


-  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

Incomplete types should not be deleted

Analyze your code

 Code Smell  Major  unpredictable

When calling `delete` on an object of incomplete type, the calling code does not have enough information to do the action properly (it does not know if this object has a trivial or a nontrivial destructor, if it has overloaded the `delete` operator...). Therefore, deleting a pointer to such an object can lead to undefined behavior.

Noncompliant Code Example

```
class Body;

class Handle {
public:
    Handle();
    ~Handle() {
        delete impl; // Noncompliant, Body is incomplete
    }
private:
    Body * impl;
};
```

Compliant Solution

```
// In header file
class Body;

class Handle {
public:
    Handle();
    ~Handle();
    // Add other special member functions to respect the rule
private:
    Body * impl;
};

// In implementation file
#include "Handle.h"
#include "Body.h" // Now Body is complete

Handle::~~Handle(){
    delete impl; // Compliant, at this point "Body" is a complete type
}
```

Or, with modern resource handling:

```
// In header file
class Body;

class Handle {
public:
    Handle();
    ~Handle();
private:
    std::unique_ptr<Body> impl; //Compliant
};

// In implementation file
#include "Handle.h"
```

 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
"memcpy" 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

```
#include "Body.h" // Now Body is complete

Handle::Handle() : impl{new Body{}} {}
Handle::~~Handle() = default; // since "Body" is complete, it
```

Available In:

 |  |  Developer Edition