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

Use type-erased "coroutine_handle" when applicable

Analyze your code

Code Smell | Minor | Quick Fix | bad-practice since-c++20

The implementation of the `await_suspend` method accepts the handle to the suspended coroutine as the parameter. This parameter can be defined with either specific promise type `coroutine_handle<PromiseType>` or type erased `coroutine_handle<>`. The former allows `await_suspend` to access the promise of the coroutine; however, it ties the implementation to a particular type. In contrast, using `coroutine_handle<>` increases the reusability of the code because this parameter type supports all promise types.

This rule raises an issue for the implementation of `await_suspend` that accepts handles to a specific promise type and yet does not use that information.

Noncompliant Code Example

```
struct Awaiter1
{
   Event& event;
   /* ... */
   bool await_suspend(std::coroutine_handle<Promise> current)
     return event.register_callback([current] {
              current.resume();
            });
   }
};

struct Awaiter2
{
   Event& event;
   /* ... */
   bool await_suspend(std::coroutine_handle<PromiseA> current
     return event.register_callback([current] {
              current.resume();
            });
   }
   bool await_suspend(std::coroutine_handle<PromiseB> current
     return event.register_callback([current] {
              current.resume();
            });
   }
};

struct Awaiter3
{
   Event& event;
   /* ... */
   template<typename PromiseType>
   bool await_suspend(std::coroutine_handle<PromiseType> curr
     return event.register_callback([current] {
              current.resume();
            });
   }
};
```

Compliant Solution

```
struct Awaiter // Instead of each of Awaiter1, Awaiter2, Awai
{
```

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

```
   Event& event;
   /* ... */
   bool await_suspend(std::coroutine_handle<> current) {
     return event.register_callback([current] {
              current.resume();
            });
   }
};

struct AwaiterUsingPromise
{
  /* ... */
  void await_suspend(std::coroutine_handle<Promise> current)
    auto wokeUpTime = std::chrono::system_clock::now() + std:
    current.promise().executor().schedule_at(wokeUpTime, curr
  }

};
```

Available In:

sonarlint sonarcloud sonarqube Developer Edition