

- 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

Multiple mutexes should not be acquired with individual locks

Analyze your code

Code Smell

Critical

cppcoreguidelines bad-practice since-c++11 pitfall

Mutexes are synchronization primitives that allow to manage concurrency. It is a common situation to have to lock multiple *mutexes* simultaneously to get access to several resources at the same time.

If this is not done properly, it can lead to deadlocks or crashes. If one thread acquires A then tries to acquire B, while another thread acquires B then tries to acquire A, both threads will wait for each other forever.

In such a case, a commonly accepted good practice is to define an order on the *mutexes* and to lock them in that order and unlock them in the reverse order. However, such an order is not always clearly defined or easy to ensure across a whole program.

C++ provides facilities to lock multiple *mutexes* in one go, with a dedicated deadlock prevention algorithm. They should be used instead. Before C++17, you should use `std::lock`, and since C++17 you can use a variadic constructor of `std::scoped_lock`. See the examples for more details.

Noncompliant Code Example

```
void g();

std::mutex m1;
std::mutex m2;

void f() {
    // The example would be the same with std::lock_guard if C++11 is used
    std::scoped_lock<std::mutex> lck1(m1); // Compliant: first lock m1
    std::scoped_lock<std::mutex> lck2(m2); // Noncompliant: acquire m2 while m1 is locked
    g();
}
```

Compliant Solution

```
void g();











std::mutex m1;
std::mutex m2;

void f() { // Compliant: C++11 solution
    std::lock(m1, m2);
    std::lock_guard<std::mutex> lck1(m1, std::adopt_lock);
    std::lock_guard<std::mutex> lck2(m2, std::adopt_lock);
    g();
}

void f() { // Compliant: C++17 solution
    std::scoped_lock<std::mutex, std::mutex> lck1(m1, m2);
    g();
}
```

See

- [C++ Core Guidelines CP.21](#) - Use `std::lock()` or `std::scoped_lock` to acquire multiple mutexes
- [cppreference.com - std::scoped_lock](#)

 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
"memcpy" 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

Available In:

sonarlint

sonarcloud

sonarqube

Developer Edition