MSC54-CPP. A signal handler must be a plain old function

The C++ Standard, [support.runtime], paragraph 10 [ISO/IEC 14882-2014], states the following:

The common subset of the C and C++ languages consists of all declarations, definitions, and expressions that may appear in a well-formed C++ program and also in a conforming C program. A POF ("plain old function") is a function that uses only features from this common subset, and that does not directly or indirectly use any function that is not a POF, except that it may use plain lock-free atomic operations. A plain lock-free atomic operation is an invocation of a function f from Clause 29, such that f is not a member function, and either f is the function atomic_is_lock_free, or for every atomic argument A passed to f, atomic_is_lock_free(A) yields true. All signal handlers shall have C linkage. The behavior of any function other than a POF used as a signal handler in a C++ program is implementation-defined.

Footnote 228 states the following:

In particular, a signal handler using exception handling is very likely to have problems. Also, invoking std::exit may cause destruction of objects, including those of the standard library implementation, which, in general, yields undefined behavior in a signal handler.

If your signal handler is not a plain old function, then the behavior of a call to it in response to a signal is implementation-defined, at best, and is likely to result in undefined behavior. All signal handlers must meet the definition of a plain old function. In addition to the restrictions placed on signal handlers in a C program, this definition also prohibits the use of features that exist in C++ but not in C (such as non-POD [non–plain old data] objects and exceptions). This includes indirect use of such features through function calls.

Noncompliant Code Example

In this noncompliant code example, the signal handler is declared as a static function. However, since all signal handler functions must have C language linkage, and C++ is the default language linkage for functions in C++, calling the signal handler results in undefined behavior.

```cpp
#include <csignal>

static void sig_handler(int sig) {
    // Implementation details elided.
}

void install_signal_handler() {
    if (SIG_ERR == std::signal(SIGTERM, sig_handler)) {
        // Handle error
    }
}
```

Compliant Solution

This compliant solution defines sig_handler() as having C language linkage. As a consequence of declaring the signal handler with C language linkage, the signal handler will have external linkage rather than internal linkage.

```cpp
#include <csignal>

extern "C" void sig_handler(int sig) {
    // Implementation details elided.
}

void install_signal_handler() {
    if (SIG_ERR == std::signal(SIGTERM, sig_handler)) {
        // Handle error
    }
}
```

Noncompliant Code Example

In this noncompliant code example, a signal handler calls a function that allows exceptions, and it attempts to handle any exceptions thrown. Because exceptions are not part of the common subset of C and C++ features, this example results in implementation-defined behavior. However, it is unlikely that the implementation's behavior will be suitable. For instance, on a stack-based architecture where a signal is generated asynchronously (instead of as a result of a call to std::abort() or std::raise()), it is possible that the stack frame is not properly initialized, causing stack tracing to be unreliable and preventing the exception from being caught properly.
#include <csignal>

static void g() noexcept(false);

extern "C" void sig_handler(int sig) {
    try {
        g();
    } catch (...) {
        // Handle error
    }
}

void install_signal_handler() {
    if (SIG_ERR == std::signal(SIGTERM, sig_handler)) {
        // Handle error
    }
}

// Called periodically to poll the signal flag.
void poll_signal_flag() {
    if (signal_flag == 1) {
        signal_flag = 0;
        try {
            g();
        } catch(...) {
            // Handle error
        }
    }
}

Risk Assessment

Failing to use a plain old function as a signal handler can result in implementation-defined behavior as well as undefined behavior. Given the number of features that exist in C++ that do not also exist in C, the consequences that arise from failure to comply with this rule can range from benign (harmless) behavior to abnormal program termination, or even arbitrary code execution.

<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>MSC54-CPP</td>
<td>High</td>
<td>Probable</td>
<td>High</td>
<td>P6</td>
<td>L2</td>
</tr>
</tbody>
</table>

Automated Detection
### Tool, Version, Checker, Description

<table>
<thead>
<tr>
<th>Tool</th>
<th>Version</th>
<th>Checker</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>Parasoft C/C++test</td>
<td>10.4.2</td>
<td>CERT_CPP-MSC54-a</td>
<td>Properly define signal handlers</td>
</tr>
<tr>
<td>PRQA QA-C++</td>
<td>4.4</td>
<td>2888</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**
  - SIG30-C. Call only asynchronous-safe functions within signal handlers
  - SIG31-C. Do not access shared objects in signal handlers

### Bibliography

[ISO/IEC 14882-2014] Subclause 18.10, "Other Runtime Support"