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

Comparision operators ("<=>", "==") should be defaulted unless non-default behavior is required

Analyze your code

Code Smell Major Quick Fix since-c++20 pitfall

Comparison operators like == or <=>, despite being not hard to write, remain a source of bugs as they need to be updated with every change in the class's member list. For instance, if a newly introduced member in the class is not considered by the operation, the issue will only manifest if two instances are identical, except for the newly introduced member. As a consequence, this type of bug is usually hard to spot.

C++20 introduced the ability to define both `operator<=>` and `operator==` as defaulted (`= default`) to indicate that they should consider all members in the order of their declaration. This not only makes code concise but also makes all the comparison operators resilient to the changes to the list of members. Thanks to operator rewriting, all other comparison operations (`!=`, <, >, <=, =>) can also rely on these robust operators.

Furthermore, when `operator<=>` is defined as defaulted, the compiler will generate a defaulted version of `operator==` if no other version is declared.

This rule raises an issue when the implementation of `operator<=>` or `operator==` has an equivalent semantic to the defaulted implementation.

Noncompliant Code Example

```
struct Comparable {
    int x;
    int y;
};

bool operator==(const Comparable& lhs, const Comparable& rhs)
    return lhs.x == rhs.x && lhs.y == rhs.y;
}

struct Ordered {
    int x;
    int y;
};

bool operator==(const Ordered& lhs, const Ordered& rhs) { //
    return lhs.x == rhs.x && lhs.y == rhs.y;
}

auto operator<=>(const Ordered& lhs, const Ordered& rhs) { //
    if (res = lhs.x <=> rhs.x; res != 0)
        return x;
    return lhs.y <=> rhs.y;
}
```

Compliant Solution

```
struct Comparable {
    int x;
    int y;

friend bool operator==(const Comparable&, const Comparable&)
};

struct Ordered {
    int x;
```

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

```
    int y;

friend auto operator<=>(const Ordered&, const Ordered&) = def
};
```

See

- {rule:cpp:S6186} - removing redundant comparison operators
- {rule:cpp:S6187} - replacing multiple comparison operators with `operator<=>`

Available In:
sonarlint sonarcloud sonarqube Developer Edition