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

Emplacement should be prefered when insertion creates a temporary with sequence containers

Analyze your code

Code Smell | Major | Quick Fix | performance since-c++11 clumsy

In some cases, `emplace_back` is more efficient and less verbose than `push_back`. It is expected to be faster when the object is constructed into the container instead of being constructed then assigned. This also happens when the pushed object has a different type from the one held by the container.

This rule supports standard sequence containers: `std::vector`, `std::list`, `std::deque`, `std::forward_list`, `std::stack`, `std::queue` and `std::priority_queue`.

An issue will only be raised when an insertion function on a supported container leads to the construction of a large temporary object that can be avoided by using the provided emplacement member function.

Noncompliant Code Example

```
class Circle { // Large object
std::string s;
int x;
int y;
int radius;
public:
  Circle(int x, int y, int radius);
}

void f() {
  std::vector<std::pair<int, std::string>> vec1;
  std::string s;
  vec1.push_back(std::make_pair(21, s)); // Noncompliant
  std::vector<std::string> vec2;
  vec2.push_back("randomStr"); // Noncompliant, conversion fr
  std::vector<Circle> circles;
  circles.push_back(Circle{2, 42, 10}); // Noncompliant
}
```

Compliant Solution

```
void f() {
  std::vector<std::pair<int, std::string>> vec1;
  std::string s;
  vec1.emplace_back(21, s); // Compliant
  std::vector<std::string> vec2;
  vec2.emplace_back("randomStr"); // Compliant
  std::vector<Circle> circles;
  circles.emplace_back(2, 42, 10); // Compliant
}
```

Exceptions

- When `emplace_back` isn't exception-safe. When emplacing in a container of smart pointers a raw new expression, the memory will be leaked if `emplace_back` throws an exception.

See

- Effective modern C++ item 42: Consider emplacement instead of insertion.

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

Available In:

sonarlint | sonarcloud | sonarqube Developer Edition