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

Inline variables should be used to declare global variables in header files

Analyze your code

Code Smell   Major   since-c++17 clumsy

`C++17` introduced inline variables. They provide a proper way to define global variables in header files. Before inline variables, it wasn’t possible to simply define global variables without compile or link errors:

```
struct A {
  static std::string s1 = "s1"; // doesn’t compile
  static std::string s2;
};

A::s2 = "s2"; // doesn’t link, violates the one definition ru
std::string s3 = "s3"; // doesn’t link, violates the one defi
```

Instead, you had to resort to less readable inconvenient workarounds like variable templates or functions that return a static object. These workarounds will initialize the variables when used instead of the start of the program, which might be inconvenient depending on the program.

This rule will detect these workarounds and suggest using inline variables instead.

Noncompliant Code Example

```
struct A {
  static std::string& getS1() { // Noncompliant
    static std::string s1 = "s1";
    return s1;
  }
};

inline std::string& gets2() { // Noncompliant
  static std::string s2 = "s2";
  return s2;
}

template <typename T = std::string>
T s3 = "s3"; // Noncompliant. Available starting C++14
```

Compliant Solution

```
struct A {
  inline static std::string s1 = "s1"; // Compliant
};

inline std::string s2 = "s2"; // Compliant
```

Available In:

sonarlint | sonarcloud | sonarqube Developer Edition

© 2008-2022 SonarSource S.A., Switzerland. All content is copyright protected. SONAR, SONARSOURCE, SONARLINT, SONARQUBE and SONARCLOUD are trademarks of SonarSource S.A. All other trademarks and copyrights are the property of their respective owners. All rights are expressly reserved.
Privacy Policy

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