SIG30-C. Call only asynchronous-safe functions within signal handlers

Call only asynchronous-safe functions within signal handlers. For strictly conforming programs, only the C standard library functions `abort()`, `_Exit()`, `quick_exit()`, and `signal()` can be safely called from within a signal handler.

The C Standard, 7.14.1.1, paragraph 5 [ISO/IEC 9899:2011], states that if the signal occurs other than as the result of calling the `abort()` or `raise()` function, the behavior is undefined if

> *...the signal handler calls any function in the standard library other than the `abort` function, the `_Exit` function, the `quick_exit` function, or the `signal` function with the first argument equal to the signal number corresponding to the signal that caused the invocation of the handler.*

Implementations may define a list of additional asynchronous-safe functions. These functions can also be called within a signal handler. This restriction applies to library functions as well as application-defined functions.

According to the C Rationale, 7.14.1.1 [C99 Rationale 2003],

> *When a signal occurs, the normal flow of control of a program is interrupted. If a signal occurs that is being trapped by a signal handler, that handler is invoked. When it is finished, execution continues at the point at which the signal occurred. This arrangement can cause problems if the signal handler invokes a library function that was being executed at the time of the signal.*

In general, it is not safe to invoke I/O functions from within signal handlers. Programmers should ensure a function is included in the list of an implementation's asynchronous-safe functions for all implementations the code will run on before using them in signal handlers.

Noncompliant Code Example

In this noncompliant example, the C standard library functions `fputs()` and `free()` are called from the signal handler via the function `log_message()`. Neither function is asynchronous-safe.

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

enum { MAXLINE = 1024 };
char *info = NULL;

void log_message(void) {
  fputs(info, stderr);
}

void handler(int signum) {
  log_message();
  free(info);
  info = NULL;
}

int main(void) {
  if (signal(SIGINT, handler) == SIG_ERR) {
    /* Handle error */
  }
  info = (char *)malloc(MAXLINE);
  if (info == NULL) {
    /* Handle Error */
  }

  while (1) {
    /* Main loop program code */

    log_message();

    /* More program code */
  }
  return 0;
}
```

Compliant Solution

Signal handlers should be as concise as possible—ideally by unconditionally setting a flag and returning. This compliant solution sets a flag of type `volatile sig_atomic_t` and returns; the `log_message()` and `free()` functions are called directly from `main()`:

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

enum { MAXLINE = 1024 };
volatile sig_atomic_t eflag = 0;
char *info = NULL;

void log_message(void) {
  fputs(info, stderr);
}

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

int main(void) {
  if (signal(SIGINT, handler) == SIG_ERR) {
    /* Handle error */
  }
  info = (char *)malloc(MAXLINE);
  if (info == NULL) {
    /* Handle error */
  }

  while (!eflag) {
    /* Main loop program code */

    log_message();

    /* More program code */
  }

  log_message();
  free(info);
  info = NULL;

  return 0;
}
```

Noncompliant Code Example (`longjmp()`)

Invoking the `longjmp()` function from within a signal handler can lead to undefined behavior if it results in the invocation of any non-asynchronous-safe fu nctions. Consequently, neither `longjmp()` nor the POSIX `siglongjmp()` functions should ever be called from within a signal handler.

This noncompliant code example is similar to a vulnerability in an old version of Sendmail [VU #834865]. The intent is to execute code in a `main()` loop, which also logs some data. Upon receiving a `SIGINT`, the program transfers out of the loop, logs the error, and terminates.

However, an attacker can exploit this noncompliant code example by generating a `SIGINT` just before the second `if` statement in `log_message()`. The result is that `longjmp()` transfers control back to `main()`, where `log_message()` is called again. However, the first `if` statement would not be executed this time (because `buf` is not set to `NULL` as a result of the interrupt), and the program would write to the invalid memory location referenced by b `uf0`.

```
#include <setjmp.h>
#include <signal.h>
#include <stdlib.h>

enum { MAXLINE = 1024 };
static jmp_buf env;

void handler(int signum) {
  longjmp(env, 1);
}

void log_message(char *info1, char *info2) {
  static char *buf = NULL;
  static size_t bufsize;
  char buf0[MAXLINE];

  if (buf == NULL) {
    buf = buf0;
    bufsize = sizeof(buf0);
  }

  /*
   * Try to fit a message into buf, else reallocate
   * it on the heap and then log the message.
   */

  /* Program is vulnerable if SIGINT is raised here */

  if (buf == buf0) {
    buf = NULL;
  }
}

int main(void) {
  if (signal(SIGINT, handler) == SIG_ERR) {
    /* Handle error */
  }
  char *info1;
  char *info2;

  /* info1 and info2 are set by user input here */

  if (setjmp(env) == 0) {
    while (1) {
      /* Main loop program code */
      log_message(info1, info2);
      /* More program code */
    }
  } else {
    log_message(info1, info2);
  }

  return 0;
}
```

Compliant Solution

In this compliant solution, the call to `longjmp()` is removed; the signal handler sets an error flag instead:

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

enum { MAXLINE = 1024 };
volatile sig_atomic_t eflag = 0;

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

void log_message(char *info1, char *info2) {
  static char *buf = NULL;
  static size_t bufsize;
  char buf0[MAXLINE];

  if (buf == NULL) {
    buf = buf0;
    bufsize = sizeof(buf0);
  }

  /*
   * Try to fit a message into buf, else reallocate
   * it on the heap and then log the message.
   */
  if (buf == buf0) {
    buf = NULL;
  }
}

int main(void) {
  if (signal(SIGINT, handler) == SIG_ERR) {
    /* Handle error */
  }
  char *info1;
  char *info2;

  /* info1 and info2 are set by user input here */

  while (!eflag) {
    /* Main loop program code */
    log_message(info1, info2);
    /* More program code */
  }

  log_message(info1, info2);

  return 0;
}
```

Noncompliant Code Example (`raise()`)

In this noncompliant code example, the `int_handler()` function is used to carry out tasks specific to `SIGINT` and then raises `SIGTERM`. However, there is a nested call to the `raise()` function, which is undefined behavior.

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

void term_handler(int signum) {
  /* SIGTERM handler */
}

void int_handler(int signum) {
  /* SIGINT handler */
  if (raise(SIGTERM) != 0) {
    /* Handle error */
  }
}

int main(void) {
  if (signal(SIGTERM, term_handler) == SIG_ERR) {
    /* Handle error */
  }
  if (signal(SIGINT, int_handler) == SIG_ERR) {
    /* Handle error */
  }

  /* Program code */
  if (raise(SIGINT) != 0) {
    /* Handle error */
  }
  /* More code */

  return EXIT_SUCCESS;
}
```

Compliant Solution

In this compliant solution, `int_handler()` invokes `term_handler()` instead of raising `SIGTERM`:

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

void term_handler(int signum) {
  /* SIGTERM handler */
}

void int_handler(int signum) {
  /* SIGINT handler */
  /* Pass control to the SIGTERM handler */
  term_handler(SIGTERM);
}

int main(void) {
  if (signal(SIGTERM, term_handler) == SIG_ERR) {
    /* Handle error */
  }
  if (signal(SIGINT, int_handler) == SIG_ERR) {
    /* Handle error */
  }

  /* Program code */
  if (raise(SIGINT) != 0) {
    /* Handle error */
  }
  /* More code */

  return EXIT_SUCCESS;
}
```

Implementation Details

POSIX

The following table from the POSIX standard [IEEE Std 1003.1:2013] defines a set of functions that are asynchronous-signal-safe. Applications may invoke these functions, without restriction, from a signal handler.

_Exit()	fexecve()	posix_trace_event()	sigprocmask()
_exit()	fork()	pselect()	sigqueue()
abort()	fstat()	pthread_kill()	sigset()
accept()	fstatat()	pthread_self()	sigsuspend()
access()	fsync()	pthread_sigmask()	sleep()
aio_error()	ftruncate()	raise()	sockatmark()
aio_return()	futimens()	read()	socket()
aio_suspend()	getegid()	readlink()	socketpair()
alarm()	geteuid()	readlinkat()	stat()
bind()	getgid()	recv()	symlink()
cfgetispeed()	getgroups()	recvfrom()	symlinkat()
cfgetospeed()	getpeername()	recvmsg()	tcdrain()
cfsetispeed()	getpgrp()	rename()	tcflow()
cfsetospeed()	getpid()	renameat()	tcflush()
chdir()	getppid()	rmdir()	tcgetattr()
chmod()	getsockname()	select()	tcgetpgrp()
chown()	getsockopt()	sem_post()	tcsendbreak()
clock_gettime()	getuid()	send()	tcsetattr()
close()	kill()	sendmsg()	tcsetpgrp()
connect()	link()	sendto()	time()
creat()	linkat()	setgid()	timer_getoverrun()
dup()	listen()	setpgid()	timer_gettime()
dup2()	lseek()	setsid()	timer_settime()
execl()	lstat()	setsockopt()	times()
execle()	mkdir()	setuid()	umask()
execv()	mkdirat()	shutdown()	uname()
execve()	mkfifo()	sigaction()	unlink()
faccessat()	mkfifoat()	sigaddset()	unlinkat()
fchdir()	mknod()	sigdelset()	utime()
fchmod()	mknodat()	sigemptyset()	utimensat()
fchmodat()	open()	sigfillset()	utimes()
fchown()	openat()	sigismember()	wait()
fchownat()	pause()	signal()	waitpid()
fcntl()	pipe()	sigpause()	write()
fdatasync()	poll()	sigpending()	

All functions not listed in this table are considered to be unsafe with respect to signals. In the presence of signals, all POSIX functions behave as defined when called from or interrupted by a signal handler, with a single exception: when a signal interrupts an unsafe function and the signal handler calls an unsafe function, the behavior is undefined.

The C Standard, 7.14.1.1, paragraph 4 [ISO/IEC 9899:2011], states

> *If the signal occurs as the result of calling the abort or raise function, the signal handler shall not call the raise function.*

However, in the description of `signal()`, POSIX [IEEE Std 1003.1:2013] states

> *This restriction does not apply to POSIX applications, as POSIX.1-2008 requires `raise()` to be async-signal-safe.*

See also undefined behavior 131.

OpenBSD

The OpenBSD `signal()` manual page lists a few additional functions that are asynchronous-safe in OpenBSD but "probably not on other systems" [OpenBSD], including `snprintf()`, `vsnprintf()`, and `syslog_r()` but only when the `syslog_data struct` is initialized as a local variable.

Risk Assessment

Invoking functions that are not asynchronous-safe from within a signal handler is undefined behavior.

Rule	Severity	Likelihood	Remediation Cost	Priority	Level
SIG30-C	High	Likely	Medium	P18	L1

Automated Detection

Tool	Version	Checker	Description
Astrée	19.04	**signal-handler-unsafe-call**	Partially checked
Axivion Bauhaus Suite	6.9.0	**CertC-SIG30**	
Compass/ROSE			Can detect violations of the rule for single-file programs
LDRA tool suite	9.7.1	**88 D, 89 D**	Partially implemented
Parasoft C/C++test	10.4.2	**CERT_C-SIG30-a**	Properly define signal handlers
Polyspace Bug Finder	R2019b	CERT C: Rule SIG30-C	Checks for function called from signal handler not asynchronous-safe (rule fully covered)
PRQA QA-C	9.7	**2028, 2030**	
RuleChecker	19.04	**signal-handler-unsafe-call**	Partially checked
Splint	3.1.1		

Related Vulnerabilities

For an overview of software vulnerabilities resulting from improper signal handling, see Michal Zalewski's paper "Delivering Signals for Fun and Profit" [Zalewski 2001].

CERT Vulnerability Note VU #834865, "Sendmail signal I/O race condition," describes a vulnerability resulting from a violation of this rule. Another notable case where using the `longjmp()` function in a signal handler caused a serious vulnerability is wu-ftpd 2.4 [Greenman 1997]. The effective user ID is set to 0 in one signal handler. If a second signal interrupts the first, a call is made to `longjmp()`, returning the program to the main thread but without lowering the user's privileges. These escalated privileges can be used for further exploitation.

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

Related Guidelines

Key here (explains table format and definitions)

Taxonomy	Taxonomy item	Relationship
ISO/IEC TS 17961:2013	Calling functions in the C Standard Library other than `abort`, `_Exit`, and `signal` from within a signal handler [asyncsig]	Prior to 2018-01-12: CERT: Unspecified Relationship
CWE 2.11	CWE-479, Signal Handler Use of a Non-reentrant Function	2017-07-10: CERT: Exact

Bibliography

[C99 Rationale 2003]	Subclause 5.2.3, "Signals and Interrupts" Subclause 7.14.1.1, "The `signal` Function"
[Dowd 2006]	Chapter 13, "Synchronization and State"
[Greenman 1997]	
[IEEE Std 1003.1:2013]	XSH, System Interfaces, `longjmp` XSH, System Interfaces, `raise`
[ISO/IEC 9899:2011]	7.14.1.1, "The `signal` Function"
[OpenBSD]	`signal()` Man Page
[VU #834865]	
[Zalewski 2001]	"Delivering Signals for Fun and Profit"

adjust column widths