Software Assurance Tips

A product of the Software Assurance Tips Team[3]

Stacy Lukins

Monday 28th June, 2021

1 Coverity and Integer Overflows

Updated Wednesday 11th February, 2026

Integer overflows have been at the root of a number of security vulnerabilities in software over the years (see [2] for examples), such as the recently identified issue in which the result of an unchecked integer operation is used for memory allocation in a number of real time operating systems.[1] The ability of static analysis tools to detect these types of integer overflow problems vary. This article explores how Coverity identifies integer overflow.

It would not be especially helpful for a tool to flag every integer operation as a potential overflow. Thus, tools need some way to differentiate potentially problematic integer operations from seemingly safe integer operations. Coverity does this by only reporting a defect on an integer operation when the following 3 conditions are met: (1) The operands are determined to be tainted sources, (2) the operation is addition or multiplication (by default), and (3) the operation's result goes to a data sink.[4] All of these conditions can be overridden by setting various checker options. For example, you can change which data sources are treated as tainted and which integer operations are examined.

It's important to understand how Coverity defines tainted sources and sinks. Data can come into a program from a variety of outside sources (command line, console, filesystem, database, environment variable, RPC request, HTTP request, HTTP header, etc.). When data from an outside source has not been scanned and validated, it is considered to be tainted, or unsafe. Coverity tracks tainted data through a program, and it will detect and report whether the tainted data is used in a sink. A sink is any source code element, such as a function, that must be protected from tainted data. Sinks can be things like memory allocators, certain system calls, array index operations, and so on.

The simple functions in the (very C-like) C++ code in Listing 1 will be used to illustrate how Coverity identifies integer overflow.

```
int fun_1() {
        int i = 0;
        std::cin >> i;     // tainted source
        int j = i + 5;     // int overflow if i is too large
        return j;          // data sink
}

int fun_2(int val) {       // val not considered tainted (w/ default options)
        int j = val + 5;   // no int overflow here (w/ default options)
        return j;          // data sink
}

int call_fun_2() {
        int i = 0;
        std::cin >> i;     // tainted source
        int j = fun_2(i);  // calling fun_2 with tainted data
        return j;          // data sink
}

int g = 0;                 // global
void fun_3() {
        int i = 0;
        std::cin >> i;     // tainted source
        g = i + 5;         // no sink after this, so no int overflow flagged
}

int fun_4(int val) {       // can set option to consider param val to be tainted
        int j = 0;
        if (val < 100) {   // checking val, so no longer tainted
                j = val + 5; // will not overflow
        }
```

```
32          return j;          // data sink
33  }
```

Listing 1: Integer Overflow Examples

By default, the `INTEGER_OVERFLOW` checker in Coverity is not enabled and must be enabled by using the **`--enable`** option of the **`cov-analyze`** command. Be aware that the **`--all`** option does not enable the `INTEGER_OVERFLOW` checker.[4] The command in Listing 2 enables integer overflow checking with default options and was used to analyze the code in Listing 1.

```
cov-analyze --dir <path_to_code_idir> --aggressiveness-level high --all --enable
    INTEGER_OVERFLOW
```

Listing 2: Cov-analyze Parameters

The first function, **`fun_1`**, is a simple overflow example that will be caught by Coverity when the `INTEGER_OVERFLOW` checker is enabled with default checker options. In **`fun_1`**, `i` comes from an outside source (the command line, line 3) and is not scanned or validated, which makes the data tainted. A potential overflow occurs on line 4 (i could be close to or at the max integer value), and the potentially overflowed value is sent to a data sink on line 5 (returns are treated as data sinks by default in Coverity). Note that the sink could have been something like a memory allocation instead of a return. Coverity will identify this as an integer overflow defect, since it meets the 3 conditions discussed previously.

The integer overflow in **`fun_1`** is the only integer overflow defect in the source code above that Coverity will identify using the default options of the `INTEGER_OVERFLOW` checker. Consider **`fun_2`**, which contains the same addition operation and sink as **`fun_1`**. Coverity will not flag this as a potential integer overflow, even when a tainted source is passed in from the **`call_fun_2`** function on line 16. By default, Coverity does not consider parameters to be tainted sources, so this example does not meet the tainted source condition and Coverity does not flag it.

Now consider **`fun_3`**, which does contain a tainted source (command line input that is not checked) and performs the same addition operation. However, there is no sink in this function, so it does not meet the third condition and Coverity does not flag this operation as a potential integer overflow problem.

Checker options can be used to change the way Coverity identifies tainted sources and sinks. For example, the `enable_tainted_params` option can be set to `true` to cause Coverity to automatically treat all function parameters as tainted. Option values for checkers are set by passing **`--checker-option`** or **`-co`** to the cov-analyze command. We can re-analyze the source code using the command in Listing 3 which enables the `INTEGER_OVERFLOW` checker and sets the **`enable_tainted_params`** option to true.

```
cov-analyze --dir <path_to_build_idir> --aggressiveness-level high --all --enable
    INTEGER_OVERFLOW -co INTEGER_OVERFLOW:enable_tainted_params:true
```

Listing 3: Cov-analyze Parameters for Tainted Parameters

Now Coverity will also flag the addition operation in **`fun_2`** as an integer overflow, since the parameter val is considered tainted (the first condition is now met).

The addition operation in the last function, **`fun_4`**, will not be flagged by Coverity as an integer overflow with either cov-analyze command used. In **`fun_4`** the function parameter `val` is considered a tainted source when passed in. However, the parameter is tested on line 29 before the addition occurs (so it is no longer tainted), and an integer overflow will not occur when the addition is performed. Coverity recognizes this and does not does not flag the operation.

There are many options you can set that affect the way Coverity identifies tainted sources and sinks. In addition, the aggressiveness level affects some of the `INTEGER_OVERFLOW` options as well. See the Coverity Checker Reference[4] for more information.

References

[1] US-CERT. ICS Advisory (ICSA-21-119-04): Multiple RTOS (Update B). CISA. May 20, 2021. URL: https://us-cert.cisa.gov/ics/advisories/icsa-21-119-04 (visited on 06/28/2021).

[2] CWE Content Team. "CWE-190: Integer Overflow or Wraparound". In: (2021). URL: https://cwe.mitre.org/data/definitions/190.html.

[3] Jon Hood, ed. SwATips. https://www.SwATips.com/.

[4] Synopsis, Inc. "Coverity 2020.09 Checker Reference". In: (2020).