SIG00-C. Mask signals handled by noninterruptible signal handlers

A signal is a mechanism for transferring control that is typically used to notify a process that an event has occurred. That process can then respond to the event accordingly. The C Standard provides functions for sending and handling signals within a C program.

Processes handle signals by registering a signal handler using the `signal()` function, which is specified as

```
void (*signal(int sig, void (*func)(int)))(int);
```

This signal handler is conceptually equivalent to

```
typedef void (*sighandler_t)(int signum);
extern sighandler_t signal(
  int signum,
  sighandler_t handler
);
```

Signal handlers can be interrupted by signals, including their own. If a signal is not reset before its handler is called, the handler can interrupt its own execution. A handler that always successfully executes its code despite interrupting itself or being interrupted is async-signal-safe.

Some platforms provide the ability to mask signals while a signal handler is being processed. If a signal is masked while its own handler is processed, the handler is noninterruptible and need not be async-signal-safe. However, even when a signal is masked while its own handler is processed, the handler must still avoid invoking async-signal-safe unsafe functions because their execution may be (or have been) interrupted by another signal.

Vulnerabilities can arise if a signal handler that is not async-signal-safe is interrupted with any unmasked signal, including its own.

Noncompliant Code Example

This noncompliant code example registers a single signal handler to process both `SIGUSR1` and `SIGUSR2`. The variable `sig2` should be set to `1` if one or more `SIGUSR1` signals are followed by `SIGUSR2`, essentially implementing a finite state machine within the signal handler.

```
#include <signal.h>

volatile sig_atomic_t sig1 = 0;
volatile sig_atomic_t sig2 = 0;

void handler(int signum) {
  if (signum == SIGUSR1) {
    sig1 = 1;
  }
  else if (sig1) {
    sig2 = 1;
  }
}

int main(void) {
  if (signal(SIGUSR1, handler) == SIG_ERR) {
    /* Handle error */
  }
  if (signal(SIGUSR2, handler) == SIG_ERR) {
    /* Handler error */
  }

  while (sig2 == 0) {
    /* Do nothing or give up CPU for a while */
  }

  /* ... */

  return 0;
}
```

Unfortunately, a race condition occurs in the implementation of `handler()`. If `handler()` is called to handle `SIGUSR1` and is interrupted to handle `SIGUSR2`, it is possible that `sig2` will not be set.

Compliant Solution (POSIX)

The POSIX `sigaction()` function assigns handlers to signals in a similar manner to the C `signal()` function, but it also allows signal masks to be set explicitly. Consequently, `sigaction()` can be used to prevent a signal handler from interrupting itself.

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

volatile sig_atomic_t sig1 = 0;
volatile sig_atomic_t sig2 = 0;

void handler(int signum) {
  if (signum == SIGUSR1) {
    sig1 = 1;
  }
  else if (sig1) {
    sig2 = 1;
  }
}

int main(void) {
  struct sigaction act;
  act.sa_handler = &handler;
  act.sa_flags = 0;
  if (sigemptyset(&act.sa_mask) != 0) {
    /* Handle error */
  }
  if (sigaddset(&act.sa_mask, SIGUSR1)) {
    /* Handle error */
  }
  if (sigaddset(&act.sa_mask, SIGUSR2)) {
    /* Handle error */
  }

  if (sigaction(SIGUSR1, &act, NULL) != 0) {
    /* Handle error */
  }
  if (sigaction(SIGUSR2, &act, NULL) != 0) {
    /* Handle error */
  }

  while (sig2 == 0) {
    /* Do nothing or give up CPU for a while */
  }

  /* ... */

  return 0;
}
```

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

Risk Assessment

Interrupting a noninterruptible signal handler can result in a variety of vulnerabilities [Zalewski 2001].

Recommendation	Severity	Likelihood	Remediation Cost	Priority	Level
SIG00-C	High	Likely	High	**P9**	**L2**

Automated Detection

Tool	Version	Checker	Description
CodeSonar	5.2p0	**BADFUNC.SIGNAL**	Use of signal

LDRA tool suite	9.7.1	**44 S**	Enhanced enforcement
Parasoft C/C++test	10.4.2	**CERT_C-SIG00-a**	The signal handling facilities of <signal.h> shall not be used
PRQA QA-C	9.7	**5019**	Partially implemented

Related Vulnerabilities

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

Related Guidelines

SEI CERT C++ Coding Standard	VOID SIG00-CPP. Mask signals handled by noninterruptible signal handlers
MITRE CWE	CWE-662, Insufficient synchronization

Bibliography

[C99 Rationale 2003]	Subclause 5.2.3, "Signals and Interrupts"
[Dowd 2006]	Chapter 13, "Synchronization and State" ("Signal Interruption and Repetition")
[IEEE Std 1003.1:2013]	XSH, System Interface, `longjmp`
[OpenBSD]	`signal()` Man Page
[Zalewski 2001]	"Delivering Signals for Fun and Profit"