



# Coverity 2020.12 Security Directive Reference

Reference for security directives used to customize Coverity Analysis behavior

Copyright 2020 Synopsys, Inc. All rights reserved worldwide.

---

## Table of Contents

1. Security configuration file .....	1
1.1. Uses of directives .....	1
1.2. Supported languages .....	2
1.3. How to invoke a custom configuration .....	2
2. Configuration file syntax .....	4
2.1. Extensions to JSON supported by the configuration file .....	4
2.2. JSON Terminology .....	4
2.3. Schema .....	5
3. Configuration file usage .....	9
3.1. User directives .....	9
3.2. Other object types used by user directives .....	62

---

# Chapter 1. Security configuration file

## Table of Contents

1.1. Uses of directives .....	1
1.2. Supported languages .....	2
1.3. How to invoke a custom configuration .....	2

A *security configuration file* either alters the default behavior of a checker that is provided with Coverity Analysis, or it defines a new, custom checker.

Coverity Analysis provides a set of directives that allow you to define new checkers and to modify the behavior of existing Web application and Android application security checkers; for example, to support new frameworks and APIs or to suppress false positive defect reports.

To use this functionality, you must create a file in JSON syntax and pass it to either the `--directive-file` option or, for the `DC.CUSTOM` checker, the `--dc-config` option of `cov-analyze`. See the section “How to invoke a custom configuration”.

The following sections describe the syntax of these JSON files, including some extensions to standard JSON that this format supports. The subsections of Section 2.3, “Schema” describe the individual directives, along with objects that support their use.

## 1.1. Uses of directives

Each directive has a specific effect on the analysis. Here are some things you can accomplish with directives:

- Create your own checker that reports a defect on XML files that match an XPath query or text files that match a regular expression (see Section 3.2.20, “RegularExpression”). See the description of `TEXT.CUSTOM_CHECKER` in the *Checker Reference* .
- Support Web frameworks that define new application entry points. See Section 3.1.29, “simple\_entry\_point” and Section 3.1.3, “async\_method”.
- Extend tainted dataflow checkers (such as SQLI and XSS) by identifying more sources of untrusted data, for example, from your custom libraries or frameworks. See Section 3.1.31, “tainted\_data”, Section 3.1.21, “method\_returns\_tainted\_data”, and Section 3.1.7, “data\_has\_tag”.
- Create your own tainted dataflow checker. See the description of `DF.CUSTOM_CHECKER` in the *Checker Reference* .
- Extend existing tainted dataflow checkers (such as SQLI and XSS ) by identifying more *sinks*, which are functions or other program locations that are vulnerable to attack if unsanitized, user-controlled (tainted) data flows into them. See Section 3.1.30, “sink\_for\_checker” and Section 3.1.7, “data\_has\_tag”.
- Extend existing dataflow checkers (such as SQLI, XSS, and `SENSITIVE_DATA_LEAK` ) by adding dataflow pass-through rules for your custom libraries or frameworks. See Section 3.1.19,

“method\_returns\_param”, Section 3.1.9, “dataflow\_through\_callsite”, Section 3.1.3, “async\_method”, Section 3.1.15, “local\_callback”, Section 3.1.16, “map\_read”, Section 3.1.17, “map\_write”, and Section 3.1.7, “data\_has\_tag”.

- Suppress false positive defect reports from certain checkers or groups of checkers. See Section 3.1.26, “sanitizer\_for\_checker”, Section 3.1.2, “android\_safe\_categories”, Section 3.1.1, “android\_protected\_intent\_actions”, Section 3.1.12, “ignore\_all\_argument\_dataflow\_to\_method”, Section 3.1.13, “ignore\_method\_dataflow”, and Section 3.1.14, “ignore\_method\_output”.
- Perform advanced tuning on the XSS checker. Section 3.1.4, “class\_like\_print\_writer\_for\_servlet\_output”, Section 3.1.11, “define\_lookup\_method\_call\_map”, Section 3.1.20, “method\_returns\_servlet\_output\_stream”, Section 3.1.23, “method\_with\_servlet\_sinks\_on\_input”, Section 3.1.24, “method\_with\_servlet\_sinks\_on\_output”, Section 3.1.25, “move\_xss\_outside\_method”, and Section 3.1.33, “xss\_sanitizer\_method”.
- Tune the WEAK\_GUARD checker. See Section 3.1.28, “sensitive\_operation”.

## 1.2. Supported languages

Each configuration file applies to a *single language category*. An analysis can employ more than one configuration file.

The following languages are supported by the configuration file:

C, C++, Objective-C, Objective-C++, C#, Java, JavaScript, .NET, and Visual Basic.

The configuration file’s `language` field specifies the language category that this configuration file supports. See Section 2.3.1, “Top-level value”.

## 1.3. How to invoke a custom configuration

When you have created a custom configuration file, you can use it in analysis by passing it to the `cov-analyze` command.

The `--directive-file` option can specify a configuration file that uses any directives *other than* `dc_checker_name` and `method_set_for_dc_checker`. For examples of what these other kinds of directives can do, see “Uses of directives”.

For example, here is a configuration to define a dataflow checker that identifies and warns about an API to which user-controllable data should not be passed:

```
{
  sink_for_checker : "DF.DANGEROUS_ROBOT",
  sink : {
    to_callsite : {
      callsite_with_static_target : {
        "named" : "battle.robot.api.RobotService.run(java.lang.String, int)void"
      },
    },
  },
  input : "arg1"
```

```
}  
}
```

To use `DF.DANGEROUS_ROBOT` in an analysis, you might save this configuration as `dangerous_robot.json` and then use a command line such as the following:

```
cov-analyze --dir localTempDir --directive-file dangerous_robot.json
```

(For more information about this sample checker, see the description of `DF.CUSTOM_CHECKER` in the *Checker Reference*.)

 **Note**

To specify the configuration file for a `DC.CUSTOM_CHECKER`, whose use is now discouraged, required a different `cov-analyze` option, `--dc-config`.

When you invoke `cov-analyze`, you can specify more than one configuration file, in order to use more than one configuration in your analysis. For example, you might want to test both C++ and Java source in a single scan.

---

## Chapter 2. Configuration file syntax

### Table of Contents

2.1. Extensions to JSON supported by the configuration file .....	4
2.2. JSON Terminology .....	4
2.3. Schema .....	5

A security directives configuration file uses a variant of the JSON format. Overall, the file consists of two parts:

1. Three initial fields. These identify the file as a security directives file, specify the directives format version it uses, and then the language to which the directives apply.
2. A `directives` object. The `directives` contains the directive sub-objects that specify what this configuration file accomplishes. It can also contain sub-objects whose specifications support the behavior of the directives themselves.

### 2.1. Extensions to JSON supported by the configuration file

Though the directive language is based on JSON, it also supports the following extensions, all of which retain the property that the file format is a subset of JavaScript. If you intend to use more standard JSON-processing tools, you might want to avoid using these extensions.

- Comments are allowed, both single-line comments that start with `//` and extend to the end of the line, and multiline comments that start with `/*` and end with `*/`.
- In standard JSON, field names must appear within double quotes. In an analysis configuration file, you may omit the quotes if the name conforms to the customary rules for identifiers: that is, if the name matches the following regular expression (regex):

```
^[a-zA-Z_][a-zA-Z0-9_]*$
```

All field names in this file format conform to that regex, so none of them requires quotes in the configuration file. However, quoting them is permissible, and conforms to standard JSON.

- String literals can be extended across multiple lines (without introducing newlines into the string contents) by joining quoted string literal fragments with the `+` token, optionally surrounded by white space (including newlines). A string value can be composed of any number of concatenated fragments. This syntax follows that of JavaScript string concatenation.
- Objects and arrays can have a final, optional comma ( `,` ).

### 2.2. JSON Terminology

JSON value

A JSON value can be one of the following:

- A literal string, in double quotes ( `"` ).

- A literal numeric value
- A Boolean value: `true` or `false`
- A JSON array
- A JSON object
- `null`

#### JSON object

A paired name (a string) and value, enclosed in braces ( `{ }` ) and separated by a colon ( `:` ).

These name+value pairs are also known as *fields*.

#### JSON array

An ordered collection of JSON values, which can include JSON objects or nested arrays. The array is enclosed in square brackets ( `[ ]` ), and its values are separated by commas ( `,` ).

## 2.3. Schema

This section describes the schema of the Security Configuration file, starting with the top-level JSON value.

### 2.3.1. Top-level value

The top-level value is a JSON object with the following fields:

- `type`: Must be the string `"Coverity analysis configuration"`.
- `format_version`: A number indicating the version of the directives format for this file. Different versions support different directives for different languages. The `type` and `format_version` fields ensure that the provided file is compatible with the current version of `cov-analyze`.



#### Recommendation

If you analyze code from several languages in the same intermediate directory, you should use version 4 or later because the `language` field restricts the evaluation of directives to source code in the specified language(s), and therefore avoids unintended application of directives and useless evaluation of directives on unintended languages.

**Table 2.1. Supported `format_version` field values in release 2020.12**

Valid value	Changes in this version
12	For Visual Basic, introduced support for <code>DC.CUSTOM_CHECKER</code> custom directives.
11	<ul style="list-style-type: none"> <li>• For Visual Basic, introduced support for <code>DF.CUSTOM_CHECKER</code> directives.</li> <li>• Introduced <code>".NET"</code> as a language value that can match C#, Visual Basic, and .NET bytecode.</li> </ul>

Valid value	Changes in this version
8	<ul style="list-style-type: none"> <li>• Added the <code>TEXT.CUSTOM_CHECKER</code> directive.</li> <li>• Changed how to specify custom don't-call (<code>DC.*</code>) and custom dataflow (<code>DF.*</code>) checkers.</li> <li>• Updated <code>CSRF</code> directives.</li> <li>• Updated <code>MISSING_AUTHZ</code> directives.</li> <li>• Added the "sink_kind" field.</li> <li>• Added the "read_from_HANA_library_import" directive.</li> </ul>
6	<ul style="list-style-type: none"> <li>• Added JavaScript support.</li> <li>• Added the Java and C# directive <code>sanitizer_for_checker</code>.</li> </ul>
5	<ul style="list-style-type: none"> <li>• For Java and C#, introduced the "with_annotation" <code>MethodSet</code> and <code>ClassSet</code> through the new <code>AnnotationSet</code> object.</li> <li>• For Java and C#, introduced support for <code>DF.CUSTOM_CHECKER</code> custom checker directives.</li> <li>• Introduced support for these new directives: <ul style="list-style-type: none"> <li>• For Java and C#: <code>sink_for_checker</code></li> <li>• For Java Android: <code>android_safe_categories</code></li> <li>• For Java Android: <code>android_protected_intent_actions</code></li> </ul> </li> </ul>
4	Required the top-level <code>language</code> field and added C# support for many former Java-only directives.
3	Introduced support for these Java directives: <ul style="list-style-type: none"> <li>• <code>method_returns_tainted_data</code></li> <li>• <code>sensitive_operation</code> (see <code>WEAK_GUARD</code> checker annotations)</li> <li>• <code>xss_sanitizer_method</code></li> </ul>
2 or greater	Introduced support for a Java directive: <code>simple_entry_point</code>
1 or greater	Support for all other directives (pre-version 2 directives) <sup>a</sup>

<sup>a</sup>See Section 3.1, "User directives".

- `language`: This field became mandatory, starting in version 4. Directives in this file apply only to source code in the specified language or language family. The following table describes valid values for the `language` field values in release 2020.12 (matches are case insensitive).

**Table 2.2. language values**

Value	Meaning
C-like	Directives apply to C, C++, Objective-C, and Objective-C++ code.
C#	Directives apply to C# code. Note that unsafe C# code blocks and raw pointer types are not supported.
Java	Directives apply to Java code.
JavaScript	Directives apply to all JavaScript code, including client-side JavaScript, JavaScript in HTML, and Node.js code.
.NET	Directives apply to all C# and Visual Basic code. This is only usable at the top level.
Visual Basic	Directives apply to all Visual Basic code.

In `format_version` : 4 and earlier, there is no `language` field. In those versions, the `"dc_checker_name"` and `"method_set_for_dc_checker"` directives for `DC.CUSTOM_CHECKER` apply to the C-like, Java, and C# languages. Other directives apply only to Java.

 **Requirement**

You can specify a maximum of one `language` field and value per file. For example, imagine that you have some directives that you want to apply to Objective-C, others that you want to apply to C++ code, and yet others that you want to apply to C# code. You need at least two directives files, one with `"language" : "C-like"` for the Objective-C and C++ directives and one with `"language" : "C#"` for the C# directives.

- `directives`: An array of User Directive values. See Section 3.1, “User directives”.

The `directives` array contains the directive fields that specify this particular configuration.

Schema example:

```
{
  "type" : "Coverity analysis configuration",
  "format_version" : 12,
  "language" : "Java",
  "directives" : [
    // directives appropriate for Java go here
  ]
}
```

### 2.3.2. Objects in the configuration file

In the configuration file, each directive is a JSON object. The directive contains fields to specify a particular analysis behavior. A configuration file can also include data objects that support the work of the directives.

The directives themselves are described in Section 3.1, “User directives”.

Objects associated with directives are described in Section 3.2, “Other object types used by user directives”.

---

## Chapter 3. Configuration file usage

### Table of Contents

3.1. User directives .....	9
3.2. Other object types used by user directives .....	62

What your configuration file does, depends on the contents of the `directives` object. These entries can be directives themselves, or definitions of other objects that the directives use.

### 3.1. User directives

Each directive is a JSON object (see Section 2.2, “JSON Terminology”) that contains fields to specify a particular analysis behavior.

#### 3.1.1. `android_protected_intent_actions`

Supported Languages: Java Android only

The `android_protected_intent_actions` directive specifies Android `Intent` actions to treat as protected.

The Android analysis considers some `Intent` actions to be protected because `Intent` objects that contain such an `Intent` action can only come from a trusted source (for example, from the Android system). If each `intent-filter` of an Android application component either contains a `safe category` or contains only protected `Intent` actions, the analysis assumes that the component can only receive `Intent` objects from trusted sources.

Examples of `Intent` actions that the analysis considers to be protected:

- `android.intent.action.AIRPLANE_MODE`
- `android.intent.action.BATTERY_CHARGED`
- `android.intent.action.BATTERY_LOW`

If multiple `android_protected_intent_actions` directives are specified, the analysis considers the union of all the `Intent` actions specified in all the `android_protected_intent_actions` directives.

##### 3.1.1.1. Fields

This directive uses the following field:

`android_protected_intent_actions`

Specifies a JSON array of strings. Each string is the name of a protected `Intent` action.

##### 3.1.1.2. See also

The `android_safe_categories` directive.

### 3.1.2. android\_safe\_categories

Supported Languages: Java Android only

The `android_safe_categories` directive specifies categories within Android `Intent` objects to treat as safe.

If an `Intent` object contains a category that the analysis deems safe, the analysis will also assume that the `Intent` object comes from a trusted source. If each `intent-filter` of an Android application component either contains a `safe` category or contains only protected `Intent` actions, the analysis considers that the component can only receive `Intent` objects from trusted sources.

By default, the analysis considers the following categories to be safe:

- `android.intent.category.HOME`
- `android.intent.category.LAUNCHER`

This directive can be used to extend the list of categories that the analysis considers to be safe.

If multiple `android_safe_categories` directives are specified, the analysis considers the union of all the categories specified in all the `android_safe_categories` directives.

#### 3.1.2.1. Fields

This directive uses the following field:

`android_safe_categories`

Specifies a JSON array of strings. Each string is the name of a safe category.

#### 3.1.2.2. See also

The `android_protected_intent_actions` directive.

### 3.1.3. async\_method

Supported Languages: JavaScript only

The `async_method` directive identifies callback functions such as event handlers, Web application entry points, or other callbacks that some framework or runtime system calls asynchronously. It also provides details about the arguments with which the function is called. This directive is similar to the `local_callback` directive, but it is for callbacks that are called asynchronously instead of immediately.

#### 3.1.3.1. Fields

This directive uses the following fields:

`"async_method"`

Specifies a `WritableProgramData` value to identify the callback function that is called later. See Section 3.2.26, “`WritableProgramData`”.

`"handler_kind"`

Specifies a JSON string to describe the kind of callback being called. This field can have one of the following values:

"event\_listener"

The callback might be called many times, as part of an event loop.

"async\_method"

The callback is called once; for example, after an asynchronous operation completes.

"webapp\_entry\_point"

The callback might be called many times, in response to requests from a client.

"input\_tags"

(Optional) Specifies a non-empty JSON array of `InputTag` values. These values indicate that particular parameters to the callback have particular tags. See Section 3.2.10, "InputTag".

"input\_taints"

(Optional) Specifies a non-empty JSON array of `InputTaint` values. These values indicate that particular parameters to the callback are tainted. See Section 3.2.9, "InputTaint".

"input\_values"

(Optional) Specifies a non-empty JSON array of `InputValue` values. If this field is present, `async_method` must be a "to\_callsite" `WritableProgramData` value. The `InputValue` elements describe how arguments at the call site that registers the callback (that is, the call site specified in the `to_callsite` sub-element of `async_method`) flow to parameters of the callback.

See Section 3.2.26.1, "to\_callsite", Section 3.1.3, "async\_method", Section 3.2.11, "InputValue", and Section 3.2.26, "WritableProgramData".

### 3.1.3.2. Examples

JavaScript example 1:

The following directive indicates that any function assigned to the `onkeydown` property of any object is an event listener (and thus potentially invoked many times in the event loop). For example, the anonymous function in `element.onkeydown = function () { flag = true; }` would be registered as an event handler.

```
{
  "async_method": {
    "write_off_any": [ { "property": "onkeydown" } ]
  },
  "handler_kind" : "event_listener"
}
```

JavaScript example 2:

The following is a simplified version of an `webapp_entry_point async_method` directive for Express.js.

Directive:

```
{
  "tag" : "ExpressApp",
  "data_has_tag" : {
```

```
    "from_callsite" : {
      "call_on" : {
        "read_from_js_require" : "express"
      }
    },
    "output" : "return"
  }
},
{
  "async_method" : {
    "to_callsite" : {
      "call_on" : {
        "read_from_object_with_tag" : "ExpressApp",
        "path" : [ { "property" : "post" } ]
      }
    },
    "input" : "arg2"
  },
  "input_tags" : [
    {
      "input" : "arg1",
      "tag" : "ExpressRequest"
    },
    {
      "input" : "arg2",
      "tag" : "ExpressResponse"
    }
  ],
  "handler_kind" : "webapp_entry_point"
}
```

The first directive says that `app` in the code below has the tag `"ExpressApp"`. The `"async_method"` directive uses this tag to recognize the anonymous function in the code below as a Web application entry point and to tag its parameters with `"ExpressRequest"` (for `req`) and `"ExpressResponse"` (for `res`). Other directives might build on these tags to define sources or sinks.

```
var app = require("express")();
app.post("/path1", function (req, res) {
  // ...
})
```

### JavaScript example 3: Directive

```
{
  "async_method" : {
    "to_callsite" : {
      "call_on" : {
        "read_path_off_global" : [ { "property" : "dbQuery" } ]
      }
    },
    "input" : "arg3"
  },
  "handler_kind" : "async_method",
}
```

```
"input_taints" : [ {
  "input" : "arg2",
  "taint_kind" : "database",
  "is_deep_taint" : true
} ]
}
```

This directive indicates that the global function `dbQuery` registers its third argument as a callback and that the second parameter of that callback is deeply tainted with a "database" taint. For example, in the code below, the anonymous function is the callback, and "data" is deep tainted with database data. that is, `data.firstName`, `data.address.city`, and so on are tainted with database data.

```
dbQuery(connectionString, "SELECT * FROM user", function (result, data) {
  // ... data.firstName ... data.address.city
});
```

### 3.1.4. class\_like\_print\_writer\_for\_servlet\_output

Supported Languages: C#, Java, and Visual Basic

The `class_like_print_writer_for_servlet_output` directive indicates classes with `print()`, `println()`, and `write()` methods that function like `PrintWriter` methods of the same name, and that should always be treated as if they were writing to a servlet output stream. The XSS checker reports a defect on tainted data that flows to a servlet output stream without proper escaping.

#### 3.1.4.1. Fields

This directive uses the following field:

```
class_like_print_writer_for_servlet_output
  Specifies a ClassSet value. See Section 3.2.6, "ClassSet".
```

#### 3.1.4.2. Examples

Configuration example:

```
/"class_like_print_writer_for_servlet_output" directive example
{
  "class_like_print_writer_for_servlet_output" :
    { "named" : "examples.LikeServletPrintWriter" }
},
```

Java code example:

```
/"class_like_print_writer_for_servlet_output" directive example

package examples;

interface LikeServletPrintWriter
{
  public void print(String s);
  public void println(String x);
```

```
    public void write(String s);
}

class Test_class_like_print_writer_for_servlet_output extends HttpServlet
{
    LikeServletPrintWriter writer;
    Locale l;
    public void doGet(HttpServletRequest request, HttpServletResponse resp)
        throws IOException
    {
        // XSS reported on 'print' 'println' and 'write' due to the directive
        // treating these calls like writing to servlet output.
        writer.print(request.getParameter("taint"));
        writer.println(request.getParameter("taint"));
        writer.write(request.getParameter("taint"));
    }
}
```

### 3.1.5. csrf\_check\_needed

Supported Languages: JavaScript only

Use the `csrf_check_needed` directive to tell the CSRF checker which function calls require CSRF protection. The CSRF checker will report a defect on any web application entry point that calls such functions without CSRF protection.

#### 3.1.5.1. Fields

This directive uses the following fields:

"csrf\_check\_needed"

Specifies a `CallSiteSet` that identifies call sites to which this directive applies.

"update\_type"

Sets a string value that specifies the type of update made to the server. Valid values include "database" and "filesystem".

#### 3.1.5.2. Examples

JavaScript example:

```
{
  "csrf_check_needed" : {
    "call_on" : {
      "read_path_off_global" : [ { "property" : "deleteDatabase" } ]
    }
  },
  "update_type" : "database"
}
```

The `csrf_check_needed` directive above matches the `deleteDatabase()` call site in this Node.js JavaScript code. This will result in a CSRF defect being reported at the web application entry point `app.get("/", function) which calls deleteDatabase()`.

```
var express = require("express");
var app = express();

app.get("/", function(req, res) {
  deleteDatabase();
});

app.listen(3000, function() {
  console.log("Listening");
});
```

### 3.1.5.3. See also

"csrf\_validator"

### 3.1.6. csrf\_validator

Supported Languages: JavaScript only

Use the `csrf_validator` directive to tell the CSRF checker which function calls protect Web application entry points from CSRF attacks; for example, by comparing a CSRF token in a user's session with the one submitted in the request. The CSRF checker does not report defects on a Web application entry point that calls one of these functions.

#### 3.1.6.1. Fields

This directive uses the following field:

```
"csrf_validator"
  Sets a CallSiteSet value that identifies call sites to which this directive applies.
```

#### 3.1.6.2. Examples

JavaScript example:

```
{
  "csrf_validator" : {
    "call_on" : {
      "read_path_off_global" : [ { "property" : "myCsrfValidator" } ]
    }
  }
}
```

The `csrf_validator` directive above matches the `myCsrfValidator()` call site in this Node.js JavaScript code. Normally, a CSRF defect would be reported at both Web application entry points `app.get("/a", function)` and `app.get("/b", function)` because they both call `db.createCollection("my_collection")`, which updates the database. However, since `app.get("/b", function)` calls `myCsrfValidator()`, no CSRF defect is reported for this Web application entry point.

```
var MongoClient = require("mongodb").MongoClient;
```

```
var express = require("express");
var app = express();

var url = "mongodb://localhost:27017/myDatabase";

app.get("/a", function(req, res) {
  MongoClient.connect(url, function(err, db) {
    console.log("Creating new database collection");
    db.createCollection("my_collection");
    res.send("Visiting /a");
  });
});

app.get("/b", function(req, res) {
  MongoClient.connect(url, function(err, db) {
    console.log("Creating new database collection");
    myCsrfValidator();
    db.createCollection("my_collection");
    res.send("Visiting /b");
  });
});

app.listen(3000, function() {
  console.log("Listening");
});
```

### 3.1.6.3. See also

"csrf\_check\_needed".

### 3.1.7. data\_has\_tag

Supported Languages: JavaScript only

The `data_has_tag` directive assigns an arbitrary string-valued `tag` to a specified piece of data; for example, to the return value of a specific function or to a specific global variable. Other structures, such as `read_from_object_with_tag` and `write_to_object_with_tag`, can refer to this piece of data using this `tag` value. Tagging this data has no other effect on the analysis: It simply enables the use of these other structures.

See Section 3.2.19.2, “`read_from_object_with_tag`” and Section 3.2.26.2, “`write_to_object_with_tag`”.

#### 3.1.7.1. Fields

This directive uses the following fields:

"data\_has\_tag"

Specifies a `ReadableProgramData` value that specifies the data to which to apply the tag. See Section 3.2.19, “`ReadableProgramData`”.

"tag"

Sets a JSON string that specifies the tag.

### 3.1.7.2. Examples

See Section 3.2.26.2, “write\_to\_object\_with\_tag” for an example.

### 3.1.8. dataflow\_checker\_name

Supported Languages: C#, Java, JavaScript, and Visual Basic

The `dataflow_checker_name` directive defines a `DF.CUSTOM_CHECKER`.

#### 3.1.8.1. Fields

The custom dataflow checker description uses the following fields:

`"dataflow_checker_name"`

Specifies a JSON string that names the custom checker. This name must begin with `"DF. "`. After that prefix, it must contain only capital letters or the underscore character (`_`). For example, `"DF.MY_CHECKER"` is allowable, but `"DF.My_Checker"` is not.

`"taint_kinds"`

Specifies a `TaintKindGroup` value that defines the kinds of taint that the checker tracks (subject to the global trust options; see Section 3.2.23, “TaintKind” for details).

See also Section 3.2.24, “TaintKindGroup”.

`"sink_message"`

Sets a JSON string to print as the event message where the tainted data flows into the sink. For defects in JavaScript code, this string appears in an event after several other events that describe the sink and the tainted data that flowed into it. For Java and C# checkers, you can use the following two placeholder values in this string:

`{0}`

This substring will be replaced by the name of the tainted expression that reached the sink.

`{1}`

This substring will be replaced by the name of the sink.

`"remediation_advice"`

Specifies a JSON string to print as remediation advice in each defect report.

`"new_issue_type"`

(Optional) Sets an `IssueTypeDefinition` value to describe the sorts of issues that this checker reports. See Section 3.2.12, “IssueTypeDefinition”. The fields of this `IssueTypeDefinition` object are all optional in this context. Missing fields default to the following values:

- `"type": "USER. "` followed by the name of your checker
- `"name": "Tainted data reached a sink."`
- `"description": "User-controllable data reached a sink."`
- `"local_effect": "Custom Dataflow Checker"`

- "impact": "Medium"
- "category": "Medium impact security"
- "quality\_kind": "false"
- "security\_kind": "true"

 **Note**

The `new_issue_type` field replaces the deprecated `checker_properties` field. If `dataflow_checker_name` specifies neither `new_issue_type` nor `checker_properties`, then *all* the default values listed above are used.

### 3.1.8.2. Deprecated fields—from prior to format version 8

As of Security Configuration format version 8, the fields described in this section are deprecated and have been replaced with the `new_issue_type` field; future Security Configuration format versions are not guaranteed to support them.

`"local_effect", "impact", "category", "cwe"`

These fields are deprecated. See the documentation of the correspondingly named fields in Section 3.2.12, “IssueTypeDefinition”.

`"long_description"`

This field is deprecated. See the documentation for the `"description"` field in Section 3.2.12, “IssueTypeDefinition”.

`"type"`

This field is deprecated. See the documentation for the `"name"` field in Section 3.2.12, “IssueTypeDefinition”. (Be aware: This deprecated `"type"` field is unrelated to the `"type"` field of `IssueTypeDefinition`.)

`"kind"`

This field is deprecated. See the documentation for the `"quality_kind"` and `"security_kind"` fields in Section 3.2.12, “IssueTypeDefinition” and the discussion of migration in the following section. Valid values for this property are: `"security"` (for security issues), `"quality"` (for quality issues), or `"both"` (for security and quality issues).

#### 3.1.8.2.1. Migrate the format from version 8 to version 12

To migrate away from using the deprecated fields to the `new_issue_type` field that replaces them, proceed as follows:

1. Ensure that your directives file has a `"format_version"` of 8 or greater.
2. Add a `"new_issue_type"` field containing a JSON object to your checker definition.
3. Move your `"category"`, `"cwe"`, `"impact"`, `"local_effect"`, `"long_description"`, and `"type"` fields into this new object, but rename `"long_description"` to `"description"` and

rename "type" to "name". If you omitted any of these fields (and thus used the default values), there's no need to create them: The defaults remain the same.

4. Set the "quality\_kind" and "security\_kind" fields of "new\_issue\_type" according to your old "kind" field.
  - "kind": "quality" translates to "quality\_kind": "true", "security\_kind": "false"
  - "kind": "security" translates to "quality\_kind": "false", "security\_kind": "true"
  - "kind": "both" translates to "quality\_kind": "true", "security\_kind": "true"
5. Remove your old "kind" field.
6. Optionally, add a "type" field of "new\_issue\_type".

### 3.1.9. dataflow\_through\_callsite

Supported Languages: JavaScript only

The `dataflow_through_callsite` directive tells the analysis how data flows from arguments to return values or to other function outputs for calls to a particular function.

#### 3.1.9.1. Fields

This directive uses the following fields:

"dataflow\_through\_call\_site"

Specifies a `CallSiteSet` that indicates the function call sites to which this directive applies. See Section 3.2.3, "CallSiteSet".

"from"

Specifies a non-empty JSON array of `InputAndAccessPathSpecifier` values that indicate the inputs to the function that flow to the outputs specified in the "to" field. See Section 3.2.8, "InputAndAccessPathSpecifier".

"to"

Specifies a non-empty JSON array of `OutputAndAccessPathSpecifier` values that indicate the outputs of the function that correspond to the inputs specified in the "from" field. See Section 3.2.16, "OutputAndAccessPathSpecifier".

### 3.1.10. dc\_checker\_name

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

The `dc_checker_name` directive defines a `DC.CUSTOM_CHECKER`.



#### Note

As of version 2020.03, if you need to migrate from the legacy checker `SECURE_CODING`, we recommend that you use `CodeXM` instead of creating a new `DC.CUSTOM_CHECKER`. We also

recommend that you migrate custom DC checkers to CodeXM code. See “Migrate DC Custom Checkers to CodeXM” in the *Checker Reference*.

### 3.1.10.1. Fields

In a configuration file, two directives manage custom DC checkers:

`"dc_checker_name"`

Defines a new DC checker and specifies the checker name; for example, `"dc_checker_name" : "DC.CUSTOM_MY_CHECKER"`.

`"method_set_for_dc_checker"`

Adds methods to a DC custom checker. Use this directive when you define a new DC checker, or to add methods to an existing DC checker. For a description, see Section 3.1.22, “`method_set_for_dc_checker`”.

Two other fields can be present:

`"new_issue_type"`

(Optional) Specifies an `IssueTypeDefinition` value that describes the sort of issues that this checker reports. See Section 3.2.12, “`IssueTypeDefinition`”.

In the context of `"new_issue_type"`, all of these fields are optional. If a field is not present, its value defaults to the value shown in the following list:

- `"type": "USER. "` followed by your custom checker name
- `"name": "Calling risky function."`
- `"description": "The called function is unsafe for security related code."`
- `"local_effect": "May result in a security violation."`
- `"impact": "Low"`
- `"category": "Security best practices violations"`
- `"quality_kind": false`
- `"security_kind": true`
- `"cwe": 676`

 **Note**

The `new_issue_type` field replaces the deprecated fields `"category"`, `"cwe"`, `"impact"`, `"kind"`, `"local_effect"`, `"long_description"`, and `"type"`. If both the `"new_issue_type"` field and the subfield it replaces are absent, all of the default values listed above are used.

"`antecedent_checker`"

A string that names the custom DC checker (or `SECURE_CODING` checker) on which the present checker is based.

Rather than use this field, unless you need to maintain legacy code we recommend that you use CodeXM to write new Don't Call checkers. See "Migrated DC Custom Checkers to CodeXM" in the *Checker Reference*.

The `antecedent_checker` field preserved existing triage results for the specified `DC.CUSTOM_NAME` checker in Coverity Connect. There were two use cases for this property:

- When migrating from `SECURE_CODING` to a `DC.CUSTOM_*` checker, you would specify `SECURE_CODING`.
- When renaming a `DC.CUSTOM_*` checker, you would specify the old name of the checker.

### 3.1.10.2. Deprecated fields—from prior to format version 8

As of Security Configuration format version 8, the fields described in this section are deprecated and have been replaced with the "`new_issue_type`" field; future Security Configuration format versions are not guaranteed to support them.

"`local_effect`", "`impact`", "`category`", "`cwe`"

These fields are deprecated. See the documentation of the correspondingly named fields in Section 3.2.12, "IssueTypeDefinition".

"`long_description`"

This field is deprecated. See the documentation for the "`description`" field in Section 3.2.12, "IssueTypeDefinition".

"`type`"

This field is deprecated. See the documentation for the "`name`" field in Section 3.2.12, "IssueTypeDefinition". (Be aware: This deprecated "`type`" field is unrelated to the "`type`" field of `IssueTypeDefinition`.)

"`kind`"

This field is deprecated. See the documentation for the "`quality_kind`" and "`security_kind`" fields in Section 3.2.12, "IssueTypeDefinition" and the discussion of migration in the following section. Valid values for this property are: "`security`" (for security issues), "`quality`" (for quality issues), or "`both`" (for security and quality issues).

#### 3.1.10.2.1. Migrate the format from version 8 to version 12

We don't recommend that you follow these steps unless it is necessary to support legacy code. See "Migrate DC Custom Checkers to CodeXM" in the *Checker Reference*.

To migrate away from using the deprecated fields to the "`new_issue_type`" field that replaces them, proceed as follows:

1. Ensure that your directives file has a "`format_version`" of 8 or greater.

2. Add a "new\_issue\_type" field containing a JSON object to your checker definition.
3. Move your "category", "cwe", "impact", "local\_effect", "long\_description", and "type" fields into this new object, but rename "long\_description" to "description" and rename "type" to "name". If you omitted any of these fields (and thus used the default values), there's no need to create them: the defaults remain the same.
4. Set the "quality\_kind" and "security\_kind" fields of "new\_issue\_type" according to your old "kind" field.
  - "kind": "quality" translates to "quality\_kind": true, "security\_kind": false
  - "kind": "security" translates to "quality\_kind": false, "security\_kind": true
  - "kind": "both" translates to "quality\_kind": true, "security\_kind": true
5. Remove your old "kind" field.
6. Optionally, add a "type" field of "new\_issue\_type".

### 3.1.11. define\_lookup\_method\_call\_map

Supported Languages: C#, Java, and Visual Basic

The `define_lookup_method_call_map` directive defines a map that can be shared across many `MethodCallSpecifier` objects in other directives. In particular, the "lookup\_by\_constant\_param" variant of `MethodCallSpecifier` can refer to this map by name.

See Section 3.2.13, "MethodCallSpecifier" and Section 3.2.13.2, "lookup\_by\_constant\_param".

#### 3.1.11.1. Fields

This directive uses the following fields:

"define\_lookup\_method\_call\_map"

A JSON string value that names the map defined by this directive.

"map"

A JSON object that consists of a series of fields, to be interpreted as follows:

- The name of each field is a lexical expression string that is mapped to the value of the field.

Valid lexical expression strings are described in the section Section 3.2.13.2, "lookup\_by\_constant\_param" value.

- The value of the field must be either a "method\_call" `MethodCallSpecifier` value or a JSON `null` literal. See Section 3.2.13.1, "method\_call".

#### 3.1.11.2. Examples

Configuration example:

See Configuration example 3 for `method_with_servlet_sinks_on_input` [p. 39 ] and Configuration example 4 for `method_with_servlet_sinks_on_input` [p. 41 ].

Java code example:

See Java code example 3 for `method_with_servlet_sinks_on_input` [p. 40 ] and Java code example 4 for `method_with_servlet_sinks_on_input` [p. 42 ].

### 3.1.12. ignore\_all\_argument\_dataflow\_to\_method

Supported Languages: C#, Java, and Visual Basic

The `ignore_all_argument_dataflow_to_method` directive applies to call sites that match a specified `MethodSet` value. See Section 3.2.15, “MethodSet”.

The dataflow analysis ignores paths from the call site arguments to parameters of the called method. The analysis also ignores any changes the method call appears to make to modifiable arguments.

Effectively, the dataflow analysis act as if the call site does not exist for the arguments, but the analysis is still capable of reporting paths within the called method.

#### 3.1.12.1. Fields

This directive uses the following field:

```
"ignore_all_argument_dataflow_to_method"
  Specifies a MethodSet value that identifies the methods whose argument dataflow will be ignored.
  See Section 3.2.15, “MethodSet”
```

#### 3.1.12.2. Examples

Configuration example:

```
//"ignore_all_argument_dataflow_to_method" directive example
{
  "ignore_all_argument_dataflow_to_method" :
  { "named" :
    "examples.Test_ignore_all_argument_dataflow_to_method.appendAndPrintString(
      java.lang.StringBuffer, java.lang.String,
      javax.servlet.http.HttpServletResponse)void"
  }
},
```

Java code example:

```
//"ignore_all_argument_dataflow_to_method" directive example
package examples;

class Test_ignore_all_argument_dataflow_to_method extends HttpServlet
```

```

{
  public void appendAndPrintString(StringBuffer sb,
  String str,
  HttpServletResponse resp)
  {
    sb.append(str);
    PrintWriter pw = resp.getWriter();
    //no XSS because the directive suppresses taint flow from callers into 'str'
    pw.println(str);
  }

  public void doGet(HttpServletRequest request, HttpServletResponse resp)
  throws IOException
  {
    PrintWriter pw = resp.getWriter();

    String taint = request.getParameter("taint");

    StringBuffer sb = new StringBuffer();
    appendAndPrintString(sb, taint, resp);

    //no XSS due to the directive
    pw.println(sb.toString());
  }
}

```

### 3.1.13. ignore\_method\_dataflow

Supported Languages: C#, Java, and Visual Basic

The `ignore_method_dataflow` directive indicates methods where the analysis should ignore all dataflow paths within the method. Dataflow paths added by the `method_returns_param` directive are not ignored.

See Section 3.1.19, “`method_returns_param`”.

#### 3.1.13.1. Fields

This directive uses the following field:

```

"ignore_method_dataflow"
  Specifies a MethodSet value that identifies the methods whose dataflow will be ignored. See
  Section 3.2.15, “MethodSet”.

```

#### 3.1.13.2. Examples

Configuration example 1:

```

//"ignore_method_dataflow" directive example 1
{
  "ignore_method_dataflow" :

```

## Configuration file usage

---

```
{ "named" :
  "examples.Test_ignore_method_dataflow1.getTaint(
    javax.servlet.http.HttpServletRequest,
    javax.servlet.http.HttpServletResponse)java.lang.String"
},
```

### Java code example 1:

```
//"ignore_method_dataflow" directive example 1

package examples;

class Test_ignore_method_dataflow1 extends HttpServlet
{
    boolean beSafe;

    // The directive suppresses all dataflow through this function.
    public String getTaint(HttpServletRequest request, HttpServletResponse resp)
    {
        if (beSafe) return "";

        PrintWriter pw = resp.getWriter();
        String taint = request.getParameter("taint");
        pw.println(taint); //no XSS due to directive

        return taint; // the directive squelches this tainted dataflow
    }

    public void doGet(HttpServletRequest request, HttpServletResponse resp)
        throws IOException
    {
        PrintWriter pw = resp.getWriter();
        String x = getTaint(request, resp); // untainted because of the directive
        pw.println(x); //no XSS due to directive
    }
}
```

### Configuration example 2:

```
//"ignore_method_dataflow" directive example 2

{
  "ignore_method_dataflow" :
  { "named" :
    "examples.Test_ignore_method_dataflow2.manyPaths(java.lang.String,
      java.lang.StringBuffer)java.lang.String"
  }
},
```

### Java code example 2:

```
//"ignore_method_dataflow" directive example 2
```

```
package examples;

class Test_ignore_method_dataflow2 extends HttpServlet
{
    String field1;
    String field2;

    public void setField2(String str) {
        field2 = str;
    }

    // This method demonstrates several kinds of dataflow paths that the directive
    // suppresses.
    public String manyPaths(String str, StringBuffer sb) {
        field1 = str;
        setField2(str);
        sb.append(str);
        return str;
    }

    public void doGet(HttpServletRequest request, HttpServletResponse resp)
        throws IOException
    {
        PrintWriter pw = resp.getWriter();
        String taint = request.getParameter("taint");
        StringBuffer sb = new StringBuffer();

        // the directive suppresses all dataflow through manyPaths
        String ret = manyPaths(taint, sb);

        pw.println(ret); //no XSS due to directive
        pw.println(sb); //no XSS due to directive
        pw.println(field1); //no XSS due to directive
        pw.println(field2); //no XSS due to directive
    }
}
```

### 3.1.14. ignore\_method\_output

Supported Languages: C#, Java, and Visual Basic

The `ignore_method_output` directive indicates methods where the analysis should ignore dataflow paths passing out of the method through the return value or a particular modified parameter, as specified by the "output" field.

This directive rarely needs to be used, but it can be useful in cases where the analysis infers incorrect data flow through a method. This directive *does not* suppress defect reports within the methods it indicates, only those that rely on flow through the indicated method *outputs*.

#### 3.1.14.1. Fields

This directive uses the following fields:

"ignore\_method\_output"

Specifies a MethodSet value that identifies the methods whose output will be ignored. See Section 3.2.15, "MethodSet".

"output"

A ParamOut value that specifies the value to ignore. See Section 3.2.18, "ParamOut".

### 3.1.14.2. Examples

Configuration example:

```
//"ignore_method_output" directive example
{
  "ignore_method_output" :
    { "named" :

      "examples.Test_ignore_method_output.getTaint( javax.servlet.http.HttpServletRequest,
        javax.servlet.http.HttpServletResponse) java.lang.String"
    },
  "output" : "return"
},
```

Java code example:

```
//"ignore_method_output" directive example
package examples;

class Test_ignore_method_output extends HttpServlet
{
  boolean beSafe;

  // The directive suppresses dataflow through the return value of this method.
  public String getTaint(HttpServletRequest request, HttpServletResponse resp)
  {
    PrintWriter pw = resp.getWriter();
    String taint = request.getParameter("taint");
    pw.println(taint); //XSS reported here unaffected by the directive

    if (beSafe) return "";

    return taint; // the directive squelches this tainted dataflow
  }

  public void doGet(HttpServletRequest request, HttpServletResponse resp)
    throws IOException
  {
    PrintWriter pw = resp.getWriter();
    String x = getTaint(request, resp); // untainted because of the directive
    pw.println(x); //no XSS due to directive
  }
}
```

```
}

```

### 3.1.15. local\_callback

Supported Languages: JavaScript only

The `local_callback` directive identifies a callback function that is called immediately (usually by some API) and provides details about the arguments with which the function is called. This directive is similar to the `async_method` directive, but it deals with callbacks that are called immediately, as opposed to asynchronously.

#### 3.1.15.1. Fields

This directive uses the following fields:

`"local_callback"`

Specifies `"to_callsite"` `WritableProgramData` value that identifies the callback function that is called immediately. See Section 3.2.26, “WritableProgramData” and Section 3.2.26.1, “to\_callsite”.

`"input_tags"`

(Optional) Specifies a non-empty JSON array of `InputTag` values that identify callback functions to indicate that particular parameters to the callback function have particular tags. See Section 3.2.10, “InputTag”.

`"input_taints"`

(Optional) Specifies a non-empty JSON array of `InputValue` values that describe how arguments at the call site that registers the callback (that is, the call site specified by `local_callback`) flow to parameters of the callback. See Section 3.2.11, “InputValue”.

#### 3.1.15.2. Examples

JavaScript Example:

The following directive indicates that passing a function as the first argument of `doCallWithArg()` invokes it immediately and passes the second argument of `doCallWithArg()` to its first argument.

```
{
  "local_callback" : {
    "to_callsite" : {
      "call_on" : {
        "read_path_off_global" : [{ "property" : "doCallWithArg" }],
      }
    },
    "input" : "arg1",
  },
  "input_values" : [
    {
      "value" : "arg2",
      "input" : "arg1"
    }
  ]
}
```

```
}

```

For example, because of the directive, the analysis sees the call to `doCallWithArg(callback, x)` as making the following function call: `callback(x)`.

```
function callback(arg) {
  // arg === x
}
doCallWithArg(callback, x);

```

### 3.1.16. map\_read

Supported Languages: JavaScript only

The `map_read` directive indicates that a function call acts like a property *read* where one of its arguments is the object whose property is read and another is the name of the property. The return value of the function is the result of the property *read*.

#### 3.1.16.1. Fields

This directive uses the following fields:

"map\_read"

Specifies a `CallSiteSet` that identifies function call sites to which this directive applies. See Section 3.2.3, "CallSiteSet".

"map"

Specifies a `ParamIn` value that indicates which argument is the object whose property is being read. See Section 3.2.17, "ParamIn".

"key"

Specifies a `ParamIn` value that indicates which argument is the name of the property that is read from "map". See Section 3.2.17, "ParamIn".

This directive only applies if the argument indicated by `key` is a string literal.

#### 3.1.16.2. Examples

JavaScript example:

The following directive indicates that `localStorage.getItem(obj, "prop")` reads property "prop" from `localStorage`, just as `localStorage.prop` would.

```
{
  "map_read" : {
    "call_on" : {
      "read_path_off_global" : [
        { "property" : "localStorage" },
        { "property" : "getItem" }
      ]
    }
  },
}
```

```
"map" : "this",
"key" : "arg1"
}
```

### 3.1.17. map\_write

Supported Languages: JavaScript only

The `map_write` directive indicates that a function call acts like a property write where one of its arguments is the object whose property is written, another is the name of the property, and a third is the value to write to that property.

#### 3.1.17.1. Fields

This directive uses the following fields.

"map\_write"

Specifies a `CallSiteSet` that identifies function call sites to which this directive applies. See Section 3.2.3, "CallSiteSet".

"map"

Specifies a `ParamIn` value that indicates which argument is the object whose property is being written. See Section 3.2.17, "ParamIn".

"key"

Specifies a `ParamIn` value that indicates which argument is the property of "map" that is being overwritten. See Section 3.2.17, "ParamIn".

This directive only applies if the argument indicated by "key" is a string literal.

"value"

Specifies a `ParamIn` value that indicates which argument is the value being written. See Section 3.2.17, "ParamIn".

#### 3.1.17.2. Examples

The following directive indicates that `localStorage.setItem(obj, "prop", value)` writes value to property "prop" of `localStorage` just as `localStorage.prop = value` would.

```
{
  "map_write" : {
    "call_on" : {
      "read_path_off_global" : [
        { "property" : "localStorage" },
        { "property" : "setItem" }
      ]
    }
  },
  "map" : "this",
  "key" : "arg1",
  "value" : "arg2"
},
```

### 3.1.18. method\_returns\_constant

Supported Languages: C#, Java, and Visual Basic

The `method_returns_constant` directive specifies a constant for a method to return.

In a program where dataflow follows an unwanted conditional path (for example, because you are certain the path is impossible in a production environment), the unwanted path can be avoided by modelling a method evaluated in the conditional expression as returning a constant value.

#### 3.1.18.1. Fields

This directive uses the following fields:

"method\_returns\_constant"

Specifies a `MethodSet` value to identify the methods to which this directive applies. See Section 3.2.15, "MethodSet".

"returns"

A `ReturnConstant` value to be returned by the identified methods. See Section 3.2.21, "ReturnConstant".

#### 3.1.18.2. Examples

Configuration example:

```
//"method_returns_constant" directive example
{
  "method_returns_constant" :
  { "named" :
    "examples.Test_method_returns_constant.check_for_error()boolean"
  },
  "returns" : { "bool" : false }
},
```

Java code example:

```
//"method_returns_constant" directive example
package examples;

class Test_method_returns_constant extends HttpServlet
{
  boolean hasError;
  boolean check_for_error() { return hasError; }
  public void doGet(HttpServletRequest request, HttpServletResponse resp)
  throws IOException
  {
    PrintWriter pw = resp.getWriter();
    String taint = request.getParameter("taint");

    if (check_for_error()) {
      pw.println(taint); //no XSS due to directive
    }
  }
}
```

```

    }
  }
}

```

### 3.1.19. method\_returns\_param

Supported Languages: C# , Java, and Visual Basic

The `method_returns_param` directive specifies a particular parameter for a method to return.

This directive indicates methods where the analysis should follow dataflow paths as if the method directly returned the specified parameter. This directive is useful when the analysis fails to infer dataflow from a method parameter to its return value.

#### 3.1.19.1. Fields

This directive uses the following fields:

`"method_returns_param"`

Specifies a `MethodSet` value to identify the methods to which this directive applies. See Section 3.2.15, “`MethodSet`”.

`"input"`

A `ParamIn` value to be returned by the identified methods. See Section 3.2.17, “`ParamIn`”.

#### 3.1.19.2. Examples

Configuration example:

```

// "method_returns_param" directive example

{
  "method_returns_param" :
  { "named" :

    "examples.Test_method_returns_param.example1(java.lang.String)java.lang.String"
  },
  "input" : "arg1"
},

{
  "method_returns_param" :
  { "named" :
    "examples.Test_method_returns_param.example2(java.lang.String,
    java.lang.String)java.lang.String"
  },
  "input" : "arg2"
},

{
  "ignore_method_dataflow" :
  { "named" :
    "examples.Test_method_returns_param.example2(java.lang.String,

```

```

        java.lang.String) java.lang.String"
    }
},

```

#### Java code example:

```

// "method_returns_param" directive example

package examples;

class Test_method_returns_param extends HttpServlet
{
    HttpServletResponse resp;

    // The directive adds a dataflow path where this method returns 'str'.
    public String example1(String str) {
        PrintWriter pw = resp.getWriter();
        pw.println(str); // XSS reported here is unaffected by the directive
        return "";
    }

    // The "ignore_method_dataflow" directive ignores the original dataflow and
    // the "method_returns_param" directive adds back a dataflow edge where the
    // method returns 'str2'. Together these directives replace the inferred
    // dataflow with something entirely new.
    public String example2(String str1, String str2) {
        PrintWriter pw = resp.getWriter();
        pw.println(str1); // no XSS due to ignore_method_dataflow
        return str1; // ignore_method_dataflow squelches this dataflow path
    }

    public void doGet(HttpServletRequest request, HttpServletResponse resp)
        throws IOException
    {
        PrintWriter pw = resp.getWriter();
        String taint = request.getParameter("taint");

        // XSS: method returns 'taint' due to 'method_returns_param' directive
        pw.println( example1(taint) );

        // no XSS: first argument no longer returned due to 'ignore_method_dataflow'
        pw.println( example2(taint, "") );
        // XSS: second argument returned due to 'method_returns_param' directive
        pw.println( example2("", taint) );
    }
}

```

### 3.1.20. method\_returns\_servlet\_output\_stream

Supported Languages: C#, Java, and Visual Basic

The `method_returns_servlet_output_stream` directive indicates that a method returns a stream that writes data to the HTTP output. The XSS checker (for cross-site scripting) reports a defect if tainted data is written to the stream without proper escaping.

In Java, the returned object type should extend the `java.io.OutputStream` or `java.io.Writer` classes. In C#, the returned object type should extend the `System.IO.Stream` or `System.IO.TextWriter` classes.

### 3.1.20.1. Fields

This directive uses the following field:

"method\_returns\_servlet\_output\_stream"  
Specifies a `MethodSet` value that identifies the methods to which this directive applies. See Section 3.2.15, "MethodSet".

### 3.1.20.2. Examples

Configuration example:

```
// "method_returns_servlet_output_stream" directive example
{
  "method_returns_servlet_output_stream" :
  { "named" :
    "examples.Test_method_returns_servlet_output_stream.getWriter()
    java.io.PrintWriter"
  }
},
```

Java code example:

```
// "method_returns_servlet_output_stream" directive example
package examples;

class Test_method_returns_servlet_output_stream extends HttpServlet
{
  PrintWriter pwField;
  PrintWriter getServletWriter() { return pwField; }
  public void doGet(HttpServletRequest request, HttpServletResponse resp)
    throws IOException
  {
    PrintWriter pw = getServletWriter();
    String taint = request.getParameter("taint");
    pw.println(taint); //XSS defect due to directive
  }
}
```

### 3.1.21. method\_returns\_tainted\_data

Supported Languages: C#, Java, and Visual Basic

The `method_returns_tainted_data` directive identifies methods that return tainted data. The returned data should extend or implement a built-in taintable type, such as a string, byte array, input stream, or collection. It cannot be used to indicate that members of a user-defined class instance are tainted. The current trust model and trust options control whether the type of taint should be distrusted.

### 3.1.21.1. Fields

This directive uses the following fields:

"method\_returns\_tainted\_data"

Specifies a `MethodSet` value that identifies the methods to which this directive applies. See Section 3.2.15, "MethodSet".

"taint\_kind"

A `TaintKind` string value to be returned by the identified methods. See Section 3.2.23, "TaintKind".

### 3.1.21.2. Examples

Configuration example:

```
// "method_returns_tainted_data" example
{
  "method_returns_tainted_data" : {
    "matching": "examples\\.Test_method_returns_tainted_data\\
\\.returns_tainted_data\\(.*)"
  },
  "taint_kind" : "http"
}
```

Java code example:

```
package examples;
import java.sql.Statement;
import java.sql.Connection;

public class Test_method_returns_tainted_data {

    Connection connection;
    Statement statement;

    String returns_tainted_data() {
        return "foo";
    }

    void test_SQLI() throws Exception {
        String val = returns_tainted_data();

        // The method call to returns_tainted_data is considered to return
        // tainted data of "http" type.

        String sqlQuery1 = "select * from " + val;

        // An SQLI defect is reported on the following line
        statement = connection.prepareStatement(sqlQuery1);
    }
}
```

### 3.1.22. method\_set\_for\_dc\_checker

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

The `method_set_for_dc_checker` directive adds a method to a `DC.CUSTOM_CHECKER`. You can use this directive to specify the method initially tested by the new DC custom checker. You can also use it to add methods to an existing `DC.CUSTOM_CHECKER`.

 **Note**

We recommend that you use CodeXM to develop custom “don’t call” checkers (which in releases before 2020.03 had to be implemented using `DC.CUSTOM_CHECKER` directives). See “Migrate DC Custom Checkers to CodeXM” in the *Checker Reference*.

### 3.1.22.1. Fields

A method set entry must contain a pair of directive fields, `method_set_for_dc_checker` and `methods`. A couple other fields can also be present.

`method_set_for_dc_checker`

Names the custom checker to which the methods will be added; for example,

`method_set_for_dc_checker` : `DC.CUSTOM_MY_CHECKER`.

`methods`

A named `MethodSet` value that identifies the method to add to the method set; for example,

`methods` : { `named` : `strcmp` }, where `strcmp()` is the method to check. See Section 3.2.15.1, “`named`”.

`txt_defect_message`

(Optional) Specifies a string to display when the checker finds an issue. This string should describe the issue.

`txt_remediation_advice`

(Optional) Specifies a string to display when the checker finds an issue. This string should describe how to avoid the issue.

### 3.1.23. `method_with_servlet_sinks_on_input`

Supported Languages: C# , Java, and Visual Basic

The `method_with_servlet_sinks_on_input` directive indicates that a method’s argument is written to the HTTP output. The XSS (cross-site scripting) checker reports a defect if tainted data is written to the HTTP output without proper escaping.

#### 3.1.23.1. Fields

This directive uses the following fields:

`method_with_servlet_sinks_on_input`

Specifies a `MethodSet` that identifies the methods to which this directive will be applied. See Section 3.2.15, “`MethodSet`”.

`input_param_sinks`

Specifies a JSON array. Each object in this array describes an argument that the method writes to the HTTP output, and how that argument is escaped.

Objects in the `input_param_sinks` array use the following fields:

"input"

A `ParamIn` value that names the argument that this object describes. See Section 3.2.17, "ParamIn".

"escaper"

Either a `MethodCallSpecifier` value or a JSON `null` literal. See Section 3.2.13, "MethodCallSpecifier".

If this `escaper` field is the `null` literal, or if it evaluates to `null`, then the "input" is written to the HTTP output as-is and without any escaping. Otherwise, the field indicates a method: The method's "input" is where input is passed in, and the method's "output" is written to the servlet output stream.

"servlet\_context"

Specifies an `HtmlOutputContext` value (see Section 3.2.7, "HtmlOutputContext").

This field indicates the HTML context (that is, the place in the HTML parse tree) into which the argument flows. To avoid cross-site scripting (XSS), different contexts imply different escaping obligations.

### 3.1.23.2. Examples

Configuration example 1:

```
//"method_with_servlet_sinks_on_input" directive example 1

// This example also demonstrates using the "html_prefix" HtmlOutputContext
// value to control the context.

{
  "method_with_servlet_sinks_on_input" :
    { "named" :

      "examples.Test_method_with_servlet_sinks_on_input1.pdata_sink(java.lang.String)void"
    },
  "input_param_sinks" : [
    {
      "input" : "arg1",
      "escaper" : null,
      "servlet_context" : { "html_prefix" : "" }
    }
  ]
},

{
  "method_with_servlet_sinks_on_input" :
    { "named" :
      "examples.Test_method_with_servlet_sinks_on_input1
      .single_quoted_attribute_value_sink(java.lang.String)void"
    },
  "input_param_sinks" : [
    {
      "input" : "arg1",
```

## Configuration file usage

---

```
    "escaper" : null,
    "servlet_context" : { "html_prefix" : "<tag foo='" }
  }
]
},
```

### Java code example 1:

```
/"method_with_servlet_sinks_on_input" directive example 1

// This example also demonstrates using the "html_prefix" HtmlOutputContext
// value to control the context.

package examples;

class Test_method_with_servlet_sinks_on_input1 extends HttpServlet
{
    public void pcd_data_sink(String val) {}

    public void single_quoted_attribute_value_sink(String val) {}

    public void doGet(HttpServletRequest request, HttpServletResponse resp)
        throws IOException
    {
        PrintWriter pw = resp.getWriter();

        String taint = request.getParameter("taint");

        // The directive makes the analysis treat the argument to this function as
        // being written to servlet output in the HTML PCDATA context, so we get an
        // XSS defect here.
        pcd_data_sink(taint);

        // The directive makes the analysis treat the argument to this function as
        // being written to servlet output in the single-quoted HTML tag value
        // context, so we get an XSS defect here.
        single_quoted_attribute_value_sink(taint);
    }
}
```

### Configuration example 2:

```
/"method_with_servlet_sinks_on_input" directive example 2

// This also demonstrates using the "html_attribute_value_where_name_is_from_param"
// HtmlOutputContext value to control the context.

{
    "method_with_servlet_sinks_on_input" :
    { "named" :
        "examples.Test_method_input_servlet_sinks2.sink(java.lang.String,
        java.lang.String)void"
    },
    "input_param_sinks" : [
```

## Configuration file usage

---

```
{
  "input" : "arg2",
  "escaper" : null,
  "servlet_context" : {
    "html_attribute_value_where_name_is_from_param" : "arg1",
    "value_quoting" : "single"
  }
}
],
},
```

### Java code example 2:

```
// "method_with_servlet_sinks_on_input" directive example 2

// This example also demonstrates using the
// "html_attribute_value_where_name_is_from_param" HtmlOutputContext value to
// control the context.

package examples;

class Test_method_input_servlet_sinks2 extends HttpServlet
{
  String unknownName;

  public void sink(String name, String val) {}

  public void doGet(HttpServletRequest request, HttpServletResponse resp)
    throws IOException
  {
    PrintWriter pw = resp.getWriter();

    String taint = request.getParameter("taint");

    // The directive makes the analysis treat 'taint' a being written to the
    // servlet output as the single-quoted value to a 'color' attribute, for
    // example:
    // "<font color='" + taint + ...
    // Thus the directive causes an XSS report here.
    sink("color", taint);

    // Similar to the above, but here it's an "onclick" single-quoted JavaScript
    // attribute value. Again the directive causes an XSS report here.
    sink("onclick", taint);

    // Here we have something other than a String literal for the attribute
    // name, so the analysis treats it as the 'color' case above (including
    // reporting an XSS defect) and logs a warning.
    sink(unknownName, taint);
  }
}
```

### Configuration example 3:

## Configuration file usage

---

```
//"method_with_servlet_sinks_on_input" directive example 3

// This example also demonstrates using a "lookup_by_constant_param"
// MethodCallSpecifier value to indicate that a boolean parameter controls an
// optional escaper.

{
  "define_lookup_method_call_map" : "escape_if_bool_is_true",
  "map" : {
    "true" : {
      "method_call" :
        "Escapers.escape_html(java.lang.String)java.lang.String",
      "input" : "arg1", "output" : "return"
    },
    "false" : null
  }
},

{
  "method_with_servlet_sinks_on_input" :
    { "named" :
      "examples.Test_method_input_servlet_sinks3.sink(java.lang.String,
boolean)void"
    },
  "input_param_sinks" : [
    {
      "input" : "arg1",
      "escaper" : {
        "lookup_by_constant_param" : "arg2",
        "lookup_map" : "escape_if_bool_is_true"
      },
      "servlet_context" : { "html_prefix" : "" }
    }
  ]
},
},
```

### Java code example 3:

```
//"method_with_servlet_sinks_on_input" directive example 3

// This also demonstrates using a "lookup_by_constant_param" MethodCallSpecifier
// value to indicate that a boolean parameter controls an optional escaper.

// NOTE: This example should include the Escapers.java code (for the
// 'escape_html' method call added by the directive).

package examples;

class Test_method_input_servlet_sinks3 extends HttpServlet
{
  boolean unknownBool;

  public void sink(String val, boolean escape) {}
}
```

## Configuration file usage

---

```
public void doGet(HttpServletRequest request, HttpServletResponse resp)
    throws IOException
{
    PrintWriter pw = resp.getWriter();

    String taint = request.getParameter("taint");

    // The directive makes the analysis (1) treat the first argument to 'sink'
    // ('taint') as if it is written to HTML PCDATA context; and also (2) if the
    // second argument to 'sink' is 'true', the analysis assumes the first
    // argument has been passed through 'escape_html' first.

    // No XSS because the 'escape_html' makes 'taint' safe for HTML PCDATA.
    sink(taint, true);

    // XSS report because 'false' implies no escaping of 'taint'
    sink(taint, false);

    // Since the second argument is not a boolean literal ('true' or 'false'),
    // the analysis does not know if the first argument is escaped. It logs a
    // warning, but does not report a defect.
    sink(taint, unknownBool);
}
}
```

### Configuration example 4:

```
// "method_with_servlet_sinks_on_input" directive example 4

// This example also demonstrates using a "lookup_by_constant_param"
// MethodCallSpecifier value to indicate that an enum parameter controls an
// optional escaper.

{
    "define_lookup_method_call_map" : "escape_if_Choice_is_YES",
    "map" : {
        "examples.Choice.YES" : {
            "method_call" :
                "Escapers.escape_html(java.lang.String)java.lang.String",
            "input" : "arg1", "output" : "return"
        },
        "examples.Choice.NO" : null,
        "null" : null
    }
},

{
    "method_with_servlet_sinks_on_input" :
        { "named" :
            "examples.Test_method_input_servlet_sinks4.sink(java.lang.String,
examples.Choice)void"
        },
}
```

## Configuration file usage

---

```
"input_param_sinks" : [
  {
    "input" : "arg1",
    "escaper" : {
      "lookup_by_constant_param" : "arg2",
      "lookup_map" : "escape_if_Choice_is_YES"
    },
    "servlet_context" : { "html_prefix" : "" }
  }
],
},
```

### Java code example 4:

```
// "method_with_servlet_sinks_on_input" directive example 4

// This example also demonstrates using a "lookup_by_constant_param"
// MethodCallSpecifier value to indicate that an enum parameter controls an
// optional escaper.

// NOTE: This example should include the Escapers.java code (for the
// 'escape_html' method call added by the directive).

package examples;

enum Choice { YES, NO }

class Test_method_input_servlet_sinks4 extends HttpServlet
{
    Choice unknownChoice;

    public void sink(String val, Choice escape) {}

    public void doGet(HttpServletRequest request, HttpServletResponse resp)
        throws IOException
    {
        PrintWriter pw = resp.getWriter();

        String taint = request.getParameter("taint");

        // Similar to the example above, the directive causes the analysis to behave
        // as if 'taint' flows to a HTML PCDATA context after being escaped with
        // 'escape_html' and so the analysis does not report a defect here.
        sink(taint, Choice.YES);

        // According to the directive, a Choice.NO argument indicates no escaping,
        // so the analysis reports an XSS defect report here.
        sink(taint, Choice.NO);

        // The directive also indicates that the null Choice argument means no
        // escaping, so the analysis reports an XSS defect here too.
        sink(taint, null); // XSS sink from directive + no escaper
    }
}
```

```
// The Choice controlling escaping is not an expected literal, so the
// analysis logs a warning but does not report a defect.
sink(taint, unknownChoice);
}
}
```

### 3.1.23.3. See also

Section 3.1.11, “define\_lookup\_method\_call\_map”, Section 3.2.13, “MethodCallSpecifier” for define\_lookup\_method\_call\_map

### 3.1.24. method\_with\_servlet\_sinks\_on\_output

Supported Languages: C#, Java, and Visual Basic

The `method_with_servlet_sinks_on_output` directive indicates that one of the outputs of a method (its return value or the final state of one of its mutable parameters) is written to the HTTP output. The XSS (cross-site scripting) checker reports a defect if tainted data is written to the HTTP output without proper escaping.

#### 3.1.24.1. Fields

This directive uses the following fields:

`"method_with_servlet_sinks_on_output"`

Specifies a `MethodSet` that identifies the methods to which this directive will be applied. See Section 3.2.15, “MethodSet”.

`"output_param_sinks"`

Specifies a JSON array. Each object in this array indicates that one of the outputs of this method (either its return value, or the final state of one of its mutable parameters) flows to the HTTP output.

Objects in the `"output_param_sinks"` array use the following fields:

`"output"`

Specifies a `ParamOut` value that names the output of the method that this object describes. See Section 3.2.18, “ParamOut”.

`"servlet_context"`

An `HtmlOutputContext` value that indicates the HTML context (that is, the place in the HTML parse tree) into which the `"output"` flows. Different contexts imply different escaping obligations to avoid cross-site scripting (XSS). See Section 3.2.7, “HtmlOutputContext”.

#### ⚠ Caution

The `"servlet_context"` is an `HtmlOutputContext` value but its type *must not* be `"html_attribute_value_where_name_is_from_param"` (see Section 3.2.7.1, “html\_attribute\_value\_where\_name\_is\_from\_param”).

#### 3.1.24.2. Examples

Configuration example:

## Configuration file usage

---

```
// "method_with_servlet_sinks_on_output" directive example

// This example also demonstrates using the "html_prefix" HtmlOutputContext
// value to control the context.

{
  "method_with_servlet_sinks_on_output" :
    { "named" :
      "examples.Test_method_with_servlet_sinks_on_output.appendString_PCDataSink(
        java.lang.StringBuffer,
        java.lang.String)void"
    },
  "output_param_sinks" : [
    {
      "output" : "arg1",
      "servlet_context" : { "html_prefix" : "" }
    }
  ]
},

{
  "method_with_servlet_sinks_on_output" :
    { "named" :
      "examples.Test_method_with_servlet_sinks_on_output.appendString_AttrValSink(
        java.lang.StringBuffer,
        java.lang.String)void"
    },
  "output_param_sinks" : [
    {
      "output" : "arg1",
      "servlet_context" : { "html_prefix" : "<tag attr='" }
    }
  ]
},
}
```

### Java code example:

```
//"method_with_servlet_sinks_on_output" directive example

// This example also demonstrates using the "html_prefix" HtmlOutputContext
// value to control the context.

package examples;

class Test_method_with_servlet_sinks_on_output extends HttpServlet
{
  public void appendString_PCDataSink(StringBuffer sb, String str) {
    // The directive makes the analysis treat appending to 'sb' as writing to
    // servlet output in the HTML PCData context, so we get an XSS defect here.
    sb.append(str);
  }

  public void appendString_AttrValSink(StringBuffer sb, String str) {
```

```
// The directive makes the analysis treat appending to 'sb' as writing to
// servlet output in the single-quoted HTML tag value context, so we get an
// XSS defect here.
sb.append(str);
}

public void doGet(HttpServletRequest request, HttpServletResponse resp)
    throws IOException
{
    PrintWriter pw = resp.getWriter();

    String taint = request.getParameter("taint");

    appendString_PCDataSink(new StringBuffer(), taint);
    appendString_AttrValsink(new StringBuffer(), taint);
}
}
```

### 3.1.25. move\_xss\_outside\_method

Supported Languages: C#, Java, and Visual Basic

The `move_xss_outside_method` directive directs the analysis to report cross-site scripting (XSS) defects outside the specified methods.

#### 3.1.25.1. Fields

This directive uses the following field:

"move\_xss\_outside\_method"

A `MethodSet` value that identifies the methods that this directive will affect. See Section 3.2.15, "MethodSet".

#### 3.1.25.2. Examples

Configuration example:

```
// "move_xss_outside_method" directive example
{
  "move_xss_outside_method" :
  { "named" :
    "examples.Test_move_xss_outside_method.addUrlPrefix(
      java.lang.String)java.lang.String"
  }
},
```

Java code example:

```
// "move_xss_outside_method" directive example
package examples;
```

```
class Test_move_xss_outside_method extends HttpServlet
{
    public String addUrlPrefix(String str) {
        return "http://" + str; //directive moves XSS out of this method. no defect
    }

    public void doGet(HttpServletRequest request, HttpServletResponse resp)
        throws IOException
    {
        PrintWriter pw = resp.getWriter();

        String taint1 = request.getParameter("taint1");
        pw.println( addUrlPrefix(taint1) ); //directive moves XSS report to here

        String taint2 = request.getParameter("taint2");
        pw.println( addUrlPrefix(taint2) ); //directive moves XSS report to here
    }
}
```

### 3.1.26. sanitizer\_for\_checker

Supported Languages: C#, Java, and Visual Basic

The `sanitizer_for_checker` directive identifies a method that when passed a tainted argument, renders the argument's value safe for a checker's sinks. That checker will no longer report defects for values that are passed through the sanitizer method. Other checkers will not be affected. Common applications include sanitizers, encoders, and escapers.

This directive can only be used with user-defined checkers such as `DF.CUSTOM_CHECKER`. It is not applicable to built-in checkers.

#### 3.1.26.1. Fields

This directive uses the following fields:

"sanitizer\_for\_checker"

A JSON string that contains the name of the checker to which this directive applies.

"sanitizer"

A "to\_call\_site" WritableProgramData value that specifies the value that will replace the tainted argument. See Section 3.2.26.1, "to\_callsite".

#### 3.1.26.2. Examples

Example (Java):

```
{
  sanitizer_for_checker : "DF.MY_CUSTOM_DATAFLOW_CHECKER",
  sanitizer : {
    to_callsite : {
      callsite_with_static_target : {
        "named" : "examples.SanitizerForChecker.Clean(java.lang.String,
boolean)java.lang.String"
```

```
    },  
  },  
  input : "arg1"  
}  
}
```

Source code for the example:

```
package examples;  
  
import javax.servlet.http.HttpServletRequest;  
  
public class SanitizerForChecker  
{  
    // This is defined as a sink for a custom checker  
    // through an sink_for_checker directive (not shown).  
    public native void SinkStuff(String data);  
  
    // This is defined as a sanitizer for the same custom checker  
    // through the sanitizer_for_checker directive above.  
    public String Clean(String data, boolean useUnderscore) {  
        if (useUnderscore) {  
            return data.replaceAll(" ", "_");  
        } else {  
            return data.replaceAll(" ", "");  
        }  
    }  
  
    public void Demonstrate(HttpServletRequest req)  
    {  
        // Read an untrusted HTTP request parameter.  
        // This is a built-in "HTTP" taint source.  
        String x = req.getParameter("unsafe");  
  
        // It is a defect to pass 'x' to the sink!  
        SinkStuff(x);  
  
        // It is safe to pass a sanitized 'x' after calling Clean.  
        SinkStuff( Clean(x, true) );  
    }  
}
```

### 3.1.27. sensitive\_action

Supported Languages: JavaScript only

Use the `sensitive_action` directive to tell the `MISSING_AUTHZ` checker which function calls perform a sensitive action that requires an authorization check. The `MISSING_AUTHZ` checker reports a defect on code that performs a sensitive action that isn't protected by an authorization check.

#### 3.1.27.1. Fields

This directive uses a single field:

"sensitive\_action"

Specifies a `CallSiteSet` that identifies calls to functions that perform sensitive operations that require authorization. See Section 3.2.3, "CallSiteSet".

### 3.1.27.2. Examples

JavaScript example:

```
{
  sensitive_action : {
    call_on : {
      read_path_off_global : [ { "property" : "addUser" } ]
    }
  },
}
```

The `sensitive_action` directive above matches the `addUser()` call site in the following Node.js JavaScript code. If such a call is not guarded by an authorization check, `MISSING_AUTHZ` reports a defect on it.

```
addUser("guest");
```

The `addAdminUser()` function is also considered a sensitive action because it calls a function that performs a sensitive action. `MISSING_AUTHZ` reports a defect on the call to `addAdminUser()` unless it is guarded by an authorization check.

```
function addAdminUser() {
  addUser("admin");
}
// ...
addAdminUser();
```

### 3.1.28. sensitive\_operation

Supported Languages: Java only

The `sensitive_operation` directive promotes a defect found by the `WEAK_GUARD` checker to high impact in programs where a weak guard is used to control the execution of a sensitive operation.

#### 3.1.28.1. Fields

This directive uses the following field:

"sensitive\_operation"

Specifies a `MethodSet` value to identify those methods that should be treated as sensitive operations. See Section 3.2.15, "MethodSet".

#### 3.1.28.2. Examples

Configuration example:

```
{
```

```
"sensitive_operation" : {  
  "named" : examples.WeakGuard.secretOperation()void"  
}  
},
```

Java code example:

```
package examples;  
  
public class WeakGuard {  
  native void secretOperation();  
  
  void test(HttpServletRequest request) throws IOException {  
    String sourceIP = request.getRemoteAddr();  
    if (sourceIP != null && sourceIP.equals("134.23.43.1")) {  
      secretOperation();  
    }  
  }  
}
```

### 3.1.29. simple\_entry\_point

Supported Languages: C#, Java, and Visual Basic

The `simple_entry_point` directive identifies methods that are entry points for a Web application. By default all parameters of the methods will be deeply tainted (meaning that the object, its fields, and the fields that belong to those fields are treated as though they are tainted) with the specified taint types. You can override this behavior with the optional `"tainted_args"` field. The level of depth of fields that are tainted is affected by the `cov-analyze` option `--webapp-security-aggressiveness-level`.

#### 3.1.29.1. Fields

This directive uses the following fields:

`"simple_entry_point"`

Specifies a `MethodSet` to identify the methods that are entry points to the Web app. See Section 3.2.15, "MethodSet".

`"taint_kinds"`

A JSON array of `"TaintKind"` values that identify the kinds of taint to report. See Section 3.2.23, "TaintKind".

`"tainted_args"`

(Optional) Specifies an array of `ParamIn` values that identify which arguments passed to the entry point should be considered tainted. If `"tainted_args"` is not present, *all* arguments passed to the entry point are considered to be tainted.

`"treat_as_xss_entry_point"`

(Optional) A JSON Boolean value.

If this value is set to `true`, any output to the HTTP response of this method will be rendered as HTML, and the the XSS checker will report defects if untrusted strings are not escaped correctly.

If this value is not specified, or if it is set to `false`, output to the method's HTTP response is not handled by the XSS checker.

### 3.1.29.2. Examples

Configuration example:

```
//"simple_entry_point" directive example

{
  "simple_entry_point" : {
    "named" : "examples.Test_simple_entry_point.entry(java.lang.String,
examples.UserBean)void"
  },
  "taint_kinds" : [ "http", "network" ]
},
```

Java code example:

```
//"simple_entry_point" directive example

package examples;
import java.sql.Connection;
import java.sql.Statement;

class InnerInnerBean
{
  private String innerInnerData;

  public String getInnerInnerData()          { return innerInnerData; }
  public void   setInnerInnerData(String arg) { innerInnerData = arg; }
}

class InnerBean
{
  private String          innerData;
  private InnerInnerBean innerInnerBean;

  public String getInnerData()          { return innerData; }
  public void   setInnerData(String arg) { innerData = arg; }
  public InnerInnerBean getInnerInnerBean()          { return innerInnerBean; }
  public void   setInnerInnerBean(InnerInnerBean arg) { innerInnerBean = arg; }
}

class UserBean
{
  private String data;
  private InnerBean innerBean;

  public String getData()          { return data; }
  public void   setData(String arg) { data = arg; }
  public InnerBean getInnerBean()          { return innerBean; }
  public void   setInnerBean(InnerBean arg) { innerBean = arg; }
}
```

```
}  
  
public class Test_simple_entry_point  
{  
    Connection connection;  
    Statement statement;  
  
    public void entry(String simpleString, UserBean customData)  
        throws Exception  
    {  
        // The string 'simpleString' is considered to be tainted with  
        // "http" and "network" taint. SQLI cares about both so it  
        // reports a defect when we see the taint (aliased to sqlQuery1)  
        // flow into connection.prepareStatement.  
        String sqlQuery1 =  
            "select * from " + simpleString;  
        statement = connection.prepareStatement(sqlQuery1); //SQLI  
  
        // This example demonstrates that we consider fields of classes  
        // as tainted in addition to simple objects like "simpleString".  
        String sqlQuery2 =  
            "select * from " + customData.getData();  
        statement = connection.prepareStatement(sqlQuery2); //SQLI  
  
        // This example demonstrates that, at default aggressiveness levels,  
        // we do not consider InnerInnerBean's fields as tainted.  
        String sqlQuery3 =  
            "select * from " +  
            customData.getInnerBean().getInnerInnerBean().getInnerInnerData();  
        statement = connection.prepareStatement(sqlQuery3); //no SQLI  
    }  
}
```

### 3.1.30. sink\_for\_checker

Supported Languages: Java, C#, JavaScript, and Visual Basic

The `sink_for_checker` directive identifies a sink for a checker.

#### 3.1.30.1. Fields

This directive uses the following fields:

"sink\_for\_checker"

A JSON string that contains the name of the checker. This checker can be one of the following checker types:

- A user-defined checker, created with the `dataflow_checker_name` directive.
- (JavaScript only) Any built-in tainted dataflow checker.

The checker indicated in "sink\_for\_checker" reports a defect when data that has a taint kind the checker cares about (and does not trust) is routed to the sink indicated by the "sink" field. The

`DF.CUSTOM_CHECKER` section in the *Checker Reference* explains in more detail how trust/distrust settings affect when a dataflow checker reports a defect.

"sink"

A `WritableProgramData` value that describes the sink; for example, by identifying a particular argument to a particular function. See Section 3.2.26, "WritableProgramData".

The analysis supports different kinds of `WritableProgramData` values for "sink" depending on the programming language to which this directive applies.

- For Java, Visual Basic, and C#, "sink" must be a "to\_callsite" `WritableProgramData` object. See Section 3.2.26.1, "to\_callsite".
- For JavaScript, "sink" can be any of the following kinds of `WritableProgramData` objects:
  - "to\_callsite"  
See Section 3.2.26.1, "to\_callsite".
  - "write\_to\_object\_with\_tag"  
See Section 3.2.26.2, "write\_to\_object\_with\_tag".
  - "write\_path\_off\_global"  
See Section 3.2.26.3, "write\_path\_off\_global".
  - "write\_off\_any"  
See Section 3.2.26.4, "write\_off\_any".

"sink\_kind"

This field is only supported by the JavaScript `SENSITIVE_DATA_LEAK` checker.

Specifies a `SinkKind` string, which specifies the type of "sink". See Section 3.2.22, "SinkKind".

### 3.1.30.2. Examples

Java directive example:

```
{
  sink_for_checker : "DF.MY_CUSTOM_DATAFLOW_CHECKER",
  sink : {
    to_callsite : {
      callsite_with_static_target : {
        "named" : "examples.SinkForChecker.SinkStuff(java.lang.String)void"
      },
    },
    input : "arg1"
  }
}
```

Java source code example:

```
package examples;

public class SinkForChecker
{
    // This could be defined in source, defined in bytecode, or
    // somewhere else. The part we care about is the "call" to
    // this method.
    public void SinkStuff(String data) {
        // Sinks the data.
    }

    // This method illustrates a call to SinkStuff. The directive
    // matches the call to SinkStuff. The directive is told that
    // "arg" (arg1) of SinkStuff is what is sinking.
    public void SomeOtherMethod()
    {
        SinkStuff("arg");
    }
}
```

Client-side JavaScript example:

The following directive adds a sink to the DOM\_XSS checker (which checks for cross-site scripting via the Document Object Model). Writing tainted data to the global variable location results in a defect report.

```
{
  "sink_for_checker" : "DOM_XSS",
  "sink" : {
    "write_path_off_global" : [ { "property" : "location" } ]
  }
}
```

### 3.1.31. tainted\_data

Supported Languages: JavaScript only

The `tainted_data` directive identifies tainted data, which is data that an attacker can influence to cause security vulnerabilities.

#### 3.1.31.1. Fields

This directive uses the following fields:

`"tainted_data"`

Specifies a `ReadableProgramData` value that indicates which data to consider tainted. See Section 3.2.19, "ReadableProgramData".

`"taint_kind"`

Specifies a `TaintKind` string that indicates the kind of taint with which `"tainted_data"` is tainted. See Section 3.2.23, "TaintKind".

The analysis considers any program data (global variable, function return value, and so on) that matches `"tainted_data"` to be tainted with a taint of kind `"taint_kind"`.

"is\_deep\_taint"

(Optional) A JSON Boolean value. When "is\_deep\_taint" is set to `true`, then properties of the "tainted\_data" (array elements, properties of the properties, and so on) are considered to be similarly tainted.

If this value is not specified, or if it is set to `false`, then properties of "tainted\_data" are not themselves considered to be tainted.

### 3.1.31.2. Examples

Configuration example:

The following is an example of using this directive for client-side JavaScript code. This example marks global variable `myLibrary.queryParam` as tainted with kind `js_client_url_query_or_fragment` (similar to the JavaScript `window.location.query`).

```
{
  "taint_kind" : "js_client_url_query_or_fragment",
  "tainted_data" : {
    "read_path_off_global" : [
      { "property" : "myLibrary" },
      { "property" : "queryParam" }
    ]
  }
}
```

JavaScript code example:

The following client-side JavaScript code illustrates the effect of this directive.

```
function tainted_data_client() {
  var t = myLibrary.queryParam;
  document.write(t);    // DOM_XSS

  var n = myLibrary.somethingElse;
  document.write(n); //no DOM_XSS
}
```

The local variable `t` is tainted because of the directive, but local variable `n` is not. When `t` flows into the first argument of a call to `document.write`, the analysis reports a `DOM_XSS` defect (this checker reports cross-site scripting via the Document Object Model).

The following is an example that uses this directive for server-side JavaScript code. This directive says that the return value of `require('myLib').getObjectFromRequestParam()` contains deeply tainted data from HTTP request parameters; in other words, that it was entirely constructed (or deserialized from) data in an HTTP request.

```
{
  "taint_kind" : "http",
  "is_deep_taint" : true,
  "tainted_data" : {
    "output" : "return",
    "from_callsite" : {
```

```
    "call_on" : {
      "read_from_js_require" : "myLib",
      "path" : [ { "property" : "getObjectFromRequestParam" } ]
    },
  }
},
```

The following Node.js code illustrates the effect of this directive. In this example, the local variable `o` is deeply tainted because of the directive. The effect of the deep taint is that `o.s.cmd` is tainted, so its flow into the argument of `exec` results in an `OS_CMD_INJECTION` defect report.

```
function node() {
  var myLib = require("myLib");
  var o = myLib.getObjectFromRequestParam();
  // Because 'o' is deeply tainted, 'o.s.cmd' is tainted.
  // Hence, passing it to an API that executes it results in a
  // OS_CMD_INJECTION defect report.
  require("child_process").exec(o.s.cmd); // OS_CMD_INJECTION
}
```

### 3.1.32. text\_checker\_name

Supported File Types: Text, XML

The `text_checker_name` directive defines a `TEXT.CUSTOM_CHECKER`.

#### 3.1.32.1. Fields

The custom text checker directive uses the following fields:

"text\_checker\_name"

A JSON string that specifies the checker name. This string must start with "TEXT. ", and what follows must consist of all capital letters or the underscore character.

For example, `TEXT.MY_CHECKER` is a valid name, but neither `"TEXT.My_Checker"` nor `"MY_CHECKER"` would be valid.

"file\_pattern"

A `RegularExpression` value that describes filename paths in which defects will be reported (see Section 3.2.20, "RegularExpression"). Files whose name does not match this pattern are not analyzed. This directive treats file names and paths in the following standardized manner:

- The name is made absolute, including the drive letter on Windows systems.
- The forward-slash character ( / ) separates name components.
- When no drive letter is present, the name begins with a forward-slash character ( / ); otherwise, a forward-slash character ( / ) follows the drive letter.

"defect\_pattern"

A `RegularExpression` value that specifies a pattern to match in files being analyzed (see Section 3.2.20, "RegularExpression"). Analysis will report a defect at each location that matches this pattern.

"defect\_message"

(Optional) A JSON string to print in the defect event message.

"remediation\_advice"

(Optional) A JSON string to print as remediation advice in each defect report.

"new\_issue\_type"

(Optional) An `IssueTypeDefinition` object that specifies the checker properties, a CWE mapping, and issue taxonomy. See Section 3.2.12, "IssueTypeDefinition".

When used as a "new\_issue\_type" value, all of the `IssueTypeDefinition` fields are optional.

### 3.1.33. xss\_sanitizer\_method

Supported Languages: C#, Java, and Visual Basic

The `xss_sanitizer_method` directive describes the string replacements that the cross-site-scripting (XSS) sanitizer method performs. Use this directive to improve the XSS checker results in cases where the checker does not correctly recognize what a sanitizer does.

#### 3.1.33.1. Fields

This directive uses the following fields:

"xss\_sanitizer\_method"

A `MethodSet` that identifies the methods to which this directive applies. See Section 3.2.15, "MethodSet".

"input"

A `ParamIn` value to identify the unsanitized input to the methods in "xss\_sanitizer\_method". See Section 3.2.17, "ParamIn".

"output"

A `ParamOut` value to identify the sanitized output from the methods in "xss\_sanitizer\_method". See Section 3.2.18, "ParamOut".

"step1"

A JSON array. Each field in this array describes a string operation that the sanitizer method performs on the "input" in order to compute the "output". In other words, the operations in each step array describe a series of character replacements.

"step2", "step3", ... and so on

(Optional) You can add additional `step` arrays, as needed. Each additional step should have the same structure as "step1".

Some sanitizers handle nested language contexts (for instance, a string inside JavaScript inside an HTML attribute value). These require multiple steps.

For another example, a step might describe HTML entity encoding (changing `&` to `&amp;`, and so on) for an HTML attribute value, while a different step describes transforming newline characters to `\n` for JavaScript strings.

The replacement operations specified in each step have the following requirements:

- They do not interfere with each other.

In other words, the order in which the replacements within a step are applied does not change the outcome of the step as a whole.

- They apply to the same language context.

For example, operations for escaping an HTML attribute value should not be mixed with operations for escaping a string value in a JavaScript program.

For more information, see “Step entries and step examples”.

### 3.1.33.2. Step entries and step examples

**‘step1’ example.** The following is an example of a step:

```
"step1":
  [
    { PREPEND_BACKSLASH : [ "\"", "'" ] },
    { JS_CHAR_CODE : [ "\n" ] },
  ],
```

This step describes how three different characters are replaced in a JavaScript string:

- Prepend a backslash in front of any single-quote or double-quote character.
- Replace the newline character with an escape sequence that is *different* from simply placing a backslash in front of the character. (This distinction is important because it removes the newline from the string.)

The replacements in this step can be performed in any order to obtain the same result, and they all apply to the same language context: a string in JavaScript.

**‘step2’ example.** If you also want the sanitizer to perform HTML entity encoding on the quote and double-quote characters, you need to add another step to use the JavaScript string in an HTML attribute value, as shown in the following example:

```
"step2":
  [
    { HTML_CHAR_REF : [ "\"", "'" ] },
  ],
```

The steps occur in order, taking the output of the preceding step. That is, `step1` replaces a quote with `\'`, and `step2` turns that into `&quot;`.

A step value is a JSON array of values representing an unordered set of replacements that apply to different characters.

Each array element is a JSON object that has a single field:

- The `name` describes the kind of replacement operation.
- The `value` describes a set of replaced characters.

The set of replaced characters can be described in two ways:

- Using an array of JSON strings that represent individual characters.

JSON string escape sequences might be needed to express certain characters.

Example:

```
"step1":  
  [  
    { REMOVE : [ "\"", "'", "\u2029" ] },  
  ],
```

- Using a regular expression to match a set of characters.

Example:

```
"step1":  
  [  
    { REMOVE : { regex-charset : "[^a-zA-Z0-9]" } },  
  ],
```

Names and meanings of character-replacement operations:

- PREPEND\_BACKSLASH

Insert a `\` in front of the character. This is used in JavaScript and CSS strings, for certain characters, to literally mean those characters. Some characters (for example, `n` in JavaScript, or `A` in CSS) cannot be escaped this way, since the result will mean something different.

Within a step, this operation can be mixed with either `JS_STRING_CHAR_CODE` or `CSS_CHAR_CODE` operations.

Example:

Replacing `"` with `\`.

*Not* an example:

Replacing newline with `\n` is *not* an example of `PREPEND_BACKSLASH`.

- HTML\_CHAR\_REF

Replace the character with a numeric or named HTML character reference.

Within a step, this operation cannot be mixed with other kinds of operations.

Examples:

Replacing `&` with `&#38;` or `&#x26;` or `&amp;`

- JS\_STRING\_CHAR\_CODE

Replace a character in a JS string with a numeric or reserved escape sequence that is different from PREPEND\_BACKSLASH.

Within a step, this operation can be mixed with PREPEND\_BACKSLASH operations.

Examples:

- `\n` for newline
- `\u000A` for newline
- **CSS\_CHAR\_CODE**

Replace a character in a CSS string with a numeric escape sequence.

Within a step, this operation can be mixed with PREPEND\_BACKSLASH operations.

Example:

`\0000A` for newline

- **URI\_PERCENT**

Replace the character with a percent escape sequence used in URIs.

Within a step, this operation *cannot* be mixed with other kinds of operations.

Example:

Replace `&` for `%26`

- **REMOVE**

Remove the character.

Within a step, this operation *cannot* be mixed with other kinds of operations.

### 3.1.33.3. Configuration and Java code examples

Configuration example:

```
// This is a 1-step sanitizer model for HTML escaping an attribute value.
{
  "xss_sanitizer_method" :
  { "named" :
    "examples.Test_xss_sanitizer_method.escapeAttributeValue(
      java.lang.String)java.lang.String"
  },
  "input" : "arg1",
  "output" : "return",
  "step1" :
  [
```

## Configuration file usage

---

```
    { HTML_CHAR_REF : [ "\"", "'", "&" ] },
  ],
},

// This is also a 1-step sanitizer model for HTML escaping an attribute value.
// This demonstrates using a regular expression for specifying the affected
// characters.
{
  "xss_sanitizer_method" :
  { "named" :
    "examples.Test_xss_sanitizer_method.escapeAttributeValue_regex_spec(
      java.lang.String)java.lang.String"
  },
  "input" : "arg1",
  "output" : "return",
  "step1":
  [
    { HTML_CHAR_REF : { regex-charset : "[\\\"'&]" } },
  ],
},

// This is a 1-step sanitizer model for removing dangerous characters from an
// attribute value.
// This also demonstrates using a regular expression to specify a character set.
{
  "xss_sanitizer_method" :
  { "named" :
    "examples.Test_xss_sanitizer_method.filterAttributeValue(
      java.lang.String)java.lang.String"
  },
  "input" : "arg1",
  "output" : "return",
  "step1":
  [
    { REMOVE : { regex-charset : "[\\\"'&]" } },
  ],
},

// This is a 1-step sanitizer model for escaping a JavaScript string.
{
  "xss_sanitizer_method" :
  { "named" :
    "examples.Test_xss_sanitizer_method.escapeJavaScriptString(
      java.lang.String)java.lang.String"
  },
  "input" : "arg1",
  "output" : "return",
  "step1":
  [
    { JS_STRING_CHAR_CODE : [ "\"", "'", "\\\" " ] },
  ],
},
},
```

## Configuration file usage

---

```
// This is a 2-step sanitizer model:
// Step 1: escape for a JavaScript string.
// Step 2: escape for an HTML attribute value.
{
  "xss_sanitizer_method" :
  { "named" :
    "examples.Test_xss_sanitizer_method.escapeJavaScriptStringInAttributeValue(
      java.lang.String)java.lang.String"
  },
  "input" : "arg1",
  "output" : "return",
  "step1":
  [
    { JS_STRING_CHAR_CODE : [ "\"" , "'", "\\\" " ] },
  ],
  "step2":
  [
    { HTML_CHAR_REF : [ "\"" , "'", "&" ] },
  ],
},
```

### Java code example:

```
package examples;

import java.util.*;
import java.io.*;
import javax.servlet.*;
import javax.servlet.http.*;

class Test_xss_sanitizer_method extends HttpServlet
{
  // The XSS analysis will use the xss_sanitizer_method directive for
  // the sanitization models, rather than these implementations.

  String escapeAttributeValue(String val) {
    return val;
  }
  String escapeJavaScriptString(String val) {
    return val;
  }
  String escapeJavaScriptStringInAttributeValue(String val) {
    return val;
  }

  public void doGet(HttpServletRequest request, HttpServletResponse resp)
    throws IOException
  {
    PrintWriter pw = resp.getWriter();
    String taint = request.getParameter("taint");

    // Demonstrate an XSS from unsanitized text in a title attribute value.
```

```
pw.print("<p title=\"" + taint + "\">"); // XSS

// Demonstrate text sanitized using the 1-step sanitizer model
// for the attribute value escaper.
String safe_text = escapeAttributeValue(taint);
pw.print("<p title=\"" + safe_text + "\">"); // no XSS

// The same as the previous example.
// The difference is that the xss_sanitizer_method uses a
// regular expression to specify the escaped characters.
String safe_text2 = escapeAttributeValue_regex_spec(taint);
pw.print("<p title=\"" + safe_text2 + "\">"); // no XSS

// Demonstrate an XSS from an unsanitized string in JavaScript
// in an onclick attribute value.
String unsafe_js = "alert('" + taint + "')";
pw.print("<div onclick=\"" + unsafe_js + "\">"); // XSS

// Demonstrate sanitizing the string-in-JavaScript-in-attribute using
// two escapers with 1-step sanitizer models.
String safe_js = escapeJavaScriptString(taint);
String safe_attrval = "alert('" + escapeAttributeValue(safe_js) + "')";
pw.print("<div onclick=\"" + safe_attrval + "\">"); // no XSS

// Demonstrate sanitizing the string-in-JavaScript-in-attribute using
// an escaper with a 2-step sanitizer model.
String safe_js_attrval =
    "alert('" +
        escapeJavaScriptStringInAttributeValue(taint) +
        "')";
pw.print("<div onclick=\"" + safe_js_attrval + "\">"); // no XSS

}
}
```

## 3.2. Other object types used by user directives

This section describes other kinds of JSON objects (see Section 2.2, “JSON Terminology”) used by the user directives. These objects specify data types used by the directives, particular directive behaviors, and so on.

### 3.2.1. AccessPathElement

**Used by these objects:** InputAndAccessPathSpecifier, OutputAndAccessPathSpecifier, ReadableProgramData, WritableProgramData

An access path from a base value to another value is represented as a non-empty array of AccessPathElement values. One AccessPathElement describes a single step in the access path.

#### 3.2.1.1. Fields

This object uses a single field:

"property"

A JSON string value that names a property of the object at this point in the access path.

### 3.2.1.2. Example

The following array of `AccessPathElement` values, if applied to the object `baseObj`, would represent the value `baseObj.x.y.z`.

```
[ { "property" : "x" }, { "property" : "y" }, { "property" : "z" } ]
```

## 3.2.2. AnnotationSet

**Used by these objects:** `ClassSet`, `MethodSet`

An `AnnotationSet` value describes a set of *analysis annotations* (called *annotations* in Java or *attributes* in C#) found in the program source. An `AnnotationSet` can be specified using one of the following field names:

- "named"
- "matching"

### 3.2.2.1. named

A "named" `AnnotationSet` matches all uses of the Java annotation or C# attribute whose entire mangled class name matches the "named" string. See Section 3.2.5, "ClassName" for a description of the mangled name format.

#### 3.2.2.1.1. Fields

A "named" `AnnotationSet` has a single field called "named":

"named"

A JSON string value that specifies the mangled class name of the analysis annotation to match.

#### 3.2.2.1.2. Examples

The following example of a named `AnnotationSet` matches uses of the Java `@Deprecated` annotation:

```
{ "named": "java.lang.annotation.Deprecated" }
```

### 3.2.2.2. matching

A "matching" `AnnotationSet` includes all uses of any Java annotation or C# attribute whose entire mangled class name matches the regular expression in the "matching" string (a substring match is insufficient). See Section 3.2.5, "ClassName" for a description of the mangled name format.

#### 3.2.2.2.1. Fields

A "matching" `AnnotationSet` has a single field called "matching":

"matching"

A JSON string that contains a Perl-style regular expression that specifies an analysis annotation class name to match.

### 3.2.2.2. Examples

The following "matching" AnnotationSet example matches uses of annotation classes named EntryPoint in any package.

```
{ "matching": ".*EntryPoint" }
```

### 3.2.3. CallsiteSet

**Used by these directives:** csrf\_check\_needed, csrf\_validator, dataflow\_through\_call\_site, map\_read, map\_write, sensitive\_action

**Used by these objects:** ReadableProgramData, WritableProgramData

A CallsiteSet identifies a set of function call sites in the source program. There are a few different ways to specify a CallsiteSet. The different kinds of CallsiteSet are supported for different programming languages. The sections that follow describe the kinds of CallsiteSet:

- "callsite\_with\_static\_target"
- "call\_on"
- "new\_on"

#### 3.2.3.1. callsite\_with\_static\_target

Supported Languages: C# , Java, and Visual Basic

A "callsite\_with\_static\_target" CallsiteSet matches call sites whose static call target (that is, the function to which the call resolves before considering virtual call resolution) is in a specified MethodSet.

##### 3.2.3.1.1. Fields

"callsite\_with\_static\_target"

Specifies a MethodSet. Function call sites that call a function in this set are included in the CallsiteSet.

##### 3.2.3.1.2. Examples

Java Example:

```
{
  callsite_with_static_target : {
    "named" : "battle.robot.api.RobotService.run(java.lang.String, int)void"
  }
},
```

The CallsiteSet above matches the call site in this Java code:

```
void doRotate(battle.robot.api.RobotService robot) {
    robot.run("rotate", 180);
}
```

### 3.2.3.2. call\_on

Supported Languages: JavaScript only

A "call\_on" CallsiteSet matches call sites where the function value itself (not the result of the call, but the expression being called) matches a specified ReadableProgramData value.

#### 3.2.3.2.1. Fields

"call\_on"

Specifies a ReadableProgramData value that the function value at a call site must match, in order to be included in this CallsiteSet.

"when"

(Optional) If present, specifies a CallsiteCondition that a call site must satisfy to be included in this CallsiteSet.

#### 3.2.3.2.2. Examples

JavaScript example:

```
{
  "call_on" : {
    "read_off_any" : [ {"property" : "addEventListener"} ]
  },
  "when" : {
    "only_if_arg_index" : 1,
    "iequals_string" : "click"
  }
}
```

The CallsiteSet above matches the JavaScript call site this call site:

```
anything.addEventListener("CLICK", x);
```

However, it does not match the following call sites because they do not satisfy the CallsiteCondition.

```
anything.addEventListener("CLACK", x);
anything.addEventListener();
```

call\_on will also match call sites using new with a construction function. Consider the following Node.js JavaScript example:

```
{
  "from_callsite" : {
    "call_on" : {
      "read_from_js_require" : "myLib",
      "path" : [ { "property" : "myCtor" } ]
    }
  }
}
```

```
    },  
    "output" : "arg1"  
  }  
}
```

The `CallsiteSet` above inside `from_callsite` matches the new `myLib.myCtor` call site in this Node.js JavaScript code:

```
var myLib = require("myLib");  
var myObject = new myLib.myCtor(myParam)
```

The example also shows that `ParamOut` values such as `"arg1"` can be used when using the `"call_on"` field to match a constructor call (this differs from using the `"new_on"` `CallsiteSet`, which only allows the `"return"` `ParamOut` value).

### 3.2.3.3. new\_on

Supported Languages: JavaScript only

A `"new_on"` `CallsiteSet` matches constructor calls that use the `new` operator where the constructor expression matches a specified `ReadableProgramData` value.

#### 3.2.3.3.1. Fields

`"new_on"`

Specifies a `ReadableProgramData` value that the constructor expression must match to be included in this `CallsiteSet`

`"when"`

(Optional) If present, specifies a `CallsiteCondition` that a call site must satisfy to be included in this `CallsiteSet`.

#### 3.2.3.3.2. Using a CallsiteSet

Follow these guidelines for best results:

- When using a `"new_on"` `CallsiteSet`, only the `"return"` `ParamOut` value is allowed for `ParamOut` fields related to the call site (for example, on a `"from_callsite"` `ReadableProgramData` value).
- If non-`"return"` `ParamOut` values such as `"arg1"` are needed, use the `"call_on"` version of `CallsiteSet` to match the constructor call.
- Use `"call_on"` for `"new"` call sites, unless the directive should match only `"new"` call sites.

#### 3.2.3.3.3. Examples

Node.js JavaScript example:

```
{  
  "from_callsite" : {  
    "new_on" : {  
      "read_from_js_require" : "myLib",  
      "path" : [ { "property" : "myCtor" } ]  
    }  
  }  
}
```

```
    },  
    "output" : "return"  
  }  
}
```

The `CallsiteSet` above inside `"from_callsite"` matches the new `myLib.myCtor()` call site in this Node.js JavaScript code:

```
var myLib = require("myLib");  
var myObject = new myLib.myCtor(myParam)
```

The `ParamOut` `"output"` field above is restricted and can only be set to `"return"` (See `ParamOut`). In this example, the result of `new myLib.myCtor(...)` is matched, but the result of a direct call to `myLib.myCtor` is not matched.

### 3.2.4. CallsiteCondition

**Used by these objects:** `CallsiteSet`

A `CallsiteCondition` value provides a condition that must be satisfied in order for a `CallsiteSet` value to match a given call site.

#### 3.2.4.1. Fields

This object uses the following fields.

The `"only_if_arg_index"` field must always be present:

`"only_if_arg_index"`

An integer value, starting from 1, that specifies the position of the argument to which this condition applies.

The following optional fields are mutually exclusive. Only one of these, if any, must be present:

`"equals_string"`

(Optional) A JSON string value. If this field is present, the argument indicated by `"only_if_arg_index"` must be a string literal that has exactly this value and capitalization.

`iequals_string"`

(Optional) A JSON string value. If this field is present, the argument indicated by `"only_if_arg_index"` must be a string literal that equals this value. The comparison for the `iequals_string` is *not* case-sensitive.

`"regex_string"`

(Optional) A JSON string value that specifies a Perl-style regular expression. If this field is present, the argument indicated by `"only_if_arg_index"` must match this regular expression. The comparison for the `"regex_string"` is case-sensitive.

`"iregex_string"`

(Optional) A JSON string value that specifies a Perl-style regular expression. If this field is present, the argument indicated by `"only_if_arg_index"` must match this regular expression. The comparison for the `"iregex_string"` is *not* case-sensitive.

"equals\_int"

(Optional) Specifies an integer value. If this field is present, the argument indicated by "only\_if\_arg\_index" must be an integer literal that equals this value.

Finally, the following field, "is\_last\_arg", can be specified alone or in concert with one of the other optional fields:

"is\_last\_arg"

(Optional) A JSON Boolean value.

If this field is present, the argument indicated by "only\_if\_arg\_index" must be the last argument in the call site.

If "is\_last\_arg" is the *only* optional field specified, then "only\_if\_arg\_index" is allowed to equal 0, in order to express that the call site has no arguments.

### 3.2.4.2. See also

The example of a "call\_on" CallsiteSet for a CallsiteCondition example.

### 3.2.5. ClassName

**Used by these objects:** AnnotationSet, ClassSet, MethodCallSpecifier

A ClassName value describes the mangled name for a class type.

The mangled name uses the fully qualified name of that type, without including any generic type arguments.

**Java.** For Java, mangled type names follow the grammar below (using regex-style notation):

```
class_name ::= ( package "." )* class ( "$" inner_class )*
package   ::= identifier
class     ::= identifier
inner_class ::= identifier
```

An identifier non-terminal is a valid source-code identifier.

**Visual Basic.** For C# and Visual Basic, mangled type names follow the grammar below (using regex-style notation):

```
class_name ::= ( namespace "." )* class ( "/" inner_class )*
namespace ::= identifier
class     ::= identifier generic_arity?
inner_class ::= identifier generic_arity?
generic_arity ::= "`" [0-9]+
```

### 3.2.6. ClassSet

**Used by these directives:** class\_like\_print\_writer\_for\_servlet\_output

**Used by these objects:** MethodSet

A `ClassSet` describes a set of classes from the program. You can specify a `ClassSet` using one of the following field names:

- "named"
- "matching"
- "with\_super"
- "with\_annotation"

### 3.2.6.1. named

A "named" `ClassSet` locates a class by name.

#### 3.2.6.1.1. Fields

This kind of `ClassSet` object has a single field:

"named"

Specifies a `ClassName` value. The "named" `ClassSet` matches this class with the mangled name in `named`. See the `ClassName` section for a description of the mangled name format.

#### 3.2.6.1.2. Examples

The following example matches the Java `String` class:

```
{ "named": "java.lang.String" }
```

### 3.2.6.2. matching

The "matching" `ClassSet` includes any class whose entire mangled name matches the regular expression in the "matching" string (a substring match is insufficient). See the `ClassName` section for a description of the mangled name format.

#### 3.2.6.2.1. Fields

This kind of `ClassSet` object has a single field:

"matching"

A JSON string that contains a Perl-style regular expression that specifies the class name to match.

#### 3.2.6.2.2. Examples

The following example matches classes with names that end with "Writer" in the `com.example` package.

```
{ "matching": "com\\.example\\.\\.\\..*Writer" }
```

### 3.2.6.3. with\_super ClassSet

A "with\_super" `ClassSet` locates classes within the "with\_super" set that are categorized as a super-class or super-interface.

### 3.2.6.3.1. Fields

This kind of `ClassSet` object has a single field:

```
"with_super"  
  A ClassSet value. A "with_super" ClassSet matches all class types with a super-class or  
  super-interface that are members of the "with_super" set.
```

### 3.2.6.3.2. Examples

The following example matches all subclasses of "java.util.Collection".

```
{ "with_super": { "named": "java.util.Collection" } }
```

### 3.2.6.4. with\_annotation

A "with\_annotation" `ClassSet` uses an `AnnotationSet` value to match a class whose definition has any of the specified annotations.

#### 3.2.6.4.1. Fields

This kind of `ClassSet` object has a single field:

```
"with_annotation"  
  An AnnotationSet value that contains names or regular expressions to identify classes that will be  
  included in the ClassSet.
```

#### 3.2.6.4.2. Examples

The following "with\_annotation" `ClassSet` example matches any class defined with the annotation `java.lang.annotation.Documented`.

```
{ "with_annotation":  
  { "matching": "java\\.lang\\.annotation\\.Documented" }  
}
```

Example of a matching Java class definition:

```
@Documented  
public class FooBar {  
  // ...  
}
```

The following "with\_annotation" `ClassSet` example matches any class defined with the annotation `MyWebController`.

```
{ "with_annotation":  
  { "named": "MyWebController" }  
}
```

Example of a matching C# class definition:

```
[MyWebController]
```

```
class AccountController {  
    // ...  
}
```

### 3.2.7. HtmlOutputContext

**Used by these directives:** `method_with_servlet_sinks_on_input`,  
`method_with_servlet_sinks_on_output`

An `HtmlOutputContext` value describes the lexical context that precedes an HTML fragment.

An `HtmlOutputContext` can be specified using one of the following field names:

- `"html_attribute_value_where_name_is_from_param_value"`
- `"html_prefix"`

#### 3.2.7.1. html\_attribute\_value\_where\_name\_is\_from\_param

Locates the HTML context using input parameter values.

##### 3.2.7.1.1. Fields

A `"html_attribute_value_where_name_is_from_param_value"` `HtmlOutputContext` has the following fields:

`"html_attribute_value_where_name_is_from_param"`

A `ParamIn` value that is evaluated by a directive against a particular call site.

`value_quoting`

A JSON string that indicates how the attribute is quoted. This string must have one of the following values:

- `"single"` indicates using single quotes.
- `"double"` indicates using double quotes.
- `"none"` indicates using no quotes (the attribute is delimited by white space).

#### 3.2.7.2. html\_prefix

Locates the HTML context using a text string.

##### 3.2.7.2.1. Fields

A `"html_prefix_value"` `HtmlOutputContext` has the following fields:

`html_prefix`

A JSON string. If this string begins an HTML page, this field represents the lexical context at the end of the string.

### 3.2.7.3. Examples

See the examples in Section 3.1.24, “method\_with\_servlet\_sinks\_on\_output” and Section 3.1.23, “method\_with\_servlet\_sinks\_on\_input”.

### 3.2.8. InputAndAccessPathSpecifier

**Used by the following directives:** `dataflow_through_call_site`

An `InputAndAccessPathSpecifier` object uses a path to locate an input value.

#### 3.2.8.1. Fields

This object uses the following fields:

`"input"`

A `ParamIn` value. This specifies a base value that is input to the call site. Without a `"path"` entry, this value is the input value.

`"path"`

(Optional) A non-empty array of `AccessPathElement` values. When present, the input value is found on this access path, using the base value.

### 3.2.9. InputTaint

**Used by the following directives:** `async_method`

An `InputTaint` value marks the parameter of a callback as tainted with a specific taint kind.

#### 3.2.9.1. Fields

This object uses the following fields:

`"input"`

A `ParamIn` value that indicates the parameter to the callback that is tainted.

`"taint_kind"`

A `TaintKind` string value that specifies the kind of taint that `"input"` has.

`"is_deep_taint"`

(Optional) Specifies a JSON Boolean value.

Setting this field to `true` indicates that properties of `"input"` are also tainted, along with the parameter itself.

Not specifying this field is equivalent to setting it to `false`.

An `InputTaint` value is used together with identifying a callback. The analysis considers the callback parameter given by `"input"` to be tainted with taint of kind `"taint_kind"`. If `"is_deep_taint"`

is `true`, the analysis also considers properties of that parameter object (including array elements, properties of those properties, and so on) to be similarly tainted.

### 3.2.10. InputTag

**Used by these directives:** `async_method`, `local_callback`

An `InputTag` value assigns an arbitrary string-valued `"tag"` to a specific parameter to a callback. Other structures, such as `read_from_object_with_tag` and `write_to_object_with_tag`, can refer to these parameters using this `"tag"` value. Tagging a callback parameter has no other effect on the analysis; it simply enables the use of these other structures.

#### 3.2.10.1. Fields

This object uses the following fields:

`"tag"`

A JSON string that names the tag. Use this string to identify the tag in other directives.

`"input"`

A `ParamIn` value that indicates the parameter of the callback to which to assign the tag.

### 3.2.11. InputValue

**Used by these directives:** `async_method`, `local_callback`

An `InputValue` indicates that some argument at a call site flows into some parameter of a callback.

#### 3.2.11.1. Fields

This object uses the following fields:

`"value"`

A `ParamIn` value that indicates the call site argument that flows to the callback's `"input"` parameter.

`"input"`

A `ParamIn` value that indicates the parameter of the callback to which `"value"` flows.

### 3.2.12. IssueTypeDefinition

**Used by these directives:** `dataflow_checker_name`, `dc_checker_name`, `text_checker_name`

An `IssueTypeDefinition` value describes the sort of issues that a checker reports. Coverity Platform and other issue-display interfaces use the fields of this object to describe issues the checker reports, and to support sorting and filtering of issues from this checker.

#### 3.2.12.1. Fields

This object uses the following fields:

"type"

A JSON string to be used as an opaque ID within the security directives file. It is not meant to be visible in the user interface.

The "type" string must contain between one and sixty-four ASCII characters, each of which must be a letter, number, underscore, or period. In other words, the string must match the following regular expression `^[A-Za-z0-9_.]{1,64}$`.

 **Note**

The user of a prefix (such as `USER.*`) is highly recommended for all user-defined types. This ensures forward-compatibility with any new built-in issue types introduced in future versions of the Coverity Analysis tool.

"name"

A JSON string that briefly describes the kind of issue that this checker reports; for example, "SQL Injection".

"description"

A JSON string that provides a longer description of the issue; for example, "Unsafe use of tainted data in constructing an SQL query".

"local\_effect"

A JSON string that explains what effect this issue might have on the execution of the code in which it is reported; for example "An attacker might be able to execute arbitrary SQL queries of their choice."

"cwe"

(Optional) A JSON integer value that indicates which entry in the Common Weakness Enumeration (CWE) best describes this issue.

"impact"

One of the following JSON string values: "High", "Medium", "Low", or "Audit".

"category"

A JSON string that describes the general class of issue that this defect belongs to; for example, "Injection Vulnerability" or "API Misuse".

"quality\_kind"

(Optional) A JSON Boolean value.

If "quality\_kind" is true, issues from this checker will have a "kind" that equals "quality", and that appears accordingly when filtering issues in the user interface. You can set either, both, or neither of "quality\_kind" and "security\_kind" to be true.

"security\_kind"

If "security\_kind" is true, issues from this checker will have a "kind" that equals "security", and that appears accordingly when filtering issues in the user interface. You can set either, both, or neither of "quality\_kind" and "security\_kind" to be true.

### 3.2.12.2. Examples

```
"new_issue_type" : {
  "type" : "leftover_debug_code",
  "name" : "Deployed test servlet",
  "description" : "A possible test servlet will be deployed.",
  "local_effect" : "Leftover debug or test code is not intended to be deployed
with the application in a production environment, and it may expose unintended
functionality or bypass security features.",

  "cwe" : 489,
  "impact" : "Medium",
  "category" : "Medium impact security",
  "security_kind" : true,
}
```

### 3.2.13. MethodCallSpecifier

**Used by these directives:** `define_lookup_method_call_map`,  
`method_with_servlet_sinks_on_input`

A `MethodCallSpecifier` value describes a method to invoke, an argument to that method, and a parameter that the method outputs.

You can specify a `MethodCallSpecifier` by using one of the following field names:

- `"method_call"`
- `"lookup_by_constant_param"`

#### 3.2.13.1. method\_call

A `"method_call"` `MethodCallSpecifier` explicitly names the method and an input and output of that method.

##### 3.2.13.1.1. Fields

The `"method_call"` `MethodCallSpecifier` has the following fields:

`"method_call"`

Specifies a `MethodName` value.

`"input"`

Specifies a `ParamIn` value to pass to `"method_call"`.

`"output"`

Specifies a `ParamOut` value for `"method_call"` to return.

#### 3.2.13.2. lookup\_by\_constant\_param

A `"lookup_by_constant_param"` `MethodCallSpecifier` indicates that the method to call depends on the value of another argument, which is interpreted within the scope of the directive that contains this `MethodCallSpecifier`.

### 3.2.13.2.1. Fields

The "lookup\_by\_constant\_param\_value" `MethodCallSpecifier` has the following fields:

"lookup\_by\_constant\_param"  
A `ParamIn` value.

Given a call site indicated by the parent directive, the `ParamIn` value indicates a particular argument.

"lookup\_map"  
A JSON string.

The "lookup\_map" value is the name of a `String` to `MethodCallSpecifier` map defined by a `define_lookup_method_call_map` directive. See Section 3.1.11, "define\_lookup\_method\_call\_map".

If the argument expression's `String` form is a key in the map, the corresponding `MethodCallSpecifier` value or JSON null literal is evaluated in place of this "lookup\_by\_constant\_param" `MethodCallSpecifier`.

If the key is not in the map, the parent directive using this lookup cannot be evaluated, and a warning is logged.

Coverity supports matching the `String` form of the following kinds of constant literals:

- `null` for a null reference
- `true/false` for a Boolean constant
- An enum constant

This is the name of the enum class (in `ClassName` value format), followed by a dot, followed by the identifier for the constant.

### 3.2.14. MethodName

**Used by these objects:** `MethodCallSpecifier`, `MethodSet`

A `MethodName` value describes the mangled name for a method.

The mangled method name uses the non-generic types of the arguments and return values.

- Unconstrained type variables are replaced with `java.lang.Object` (Java) or `System.Object` (C#).
- Constrained type variables are replaced with their upper bound.

**Java:** For Java, mangled method names follow the grammar below (using regex-style notation):

```
method_name ::= class_name "." method "(" arg_list? ")" return_type
method      ::= identifier
```

```

class_name ::= ( package "." )* class ( "$" inner_class )*
package    ::= identifier
class      ::= identifier
inner_class ::= identifier

arg_list   ::= ( arg_type ", " )* arg_type
arg_type   ::= type

return_type ::= type | "void"

type       ::= array_type | class_name | "boolean" | "byte" | "short" | "char" |
              "int" | "long" | "float" | "double"
array_type ::= type "[]"
    
```

An identifier non-terminal is a valid source code identifier.

Constructors have the string "<init>" for the method identifier and "void" for the return\_type.

**.NET:** For C# and Visual Basic, mangled method names follow the grammar below (using regex-style notation):

```

method_name ::= class_name "::" method "(" arg_list? ")" return_type
method      ::= identifier generic_arity?

class_name  ::= ( namespace "." )* class ( "/" inner_class )*
namespace   ::= identifier
class       ::= identifier generic_arity?
inner_class ::= identifier generic_arity?

arg_list    ::= ( arg_type ", " )* arg_type
arg_type    ::= type

return_type ::= type | "System.Void"

type        ::= array_type | class_name
array_type  ::= type "[]"

generic_arity ::= "`" [0-9]+
    
```

Constructors have the string `.ctor` for the method identifier and `System.Void` for the return\_type. For example:

```
NS.Foo::.ctor()System.Void
```

Static constructors have the string `.cctor` for the method identifier and `System.Void` for the return\_type. For example:

```
NS.Foo::.cctor()System.Void
```



**Note**

Primitive names are converted to the corresponding fully qualified class name; for example:

```
bool -> System.Boolean
byte -> System.Byte
sbyte -> System.SByte
```

### 3.2.15. MethodSet

**Used by these directives:** `ignore_all_argument_dataflow_to_method`, `ignore_method_dataflow`, `ignore_method_output`, `method_returns_constant`, `method_returns_param`, `method_returns_servlet_output_stream`, `method_returns_tainted_data`, `method_set_for_dc_checker`, `method_with_servlet_sinks_on_input`, `method_with_servlet_sinks_on_output`, `move_xss_outside_method`, `sensitive_operation`, `simple_entry_point`, `xss_sanitizer_method`

**Used by these objects;:** `CallsiteSet`

A `MethodSet` value describes a set of methods from the program. You can specify a `MethodSet` value by using one of the following field names:

- "named"
- "matching"
- "overrides"
- "implemented\_in\_class"
- "and"
- "with\_annotation"

#### 3.2.15.1. named

A "named" `MethodSet` matches the method with the mangled name in the "named" field.

See the `MethodName` section for a description of the mangled name format.

##### 3.2.15.1.1. Fields

"named"

A `MethodName` value to identify the method.

##### 3.2.15.1.2. Examples

The following example of a "named" `MethodSet` matches a single `print` method in `mypackage.MyClass`.

```
{ "named": "mypackage.MyClass.print(java.lang.String)void" }
```

### 3.2.15.2. matching

A "matching" MethodSet matches method names by using a Perl-style regular expression.

#### 3.2.15.2.1. Fields

A "matching" MethodSet has a single field.

"matching"

A JSON string that contains a regular expression. It matches any method whose mangled name satisfies the regular expression. The entire name must be matched; a substring match is insufficient.

See the `MethodName` section for a description of the mangled name format.

#### 3.2.15.2.2. Examples

The following "matching" MethodSet example matches any method named `print()` in `mypackage.MyClass`, regardless of the method's signature (for example, `mypackage.MyClass.print(int)int` and `mypackage.MyClass.print(java.lang.String)void`).

```
{ "matching": "mypackage\\.MyClass\\.print\\(\\.*" }
```



#### Note

While `.` (a dot) and `$` (a dollar sign) are characters that can appear in mangled names, they are also regex metacharacters and so must be backslash-escaped. Since a backslash is a metacharacter in JSON, it too must be escaped. Hence, when using one of these characters as a literal in a regex context, you need to escape it by prefixing it with *two* backslashes (`\\.`  or `\\$`) as in the example above. If instead, the regex above were `mypackage.MyClass.print.*`, it would match mangled names such as `mypackageXMyClass.print(char)void`.

### 3.2.15.3. overrides

An "overrides" MethodSet specifies a MethodSet to match any method that overrides a method in `overrides`. This includes methods in "overrides" itself.

#### 3.2.15.3.1. Fields

The "overrides" MethodSet has a single field:

"overrides"

A MethodSet. If a method in this set overrides a method in `overrides`, the overridden method is matched.

#### 3.2.15.3.2. Examples

For example, the following "overrides" MethodSet matches methods such as `java.util.ArrayList.add(java.lang.Object)boolean`:

```
{ "overrides":
  { "named": "java.util.Collection.add(java.lang.Object)boolean" } }
```

### 3.2.15.4. implemented\_in\_class

An "implemented\_in\_class" MethodSet uses a ClassSet to identify methods.

#### 3.2.15.4.1. Fields

The "implemented\_in\_class" MethodSet has a single field:

"implemented\_in\_class"  
 A ClassSet value. The "implemented\_in\_class" MethodSet matches any method that is a member of this set, including constructors and static initializers but *not including* any methods inherited from super-classes.

#### 3.2.15.4.2. Examples

For example, given the class A below, the "implemented\_in\_class" MethodSet that follows the class declaration would match these objects:

- The method A.getX()int
- The A constructor A.<init>(int)void
- The implicitly created, static initializer of A

It *would not* match methods that A inherits, such as java.lang.Object.hashCode()int.

```
class A {
  int x;
  int getX() { return x; }
  A(int x0) { x = x0; }
}
```

```
{ "implemented_in_class": { "named": "A" } }
```

### 3.2.15.5. and

An "and" MethodSet creates a new MethodSet by performing a logical AND between methods in the source codes and methods in a specified array of MethodSet values.

#### 3.2.15.5.1. Fields

The "and" MethodSet has a single field:

"and"  
 A JSON array of MethodSet values. The "and" MethodSet matches the intersection of the methods in the source and the methods matched by methods that are members of the MethodSet array.

#### 3.2.15.5.2. Examples

For example, the following "and" MethodSet matches methods in a particular package that override a particular method.

```
{ "and":
  [
    { "overrides": { "named": "com.example.C.print(java.lang.String)void" } },
    { "matching": "com\\.example\\.package\\.\\.*" }
  ]
}
```

### 3.2.15.5.3. See also

For additional details, see Section 3.2.15.2, “matching” and Section 3.2.15.3, “overrides”.

### 3.2.15.6. with\_annotation

A "with\_annotation" MethodSet uses an AnnotationSet to match methods in the source code that contain the annotations specified by the set.

#### 3.2.15.6.1. Fields

The "with\_annotation" MethodSet has a single field:

"with\_annotation"

An AnnotationSet value. The "with\_annotation" MethodSet matches a method whose definition contains any of the specified analysis annotations.

#### 3.2.15.6.2. Examples

The following "with\_annotation" MethodSet example matches any method defined with the annotation `java.lang.annotation.Documented`.

```
{ "with_annotation":
  { "matching": "java\\.lang\\.annotation\\.\\.Documented" }
}
```

Example of a matching Java method definition:

```
@Documented
void printHello() {
    System.out.println("Hello!");
}
```

The following "with\_annotation" MethodSet example matches any method defined with the annotation `MyCsharpAttribute`.

```
{ "with_annotation":
  { "named": "MyCsharpAttribute" }
}
```

Example of a matching C# method definition:

```
[MyCsharpAttribute]
string GetCorporateName() {
    return "Synopsys";
}
```

### 3.2.16. OutputAndAccessPathSpecifier

**Used by these directives:** `dataflow_through_call_site`

#### 3.2.16.1. Fields

An `OutputAndAccessPathSpecifier` uses the following fields:

"output"

A `ParamOut` value to specify a base value that is output from the call site. If a "path" field is not present, this is the output value itself.

"path"

(Optional) A non-empty array of `AccessPathElement` values. When this field is present, the "output" value is found on this access path, using the base value.

In an `OutputAndAccessPathSpecifier` values, if the "path" field is not specified, then "return" is the only allowed `ParamOut` value for the "output" field.

#### 3.2.16.2. Examples

Directive for JavaScript example:

```
{
  dataflow_through_callsite: {
    "call_on" : {
      "read_off_any" : [ {"property" : "returnsArgDotX"} ]
    },
  },
  from: [ {input: "arg1", path: [ {"property": "x"} ] } ],
  to: [ { output: "return" } ]
}
```

The directive above indicates that a call to `returnsArgDotX()` returns the `x` property of its argument. The following client-side JavaScript code shows how this directive can result in a `DOM_XSS` defect report (this checker reports cross-site scripting via the Document Object Model).

The directive indicates that the call to `returnsArgDotX(o)` returns `o.x`. Since `o.x` contains tainted data and the return value of the function flows into the argument of `document.write()` (a `DOM_XSS` sink), the analysis reports a `DOM_XSS` defect.

```
var o = { x: location.hash, y: "safe" };
document.write(returnsArgDotX(o)); // DOM_XSS
```

### 3.2.17. ParamIn

**Used by these directives:** `map_read`, `map_write`, `method_returns_param`, `method_with_servlet_sinks_on_input`, `xss_sanitizer_method`

**Used by these objects:** `HtmlOutputContext`, `InputAndAccessPathSpecifier`, `InputTaint`, `InputTag`, `InputValue`, `MethodCallSpecifier`, `WritableProgramData`

A `ParamIn` value describes an input to a function call.

### 3.2.17.1. Fields

This object can use the following fields:

`"this"`

When present, indicates the receiver object on instance methods.

`"arg1", "arg2", ...`

These fields represent the parameters (arguments) to the function.

The first non-`this` parameter field is `"arg1"`, and subsequent argument fields are numbered in sequence.

### 3.2.18. ParamOut

**Used by these directives:** `method_with_servlet_sinks_on_output`,  
`xss_sanitizer_method`

**Used by these objects:** `CallSiteSet`, `MethodCallSpecifier`,  
`OutputAndAccessPathSpecifier`, `ReadableProgramData`

A `ParamOut` value describes an output of a function call.

### 3.2.18.1. Fields

This object can use the following fields:

`"return"`

Indicates the function's return value.

`"this"`

When present, indicates the receiver object on instance methods.

`"arg1", "arg2", ...`

These fields represent the parameters (arguments) to the function.

The first non-`this` parameter field is `"arg1"`, and subsequent argument fields are numbered in sequence.

### 3.2.19. ReadableProgramData

**Used by these directives:** `data_has_tag`, `tainted_data`

A `ReadableProgramData` object identifies the location of a readable value: either for the purpose of noticing *reads* from that location, or to indicate that something is read from that location. You can specify a `ReadableProgramData` object by using one of the following field names:

- `"from_call_site"`

- "read\_from\_object\_with\_tag"
- "read\_path\_off\_global"
- "read\_off\_any"
- "read\_from\_js\_require"
- "read\_from\_HANA\_library\_import"

### 3.2.19.1. from\_callsite

Supported Languages: JavaScript only

A "from\_callsite" ReadableProgramData value identifies readable values produced by call sites.

#### 3.2.19.1.1. Fields

The "from\_call\_site" ReadableProgramData object has the following fields:

"from\_call\_site"

A CallsiteSet value that identifies call sites of interest.

"output"

A ParamOut value that identifies a base value passed in or returned from a call site.

"path"

(Optional) a non-empty array of AccessPathElement values. This is an access path to apply to the base value.

If no "path" field is specified, then the readable value is the base value indicated by the "output" field. Adding a "path" field indicates a readable value along the access path that is off of the base value.

#### 3.2.19.1.2. Examples

```
{
  "from_callsite" : {
    "call_on" : {
      "read_off_any" : [ { "property" : "exampleCall" } ]
    }
  },
  "output" : "return",
  "path" : [ { "property" : "f"}, { "property" : "g" } ]
}
```

The "from\_callsite" ReadableProgramData value above will match the readable value `exampleCall().f.g` based off of the return value of this call site:

```
exampleCall();
```

### 3.2.19.2. read\_from\_object\_with\_tag

Supported Languages: JavaScript only

A "read\_from\_object\_with\_tag" ReadableProgramData value identifies readable values found along an access path relative to a value that has been tagged by a data\_has\_tag directive. See Section 3.1.7, "data\_has\_tag".

### 3.2.19.2.1. Fields

The "read\_from\_object\_with\_tag" ReadableProgramData object has the following fields:

"read\_from\_object\_with\_tag"

A string value to identify values tagged by any data\_has\_tag directive that has the specified name. See Section 3.1.7, "data\_has\_tag".

"path"

A non-empty array of AccessPathElement values. This field specifies an access path to apply to the tagged values.

### 3.2.19.2.2. Examples

```
{
  "read_from_object_with_tag" : "myTagName",
  "path" : [ { "property" : "f"}, { "property" : "g" } ]
},
```

The "read\_from\_object\_with\_tag" ReadableProgramData value above with the following data\_has\_tag will match the readable value at location exampleTaggedValue.f.g because it tags the property exampleTaggedValue with the tag "myTagName".

```
{
  "data_has_tag" : { "read_off_any" : [ { "property" : "exampleTaggedValue" } ] },
  "tag" : "myTagName"
}
```

### 3.2.19.3. read\_path\_off\_global

Supported Languages: JavaScript only

A "read\_path\_off\_global" ReadableProgramData value identifies a readable value that is found along a given access path off of the global variable.

#### 3.2.19.3.1. Fields

The "read\_path\_off\_global" ReadableProgramData object has the following field:

"read\_path\_off\_global"

A non-empty array of AccessPathElement values that specify paths where readable values can be found.

#### 3.2.19.3.2. Examples

For examples that use "read\_path\_off\_global" ReadableProgramData, see Section 3.1.31, "tainted\_data", Section 3.1.3, "async\_method", Section 3.1.15, "local\_callback", and Section 3.1.17, "map\_write".

#### 3.2.19.4. read\_off\_any

Supported Languages: JavaScript only

A "read\_off\_any" ReadableProgramData value identifies a readable value found along a given access path.

##### 3.2.19.4.1. Fields

A "read\_off\_any" ReadableProgramData object has the following field:

"read\_off\_any"

A non-empty array of AccessPathElement values that specify paths where readable values can be found. Paths in "read\_off\_any" do not have to be relative to the global variable.

##### 3.2.19.4.2. Examples

For examples that use "read\_off\_any" ReadableProgramData, see Section 3.1.9, "dataflow\_through\_callsite", "to\_callsite", "from\_callsite", and "write\_to\_object\_with\_tag".

#### 3.2.19.5. read\_from\_js\_require

Supported Languages: JavaScript only

A "read\_from\_js\_require" ReadableProgramData value identifies a readable value along an access path that is relative to a JavaScript module value returned from a require call site. Calling require with the name of a module is a common approach to using modules, such as in Node.js programs.

##### 3.2.19.5.1. Fields

The "read\_from\_js\_require" object has the following fields:

"read\_from\_js\_require"

A string value that names the JavaScript module specified in the require call site.

"path"

(Optional) A non-empty array of AccessPathElement values for read\_from\_js\_require to use.

##### 3.2.19.5.2. Examples

For examples that use this ReadableProgramData, see Section 3.1.31, "tainted\_data" and "new\_on".

#### 3.2.19.6. read\_from\_HANA\_library\_import

Supported Languages: JavaScript only

A "read\_from\_HANA\_library\_import" ReadableProgramData value identifies a readable value along an access path that is relative to a HANA XSC library returned from a \$.import call site found in a .xsjs or .xsjslib source file.

This directive mirrors the \$.import method, in that two import formats are supported:

### 1. A file path

This directive format uses the `"read_from_HANA_library_import"` field to specify a file path that locates the library source file. The optional `"package"` field is not used.

This format corresponds to a call to `$.import` using a single argument.

### 2. A package specifier and a library name

In this directive format, the `"read_from_HANA_library_import"` field specifies the library name (without the `.xsjslib` extension), and the `"package"` field specifies the library's `"."`-separated package name.

This format corresponds to a call to `$.import` using two arguments.

#### 3.2.19.6.1. Fields

A `"read_from_HANA_library_import"` `ReadableProgramData` object has the following fields:

`"read_from_HANA_library_import"`

A string value that is either the name of the HANA XSC module specified in the `$.import` call site, or a file path that locates the library file.

`"package"`

A string value. If this field is present, then import method #2, described above, is used.

`"path"`

(Optional) A non-empty array of `AccessPathElement` values, for use with import method #2.

#### 3.2.19.6.2. Examples

The following two example directives are equivalent:

```
{
  "read_from_HANA_library_import" : "/package/name/lib.xsjslib",
  "path" : [ { "property" : "p" } ]
},
```

```
{
  "read_from_HANA_library_import" : "lib",
  "package" : "package.name",
  "path" : [ { "property" : "p" } ]
},
```

Both directives specified above will match both of the following equivalent import expressions found within an `.xsjs` or `.xsjslib` source file.

```
{
var p1 = $.import("/package/name/lib.xsjslib").p;
var p2 = $.import("package.name", "lib").p;
},
```

### 3.2.20. RegularExpression

**Used by these objects:** AnnotationSet, IssueTypeDefinition

A `RegularExpression` value is a JSON string that represents a regular expression (regex) in Perl syntax. Typically, a regular expression can match a substring of a target string. The regular expression can include anchors (such as an opening `^` or closing `$`) to explicitly specify the beginning of a target string, the end of the target string, or both.

Because the backslash is an escaping character in both JSON strings and Perl regular-expression syntax, a backslash used in a Perl regular expression needs to be escaped with *another* backslash in the JSON string. In a Perl regular expression, for example, a two-backslash sequence (`\\`) matches a single backslash in the target string—so to match a single backslash, a `RegularExpression` value requires *four* backslash characters: `\\\\`.

### 3.2.21. ReturnConstant

**Used by these directives:** `method_returns_constant`

A `ReturnConstant` value is a JSON object that describes the constant value returned by a method.

#### bool ReturnConstant value

- A JSON object describing a Boolean constant returned by a method.
- It has a field `bool`, taking a JSON Boolean value corresponding to the returned constant.

### 3.2.22. SinkKind

**Used by these directives:** `sink_for_checker`

A `SinkKind` value is a JSON string used in the `sink_for_checker` directive to specify a type of sink. It is only applicable to `SENSITIVE_DATA_LEAK` sinks. For possible values, see the `SENSITIVE_DATA_LEAK SinkKind` column of Table 4.2 Sensitive Data Sink types in the *Checker Reference*. See also Section 3.1.30, “`sink_for_checker`”.

### 3.2.23. TaintKind

**Used by these directives:** `dataflow_checker_name`, `method_returns_tainted_data`, `simple_entry_point`, `tainted_data`

**Used by these objects:** InputAndAccessPathSpecifier, TaintKindGroup

A `TaintKind` value describes a single taint kind. It has one of the values listed in this section. `TaintKind` values are divided into different groups that you can refer to collectively in some directives by using a `TaintKindGroup`.

The following taint kinds are relevant to server-side Web applications and other server-side applications:

- `"cookie"`: Data from HTTP cookies. See the `--trust-cookies` and `--distrust-cookies` options to the `cov-analyze` command.

- "command\_line": Data from the command line. See the `--trust-command-line` and `--distrust-command-line` options to the `cov-analyze` command.
- "console": Data from the console. See the `--trust-console` and `--distrust-console` options to the `cov-analyze` command.
- "database": Data from a database. See the `--trust-database` and `--distrust-database` options to the `cov-analyze` command.
- "environment": Data from environment variables. See the `--trust-environment` and `--distrust-environment` options to the `cov-analyze` command.
- "filesystem": Data read from a file. See the `--trust-filesystem` and `--distrust-filesystem` options to the `cov-analyze` command.
- "http": Data from incoming HTTP requests. This does not include headers or cookies. See the `--trust-http` and `--distrust-http` options to the `cov-analyze` command.
- "http\_header": Data from HTTP headers. See the `--trust-http-header` and `--distrust-http-header` options to the `cov-analyze` command.
- "network": Data from network connections. This does not include data from incoming HTTP requests or remote procedure calls. See the `--trust-network` and `--distrust-network` options to the `cov-analyze` command.
- "rpc": Data returned from remote procedure calls (RPC). See the `--trust-rpc` and `--distrust-rpc` options to the `cov-analyze` command.
- "system\_properties": Data on system properties. See the `--trust-system-properties` and `--distrust-system-properties` options to the `cov-analyze` command.

The following taint kinds are relevant to client-side JavaScript code (that is, JavaScript that runs in a Web browser):

"js\_client\_cookie"

Data from the JavaScript `document.cookie`. See the `--trust-js-client-cookie` and `--distrust-js-client-cookie` options to the `cov-analyze` command.

"js\_client\_external"

Data from the response to an `XMLHttpRequest` or similar. See the `--trust-js-client-external` and `--distrust-js-client-external` options to the `cov-analyze` command.

"js\_client\_html\_element"

Data from user input on HTML elements such as `textarea` and `input` elements. See the `--trust-js-client-html-element` and `--distrust-js-client-html-element` options to the `cov-analyze` command.

"js\_client\_http\_referer"

Data from the `'referrer'` HTTP header (from `document.referrer`). See the `--trust-js-client-http-referer` and `--distrust-js-client-http-referer` options to the `cov-analyze` command.

"js\_client\_http\_header"

Data from the HTTP response header of the response to an `XMLHttpRequest` or similar. See the `--trust-js-client-http-header` and `--distrust-js-client-http-header` options to the `cov-analyze` command.

"js\_client\_other\_origin"

Data from content in another frame or from another origin, for instance, from `window.name`. See the `--trust-other-origin` and `--distrust-other-origin` options to the `cov-analyze` command.

"js\_client\_url\_query\_or\_fragment"

Data from the query or fragment part of the URL, for instance, `location.hash` or `location.query`. See the `--trust-url-query-or-fragment` and `--distrust-url-query-or-fragment` options to the `cov-analyze` command.

The following taint kinds are relevant to mobile applications:

"mobile\_other\_app"

Data received from any mobile application that does not require a permission to communicate with the current application component. See the `--trust-mobile-other-app` and `--distrust-mobile-other-app` options to the `cov-analyze` command.

"mobile\_same\_app"

Data received from the same mobile application. See the `--trust-mobile-same-app` and `--distrust-mobile-same-app` options to the `cov-analyze` command.

"mobile\_user\_input"

Data obtained from user inputs into a mobile application. See the `--trust-mobile-user-input` and `--distrust-mobile-user-input` options to the `cov-analyze` command.

"mobile\_other\_privileged\_app"

Data received from any mobile application that requires a permission to communicate with the current application component. See the `--trust-mobile-other-privileged-app` and `--distrust-mobile--other-privileged-app` options to the `cov-analyze` command.

The following taint kinds represent sensitive data, rather than data controlled by an attacker:

"decrypted"

Decrypted data

"password"

A password

"token"

An authentication token

"session\_id"

A session ID

"mobile\_id"

A mobile device ID

"user\_id"  
An application user ID

"national\_id"  
A national ID

"persistent\_secret "  
Persistent secret data, such as an encryption key

"transient\_secret "  
Transient secret data, such as a TLS ticket

"seed "  
A seed for a randomization algorithm

"cardholder\_data "  
Payment cardholder data

"account "  
Account data

"transaction "  
Transaction data

"medical "  
Medical data

"biometric "  
Biometric data

"geographical "  
Sensitive geographical data

"exception "  
Information about an application exception

"source\_code "  
Application source code

"configuration "  
Configuration data

"bug "  
Information about a bug in the application

"file\_path "  
A path on the filesystem

"directory\_listing "  
A directory listing

"system\_memory"  
Information about system memory usage

"system\_user"  
System user data

"platform"  
Information about the runtime platform

### 3.2.24. TaintKindGroup

**Used by these directives:** `dataflow_checker_name`

A `TaintKindGroup` value describes a set of taint kinds. It consists of a JSON array of strings, each of which is either a `TaintKind` string or one of the following special strings that denotes a set of related taint kinds:

"all\_server\_taints"  
(Java, C#, JavaScript) Includes all taint kinds that are relevant to server-side Web applications and other server-side applications. See Section 3.2.23, "TaintKind".

"all\_jsclient\_taints"  
(JavaScript only) Includes all taint kinds that are relevant to client-side JavaScript code (JavaScript that runs in a Web browser). See Section 3.2.23, "TaintKind".

"all\_android\_taints"  
(Java only) Includes all taint kinds that are relevant to Android applications. See Section 3.2.23, "TaintKind".

### 3.2.25. TextPattern

**Used by these directives:** `text_checker_name`

A `TextPattern` describes a pattern for matching text strings. It is used to define custom text checkers. You can specify a `TextPattern` using one of the following field names:

- "regex"
- "xpath"

#### 3.2.25.1. regex

A "regex" `TextPattern` describes a regular expression to match a string or the text contents of a file.

##### 3.2.25.1.1. Fields

The "regex" `TextPattern` uses these fields:

"regex"  
A string value that specifies a Perl-style regular expression. For the JSON code to parse correctly, any special characters within the string need to be appropriately escaped.

"case\_sensitive"

(Optional) A Boolean value. If set to `false`, the match will be insensitive to case. The default value is `true`.

"line\_match"

(Optional) A Boolean value. If `true`, the caret ( `^` ) and dollar-sign ( `$` ) symbols, respectively, match the beginning and end of a line. This is equivalent to the Perl modifier `//m`. The default value is `true`.

"dot\_matches\_newline"

(Optional) A Boolean value. If `true`, the dot ( `.` ) character matches a newline character. This is equivalent to the Perl modifier `//s`. The default value is `true`.

### 3.2.25.1.2. Examples

```
{
  "regex" : "WEB-INF\\/(.+\\.xml$",
  "case_sensitive" : false
},
```

#### Note

The `.` (dot) and `/` (slash) characters are regex metacharacters and so must be backslash-escaped. Since a backslash is a metacharacter in JSON, it too must be escaped. Hence, when using one of these characters as a literal in a `regex` context, you need to escape it by prefixing it with two backslashes (`\\` or `\\/`) as in the example above.

### 3.2.25.2. xpath

An "xpath" `TextPattern` describes an Xpath 1.0 expression that can be used to match elements in an XML document.

If this pattern is applied to an input that is not parsable as XML, it will not match.

#### 3.2.25.2.1. Fields

The "xpath" `TextPattern` object has a single field:

"xpath"

A string value that specifies an Xpath expression.

#### 3.2.25.2.2. Examples

```
{ "xpath" : "/Catalog/Product[@name = \"soup\"]" },
```

```
{ "xpath" : "/*[local-name()='project']/*[local-name()='dependencies'] and
  child::*[local-name()='artifactId']" },
```

Notice that the double quotes in the Xpath expression have been escaped for JSON, using a backslash.

#### Tip

The `local-name()` function can be a convenient way to ignore the stricter namespace-specific element matching.

### 3.2.26. WritableProgramData

**Used by these directives:** `async_method`, `local_callback`, `sanitizer_for_checker`, `sink_for_checker`

A `WritableProgramData` value identifies the location of a writable value: either for the purpose of noticing writes to that location, or to indicate that something is written to that location. You can specify a `ReadableProgramData` object by using one of the following field names:

- `"to_callsite"`
- `"write_to_object_with_tag"`
- `"write_path_off_global"`
- `"write_off_any"`

#### 3.2.26.1. to\_callsite

A `"to_callsite"` `WritableProgramData` value identifies writable values consumed by call sites.

##### 3.2.26.1.1. Fields

The `"to_call_site"` `WritableProgramData` object has the following fields:

`"to_callsite"`

A `CallsiteSet` that identifies call sites of interest.

`"input"`

A `ParamIn` value that identifies a base value passed into the call site.

`"path"`

(Optional) An array of `AccessPathElement` values that specify an access path to apply to the base value.

The `"path"` field is only allowed for JavaScript uses of `"to_callsite"` `WritableProgramData`.

If no `"path"` field is specified, the writable value is the base value indicated by the `"input"` field. Adding a `"path"` field indicates a writable value along the access path off of the base value.

##### 3.2.26.1.2. Examples:

JavaScript example:

```
{
  "to_callsite" : {
    "call_on" : {
      "read_off_any" : [ { "property" : "exampleCall" } ]
    }
  },
  "input" : "arg1",
  "path" : [ { "property" : "f"}, { "property" : "g" } ]
}
```

```
}

```

The "to\_callsite" WritableProgramData value above will match the writable value passedInValue.f.g when passedInValue is passed into this call site:

```
exampleCall(passedInValue);

```

### 3.2.26.2. write\_to\_object\_with\_tag

A "write\_to\_object\_with\_tag" WritableProgramData value identifies writable values found along an access path relative to a value that has been tagged by a data\_has\_tag directive. See Section 3.1.7, "data\_has\_tag".

#### 3.2.26.2.1. Fields

The "write\_to\_object\_with\_tag" WritableProgramData object has the following fields:

"write\_to\_object\_with\_tag"

A string value that names values tagged by any data\_has\_tag directive. See Section 3.1.7, "data\_has\_tag".

"path"

A non-empty array of AccessPathElement values. This specifies an access path to apply to the tagged values.

#### 3.2.26.2.2. Examples

```
{
  "data_has_tag" : { "read_off_any" : [ { "property" : "exampleTaggedValue" } ] },
  "tag" : "myTagName"
},
{
  sink_for_checker : "DOM_XSS",
  sink : {
    "write_to_object_with_tag" : "myTagName",
    "path" : [ { "property" : "f"}, { "property" : "g" } ]
  },
}
```

The "data\_has\_tag" directive marks property accesses off any object (including the global object) with "myTagName". The "write\_to\_object\_with\_tag" WritableProgramData matches property writes, such as "exampleTaggedValue.f.g = x".

### 3.2.26.3. write\_path\_off\_global

Supported Languages: JavaScript only

A "write\_path\_off\_global" WritableProgramData value identifies a writable value found along a given access path off of the global variable.

#### 3.2.26.3.1. Fields

The "write\_path\_off\_global" WritableProgramData object has the following field:

"write\_path\_off\_global"

A non-empty array of `AccessPathElement` values, specifying the path off the global variable.

### 3.2.26.3.2. Examples

See Section 3.1.30, "sink\_for\_checker" for an example of the use of "write\_path\_off\_global" `WritableProgramData`.

### 3.2.26.4. write\_off\_any

Supported Languages: JavaScript only

A "write\_off\_any" `WritableProgramData` value identifies a writable value found along a given access path.

#### 3.2.26.4.1. Fields

The "write\_off\_any" `WritableProgramData` object has the following field:

"write\_off\_any"

A non-empty array of `AccessPathElement` values, specifying a path that is not necessarily based off of the global variable.

#### 3.2.26.4.2. Examples

See Section 3.1.3, "async\_method" for an example of the use of "write\_off\_any" `WritableProgramData`.