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

"operator=" should check for assignment to self

Analyze your code

Bug Minor

In some cases, we might end up with some code that assigns an object to itself. Therefore, when writing an `operator=`, we must ensure that this use case works correctly, which may require special care. One technique to achieve this is to explicitly check at the start of `operator=` if we are assigning to ourselves, and in that case, just do nothing.

It is usually a bad idea to perform this check for optimization purposes only, because it optimizes for a very rare case while adding an extra check for the more common case. But when it is necessary for correctness, it should be added.

This rule raises an issue when an `operator=` does not check for assignment to self before proceeding with the assignment.

Noncompliant Code Example

```
class MyClass
{
  private:
  int someVal;
  char* pData;

  MyClass& operator=(const MyClass& rhs)
  {
    this->someVal = rhs.someVal;              // useless oper
    delete [] pData;                          // data is lost
    pData = new char[strlen(rhs.pData) +1];   // null pointer
    strcpy(pData, rhs.pData);

    return (*this);
  }
};
```

Compliant Solution

```
class MyClass
{
  private:
  int someVal;
  char* pData;

  MyClass& operator=(const MyClass& rhs)
  {
    if (this != &rhs)
    {
      this->someVal = rhs.someVal;
      delete [] pData;
      pData = new char[strlen(rhs.pData) +1];
      strcpy(pData, rhs.pData);
    }
    return (*this);
  }
};
```

Or much better:

```
class MyClass
{
```

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

```
  private:
  int someVal;
  std::string data; // No need for manual operator=
};
```

Deprecated

This rule is deprecated, and will eventually be removed.

Available In:

sonarlint sonarcloud sonarqube Developer Edition