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

Use discriminated unions or "std::variant"

Analyze your code

Code Smell   Critical   clumsy pitfall

In order to save memory, unions allow you to use the same memory to store objects from a list of possible types as long as one object is stored at a time. Unions are not inherently safe, as they expect you to externally keep track of the type of value they currently hold.

Wrong tracking has the potential to corrupt memory or to trigger undefined behaviors.

A straightforward way to avoid it is storing the information about the currently active alternative along with the union. Here follow suggested patterns to do that:

```
typedef int   altType1;
typedef float altType2;

// Pattern 1
union alternativesCommonStartingFieldPattern {
  struct {
    bool isAlt1;
    altType1 a1;
  } one;

  struct {
    bool isAlt1;
    altType2 a2;
  } two;
};

double getValueAsDouble(alternativesCommonStartingFieldPatter
  return pattern1->one.isAlt1?pattern1->one.a1:pattern1->two.
}
```

This pattern uses the fact that when two alternatives of a standard layout union are standard-layout-structs that share a common initial sequence, it is allowed to read this common initial sequence on one alternative even if the other alternative is the one currently active. This is commonly used to limit the number of bits required to store the discriminant.

```
// Pattern 2
struct wrappedUnionPattern {
  enum {ALTTYPE1, ALTTYPE2} type;

  union {
    altType1 a1;
    altType2 a2;
  };
};

double getValueAsDouble(wrappedUnionPattern *pattern2) {
  return (pattern2->type==wrappedUnionPattern::ALTTYPE1)?patt
}
```

This pattern is more straightforward, and wraps the union inside a structure that will also store the discriminant. Note that in this case, the union itself can be anonymous.

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

```
// Pattern 3 (C++17)
using stdVariantPattern = std::variant<altType1, altType2>;

double getValueAsDouble(stdVariantPattern *pattern3) {
  return std::visit([](auto&& alternative) -> double { return
}
```

This pattern relies on C++17's `std::variant` to store the alternative.

In general, `std::variant` is:

- Safer as the type of the current value is always known and checked before usage.
- More practical as it can have members of any type, including non trivial types (see {rule:cpp:S6025}). It also supports redundant types, which is useful when alternatives have the same type with different semantic meanings.
- Easier to use as it provides many member/helper functions.

One noticeable difference with unions is that the alternatives in a `std::variant` do not have a name. You can access them by type or by index, using `std::get` (throws if the wrong alternative is accessed) or `std::get_if` (returns a null pointer if the wrong alternative is used). But very often, instead of accessing a specific alternative, visitors are used to distinguish cases of the variant.

This rule raises an issue when unions are used outside of the 3 suggested patterns.

Noncompliant Code Example

```
void rawUnion() {
  union IntOrDouble { // Noncompliant: union is not wrapped
    int i;
    double d;
  };
  IntOrDouble intOrDouble;
  intOrDouble.d = 10.5;
}
```

Compliant Solution

```
struct IntOrChar {
  enum { INT, CHAR } tag;
  union { // Compliant
    int i;
    char c;
  };
};

void simpleVariant() {
  std::variant<int, double> intOrDouble = 10.5; // Compliant
}{code}
```

Available In:

sonarlint sonarcloud sonarqube Developer Edition