



ABAP

Apex Apex

c c

C++

CloudFormation

COBOL COBOL

C# C#

E CSS

X Flex

GO Go

HTML

Js JavaScript

Kotlin

Java

Kubernetes

6 Objective C

PHP PHP

PL/I

L/SQL PL/SQL

Python

RPG RPG

Ruby

Scala

Swift

Terraform

Text

TS TypeScript

T-SQL

VB VB.NET

VB6 VB6

XML XML



C++ static code analysis

Unique rules to find Bugs, Vulnerabilities, Security Hotspots, and Code Smells in your C++ code

All 578 rules Vulnerability 13

📆 Bug (111)

Security Hotspot

(18)

e Code 436

Quick 68 Fix

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

<table-of-contents> 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::move" should only be added when necessary

Analyze your code

Code Smell

 cppcoreguidelines performance bad-practice since-c++11 clumsy

Usually, when copying an object, the source object is unchanged, which means that all resources owned by the source objects have to be duplicated during the copy operation. In the case that the source object will no longer be used, this duplication is not efficient. Since C++11, a mechanism named move semantic has been added to detect such cases and replace the expensive copy by a much cheaper move operation that will steal resources.

The cornerstone of move semantic is the ability to detect during a "copy" if the source object will be reused or not. There are three situations:

- The object is a temporary object, with no name, and if it can't be named, it can't be used
- The object is used in some specific places, such as a return statement
- The user explicitly promises to the compiler that he won't care for the current value of the object any longer. He does so by using the specific cast operation named std::move.

If the user write std::move in one situation that is already handled by the first two cases, it has two drawbacks:

- $\bullet\,$ It is clumsy, useless code, which make understanding the code more complex
- In some cases, it can decrease performances, because this can deactivate another optimization of the compiler, named copy elision.

When copy elision occurs, the object is neither copied nor moved (even if the copy/move constructors have side effects), in fact, the two objects are collapsed into only one memory location. When copy elision occurs is compiler-dependent, but is mandatory in the following cases:

- in a return statement if the returned object is a prvalue of the same class type as the function return type
- in the initialization of a variable if the initializer expression is a prvalue of the same class type as the variable type

This rule reports an issue when the use of std::move prevents the copy elision from happening.

Noncompliant Code Example

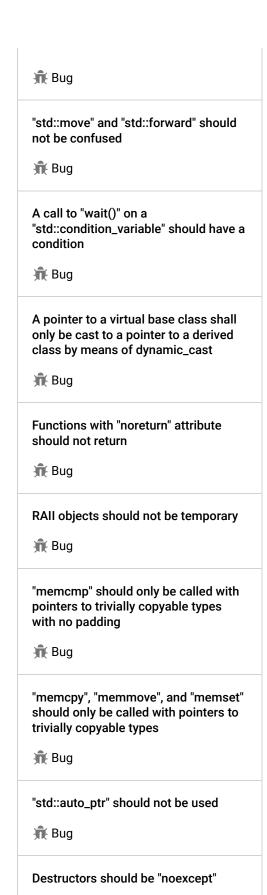
```
class A {};
A getA();

A f() {
    A a = std::move(getA()); // Noncompliant, prevents copy eli
    vector<A> v;
    v.push_back(std::move(getA())); // Noncompliant
    return std::move(a); // Noncompliant, prevents copy elision
}
```

Compliant Solution

```
class A {};
void test(A a);

A f() {
  A a = getA(); // Compliant
  vector<A> v;
  v.push_back(getA()); // Compliant
```



📆 Bug

```
return a; // Compliant
}

See

• C++ Core Guidelines I.4: Don't return std::move(local)

Available In:

sonarlint ○ | sonarcloud ○ | sonarqube | Developer Edition
```

© 2008-2022 SonarSource S.A., Switzerland. All content is copyright protected. SONAR, SONARSOURCE, SONARLINT, SONARQUBE and SONARCLOUD are trademarks of SonarSource S.A. All other trademarks and copyrights are the property of their respective owners. All rights are expressly reserved.

Privacy Policy