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

Designated initializers should be used in their C++ compliant form

Analyze your code

Code Smell | Major | pitfall

C++20 introduced a restricted form of designated initializers for aggregates (i.e. arrays or classes which respect specific criterion). Designated initializers enable initialization of aggregates by naming their fields explicitly:

```
struct Point {
  float x = 0.0;
  float y = 0.0;
  float z = 0.0;
};

Point p = {
  .x = 1.0,
  .y = 2.0,
  // z will be 0.0
};
```

This initialization style is similar to designated initializers in C and in many C++ compiler extensions predating C++20.

However, it is more restricted because some forms are not supported by the C++20 standard, namely:

- listing the fields out of order
- array initialization (including sparse array initialization)
- initialization of nested fields
- mixed initialization

This rule reports non-C++-compliant forms of designated initializers.

Noncompliant Code Example

```
struct A { int x, y; };
struct B { struct A a; };

struct A a = {.y = 1, .x = 2}; // Noncompliant: valid C, inva
int arr[3] = {[1] = 5};        // Noncompliant: valid C, inva
struct B b = {.a.x = 0};       // Noncompliant: valid C, inva
struct A c = {.x = 1, 2};      // Noncompliant: valid C, inva
```

Compliant Solution

```
struct A { int x, y; };
struct B { struct A a; };

struct A a = {.x = 2, .y = 1};
int arr[3] = {0, 5};
struct B b = {.a = {.x = 0}};
struct A c = {.x = 1, .y = 2};
```

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