sonar RULES

Products

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

Non-const global variables should not be used

Analyze your code

Code Smell   Critical   cppcoreguidelines bad-practice pitfall

A global variable can be modified from anywhere in the program. At first, this might look convenient, but in fact, it makes programs very hard to understand: When you see a function call, you cannot know if the function will affect the value of the variable or not. You have lost the ability to reason locally about your code and must always have the whole program in mind.

Additionally, in multi-threaded environments, global variables are often subject to race conditions.

Some global variables defined in external libraries (such as `std::cout`, `std::cin`, `std::cerr`) are fine to use, but you should have a good reason to create your own. If you do use a global variable make sure that they can be safely accessed concurrently.

This rule detects all declarations of global variables (in the global namespace or in any namespace) that are not constant.

Noncompliant Code Example

```
double oneFoot = 0.3048;
double userValue;
void readValue();
void writeResult();

int main() {
  readValue();
  writeResult();
}
```

Compliant Solution

```
constexpr double footToMeter = 0.3048;

double readValue();
void writeResult(double);

int main() {
  auto userValue = readValue();
  writeResult(userValue * footToMeter);
}
```

See

- C++ Core Guidelines I.2 - Avoid non-const global variables

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