Secrets
ABAP
Apex
C
C++
CloudFormation
COBOL
C#
CSS
Flex
Go
HTML
Java
JavaScript
Kotlin
Kubernetes
Objective C
PHP
PL/I
PL/SQL
Python
RPG
Ruby
Scala
Swift
Terraform
Text
TypeScript
T-SQL
VB.NET
VB6
XML

C static code analysis

Unique rules to find Bugs, Vulnerabilities, Security Hotspots, and Code Smells in your C code

All rules 311 | Vulnerability 13 | Bug 74 | Security Hotspot 18 | Code Smell 206 | Quick Fix 14

Tags

Search by name...

"memset" should not be used to delete sensitive data
Vulnerability

POSIX functions should not be called with arguments that trigger buffer overflows
Vulnerability

XML parsers should not be vulnerable to XXE attacks
Vulnerability

Function-like macros should not be invoked without all of their arguments
Bug

The address of an automatic object should not be assigned to another object that may persist after the first object has ceased to exist
Bug

"pthread_mutex_t" should be unlocked in the reverse order they were locked
Bug

"pthread_mutex_t" should be properly initialized and destroyed
Bug

"pthread_mutex_t" should not be consecutively locked or unlocked twice
Bug

Functions with "noreturn" attribute should not return
Bug

"memcmp" should only be called with pointers to trivially copyable types with no padding
Bug

Bitwise operators should not be applied to signed operands

Analyze your code

Bug | Major | cwe based-on-misra lock-in bad-practice cert

Most built-in bitwise operators (~, >>, >>=, &, &=, ^, ^=, |, and |=) have implementation-dependent results when performed on signed operands, and bitwise left shift (<< and <<=) has unspecified or undefined behavior when performed on negative operands.

Therefore bitwise operations should not be performed on signed operands.

Starting with C++20, the behaviors have been defined more accurately (negative values have to be represented using two's complement), and therefore this rule will only report an issue when the second operand of a shift operator is signed (shifting by a negative value is still undefined behavior).

Noncompliant Code Example

```
if ( ( uint16_a & int16_b ) == 0x1234U ) // Noncompliant unti
if ( ~int16_a == 0x1234U ) // Noncompliant until C++20

auto f(int i) {
    return 1 << i; // Noncompliant
}
```

Compliant Solution

```
if ( ( uint16_a | uint16_b ) == 0x1234U )
if ( ~uint16_a == 0x1234U )

auto f(unsigned int i) {
    return 1 << i;
}
```

Exceptions

When used as bit flags, it is acceptable to use preprocessor macros as arguments to the & and | operators even if the value is not explicitly declared as unsigned.

```
fd = open(file_name, UO_WRONLY | UO_CREAT | UO_EXCL | UO_TRUN
```

If the right-side operand to a shift operator is known at compile time, it is acceptable for the value to be represented with a signed type provided it is positive.

```
#define SHIFT 24
foo = 15u >> SHIFT;
```

When combining several bitwise operations, even if all leaf operands are unsigned, if they are smaller than an `int`, some intermediate results will be of type `signed int`, due to integral promotion. However, this situation is usually not an issue, and is an exception for this rule:

Stack allocated memory and non-owned memory should not be freed

Bug

Closed resources should not be accessed

Bug

Dynamically allocated memory should be released

Bug

Freed memory should not be used

```
unsigned int f(unsigned short src) {
  return (src >> 3) & 0x1F; // (src >> 3) is of type signed i
}
```

See

- MISRA C:2004, 12.7 - Bitwise operators shall not be applied to operands whose underlying type is signed
- MISRA C++:2008, 5-0-21 - Bitwise operators shall only be applied to operands of unsigned underlying type
- MISRA C:2012, 10.1 - Operands shall not be of an inappropriate essential type
- CERT, INT13-C. - Use bitwise operators only on unsigned operands
- MITRE, CWE-682 - Incorrect Calculation

Available In:

sonarlint | sonarcloud | sonarqube Developer Edition