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

rvalue reference members should not be copied accidentally

Analyze your code

Code Smell Major performance since-c++11 pitfall

C++11 introduced the concept of *forwarding-reference*, as a way to transfer values efficiently. In combination with `std::forward`, their usage allows passing values without unnecessary copies.

The expression `std::forward<T>(obj).mem`, can be used to forward the value of the member, according to the type of `obj`: move the value of member `mem` if the `obj` is an rvalue reference and copy it otherwise. However, in the corner case, when the member `mem` is of rvalue reference type, the value it references will be copied even if `obj` itself is an rvalue, the referenced value will not be moved.

Similarly for `std::move`: if `mem` is of rvalue reference type, `std::move(obj).mem` will copy the value referenced by `mem`.

This rule raises issues when a templates is instantiated with a type that leads to an accidental copy of members of forwarded objects.

Noncompliant Code Example

```
template<typename... Ts>
void consume(Ts&&... ts)


template<typename T, typename U>
void consumePair(std::pair<T, U>&& p) {
  consume(std::move(p).first, std::move(p).second); // Noncom
}
void use1() {
  std::string x = "x", y = "y";
  std::pair<std:string&&, std::string&&> rRefPair(std::move(x
  consumePair(std::move(rRefPair)); // Triggers noncompliant
                                    // with T = std:::string&
}


template<typename Pair>
void forwardPair(Pair&& p) {
  consume(std::forward<Pair>(p).first, std::forward<Pair>(p).
}
void use2() {
  std::string x = "x", y = "y";
  std::pair<std:string&&, std::string&&> rRefPair(std::move(x
  forwardPair(rRefPair); // OK, lvalue is passed, and the mem
                         // Pair = std::pair<std:string&&, st
  forwardPair(std::move(rRefPair)); // Triggers noncompliant
                                    // with Pair = std::pair<
}


template<typename Pair>
void forwardStruct(T&& p) {
  consume(std::forward<T>(p).mem); // Noncompliant (see later
}
struct Proxy {
    std::vector<int>&& mem;
};
void use3() {
  std::vector<int> v;
  Proxy proxy{std::move(v)};
  forwardStruct(proxy); // OK, lvalue is passed, and the memb
                        // T = Proxy&
```

🐞 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

```
  forwardStruct(std::move(proxy)); // Triggers noncompliant i
                                   // with T = Proxy
}


void compiler_error() {
  std::unique_ptr<int> u;
  std::pair<std::unique_ptr<int>&&, int> pair(std::move(u), 1
  // std::unique_ptr<int> u2 = std::move(pair).first; // ill-
}
```

Compliant Solution

```
template<typename T, typename U>
void consumePair(std::pair<T, U>&& p) {
    consume(std::get<0>(std::move(p)), std::get<1>(std::move(
}


template<typename Pair>
void forwardPair(Pair&& p) {
    consume(std::get<0>(std::forward<Pair>(p)), std::get<1>(s
}


template<typename Pair>
void forwardStruct(T&& t) {
  constexpr bool isMoveOfRvalueReferenceMember
      = std::is_rvalue_reference_v<decltype(t.mem)> && std::i
  if constexpr (isMoveOfRvalueReferenceMember) {
    consume(std::move(t.mem));
  } else {
    consume(std::forward<T>(t).mem);
  }
}


void compiler_error() {
  std::unique_ptr<int> u;
  std::pair<std::unique_ptr<int>&&, int> pair(std::move(u), 1
  std::unique_ptr<int> u2 = std::move(pair.first);
}
```

Available In:

sonarlint sonarcloud sonarqube Developer Edition