

- 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

"std::filesystem::path" should be used to represent a file path

Analyze your code

Code Smell Major performance since-c++17 clumsy

Since C++17, the class `std::filesystem::path` can be used to store a file path. Compared to a regular string, it offers several advantages:

- Having a dedicated type makes the intention clear
- This class stores the path with an encoding that is appropriate to the OS where the program runs
- It provides several functions that make it more convenient to manipulate than a `string` (for instance `operator/` for concatenations)
- It provides a normalized way to specify the path, easing the portability of the code (on Windows and Linux, the native way is equivalent to the normalized way, which reduces overhead).

This rule raises an issue when the same `string` is converted several times to a `path` because it indicates that a single `path` object could have been used in all occurrences. Additionally, it can also be more efficient, since a conversion from `string` to `path` may require a change of encoding and a memory allocation.

### Noncompliant Code Example

```
std::string getUserData();
namespace fs = std::filesystem;
void f() {
    std::string const filePath = getUserData();
    if (fs::exists(filePath)) {
        logTime(fs::last_write_time(filePath)); // Noncompliant
    }
}
```

### Compliant Solution

```
std::string getUserData();
namespace fs = std::filesystem;
void f() {
    fs::path const filePath = getUserData();
    if (fs::exists(filePath) {
        logTime(fs::last_write_time(filePath)); // Compliant
    }
}
```

Available In:

sonarlint

sonarcloud

sonarqube

Developer Edition

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