



ABAP Apex

С

C++

CloudFormation

COBOL

C#

CSS

Flex

Go =GO

5 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 578 6 Vulnerability 13 rules

R Bug (111)

• Security Hotspot ⊗ 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

📆 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

The return value of "std::move" should be used in a function

Analyze your code

suspicious

std::move is not really moving anything, but tells the compiler that a value can be considered as no longer useful. It is technically a cast to a RValue, and allows overload resolution to select the version of a function that will perform destructive operations on that value (therefore actually moving from it).

```
void f(A const &a); // Just reading from a
void f(A\&\& a); // I can perform destructive operations on a,
void g() {
    A a;
    f(a); // First overload is selected
    f(std::move(a)); // Second overload is selected
}
```

As a consequence, calling std::move on an object and then not directly using the returned value as a function argument is not the typical pattern, and may be indicative of a bug. Note that calling a member function on the result of std::move is considered as passing it to a function (as the hidden this parameter), as well as using it as an operand (the called function is the overloaded operator) or initializing an object with it (the called function is the constructor).

Noncompliant Code Example

```
void sink(A &&a) {
  std::move(a); // Noncompliant, and not doing anything
}
void f(A &&a) {
  // May or may not move the member name, depending on its ty
  // for instance, is `a` supposed to be in a moved-from stat
  g(std::move(a).name); // Noncompliant
}
```

Compliant Solution

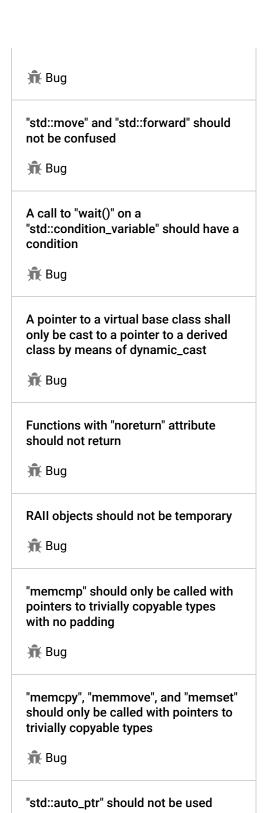
```
void f(A &&a) {
  g(std::move(a.name)); // Compliant, `a.name` is in moved-fr
```

Exceptions

Even if calling a built-in operator or initializing data of built-in type are not function calls, for consistency with cases that involve user-defined types, this rule will not report in those cases:

```
struct Data {A a; int b; };
Data add(Data &&d1, Data &&d2) {
  return Data{
    std::move(d1.a) + std::move(d2.a), // Compliant, operator
    std::move(d1.b) + std::move(d2.b) // Compliant by except
 };
```

Available In:



📆 Bug

📆 Bug

Destructors should be "noexcept"

sonarlint ⊖ | sonarcloud ↔ | sonarqube | beveloper 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