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

Function template parameters should be named if reused

Analyze your code

Code Smell | Major | since-c++20 clumsy pitfall

C++20 introduces full template support for lambda functions on par with the regular template functions. The full template syntax for a lambda adds a template-arguments clause after the capture clause completing the panoply of brackets: []<>(){}. For example:

```
[]<typename T>(T arg) { return arg; }
```

Although more verbose than using `auto` for the types of the arguments, this syntax enables you to name the types for the parameters, constrain these types (see Concepts), and reuse these types for multiple arguments.

One common use case for the named template argument is a lambda with multiple arguments of the same type. Pre-C++20 code had to resort to the use of `decltype`: `[](auto arg1, decltype(arg1) arg2) ...` . Not only is it obscure it also only approximates our goal: it requires the second-argument type to be convertible to the first-argument type.

Moreover, similar issues may appear for normal functions, that declare parameters with `auto` in place of type using C++20 abbreviated template syntax.

This rule reports the use of `decltype(arg)` for parameters introduced with `auto`.

Noncompliant Code Example

```
void f1() {
  auto sum = [](auto fir, decltype(fir) sec) { return fir + s
  std::cout << sum(true, 1); // Prints 2
}

void f2(auto param) {  // Noncompliant
   decltype(param) copy = param;
}
```

Compliant Solution

```
void f1() {
  auto sum = []<class T>(T fir, T sec) { return fir + sec; };
  // std::cout << sum(true, 1); - compilation error
}

template<class T>
void f2(T param) { // Compliant
    T copy = param;
}
```

See

- Modern C++: More powerful lambdas with C++20

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