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

"std::to_address" should be used to convert iterators to raw pointers

Analyze your code

Code Smell | Minor | since-c++20 confusing suspicious

For the integration with the C or just older APIs, it may be useful to convert a contiguous iterator to a raw pointer to the element. In C++20 `std::to_address` was introduced to perform this operation on both iterators and smart pointers, which supersedes non-portable and potentially buggy workarounds, that were required before:

- The first option was to take the address of the element pointed by the iterator: `&*it`. However, this operation has undefined behavior if the iterator is not pointing to any element. This may happen for the iterator returned by a call to `end()` on the container. This may also be the case when we need the address to construct a new object (via placement new) at the location pointed to by the iterator. `std::to_address(it)` works in such cases.
- The second option was to exploit the nature of `operator->` overloading and call it explicitly on the iterator: `it.operator->()`. This option avoids the pitfalls of the previous one, at the cost of not being portable. It would fail on the implementations that use raw-pointers as iterators for contiguous ranges like `std::vector` or `std::span`. Moreover, it is confusing, as this functional notation syntax for operators is rarely used.

While both `std::to_address and above workarounds, can be always used to get the address of the element that the iterator is pointing to (if any), incrementing or decrementing may have undefined behavior. Performing pointer arithmetic on pointer to elements is safe only in the case of contiguous iterators (e.g. iterators of `std::vector`, `std::array`, `std::span`, `std::string` or `std::string_view`).

This rule raises an issue when dereferencing a pointer-like object is immediately followed by taking the address of the result (`&*x` or `std::addressof(*x)`) or when `operator->` is called through an explicit functional notation (`x.operator->()`).

Noncompliant Code Example

```
void check(int* b, int* e);

void func1(std::vector<int>& v) {
    check(v.begin().operator->(), v.end().operator->()); // N
}

void func2(span<int> s) {
     check(&*s.begin(), &*s.end()); // Noncompliant
}
```

Compliant Solution

```
void func1(std::vector<int>& v) {
    check(std::to_address(v.begin()), std::to_address(v.end()
}

void func2(span<int> s) {
     check(std::to_address(s.begin()), std::to_address(s.end(
}
```

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