sonar RULES

Products

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

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

Relational and subtraction operators should not be used with pointers to different arrays

Analyze your code

Bug Critical cppcoreguidelines based-on-misra

Attempting to make a comparison between pointers using >, >=, < or <= will produce undefined behavior if the two pointers point to different arrays.

Additionally, directly comparing two arrays for equality or inequality has been deprecated in C++.

However, equality or inequality between an array and a pointer is still valid

Noncompliant Code Example

```
void f1 ( )
{
  int a1[ 10 ];
  int a2[ 10 ];
  int * p1 = a1;
  if ( p1 < a2 ) // Non-compliant, p1 and a2 point to differe
  {
  }
  if ( p1 - a2 > 0 ) // Non-compliant, p1 and a2 point to dif
  {
  }
  if ( a1 == a2) // Non-compliant (in C++). Comparing differe
  {
  }
}
```

Compliant Solution

```
void f1 ( )
{
  int a1[ 10 ];
  int * p1 = a1;
  if ( p1 < a1 ) // Compliant, p1 and a1 point to the same ar
  {
  }
  if ( p1 - a1 > 0 )  // Compliant, p1 and a1 point to the sa
  {
  }
  if ( p1 == a2 ) // Compliant, comparing a pointer and an ar
  {
  }
}
```

See

- MISRA C:2004, 17.3 - >, >=, <, <= shall not be applied to pointer types except where they point to the same array.
- MISRA C++:2008, 5-0-18 - >, >=, <, <= shall not be applied to objects of pointer type, except where they point to the same array.
- C++ Core Guidelines ES.62 - Don't compare pointers into different arrays

Available In:

sonarlint sonarcloud sonarqube Developer Edition