EXP54-CPP. Do not access an object outside of its lifetime

Every object has a lifetime in which it can be used in a well-defined manner. The lifetime of an object begins when sufficient, properly aligned storage has been obtained for it and its initialization is complete. The lifetime of an object ends when a nontrivial destructor, if any, is called for the object and the storage for the object has been reused or released. Use of an object, or a pointer to an object, outside of its lifetime frequently results in undefined behavior.

The C++ Standard, [basic.life], paragraph 5 [ISO/IEC 14882-2014], describes the lifetime rules for pointers:

Before the lifetime of an object has started but after the storage which the object will occupy has been allocated or, after the lifetime of an object has ended and before the storage which the object occupied is reused or released, any pointer that refers to the storage location where the object will be or was located may be used but only in limited ways. For an object under construction or destruction, see 12.7. Otherwise, such a pointer refers to allocated storage, and using the pointer as if the pointer were of type void*, is well-defined. Indirection through such a pointer is permitted but the resulting lvalue may only be used in limited ways, as described below. The program has undefined behavior if:

- the object will be or was of a class type with a non-trivial destructor and the pointer is used as the operand of a delete-expression,
- the pointer is used to access a non-static data member or call a non-static member function of the object, or
- the pointer is implicitly converted to a pointer to a virtual base class, or
- the pointer is used as the operand of a static_cast, except when the conversion is to pointer to cv void, or to pointer to cv void and subsequently to pointer to either cv char or cv unsigned char, or
- the pointer is used as the operand of a dynamic_cast.

Paragraph 6 describes the lifetime rules for non-pointers:

Similarly, before the lifetime of an object has started but after the storage which the object will occupy has been allocated or, after the lifetime of an object has ended and before the storage which the object occupied is reused or released, any glvalue that refers to the original object may be used but only in limited ways. For an object under construction or destruction, see 12.7. Otherwise, such a glvalue refers to allocated storage, and using the properties of the glvalue that do not depend on its value is well-defined. The program has undefined behavior if:

- an lvalue-to-rvalue conversion is applied to such a glvalue,
- the glvalue is used to access a non-static data member or call a non-static member function of the object, or
- the glvalue is bound to a reference to a virtual base class, or
- the glvalue is used as the operand of a dynamic_cast or as the operand of typeid.

Do not use an object outside of its lifetime, except in the ways described above as being well-defined.

Noncompliant Code Example

In this noncompliant code example, a pointer to an object is used to call a non-static member function of the object prior to the beginning of the pointer’s lifetime, resulting in undefined behavior.

```cpp
struct S {
    void mem_fn();
};

void f() {
    S *s;
    s->mem_fn();
}
```

Compliant Solution

In this compliant solution, storage is obtained for the pointer prior to calling S::mem_fn().

```cpp
struct S {
    void mem_fn();
};

void f() {
    S *s = new S;
    s->mem_fn();
    delete s;
}
```
An improved compliant solution would not dynamically allocate memory directly but would instead use an automatic local variable to obtain the storage and perform initialization. If a pointer were required, use of a smart pointer, such as `std::unique_ptr`, would be a marked improvement. However, these suggested compliant solutions would distract from the lifetime demonstration of this compliant solution and consequently are not shown.

Noncompliant Code Example

In this noncompliant code example, a pointer to an object is implicitly converted to a virtual base class after the object’s lifetime has ended, resulting in **undefined behavior**.

```cpp
struct B {};
struct D1 : virtual B {};
struct D2 : virtual B {};
struct S : D1, D2 {};
void f(const B *b) {}
void g() {
  S *s = new S;
  // Use s
  delete s;
  f(s);
}
```

Despite the fact that `f()` never makes use of the object, its being passed as an argument to `f()` is sufficient to trigger undefined behavior.

Compliant Solution

In this compliant solution, the lifetime of `s` is extended to cover the call to `f()`.

```cpp
struct B {};
struct D1 : virtual B {};
struct D2 : virtual B {};
struct S : D1, D2 {};
void f(const B *b) {}
void g() {
  S *s = new S;
  // Use s
  f(s);
  delete s;
}
```

Noncompliant Code Example

In this noncompliant code example, the address of a local variable is returned from `f()`. When the resulting pointer is passed to `h()`, the lvalue-to-rvalue conversion applied to `i` results in **undefined behavior**.
int *g() {  
    int i = 12;  
    return &i;  
}  

void h(int *i);  

void f() {  
    int *i = g();  
    h(i);  
}  

Some compilers generate a diagnostic message when a pointer to an object with automatic storage duration is returned from a function, as in this example.

**Compliant Solution**

In this compliant solution, the local variable returned from \(g()\) has static storage duration instead of automatic storage duration, extending its lifetime sufficiently for use within \(f()\).  

```
int *g() {  
    static int i = 12;  
    return &i;  
}  

void h(int *i);  

void f() {  
    int *i = g();  
    h(i);  
}
```

**Noncompliant Code Example**

A `std::initializer_list<>` object is constructed from an initializer list as though the implementation allocated a temporary array and passed it to the constructor. This temporary array has the same lifetime as other temporary objects except that initializing a `std::initializer_list<>` object from the array extends the lifetime of the array exactly like binding a reference to a temporary [ISO/IEC 14882-2014](https://www.iso.org/standard/70142.html).

In this noncompliant code example, a member variable of type `std::initializer_list<int>` is list-initialized within the constructor’s `ctor-initializer`. Under these circumstances, the conceptual temporary array’s lifetime ends once the constructor exits, so accessing any elements of the `std::initializer_list<int>` member variable results in undefined behavior.

```
#include <initializer_list>  
#include <iostream>  

class C {  
    std::initializer_list<int> l;  

public:  
    C() : l{1, 2, 3} {}  

    int first() const { return *l.begin(); }  
};  

void f() {  
    C c;  
    std::cout << c.first();  
}
```

**Compliant Solution**

In this compliant solution, the `std::initializer_list<int>` member variable is replaced with a `std::vector<int>`, which copies the elements of the initializer list to the container instead of relying on a dangling reference to the temporary array.
```cpp
#include <iostream>
#include <vector>

class C {
  std::vector<int> l;

public:
  C() : l{1, 2, 3} {}  
  int first() const { return *l.begin(); }
};

void f() {
  C c;
  std::cout << c.first();
}
```

**Noncompliant Code Example**

In this noncompliant code example, a lambda object is stored in a function object, which is later called (executing the lambda) to obtain a constant reference to a value. The lambda object returns an `int` value, which is then stored in a temporary `int` object that becomes bound to the `const int &` return type specified by the function object. However, the temporary object’s lifetime is not extended past the return from the function object’s invocation, which causes undefined behavior when the resulting value is accessed.

```cpp
#include <functional>

void f() {
  auto l = [](const int &j) { return j; };
  std::function<const int&(const int &)> fn(l);

  int i = 42;
  int j = fn(i);
}
```

**Compliant Solution**

In this compliant solution, the `std::function` object returns an `int` instead of a `const int &`, ensuring that the value is copied instead of bound to a temporary reference. An alternative solution would be to call the lambda directly instead of through the `std::function<>` object.

```cpp
#include <functional>

void f() {
  auto l = [](const int &j) { return j; };
  std::function<int(const int &)> fn(l);

  int i = 42;
  int j = fn(i);
}
```

**Noncompliant Code Example**

In this noncompliant code example, the constructor for the automatic variable `s` is not called because execution does not flow through the declaration of the local variable due to the `goto` statement. Because the constructor is not called, the lifetime for `s` has not begun. Therefore, calling `S::f()` uses the object outside of its lifetime and results in undefined behavior.

```cpp
```
class S {
    int v;

public:
    S() : v(12) {} // Non-trivial constructor
    void f();
};
void f() {
    // …
    goto bad_idea;
    // …
    S s; // Control passes over the declaration, so initialization does not take place.
    bad_idea:
    s.f();
}

Compliant Solution

This compliant solution ensures that `s` is properly initialized prior to performing the local jump.

class S {
    int v;

public:
    S() : v(12) {} // Non-trivial constructor
    void f();
};
void f() {
    S s;
    // …
    goto bad_idea;
    // …
    bad_idea:
    s.f();
}

Noncompliant Code Example

In this noncompliant code example, `f()` is called with an iterable range of objects of type `S`. These objects are copied into a temporary buffer using `std::copy()`, and when processing of those objects is complete, the temporary buffer is deallocated. However, the buffer returned by `std::getTemporaryBuffer()` does not contain initialized objects of type `S`, so when `std::copy()` dereferences the destination iterator, it results in undefined behavior because the object referenced by the destination iterator has yet to start its lifetime. This is because while space for the object has been allocated, no constructors or initializers have been invoked.
#include <algorithm>
#include <cstddef>
#include <memory>
#include <type_traits>

class S {
    int i;

public:
    S() : i(0) {}
    S(int i) : i(i) {}
    S(const S&) = default;
    S& operator=(const S&) = default;
};

template <typename Iter>
void f(Iter i, Iter e) {
    static_assert(std::is_same<typename std::iterator_traits<Iter>::value_type, S>::value,
                  "Expecting iterators over type S");
    ptrdiff_t count = std::distance(i, e);
    if (!count) {
        return;
    }

    // Get some temporary memory.
    auto p = std::get_temporary_buffer<S>(count);
    if (p.second < count) {
        // Handle error; memory wasn't allocated, or insufficient memory was allocated.
        return;
    }
    S *vals = p.first;

    // Copy the values into the memory.
    std::copy(i, e, vals);

    // ...

    // Return the temporary memory.
    std::return_temporary_buffer(vals);
}

**Implementation Details**

A reasonable implementation of `std::get_temporary_buffer()` and `std::copy()` can result in code that behaves like the following example (with error-checking elided).

```cpp
unsigned char *buffer = new (std::nothrow) unsigned char[sizeof(S) * object_count];
S *result = reinterpret_cast<S*>(buffer);
while (i != e) {
    *result = *i; // Undefined behavior
    ++result;
    ++i;
}
```

The act of dereferencing `result` is undefined behavior because the memory pointed to is not an object of type `S` within its lifetime.

**Compliant Solution (`std::uninitialized_copy()`)**

In this compliant solution, `std::uninitialized_copy()` is used to perform the copy, instead of `std::copy()`, ensuring that the objects are initialized using placement `new` instead of dereferencing uninitialized memory. Identical code from the noncompliant code example has been elided for brevity.

```cpp
//...
// Copy the values into the memory.
    std::uninitialized_copy(i, e, vals);
// ...
```
Compliant Solution (std::raw_storage_iterator)

This compliant solution uses std::copy() with a std::raw_storage_iterator as the destination iterator with the same well-defined results as using std::uninitialized_copy(). As with the previous compliant solution, identical code from the noncompliant code example has been elided for brevity.

```
// ...  
// Copy the values into the memory.
std::copy(i, e, std::raw_storage_iterator<S*, S>(vals));
// ...
```

Risk Assessment

Referencing an object outside of its lifetime can result in an attacker being able to run arbitrary code.

<table>
<thead>
<tr>
<th>Rule</th>
<th>Severity</th>
<th>Likelihood</th>
<th>Remediation Cost</th>
<th>Priority</th>
<th>Level</th>
</tr>
</thead>
<tbody>
<tr>
<td>EXP54-CPP</td>
<td>High</td>
<td>Probable</td>
<td>High</td>
<td>P6</td>
<td>L2</td>
</tr>
</tbody>
</table>

Automated Detection

<table>
<thead>
<tr>
<th>Tool</th>
<th>Version</th>
<th>Checker</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>Astére</td>
<td>22.10</td>
<td>return-reference-local</td>
<td>Partially checked</td>
</tr>
<tr>
<td></td>
<td></td>
<td>dangling_pointer_use</td>
<td></td>
</tr>
<tr>
<td>Clang</td>
<td>3.9</td>
<td>-Wdangling-initializer-list</td>
<td>Catches some lifetime issues related to incorrect use of std::initializer_list&lt;&gt;</td>
</tr>
<tr>
<td>CodeSonar</td>
<td>7.2p0</td>
<td>IO.UAC ALLOC.UAF</td>
<td>Use after close Use after free</td>
</tr>
<tr>
<td>Helix QAC</td>
<td>2022.3</td>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td></td>
<td>C++2012, C++2013, C++2014, C++2930, C++2931, C++2932, C++2933, C++2934, C++4003, C++4026</td>
<td></td>
</tr>
<tr>
<td>Klocwork</td>
<td>2022.3</td>
<td>CL.FFM.ASSIGN CL.FFM.COPY LOCRET.ARG LOCRET.GLOB LOCRET.RET UFM.DEREF.MIGHT UFM.DEREF.MUST UFM.FFM.MIGHT UFM.FFM.MUST UFM.RETURN.MIGHT UFM.RETURN.MUST UFM.USE.MIGHT UFM.USE.MUST UNINIT.HEAP.MIGHT UNINIT.HEAP.MUST UNINIT.STACK.ARRAY.MIGHT UNINIT.STACK.ARRAY.ARRAY.MUST UNINIT.STACK.ARRAY.PARTIAL.MUST UNINIT.STACK.MIGHT UNINIT.STACK.MUST</td>
<td></td>
</tr>
<tr>
<td>LDRA tool suite</td>
<td>9.7.1</td>
<td>42 D, 53 D, 77 D, 1 J, 71 S, 565 S</td>
<td>Partially implemented</td>
</tr>
<tr>
<td>Parasoft C /C++test</td>
<td>2022.1</td>
<td>CERT_CPP-EXP54-a CERT_CPP-EXP54-b CERT_CPP-EXP54-c</td>
<td>Do not use resources that have been freed Do not use resources that have been freed</td>
</tr>
<tr>
<td></td>
<td></td>
<td>The address of an object with automatic storage shall not be returned from a function The address of an object with automatic storage shall not be assigned to another object that may persist after the first object has ceased to exist</td>
<td></td>
</tr>
<tr>
<td>Parasoft Insure++</td>
<td></td>
<td></td>
<td>Runtime detection</td>
</tr>
<tr>
<td>Polyspace Bug Finder</td>
<td>R2022b</td>
<td>CERT C++: EXP54-CPP</td>
<td>Checks for: Non-initialized variable or pointer Use of previously freed pointer Pointer or reference to stack variable leaving scope Accessing object with temporary lifetime</td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td>Rule partially covered.</td>
</tr>
</tbody>
</table>
Related Vulnerabilities

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

Related Guidelines

<table>
<thead>
<tr>
<th>SEI CERT C Coding Standard</th>
<th>DCL30-C. Declare objects with appropriate storage durations</th>
</tr>
</thead>
</table>

Bibliography

<table>
<thead>
<tr>
<th>[Coverity 2007]</th>
<th>Subclause 3.8, &quot;Object Lifetime&quot;</th>
</tr>
</thead>
<tbody>
<tr>
<td>[ISO/IEC 14882-2014]</td>
<td>Subclause 8.5.4, &quot;List-Initialization&quot;</td>
</tr>
</tbody>
</table>