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

Function templates should not be specialized

Analyze your code

Code Smell | Blocker | cppcoreguidelines | based-on-misra

Explicit specializations of function templates are not considered in overload resolution, only the main template. As a consequence, the function that will be selected might very well be different from what seems natural to the developer, leading to hard to understand bugs. Moreover, function templates don't allow partial specialization.

Instead of specializing a function template, you may choose to overload it with another template or non template function, since a more specialized overload will be preferred to a generic overload.

Noncompliant Code Example

```
template <typename T> void f ( T );
template <> void f<char*> ( char * ); // explicit specializat
```

Compliant Solution

```
template <typename T> void f ( T );
void f( char * ); // overload, compliant
```

Exceptions

This rule ignores cases where none of the main function template arguments depend on a template parameter: Even if the code could still be written without function template specialization (by deferring the real work to a class template, and offering specializations of this class template as customization point to the user), there is no risk of confusion for overload resolution in these cases.

```
// For real code, use std::numeric_limits instead...
template <class T> T max();
template <> float max<float>() { return FLT_MAX; }  // Ignore

template<class T>
bool isMax(T t){
    return t == max<T>();
}
```

See

- MISRA C++:2008, 14-8-1
- C++ Core Guidelines T.144 - Don't specialize function templates

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