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

Atomic types should be used instead of "volatile" types

Analyze your code

Code Smell | Major | cppcoreguidelines c11 multi-threading cert since-c++11

The main intended use-case for `volatile` in C and C++ is to access data that can be modified by something external to the program, typically some hardware register. In contrast with other languages that provide a `volatile` keyword, it does not provide any useful guarantees related to atomicity, memory ordering, or inter-thread synchronization. It is only really needed for the kind of low-level code found in kernels or embedded software, i.e. using memory-mapped I/O registers to manipulate hardware directly.

According to the C standard:

> `volatile` is a hint to the implementation to avoid aggressive optimization involving the object because the value of the object might be changed by means undetectable by an implementation.

Only C11/C++11 "atomic types" are free from data races, and you should use them or synchronization primitives if you want to avoid race conditions.

This rule raises an issue when a local variable or class data member is declared as `volatile` (at the top level of the type, pointers to volatile are not reported).

Noncompliant Code Example

```
volatile int counter; // Noncompliant
User * volatile vpUser; // Noncompliant; pointer is volatile
User volatile * pvUser;  // Compliant; User instance is volat
```

Compliant Solution

```
atomic_int counter;
std::atomic<User*> vpUser;
User volatile * pvUser;
```

See

- CERT CON02-C - Do not use volatile as a synchronization primitive
- C++ Core Guidelines CP.200 - Use volatile only to talk to non-C++ memory

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