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

Overriding member functions should do more than simply call the same member in the base class

Analyze your code

Code Smell | Minor | Quick Fix | redundant clumsy

Overriding a function just to call the overridden function from the base class without performing any other actions can be useless and misleading.

There are cases when it is justified, because redeclaring the function allows some side effects:

- Changing the visibility of the function in the derived class
- Preventing the base class function from being hidden by an overload added in the derived class (a using declaration could have the same effect)
- To resolve ambiguities in cases of multiple inheritance
- To make an inherited function final

This rule raises an issue when an override which is not in one of the aforementioned situation only calls the overridden function, directly forwarding its arguments.

Noncompliant Code Example

```
class Base {
public:
  virtual void f();
};

class Derived : public Base {
public:
  virtual void f() {
    Base::f(); // Noncompliant
  }
};
```

Compliant Solution

```
class Base {
public:
  virtual void f();
};

class Derived : public Base {
};
```

or

```
class Base {
public:
  void f();
};

class Derived : public Base {
private: // change of visibility
  virtual void f() {
    Base::f();
  }
};
```

or

```
class Base {
```

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

```
public:
  void f();
};

class Derived : public Base {
public:
  void f(int i);
  void f() { // Prevents hiding by f(int)
    Base::f();
  }
};
```

or

```
class Base {
public:
  virtual void f();
};

class Derived : public Base {
public:
  void f() final { // final
    Base::f();
  }
};
```

Available In:

sonarlint | sonarcloud | sonarqube Developer Edition