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

Exceptions should not be thrown in "noexcept" functions

Analyze your code

Code Smell | Critical | error-handling bad-practice pitfall

`noexcept` is a specifier that can be applied to a function declaration to state whether or not this function might throw an exception.

This specifier is a crucial information for the compiler as it enables it to perform automatic optimizations. It is also used by the `noexcept` operator, so that a developer can know whether an expression can throw, and adapt the code accordingly (for instance, to decide to `move` or `copy` an object).

When a function is specified `noexcept`, the compiler does not generate any code to throw exceptions and any uncaught exception will result in a call to std::terminate. This means that writing a `noexcept` function is an implicit agreement to the statement : "my program will terminate if any exception is thrown inside this function".

It is a very strong commitment as there are so many ways to get an exception including any dynamic allocation.

This rule raises an issue when an exception is thrown, directly or indirectly, from a function declared `noexcept`.

Noncompliant Code Example

```
#include <exception>
#include <memory>

using namespace std;

class SafetyException {};
class Engine {};
unique_ptr<Engine> engine;

bool safety_check() noexcept;
void other_checks();

void critical_checks() {
  if (!safety_check()) {
    throw SafetyException{};
  }
}

void do_checks() {
  critical_checks(); // can throw
  other_checks(); // can throw
}

void init() noexcept(true) { // noncompliant because...
  do_checks(); // can throw
  engine = std::make_unique<Engine>(); // can throw
}
```

Compliant Solution

```
#include <exception>
#include <memory>

using namespace std;

class SafetyException {};
```

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

```
class Engine {};
unique_ptr<Engine> engine;

bool safety_check();
void other_checks();

void critical_checks() {
  if (!safety_check()) {
    throw SafetyException{};
  }
}

void do_checks() {
  critical_checks();
  other_checks();
}

void init() noexcept(true) { // compliant because ...
  try {
    do_checks(); // exception caught
    engine = std::make_unique<Engine>(); // exception caught
  } catch(std::exception e) {
    std::terminate();
  }
}
```

Exceptions

Destructors are not handled by this rule because there is a specific rule about exceptions in destructors (see ExceptionInDestructor).

See

- C++ noexcept and move constructors effect on performance in STL containers

Available In:

sonarlint | sonarcloud | sonarqube Developer Edition