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

Nested code blocks should not be used

Analyze your code

Code Smell | Minor | bad-practice

Nested code blocks can be used to create a new scope: variables declared within that block cannot be accessed from the outside, and their lifetime end at the end of the block.

While this might seem convenient, using this feature in a function often indicates that it has too many responsibilities and should be refactored into smaller functions.

A nested code block is acceptable when it surrounds all the statements inside an alternative of a `switch` (a `case xxx:` or a `default:`) because it prevents variable declarations from polluting other `cases`.

Noncompliant Code Example

```
void f(Cache &c, int data) {
  int value;
  { // Noncompliant
    std::scoped_lock l(c.getMutex());
    if (c.hasKey(data)) {
      value = c.get(data);
    } else {
      value = compute(data);
      c.set(data, value);
    }
  } // Releases the mutex

  switch(value) {
    case 1:
    { // Noncompliant, some statements are outside of the blo
       int result = compute(value);
       save(result);
    }
    log();
    break;
    case 2:
    // ...
  }
}
```

Compliant Solution

```
int getValue(Cache &c, int data) {
  std::scoped_lock l(c.getMutex());
  if (c.hasKey(data)) {
    return c.get(data);
  } else {
    value = compute(data);
    c.set(data, value);
    return value;
  }
}

void f(Cache &c, int data) {
  int value = getValue(c, data);

  switch(value) {
    case 1:
    { // Compliant, limits the scope of "result"
       int result = compute(value);
```

```
      save(result);
      log();
    }
    break;
    case 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