**EXP61-CPP. A lambda object must not outlive any of its reference captured objects**

Lambda expressions may capture objects with automatic storage duration from the set of enclosing scopes (called the reaching scope) for use in the lambda's function body. These captures may be either explicit, by specifying the object to capture in the lambda's capture-list, or implicit, by using a capture-default and referring to the object within the lambda's function body. When capturing an object explicitly or implicitly, the capture-default indicates that the object is either captured by copy (using =) or captured by reference (using &). When an object is captured by copy, the lambda object will contain an unnamed nonstatic data member that is initialized to the value of the object being captured. This nonstatic data member's lifetime is that of the lambda object's lifetime. However, when an object is captured by reference, the lifetime of the referent is not tied to the lifetime of the lambda object.

Because entities captured are objects with automatic storage duration (or this), a general guideline is that functions returning a lambda object (including returning via a reference parameter), or storing a lambda object in a member variable or global, should not capture an entity by reference because the lambda object often outlives the captured reference object.

When a lambda object outlives one of its reference-captured objects, execution of the lambda object's function call operator results in undefined behavior once that reference-captured object is accessed. Therefore, a lambda object must not outlive any of its reference-captured objects. This is a specific instance of EXP54-CPP. Do not access an object outside of its lifetime.

**Noncompliant Code Example**

In this noncompliant code example, the function `g()` returns a lambda, which implicitly captures the automatic local variable `i` by reference. When that lambda is returned from the call, the reference it captured will refer to a variable whose lifetime has ended. As a result, when the lambda is executed in `f()`, the use of the dangling reference in the lambda results in undefined behavior.

```cpp
auto g() {
    int i = 12;
    return [&] { i = 100; return i; };
}

void f() {
    int j = g()();
}
```

**Compliant Solution**

In this compliant solution, the lambda does not capture `i` by reference but instead captures it by copy. Consequently, the lambda contains an implicit nonstatic data member whose lifetime is that of the lambda.

```cpp
auto g() {
    int i = 12;
    return [=] () mutable {
        i = 100;
        return i;
    };
}

void f() {
    int j = g()();
}
```

**Noncompliant Code Example**

In this noncompliant code example, a lambda reference captures a local variable from an outer lambda. However, this inner lambda outlives the lifetime of the outer lambda and any automatic local variables it defines, resulting in undefined behavior when an inner lambda object is executed within `f()`.
auto g(int val) {
    auto outer = [val] {
        int i = val;
        auto inner = [i] {
            i += 30;
            return i;
        };
        return inner;
    };
    return outer();
}

void f() {
    auto fn = g(12);
    int j = fn();
}

Compliant Solution

In this compliant solution, the inner lambda captures i by copy instead of by reference.

auto g(int val) {
    auto outer = [val] {
        int i = val;
        auto inner = [i] {
            return i + 30;
        };
        return inner;
    };
    return outer();
}

void f() {
    auto fn = g(12);
    int j = fn();
}

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>EXP61-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>Astrée</td>
<td>22.10</td>
<td>invalid_pointer_dereference</td>
<td>Checks for situations where object escapes scope through lambda expressions (rule fully covered)</td>
</tr>
<tr>
<td>Helix QAC</td>
<td>2022.3</td>
<td>C++4706, C++4707, C++4708</td>
<td></td>
</tr>
<tr>
<td>Klocwork</td>
<td>2022.3</td>
<td>LOCRET.RET</td>
<td></td>
</tr>
<tr>
<td>Parasoft C/C++test</td>
<td>2022.1</td>
<td>CERT_CPP-EXP61-a, CERT_CPP-EXP61-b, CERT_CPP-EXP61-c</td>
<td>Never return lambdas that capture local objects by reference</td>
</tr>
<tr>
<td>Polyspace Bug Finder</td>
<td>R2022b</td>
<td>CERT C++: EXP61-CPP</td>
<td>Checks for situations where object escapes scope through lambda expressions (rule fully covered)</td>
</tr>
<tr>
<td>PVS-Studio</td>
<td>7.22</td>
<td>V1047</td>
<td></td>
</tr>
</tbody>
</table>
Related Vulnerabilities

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

Related Guidelines

| SEI CERT C++ Coding Standard | EXP54-CPP. Do not access an object outside of its lifetime |

Bibliography

<table>
<thead>
<tr>
<th>ISO/IEC 14882-2014</th>
<th>Subclause 3.8, &quot;Object Lifetime&quot;</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td>Subclause 5.1.2, &quot;Lambda Expressions&quot;</td>
</tr>
</tbody>
</table>
