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 symmetric transfer to switch execution between coroutines

Analyze your code

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

With C++20 coroutines, the `co_await`/`co_yield` expression suspends the currently executed coroutine and resumes the execution of either the caller or the coroutine function or to some already suspended coroutine (including the current coroutine).

The resumption of the coroutine represented by the `std::coroutine_handle` object is usually performed by calling the `.resume()` on it. However, performing such an operation during the execution of `await_suspend` (that is part of `co_await` expression evaluation) will preserve the activation frame of the `await_suspend` function and the calling code on the stack. This may lead to stack overflows in a situation where the chain of directly resumed coroutines is deep enough.

The use of the symmetric transfer may avoid this problem. When the `await_suspend` function returns a `std::coroutine_handle`, the compiler will automatically use this handle to resume its coroutine after `await_suspend` returns (and its activation frame is removed from the stack). Or, when a `std::noop_coroutine_handle` is returned, the execution will be passed to the caller.

Symmetric transfer solution can also be used to resume the current coroutine (by returning handle passed as the parameter). However, in such cases, conditional suspension can be a more optimal solution.

This rule raises an issue on `await_suspend` functions that could use symmetric transfer.

Noncompliant Code Example

```
struct InvokeOtherAwaiter {
  /* .... */
  void await_suspend(std::coroutine_handle<PromiseType> curre
    if (auto other = current.promise().other_handle) {
      other.resume(); // Noncompliant
    }
  }
};

struct WaitForAwaiter {
  Event& event;
  /* .... */
  void await_suspend(std::coroutine_handle<> current) {
    if (bool ready = event.register_callback(current)) {
      current.resume(); // Noncompliant
    }
  }
};

struct BufferedExecutionAwaiter {
  std::queue<std::coroutine_handle<>>& taskQueue;
  /* .... */
  void await_suspend(std::coroutine_handle<> current) {
    if (taskQueue.empty()) {
      current.resume(); // Noncompliant
    }
    auto next = taskQueue.front();
    taskQueue.pop();
    taskQueue.push(current);
    next.resume(); // Noncompliant
  }
```

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

```
};
```

Compliant Solution

```
struct InvokeOtherAwaiter {
  /* .... */
  std::coroutine_handle<> await_suspend(std::coroutine_handle
    if (auto other = current.promise().other_handle) {
      return other;
    } else {
      return std::noop_coroutine();
    }
  }
};

struct WaitForAwaiter {
  Event& event;
  /* .... */
  std::coroutine_handle<> await_suspend(std::coroutine_handle
    if (bool ready = event.register_callback(current)) {
      return current;
    } else {
      return std::noop_coroutine()
    }
  }
  // Alternatively
  bool await_suspend(std::coroutine_handle<> current) {
    return !event.register_callback(current);
  }
};

struct BufferedExecutionAwaiter {
  std::queue<std::coroutine_handle<>>& taskQueue;
  /* .... */
  std::coroutine_handle<> await_suspend(std::coroutine_handle
    if (taskQueue.empty()) {
       return current;
    }
    auto next = list.front();
    taskQueue.pop();
    taskQueue.push(current);
    return next;
  }
};
```

See

{rule:cpp:S6366} - conditionally suspending current coroutine in optimal way