages in addition to parse errors, so finding the parse errors can be difficult.

To view parse errors in Coverity Connect:

1. Run the `cov-build` (or `cov-translate`) command.

2. Run the `cov-analyze` command with the `--enable PARSE_ERROR` option to include parse errors in the analysis.

3. Commit the defects to the Coverity Connect database with the `cov-commit-defects` command.

4. Log in to Coverity Connect and look for defects named PARSE_ERROR.

   You can view these errors in the source code that caused the error, and the specific error message.

If the compiler is able to recover from a parse error, it is identified as a recovery warning, not a parse error. Recovery warnings have the prefix RW. For more information, see Recovery Warnings in the *Coverity 2020.12 Checker Reference*.

3.3.2.3.2. Preprocessing source files

The first step in debugging many parsing problems is to run the source file through the preprocessor to expand macros and include files. This process reveals the text of the entire translation unit that the compiler actually sees.

The `cov-preprocess` command can automatically preprocess an already emitted file. The syntax is:

```
> cov-preprocess [--diff] <file_to_preprocess>
```

The name of the file to preprocess can be a full path or just a file name. If you only specify a file name, the command looks for it in the intermediate directory, and preprocesses it if it is unique. Otherwise, it outputs a list of possible candidates. If the file name is an absolute path, the command will only preprocess the given file if it exists. This can be much faster when there is a large amount of intermediate data. The resulting preprocessed file is stored in:

`<intermediate_directory>/output/preprocessed/file.i`

If you use the `--diff` option, the program tries to preprocess the file with the compiler originally used to compile it, by adding `-E` to the command line. After, it will try to identify if the files differ, and notably if the order in which files are included is different.

If the preprocessing program does not work for you, you can also manually preprocess a source file by looking in `build-log.txt` for the invocation of `cov-emit` for the file of interest. Above this line is a line that includes `CWD=<dir>` which is the directory to change into when running the preprocessing command. Take the `cov-emit` command line for the file and remove the `--emit <dir>` option. Next, add the `-E` option before the source-file name; leave the source-file name as the last argument to `cov-emit`. Run the command, with a redirect to a file that is to contain the preprocessed output:

```
> cd src_dir
> cov-emit <args...> -E file.c > file.i
```

Inspect the output file `file.i` to see if the location where the parse error occurs appears to be different from the original source file.

3.3.2.3.3. Building with preprocessing first

Sometimes differences in preprocessed files are very difficult to diagnose or to solve. In this case, it is possible to tell `cov-build` to preprocess files with the native compiler and use these preprocessed files to emit code.

To do so, you can either run the cov-build command with the --preprocess-first option, or edit your .xml to add a <preprocess_first>yes</preprocess_first> tag in the <coverity><config><prevent> section. If the section does not exist, create it.

3.3.2.3.4. Testing hypotheses

It is often useful to perform small experiments to determine the root cause of parse errors. For example, copy the original source file into a temporary file and add the identifiers to macros whose value you wish to test at the end of the temporary file. Next, preprocess the temporary file and look at the expansion of the macros.

Another useful method is to reduce a preprocessed source file while preserving the parse error. If a small enough example can be generated this way, it might be possible to send Coverity an example that exhibits the problem. This greatly increases the chances that Coverity is able to find a workaround for the problem in a timely manner.

3.3.2.3.5. Re-running failed compiles without re-running the build

When a compile failure occurs, it would be useful to re-run the Coverity compiler over just the file or files that failed without incurring the overhead of re-running the entire build. The build might not work fast incrementally, or there might be additional overhead to launching a complete build. As an alternative, the --replay-failures option to cov-build uses information that is cached in the intermediate directory from each failed compile to re-run the Coverity compiler on just those files that failed to compile. If compilation failures are fixed, subsequent runs of cov-build --dir <intermediate_directory> --replay-failures recognize that a previously failed compile is now fixed and the subsequent runs do not attempt to re-compile the (now-fixed) compilation failure again.

Each time that cov-build --replay-failures finds a record of a compile failure in the intermediate directory, it reads both the command line used to invoke the native compile of that file and the full environment that was set when the native compile was attempted. After restoring this environment, it re-invokes cov-translate <native_cmd>, where <native_cmd> is the original compile command used in the build. The benefit of re-invoking cov-translate rather than calling cov-emit directly is that you can test both configuration changes to the .xml files and patches to cov-translate supplied by Coverity without re-running the build. These changes are applied when the compile failures are replayed. There are some cases where you might not want to re-translate with cov-translate. To avoid this step and have cov-build --replay-failures invoke cov-emit directly, specify the --no-refilter option to cov-build.

To summarize the different options for replaying compile failures:

```
> cov-build --dir <intermediate_directory> --replay-failures
```

finds all compile failures and re-applies cov-translate to the compile command used in your build. The build-time environment is restored before the re-translate is run.

```
> cov-build --dir <intermediate_directory> --replay-failures --no-refilter
```

finds all compile failures and re-applies cov-emit to the translated argument list. This option runs faster than without --no-refilter, but it does not allow you to verify fixes to the Coverity configuration files and it does not allow you to verify cov-translate patches supplied by Coverity.

3.3.2.4. Linkage information

For C and C++ source code, the same file is sometimes compiled several times with different command-line options. Due to the inherent difficulty of tracking linkage information, the `cov-analyze` command cannot automatically determine which files are linked together. To avoid errors in function call resolution (especially in C code, which does not have any name mangling), you can use the `cov-link` command to get this information.

The following two examples cover common uses of this feature. For a complete list of command line options and additional examples, see the `cov-link` documentation.

3.3.2.4.1. Example 1

Assume that you have a single project <`productA`> with two target architectures, ARM and MIPS. Each function is compiled twice, possibly using different semantics. For example, arrays do not have the same size, and you have OVERRUN false positives. Assume that the target architecture is specified to the compiler by `-march=<arch>`. Use the following steps to resolve the duplicate function calls:

1. Run the build:

   ```
   > cov-build --dir productA_dir
   ```

2. Collect linkage information and save it to a link file::

   ```
   > cov-link --dir productA_dir -of all.link --collect
   ```

   Working with link files is faster than collecting data from the intermediate directory multiple times.

3. Create a link file for each target architecture, using the link file created in the previous step as input:

   ```
   > cov-link --dir productA_dir -a -march=arm -of arm.link all.link
   > cov-link --dir productA_dir -a -march=mips -of mips.link all.link
   ```

4. Create separate intermediate directories for ARM and MIPS by using the target architecture link files:

   ```
   > cov-link --dir productA_dir --output-dir arm-emit arm.link
   > cov-link --dir productA_dir --output-dir mips-emit mips.link
   ```

   Note that `cov-link` appends new files to an existing target intermediate directory repository, but never removes files. If you want to remove files, first delete the intermediate directory completely.

   Note

   To allow incremental analysis to work, keep the intermediate directory intact.

5. Run the analysis and commit the errors for the `arm-emit` and `mips-emit` repositories:

   ```
   > cov-analyze -dir arm-emit
   > cov-commit-defects --dir arm-emit --target "ARM target" \
     --stream productA-ARM
   > cov-analyze --dir  mips-emit
   ```

```
> cov-commit-defects --dir  mips-emit --target "MIPS target" \
  --stream productA-MIPS
```

This creates two separate runs with target attributes named `ASM target` and `MIPS target`. You can run the commands to analyze and commit the `arm-emit` and `mips-emit` data concurrently, if you specify a different intermediate directory for each. To do, use the `--dir` option to `cov-commit-defects`.

3.3.2.4.2. Example 2

In this example, assume that you have two projects named `proj1` and `proj2` that share the library `lib`. The two projects define the same functions with different semantics, so you need linkage information. Assuming that the source files are located in directories named `proj1`, `proj2`, and `lib`, use the following steps to resolve the duplicate function calls:

1. Run the build:

   ```
   > cov-build --dir proj1_proj2
   ```

2. Collect linkage information and save it to a link file:

   ```
   > cov-link --dir  proj1_proj2 -of all.link --collect
   ```

   Working with link files is faster than collecting data from an emit repository multiple times.

3. Generate a link file for the library, using the link file created in the previous step as input:

   ```
   > cov-link --dir proj1_proj2 -s /lib/ -of lib.link all.link
   ```

4. Because the projects share the same library, make two copies of the link file (`proj1.link` and `proj2.link`) for the library:

   ```
   > mv lib.link proj1.link
   > cp proj1.link proj2.link
   ```

5. Append the project linkage information to the project link files:

   ```
   > cov-link --dir proj1_proj2 -s /proj1/ -of proj1.link all.link
   > cov-link --dir proj1_proj2 -s /proj2/ -of proj2.link all.link
   ```

6. Create intermediate directories for `proj1` and `proj2` by using the project link files:

   ```
   > cov-link --dir proj1_proj2 --output-dir proj1-emit proj1.link
   > cov-link --dir proj1_proj2 --output-dir proj2-emit proj2.link
   ```

7. Run the analysis and commit the defects for `proj1` and `proj2` by using the project-specific intermediate directories that were created in the previous step:

   ```
   > cov-analyze --dir  proj1-emit
   > cov-commit-defects --dir  proj1-emit --description "proj1" \
     --stream proj1
   > cov-analyze --dir proj2-emit
   > cov-commit-defects --dir proj2-emit --description "proj2" \
   ```

```
  --stream proj2
```

3.3.2.5. Record/Replay - Deferred builds and parallelizing single process builds

3.3.2.5.1. Running cov-build with --record-only

Coverity Analysis has the ability (for C/C++ only) to record the environment, working directory, and command line for each file in the build, and replay all of those recorded commands either with a single process or multiple processes at a later time. The advantages of this approach are:

- If build-time is critical for the native build, you can allow the native build to complete with minimal overhead (~10%), and run the Coverity build at a later time when the machines are idle or the build timing is not as critical.
- If your build cannot be made parallel by default, using the record/replay mechanism allows you to at least parallelize the Coverity portion of the build if you have more than one processor on the build machine.

The required operations to record the environment, command line, and working directory are executed during each invocation of `cov-build`. If you want to run `cov-build` with *just* the record step, either specify the `--record-only` option of the `cov-build` command or the `cov-translate` command:

```
> cov-build --dir <intermediate_directory> --record-only <build command>
```

```
> cov-translate --record-only <compile command>
```

After a record-only build is complete, use the recorded information to run the Coverity compiler with the `--replay` option:

```
> cov-build --dir <intermediate_directory> --replay
```

The `--replay` functionality can also be run using multiple processes on a single machine. To specify more than one process on a single machine, use the `-j <process count>` option:

```
> cov-build --dir <intermediate_directory> --replay -j 4
```

This command line replays all of the recorded compilations using 4 processes. At the end of the replay step, all of the information from the 4 replay processes is aggregated into a single `replay-log.txt` file, which you can then use to discover and diagnose compilation failures.

Note

Only run one `cov-build --replay` command or `cov-build --replay-failures` command with a given `--dir <intermediate_directory>` option at any one time.

3.3.2.5.2. Running cov-build with --record-with-source

You can use the `--record-with-source` option to run `cov-build` through the record step, and also collect all of the necessary source files in the build (for C/C++ and Java only). Then you can then complete the `cov-build` run at a later time using the `replay-from-emit` option:

```
> cov-build --dir <intermediate_directory> --record-with-source <build command>
> cov-translate --record-with-source <compile command>
```

After a record-with-source build is complete, use the recorded information to run the Coverity compiler with the `--replay-from-emit` option:

```
> cov-build --dir <intermediate_directory> --replay-from-emit
```

This is helpful if you need the ability to complete the replay build on a different platform than you started from. For example, you could complete the `cov-build --record-with-source` step on a Windows machine, then transfer the emit file and complete the `cov-build --replay-from-emit` step on a Linux machine. The `--record-with-source` option is also beneficial for recording builds with transient files, such as `#import` files; `--record-only` fails when attempting to record these builds.

Note

Running `cov-build` with the `--record-with-source` option takes significantly longer than using `--record-only`.

Note

The recording of Java webapps needs to be done outside of the `cov-build --record-with-source` command. Refer to the `cov-record-source` command in the *Coverity 2020.12 Command Reference* for details.

3.3.2.6. Error handling with commands

In general, commands return a non-zero exit code whenever there is a catastrophic failure that prevents the command from proceeding. If a command appears to fail while still returning an exit code of zero, there are two possibilities: either the failure that appears to be reported did not prevent the command from continuing to run and is merely a warning, or the command is not behaving properly and you should contact `software-integrity-support@synopsys.com`.

3.3.2.7. Troubleshooting build problems

The `build-log.txt` file is generated but there are no COMPILING lines and no "Emit for file complete" messages.

Potential causes:

- The compiler is not configured properly in `coverity_config.xml`. Common problems include:
  - A syntax error in the `coverity_config.xml` file. It must be a valid XML file according to the DTD `<install_dir_sa>/dtd/coverity_config.dtd`. Look carefully at the initial output to the terminal when `cov-build` is invoked. Consider using an XML syntax or schema validator such as `xmllint` to make sure that the file is valid.
  - The configured path name of a compiler is empty or missing, in the `<comp_dir>` tag. This field should identify the actual path name for the configured compiler, although any executed compilation with the same command name is analyzed as if it were the configured version. If incompatible versions are in use, you can configure them with a template, or you can separately configure each pathname that is in use.

The build stops before all files have been compiled.

Potential causes:

- The native build is failing. The `cov-build` command relies on the native build to be able to complete the compile. The `cov-build` command cannot proceed beyond the native build.
  On many build systems, there is a way to keep compiling files even when an error occurs. For example, the `-i` flag to `make` forces `make` to ignore any errors during the build. Coverity Analysis does not require a 100% complete build to produce good results.

- The `cov-build` command could be interfering with the native build. Contact Coverity support for assistance.

Some or all files give compiler error messages in `build-log.txt`.

Potential causes:

- The compiler translator or options are not configured properly. If you manually modified or generated the `coverity_config.xml` file, reread Section 1.4.6.3, “Using Coverity Analysis configuration files in the analysis”. The most common problem is a mismatch between the predefined macros in `nodefs.h` and the predefined macros supplied by the build's compiler. Consider using the `cov-configure` command to generate a configuration file automatically. Make sure to specify the compiler version.

- Some of the macro suppressions in `nodefs.h` are causing parsing problems. Consider removing the offending predefine in `nodefs.h` if the offending nodef is not required. For C++, a prototype declaration might need to be added to `nodefs.h`.

- The pre-include directories are not set properly. The build compiler has a list of built-in directories to search to find include files. The `<include_dir>` and `<sysinclude_dir>` options in `coverity_config.xml` need to reflect these built-in search paths. Note that the `<include_dir>` has precedence over `<sysinclude_dir>`, and that and parsing might change in "system" headers. Both are searched whether "" or <> is used. The `cov-configure` command automatically finds these search paths for most compilers.

- The `cov-emit` command is not able to parse the source code. There are some non-standard, compiler-specific constructs that `cov-emit` might not be able to parse correctly. For a detailed discussion of the potential problems and solutions, see Chapter 2.7, *Configuring compilers for Coverity Analysis*.

I am using `clearmake` and the Coverity build only seems to compile a small subset of my source files.

Potential causes:

The `clean` command with `clearmake` generally does not cause a subsequent invocation to re-build all of the source files in the build with the compiler. The Coverity build system looks for invocations of the compiler to decide which source files to analyze, so any `clearmake` optimizations that circumvent actually running the compiler will interfere with the Coverity build. In particular, you must:

1. Delete all of the object files that correspond to the source files that you want to compile.

2. Turn off *winking* by specifying the appropriate option to `clearmake`.

3.3.2.8. Platform-specific cov-build issues

Some platforms have special issues that might interfere with the operation of `cov-build`.

3.3.2.8.1. Linux

No special issues.

3.3.2.8.2. Solaris

The `cov-build` command fails if the build command, such as `make`, is a `setuid` executable. To run the `cov-build` command, you can turn off the `setuid` bit with the following command:

```
> chmod u-s <path>/<build_command>
```

3.3.2.8.3. Windows

The `cov-build` command uses a different mechanism to capture compiler process creation on Windows than on UNIX platforms. The Windows version of `cov-build` runs the build command and its child processes in debug mode. If your build system has problems running in debug mode, try using the `--instrument` option with `cov-build`. This option might be useful is for capturing a 32-bit `javac` compilation on 64-bit Windows.

Some build systems on Windows are invoked from an integrated development environment (IDE) such as Visual Studio. There are several ways of integrating Coverity Analysis with an IDE:

- Invoke the IDE binary with the `cov-build` command wrapped around it. For Visual Studio 2005 and 2008, the IDE is typically invoked with the `devenv` command. For example:

  ```
  > cov-build --dir intermDir devenv
  ```

  After you run the command, perform the necessary actions in the IDE to perform the build and then exit the IDE. Because the `devenv` command runs the compiles, `cov-build` can capture the build.

- For Visual Studio 2010 and subsequent releases, the `devenv` command builds applications in a separate hosted instance of the `msbuild` tool.

  Analysis support for Visual Basic was introduced with Visual Studio 2013.

- Use the command line to perform the build.

  Example using `devenv`:

  ```
  > cov-build --dir intermDir devenv solutionfile /build solutionconfig
  ```

  Example using `msbuild`:

  ```
  > cov-build --dir intermDir msbuild solutionfile /p:Configuration=Release
  ```

- Use the Visual C++ compiler directly (`cl.exe`) within a makefile and then run `make` or `nmake` with the `cov-build` command. This is the same process you would use to build with a compiler, such as gcc, on UNIX systems.

3.3.2.8.4. FreeBSD

Many versions of FreeBSD have a statically linked sh and make. The cov-build command relies on intercepting exec() at the shared library level and cannot intercept compiler invocations from static build programs such as sh and make. The solution is to change the <comp_name> variable in the coverity_config.xml file to recognize cc1 as the compiler. This works because gcc is usually not statically linked, and gcc is a driver program that calls cc1 to actually perform the compile. Some features of cov-build, such as automatic preprocessing of files to diagnose compile errors, might not work in such case.

On FreeBSD 5.3 or later, Coverity Analysis can fail with a bad system call error and a core dump. This is because Coverity Analysis is compiled for FreeBSD 4.x. To use Coverity Analysis on FreeBSD 5.3 or later, compile the system kernel with the with COMPAT_FREEBSD4 set.

3.3.2.8.5. AIX

The cov-build and cov-analyze commands are not provided on AIX. Instead, you need to manually integrate the necessary cov-translate commands (see Section 3.3.3, “Alternative build command: cov-translate”) into your build system. For example:

```
> CC="cov-translate --dir int-dir --run-compile cc" make
```

After running cov-translate, you need to copy the resulting intermediate directory to a different (non-AIX) machine on which a compatible version of cov-analyze is installed and then run the following commands to complete the analysis:

- cov-manage-emit with its non-filtered sub-command reset-host-name
- cov-analyze

3.3.3. Alternative build command: cov-translate

The cov-translate command translates native compiler command-line arguments to arguments appropriate for the Coverity compiler, and then calls the compiler with the cov-emit command. If you use cov-build to integrate with the build, there is no need to deal explicitly with cov-translate. All of the options that control how cov-translate works are in the coverity_config.xml file. You can specify the intermediate directory, with an emit repository, on the cov-translate command line using the --dir option.

To perform manual integration with a build system, the build system needs to be modified to have an additional target that calls cov-translate instead of the usual compiler. For more information, see Figure 3.3.3, “Coverity Analysis integration by modifying build targets”.

3.3.3.1. The cov-translate command in place of the native compiler

If the default method of build integration using the Coverity build utility (cov-build) is unsuitable for any reason, you can use the cov-translate command as a replacement for any of the supported compilers. In this mode, cov-translate can be prepended to any compile line and, when supplied the appropriate arguments, can run both the native compile and the Coverity Analysis compile. For example, you need to follow this procedure to run the Coverity compiler on AIX, which does not support the cov-build command.

The --run-compile option to cov-translate indicates that it runs both the native compile and the Coverity Analysis compile. For example, the following command creates the object file test.o, and adds the analysis intermediate form for test.c to the emit repository:

```
> cov-translate --dir <intermediate_directory> --run-compile gcc -c test.c
```

For most build systems, it is sufficient to prepend the compiler name with the command sequence <install_dir_sa>/bin/cov-translate --run-compile command. For example, you can specify the following to run make with its CC/CXX macro defined as a cov-translate command that is configured to execute the appropriate native C/C++ compiler:

```
> CC="cov-translate --dir int-dir --run-compile cc" make
```

Manually integrating cov-translate into a Makefile becomes more complex when a build system includes scripts that rely on the exact format of the output to stdout from the compilation. For example, any build that invokes GNU autoconf configuration scripts during the build requires that the compilations invoked within the autoconf scripts mirror the output of the native gcc compiler invocations exactly. To address this issue, Coverity Analysis provides an argument translator, the --emulate-string option to the cov-translate command. This option is used to specify a regular express that, if matched on the command line, makes the command to run the native compiler command line only (that is, without attempting to call cov-emit). The output from the native compiler invocation is printed verbatim to stdout, and cov-translate does not make any attempt to run the Coverity compiler.

The regular expressions to the --emulate-string option are Perl regular expressions. For example, to indicate that any option to gcc containing the word dump should cause the emulation behavior, the cov-translate command line can be specified as follows:

```
> cov-translate --dir <intermediate_directory> --run-compile --emulate-string ^-dump.*
 gcc -dumpspecs
```

This command causes the verbatim output of gcc -dumpspecs to be printed to stdout. Note that the ^ and $ elements of the Perl regular expression are implicitly added to the beginning and end of the specified regular expression when they are not present. This addition means that the terminating .* at the end of the option in the above example is required to ensure that any sequence of characters can follow -dump.

For gcc in particular, the following arguments should be emulated using the emulate-string option because they are commonly used by the GNU autoconf-generated configure scripts:

- -dumpspecs
- -dumpversion
- -dumpmachine
- -print-search-dirs
- -print-libgcc-file-name

- `-print-file-name=.*`
- `-print-prog-name=.*`
- `-print-multi-directory`
- `-print-multi-lib`
- `-E`

3.3.4. Running parallel builds

Coverity Analysis for C/C++ supports multiple parallel build scenarios to provide integration with a native build system with minimal or no system modifications. Because of I/O and synchronization costs, parallel builds might not take place more quickly than the builds described in Section 3.3.2.5, “Record/Replay - Deferred builds and parallelizing single process builds”.

3.3.4.1. Single build on a single machine

The `cov-build` command can capture parallel builds. Examples of commonly seen parallel build commands would be `make -j` or `xcodebuild -jobs`. One problem with parallel builds is that the `build-log.txt` log file contains interleaved output, which might make it difficult to determine if a given source file has been parsed and output to the intermediate directory. In such case, the intermediate directory is still created without problems.

3.3.4.2. Multiple builds on a single machine

A build on a single host can use a single build command to create multiple, concurrent compilation processes. There are several ways to capture information for build and C/C++ analyses.

To capture information for a build, C/C++ analysis, or both, you can run a single `cov-build` command with a `make -j` or similar command.

To capture information for a C/C++ analysis, you can use multiple `cov-build` commands sequentially:

```
cov-build --capture ... make [-j N] ...
cov-build --capture ... make [-j N] ...
cov-build --capture ... make [-j N] ...
```

To capture information for a C/C++ analysis, you can explicitly call `cov-translate` from the build system:

```
make [-j N] CC="cov-translate ..." ...
```

If all `cov-translate` processes are concurrently running on the same machine, Coverity recommends using a single intermediate directory. If `cov-translate` processes run on different machines, then use multiple local intermediate directories and merge them using `cov-manage-emit` after the build is finished. Running `cov-translate` in parallel on NFS is not recommended.

If you use multiple `cov-build` commands sequentially, the `--capture` flag is not needed.

3.3.4.3. Multiple builds on multiple machines

Because the `cov-build` command relies on capturing calls to `exec()`, distributed builds that use remote procedure calls or other network communication to invoke builds are not detected. Distributed builds can be handled by modifying the build system to add an additional Coverity Analysis target that uses the `cov-translate` program. For more information, see Section 3.3.3, “Alternative build command: cov-translate”.

Distributed builds using a common intermediate directory on an NFS partition that is shared by all contributing servers are supported on Linux and Solaris systems that have the same Coverity Analysis distribution, version, and compiler configuration.

Note

> The `cov-emit` command can either run by itself, or be invoked indirectly by `cov-build` or `cov-translate`. You cannot directly or indirectly run `cov-emit` on one platform and `cov-analyze` on another platform.

Build systems can explicitly call `cov-translate` in the following ways:

- Multiple build commands run on multiple machines, which each locally run `cov-translate`.
- A single `make` or similar command distributes individual compilations to multiple configured servers via `ssh` or another remote job execution service.

3.3.4.3.1. Sharing a common intermediate directory on an NFS partition

To distribute a build:

1. Run `cov-build` once without a build command to initialize the intermediate directory:

   ```
   cov-build --dir <intermediate_directory> --initialize
   ```

2. Run one or more `cov-build`, `make`, or equivalent command per host machine:

   ```
   cov-build --dir <intermediate_directory> --capture make [-j N]
   make [-j N] CC="cov-translate ..."
   ```

   The `--capture` option ensures that `cov-build` log and metric files are merged and not replaced.

3. Combine the log and metrics files from all contributing hosts, and identify any commands that need to be run on the machine that is used for subsequent analyses:

   ```
   cov-build --dir <intermediate_directory> --finalize
   cov-manage-emit --dir <intermediate_directory> add-other-hosts
   ```

   After the build is finalized, and the indicated commands run, the `<intermediate_directory>` is ready for analysis. The `cov-manage-emit` command must run after a distributed build to aggregate the data captured on other hosts, and on the host machine that will run the `cov-analyze` command.

3.3.4.3.2. Copying intermediate directories from local disks

To distribute a build:

1. Run `cov-build` once on each server to initialize an intermediate directory on a local disk used only by that build server:

   ```
   cov-build --dir <intermediate_directory> --initialize
   ```

2. Run one or more `cov-build`, `make`, or equivalent command per build server:

   ```
   cov-build --dir <intermediate_directory> --capture make [-j N]
   make [-j N] CC="cov-translate ..."
   ```

   The `--capture` option ensures that `cov-build` log and metric files are merged and not replaced.

3. Complete the build(s) on each build server:

   ```
   cov-build --dir <intermediate_directory> --finalize
   ```

4. Copy the complete intermediate directory tree from each build server to a local disk on the machine on which you will run `cov-analyze`.

   For example:

   - Use a remote-copy utility such as `scp -r`.

   - Use an NFS partition or network file share.

5. Merge the intermediate directory that was copied from each build server with the intermediate directory that you want to analyze:

   ```
   cov-manage-emit --dir <copied_directory> reset-host-name
   ```

   ```
   cov-manage-emit --dir <intermediate_directory> add <copied-directory>
   ```

   After the build finalizes, and the indicated commands run, the `<intermediate_directory>` is ready for analysis.

Chapter 3.4. Using SSL with Coverity Analysis

Table of Contents

3.4.1. Trust store overview

This section describes the trust store, a storage location for certificates used by `cov-commit-defects` and other Coverity Analysis applications that connect using SSL. This trust store is specific to the Coverity Analysis client; for information on the server-side trust store, see the *Coverity Platform 2020.12 User and Administrator Guide* ↗.

Note

This trust store is not the same as the one used by Java-based command line tools (`cov-manage-im`, `cov-integrity-report`, and `cov-security-report`).

The discussion assumes a basic level of familiarity with SSL. Comprehensive information on SSL, can be found at http://en.wikipedia.org/wiki/Transport_Layer_Security ↗.

When connecting to a network peer (such as a Coverity Connect server, in the case of `cov-commit-defects`), the SSL protocol must authenticate the peer, that is, it must prove that the peer has the identity that it claims to have. The authentication step uses a digital certificate to identify the peer. To authenticate, the application must find a digital certificate of a host that it trusts; that certificate must vouch for the veracity of the peer's certificate. Any number of certificates may be used to form a chain of trust between the peer's certificate and a certificate trusted by the application. If the application is successful in finding such a chain of trust, it can then treat the peer as trusted and proceed with the data exchange.

Coverity Analysis uses the trust store as the location for storing trusted certificates. When initially installed the trust store directory (`<install_dir_sa>/certs`) contains one file, `ca-certs.pem`, which contains a collection of certificates published by certificate authorities such as Verisign. (Coverity gets this list from the corresponding list, `cacerts`, in the Java Runtime Environment.)

There are two trust modes for certificates in Coverity Analysis.

- **fully authenticated** mode - The application accepts a chain of trust only if it ends in a certificate in `ca-certs.pem`.
- **trust-first-time** mode - The application uses a weaker standard, where it accepts a certificate as trusted if either of the following is true:
  - The same peer has sent the same certificate in the past.
  - The certificate is self-signed (that is, the certificate's next link in the chain of trust is itself) and Coverity does not already have a certificate stored for that host/port combination.

In other words, when the application receives a self-signed certificate it has not encountered before from that peer and port, it stores the certificate in the trust store in its own file. Subsequent connections to the same peer and port verify that the peer's certificate matches the certificate in the file.

Both trust modes result in an encrypted connection. The difference between them is that connections secured using trust-first-time mode do not have the same level of assurance of the identity of the peer. Specifically, the first time you use a certificate in trust-first-time mode, you need to take a leap of faith that the peer your application contacted is not being impersonated by another peer.

Both trust modes are provided because there is an administrative cost involved in setting up fully authenticated mode: the administrator must get the server's certificate from a certificate authority and install it in the server. If the certificate authority's root certificate is not included in `ca-certs.pem`, then the administrator must also add it to that file on every client. See the *Coverity Platform 2020.12 User and Administrator Guide* for additional details. In contrast, trust-first-time mode requires no administrative work to allow the application to encrypt its communications with the peer.

See the `--authenticate-ssl` option to `cov-commit-defects` for more discussion of the difference between these trust modes.

3.4.2. Configuring Coverity Analysis to use SSL

This procedure allows you to use SSL with commands that send data to Coverity Connect, such as `cov-commit-defects`, `cov-run-desktop`, and `cov-manage-history`. Note that it discusses authentication modes described in Section 3.4.1, "Trust store overview".

1. Make sure that Coverity Connect is configured to use SSL.

   For the setup procedure, see "Configuring Coverity Connect to use SSL" in *Coverity Platform 2020.12 User and Administrator Guide*

2. Verify browser access to Coverity Connect over HTTPS.

   Simply type the Coverity Connect URL, including the HTTPS port number into your browser, for example:

   ```
   https://connect.example.com:8443/
   ```

3. If necessary, install a certificate on each client, using one of the following modes:

   - The fully authenticated mode: If your certification authority certificate is in `ca-certs.pem` (which is typical if you paid an external certification authority entity, such as Verisign, for your certificate), no action is needed. Otherwise, follow the instructions in Section 3.4.3.4, "Adding a certificate to ca-certs.pem".

   - The trust-first-time mode: If you use the Coverity Connect self-signed certificate that was installed with Coverity Connect and you commit using trust-first-time, no action is needed.

4. Use `cov-commit-defects` to test a commit using SSL.

5. Inspect the new certificate, if any, in the trust store.

For details on viewing certificates, see Section 3.4.3, “Working with the trust store”.

3.4.3. Working with the trust store

The trust store is implemented as a directory: `<install-dir>/certs`. There are two kinds of files in the trust store. The first is the collection of certificate authority certificates mentioned above, `ca-certs.pem`. Secondly, there may be single-certificate files with names like `host-<host-name>,port-<port-number>.der`. These files store trust-first-time certificates. The file name tells which host and port the certificate was seen on.

3.4.3.1. Viewing trust-first-time certificates

Trust-first-time certificates are stored in DER format. They can be read using the `openssl` command, present on most linux systems, or using the `keytool` command, present in the Java Runtime Environment at `<install-dir>/jre/bin/keytool`. For example,

```
openssl x509 -in host-d-linux64-07,port-9090.der -inform der -noout -text
```

or

```
keytool -printcert -file host-d-linux64-07,port-9090.der -v
```

3.4.3.2. Viewing certificate authority certificates

The certificate-authority certificates in `ca-certs.pem` are stored in PEM format, which encodes the certificates as ASCII text. The file is a simple list of certificates. An example certificate is shown below:

```
-----BEGIN CERTIFICATE-----
MIIDGzCCAoSgAwIBAgIJAPWdpLX3StEzMA0GCSqGSIb3DQEBBQUAMGcxCzAJBgNV
BAYTAlVTMRAwDgYDVQQIEwdVbmtub3duMQ8wDQYDVQQHEwZVbmtvd24xEDAOBgNV
BAoTB1Vua25vd24xEDAOBgNVBAsTB1Vua25vd24xETAPBgNVBAMTCFFBVGVzdENB
MCAXDTEzMDIyNTIyMTA1MloYDzIxMTMwMjAxMjIxMDUyWjBnMQswCQYDVQQGEwJV
UzEQMA4GA1UECBMHVW5rbm93bjEPMA0GA1UEBxMGVW5rb3duMRAwDgYDVQQKEwdV
bmtub3duMRAwDgYDVQQLEwdVbmtub3duMREwDwYDVQQDEwhRQVRlc3RDQTCBnzAN
BgkqhkiG9w0BAQEFAAOBjQAwgYkCgYEA196ZPKzj6LKVrR9iZeDrqmrv25Zv3+9/
itiRN6xbJW0FvU/cIz2zoZxTIvlCFInC6qZ0BQcNJRsYmtJQsr/ka6MFuneULh3g
cYNxDTBRCJ2Lbs5xDjYMfEg6XJSwyBo/iG3fxb6IBdiAnjPdUFT5THkNheUhh62f
rISUU9zwAWcCAwEAAaOBzDCByTAdBgNVHQ4EFgQUn3hosvIlr4Md80enOS/kC/p3
JL4wgZkGA1UdIwSBkTCBjoAUn3hosvIlr4Md80enOS/kC/p3JL6ha6RpMGcxCzAJ
BgNVBAYTAlVTMRAwDgYDVQQIEwdVbmtub3duMQ8wDQYDVQQHEwZVbmtvd24xEDAO
BgNVBAoTB1Vua25vd24xEDAOBgNVBAsTB1Vua25vd24xETAPBgNVBAMTCFFBVGVz
dENBggkA9Z2ktfdK0TMwDAYDVR0TBAUwAwEB/zANBgkqhkiG9w0BAQUFAAOBgQAY
97hV0EM2uMg/kg2bUllyDtCnQLFdbv/NJ5b+SlHyAQAhaTchM7WBW7OVY4fTS9xZ
Uh8k7uvKicBAd48kdkU6K4LF3SowwjWdOmyGvOnyUHSvCCfa/+G/rPzMReIVQo2H
HIUtgMWvzOtZh6nYLV4JDbQcYJ0d7eBcvebetFAxyA==
-----END CERTIFICATE-----
```

To view these certificates you need to split them into separate files, with one certificate per file. Then the commands to read them are

```
openssl x509 -in <certificate-file-name> -noout -text
```

or

```
keytool -printcert -file <certificate-file-name> -v
```

3.4.3.3. Interpreting a certificate file

You typically will not need to interpret an individual certificate file, but a sample certificate, as dumped by keytool, is displayed below. Descriptions of the individual elements follow.

```
Owner: C=None, L=None, O=None, OU=None, CN=d-linux64-07
Issuer: C=None, L=None, O=None, OU=None, CN=d-linux64-07
Serial number: 555b70a6
Valid from: Fri Dec 20 16:21:15 PST 2013 until: Tue Dec 20 16:51:15 PST 2033
Certificate fingerprints:
       MD5:  78:0D:07:53:3E:BF:A2:76:B1:C2:9E:2C:86:A6:2C:5B
       SHA1: AD:66:3E:5C:40:FC:49:84:F6:21:3E:B2:37:9A:32:25:B2:33:38:4D
       Signature algorithm name: SHA256withRSA
       Version: 3
```

The `Owner` string identifies the peer. In particular the `CN` portion of the owner field contains the host name of the peer. In SSL, the other fields of the owner string are ignored. The `Issuer` string identifies the entity that created the certificate. In this case, the issuer matches the owner, which means the certificate is self-signed. The `Valid from` and `until` fields show the dates on which the certificate will pass into and out of validity. The `fingerprints` are `MD5` and `SHA1` hashes of the DER form of the certificate.

3.4.3.4. Adding a certificate to `ca-certs.pem`

You may want to add a certificate to `ca-certs.pem` if you want to tell the application (for example, `cov-commit-defects`) that a certain certificate is trusted as a certificate authority certificate. This is necessary if you want to use the fully authenticated mode, but your certificate authority is not among those listed in `ca-certs.pem`. This will be the case if you use an internal certificate authority. To add it, there are two steps. First, if the certificate is not already in PEM format, use `openssl` to convert it to PEM format. For example, for a certificate in DER format, the `openssl` command is

```
openssl x509 -in <certificate-file-name> -inform der -outform PEM > cert.pem
```

Alternatively, to do this using `keytool`, you first have to import the certificate into a temporary keystore, then export it as a PEM file:

```
keytool -keystore new.jks -storepass newnew -importcert -alias
new -file <certificate-file-name>
keytool -keystore new.jks -storepass newnew -exportcert -alias
new -file cert.pem -rfc
```

After getting your certificate as a PEM file, prepend it to the front of your `ca-certs.pem` file, or, if you are not using an external certification authority, simply replace `ca-certs.pem` with your certificate in PEM format.

On Linux:

```
> cat cert.pem ca-certs.pem > new-ca-certs.pem
> mv new-ca-certs.pem ca-certs.pem
```

On Windows:

```
> type cert.pem ca-certs.pem > new-ca-certs.pem
> del ca-certs.pem
> ren new-ca-certs.pem ca-certs.pem
```

3.4.3.5. Removing a trust-first-time certificate from the trust store

To stop trusting a trust-first-time certificate, delete its file from the trust store.

3.4.3.6. Removing certificates from `ca-certs.pem`

To stop trusting a certificate authority certificate in `ca-certs.pem`, complete the following steps:

1. Split `ca-certs.pem` into separate certificates, as indicated in Section 3.4.3.2, “Viewing certificate authority certificates”.
2. Rename `ca-certs.pem` to `old-ca-certs.pem`.
3. Use `openssl` or `keytool` on each certificate to find the ones you want to include in the new `ca-certs.pem`.
4. Concatenate the certificates you want to include, and write the result to a new `ca-certs.pem` file.
5. Test with `cov-commit-defects`.

Chapter 3.5. Using a Network File System (NFS) with Coverity Analysis

NFS is supported for use with Coverity Analysis in many cases. Support is the same for all Coverity commands.

For all operating systems:

- Source code, native compilers, and native build system files (for example, Makefiles) may reside on NFS.
- User and derived model files may reside on NFS.

  See *Coverity 2020.12 Checker Reference* for details about models.

For Unix-like operating systems only (not Windows, no Windows clients):

- The Coverity intermediate directory can reside on NFS. However, for performance reasons, the local disk is recommended (see Section 3.3.1, “The intermediate directory”).

  For parallel builds, Coverity provides specific recommendations that involve the use of NFS (see Section 3.3.4, “Running parallel builds”). See also the `--capture` option to `cov-build` in *Coverity 2020.12 Command Reference* for additional guidance.
- The Coverity Client tools (Coverity Analysis, Test Advisor, and the Coverity Desktop) may be installed on NFS.
- Compiler configuration files in the `<install_dir>/config` directory (`-c` argument) may reside on NFS.

Chapter 3.6. Coverity Analysis Updates

Whether or not you can download Coverity Analysis updates from the command line depends on how your Coverity administrator configures the Coverity Analysis update feature.

- Starting with the Coverity 2018.03 release, Synopsys allows you to download and install major or minor release Coverity Analysis upgrades from the command line.
- Starting with the Coverity 2017.07-SP2 release, Synopsys allows you to download and install incremental release Coverity Analysis updates from the command line.

If the Coverity Analysis update feature is turned on, then after a successful commit, `cov-commit-defects` checks for any new Coverity Analysis updates. If there are updates, a message appears with the number of updates that you can download. Coverity Connect determines which updates are relevant based on the commit. Coverity Connect notifies you only about relevant updates and makes them available to download. Any other updates are ignored.

The Coverity Analysis *update* files delivered in an incremental release are typically smaller than the Coverity Analysis *upgrade* files you receive as part of a major or minor release. Typically, they do not contain all of the files in a Coverity Analysis installation image, and they might or might not overwrite configuration files. To ensure configuration files are not inadvertantly overwriten, the installer first checks to see if the files to be overwritten have changed. If they have changed, the installer lists the modified files and stops. (If you decide to overwite these changed files anyway, re-run the installer using the `--force` option.

Use the `cov-install-updates` command with its sub-commands and options to query and list the available updates, install the updates in order, and if required, rollback an undesired update. For more information, see the *Coverity 2020.12 Command Reference* ↗.

Part 4. Capturing specific build systems

Table of Contents

Chapter 4.1. Using IncrediBuild

Table of Contents

4.1.1. Building code with IncrediBuild as part of the analysis process

Coverity analysis tools allow you to use Xoreax Software IncrediBuild to accelerate the build of your Windows-based code. IncrediBuild runs the build/capture separate from the Coverity build utilities (`cov-build`/`cov-translate`/`cov-emit`) and produces a JSON script that can be replayed along with the source. The basic workflow is as follows:

1. Perform the build using IncrediBuild.

2. IncrediBuild produces a JSON script that itemizes the compile commands and the environments for the commands.

3. Specify the JSON script to `cov-manage-emit replay-from-script` which reads the JSON script, builds a list of compile commands and executes each of the compile commands against `cov-translate`.

4. Run Coverity analysis tools.

4.1.1.1. Using IncrediBuild to build your code

Important

Integration to Coverity depends on an optional extension package available from Incredibuild. Contact Xoreax Software regarding the availability and licensing of such features.

4.1.1.1.1. Building from the command line

The following section describes the steps for building code with IncrediBuild on the command line.

1. Before you run the build, validate that IncrediBuild is successfully integrated with Coverity tools. For example:

   ```
   BuildConsole /nologo /QueryPackage="COVERITY"
   ```

   If the integration is successful, you should receive a message similar to the following:

   ```
   "Coverity" package is allocated to this Agent (exit code 0)
   ```

2. Perform your build using IncrediBuild with an option (`/ca_file="<file>.json"`) to the `BuildConsole` command. For example:

```
BuildConsole <existing_options> /ca_file="build-integration.json"
```

IncrediBuild produces an additional output of a JSON script that itemizes all the compile commands complete with the environment for each command.

3. Specify the JSON script to `cov-manage-emit replay-from-script`. For example:

```
cov-manage-emit --dir <idir> replay-from-script -if build-integration.json   \
      --compilation-log compile.log -j <num_cores_or_cpus>
```

The `cov-manage-emit` command reads the JSON script, builds a list of compile commands, and executes each of the compile commands against `cov-translate`.

4. Run `cov-analyze`, specifying the intermediate directory produced by the previous step.

4.1.1.1.2. Building in Visual Studio

The following section describes the steps for building code with the IncrediBuild Visual Studio plug-in.

1. Before you run the build, validate that IncrediBuild is successfully integrated with Coverity tools. For example:

```
BuildConsole /nologo /QueryPackage="COVERITY"
```

If the integration is successful, you should receive a message similar to the following:

```
"Coverity" package is allocated to this Agent (exit code 0)
```

2. Perform your build using the IncrediBuild Visual Studio plug-in.

IncrediBuild produces an additional output of a JSON script that itemizes all of the compile commands complete with the environment for each command. The JSON script will be written to:

```
<Incredibuild install dir>\Temp\Ib_<solution name>_<date>.json
```

3. Specify the JSON script to `cov-manage-emit replay-from-script`. For example:

```
cov-manage-emit --dir <idir> replay-from-script -if <json script>.json   \
      --compilation-log compile.log -j <num_cores_or_cpus>
```

The `cov-manage-emit` command reads the JSON script, builds a list of compile commands, and executes each of the compile commands against `cov-translate`.

4. Run `cov-analyze`, specifying the intermediate directory produced by the previous step.

4.1.1.2. Important usage notes

The IncrediBuild build process has the following usage notes:

- Supports Microsoft Visual Studio response files to `CL.exe`.
- Supports Microsoft Visual Studio `PCH` files.

- Does NOT support source files or PCH files that are overwritten, deleted or moved during the build. The source files must exist after the build at the same location at which they were compiled.

  For example, a PCH file that is overwritten multiple times in the same build with different contents cannot be used.

- Does NOT support #import.

4.1.2. Coverity Desktop Analysis

IncrediBuild can also be used to capture a build for Coverity Desktop Analysis. The steps are the same as before, but use --record-only when importing the information into the emit.

```
cov-manage-emit --dir <intermediate_directory> replay-from-script --record-only \
  -if <json_script>.json --compilation-log compile.log -j <number_of_cores_or_cpus>
```

Chapter 4.2. Building with Xcode

Table of Contents

4.2.1. Building Xcode projects that use pre-compiled headers

By default, Xcode projects that utilize pre-compiled header (PCH) files will use a cache directory for generated and referenced PCH files. When capturing an Xcode project based build using `cov-build`, if the PCH cache directory contains PCH files that are used (but not generated) by the build, then their generation will not be observed by `cov-build`. This may result in Coverity compilation errors corresponding to native compiler invocations for source files that require use of the PCH files for successful compilation. In particular, problems arise when compiling source files that depend on the existence of a pre-compiled prefix header, but do not contain a `#include` directive to include the header.

The following techniques can be used to workaround this problem when building with the '`xcodebuild`' utility:

- Specify the '`clean`' build action with the '`xcodebuild`' invocation so that previously cached PCH files are removed and re-generated.

  ```
  > xcodebuild -project my-project.xcodeproj clean build
  ```

- Set the `SHARED_PRECOMPS_DIR` Xcode setting to the path to an empty temporary directory.

  This setting may be specified in the '`xcodebuild`' command line invocation or in an Xcode config file - either in the default location (`~/.xcconfig`) or as specified by the '`-xcconfig`' command line option or the `XCODE_XCCONFIG_FILE` environment variable.

  ```
  > xcodebuild SHARED_PRECOMPS_DIR=/tmp/shared-precomps-dir -project my-
  project.xcodeproj
  ```

- Set the `GCC_PRECOMPILE_PREFIX_HEADER` Xcode setting to disable use of pre-compiled prefix headers. This is only an option if the source project is designed to build successfully without a prefix header, or when the pre-compiled prefix header is not built.

  This setting may be specified in the '`xcodebuild`' command line invocation or in an Xcode config file - either in the default location (`~/.xcconfig`) or as specified by the '`-xcconfig`' command line option or the `XCODE_XCCONFIG_FILE` environment variable.

  ```
  > xcodebuild GCC_PRECOMPILE_PREFIX_HEADER=no -project my-project.xcodeproj
  ```

4.2.2. Building projects that use Xcode 10's new build system

Apple introduced a new build system in Xcode 10 that is not yet supported by Coverity for C, C++, or Objective-C code compiled with Clang. Attempting to capture such projects will result in a 0% capture rate for C, C++, and Objective-C source code. To work around this issue, configure the project to use the

"Legacy Build System". You can do this within the Xcode IDE by selecting the **File** > **Project Settings** or **File** > **Workspace Settings** menu option and selecting **Legacy Build System** from the drop down option for **Build System**. Alternatively, you can select the legacy build system at build time by passing the `-UseModernBuildSystem=NO` option to `xcodebuild`.

Chapter 4.3. Building with Visual Studio 2015+ or .NET Core SDK (‘dotnet’)

Visual Studio 2015 (VS2015) introduced a mechanism called a *shared compiler* for builds of C# and Visual Basic. VS2015, subsequent versions of Visual Studio, and the .NET Core SDK (`dotnet`), use a shared compiler by default.

`cov-build` will not work when a shared compiler is used.

`cov-build` attempts to disable the use of a shared compiler by setting the `UseSharedCompilation` environment variable to `false`. However, this does not always disable the use of a shared compiler. For example, a user can override the environment variable in an MSBuild targets file or a project file.

If you are expecting to see C# or Visual Basic files emitted from your Visual Studio or .NET Core SDK build and are not seeing them, you have the following options:

MSBuild

If you use MSBuild, you can use the command line to force it to not use the shared compiler line. This will override all environment/project-specified values. For example, if your original command is:

```
> msbuild /t:rebuild myproject.sln
```

Change it to:

```
> msbuild /t:rebuild /p:UseSharedCompilation=false myproject.sln
```

.NET Core SDK (`dotnet`)

If you use `dotnet`, you can use the command line to force it to not use the shared compiler line. This will override all environment/project-specified values. For example, if your original command is:

```
> dotnet build --no-incremental myproject.sln
```

Change it to:

```
> dotnet build --no-incremental -p:UseSharedCompilation=false myproject.sln
```

`devenv`

There is not currently a way to specify `UseSharedCompilation=false` for devenv-based build commands on the command line. In the case of devenv, you have two options:

1. Use MSBuild instead of devenv. Then you can use the technique specified for MSBuild above.
2. Modify your `.csproj` files to not set `UseSharedCompilation` to `true` when attempting to capture with `cov-build`. This allows `cov-build` to disable shared compilation using the environment variable mentioned above.

Chapter 4.4. Building with Cygwin

Observe the following guidelines and limitations when working with Cygwin. Cygwin 1.7 (32-bit) is supported on all supported versions of Windows. The 64-bit version is not supported.

- Versions 1.7.1–1.7.9. of Cygwin have a bug that can cause `cov-build` to fail to print the native build's output, and might cause build failures. You can fix this issue by upgrading your version of Cygwin. You can work around this issue by using `--instrument` or by redirecting the output of `cov-build`.

- Due to a change in Cygwin, `cov-build` cannot capture builds for Cygwin versions 1.7.21 and 1.7.22. Cygwin has provided a workaround in the form of an environment variable in versions 1.7.23 or later. The `cov-build` command will now attempt to automatically set this environment variable if Cygwin version 1.7.23 (or later) is detected. You can set the environment variable manually from a Cygwin shell prompt as follows:

  ```
  $ export CYGWIN=wincmdln
  ```

- Cygwin processes are known to be vulnerable to a Microsoft Windows defect (see Known Issue 58684 in the *Coverity Release Notes Archive*) and compiler mis-configurations might occur when using Cygwin compilers. Upgrade to Cygwin version 1.7.18, which contains a workaround for this issue. If you are using later versions of Cygwin, note that Coverity is unable to support Cygwin versions 1.7.21 and 1.7.22 (see above). Coverity recommends Cygwin 1.7.23 or later.

Part 5. Using the Compiler Integration Toolkit (CIT)

Table of Contents

Chapter 5.1. Compiler Integration overview

Table of Contents

Coverity provides support for many native compilers. There are instances, however, when the native compiler accepts non-standard code or has an option that the Coverity compiler misinterprets or does not understand. Additionally, there are native compilers for which Coverity does not provide support. The compiler integration for the Analysis build system is highly configurable and can be customized to accommodate many different compilers and code bases. This document describes many of the compiler integration options and how to use the options to configure native compilers.

This document assumes that as a user of the Coverity Analysis build system, you are familiar with `cov-configure` and the `coverity_config.xml` configuration file. This configuration describes information about a specific installation of a compiler, such as where the configuration should search for system header files, what macros it defines, information about the dialect of C/C++ it accepts, and so forth. This configuration tells the Coverity Analysis build system how it should try to emulate the native compiler. The Coverity Analysis build system can then intercept the calls of the native compiler to facilitate the capture and understanding of the code base that is going to be analyzed.

The Compiler Integration Toolkit (CIT) provides a mechanism for describing the general behavior of a native compiler. A Compiler Integration Toolkit (CIT) configuration is essentially a meta-configuration; its primary function is to tell the `cov-configure` command how to generate a `coverity_config.xml` file for a specific compiler installation. The `coverity_config.xml` and the Compiler Integration Toolkit (CIT) configuration XML use the same DTD and have much in common. Some of the other Compiler Integration Toolkit (CIT) configuration files are passed through verbatim and will used by the `cov-translate` command in addition to the `coverity_config.xml` file.

Most compilers that are supported by Coverity have that support implemented as a Compiler Integration Toolkit (CIT) configuration. These integrations have the most options for customization and bug fixing. Some of Coverity's earliest compiler integrations are not implemented using the Compiler Integration Toolkit (CIT) and are hard-coded into the product. The customization of these implementations is limited and is achieved by manipulating the `coverity_config.xml` file using the `cov-configure --xml-option` option, or by editing the `coverity_config.xml` file directly after `cov-configure` runs.

The Coverity Analysis build system and the Compiler Integration Toolkit (CIT) provide the flexibility to support many native compilers and code bases. For a list of native compilers with successful integrations, see the *Coverity 2020.12 Installation and Deployment Guide*.

5.1.1. Before you begin

This manual assumes that you have a working knowledge of Coverity Analysis and have experience configuring compilers for Coverity Analysis. For information, see Chapter 2.7, *Configuring compilers for Coverity Analysis*.

For information about Coverity commands (`cov-configure`, `cov-emit`, `cov-translate`) including their usage and options, see *Coverity 2020.12 Command Reference*.

5.1.2. Basic requirements

Compiler Integration Toolkit (CIT) configurations require the following:

- Coverity Analysis 6.0 or later
- A licensed copy of the native compiler
- (Not required, but highly recommended) Documentation about the native compiler, including all command line options

Chapter 5.2. The Coverity Analysis build system

Table of Contents

Before you attempt a native compiler integration, it is useful to understand the commands that are run as part of the Coverity Analysis build process, and what type of information they need to successfully complete. The Coverity Analysis build process has the following command binaries:

`cov-configure`

> Probes the native compiler and creates the configuration used by the build system to emulate the native compiler. See Section 5.2.1, “The cov-configure command”.

`cov-build`

> Monitors the native build and invokes `cov-translate` for every invocation of the native compiler. `cov-build` is not relevant to the discussion of the compiler configuration customization and is not covered in this document. For more information, see `cov-build` in the Command Reference.

`cov-translate`

> Emulates the native compiler by mapping the native compiler command line to the command line used by the Coverity compiler. See Section 5.2.2, “The cov-translate command ”.

`cov-emit` (Coverity compiler)

> Parses the code and stores it in the Coverity emit database. This command is not covered in this document. See `cov-emit` in the Command Reference.

`cov-preprocess`

> Produces preprocessed source using either the native compiler or the Coverity compiler. This is useful for debugging parse errors. See Section 5.2.3, “The cov-preprocess command ”.

`cov-manage-emit`

> Manipulates the Coverity emit database in many different ways. It can be used to call `cov-translate`, `cov-emit`, or `cov-preprocess` on a previously captured code base. It can be viewed as a wrapper for `cov-translate`, `cov-emit`, and `cov-preprocess`. This command is not covered in this document. For more information, see `cov-manage-emit` in the Command Reference.

5.2.1. The `cov-configure` command

The `cov-configure` can be used in two ways:

- template mode
- non-template mode

When used in template mode, the generation of a configuration for that compiler is deferred until the compiler is used in the build. In either case, the following steps describe an overview of the configuration generation process:

1. Determine required options.

   Identify any arguments that must be specified for the configuration to be valid. Certain compilers require that different configurations be generated depending on the presence of certain switches.

   For example, the GCC `-m32` switch will cause the compiler to target a 32-bit platform, while `-m64` will target a 64-bit platform. `cov-configure` will record these options in the configuration so the generated configuration will only be used when those options are specified. When using a compiler integration implementation using the Compiler Integration Toolkit (CIT), the required options must have the `oa_required` flag included in the switch specification in the compiler switch file (`<compiler>_switches.dat`). For more information, see Section 5.4.2, “The compiler switch file ”.

2. Test the compiler.

   Runs tests against the compiler to determine its behavior, for example, to determine type sizes and alignments. See Section 5.4.1.4.1, “Test tags” for descriptions of the testing tags that you can set in the configuration file.

3. Determine include paths.

   `cov-configure` determines the include path in three ways and each involves opening standard C and C++ headers:

   - strace - Look at system calls to see what directories are searched. This is not supported on Windows systems.

   - dry run - `cov-configure` can parse include paths from the output of a sample compiler invocation. You can change this behavior with `<dryrun_switch>` and `<dryrun_parse>`.

   - preprocess - The most general solution is to preprocess the file and look for the `#line` directives which details where the files are located.

   For a C compiler, the test gives the compiler these files: `stdio.h`, `stdarg.h`

   For a C++ compiler the test gives the compiler these file `stdio.h`, `stdarg.h`, `cstdio`, `typeinfo`, `iostream`, `iostream.h`, and `limits`. A Compiler Integration Toolkit (CIT) configuration can add additional files to the list of headers. For more information, see Section 5.4.1.4.2, “Additional configuration tags”.

4. Determine macros:

   `cov-configure` determines macros in the following ways:

   - dump - Native compilers can dump intrinsically defined macros when they are invoked with certain switches. You can change this behavior with `<dump_macros_arg>`.

- preprocess - Candidate macros are inserted into a file and the file is preprocessed to determine the macros value. Candidate macros are identified in two ways:

  a. Specified as part of the compiler implementation. Additional macro candidates can be added using the Compiler Integration Toolkit (CIT). For more information, see Section 5.4.1.4.2, “Additional configuration tags”.

  b. System headers are scanned for potential macros.

5. Test the configuration

   Run tests against the configuration to see if it works correctly and then tailor them appropriately. Currently, the only test that is performed is to determine if `–no_stdarg_builtin` should be used or not.

5.2.2. The `cov-translate` command

The `cov-translate` process takes a single invocation of the native compiler and maps it into zero or more invocations of the Coverity compiler. The following steps provide an overview of the `cov-translate` command process:

1. Find the compiler configuration file that corresponds to the native compiler invocation. This involves finding a configuration with the same compiler name, the same compiler path, and the same required arguments. If such a configuration is not found, and the compiler was configured as a template, `cov-translate` will generate an appropriate configuration.

2. The native command line is then transformed into the Coverity compiler commands. All compilers tend to do similar things, so `cov-translate` is broken into phases. Each phase takes the command lines produced in the previous phase as input and produces transformed commands as output. Each phase has a default set of actions and will only appear in a configuration if needed by a particular compiler.

   Expand
   : Expands the command line to contain all arguments. This usually means handling any text files that expand to command line arguments, native compiler configuration files, and environment variables. After this phase, all of the compiler switches should be on the command line.

   Post Expand
   : If the results of transforming the command line in the Expand phase will result in a command line that is not valid for the native compiler, that portion of the transformation should be deferred to the Post Expand phase. The side effect of deferring transformation is that when preprocessing is attempted, or if the replay of a build occurs later, all of the files or environment elements might no longer be present.

   Pre-translate
   : Maps the native compiler switches to the equivalent Coverity compiler switches, or drop the native compiler switches if they do not affect compilation behavior.

Split
: Removes source files from the command line, splitting them into language groups. The default behavior performs the split based on the suffixes of the files.

Translate
: This phase applies to actions that are not explicitly listed in any phases in the configuration XML. For example, the presence of <append_arg>-DFOO</append_arg> outside of any phase tags (such as <post_expand/>) appends `-DFOO` to the command line during the Translate phase. Also, part of this phase is the decision to skip command lines with arguments that you do not want to be emitted. For example, you might want to skip any invocations of the compiler that are only doing preprocessing.

Post Translate
: Applies Coverity compiler command transformation that cannot be performed in the Translate phase.

Source Translate
: Because the split phase removed source files from the command line, there is no opportunity to do command line transformations that are dependent on the name of the source file. For C/C++, this phase will be executed once for each source file to be compiled. For example, for GCC Precompiled header (PCH) file support, you can use this phase to append additional arguments if the source file is a C/C++ header file.

Final Translate
: This translation phase is the last one before the arguments are passed to the Coverity compiler. This phase is reserved for the Coverity Support team to work around any command lines that are improperly handled by their implementations.

3. For the command lines produced by the phases of transformation, the Coverity compiler is then invoked unless `cov-build --record-only` is specified, in which case, Compiler Integration Toolkit (CIT) simply records the Coverity compiler command line for a later invocation as part of `cov-build --replay`.

5.2.3. The `cov-preprocess` command

The first step of the `cov-preprocess` process is to find the appropriate configuration, similar to `cov-translate`. For `cov-translate`, however, a native compiler command line is mapped to what the Coverity compiler expects. For native preprocessing, a native compiler compile command line must be mapped into a native compiler pre-process command line. While it is not as complicated as the former mapping, it is not as simple as adding an option for preprocessing. For example, the following is the command to compile a file with GCC:

```
gcc -c src.cpp -o src.o
```

If it is simply transformed by adding the `-E` option for preprocessing as in the following example, the result would be that the preprocessing would output to `src.o`:

```
gcc -E -c src.cpp -o src.o
```

The mechanism for transforming a native compile command into a native pre-process command is described in Section 5.3.2.1, “Tags used for native preprocessing”.

5.2.4. The `cov-test-configuration` command

The `cov-test-configuration` command is used to test command-line translations of a configuration by making assertions about the translations. It parses an input script and confirms that the commands are true or false.

Example:

```
cov-configure --config myTest/coverity_config.xml --msvc
cov-test-configuration --config myTest/coverity_config.xml MyTests.json
```

Output of the `cov-test-configuration` example:

```
Section [0] My Section Label
Tests run: 1, Failures: 0, Errors: 0
Sections run: 1, Tests run: 1, Failures: 0, Errors: 0
```

Examples of the format to use are found at `<install_dir>/config/templates/*/test-configuration.*.json` for the supported compilers.

Chapter 5.3. Understanding the compiler configuration

Table of Contents

The `cov-configure` command will generate a compiler configuration for every compiler binary and two configurations if the compiler binary can be used for C and C++. This configuration contains two sections, <compiler> and <options>. The following is an example configuration:

```
<?xml version="1.0" encoding="UTF-8"?>
<!DOCTYPE coverity SYSTEM "coverity_config.dtd">
<coverity>

<!-- THIS FILE IS AUTOMATICALLY GENERATED - YOU MAY ADD XML ENTITIES -->
<!-- TO UPDATE THE COMPILER CONFIGURATION AFTER THE begin_command_line_config ENTITY.
 -->
  <cit_version>1</cit_version>
  <config>
    <build>
      <compiler>
        <comp_name>gcc-4</comp_name>
        <comp_dir>C:\cygwin\bin</comp_dir>
        <comp_translator>g++</comp_translator>
        <comp_require>-m32</comp_require>
        <id>g++-gcc-4-4.3.4-0</id>
        <version>4.3.4</version>
      </compiler>
      <options>
        <id>g++-gcc-4-4.3.4-0</id>
        <sysinclude_dir>/usr/include</sysinclude_dir>
        <sysinclude_dir>/usr/include/w32api</sysinclude_dir>
        <preprocess_remove_arg>-o.+</preprocess_remove_arg>
        <preprocess_remove_arg>-c</preprocess_remove_arg>
        <preprocess_remove_next_arg>-o</preprocess_remove_next_arg>
        <preprocess_switch>-E</preprocess_switch>
        <preprocess_compile_switch>-C</preprocess_compile_switch>
        <cygwin>true</cygwin>
        <pre_preinclude_file>$CONFIGDIR$\coverity-macro-compat.h</pre_preinclude_file>
        <pre_preinclude_file>$CONFIGDIR$\coverity-compiler-compat.h</
pre_preinclude_file>
        <prepend_arg>--gnu_version=40304</prepend_arg>
        <prepend_arg>--no_stdarg_builtin</prepend_arg>
        <prepend_arg>--size_t_type=j</prepend_arg>
        <prepend_arg>--ptrdiff_t_type=i</prepend_arg>
        <prepend_arg>--type_sizes=w2x8li4s2e12d8fP4</prepend_arg>
        <prepend_arg>--type_alignments=w2x8li4s2e4d8fP4</prepend_arg>
        <prepend_arg>--ppp_translator</prepend_arg>
        <prepend_arg>replace/typedef(\s|\w)*\swchar_t\s*;/</prepend_arg>
        <skip_arg>-E</skip_arg>
        <skip_file>conftest\.cpp$</skip_file>
        <skip_file>\.y+\.c$</skip_file>
        <opt_preinclude_file>$CONFIGDIR$/../user_nodefs.h</opt_preinclude_file>
        <begin_command_line_config></begin_command_line_config>
      </options>
    </build>
  </config>
  <md5>7121697409837c393faad8ab755fff3b</md5>
</coverity>
```

5.3.1. The <compiler> tags

The <compiler> tag is used to identify which compiler invocations this configuration applies to using the configuration matching rules in Section 5.2.2, “The cov-translate command ”. The possible tags in this section include:

<comp_desc>
: The optional description of this compiler. This information is provided by `cov-configure` for Compiler Integration Toolkit (CIT) implementations.

<comp_dir>
: Specifies the directory name for this compiler.

<comp_generic>
: Provides the name of the directory where Compiler Integration Toolkit (CIT) files (for example, the switch table, compatibility headers, and configuration XML) for a given compiler are stored. For example,

```
<comp_generic>csc</comp_generic>
```

means that the Compiler Integration Toolkit (CIT) files for the compiler are stored under the following directory:

`<template_dir>/csc`

The default value of `<template_dir>` is `<install_dir>/config/templates`. You can change this default with <template_dir>.

<comp_name>
: Specifies the binary name for this compiler.

<comp_next_type>
: Gives the `comptype` for another possible configuration if the language of this configuration is not appropriate after the source is split between C and C++.

<comp_require>
: Defines the parameters that are required before the compiler matches a particular <compiler> tag.

<comp_translator>
: The command-line translator to use for this compiler. This specifies which compiler command line the `cov-translate` program should imitate. You can get a list of supported translators by running `cov-`

`configure --list-compiler-types`. The translators are the first, comma-separated entries on each line in the list. (See sample command output in Chapter 2.7, *Configuring compilers for Coverity Analysis*.) Required.

<could_require_regen>
Indicates cov-translate needs to invoke the native compiler to re-generate files (such as .TLH files) needed by compilation when it replays a compilation command.

<file_exclude_pattern>
Use only with filesystem capture. Files and subdirectories that match the specified regular expression are excluded from the search results and are not included in the analysis. For example, the following tag excludes all paths that contain a directory named `node_modules`.

```
<file_exclude_pattern>[/\\]node_modules[/\\]</file_exclude_pattern>
```

Contents of excluded directories are not searched for further matches.

<file_include_pattern>
Use only with filesystem capture. Specify a regular expression pattern to match source files to be passed to the associated compiler. For example, the following tags comprise a configuration that captures files with a `.js` extension that will be compiled as JavaScript:

```
<comp_generic>javascript</comp_generic>
<file_include_pattern>^.*\.js$</file_include_pattern>
```

Note that the regular expression matches only on filenames and not on directories or path information.

<id>
A unique name for this compiler.

<is_ide>
Indicates the configured target is an IDE binary.

<target_platform_fn>
Specifies the internal function to be used to determine target platform for code instrumentation.

<use_mspch>
Indicates the compiler uses Microsoft style precompiled header (PCH).

<version>
Specifies a version string for the compiler. This tag is only descriptive.

<version_macro>
A macro that contains compiler version information.

gcc version macros:

- `<version_macro>__GNUC__</version_macro>`
- `<version_macro>__GNUC_MINOR__</version_macro>`

<version_output_stream>

By default, compiler version auto detection looks for output on `stdout`. This can be overridden with the value 2, which specifies output to standard error (`stderr`).

<version_regex>

An arbitrary number of regular expressions can be specified in `<version_regex>` tags to form the compiler output into the required format. The expressions are applied in the order they are given in the configuration. For example:

```
<version_regex>replace/.*([0-9]+\.[0-9]+) \[Build .*/$1</version_regex>
```

The following example takes the result of version macros `__GNUC__=3` and `__GNUC_MINOR_=4` and returns `3.4`:

```
<version_regex>replace/(\d+)\s(\d+)/$1.$2</version_regex>
```

<version_switch>

Enables `cov-configure` to attempt to automatically detect the compiler's version number. The value is the compiler switch that prints out the version. For example, for gcc:

```
<version_switch>--version</version_switch>
```

If a compiler prints out the version information when invoked with no arguments, you should add this option with an empty value.

If the wrong version is being reported, you can override the result by manually providing the version number to `cov-configure`. For example:

```
cov-configure --version 2.1 --comptype ...
```

<wchar_t_name>

Defines a custom identifier for the `wchar_t` type. During compiler probes, this type name is used in place of `wchar_t`.

5.3.2. <options> tags in `coverity_config.xml`

The following options control how the command line from your compiler is translated by Coverity tools.

5.3.2.1. Tags used for native preprocessing

<fix_macro_regex>

Specifies a regex that describes how to transform a command line switch from the Coverity `-DMACRO=VALUE` syntax, to the native compiler's format. This is required to add macros to the native command line when it is used for preprocessing.

<preprocess_command>

Runs a command, if any, that replaces the real compiler, to preprocess a file.

Run `cpp` instead of `gcc`:

```
<preprocess_command>cpp</preprocess_command>
```

<preprocess_compile_switch>

Indicates options that are added to the native compiler command line when preprocessing a source file. This is used in addition to <preprocess_switch>. This switch is not used during `cov-configure` when the native compiler is probed.

<preprocess_output>

Indicates the output of the `cov-preprocess` command by using a value.

The value **`1`** or **`-`** specifies standard output; **`2`** specifies standard error; any other value is considered a file name. A file name can contain the special values **`$FILE$`** to indicate the name of the file, **`$FILEBASE$`** to indicate the name of the file without its extension, and **`$PPEXT$`** to indicate `i` for a C file, or `ii` for a C++ file.

Transform `test.c` into `test.i` and `test.cc` into `test.ii`:

```
<preprocess_output>$FILEBASE$.$PPEXT$</preprocess_output>
```

<preprocess_remove_arg>

A Perl regular expression that indicates arguments that should be removed from a compile line to preprocess a file.

Remove output files and compile arguments:

```
<preprocess_remove_arg>-o.+</preprocess_remove_arg>
<preprocess_remove_arg>-c</preprocess_remove_arg>
```

<preprocess_remove_next_arg>

A Perl regular expression that indicates arguments that should be removed, as well as the argument immediately following it, from a compile line to preprocess a file (e.g. **`-o`**).

Remove output files:

```
<preprocess_remove_next_arg>-o</preprocess_remove_next_arg>
```

<preprocess_switch>

Adds an argument to the compiler line to preprocess a file.

Use `-E` to preprocess files:

```
<preprocess_switch>-E</preprocess_switch>
```

<preprocess_response_file>

Instructs `cov-translate` to use a response file when invoking the native compiler to preprocess. You can use `<switch>` to specify the native response file switch, `<suffix>` to specify response file suffix, and `<format>` to specify how response files should be formatted. The following is an example.

```
        <preprocess_response_file>
            <switch>@</switch>
            <suffix>.rsp</suffix>
            <format>default</format>
        </preprocess_response_file>
```

`default` is currently the only allowed value for <format>. It causes each compiler switch to be written on a separate line in the response file.

<preprocess_output_dir_switch>
Specifies the native switch that instructs the native compiler to generate the preprocess output to a particular directory.

<supports_pch>
Indicates that the compiler supports precompiled headers (PCH). By default, PCH support is disabled unless this tag is provided with a value of `true`.

<trailing_preprocess_switch>
Similar to the <preprocess_switch> arguments. The <trailing_preprocess_switch> argument is added near the end of the command line (that is, after the arguments and before the file name) rather than at the beginning.

5.3.2.2. Tags for skipping compilations

<emulate_compile_arg>
Used in combination with `cov-translate --run-compile`. When the `arg` matches the native command line, `cov-emit` will not be invoked and the output of the native compiler will be passed through verbatim.

<skip_arg>
Skips compiles that contain the value given. This causes the translator to not call `cov-emit` whenever this value is seen on the native compiler's command line as a separate, complete argument.

Do not call `cov-emit` on compiler invocations with the "`-G`" argument:

```
<skip_arg>-G</skip_arg>
```

<skip_arg_icase>
Identical to <skip_arg>, except that this tag ignores the case of the expression. The following tag set ignores command lines that contain arguments with the string '-HeLp', '-HELP', '-help', and so forth:

```
<skip_arg_icase>-help</skip_arg_icase>
```

<skip_arg_regex>
<skip_arg_regex> works the same as <skip_arg>, except it performs a regex match. This is similar to <skip_substring> as well, however it provides for situations where a simple substring would not work.

The following example shows how to match `--preprocess=cnl`, `--preprocess=nl`, but not `--preprocess=cl` or `--preprocess`.

```
<skip_arg_regex>--preprocess=.*n.*</skip_arg_regex>
```

<skip_arg_regex_icase>
Identical to <skip_arg_regex>, except that this tag ignores the case of the expression. The following tag set ignores command lines that start with '-h' or '-H'.:

```
<skip_arg_regex_icase>-h.*</skip_arg_regex_icase>
```

<skip_substring>
Skips compiles that contain the value given as a substring of any argument. This causes the translator to not call `cov-emit` whenever any argument on the native compiler's command line contains the value as a substring.

Do not call `cov-emit` on compiler invocations with ".s" as a substring of any argument on the command line:

```
<skip_substring>.s</skip_substring>
```

<skip_substring_icase>
The tag is identical to <skip_substring>, except that this tag ignores the case of the command line when matching. In the following example, command lines will be ignored that have options or arguments that contain "skipme", "SKIPME", "sKiPmE", and so forth. example:

```
<skip_substring_icase>skipme</skip_substring_icase>
```

<skip_file>
Do not compile files that match the given Perl regular expression. This only affects the compilation of the given files, so if several files are on a single command line it will only skip those that actually match (unlike <skip_arg> or <skip_substring>). The file being matched is the completed file name (for example, the current directory is put in front of relative file names), with `/` as a directory separator (even on windows). The match is partial: use ^ and `$` to match boundaries.

Do not compile parser files ending with ".tab.c":

```
<skip_file>\.tab\.c$</skip_file>
```

Java limitation

Though this option removes matching files from the `cov-emit-java` command line, the command will nevertheless emit files that it identifies as dependencies, even if they match the <skip_file> value.

5.3.2.3. Tags that influence translation

<preprocess_first>
Specifies if the build fails because of errors in `cov-emit`'s preprocessing. If this is specified, `cov-build` tries to preprocess each file with the native compiler before sending it to `cov-emit`. This tag does not take a value.

The command to run to preprocess a file is configured by the <preprocess_> options given next. It is constructed based on the command line used to actually compile the file.

<cygwin>
Indicates that the given compiler supports Cygwin file processing.

5.3.2.4. Tags used for transforming the native command line to the Coverity compiler

<id>

A string matching the compiler to which the options under the current <options> tag apply. If you do not specify the <id> tag, the options will apply to all compilers. You can specify multiple compiler <id> tags under a single <options> tag, and the options will apply to all specified compilers.

Make the current <options> tag apply to the compiler with the identifier `gcc`:

```
<id>gcc</id>
```

<native_pch_suffix>

Specifies the suffix of the native compiler's precompiled header (PCH).

<remove_arg>

Removes a single argument from the `cov-emit` command line. This is only needed if for some reason the `cov-translate` program is putting something undesirable onto the `cov-emit` command line.

Remove the `-ansi` argument from the `cov-emit` command line (only needed if `-ansi` appears and is causing a parsing problem):

```
<remove_arg>-ansi</remove_arg>
```

<remove_args>

Removes several arguments from the `cov-emit` command line. This is only needed if for some reason the `cov-translate` program is putting something undesirable onto the `cov-emit` command line. This differs from <remove_arg> in that you can specify the additional number of arguments after the matching <arg> to remove.

Remove `-foo a b` from the `cov-emit` command line, where a and b are the two arguments that follow `-foo`:

```
<remove_args>
  <arg>-foo</arg>
  <num>2</num>
</remove_args>
```

<replace_arg> , <replace_arg_regex>

<replace_arg> replaces an argument from the original compiler command line with an argument that should go into the `cov-emit` command line. <replace_arg_regex> replaces a regular expression from the original compiler command line with a regular expression that should go onto the `cov-emit` command line. These tags are useful if the translator does not understand a custom command line option that can be handled by `cov-emit`.

For example for <replace_arg>, if the compiler command line contains `-mrtp`, add `-D__RTP__` to the `cov-emit` command line:

```
<replace_arg>
  <replace>-mrtp</replace>
  <with>-D__RTP__</with>
</replace_arg>
```

For example for <replace_arg_regex>, if the compiler command line contains `-i<directory>`, add `--include=<directory>` to the `cov-emit` command line:

```
<replace_arg_regex>
  <replace>-i(.*)</replace>
  <with>--include=$1</with>
</replace_arg_regex>
```

Both <replace_arg> and <replace_arg_regex> accept multiple <with> tags, so it is possible to translate a single argument to multiple output arguments. For example (using <replace_arg_regex>):

```
<replace_arg_regex>
  <replace>-foo=(.*)</replace>
  <with>-bar=$1</with>
  <with>-baz=$1</with>
</replace_arg_regex>
```

In this case, `-foo=test` will be replaced with `-bar=test` and `baz=test`.

When a <replace_arg> or <replace_arg_regex> tag is matched, the resulting output is inserted in-place, meaning that the order of the resulting command line is unchanged. Furthermore, <replace_arg> and <replace_arg_regex> tags are applied in the order they appear in the XML, and the results of a given replacement are passed to the next possible replacement. For example:

```
<replace_arg>
  <replace>-foo</replace>
  <with>-bar</replace>
</replace_arg>
<replace_arg>
  <replace>-bar</replace>
  <with>-baz</replace>
</replace_arg>
```

In this case, `-foo` will be replaced by `-baz`, because the second <replace_arg> tag will match the output of the first.

<replace_icase>

A child tag to <replace_arg> and <replace_arg_regex>. When this tag is used, the replacement is applied in a case sensitive manner. For example:

```
<replace_arg>
  <replace_icase>-FOO</replace_icase>
  <with>-bar</with>
</replace_arg>
```

In this case, `-FOO`, `-foo`, `-Foo`, and all other combinations, will be replaced.

<extern_trans>

Invokes an external command. The syntax is:

```
<extern_trans>
  <extern_trans_path>path to your executable</extern_trans_path>
  <extern_trans_arg>…</extern_trans_arg>
  <extern_trans_arg>…</extern_trans_arg>
<extern_trans>
```

The path to the executable is required, but the arguments are optional and will depend on how the executable works. If the path is relative to the Coverity Analysis installation directory, you can use the `$CONFIGDIR$` environment variable, which expands to the absolute path of the installation's `/config` directory.

Example:

```
<extern_trans>
  <extern_trans_path>$CONFIGDIR$/../translator.exe</extern_trans_path>
<extern_trans>
```

In addition to whatever arguments you specify, the following additional arguments will be added:

- The filename containing all the command line arguments that need to be processed, one argument per line.
- The filename of where you should write the new command line, one argument per line.
- After the first two arguments, there are the following optional arguments that are useful to locate helpful files, such as the compiler switch table:
  - `--compiler_executable_dir <path>` - Encodes the location of the native compiler executable.
  - `--compiler_version <version>` - Encodes the compiler version of the native compiler being translated.
  - `--cov-home <path>` - Encodes the location of the Coverity Analysis installation directory.
  - `--cov-type <comp_type>` - Encodes the compiler type.
  - `--template_subdir <path>` - Encodes the `/template` subdirectory for the compiler.

The native command line arguments are not put on the command line to avoid any command line length issues and some instability in pipes on Windows.

<intern_trans>

Invokes a command that is built in to the product. For example:

```
<intern_trans>lintel_pre_translate</intern_trans>
```

This built-in command can be overridden by providing an external translator. The external translator will be found in the same directory as the Compiler Integration Toolkit (CIT) configuration and will have the same name as the built-in command. No user specified arguments are permitted. Only the extra options that were previously described for <extern_trans> are passed. If no extra options are required, specifying <extern_trans> is not necessary.

When a valid internal command is specified, and an external translator of the same name is present in the same directory as the Coverity Compiler Integration Toolkit configuration, the external translator is preferred over the internal command without requiring the presence of <extern_trans>.

It is also possible to specify an external translator within <intern_trans> that is not named the same as any preexisting internal command. In that case, the configuration would then be completely dependent upon the presence of the external translator.

<args_from_env_var>

Specifies an environment variable from which to extract options, in addition to the command line. The <prepend> attribute specifies the name of the environment variable to be used, while the optional <ignore> attribute specifies the name of another environment variable which may or may not be defined. (*Note*: The <add> subtag may be used instead of the <prepend> subtag. However, a warning will be issued if the <add> subtag is used.) Environment variables specified within the <append> subtag will be appended to the command line. If the second environment variable is specified in the <ignore> tag, and that variable is defined in the environment, then all other environment variables contained in the <args_from_env_var> tag are ignored. For example:

```
<args_from_env_var>
    <append>CCTS_OPTIONS</append>
    <prepend>CCTS_OPTIONS</prepend>
    <ignore>CCTS_IGNORE_ENV</ignore>
</args_from_env_var>
```

A fifth subtag, <append_args_found_after_delimiter>, can also be used in conjunction with the above. This tag allows specifying one delimiter, where any arguments found in the environment variable following that delimiter will be appended, and any arguments preceding it will be prepended to the command line. For example:

```
<args_from_env_var>
    <prepend>CCTS_OPTIONS</prepend>
    <append_args_found_after_delimiter>|</append_args_found_after_delimiter>
</args_from_env_var>

CCTS_OPTIONS="-prepended_arg_1 -prepended_arg_2 | -appended_arg_1 -appended_arg_2"
```

The <accept_quoted_delimiters> sub tag affects all of the delimiters specified by the <append_args_found_after_delimiter> tag. For example:

```
<args_from_env_var>
    <append>FOO</append>
    <append_args_found_after_delimiter>|</append_args_found_after_delimiter>
    <accept_quoted_delimiters>true</accept_quoted_delimiters>
</args_from_env_var>

FOO="|-DBAR"
```

In the example above, `-DBAR` is added to the command line while the leading vertical bar `|` has been removed because it is treated as a delimiter. The <accept_quoted_delimiters> tag prevents the value `|-DBAR` from being treated as the additional command line switch. Note that the vertical bar isn't removed because it is not being treated as a delimiter in this case.

<includes_from_env_var>

Specifies an environment variable that defines additional include directories that should be searched during source parsing.

<version_includes_from_env_var>

Specifies a regex that is run against the compiler version to determine if the environment variable should be used.

The specified environment variable is added to the includes if the given regex matches any part of the version string after applying the substitution given by the `version_regex` tag.

The syntax is as follows:

```
<version_includes_from_env_var>
   <version_regex><regex></version_regex>
   <name><environment_variable_name></name>
</version_includes_from_env_var>
```

`<native_pch_suffix>`

Specifies the suffix of native precompiled header (PCH) files.

<include_dir>

This is the directory where user headers are located, to be used by the `cov-emit` command line. The directory is appended with the `cov-emit -I` option.

<sysinclude_dir>

This is the directory where system headers are located, to be used by the `cov-emit` command line. The directory is appended with the `cov-emit --sys_include` option.

<pre_prepend_arg>

Adds an argument to the beginning of the cov-emit command line, ensuring that arguments precede arguments added by <prepend_arg>. Successive arguments will be placed in the order they are declared, the last one being just before the arguments added by <prepend_arg>. Only use this to force certain arguments to come first on the `cov-emit` command line.

<prepend_arg>

Adds an argument to the beginning of the `cov-emit` command line, preceding arguments put out by `cov-translate`. Successive arguments will be placed in the order they are declared, the last one being just before the arguments put out by `cov-translate`. Use <prepend_arg> unless a compelling reason is present to use <pre_prepend_arg> or <append_arg>.

Add `--ignore_std` to the `cov-emit` command line to ignore the `std` namespace for C++ compiles:

```
<prepend_arg>--ignore_std</prepend_arg>
```

Add `--ppp_translator <translator>` to the `cov-emit` command line to translate files before they are preprocessed.

```
<prepend_arg>--ppp_translator</prepend_arg>
```

```
<prepend_arg>replace/(int) (const)/$2 $1</prepend_arg>
```

Prepend "-DNDEBUG" to the cov-emit command line to add the NDEBUG define:

```
<prepend_arg>-DNDEBUG</prepend_arg>
```

<append_arg>

Adds an argument to the end of the cov-emit command line, after arguments put out by cov-translate. Only use this to override erroneous arguments put out by cov-translate.

<drop_prefix> <drop_string>

The cov-translate command attempts to match the next argument (--foobar) after the prefix with the <drop_string> value. If this argument matches, it is ignored. Each time an argument is successfully matched and ignored, it tries to match the next argument against the list of <drop_string> values. As soon as the next argument does not match one of the <drop_string> values, it stops trying, and assumes the next argument after that is the compiler name. You can also use the <drop_count> tag to specify the number of additional arguments after the matching argument to unconditionally drop.

Skip the --skip_me_1 argument, and also the next two arguments:

```
<drop_prefix>
    <drop_string>--skip_me_1</drop_string>
    <drop_count>2</drop_count>
</drop_prefix>
```

<pre_preinclude_file>

Specifies a header file to be included before all other source and header files when you invoke cov-emit. This is equivalent to the --pre_preinclude option of the cov-emit command. The header files that you specify with this tag are processed with cov-emit before all other header or source files. This tag is typically used to include the Coverity compiler and macro compatibility header files that the cov-configure command generates.

<preinclude_file>

Specify <file.h> to be included before most of the source and header files except for those specified with <pre_preinclude_file>, when you invoke cov-emit. This is equivalent to the --preinclude option of the cov-emit command. Header files that you specify with this tag are processed by cov-emit immediately after those that are specified with the <pre_preinclude_file> tag and those passed to cov-emit via the --preinclude_macros option. This option is typically used to include special nodef files that contain macro suppression directives and macros predefined by the compiler.

Preinclude the /nfs/foo/PrefixHeaderForCoverity.h file:

```
<preinclude_file>
/nfs/foo/PrefixHeaderForCoverity.h
</preinclude_file>
```

<opt_preinclude_file>

Specify a file to preinclude during compilation. The file is optional. If no file is specified, this option is ignored.

Add the `nodefs.h` file in the same directory as the current `coverity_config.xml` configuration file to the `cov-emit` command line:

```
<opt_preinclude_file>$CONFIGDIR$/nodefs.h</opt_preinclude_file>
```

5.3.2.5. Tags for phases of command line transformations

All of the options for manipulating command lines (see Section 5.3.2.4, “Tags used for transforming the native command line to the Coverity compiler”) can go directly into the <options> tag. For more control over when the transformation occurs, they can be placed into one of the translation phases using one of the following tags:

<expand>

The `--coverity_resp_file` option is processed during the expand phase. It takes the contents of a text file and adds it to the command line. For example, to map from `@file` you would use the following XML:

```
<expand>
    <options>
        <replace_arg_regex>
            <replace>@(.*)</replace>
            <with>--coverity_resp_file=$1</with>
        </replace_arg_regex>
    </options>
</expand>
```

During this phase, the following switches are processed:

- `--coverity_config_file` - Takes the form `--coverity_config_file=<value>` where value is the name of a response file. Only the argument to the last `--coverity_config_file` will be used.

- `--coverity_resp_file_or_env_var` - Takes the form `--coverity_resp_file_or_env_var=<value>` where `<value>` is either a file name or an envionment variable name. If the environment variable named `<value>` exists and is non-empty, then its value will be added to the command line. Otherwise, `<value>` will be treated as the file name of a response file, and this will be equivalent to `--coverity_resp_file=<value>`.

- `--coverity_translate_config` - Takes the form `--coverity_translate_config=<value>` where `<value>` is a response file filter. `<value>` should be a regular expression to be applied to response files before they are interpreted; you might think of it as `ppp_translator` for response files. The swtich applies to a `--coverity_config_file` specified earlier or later in the command line but only applies to `--coverity_resp_files` specified later in the command line.

<post_expand>

Processes the same switches as <expand>.

- `-coverity_create_pch` - Creates a Coverity precompiled header (PCH) for the specified header file.

- `-coverity_use_pch` - Searches for a Coverity precompiled header (PCH) to be used in compiling the specified file.

<pre_trans>
During this phase, the compiler switch file will be processed.

<split>
During this phase, the following switches are processed:

- `-coverity_no_default_suffixes` - Only treats explicitly defined source file suffixes (for example, those defined through switches such as `-coverity_c_suffixes`, `-coverity_cxx_suffixes`, and so on) as source file name extensions. Default file name extensions such as `.c` for C source files will be disabled. This option should be added during the pre-translate phase and is not implemented for non-CIT (that is, non-Compiler Integration Toolkit (CIT)) compilers.

- `-coverity_c_suffixes` - Takes the form `-coverity_c_suffixes <extension>[;<extension>;<extension...]`. Treats the given file name extensions as C source files. Example: `-coverity_c_suffixes c;i` treats files named `src.c` and `src.i` as files that contain C code. See `-coverity_no_default_suffixes`.

- `-coverity_c_header_suffixes` - Treats the given file name extensions as C header files. See `-coverity_c_suffixes`.

- `-coverity_cxx_suffixes` - Treats the given file name extensions as C++ source files. See `-coverity_c_suffixes`.

- `-coverity_cxx_header_suffixes` - Treats the given file name extensions as C++ header files. See `-coverity_c_suffixes`.

- `-coverity_objc_suffixes` - Treats the given file name extensions as Objective-C source files. See `-coverity_c_suffixes`.

- `-coverity_objc_header_suffixes` - Treats the given file name extensions as Objective-C header files. See `-coverity_c_suffixes`.

- `-coverity_objcxx_suffixes` - Treats the given file name extensions as Objective-C++ source files. See `-coverity_c_suffixes`.

- `-coverity_objcxx_header_suffixes` - Treats the given file name extensions as Objective-C++ header files. See `-coverity_c_suffixes`.

<trans>
Intended to add any additional prevent compiler switches that are specific to whether the source is C or C++. You can avoid this phase in Compiler Integration Toolkit (CIT) implementations by using the `–coverity_cxx_switch` and `–coverity_c_switch` options to specify language specific switches.

- `–coverity_c_switch` - Takes the form `-coverity_c_switch,<switch>[,<switch>,switch...]`. Specify the given switches for compiling the C sources on the command line only. For example, `-coverity_c_switch,-DNOT_CPP src.c src.cpp` will provide `-DNOT_CPP` for `src.c` but not `src.cpp`.

- `–coverity_cxx_switch` - Takes the form `-coverity_cxx_switch,<switch>[,<switch>,switch...]`. Specify the given switches for compiling the C++ sources on the command line only. See `–coverity_c_switch`.

The following options are processed in this phase to emulate the Microsoft-style precompiled header (PCH). They shouldn't be used for any other purpose.

- `-coverity_mspch_create`
- `-coverity_mspch_filename`
- `-coverity_mspch_headername`
- `-coverity_mspch_none`
- `-coverity_mspch_use`
- `-coverity_determine_dir_of_file`

<post_trans>

Translates the command line after the `trans` phase. This is primarily useful for manipulating the command line in the context of legacy compilers.

During this phase, the following switches are processed:

- `-coverity_create_pch` Creates a PCH TU.
- `-coverity_use_pch`: Searches for an existing PCH TU to be used in compiling the current TU.

<src_trans>

During this phase, the following switch is processed:

- `--coverity_remove_preincludes` - Erases all `--preinclude` and `--pre_preinclude` switches from the command line that appear before `--coverity_remove_preincludes`.

  Usage example:

  ```
  --add-arg --preinclude --add-arg foo --add-arg --coverity_remove_preincludes
  ```

5.3.2.6. Tags used to internally pass information from `cov-build`

These XML tags are used internally by `cov-build` to pass information to `cov-translate`.

<cygpath>

Specifies the path in which Cygwin is installed. This does not appear in a configuration file.

<encoding>

Indicates what file encoding to use. This does not appear in a configuration file.

<encoding_rule>

Specifies file encodings on a per-file basis using regular expressions. Within <encoding_rule>, you use <encoding> to specify an encoding for files with names that match the regular expression you specify with <path_regex> or <path_regex_icase>, for example:

```
<encoding_rule>
    <encoding>UTF-8</encoding>
    <path_regex>someFile\.c</path_regex>
</encoding_rule>
```

For case-insensitive regular expressions, you use <path_regex_icase>, for example:

```
<encoding_rule>
    <encoding>Shift_JIS</encoding>
    <path_regex_icase>iregex</path_regex_icase>
</encoding_rule>
```

To use more than one regular expression to match multiple files that use a specific encoding, you can specify more than one <path_regex> and/or <path_regex_icase> under the same <encoding_rule>, for example:

```
<encoding_rule>
    <encoding>EUC_JP</encoding>
    <path_regex>regex.*\.c</path_regex>
    <path_regex>regex2.*\.c</path_regex>
    <path_regex_icase>iregex.*\.c</path_regex_icase>
    <path_regex_icase>iregex2.*\.c</path_regex_icase>
</encoding_rule>
```

For each <encoding_rule>, it is necessary to specify an <encoding> tag and at least one <path_regex> or <path_regex_icase> tag.

Note

Currently, Coverity does not support <encoding_rule> for Java, C#, and the Clang C/C++ compiler.

<enable_pch>

Uses `cov-emit` PCH processing capability to speed up parsing. This does not appear in a configuration file.

<no_caa_info>

Disables additional XREF information necessary for Coverity Architecture Analysis.

5.3.2.7. Tags used to handle response files

You can specify a function that should be used to split text found in response files into separate arguments.

Similar to the `<pre_translate>` function the internal function can be overridden by an external executable if necessary. The added configuration options are both located in the `<options>` section of the `<expand>` tag are as follows:

<intern_split_response_file_fn>

Specify the function that should be used with the function name as the value. For example:

```
<expand>
    <options>
       <option>
         <intern_split_response_file_fn>foo</intern_split_response_file_fn>
       </option>
    </options>
</expand>
```

The choices for internal function:

- `arm_split` - Specifies ARM compilers. ARM compilers have a specific syntax, so they need a different function.
- `default_split` - The default choice. Should handle most cases.
- `line_split` - Specifies that each full line in the response file is an argument (that is, not separated by tabs or spaces). This value is currently set by Compiler Integration Toolkit (CIT) for the Java configuration.

<extern_split_response_file_exe>

Specifies the function that should override the internal function, with the name of the executable that should be used. For example:

```
<expand>
    <options>
       <option>
         <extern_split_response_file_exe>foo</extern_split_response_file_exe>
       </option>
    </options>
    </expand>
```

If both `<intern_split_response_file_fn>` and `<extern_split_response_file_exe>` appear in the configuration, the external executable takes precedence.

<response_file_filter>

Allows regex filters to process the response file prior to parsing it for arguments. These filters are cleared between phases in case different response file formats are used.

<response_file_extension>

Allows for an optional extension to apply to the response file. If the specified response file does not exist, this extension is used to find the response file.

5.3.2.8. Tags to process commented lines in response files

You can specify tags to control whether or not comments should be removed for response files. If you do not specify these tags, the compiler considers everything (including the comments) to be a switch.

These tags are child tags to <expand> and<post_expand>. Acceptable arguments are `yes` or `no`, where `yes` enables the processing of the commented line.

<response_file_merge_lines>
Merges lines that end with backslashes in the response file.

<response_file_strip_comments>
Enables all of the comment filters.

<response_file_strip_poundsign_comments>
Strips a single commented line that begins with the pound sign (#). This filter respects line merges for lines that end with a backslash (\).

<response_file_strip_semicolon_comments>
Strips a single commented line that begins with an unquoted, un-escaped semicolon (;). This filter respects line merges for lines that end with a backslash (\).

<response_file_strip_slashslash_comments>
Strips a single commented line that begins with double slashes (//). This filter respects line merges for lines that end with a backslash (\).

<response_file_strip_slashstar_comments>
Strips all commented lines that begin with a slash star (/*) and end with a star slash (*/).

For example:

```
<post_expand>
   <options>
      <response_file_strip_comments>yes</response_file_strip_comments>
      <response_file_merge_lines>yes</response_file_merge_lines>
   </options>
</post_expand>
```

Will strip the following (example) commented lines:

```
/*
* Add switches to compiler command line
*/
// This one is especially \
     important
-DDEFINE_ME=1
# So is this one
-UUNDEFINE_ME
```

5.3.2.9. Tags to direct which groups the options are applied to

option_group
The options within this group are applied to the variants specified by the `<applies_to>` tag. Please see the following example:

```
<options>
     <option_group>
         <applies_to>gcc,g++</applies_to>
         <compile_switch>-c</compile_switch>
         <preprocess_switch>-E</preprocess_switch>
```

```
    </option_group>
</options>
```

5.3.3. Editing the Coverity configuration file - `coverity_config.xml`

If a compiler cannot be successfully configured and the issues cannot be fixed in the Compiler Integration Toolkit (CIT) configuration, you can modify the Coverity configuration file, `coverity_config.xml`. Use the `cov-configure --xml-option` option and add any of the transformation options. For more information about `cov-configure --xml-option`, see `cov-configure` in the Command Reference.

For the most part, if they are correct, you do not need to edit the `cov-configure` generated files. If there is an incompatibility between your compiler and `cov-emit`, editing the configuration file can be a short-term fix while Coverity improves compiler support in subsequent releases.

All command-line manipulations in the generated configuration are defined with an `<option>` tag. Each `<option>` tag lists all of the automatically generated options, followed by an empty tag of the following form:

```
<begin_command_line_config></begin_command_line_config>
```

You can add all additional command-line manipulations to the `<option>` tag on the lines after the `begin_command_line_config` entity. Do not put modifications inside of the `begin_command_line_config` entity. .

Chapter 5.4. Using the Compiler Integration Toolkit (CIT)

Table of Contents

The Compiler Integration Toolkit (CIT) consists of the following files in order to construct a native compiler configuration:

- The compiler configuration file
- A compiler switch file
- Compatibility header files
- Custom translator script

5.4.1. The Compiler Integration Toolkit (CIT) compiler configuration file

Coverity Analysis detects Compiler Integration Toolkit (CIT) implementations in the `<compiler>_config.xml`. The name of the compiler in the configuration does not have to match the name of the file, but it must match the name of the template directory, as it is what `cov-configure` looks for when it searches for a compiler type. For example, `config/templates/qnx` must have the configuration and switch table named `qnx_config.xml` and `qnx_switches.dat`, respectively. The compiler compatibility headers must match the comptype specified in the configuration file; one per comptype and must be named `compiler-compat-<comptype>.h.`

The configuration file basically describes the following:

1. A high level description of the compiler, for example, compiler type, text description, and whether it is C or C++. It also describes the next configuration if multiple configurations are generated for a single binary.

2. Provides information to `cov-configure` that is useful for generating the configuration. For example, should the compiler attempt to dynamically determine what the correct sizes are for types? Or, what macros are actually defined so that you need not worry about adding a macro that is not present? You can also specify macros that you do not want `cov-configure` to detect.

3. Provides information about the options that the compiler uses, the switches used for compiling and preprocessing, and where the pre-process output is saved.

The rest of the options section is copied verbatim into the generated compiler configuration and consists of a series of actions `cov-translate` is to take during the translation process. For more information, see Section 5.3.2.5, “Tags for phases of command line transformations”.

5.4.1.1. `<cit_version>` tag

The CIT version tag (`<cit_version>`) identifies the compatibility version used for a given template or configuration. The CIT version is a single unsigned integer, with larger numbers representing newer versions. This is used for backwards compatibility between releases.

Newer static analysis releases will be able to understand older CIT compatibility versions, but older releases may not be able to understand newer compatibility versions. The current compatibility version is 1.

5.4.1.2. `<compiler>` and `<variant>` tags

The Compiler Integration Toolkit (CIT) allows you to generate multiple configurations for a single compiler binary. This is done using the `<variant>` tags. Everything defined inside of the `<variant>` tags is specific to a particular configuration. Everything that is not included in the `<variant>` tags is common to all variants. For example, the following configuration will generate C and C++ configurations for a single binary (note how the `comp_next_type points` to the next variant):

```
<?xml version="1.0" encoding="ISO-8859-1"?>
<!DOCTYPE coverity SYSTEM "coverity_config.dtd">
<coverity>
    <cit_version>1</cit_version>
    <config>
       <build>
         <variant>
           <compiler>
             <comp_translator>multi</comp_translator>
             <comp_desc>UNIX like, standards compliant, C compiler</comp_desc>
             <comp_cxx>false</comp_cxx>
             <comp_next_type>multi_cxx</comp_next_type>
           </compiler>

          <options>
             <post_trans>
               <options> <prepend_arg>--c</prepend_arg> </options>
              </post_trans>
           </options>
         </variant>

         <variant>
           <compiler>
             <comp_translator>multi_cxx</comp_translator>
             <comp_desc>UNIX like, standards compliant, C compiler</comp_desc>
             <comp_cxx>true</comp_cxx>
           </compiler>
         </variant>
         <config_gen_info>
               Same as simple XML …
         </config_gen_info>

         <options>
           <compile_switch>-c</compile_switch>
           <preprocess_switch>-E</preprocess_switch>
           <preprocess_output>-</preprocess_output>

            <pre_trans>
               <options>
                   <remove_arg>-c</remove_arg> </options>
            </pre_trans>
         </options>
      </build>
   </config>
</coverity>
```

Multiple compiler names can be specified in a Compiler Integration Toolkit (CIT) compiler configuration for the same compiler type. This allows for easier configuration for compilers with multiple names and are of the same type.

The `<compiler>` tags are as follows:

`<comp_translator>`
The command-line translator to use for this compiler. This specifies which compiler command line the `cov-translate` program should imitate. You can get a list of supported translators by running `cov-configure --list-compiler-types`.

`<comp_desc>`
Descriptive text that is displayed in the configuration files, and when you use the `dump_info` option.

`<could_require_regen>`
Indicates cov-translate needs to invoke the native compiler to re-generate files (such as .TLH files) needed by compilation when it replays a compilation command.

`<is_ide>`
Indicates the configured target is an IDE binary.

`<target_platform_fn>`
Specifies the internal function to be used to determine target platform for code instrumentation.

`<comp_cxx>`
Defines whether the compiler is a C++ compiler (true) or not (false). The `cov-build` command uses this to identify which configuration to use for C files, separate from C++ files.

`<comp_next_type>`
For multiple `<compiler>` definitions, this tag tells `cov-configure` to scan the next `<comp_translator>` section for more possible variants.

`<comp_name>` (optional)
Specifies the binary name that is expected for the compiler type. `cov-configure` uses `<comp_name>` in two ways:

1. If the compiler type is not specified with the `cov-configure --comptype` switch, `cov-configure` attempts to find a compiler type by matching the binary name. Note that in this scenario, `cov-configure` might get the wrong compiler type if more than one have the same binary name.

2. If the binary name matches for the first occurrence of the compiler type AND the compiler type specifies <`comp_type_next`> AND that <`comp_type_next`> has a different binary name, `cov-configure` will search for that different binary and configure it as well, assuming that it is found.

Multiple <`comp_name`> tags are supported for scenario 1 above. For scenario 2, however, the search is only performed if the first <`comp_name`> matches the binary name, and it only searches for the first <`comp_name`> of the second compiler type.

5.4.1.3. <`options`> tags that are specific to the Compiler Integration Toolkit (CIT)

The following tags are specific to the Compiler Integration Toolkit (CIT) configuration. For a complete list of compiler tags, see Section 5.3.1, "The <compiler> tags".

5.4.1.3.1. Tags used for invoking the native compiler and probing

`<dependency_switch>`
: Specifies which switch or switches to add to a compiler command line to get it to dump the list of include files it is using. For example, `gcc -M`.

`<dependency_output>`
: Indicates the output from the file dependency listing. If you do not specify a <`dependency_output`> value, the default is the value set by <`preprocessor_output`>. The value **`1`** or **`-`** specifies standard output; **`2`** specifies standard error; any other value is considered a file name.

  See the <`preprocess_output`> tag.

`<compile_switch>`
: Specifies which switch or switches to add to a compiler command line so it can compile a source file. For example `-c` is the compile switch for `gcc -c t.cpp`.

`<dryrun_switch>`
: Specifies which switch or switches to add to a compiler command line so it can dump its dryrun or verbose output. This usually describes the internal processes that are invoked to support the native compiler. By processing this, `cov-emit` can discover the include directories and predefined macros used by the native compiler.

`<dryrun_parse>`
: Indicates which format of the native compiler dryrun output. The supported formats are generic, gcc, and qnx.

`<dump_macros_arg>`
: Specifies which switch or switches to add to a compiler command line to get it to dump the macros that are predefined by this compiler. For example, `gcc -dM -E t.cpp`. Not all compilers support this option.

`<dump_macros_output>`
: Specifies where the compiler dumps the macros that are predefined by this compiler. The value `1` or `-` specifies standard output. 2 specifies standard error, and any other value is considered a file name. A file name can contain the special values `$FILE$` to indicate the name of the file, `$FILEBASE$` to indicate the name of the file without its extension, and `$PPEXT$` to indicate `i` for a C file, or `ii` for a C++ file.

5.4.1.4. `<config_generic_info>` tags

5.4.1.4.1. Test tags

The following tags are used to configure basic test for your compiler through the `<compiler>_config.xml` file. Not all of the tests are enabled by default. To ensure that the tests are enabled or disabled, explicitly specify the test in the format, `<test>true|false</test>`.

`<custom_test>`
: Probes for arbitrary switches using custom code. For example:

```
<custom_test>
     <source>
        int __global;
     </source>
     <prepend_arg>--no_sun_linker_scope</prepend_arg>
</custom_test>
```

`<disable_comp_tests>`
: When set to true, all compiler probes are disabled by default, but may be individually enabled using the other tags described in this section. Defaults to false when no setting is specified.

`<test_128bit_ints>`
: Tests whether or not 128-bit ints are enabled. This test is disabled by default.

`<test_alternative_tokens>`
: Tests whether or not alternative tokens are in use. This test is disabled by default.

`<test_altivec>`
: Probes for altivec extension support and adds the appropriate compiler switches. This test is disabled by default.

`<test_arg_dependent_overload>`
: Tests whether or not function overloading is argument dependent. This test is disabled by default.

`<test__Bool>`
: Determines whether or not _Bool is a keyword in GNU mode. This test is disabled by default.

`<test_c99>`
: Tests to determine whether or not c99 mode is on by default. This test is disabled by default.

> **Note**
>
> This tag is deprecated, and will be removed in a future release. It is replaced by the `<test_c_version>` tag.

`<test_c_version>`
Tests to determine whether or not c99 mode is on by default. This test is disabled by default.

`<test_char_bit_size>`
Tests to determine the bit width of a single character. If the probe is disabled or the tests are inconclusive, defaults to 8 bits. This test is enabled by default.

`<test_const_string_literals>`
Tests whether or not string literals are const char * or char *. This test is disabled by default.

`<test_cr_term>`
Test whether carriage return characters (`\r`) are treated as line terminators. This test is enabled by default.

`<test_csharp6>`
Test whether the compiler supports C#6 features. This test is enabled by default.

`<test_cygwin>`
Test whether the compiler is Cygwin-aware and can understand Cygwin-style paths. Only applicable on Windows platforms. This test is disabled by default.

`<test_declspec>`
Tests for the presence of `__declspec` is present in the native compiler, and whether it is as a macro or a keyword. This test is enabled by default.

`<test_exceptions>`
Tests whether the native compiler supports exceptions by default in C++ modes. This test is disabled by default.

`<test_gnu_version>`
Checks to see if GCC derived compilers support extensions added by GCC. This test is disabled by default.

`<test_ignore_std>`
Tests whether the native compiler ignores the `std::` namespace, that is, whether it can directly use the names without specifying `using namespace std;`. This test is enabled by default.

`<test_include_path>`
Attempt to determine the compiler's include search paths by probing its behavior. This test is enabled by default.

`<test_include_paths_with_strace>`
Attempt to determine the compiler's `include` search paths by probing its behavior with `strace`. This test is enabled by default.

`<test_incompat_proto>`
Tests whether the compiler accepts incompatible prototypes. Incompatible prototypes still need to have compatible return types. This test is disabled by default.

test_inline_keyword

Tests for the presence of the ISO C99 `inline` keyword in the native compiler. Enabled by default.

`<test_macro_stack_pragmas>`

Tests whether or not the compiler supports macro stack pragmas. This test is enabled by default.

`<test_multiline_string>`

Tests whether the native compiler tolerates newlines within a string. For example:

```
char *c = "Hello
    World";
```

This test is disabled by default.

`<test_new_for_init>`

Tests whether the native compiler uses modern C++ `for` loop scoping rules, or the old Cfront-compatible scoping rules. For example:

```
{
    for (int i = 0; i < a; ++i) { }
    for (int i = 0; i < a; ++i) { }
}
```

This code is valid in modern C++, since the scope of the '`i`' in the first loop ends at the closing brace. However, compilers that implement the old scoping rules will usually issue an error: '`i`' is in scope for the entire enclosing block after its declaration, and the second loop redeclares it. This test is disabled by default.

`<test_old_style_preprocessing>`

Tests whether macros are checked for number of arguments before expansion. This test is disabled by default.

`<test_restrict_keyword>`

Test whether the compiler supports the `restrict` keyword. This test is enabled by default.

`<test_rtti>`

Tests whether or not the native compiler supports RTTI by default. This test is disabled by default.

`<test_size_t>`

Test whether the compiler intrinsically supports the `size_t` type. This test is enabled by default.

`<test_stdarg>`

Test whether the compiler intrinsically supports types and functions from the `<stdarg.h>` header. This test is enabled by default.

`<test_target_platform>`

Determines the code instrumentation target platform by examining the platform macros expanded by the native compiler. This test is disabled by default. The allowed values for `<platform>` are `x64`, `x86`, and `all`. For example, with the follow tags the code instrumentation target platform is set to x64 if the macro _M_AMD64 is defined by the native compiler.

```
<platform_if_macro>
     <macro_name>_M_AMD64</macro_name>
     <platform>x64</platform>
</platform_if_macro>
```

`<test_trigraphs>`

Tests whether or not trigraphs are supported by the compiler. This test is disabled by default.

`<test_type_size>`

Runs tests for basic data types to determine their respective sizes and alignments. This test is enabled by default.

`<test_type_size_powers_of_two_only>`

The same as `test_type_size`, but assumes power of two sizes. Makes `cov-configure` finish the tests slightly faster. This test is disabled by default. If enabled, `test_type_size_powers_of_two_only` will only take effect if `test_type_size` is also enabled.

`<test_type_traits_helpers>`

Tests whether or not the native compiler has type traits helpers enabled by default. This test is disabled by default.

`<test_vector_intrinsics>`

Tests whether the native compiler supports various vector type intrinsics, such as the `__vector` keyword. This test is disabled by default.

`<test_wchar_t>`

Tests for the presence of the `wchar_t` keyword in the native compiler. This test is enabled by default.

`<test_x86_calling_conventions>`

Enables or disables tests to determine whether the compiler supports and enforces x86 calling convention specifications. When disabled, the compiler is assumed to enforce calling conventions.

These tests are disabled by default.

`<use_gnu_version_macro>`

When `test_gnu_version` is enabled, this tag determines how the compiler is probed to determine the GNU compiler version. When set to true, the GNU intrinsic version macros are used (i.e., `__GNUC__`, `__GNUC_MINOR__` and `__GNUC_PATCHLEVEL__`). When unset or explicitly set to false, a heuristic approach is used to determine the GNU version.

5.4.1.4.2. Additional configuration tags

This section describes additional general tags that can have an impact on the Compiler Integration Toolkit (CIT) configuration.

`<macro_candidate>`

Adds a macro to the list of macros that `cov-configure` should try to determine if it should be defined.

`<excluded_macro_candidate>`

Ensures that this macro is excluded from the list of macros that `cov-configure` tries to determine if it should define. A macro is usually excluded if its definition will be controlled by the handling of a command line option.

`<excluded_macro_candidate_regex>`

Ensures that any macros matching the given regex are excluded from the list of macros that `cov-configure` tries to determine if it should define. A macro is usually excluded if its definition will be controlled by the handling of a command line option.

`<extra_header>`

Specifies additional headers to be searched for when trying to determine the include path for a compiler. This is necessary when the detected include path is incomplete.

`<extra_compat_header>`

Specifies additional compatibility headers that should be appended to the generated compatibility headers by `cov-configure`. This can be useful for sharing compatibility header information between different compiler configurations.

To specify the compatibility header in a different folder, you can either use a relative path or `$CONFIG_TEMPLATES_BASE_DIR$`, which is expanded to the absolute path name of the directory that contains the configuration files. For example, the following text, if used in a configuration file, specifies that `compiler-compat-clang-common.h` in the clang folder will be appended to the generated compatibility headers by `cov-configure`.

```
<extra_compat_header>$CONFIG_TEMPLATES_BASE_DIR$/clang/compiler-compat-clang-
common.h</extra_compat_header>
```

`<function_like_macro_candidate>`

Like `<macro_candidate>`, adds the macro specified via the `<name>` tag to the list of candidate macros that `cov-configure` probes, in order to determine an appropriate definition.

The named macro is assumed to be a function-like macro that takes a single argument, and the set of possible arguments is given by the `<argument_candidate>` tags that follow. For example:

```
<function_like_macro_candidate>
    <name>fcn_name</name>
    <argument_candidate>argument_the_1st</argument_candidate>
    <argument_candidate>argument_the_2nd</argument_candidate>
</function_like_macro_candiate>
```

`<include_dependency>`

Specifies that `cov-configure` should use dependency information instead of preprocessing to determine include paths.

`<intern_generate_headers>`, `<extern_generate_headers>`

Experimental feature to allow external programs to generated extra compatibility headers during `cov-configure`. These headers might be removed in a future release.

`<no_header_scan>`

Disable performing a header scan for macro candidates during the probing of a compiler. Values are True or False.

`<platform_if_macro>`

Specifies the <macro_name> and <platform> pairs used by the target platform probe. Please refer to <test_target_platform> .

`<set_env_var>`

Sets the environment variable to the specified value before probing the native compiler. For example, the following tags cause environment variable FOO to be set to value BAR before the native compiler is probed. The value of this environment variable is restored after probing.

```
<set_env_var>
    <env_name>FOO</env_name>
    <env_value>BAR</env_value>
 </set_env_var>
```

`<unset_env_var>`

Unsets the specified environment variable before probing the native compiler. The value of this environment variable is restored after probing.

5.4.2. The compiler switch file

The compiler switch file filters the compiler command line options that are useful to the process. The file also removes and "cleans up" the options that `cov-emit` does not require. The compiler switch file exists in the same directory as the Compiler Integration Toolkit (CIT) configuration and uses the following naming convention:

`<compiler>_switches.dat`

A switch table can import switches from another switch table. For example, the following statement, if used in a switch file, imports all the switches defined in `gnu_switches.dat` to the current switch table. `$CONFIG_TEMPLATES_BASE_DIR$` is expanded to the absolute path name of the directory that contains the configuration files.

```
import $CONFIG_TEMPLATES_BASE_DIR$/gnu/gnu_switches.dat
```

The compiler switch file requires an entry for every option that can be used with the target compiler. If you do not specify an entry and the switch is encountered on the command line, it is passed through to the next phase. If the target compiler switch is never handled, it is only passed to `cov-emit` if `cov-emit` understands the switch. Otherwise, the switch is dropped and a warning is issued. However, this method of determining missing switches is not reliable, as `cov-emit` might understand a switch differently than the native compiler does. So, your switch table should never be incomplete. If a switch has the same meaning to both the Coverity compiler and the native compiler, specify the `<oa_copy>` flag in the switch's description.

If you just have one subtype of a compiler, then just the one compiler switch file is read. The easiest way to support multiple compiler subtypes is to create independent Compiler Integration Toolkit (CIT) configurations, each with its own compiler switch file. If a compiler switch file exists in the subtype directory and the parent directory, the two files will be appended together.

For every option that a compiler generates, there should be a line in the compiler switch file that is in the following format:

```
[ <option>, option_type ]
```

The option should be shown without any of the prefixes that the compiler might use. For example, `-I` should be entered just as `I` without the dash. The `option_type` is a combination of possible (or relevant) ways in which the option might be expressed. The following table lists possible switch options:

Table 5.4.1. Flag options

Flag	Description
oa_abbrev_match	May be abbreviated by any amount up to the short form in capitals.
oa_additional	Indicates that there are *two* arguments that follow a switch, with the second one always of the "unattached" variety. The first can be attached, unattached, or optional. Here is a Sun compiler example where both arguments to `Qoption` are unattached: `-Qoption ccfe -features=bool,-features=iddollar`
oa_alternate_table	Designates that the switch is for specifying switches to another program, such as the preprocessor, and to use an alternate switch table to interpret it. For example, the following signifies that the value to `Xpreprocessor` should be interpreted by `<compiler>_preprocessor_switches.dat` and the results should be appended to the command line: `{"Xpreprocessor", oa_dash|oa_alternate_table, "preprocessor", oa_append},`
oa_append	Options interpreted by the alternate switch table should be appended to the end of the command line. This flag is only valid in conjunction with `oa_alternate_table`. For example: `{"Xpreprocessor", oa_dash|oa_alternate_table, "preprocessor", oa_append},`
oa_attached	Must have an argument attached to the switch, for example `-DMACRO`.
oa_case_insensitive	Will accept upper or lower case, for example: `-D` or `-d`
oa_copy	Passes all instances to `cov-emit`, for example: `-I. -I..`
oa_copy_c_only	Passes to `cov-emit` when compiling C file. This flag overrides `oa_copy` only when language sensitivity is set to true in the translation routine. Otherwise, it behaves identically to `oa_copy`. Most of the Compiler Integration Toolkit (CIT) compilers default to no language sensitivity, but this generally does not cause a problem as a language-sensitive argument only occurs when compiling that mode. Alternatively, you can use `oa_map` instead and map to `–coverity_c_switch,<original switch>`.
oa_copy_cxx_only	Passes to `cov-emit` when compiling C++ file. This flag overrides `oa_copy` only when language sensitivity is set to true in the translation routine. Otherwise, it behaves identically to `oa_copy`. Most of the Compiler Integration Toolkit (CIT)

Flag	**Description**
	compilers default to no language sensitivity, but this generally does not cause a problem as a language-sensitive argument only occurs when compiling that mode. Alternatively, you can use `oa_map` instead and map to –`coverity_cxx_switch,<original switch>`.
oa_copy_single	Passes switch along, however, collapse the switch and its argument into a single argument. For example, `-I dir` would become `-Idir`.
oa_custom	Indicates that this switch will be handled in the custom code of a custom translator.
oa_dash	May be preceded by a dash (`-`), for example: `-D`
oa_dash_dash	May be preceded by two dashes (`--`, for example: `--D`)

oa_discard_prefix	This is the default option for `oa_alternate_table` and is the opposite of `oa_keep_prefix`. `oa_discard_prefix` will take precedence if `oa_keep_prefix` is specified on the `oa_alternate_table` switch and `oa_discard_prefix` is specified in the switch found in the alternate table.
	With the following switch table configuration using `<compiler>_switches.dat`:
	`{"Xpreprocessor", oa_dash|oa_alternate_table, "preprocessor", oa_prepend|oa_keep_prefix},`
	and `<compiler>_preprocessor_switches.dat`:
	`{"D", oa_dash|oa_attached|oa_copy|oa_discard_prefix}, {"I", oa_dash|oa_attached|oa_copy},`
	The following command line:
	`<compiler> -Xpreprocessor -DTRUE=1 -Xpreprocessor -Idir <source_file>`
	will translate into:
	`<compiler> -DTRUE=1 -Xpreprocessor -Idir <source_file>`

oa_equal	May have an argument following an equal sign (=, for example: `-D=value`)
oa_hyphen_is_underscore	Allows non-prefix hyphens within a switch to be interchangeable with underscores. For example, all of the following are recognized as the same switch: • `--this-is-a-switch` • `--this_is_a_switch` • `--this-is_a_switch`
oa_keep_duplicate_prefix	By default, switches that are interpreted by an alternate table will cause the switch that specified the alternate table to be dropped. For example, given these switch tables in `<compiler>_switches.dat`:

Flag	Description

	`{"Xpreprocessor", oa_dash|oa_alternate_table, "preprocessor", oa_prepend},`
	and in `<compiler>_preprocessor_switches.dat`:
	`{"F", oa_dash|oa_unattached|oa_copy},`
	The following command line:
	`<compiler> -Xpreprocessor -F -Xpreprocessor foo <source_file>`
	will result in:
	`-F foo <source_file>`
	However, if `<compiler>_preprocessor_switches.dat` instead has the following:
	`{"F", oa_dash|oa_unattached|oa_copy|oa_keep_prefix},`
	Then the following command line will be unaltered in translation:
	`-Xpreprocessor -F -Xpreprocessor foo <source_file>`
	`oa_keep_duplicate_prefix` can be specified in the primary table as a default for the table:
	`{"Xpreprocessor", oa_dash|oa_alternate_table, "preprocessor", oa_prepend|oa_keep_duplicate_prefix},`
	Note that `oa_keep_duplicate_prefix` and `oa_keep_prefix` differ in the sense that with `oa_keep_prefix`, only the first instance of the prefix is kept, so when `oa_keep_prefix` is used, the command line `<compiler> -Xpreprocessor -F -Xpreprocessor foo <source_file>` yields this result:
	`-Xpreprocessor -F foo <source_file>`
	.

oa_keep_prefix	By default, switches interpreted by an alternate table will have the switch that specified the alternate table dropped. For example, given switch tables in `<compiler>_switches.dat`:
	`{"Xpreprocessor", oa_dash|oa_alternate_table, "preprocessor", oa_prepend},`
	and in `<compiler>_preprocessor_switches.dat`:
	`{"D", oa_dash|oa_attached|oa_copy},`

<table>
<tr><th>Flag</th><th>Description</th></tr>
<tr><td></td><td>The following command line:
<code><compiler> -Xpreprocessor -DTRUE=1 <source_file></code>
will result in:
<code>-DTRUE=1 <source_file></code>
However, if <code><compiler>_preprocessor_switches.dat</code> instead has the following:
<code>{"D", oa_dash|oa_attached|oa_copy|oa_keep_prefix},</code>
Then the following command line will be unaltered in translation.:
<code>-Xpreprocessor -DTRUE=1 <source_file></code>
<code>oa_keep_prefix</code> can be specified in the primary table as a default for the table:
<code>{"Xpreprocessor", oa_dash|oa_alternate_table, "preprocessor",</code>
<code> oa_prepend|oa_keep_prefix},</code></td></tr>
<tr><td>oa_map</td><td>Specifies a switch mapping. For example, to map switch <code>-i</code>, which takes an argument either attached or unattached, to <code>-I</code> which takes the argument attached, specify:
<code>{"i", oa_dash|oa_attached|oa_unattached|oa_map, "I", oa_dash|oa_attached }</code></td></tr>
<tr><td>oa_merge</td><td>Removes white space from values with commas, for example: <code>-Ival, val2 --> -Ival,val2</code></td></tr>
<tr><td>oa_optional</td><td>Adds an optional argument to a compiler switch. This flag is mutually exclusive with <code>oa_unattached</code>.</td></tr>
<tr><td>oa_parens</td><td>Must have an argument specified in parentheses that is either attached or unattached to the switch. For example: "<code>-D(MACRO)</code>" or "<code>-D (MACRO)</code>".</td></tr>
<tr><td>oa_path</td><td>Indicates that an <code>oa_required</code> switch is a path and should be converted to an absolute path during probing. If <code>oa_path</code> is not paired with <code>oa_required</code>, <code>oa_path</code> will have no effect.</td></tr>
<tr><td>oa_plus</td><td>May be preceded by a plus sign (+), for example: <code>+D</code></td></tr>
<tr><td>oa_prepend</td><td>Options interpreted by the alternate switch table should be prepended to the beginning of the command line. This flag is only valid in conjunction with <code>oa_alternate_table</code>. For example:
<code>{"Xpreprocessor", oa_dash|oa_alternate_table, "preprocessor",</code>
<code> oa_prepend},</code></td></tr>
<tr><td>oa_required</td><td>Indicates to <code>cov-configure</code> that the switch significantly changes the behaviour of the compiler in ways that might invalidate the results of the Coverity compiler's probes (For example, -m32 or –m64 for GCC). This tells <code>cov-configure</code> to require that a configuration be created with the same combination of required arguments as those that are present on the command line. In the event of template configurations, <code>cov-</code></td></tr>
</table>

Flag	Description
	translate and cov-build will automatically instantiate the needed configuration if one is not already made. If no template is present, cov-translate and cov-build will fail when encountering a missing configuration.
oa_skip_arg	Indicates that compiler invocations that use the switch are to be skipped by cov-translate. This flag imposes similar semantics as the <skip_arg> family of compiler configuration option tags, but with the following improvements: • Allowed switch prefixes and case insensitivity will be correctly matched without the need for duplicate <skip_arg> tags (e.g., <skip_arg>-E</skip_arg><skip_arg>/E</skip_arg>) or use of regular expressions (e.g., <skip_arg>--?clr</skip_arg>). • Switches that can appear in operands of options associated with an alternate switch table are correctly matched. For example, <skip_arg>-E</skip_arg> won't match gcc -Wp,-E, but if the -E option is specified with oa_skip_arg, the compiler invocation will be correctly skipped. • Compiler configurations that import switch definition files from other compiler configurations will automatically attain the intended skip arg semantics without having to duplicate a set of <skip_arg> directives.
oa_slash	May be preceded by a slash (/), for example: /D
oa_split	Breaks apart values that are really a list of values. A delimiter should follow oa_split, such as in oa_split"," to split on commas. For example, an input switch of -Iinc1,inc2 with oa_dash|oa_attached|oa_copy|oa_split"," will result in -Iinc1 -Iinc2.
oa_strip_quotes	If there are quotes within the value of the switch, erase the outermost set of matching quotes. For example, "-DMACRO='VALUE'" will become "-DMACRO=VALUE". This argument is passed to the compiler after all shell processing of quotes has occurred.
oa_unattached	May have a value after a whitespace, for example: -D value
oa_unsupported	Indicates that the switch is unsupported. If cov-translate encounters this switch it will issue an error and exit with a nonzero result.

Note

The Compiler Integration Toolkit (CIT) only supports one switch per line. In addition, you cannot break a switch's description across multiple lines, as this will cause the translation to not properly execute.

The options can be combined by ORing them together. For example, if the compiler accepts -Dvalue and -D value, then the *option_type* is set to: oa_dash | oa_attached | oa_unattached

If a particular option is to be passed through to cov-emit, then one of the oa_copy options should also be used. In the case of -Dvalue, you can use oa_dash | oa_attached | oa_unattached | oa_copy

The compiler switch files are sorted (longest switches first) to prevent accidental bugs caused by similar switches overlapping. For example, in the following scenario, the description for -D would prevent the description for -DCPU from ever being used:

```
{ "D", oa_dash|oa_attached }

{"DCPU", oa_dash|oa_equal|oa_required }
```

With switch sorting, this scenario does not occur, and -DCPU=XXX appropriately flags a new configuration.

5.4.3. Compiler compatibility header files

Compiler compatibility headers are pre-included by cov-emit to define things that are predefined by the native compiler like macros, intrinsics, or built-in types. Create a file called config/templates/<name>/compile-compat-<comptype>.h and cov-configure will arrange for it to be included in every invocation of cov-emit.

5.4.4. Custom translation code

Custom translation code can be created and executed using the <extern_trans> and <intern_trans> tags. The <intern_trans> tag can only be used by Coverity since the code gets linked directly into cov-translate. The source code for these translators is shipped with the product and can be converted to an external translator by compiling it in combination with intern_to_extern_phase.cpp. For example, tm_compilers.cpp can be compiled as follows (this command is an example and should be adjusted for your compiler. If you do not have a compiler that produces binaries for your system, you can use the Extend SDK compiler):

```
cd <install_dir_ca>/config/templates/tm
```

The compilation should be executed from the <install_dir_ca>/config/templates/<compiler> directory because the binary will be placed into the current working directory and will be automatically retrieved by cov-translate without modifying the configuration file.

The binary name produced by the compilation should match the internal translator specified by the <intern_trans> tag in the <install_dir_ca>/config/templates/<compiler>/<compiler>_config.xml file. However, you should not modify the <intern_trans> tag.

Execute the compilation, for example:

```
<install_dir_ca>/sdk/compiler/bin/g++ -std=c++11 -o trimedia_pre_translate -I. -
I../../cit  \
-DCOMPILER_FILE=tm_compilers.cpp -DFUNCTION=trimedia_pre_translate  \
--static ../../cit/intern_to_extern_phase.cpp
```

Note

There are known issues with Cygwin gcc, so you should use statically linked binaries and the Extend SDK compiler wherever possible.

The following example is typical for a translator. The CompilerOptions class is an executable representation of the compiler switches file.

```
#include "translate_options.hpp"

void trimedia_pre_translate(const CompilerOptions &opts, arg_list_t& in)
{
    arg_list_t out;
    arg_processor mopt(in, out, opts);

    while (!in.empty()) {
        if (mopt("Xc")) {
            if (mopt.extra_arg == "ansi"
                || mopt.extra_arg == "knr"
                || mopt.extra_arg == "mixed"
               ) {
                out.push_back("-coverity_source=c,h");
         }
            else if (mopt.extra_arg == "arm"
                     || mopt.extra_arg == "cp"
                    ) {
                out.push_back("-coverity_source=c++,hpp");
            }
        }
        else if (mopt("Xchar")) {
         if (mopt.extra_arg == "signed") {
             out.push_back("--signed_chars");
         }
            else if (mopt.extra_arg == "unsigned") {
             out.push_back("--unsigned_chars");
         }
     }
        //Automatically translate based on the switch table.
        //Do not remove this call.
     else if (mopt.translate_one_arg()) { }
     else {
         out.push_back(in.front());
                    in.pop_front();
     }
    }
    mopt.finalize();

    in = out;
}
```

5.4.5. Creating a Compiler Integration Toolkit (CIT) configuration for a new compiler

Before you attempt to configure a new, unsupported compiler, there are a number of templates available upon which you can base your configuration (if your compiler is based on an existing compiler type). For example, some compilers are GNU compilers with extensions and modifications that are specific to a particular industry. A number of supported compiler configurations are located in the following directory:

`<install_dir>/config/templates`

If you do not have a compiler that can "share" configuration from one of the templates, then you can start by using the `/generic` template directory.

5.4.6. Creating a compiler from an existing Compiler Integration Toolkit (CIT) implementation

You can create a new Compiler Integration Toolkit (CIT) compiler by deriving from an existing Compiler Integration Toolkit (CIT) implementation. With this feature, you do not have to compile new code to add a new compiler. All that is required is creating a new directory for the compiler under the Compiler Integration Toolkit (CIT) `<install_dir_ca>/config/templates` directory AND a properly formatted derived compiler configuration file within it. Optionally, a switch file as well as additional `compiler-compat` files can be specified. This functionality is only intended for compilers that are extremely similar to compilers that already have Compiler Integration Toolkit (CIT) implementations.

The `pre_translate` function that gets used is the one that is specified in the configuration file of the compiler from which it is being derived. Similarly to how regular configuration files are structured, this can be overwritten through the use of the existing `extern_trans` functionality.

5.4.6.1. Configuration format for derived compilers

There is a new config format for derived compilers, as shown in the example below. Note that lines with a asterisk (*) at the end indicate mandatory tags:

```
<config>
  <build>
    <comp_derived_from>example:compiler</comp_derived_from>*
    <derived_compiler>*
      <comp_name>newCompilerName</comp_name>*
      <default_comp_name>newCompilerDefaultName</default_comp_name>*
      <comp_translator>new:compilercc</comp_translator>*
      <derived_comp_type>example:compilercc</derived_comp_type>*
      <comp_desc>New Compiler CC (CIT)</comp_desc>*

      <comp_family_head>true</comp_family_head>
      <comp_next_type>new:compilercpp</comp_next_type>
      <extra_comp>
        ...
      </extra_comp>
      <config_gen_info>
```

```
        ...
      </config_gen_info>
      <options>
        ...
      </options>
    </derived_compiler>*
//OPTIONAL EXTRA DERIVED COMPILER(S)
    <derived_compiler>
      <comp_name>newCompilerName</comp_name>
      <default_comp_name>newCompilerDefaultName</default_comp_name>
      <comp_translator>new:compilercpp</comp_translator>
      <derived_comp_type>example:compilercpp</derived_comp_type>
      <comp_desc>New Compiler CPP (CIT)</comp_desc>

      <config_gen_info>
        ...
      </config_gen_info>
      <options>
        ...
      </options>
    </derived_compiler>

    <config_gen_info>
      ...Config gen info not specific
    </config_gen_info>

    <options>
      ...
    </options>
  </build>
</config>
```

Each listed `<derived_compiler>` is analogous to a variant from the regular configuration structure. You can add compiler-specific configuration generation information and options under each derived compiler tag, as well as more general configuration generation info and options that will be used for every derived compiler that is listed.

The `<derived_compiler>` tags are:

`<comp_derived_from>`

Used to "find" the configuration file of the compiler that is being derived from. As an example, if you were to derive from the IAR R32C compiler:

```
<comp_derived_from>iar:r32c</comp_derived_from>
```

This corresponds to the directory and subdirectory of the compiler being derived from in the Compiler Integration Toolkit (CIT) templates directory.

`<derived_comp_type>`

Used to find the correct compiler to match within the config file of the compiler being derived from. For example, when deriving from the IAR R32C compiler:

```
<derived_comp_type>renesascc:r32c</derived_comp_type>
```

All of the other tags used in the example above have identical structure and functionality to how they are used in normal configuration files. For more information, see Section 5.4.1, “The Compiler Integration Toolkit (CIT) compiler configuration file”.

Anything that can be specified in a normal configuration file can be specified within the proper section in the derived compiler configuration. In order to override something specified in the configuration file that is being derived from there must be an opposing option. For example, if there is a test that is disabled under the `<config_gen_info>` tag for the compiler being derived from, you only need to enable the test in the derived compiler configuration file.

5.4.6.2. Derived switch files and compat header files

A new switch file and compiler-compat header files can be created within the directory of the new compiler. These files must abide by the current naming format. For example, if a new derived compiler implementation is created in the directory `<install_dir_ca>/config/templates/newcompiler`, the switch file must be named `newcompiler_switches.dat`, and compiler compat files must use the existing naming formats unless the file is manually specified within the configuration file as an extra compat file.

The derived compiler will use the compiler-compat headers and the switch files of the compiler being derived from. Any additional files created in the new compiler directory are added to those when creating the compiler-compat files during configuration. For the switch files, additional switches can be added but existing switches cannot be overridden.

Unless an `<extern_trans>` is specified, the usefulness of the additional switch file is limited to those options that can be fully handled with `<oa_map>`, or those options that just need to be ignored. If additional functionality is required, such as manual handling in a `<pre_translate>` function, then either a regular, non-derived Compiler Integration Toolkit (CIT) implementation must be created for the compiler requiring it, or an `<extern_trans>` must be used in the derived compiler configuration.

Chapter 5.5. Troubleshooting the build integration

Table of Contents

This section describes causes and solutions to problems that you might encounter after the build integration.

5.5.1. Why is no build log generated?

Check the permissions of the directory that is being written to. Even though the final message may indicate that a file is available, you will not see any error message when the file is not written out.

If you cannot write to the expected directory, then give the –dir option either an absolute path to a directory in your home directory, or a relative path to a better location.

5.5.2. I see a header file error: `expected an identifier`

Error:

```
"tasking/c166v86r3/include/stdio.h", line 21: error:
expected an identifier
#ifndef#define#endif
"/tasking/c166v86r3/include/stdio.h",
line 14: error: the #endif for this directive is missing #ifndef _STDIO_H
```

Solution:

There are missing macros. Look at the `stdio.h` file to identify the macro in title. The macro could be removed through a number of reasons. The first to check is the compiler macro and compat files to see if the string has been #define'd to nothing. The next is to check in the coverity_config.xml file for the compiler. The directory to look in, for the file, will be shown in the cov-emit line that failed. There will be a preinclude option followed by a path to the coverity-compiler-compat.h. All the files used are in the same directory. Please note that Microsoft Visual Studio compilers may use “response” files. These are a list of files and options in an external file that is passed to cov-translate as an 'rsp' file. If this is the case, you may not see the complete cov-emit line. To work out which configuration that was being used, you would manually have to work out which compiler was being used and look that up at the top of the build-log.txt file.

5.5.3. I see a header file error: `expected a ';'`

Error:

```
"/workarea09/wa_s30/desyin/removeFrag/sb8/swtools/all_platforms/tasking/c166v86r3/
include/stdio.h", line 136: error:
      expected a ";"
extern   _USMLIB int    fscanf   ( FILE *, const char *, ... );
```

Solution:

This is due to the `_USMLIB` macro not being understood. There are three possible solutions:

- A `<macro_candidate>` tag is needed to probe the compiler for this value during `cov-configure`.
- Another, totally independent macro needs to be defined so that this macro definition gets created.
- The macro must be defined on the command line every time the compiler is invoked (least likely)

It is possible that the native compiler will recognise this and convert it to another text string during the compilation. In this case, you will need to work out what the new text string means. If it has no effect on our analysis, then you can remove the original macro by doing a `#define` of it (to nothing) in the `coverity-compat-<compiler>.h` file in the `/template` directory.

5.5.4. Why is the standard header file not found?

It is possible that during the probing of the compiler, that it does not report all the directories needed. The mechanism that `cov-configure` uses is to give the compiler a small file that does a `#include` of some standard filenames and then looks at the preprocessed output to see where the file came from on the system.

For a C compiler, the test gives the compiler these file:

- `stdio.h`
- `stdarg.h`

For a C++ compiler the test gives the compiler these file:

- `stdio.h`
- `stdarg.h`
- `cstdio`
- `typeinfo,`
- `iostream,`
- `iostream.h`
- `limits`

The paths that are recorded for these files are passed to the `cov-emit` process as a `--sys_include` option. If the directory that a particular file is in is not listed on the `cov-emit` command line, then you can add an extra header filename to the template files. To do this, add a line similar to the following to the `<type>_config.xml` file:

```
<extra_header>headerfile.h</extra_header>
```

If you have multiple variants defined in your `<type>_config.xml` file and the header file only applies to one variant, then the `<extra_header>` line would go in the `<options>` section for that particular variant. The entry is just the filename itself unless you want `cov-configure` to pick up a parent directory. This may be the case when the source code being built might have lines similar to the following:

```
#include <sys/compiler.h>
```

5.5.5. I see the message: `#error No Architecture defined`

Macros are not defined. Some compilers have to be explicitly probed for particular macros. There are a number of reasons why this needs to be done, for instance:

- The compiler can support a number of OS architectures.
- The compiler needs to know a particular variant of the processor.
- A particular macro definition causes the inclusion of particular header files that define a number or related macros.

The probing of the compiler by the `cov-configure` program may require a specific option to be defined on the command line. For example, the Greenhills compiler toolchain uses the `-bsp` option to determine what directory to use `#include` files from. To add this option to the `cov-configure` process, you would need to use the "`--`" option, for example:

```
cov-configure -co ccintppc.exe -pgreen_hills -- -bsp SLS_Debug -os_dir ...
```

Options that are put after the `--` are then put into the `<comp_require>` tag by the `cov-configure` program. This ensures that you can configure the same compiler for more than one usage.

If the compiler will only tell you about a macro if you already know about it, then you will need to trawl through the manual for the compiler and add the macros using the `<macro_candidate>` tag.

Some compilers can be told to give all the macros that they have defined internally to the standard output. For example, the gcc compiler will do this if it is given the option `-dM` when you are preprocessing a file (`-E`). If the compiler is capable of doing this, then `cov-configure` can make use of it to find more macros. If you have the manual for the compiler, find the option(s) that have the desired effect and add them to the configuration file using the `<dump_macros_arg>` tag. For example, for gcc:

```
<options>
     <dump_macros_arg>-dM</dump_macros_arg>
     <dump_macros_arg>-E</dump_macros_arg>
<options>
```

Part 6. Using the Third Party Integration Toolkit

Table of Contents

Chapter 6.1. Overview

The Coverity Connect Third Party Integration Toolkit is a command line tool that imports issues discovered by a third-party analysis tool and the source code files that contain these issues. The issues are then displayed in Coverity Connect allowing you to examine and manage the issues in the same way that you would manage an issue discovered by Coverity Analysis.

For example, the Third Party Integration Toolkit can import results from an analysis run by PMD and can then be viewed in Coverity Connect, alongside analysis results from Coverity Analysis. PMD issues can then be triaged and annotated in Coverity Connect.

The Third Party Integration Toolkit imports your third-party issues through the `cov-import-results` command. `cov-import-results` accepts issue and source file information provided in a JSON import file. The import file is typically created by a tool, such as a script, that you provide (it is not provided by Coverity).

This book provides the following information:

- A tutorial describing the process of running the Third Party Integration Toolkit
- Sample JSON and source files that serve as the basis of the tutorial
- A reference of the JSON elements used in the import file
- Important capacity and performance information and recommendations

Chapter 6.2. Running the Third Party Integration Toolkit

This section demonstrates the work-flow for running the Third Party Integration Toolkit. Certain steps refer to the examples (the JSON file and its referenced source files), which are provided so you can see the relationship of the files and how the information in the files is displayed in Coverity Connect. You can copy these files and use them with `cov-import-results` as a demonstration of the utility.

To run the Third Party Integration Toolkit:

1. Create a JSON file in the format shown in the JSON file example.

   There are some notes to consider for this step:

   - In the example, the `"file"` element references a file named `missing_indent_source.c`. This is the source file that contained the issue discovered by the checker that is described in the JSON file. All filenames must have absolute pathnames, so you will need to update the paths that are used in the example to match your directory structure.

   - See the Import file reference section to see how to integrate multiple source files and their related issue data.

2. Run the `cov-import-results` command to extract the issue data from the JSON file. For example:

   ```
   cov-import-results --dir dirEx --cpp doc_example.json
   ```

   - The command is located in `<install_dir_ca>/bin`.

     Note

     If you run separate `cov-import-results` commands or run `cov-import-results` after `cov-analyze`, you must add the `--append` option to add the results to the intermediate directory. Otherwise, the `cov-import-results` will replace the contents of the intermediate directory with the its results.

   - `--dir` specifies the intermediate directory from where you will commit your third party issues.

   - `--cpp` is the domain (language) for the issues. In this case, the domain is C/C++. The Third Party Integration Toolkit also accepts `--java` (Java), `--cs` (C#), and `--other-domain` (another domain/language).

     Note

     - You can only specify one domain at a time for `cov-import-results`. If you want to import issues from different domains, you must run separate `cov-import-results` commands and commit each of them (see the next step).

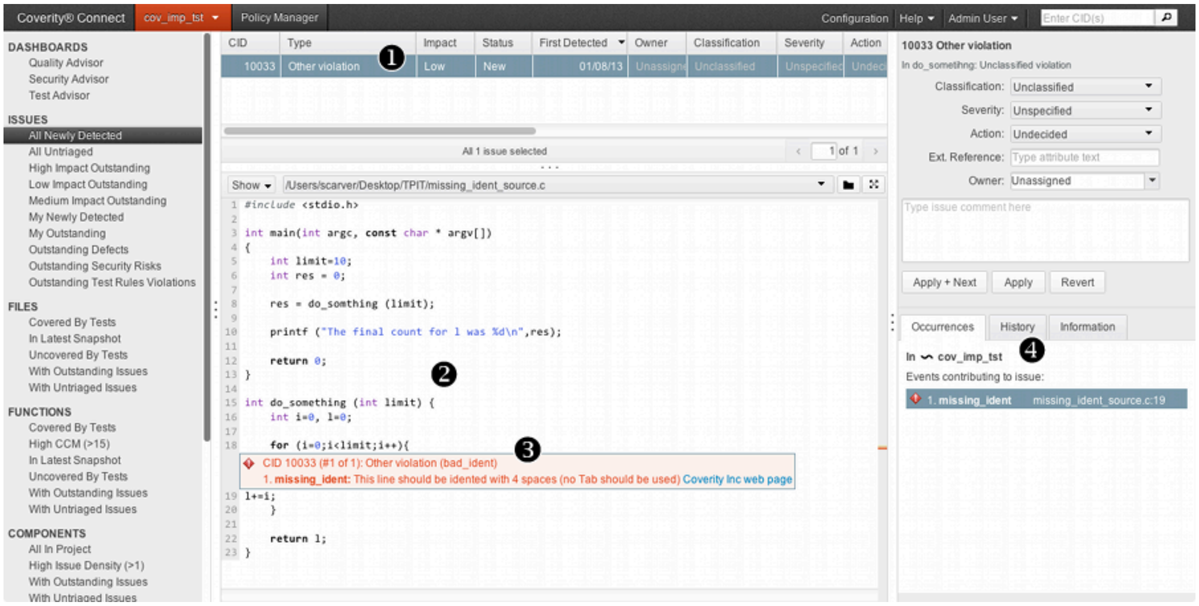
3. Commit the issues to Coverity Connect. For example:

   ```
   cov-commit-defects --dir dirEx --host localhost --user admin --port 8008
   --stream cov_imp_tst
   ```

If you have imported issues with different specified domains you need to run a separate `cov-commit-defects` command line for each domain type. The stream you commit to also must match the domain type that you specify.

4. Log into Coverity Connect, and navigate to your issues list.

Figure 6.2.1. Coverity Connect with imported third-party issues



CID	Type	Impact	Status	First Detected	Owner	Classification	Severity	Action
10033	Other violation	Low	New	01/08/13	Unassigne	Unclassified	Unspecified	Undeci

All 1 issue selected 1 of 1

/Users/scarver/Desktop/TPIT/missing_ident_source.c

```
#include <stdio.h>

int main(int argc, const char * argv[])
{
    int limit=10;
    int res = 0;

    res = do_somthing (limit);

    printf ("The final count for l was %d\n",res);

    return 0;
}

int do_something (int limit) {
    int i=0, l=0;

    for (i=0;i<limit;i++){
```

CID 10033 (#1 of 1): Other violation (bad_ident)
1. **missing_ident:** This line should be idented with 4 spaces (no Tab should be used) Coverity Inc web page

```
l+=i;
    }

    return l;
}
```

10033 Other violation

In do_somethng: Unclassified violation

Classification: Unclassified
Severity: Unspecified
Action: Undecided
Ext. Reference: Type attribute text
Owner: Unassigned

Type issue comment here

In cov_imp_tst

Events contributing to issue:

1. **missing_ident** missing_ident_source.c:19

The image above shows how third-party issues are displayed in Coverity Connect. This display image is the result of a commit with the example import file using the example missing_indent_source.c source file (both are described in the next chapter. The call-outs denote the area of the Coverity Connect UI that displays the relevant import file elements. Additionally, the items listed below link to a description of the displayed elements :

1. Issue listing:
   - issues:subcategory
2. Source code in the Source browser:
   - issues:function
3. Event information leading to the issue in the source:
   - issues:subcategory
   - issues:checker
   - events:description

- events:main

4. *Occurrences* tab describes event information that leads to the issue:

   - events:tag
   - events:file or issues:file
   - events:line

Note

If you have created custom checkers, you can import the issue results found by those checkers using `cov-import-results`. By importing the results, you can utilize the checker-based Coverity Connect and Coverity Policy Manager filters (such as, by Impact rating and Checker name).

For information about creating custom checkers, see the *Coverity Platform 2020.12 User and Administrator Guide*.

When `cov-import-results` runs on high-density files (files with more than 100 issues that also average more than 1 issue for every 10 lines of code), the console will print a warning that names all the files that exceed the threshold, and the import process will exclude all issues associated with the affected files from the intermediate directory. This change prevents the Coverity Connect source browser from becoming too crowded with issues.

To suppress this density check (allowing all issues to be imported) in version 7.0, define the environment variable COVERITY_ALLOW_DENSE_ISSUES when running the commands.

Chapter 6.3. Import file format and reference

Table of Contents

6.3.1. Import file format examples

This section describes the format and attribute values of the JSON file that you must construct in order to import third-party issues into Coverity Connect, including:

- A sample import file
- Sample source files that the import file references
- Import file reference that describes the elements used in the import file

Any field preceded by a question mark (?) is optional.

```
{
"header" : {
    "version" : 1,
    "format" : "cov-import-results input"
},
"sources" : [{
        "file" : "/projects/cov-import-test/doc_example/missing_indent_source.c",
       ? "encoding" : "ASCII",
       ? "language" : string
    },
    {
     "file" : "/projects/cov-import-test/doc_example/too_many_characters.c",
        "encoding" : "ASCII"
    }
],
"issues" : [{
    "checker" : "bad_indent",
    "extra" : "bad_indent_var",
    "file" : "/projects/cov-import-test/doc_example/missing_indent_source.c",
  ? "function" : "do_something",
    "subcategory" : "small-mistakes",
  ? "domain" : string

  ? "properties" : {
      ? "type" : "Type name",
        "category" : "Category name",
        "impact" : "Medium",
      ? "cwe" : 123,
        "longDescription" : "long description",
        "localEffect" : "local effect",
        "issueKind" : "QUALITY"
    },
    "events" : [{
        "tag" : "missing_indent",
        "description" : "Indent line with 8 spaces (do not use Tab)",
      ? "linkUrl" : "http://www.synopsys.com/",
      ? "linkText" : "Synopsys, Inc web page",
        "line" : 19,
      ? "main" : true
        }
    ] },
{
    "checker" : "line_too_long",
    "extra" : "line_too_long_var",
    "file" : "/projects/cov-import-test/doc_example/too_many_characters.c",
    "function" : "do_something_else",
    "subcategory" : "small-mistakes",
    "events" : [
        {
        "tag" : "long_lines",
        "description" : "This line exceeds the 80 character limit",
     ?  "linkUrl" : "http://www.synopsys.com/",
     ?  "linkText" : "Synopsys, Inc web page",
        "line" : 4,
     ?  "main" : true
        }
    ] }
] }
```

Example 6.3.2. Source file 1 - missing_indent_source.c

```
#include <stdio.h>

int main(int argc, const char * argv[])
{
    int limit=10;
    int res = 0;

    res = do_something (limit);

    printf ("The final count for l was %d\n",res);

    return 0;
}

int do_something (int limit) {
    int i=0, l=0;

    for (i=0;i<limit;i++){
        l+=i;
    }

    return l;
}
```

Example 6.3.3. Source file 2 - too_many_characters.c

```
#include  <stdio.h>

int do_something_else () {
 printf("This is an example of a pretty long line, which will exceed the 80 character
 rule");
}
```

6.3.2. Import format reference

The following syntax explains the structure of the JSON import file. Note the following:

- Items shown in bold are to be entered in your import file exactly as shown.
- Items shown in italics refer to subsequent items, or to items of JSON syntax.
- Items shown with ellipses (...) indicate that there can multiple occurrences of that item.
- Definitions and usage notes for the items are listed in the Table 6.3.1, “Import file item definitions”.

```
file ← {
        "header": header,
        "sources" : [
             source , …
         ] ,
        "issues": [
             issue, …
         ]
}
header ← {
        "version" : integer ,
        "format" : string
}
source ← {
        "file" : string ,
        "language" : string ,
        "encoding" : string
}
issue ← {
        "checker" :  string ,
        "extra" :  string ,
        "file" : string ,
        "function" : string ,
        "domain" : string ,
        "subcategory" : string ,
        "properties" : properties ,
        "events" : [
             event, …
         ]
}
properties ← {
        "category" : string ,
        "impact" : string ,
        "type" : string ,
        "cwe" : integer ,
        "longDescription" : string ,
        "localEffect" : string ,
        "issueKind" : string
}
event ← {
        "tag" : string ,
        "description" : string ,
        "file" : string ,
        "linkUrl" : string ,
        "linkText" : string ,
        "line" : integer ,
        "main" : boolean
}
```

The following table is a reference for the JSON elements that are used to construct the import file and defines the following:

- *JSON element* is the name of the JSON element listed in the import file.
- *Required* tells if the element is required or optional.
- *Descriptions* defines the JSON element value.
- *GUI Display* shows in what area the element value is displayed in Coverity Connect using the example data provided in this section.

- *Merge Key* shows which elements in the file affect the way in which issues (CIDs) are merged and displayed in Coverity Connect.

  The Merge Key is a unique identifier for an issue. It is used to determine if two issues are the "same", for example, if they were detected in two slightly different versions of the same code base. Every CID corresponds to a single Merge Key.

  Every issue specified in the JSON file should include the checker name (`issue.checker`) and the file name (`issue.file`). While the function name (`issue.function`) is optional, it is strongly recommended to set it for all defects in code. It can be left unset for defects in configuration files, text files, or unparseable code, which are not applicable. Excluding the function name can produce unexpected results.

  If `issue.function` is set, then the merge key is exactly a function of the following: `issue.checker`, `issue.extra`, and `issue.function`. On the other hand, if `issue.function` is unset, then the merge key will instead be a function of `issue.checker`, `issue.extra`, and the file name (not the complete path) from `issue.file`.

  Any functions named `main` are handled specially, and also include the file name or the first parent directory from `issue.file`.

  The data used to calculate the Merge Key should generally be stable over time. If any one of the values change, a new Merge Key (and new CID) will result, and issues associated with the old Merge Key will no longer be detected, and will appear as "fixed".

Note

`cov-import-results` does not accept JSON import files that contain Windows file paths. You must use forward slashes ("/") to separate paths for Windows and include drive-letter syntax. For example:

```
"file" : "C:/projects/cov-import-test/doc_example/missing_indent_source.c",
```

For more information about JSON and its syntax, see http://www.json.org.

Table 6.3.1. Import file item definitions

JSON element	Required	Description	GUI Display	Merge Key
header	required	The object that identifies the file format. Do not change the values.		
version	required	The value is "1".		
format	required	The value is "`cov-import-results input`".		
sources [ ]	required	An array of source objects. Source objects identify information pertaining to a source file that contains the issue. You can specify 0 or more sources.		
source.file	required	The full pathname and filename of the source file that you want to import so that it displays in the		

JSON element	Required	Description	GUI Display	Merge Key
		Source browser in Coverity Connect. On Windows systems, you must use the drive letter format and forward slashes ("/") to denote path separation, such as `"C:/path/filename"`. You can trim portions of the pathname using the `--strip-path` option.		
source.encoding	optional	The encoding type for the file. The encoding types are the same that are accepted by the `cov-emit` command. Defaults to the system default encoding.		
source.language	optional	The primary source language of the source file.		
issues [ ]	required	An array of issue objects. Issue objects describe all of the information about the specific third-party issues and how that information is displayed in the Coverity Connect UI. You can specify 0 or more issues.		
issue.checker	required	Name of the checker that found the issue. The checker name lengths must be between 3 and 256 characters.	3	Yes
issue.domain	optional	The analysis domain associated with this issue.		
issue.extra	required	A string that allows Coverity Connect to determine if a given issue is new, or if it is an additional instance of an existing issue. Coverity Connect combines the checker, file name, function, and extra fields to define a unique signature for each issue. If the signature matches an existing signature, the two issues are considered to be the same (merged).		Yes
issue.file	required	The full pathname and filename of the source file that contains the issue. You can trim portions of the pathname using the `--strip-path` option. The file must match a source file in the “sources” array, or a source file already present in the intermediate directory (placed there by a preceding invocation of `cov-build` or `cov-import-results`). On Windows systems, you must use the drive letter format and forward slashes ("/") to denote path separation, such as such as "C:/path/filename".	3, 4	Yes, but only if issue.function is not present, or is present but is ambiguous.

JSON element	Required	Description	GUI Display	Merge Key
issue.function	optional	The name of the function that contains the issue. Name mangling is optional.	2	Yes
issue.subcategory	required	The `subcategory` and `tag` attributes, along with the domain definition specified in `cov-import-results`, are used to identify the issue's *type*. *type* is a brief description of the kind of issue that was uncovered by one or several checkers, and is displayed in the event's message in the source browser. If you want to categorize, and accordingly display *type* for an issue, a custom checker description must be defined in Coverity Connect. If you do not define a custom checker description, the issue's *type* is displayed as *Other violation* in Coverity Connect. For more information, see "Configuring custom checker descriptions"in the *Coverity Platform 2020.12 User and Administrator Guide*.	1, 3	
properties	optional	The object that identifies properties of software issues, the same sort of properties that are associated with issues found by checkers. If this element is present in the file, all of its fields *except for cwe* are required. Invalid values will be rejected by `cov-import-results`.		
property.category	required[a]	A string between 1 and 100 characters long that identifies an issue category. See issue category.		
property.impact	required[a]	A string that describes the impact of the issue. It is displayed in Coverity Connect UI elements, such as columns and filters. See impact. Valid values: "Low", "Medium", "High".		
property.type	required[a]	A string between 1 and 100 characters long that describes the checker type. It is displayed in Coverity Connect UI elements, such as columns and filters. See type.		
property.cwe	optional	Integer that maps issues found by the checker to a Common Weakness Enumeration for software weaknesses. It is displayed in Coverity Connect UI elements, such as columns and filters. See CWE.		

JSON element	Required	Description	GUI Display	Merge Key
property.localEffect	required[a]	A string of 0 to unlimited length that is displayed in the Coverity Connect triage pane. See local effect.		
property.longDescription	required[a]	A string of 0 to unlimited length that serves as a description of the issue. It is displayed in the Coverity Connect triage pane. See long description.		
property.issueKind	required[a]	A string that identifies the kind of issue found. It is displayed in Coverity Connect UI elements, such as columns and filters. See kind. Valid strings: "QUALITY", "SECURITY", "TEST", or "QUALITY,SECURITY".		
events [ ]	required	Array of event objects. Event objects describe all of the even information that leads to the issue. You can specify 0 or more event objects.		
event.tag	required	See subcategory.	4	
event.description	required	A description of the event, helping you to identify the impact of the issue. Event descriptions should be a single, short sentence, providing explanatory information for Coverity Connect users. For example, an event message for the existing RESOURCE_LEAK checker is "At (3): Variable "p" going out of scope leaks the storage it points to." See Example 6.3.1, “JSON file - example.json” for more event description examples.	3	
event.file	optional	The full pathname and filename of the file containing the event. This is normally not needed. The default is the filename of the issue. On Windows systems, you must use the drive letter format and forward slashes ("/") to denote path separation, such as "C:/path/filename".		
event.linkUrl	optional	Any valid URL that you wish to include as part of the event message, such as a link to an internal site containing more information about the issue. You can only specify one link for each event.	3	
event.linkText	optional	The text that is displayed in the event message that serves as the hyperlink to the URL provided in `event.linkUrl`.	3	
event.line	required	The line number of the source code in which the event occurs. You must specify one or more.	3	

JSON element	Required	Description	GUI Display	Merge Key
event.main	optional	Denotes the nature of the event's path. The value can be `true` or `false`. It is `true` if this event is the main event.	3	

[a] Required only if properties is present in the JSON file.

Chapter 6.4. Capacity and performance

The Third Party Integration Toolkit does not impose any hard limit on the number of source files, issues, events, and so forth, that can be imported into Coverity Connect. However, you must make sure that Coverity Connect is properly sized to handle the size of code base, issue types, and issue density, as well as the number of concurrent commits, and that the Coverity Connect UI is performing well. See *Coverity Platform 2020.12 User and Administrator Guide* for information about Coverity Connect tuning.

If you import a high issue density, large source files, and so forth, you might notice degradation in performance of Coverity Connect. Frequent commits of non-Coverity issues might cause the database to increase in size, which might result in further performance degradation, causing Coverity Connect to become unresponsive.

Because of this, it is recommended that the following limits be considered when building integration with a third party analysis tool. Ignoring one of the following might cause performance degradation of Coverity Connect:

1. Size of a single source file should not exceed 1MB
2. Number of source files should not exceed 30,000
3. Size of JSON file should not exceed 60MB
4. Database size should not exceed 300GB
5. Density should not exceed 100 issues per thousand lines of code
6. Events per issue should not exceed 25 events per issue
7. Size of a single event should not exceed 300 characters
8. Total Emit directory size should not exceed 8GB

Appendix A. Coverity Analysis Reference

Table of Contents

A.1. Troubleshooting Coverity Analysis

A.1.1. Windows systems

You might encounter the following issues if you use Coverity Analysis on Windows systems.

When using the `cov-build` command with Cygwin `make`, I get an error about not being able to load `cygwin.dll`.

Run the `cov-build` command in a Bourne or Bash shell. For example:

```
> <install_dir_sa>/bin/cov-build.exe --dir <intermediate_directory> bash make
```

The problem is that `cov-build` executes `make` as if it were a native Windows program, but `make` is usually invoked from the Cygwin `bash` shell, which invokes it differently. Having `cov-build` use a Bourne or Bash shell lets the shell invoke `make` in the correct manner.

When using the `cov-build` with Cygwin and a shell script that invokes a build, I get a CreateProcess error.

Execute the build script in a Bourne or Bash shell using the format `sh|bash <script_name>`, where `<script_name>` is the script that executes the build.

For example:

```
> <install_dir_sa>/bin/cov-build.exe --dir <intermediate_directory> \
  sh build.sh
```

The problem is that Windows does not know how to associate a Cygwin shell script with the Cygwin shell that processes it. Therefore, you need to explicitly reference the shell when using the script.

The `cov-commit-defects.exe` command hangs when an invalid port is used for the remote host. When the host running the Coverity Connect uses Windows firewall and an invalid port is used with `cov-commit-defects.exe --host`, the command fails without an immediate error message. Eventually, a timeout error is returned.

Make sure to use the correct port. Also, check that the Windows firewall is configured to unblock the necessary port, or allow the Coverity commands to run as exceptions. See also the previous two questions.

The cov-analyze command returns error: boost::filesystem::path: invalid name

For cov-analyze, the --dir option does not support a path name with just the root of a drive, such as d:\.

For cov-analyze, the --dir option does not support a path name with just the relative directory of a drive, such as d:foo. Valid values for path names with drives use the full directory name in addition to the drive letter (for example, d:\cov_apache_analysis), or a relative directory path name without a drive letter.

The cov-analyze command returns error: [FATAL] No license file (license.dat) or license configuration file (license.config) found

If you get a fatal No license found error when you attempt to run this command, you need to make sure that license.dat was copied correctly to <install_dir>/bin.

On some Windows platforms, you might need to use administrative privileges when you copy the Coverity Analysis license to <install_dir>/bin. Due to file virtualization in some versions of Windows, it might look like license.dat is in <install_dir>/bin when it is not.

Typically, you can set the administrative permission through an option in the right-click menu of the executable for the command interpreter (for example, Cmd.exe or Cygwin) or Windows Explorer.

The cov-configure returns error: access denied

On some Windows platforms, you might need to use Windows administrative privileges when you run cov-configure.

Typically, you can set the administrative permission through an option in the right-click menu of the executable for the command interpreter (for example, Cmd.exe or Cygwin) or Windows Explorer.

A.1.2. All operating systems

You might encounter the following issues if you use Coverity Analysis on all operating systems.

http_proxy, https_proxy, and no_proxy environment variables

Analysis may fail or return inaccurate results when run on networks using HTTP client proxies. Specifically, issues are known to arise when the http_proxy or https_proxy environment variable is a machine name rather than an IP address, or when there are wildcards in the no_proxy environment variable.

A.2. Using Cygwin to invoke cov-build

Coverity Analysis supports the Cygwin development environment. The cov-build command supports build procedures that run within Cygwin, so you can use build procedures without modifications.

You can run Coverity Analysis commands from within Cygwin. However, when running these commands, you cannot use Cygwin paths as command line option values. Cygwin paths are UNIX-style paths that Cygwin translates into Windows paths. Instead, use only Windows paths. You can convert Cygwin paths to Windows paths with the Cygwin utility cygpath -w.

The command that cov-build runs is found through a Windows path. If cov-build cannot find the correct build command, invoke bash first. For example:

```
> cov-build --dir <intermediate_directory> bash -c "<cygwin command>"
```

A.3. Finding Third-party Licenses

Coverity Analysis includes third-party software. For the terms of the associated licenses, see the files in the `<install_dir>/doc/licenses` subdirectory. Some of this software is covered by the Lesser GNU Public License (LGPL). Coverity will provide materials on demand, including source code for components covered by the LGPL, as required by the terms of the LGPL.

A.4. Incompatible #import Attributes

The Microsoft Visual C++ `#import` directive is used to incorporate information from a type library. The extracted information is then converted into valid C++ code and fed into the compiler. The Coverity compiler also uses this generated code. The code, however, can be generated incorrectly if during a single compilation a type library is included multiple times with different attributes. The Coverity compiler generates the following warning when this happens:

```
"t.cpp", line 2: warning: incompatible #import attributes (previous import at
    line 1)
#import "t.tlb" no_namespace
```

To avoid this issue, you need to add guards around every `#import`, for example:

```
#ifndef __import_MSVBBM60_dll
#define __import_MSVBBM60_dll
#import "MSVBVM60.dll" raw_native_types raw_interfaces_only
#endif

#ifndef __import_MSVBBM60_dll
#define __import_MSVBBM60_dll
#import "MSVBVM60.dll" raw_native_types
#endif
```

Appendix B. Coverity Glossary

Table of Contents

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

The number of execution paths in a function, with loops counted one time at most. The following assumptions are also made:

- `continue` breaks out of a loop.
- `while` and `for` loops are executed exactly 0 or 1 time.
- `do…while` loops are executed exactly once.
- `goto` statements which go to an earlier source location are treated as an exit.

Acyclic (Statement-only) Path Count adds the following assumptions:

- Paths within expressions are not counted.
- Multiple case labels at the same statement are counted as a single case.

advanced triage

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.

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:

- 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*.

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

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.

By default, an ignored defect is classified as `Intentional`. See "Models and Annotations in C/C++" in the *Coverity Checker Reference*.

See also function annotation.

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 *instances* (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 `else` branch of an `if` statement which tests that condition cannot be reached on that path. Any defects found in the `else` 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 *Coverity Checker Reference*.

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.

I

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

A directory that is specified with the `--dir` 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 `--dir` option also write data to or read data from this directory.

The intermediate representation of the build is stored in `<intermediate_directory>/emit` directory, while the analysis results are stored in `<intermediate_directory>/output`. This directory can contain builds and analysis results for multiple languages.

See also data directory.

intermediate representation

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.

interprocedural analysis

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.

intraprocedural analysis

An analysis for defects within a single procedure or function, as opposed to interprocedural analysis.

issue

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.

Note that this glossary includes additional entries for the various types of issues, for example, an inspected issue, issue category, and so on.

issue category

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.

Examples:

- `Memory - corruptions`
- `Incorrect expression`
- `Integer overflow Insecure data handling`

Impact tables in the *Coverity 2020.12 Checker Reference* list issues found by checkers according to their category and other associated checker properties.

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

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.

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.

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.

modeling primitive

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.

For example, the C/C++ checker CHECKED_RETURN is associated with the modeling primitive `_coverity_always_check_return_()`. This primitive tells CHECKED_RETURN to verify that the function being analyzed really does return a value.

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.

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

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.

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.

snapshot scope

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.

source

An entry point of untrusted data. Examples include environment variables, command line arguments, incoming network data, and source code.

static analysis

Analysis of software code without executing the compiled program. See also dynamic analysis.

status

Describes the state of an issue. Takes one of the following values: `New`, `Triaged`, `Dismissed`, `Absent Dismissed`, or `Fixed`.

store

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.

stream

A sequential collection of snapshots. Streams can thereby provide information about software issues over time and at a particular points in development process.

T

tainted data

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.

translation unit

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.

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 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 C. Coverity Legal Notice

Table of Contents

C.1. Legal Notice

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`), 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.

Redistribution and use in source and binary forms, with or without modification, are permitted provided that the following conditions are met:

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 are discoverable or detectable.

SYNOPSYS 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. SYNOPSYS AND ITS SUPPLIERS SPECIFICALLY DISCLAIM ALL IMPLIED WARRANTIES, CONDITIONS AND REPRESENTATIONS ARISING OUT OF COURSE OF DEALING, USAGE OR TRADE.