

- 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

"[*this]" should be used to capture the current object by copy

Analyze your code

Code Smell

Major ?

Quick Fix ?

since-c++17 clumsy pitfall

When you are using lambdas in a member function, you can capture `this` implicitly through `[=]` or `[&]` or explicitly through `[this]`. This will capture the current object pointer by reference or by value, but the underlying object will always be captured by reference (see {rule:cpp:S5019}).

This will become a problem:

- When the lifetime of the lambda exceeds the one of the current object.
- When you want to capture the current state of the object.
- When you want to pass a copy of the object to avoid any concurrency issue.

C++14 provides a workaround to solve this problem. Where you can take the underlying object by copy using the following pattern:

```
auto lam = [copyOfThis = *this] { std::cout << copyOfThis.fie
```

This is verbose and error-prone, as you might implicitly not use the copied object:

```
auto lam = [& , copyOfThis = *this] {
    std::cout << field; // implicitly calling "this" captured by
};
```

C++17 solves this problem by introducing an explicit consistent way to capture `this` by copy:

```
auto lam = [&, *this] {
    std::cout << field // implicitly calling "this" captured by c
};
```

This rule will flag the C++14 way of capturing the current object by copy and suggest to replace it with the C++17 way.

Noncompliant Code Example

```
struct A {
    int field = 0;
    void memfn() const {
        auto lam = [copyOfThis = *this] { // Noncompliant
            std::cout << copyOfThis.field;
        };
    }
};
```

Compliant Solution

```
struct A {
    int field = 0;
    void memfn() const {
        auto lam = [*this] { // Compliant
            std::cout << field;
        };
    }
};
```

 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

Available In:

sonarlint

sonarcloud

sonarqube

Developer Edition