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

"try_lock", "lock" and "unlock" should not be directly used for mutexes

Analyze your code

Code Smell | Critical | cppcoreguidelines bad-practice since-c++11 pitfall

Mutexes are synchronization primitives that allow to manage concurrency using a mechanism of `lock/unlock`.

While explicitly locking or unlocking a *mutex* is possible, it is error prone. And this is particularly true in complex code paths (or with exceptions) where it is easy to have a mismatch between `locks` and `unlocks`.

As a result, *mutexes* should not be locked or unlocked manually.

Adopting the C++ RAII idiom solves this problem by creating an object that will lock the *mutex* on creation and unlock it on destruction. Furthermore, using this idiom can also greatly improve the readability of the code.

Several classes are available as RAII wrappers:

- `std::scoped_lock` is the default, most efficient wrapper for simple cases (only available since C++17)
- `std::lock_guard` is similar to `std::scoped_lock`, but with less features. It should only be used if you don't have access to `std::scoped_lock`.
- `std::unique_lock` allows more manual unlocking/locking again, and should only be used when these features are needed, for instance with condition variables.

Noncompliant Code Example

```
#include <mutex>

class DataItem;

class DataStore {
public:
  bool store(const DataItem &dataItem);
  bool has(const DataItem &dataItem);
};

DataStore sharedDataStore;
std::mutex sharedDataStoreMutex;

bool storeIfRelevantInSharedContext(const DataItem &dataItem)
  sharedDataStoreMutex.lock(); // Noncompliant
  if (sharedDataStore.has(dataItem)) {
    sharedDataStoreMutex.unlock(); // Noncompliant
    return false;
  }
  bool result = sharedDataStore.store(dataItem);
  sharedDataStoreMutex.unlock(); // Noncompliant
  return result;
}
```

Compliant Solution

```
#include <mutex>

class DataItem;

class DataStore {
public:
```

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

```
  bool store(const DataItem &dataItem);
  bool has(const DataItem &dataItem);
};

DataStore sharedDataStore;
std::mutex sharedDataStoreMutex;

bool storeIfRelevantInSharedContext(const DataItem &dataItem)
  std::scoped_lock<std::mutex> lock(sharedDataStoreMutex);
  if (sharedDataStore.has(dataItem)) {
    return false;
  }
  return sharedDataStore.store(dataItem);
}
```

See

- C++ Core Guidelines CP.20 - Use RAII, never plain lock()/unlock()

Available In:

sonarlint sonarcloud sonarqube Developer Edition