SIG01-C. Understand implementation-specific details regarding signal handler persistence

The signal() function has implementation-defined behavior and behaves differently on Windows, for example, than it does on many UNIX systems.

The following code example shows this behavior:

```c
#include <stdio.h>
#include <signal.h>

volatile sig_atomic_t e_flag = 0;

void handler(int signum) {
    e_flag = 1;
}

int main(void) {
    if (signal(SIGINT, handler) == SIG_ERR) {
        /* Handle error */
    }
    while (!e_flag) {}
    puts("Escaped from first while ()");
    e_flag = 0;
    while (!e_flag) {}
    puts("Escaped from second while ()");
    return 0;
}
```

Many UNIX (and UNIX-like) systems automatically reinstall signal handlers upon handler execution, meaning that the signal handler defined by the user is left in place until it is explicitly removed. For example, when this code is compiled with GCC 3.4.4 and executed under Red Hat Linux, SIGINT is captured both times by handler:

```
% ./test
^C
Escaped from first while ()
^C
Escaped from second while ()
%
```

When a signal handler is installed with the signal() function in Windows and some UNIX systems, the default action is restored for that signal after the signal is triggered. This means that signal handlers are not automatically reinstalled. For example, when this code is compiled with Microsoft Visual Studio 2005, version 8.0, only the first SIGINT is captured by handler:

```
> test.exe
^C
Escaped from first while ()
^C
> 
```

The second SIGINT executes the default action, which is to terminate program execution.

Different actions must be taken depending on whether or not the application requires signal handlers to be persistent.

Persistent Handlers

Asynchronous signals may originate from malicious actors external to the process. Consequently, vulnerabilities may exist if the signal-handler-persistence behavior is inconsistent with the developer's expectations, such as when the developer expects the signal handler to persist but it does not.

Noncompliant Code Example

This noncompliant code example fails to persist the signal handler on Windows platforms and on those UNIX systems where handlers are not persistent by default:
void handler(int signum) {
    /* Handle signal */
}

Noncompliant Code Example

A common approach to create persistent signal handlers is to call \texttt{signal()} from within the handler itself, consequently \textit{unresetting} the reset signal:

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

Unfortunately, this solution still contains a race window, starting when the host environment resets the signal and ending when the handler calls \texttt{signal()}. During that time, a second signal sent to the program will trigger the default signal behavior, defeating the persistent behavior. (See \texttt{SIG34-C}. Do not call \texttt{signal()} from within interruptible signal handlers.)

A secure solution must prevent the environment from resetting the signal in the first place, guaranteeing persistence. Unfortunately, Windows does not provide a secure solution to this problem.

Compliant Solution (POSIX)

The POSIX \texttt{sigaction()} function assigns handlers to signals in a manner similar to the C \texttt{signal()} function but also allows signal-handler persistence to be controlled via the \texttt{SA_RESETHAND} flag. (Leaving the flag clear makes the handler persistent.)

/*
 * Equivalent to signal(SIGUSR1, handler) but makes
 * signal persistent.
 */

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 */
}

POSIX recommends \texttt{sigaction()} and deprecates \texttt{signal()}. Unfortunately, \texttt{sigaction()} is not defined in the C Standard and is consequently not as portable a solution.

Nonpersistent Handlers

Errors may also occur when the developer expects the default action to be restored for a signal but the signal handler persists instead.

Noncompliant Code Example (UNIX)

This noncompliant code example fails to reset the signal handler to its default behavior on systems where handlers are persistent by default:

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

Compliant Solution (UNIX and Windows)

A C-compliant solution to reset the handler on a UNIX system is to rebind the signal to the default handler in the first line of the handler itself. Windows, however, automatically resets handlers to their default behavior.
With the compliant solution for UNIX, no race condition occurs that can be exploited by an attacker sending a second signal. This is because a second signal sent to the handler, before the latter calls `signal(signum, SIG_DFL)`, will only cause the handler to restart and call `signal()` anyway.

This solution is an exception to SIG34-C. Do not call `signal()` from within interruptible signal handlers.

Compliant Solution (POSIX)

The POSIX `sigaction()` function assigns handlers to signals in a manner similar to the C `signal()` function but also allows signal-handler persistence to be controlled via the `SA_RESETHAND` flag. (Setting the flag makes the handler nonpersistent.)

```c
/*
 * Equivalent to signal(SIGUSR1, handler) but makes
 * signal nonpersistent.
 */
struct sigaction act;
act.sa_handler = handler;
act.sa_flags = SA_RESETHAND;
if (sigemptyset(&act.sa_mask) != 0) {
    /* Handle error */
}
if (sigaction(SIGUSR1, &act, NULL) != 0) {
    /* Handle error */
}
```

Risk Assessment

Failure to understand implementation-specific details regarding signal-handler persistence can lead to unexpected behavior.

<table>
<thead>
<tr>
<th>Recommendation</th>
<th>Severity</th>
<th>Likelihood</th>
<th>Remediation Cost</th>
<th>Priority</th>
<th>Level</th>
</tr>
</thead>
<tbody>
<tr>
<td>SIG01-C</td>
<td>Low</td>
<td>Unlikely</td>
<td>Low</td>
<td>P3</td>
<td>L3</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>CodeSonar</td>
<td>5.2p0</td>
<td>BADFUNC</td>
<td>Use of signal</td>
</tr>
<tr>
<td>Compass/ROSE</td>
<td></td>
<td></td>
<td>Could detect possible violations by flagging any signal handler that calls signal() to (re)assert itself as the handler for its signal</td>
</tr>
<tr>
<td>LDRA tool suite</td>
<td>9.7.1</td>
<td>97 D</td>
<td>Partially implemented</td>
</tr>
<tr>
<td>Parasoft C/C++test</td>
<td>10.4.2</td>
<td>CERT_C-SIG01-a</td>
<td>The signal handling facilities of &lt;signal.h&gt; shall not be used</td>
</tr>
<tr>
<td>PRQA QA-C</td>
<td>9.7</td>
<td>5020</td>
<td>Partially implemented</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 | VOID SIG01-CPP. Understand implementation-specific details regarding signal handler persistence |