

- 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 rules578

Vulnerability13

Bug111

Security Hotspot18

Code Smell436

Quick Fix68

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

## Threads should not be detached

Analyze your code

Code SmellMinor?cppcoreguidelines since-c++11 clumsy

Sometimes, you might want to make a thread run indefinitely in the background by not binding it to its creation scope. Even though calling `detach()` on an `std::thread` or `std::jthread` object would satisfy this need, it is not the easiest way to do it: there will be no direct way to monitor and communicate with the detached thread, the `std::thread` or `std::jthread` object is no longer associated to any thread.

An easier alternative to satisfy this need is giving the thread a global scope. This way the thread will run as long as the program does. The thread will not be bound to any scope. It is also possible to do it by giving the `std::thread` or `std::jthread` a scope that is big enough for your use case. For example, the program's main function.

### Noncompliant Code Example

```
void backgroundTask();
void startBackgroundTask(){
    // Assume you want the thread to run after the end of start
    std::jthread backgroundThread(backgroundTask);
    backgroundThread.detach(); // Noncompliant
}
```

### Compliant Solution

```
void backgroundTask();
std::jthread backgroundThread;

void startBackgroundTask(){
    // Assume you want the thread to run after the end of start
    backgroundThread = std::move(std::jthread{backgroundTask});
}
```

### See

- C++ Core Guidelines CP.26 - Don't detach() a thread

Available In:

sonarlint | sonarcloud | sonarqube Developer Edition

 Bug
<p><b>"std::move" and "std::forward" should not be confused</b></p>  Bug
<p><b>A call to "wait()" on a "std::condition_variable" should have a condition</b></p>  Bug
<p><b>A pointer to a virtual base class shall only be cast to a pointer to a derived class by means of dynamic_cast</b></p>  Bug
<p><b>Functions with "noreturn" attribute should not return</b></p>  Bug
<p><b>RAII objects should not be temporary</b></p>  Bug
<p><b>"memcmp" should only be called with pointers to trivially copyable types with no padding</b></p>  Bug
<p><b>"memcpy", "memmove", and "memset" should only be called with pointers to trivially copyable types</b></p>  Bug
<p><b>"std::auto_ptr" should not be used</b></p>  Bug
<p><b>Destructors should be "noexcept"</b></p>  Bug