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

Memory should not be managed manually

Analyze your code

Code Smell | Critical | cppcoreguidelines bad-practice since-c++11

If you manage memory manually, it's your responsibility to `delete` all memory created with `new`, and to make sure it's `deleted` once and only once. Ensuring this is done is error-prone, especially when your function can have early exit points.

Fortunately, the C++ language provides tools that automatically manage memory for you. Using them systematically makes the code simpler and more robust without sacrificing performance.

This rule raises an issue when you use:

- `new` - you should prefer a factory function that returns a smart pointer, such as `std::make_unique` or, if shared ownership is required, `std::make_shared`,
- `new[]` - you should prefer a container class, such as `std::vector`,
- `delete` or `delete[]` - if you followed the previous advice, there is no need to manually release memory.

If your compiler does not support `make_unique`, it's easy to write your own:

```
template<typename T, typename... Args>
std::unique_ptr<T> make_unique(Args&&... args) {
    return std::unique_ptr<T>(new T(std::forward<Args>(args).
}
```

Noncompliant Code Example

```
void f() {
  auto c = new Circle(0, 0, 5);
  c->draw();
  delete c;
}
```

Compliant Solution

```
void f() {
  auto c = make_unique<Circle>(0, 0, 5);
  c->draw();
  unique_ptr<Circle> c2{new Circle(0, 0, 5)}; // Clumsy, but
}
```

Exceptions

If the result of a new is immediately passed as an argument to a function, we assume that the function takes ownership of the newly created object, and won't raise an issue.

See

- C++ Core Guidelines R.11 - Avoid calling new and delete explicitly
- C++ Core Guidelines C.149 - Use unique_ptr or shared_ptr to avoid forgetting to delete objects created using new

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