SIG34-C. Do not call signal() from within interruptible signal handlers

A signal handler should not reassert its desire to handle its own signal. This is often done on nonpersistent platforms—that is, platforms that, upon receiving a signal, reset the handler for the signal to SIG_DFL before calling the bound signal handler. Calling signal() under these conditions presents a race condition. (See SIG01-C. Understand implementation-specific details regarding signal handler persistence.)

A signal handler may call signal() only if it does not need to be asynchronous-safe (that is, if all relevant signals are masked so that the handler cannot be interrupted).

Noncompliant Code Example (POSIX)

On nonpersistent platforms, this noncompliant code example contains a race window, starting when the host environment resets the signal and ending when the handler calls signal(). During that time, a second signal sent to the program will trigger the default signal behavior, consequently defeating the persistent behavior implied by the call to signal() from within the handler to reassert the binding.

If the environment is persistent (that is, it does not reset the handler when the signal is received), the signal() call from within the handler() function is redundant.

```c
#include <signal.h>

void handler(int signum) {
    if (signal(signum, handler) == SIG_ERR) {
        /* Handle error */
    } /* Handle signal */
}

void func(void) {
    if (signal(SIGUSR1, handler) == SIG_ERR) {
        /* Handle error */
    }
}
```

Compliant Solution (POSIX)

Calling the signal() function from within the signal handler to reassert the binding is unnecessary for persistent platforms, as in this compliant solution:

```c
#include <signal.h>

void handler(int signum) {
    /* Handle signal */
}

void func(void) {
    if (signal(SIGUSR1, handler) == SIG_ERR) {
        /* Handle error */
    }
}
```

Compliant Solution (POSIX)

POSIX defines the sigaction() function, which assigns handlers to signals in a similar manner to signal() but allows the caller to explicitly set persistence. Consequently, the sigaction() function can be used to eliminate the race window on nonpersistent platforms, as in this compliant solution:
#include <signal.h>
#include <stddef.h>

void handler(int signum) {
    /* Handle signal */
}

void func(void) {
    struct sigaction act;
    act.sa_handler = handler;
    act.sa_flags = 0;
    if (sigemptyset(&act.sa_mask) != 0) {
        /* Handle error */
    }
    if (sigaction(SIGUSR1, &act, NULL) != 0) {
        /* Handle error */
    }
}

Although the handler in this example does not call signal(), it could do so safely because the signal is masked and the handler cannot be interrupted. If the same handler is installed for more than one signal, the signals must be masked explicitly in act.sa_mask to ensure that the handler cannot be interrupted because the system masks only the signal being delivered.

POSIX recommends that new applications should use sigaction() rather than signal(). The sigaction() function is not defined by the C Standard and is not supported on some platforms, including Windows.

Compliant Solution (Windows)

There is no safe way to implement persistent signal-handler behavior on Windows platforms, and it should not be attempted. If a design depends on this behavior, and the design cannot be altered, it may be necessary to claim a deviation from this rule after completing an appropriate risk analysis.

The reason for this is that Windows is a nonpersistent platform as discussed above. Just before calling the current handler function, Windows resets the handler for the next occurrence of the same signal to SIG_DFL. If the handler calls signal() to reinstall itself, there is still a race window. A signal might occur between the start of the handler and the call to signal(), which would invoke the default behavior instead of the desired handler.

Exceptions

SIG34-C-EX1: For implementations with persistent signal handlers, it is safe for a handler to modify the behavior of its own signal. Behavior modifications include ignoring the signal, resetting to the default behavior, and having the signal handled by a different handler. A handler reasserting its binding is also safe but unnecessary.

The following code example resets a signal handler to the system's default behavior:

```c
#include <signal.h>

void handler(int signum) {
    #if !defined(_WIN32)
        if (signal(signum, SIG_DFL) == SIG_ERR) {
            /* Handle error */
        }
    #endif
    /* Handle signal */
}

void func(void) {
    if (signal(SIGUSR1, handler) == SIG_ERR) {
        /* Handle error */
    }
}
```

Risk Assessment

Two signals in quick succession can trigger a race condition on nonpersistent platforms, causing the signal's default behavior despite a handler's attempt to override it.
### 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>19.04</td>
<td>signal-handler-signal-call</td>
<td>Partially checked</td>
</tr>
<tr>
<td>Axivion Bauhaus Suite</td>
<td>6.9.0</td>
<td>CertC-SIG34</td>
<td></td>
</tr>
<tr>
<td>CodeSonar</td>
<td>5.2p0</td>
<td>BADFUNC.SIGNAL</td>
<td>Use of signal</td>
</tr>
<tr>
<td>Compass/ROSE</td>
<td></td>
<td></td>
<td>Can detect violations of this rule. However, false positives may occur on systems with persistent handlers</td>
</tr>
<tr>
<td>LDRA tool suite</td>
<td>9.7.1</td>
<td>97 D</td>
<td>Fully implemented</td>
</tr>
<tr>
<td>Parasoft C/C++test</td>
<td>10.4.2</td>
<td>CERT_C-SIG34-a</td>
<td>The signal handling facilities of &lt;signal.h&gt; shall not be used</td>
</tr>
<tr>
<td>Polyspace Bug Finder</td>
<td>R2019b</td>
<td>CERT C: Rule SIG34-C</td>
<td>Checks for signal call from within signal handler (rule fully covered)</td>
</tr>
<tr>
<td>PRQA QA-C</td>
<td>9.7</td>
<td>5021</td>
<td>Partially implemented</td>
</tr>
<tr>
<td>PRQA QA-C++</td>
<td>4.4</td>
<td>5022</td>
<td></td>
</tr>
<tr>
<td>RuleChecker</td>
<td>19.04</td>
<td>signal-handler-signal-call</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

Key here (explains table format and definitions)

<table>
<thead>
<tr>
<th>Taxonomy</th>
<th>Taxonomy item</th>
<th>Relationship</th>
</tr>
</thead>
<tbody>
<tr>
<td>CERT C Secure Coding</td>
<td>SIG01-C. Understand implementation-specific details regarding signal handler persistence</td>
<td>Prior to 2018-01-12: CERT: Unspecified Relationship</td>
</tr>
<tr>
<td>Standard</td>
<td></td>
<td></td>
</tr>
</tbody>
</table>