

-  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

Structured binding should be used

Analyze your code

Code Smell Major ? cppcoreguidelines since-c++17 clumsy

C++17 introduced structured binding, a syntax that initializes multiple entities by elements or members of an object. It is especially useful to emulate multiple return values from a function.

Suppose you have a function that returns a pair:

```
std::pair<std::string, std::string> getPair();
```

Structured binding allows you to bind the members of the `std::pair` class directly to new names:

```
auto [firstName, lastName] = getPair();
```

The names `firstName` and `lastName` are called structured bindings. As you can see, structured binding makes the code more readable as they allow binding values to names that carry information about their purpose.

Structured binding works with:

- Raw arrays, by binding a name to each element
- Any type that has a tuple-like API
- Classes and structures where all non-static data member are publicly accessible

This rule will detect places where `std::pair` and `std::tuple` can be effortlessly replaced by a structured binding.

Noncompliant Code Example

```
void printingMap(const std::map<int, std::string>& map) {
    for (const auto& elem : map) { // Noncompliant
        std::cout << elem.first << ": " << elem.second << "\n";
    }
}
```

Compliant Solution

```
void printingMap(const std::map<int, std::string>& map) {
    for (const auto& [key, value] : map) { // Compliant
        std::cout << key << ": " << value << "\n";
    }
}
```

See

- [C++ Core Guidelines F.21](#) - To return multiple “out” values, prefer returning a struct or tuple

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