- 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

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

Thread local variables should not be used in coroutines

Analyze your code

Code Smell   Major   confusing since-c++20 suspicious unpredictable

In contrast to normal functions, coroutines can suspend and later resume their execution. Depending on the program, the coroutine may resume on a different thread of execution than the one it was started or run previously on.

Therefore, the access to the "same" variable with `thread_local` storage may produce different values as illustrated below:

```
thread_local std::vector<Decorator> decorators;
lazy<Thingy> doSomething() {
  // evaluation started on thread t1
  /* .... */
  const std::size_t decoratorCount = decorators.size(); // va
  auto result = co_await produceThingy();
  // after co_await, execution resumes on thread t2
  for (std::size_t i = 0; i < decoratorCount; ++i) {
    decorators[i].modify(result); // access value specific to
    // miss some tasks if t1:decorators.size() < t2:decorator
    // undefined behavior if t1:decorators.size() > t2:decora
  }
  co_return result;
}
```

This behavior is surprising and unintuitive compared to normal functions that are always evaluated on a single thread. The same issue can happen for the use of different thread-local variables if their values are interconnected (e.g., one is the address of the buffer, and the other is the number of elements in the buffer).

Moreover, access to `thread-local` variables defined inside the coroutine may read uninitialized memory. Each such variable is initialized when a specific thread enters the function for the first time, and if the function was never called from a thread on which the coroutine is resumed, it is uninitialized.

This rule raises an issue on the declaration of `thread_local` variables and access to `thread_local` variables in coroutines.

Noncompliant Code Example

```
thread_local std::vector<Decorator> decorators;
lazy<Thingy> doSomething() {
  thread_local Decorator localDecorator; // Noncompliant
  const std::size_t decoratorCount = decorators.size(); // No
  /* ... */
  auto result = co_await produceThingy();
  for (std::size_t i = 0; i < taskCount; ++i) {
    decorators[i].modify(result);
  }
  localDecorator.modify(result); // Noncompliant
  co_return result;
}
```

Available In:

sonarlint | sonarcloud | sonarqube Developer Edition