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::move" should only be added when necessary

Analyze your code

Code Smell | Major | 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:

- 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
```

```
  return a; // Compliant
}
```

See

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

Available In:

sonarlint | sonarcloud | sonarqube Developer Edition

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