ERR58-CPP. Handle all exceptions thrown before main() begins executing

Not all exceptions can be caught, even with careful use of function-try-blocks. The C++ Standard, [except.handle], paragraph 13 [ISO/IEC 14882-2014], states the following:

Exceptions thrown in destructors of objects with static storage duration or in constructors of namespace scope objects with static storage duration are not caught by a function-try-block on main(). Exceptions thrown in destructors of objects with thread storage duration or in constructors of namespace-scope objects with thread storage duration are not caught by a function-try-block on the initial function of the thread.

When declaring an object with static or thread storage duration, and that object is not declared within a function block scope, the type's constructor must be declared noexcept and must comply with ERR55-CPP. Honor exception specifications. Additionally, the initializer for such a declaration, if any, must not throw an uncaught exception (including from any implicitly constructed objects that are created as a part of the initialization). If an uncaught exception is thrown before main() is executed, or if an uncaught exception is thrown after main() has finished executing, there are no further opportunities to handle the exception and it results in implementation-defined behavior. (See ERR50-CPP. Do not abruptly terminate the program for further details.)

For more information on exception specifications of destructors, see DCL57-CPP. Do not let exceptions escape from destructors or deallocation functions.

Noncompliant Code Example

In this noncompliant example, the constructor for S may throw an exception that is not caught when globalS is constructed during program startup.

```
struct S {
    S() noexcept(false);
};
static S globalS;
```

Compliant Solution

This compliant solution makes globalS into a local variable with static storage duration, allowing any exceptions thrown during object construction to be caught because the constructor for S will be executed the first time the function globalS() is called rather than at program startup. This solution does require the programmer to modify source code so that previous uses of globalS are replaced by a function call to globalS().

```
struct S {
    S() noexcept(false);
};

S &globalS() {
    try {
        static S s;
        return s;
    } catch (...) {
        // Handle error, perhaps by logging it and gracefully terminating the application.
    }
    // Unreachable.
}
```

Noncompliant Code Example

In this noncompliant example, the constructor of global may throw an exception during program startup. (The std::string constructor, which accepts a const char * and a default allocator object, is not marked noexcept and consequently allows all exceptions.) This exception is not caught by the function-try-block on main(), resulting in a call to std::terminate() and abnormal program termination.

```
struct S {
    S() noexcept(false);
};

S &globalS() {
    try {
        static S s;
        return s;
    } catch (...) {
        // Handle error, perhaps by logging it and gracefully terminating the application.
    }
    // Unreachable.
}
```
```
#include <string>

static const std::string global("...");

int main()
try {
    // ...
} catch(...) {
    // IMPORTANT: Will not catch exceptions thrown
    // from the constructor of global
}
```

**Compliant Solution**

Compliant code must prevent exceptions from escaping during program startup and termination. This compliant solution avoids defining a `std::string` at global namespace scope and instead uses a `static const char *`.

```
static const char *global = "...";

int main() {
    // ...
}
```

**Compliant Solution**

This compliant solution introduces a class derived from `std::string` with a constructor that catches all exceptions with a function try block and terminates the application in accordance with ERR50-CPP-EX1 in ERR50-CPP. Do not abruptly terminate the program in the event any exceptions are thrown. Because no exceptions can escape the constructor, it is marked `noexcept` and the class type is permissible to use in the declaration or initialization of a static global variable.

For brevity, the full interface for such a type is not described.

```
#include <exception>
#include <string>

namespace my {
    struct string : std::string {
        explicit string(const char *msg,
            const std::string::allocator_type &alloc = std::string::allocator_type()) noexcept
            try : std::string(msg, alloc) {} catch(...) {
                extern void log_message(const char *) noexcept;
                log_message("std::string constructor threw an exception");
                std::terminate();
            }  
        } // ...
    };

    static const my::string global("...");

    int main() {
        // ...
    }
}
```

**Noncompliant Code Example**

In this noncompliant example, an exception may be thrown by the initializer for the static global variable `i`.
extern int f() noexcept(false);
int i = f();

int main() {
    // ...
}

Compliant Solution

This compliant solution wraps the call to f() with a helper function that catches all exceptions and terminates the program in conformance with ERR50-CPP-EX1 of ERR50-CPP. Do not abruptly terminate the program.

```c
#include <exception>

int f_helper() noexcept {
    try {
        extern int f() noexcept(false);
        return f();
    } catch (...) {
        extern void log_message(const char *) noexcept;
        log_message("f() threw an exception");
        std::terminate();
    }
    // Unreachable.
}

int i = f_helper();

int main() {
    // ...
}
```

Risk Assessment

Throwing an exception that cannot be caught results in abnormal program termination and can lead to denial-of-service attacks.

<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>ERR58-CPP</td>
<td>Low</td>
<td>Likely</td>
<td>Low</td>
<td>P9</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>20.10</td>
<td>potentially-throwing-static-initialization</td>
<td>Partially checked</td>
</tr>
<tr>
<td>Axivion Bauhaus Suite</td>
<td>7.2.0</td>
<td>CertC++-ERR58</td>
<td></td>
</tr>
<tr>
<td>Clang</td>
<td>3.9</td>
<td>cert-err58-cpp</td>
<td>Checked by clang-tidy</td>
</tr>
<tr>
<td>Helix QAC</td>
<td>2021.2</td>
<td>C++4634, C++4636, C++4637, C++4639</td>
<td></td>
</tr>
<tr>
<td>Parasoft C/C++test</td>
<td>2021.2</td>
<td>CERT_CPP-ERR58-a</td>
<td>Exceptions shall be raised only after start-up and before termination of the program</td>
</tr>
<tr>
<td>Polyspace Bug Finder</td>
<td>R2021b</td>
<td>CERT C++: ERR58-CPP</td>
<td>Checks for exceptions raised during program startup (rule fully covered)</td>
</tr>
<tr>
<td>PRQA QA C++</td>
<td>4.4</td>
<td>4634, 4636, 4637, 4639</td>
<td></td>
</tr>
<tr>
<td>RuleChecker</td>
<td>20.10</td>
<td>potentially-throwing-static-initialization</td>
<td>Partially checked</td>
</tr>
</tbody>
</table>

Related Vulnerabilities

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

Related Guidelines
This rule is a subset of ERR50-CPP. Do not abruptly terminate the program

<table>
<thead>
<tr>
<th>SEI CERT C++ Coding Standard</th>
<th>DCL57-CPP. Do not let exceptions escape from destructors or deallocation functions</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td>ERR55-CPP. Honor exception specifications</td>
</tr>
</tbody>
</table>

Bibliography

[ISO/IEC 14882-2014] Subclause 15.4, "Exception Specifications"

[Sutter 2000] Item 8, "Writing Exception-Safe Code—Part 1"