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 578 | Vulnerability 13 | Bug 111 | Security Hotspot 18 | Code Smell 436 | Quick Fix 68

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

Assigning to an optional should directly target the optional

Bug

Result of the standard remove algorithms should not be ignored

Bug

"std::scoped_lock" should be created with constructor arguments

Bug

Objects should not be sliced

Bug

Immediately dangling references should not be created

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

"std::move" and "std::forward" should not be confused

Bug

A call to "wait()" on a "std::condition_variable" should have a condition

Bug

A pointer to a virtual base class shall only be cast to a pointer to a derived class by means of dynamic_cast

Bug

Functions with "noreturn" attribute should not return

Bug

RAII objects should not be temporary

Bug

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

Bug

"memcpy", "memmove", and "memset" should only be called with pointers to trivially copyable types

Bug

"std::auto_ptr" should not be used

Bug

Destructors should be "noexcept"

Bug

Coroutines should not take const references as parameters

Code Smell Major

Coroutines, introduced in C++20, are functions in which execution can be suspended and resumed. When a coroutine resumes, it takes over where it left thanks to the coroutine state.

A *coroutine state* is an object which contains all the information a coroutine needs to resume its execution correctly: local variables, copy of the parameters…

This means that if a coroutine has a parameter that is a reference to an object, this object must exist as long as the coroutine is not destroyed. Otherwise, the reference stored in the *coroutine state* will become a dangling reference and will lead to undefined behavior when the coroutine resumes.

The issue is raised for all coroutine parameters with reference-to-const semantics (such as a `const` reference, a `std::string_view`, or a `std::span` with `const` elements) that might be used after the coroutine is suspended.

To fix the issue, you can either pass the parameter by value, or not use the parameter after the first suspension point (`co_await`, `co_yield`, or `initial_suspend`).

Noncompliant Code Example

```
generator<char> spell(const std::string& m) { // Noncompliant
    for (char letter : m) {
        co_yield letter;
    }
}

void print() {
    for (char letter : spell("hello world")) { // Here the pa
        std::cout << letter << '\n';           // and becomes
    }
}
```

Compliant Solution

```
generator<char> spell(const std::string m) { // Compliant: ta
    for (char letter : m) {
        co_yield letter;
    }
}

void print() {
    for (char letter : spell("hello world")) {
        std::cout << letter << '\n';
    }
}
```

Exceptions

This rule does not raise an issue for `std::reference_wrapper` parameters taking it as a witness of the care taken to prevent the reference to become dangling.