



ABAP

Apex

C

C++

CloudFormation

COBOL

C#

CSS

Flex

Go **GO**

5 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

578 ΑII 6 Vulnerability (13) rules

R Bug (111)

• Security Hotspot

⊗ Code (436)

Quick 68 Fix

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 "std::visit" should be used to switch on the type of the current value in a "std::variant"

Analyze your code

since-c++17 clumsy

std::variant is a type-safe union that can hold values of a type out of a fixed list of types.

Depending on the current alternative inside a variant, it is common to execute dedicated code. There are basically two ways to achieve that:

- Writing code that checks the current alternative, then getting it and running specific code
- Letting std::visit perform the check and select the code to run by using overload resolution with the different alternatives

The second option is usually preferable:

- It requires less boilerplate code.
- · It is easy to handle multiple similar alternatives together if desired.
- It is usually more robust: if a new alternative is added to the variant, but the visitor does not support it, it will not compile.

This rule raises an issue when variant::index is called, or when variant::holds_alternative or variant::get_if is used in a series of if else if (calling one of these functions in isolation can be an acceptable lightweight alternative to std::visit in some cases).

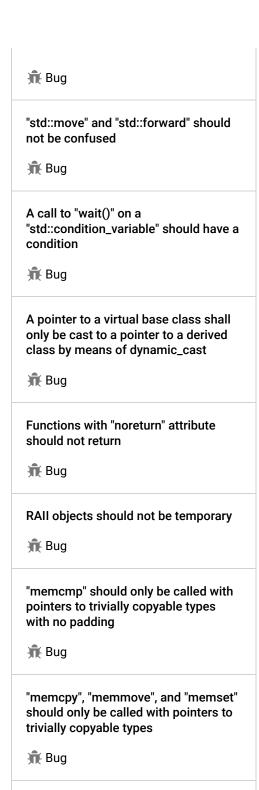
Note: When defining the visitor of a variant, it can be nicer to use a series of lambdas by making use of the overloaded pattern

Noncompliant Code Example

```
using Variant = std::variant<int, float, string>;
void printType1(Variant const &v) {
    switch(v.index()) { // Noncompliant
        case 0: cout << "int " <<get<int>(v) << "\n"; break;</pre>
        case 1: cout << "float " <<get<float>(v) << "\n"; bre</pre>
        case 2: cout << "string " <<get<string>(v) << "\n";br</pre>
}
void printType2(Variant const &v) {
    if(auto p = get_if<int>(&v)) { // Noncompliant
        cout << "int " << *p << "\n";
    } else if (auto p = get_if<float>(&v)) {
        cout << "float " << *p << "\n";
    } else if (auto p = get_if<string>(&v)) {
        cout << "string " << *p << "\n";</pre>
}
```

Compliant Solution

```
using Variant = std::variant<int, float, string>;
struct VariantPrinter {
    void operator() (int i) { cout << "int " << i << "\n"; }</pre>
    void operator() (float f) { cout << "float " << f << "\n"</pre>
    void operator() (std::string const &s) { cout << "string</pre>
};
void printType3(Variant const &v) {
    std::visit(VariantPrinter{}, v);
```



"std::auto_ptr" should not be used

Destructors should be "noexcept"

📆 Bug

📆 Bug

```
// Same principle, but using the overloaded pattern
void printType4(Variant const &v) {
    std::visit(overloaded{
        [](int i){cout << "int " << i << "\n";},
        [](float f){cout << "float " << f << "\n";},
        [](std::string const &s){cout << "string " << s << "\
        }, v);
}

Available In:
sonarlint  sonarcloud  sonarqube  Developer Edition
```

© 2008-2022 SonarSource S.A., Switzerland. All content is copyright protected. SONAR, SONARSOURCE, SONARLINT, SONARQUBE and SONARCLOUD are trademarks of SonarSource S.A. All other trademarks and copyrights are the property of their respective owners. All rights are expressly reserved.

Privacy Policy