MEM56-CPP. Do not store an already-owned pointer value in an unrelated smart pointer

Smart pointers such as `std::unique_ptr` and `std::shared_ptr` encode pointer ownership semantics as part of the type system. They wrap a pointer value, provide pointer-like semantics through `operator *()` and `operator->()` member functions, and control the lifetime of the pointer they manage. When a smart pointer is constructed from a pointer value, that value is said to be *owned* by the smart pointer.

Calling `std::unique_ptr::release()` will relinquish ownership of the managed pointer value. Destruction of, move assignment of, or calling `std::unique_ptr::reset()` on a `std::unique_ptr` object will also relinquish ownership of the managed pointer value, but results in destruction of the managed pointer value. If a call to `std::shared_ptr::unique()` returns true, then destruction of or calling `std::shared_ptr::reset()` on that `std::shared_ptr` object will relinquish ownership of the managed pointer value but results in destruction of the managed pointer value.

Some smart pointers, such as `std::shared_ptr`, allow multiple smart pointer objects to manage the same underlying pointer value. In such cases, the initial smart pointer object owns the pointer value, and subsequent smart pointer objects are related to the original smart pointer. Two smart pointers are *related* when the initial smart pointer is used in the initialization of the subsequent smart pointer objects. For instance, copying a `std::shared_ptr` object to another `std::shared_ptr` object via copy assignment creates a relationship between the two smart pointers, whereas creating a `std::shared_ptr` object from the managed pointer value of another `std::shared_ptr` object does not.

Do not create an unrelated smart pointer object with a pointer value that is owned by another smart pointer object. This includes resetting a smart pointer's managed pointer to an already-owned pointer value, such as by calling `reset()`.

Noncompliant Code Example

In this noncompliant code example, two unrelated smart pointers are constructed from the same underlying pointer value. When the local, automatic variable `p2` is destroyed, it deletes the pointer value it manages. Then, when the local, automatic variable `p1` is destroyed, it deletes the same pointer value, resulting in a double-free vulnerability.

```
#include <memory>

void f() {
  int *i = new int;
  std::shared_ptr<int> p1(i);
  std::shared_ptr<int> p2(i);
}
```

Compliant Solution

In this compliant solution, the `std::shared_ptr` objects are related to one another through copy construction. When the local, automatic variable `p2` is destroyed, the use count for the shared pointer value is decremented but still nonzero. Then, when the local, automatic variable `p1` is destroyed, the use count for the shared pointer value is decremented to zero, and the managed pointer is destroyed. This compliant solution also calls `std::make_shared()` instead of allocating a raw pointer and storing its value in a local variable.

```
#include <memory>

void f() {
  std::shared_ptr<int> p1 = std::make_shared<int>();
  std::shared_ptr<int> p2(p1);
}
```

Noncompliant Code Example

In this noncompliant code example, the `poly` pointer value owned by a `std::shared_ptr` object is cast to the `D *` pointer type with `dynamic_cast` in an attempt to obtain a `std::shared_ptr` of the polymorphic derived type. However, this eventually results in undefined behavior as the same pointer is thereby stored in two different `std::shared_ptr` objects. When `g()` exits, the pointer stored in `derived` is freed by the default deleter. Any further use of `poly` results in accessing freed memory. When `f()` exits, the same pointer stored in `poly` is destroyed, resulting in a double-free vulnerability.

```
#include <memory>

struct B {
  virtual ~B() = default; // Polymorphic object
  // ...
};
struct D : B {};

void g(std::shared_ptr<D> derived);

void f() {
  std::shared_ptr<B> poly(new D);
  // ...
  g(std::shared_ptr<D>(dynamic_cast<D *>(poly.get())));
  // Any use of poly will now result in accessing freed memory.
}
```

Compliant Solution

In this compliant solution, the `dynamic_cast` is replaced with a call to `std::dynamic_pointer_cast()`, which returns a `std::shared_ptr` of the polymorphic type with the valid shared pointer value. When `g()` exits, the reference count to the underlying pointer is decremented by the destruction of `derived`, but because of the reference held by `poly` (within `f()`), the stored pointer value is still valid after `g()` returns.

```
#include <memory>

struct B {
  virtual ~B() = default; // Polymorphic object
  // ...
};
struct D : B {};

void g(std::shared_ptr<D> derived);

void f() {
  std::shared_ptr<B> poly(new D);
  // ...
  g(std::dynamic_pointer_cast<D, B>(poly));
  // poly is still referring to a valid pointer value.
}
```

Noncompliant Code Example

In this noncompliant code example, a `std::shared_ptr` of type S is constructed and stored in `s1`. Later, `S::g()` is called to get another shared pointer to the pointer value managed by `s1`. However, the smart pointer returned by `S::g()` is not related to the smart pointer stored in `s1`. When `s2` is destroyed, it will free the pointer managed by `s1`, causing a double-free vulnerability when `s1` is destroyed.

```
#include <memory>

struct S {
  std::shared_ptr<S> g() { return std::shared_ptr<S>(this); }
};

void f() {
  std::shared_ptr<S> s1 = std::make_shared<S>();
  // ...
  std::shared_ptr<S> s2 = s1->g();
}
```

Compliant Solution

The compliant solution is to use `std::enable_shared_from_this::shared_from_this()` to get a shared pointer from S that is related to an existing `std::shared_ptr` object. A common implementation strategy is for the `std::shared_ptr` constructors to detect the presence of a pointer that inherits from `std::enable_shared_from_this`, and automatically update the internal bookkeeping required for `std::enable_shared_from_this::shared_from_this()` to work. Note that `std::enable_shared_from_this::shared_from_this()` requires an existing `std::shared_ptr` instance that manages the pointer value pointed to by `this`. Failure to meet this requirement results in undefined behavior, as it would result in a smart pointer attempting to manage the lifetime of an object that itself does not have lifetime management semantics.

```
#include <memory>

struct S : std::enable_shared_from_this<S> {
  std::shared_ptr<S> g() { return shared_from_this(); }
};

void f() {
  std::shared_ptr<S> s1 = std::make_shared<S>();
  std::shared_ptr<S> s2 = s1->g();
}
```

Risk Assessment

Passing a pointer value to a deallocation function that was not previously obtained by the matching allocation function results in undefined behavior, which can lead to exploitable vulnerabilities.

Rule	Severity	Likelihood	Remediation Cost	Priority	Level
MEM56-CPP	High	Likely	Medium	**P18**	**L1**

Automated Detection

Tool	Version	Checker	Description
Parasoft C/C++test	10.4.2	**CERT_CPP-MEM56-a**	Do not store an already-owned pointer value in an unrelated smart pointer
PVS-Studio	6.23	**V1006**	

Related Vulnerabilities

Search for other vulnerabilities resulting from the violation of this rule on the CERT website.

Related Guidelines

SEI CERT C++ Coding Standard	MEM50-CPP. Do not access freed memory MEM51-CPP. Properly deallocate dynamically allocated resources
MITRE CWE	CWE-415, Double Free CWE-416, Use After Free CWE 762, Mismatched Memory Management Routines

Bibliography

[ISO/IEC 14882-2014]	Subclause 20.8, "Smart Pointers"