SIG31-C. Do not access shared objects in signal handlers

Accessing or modifying shared objects in signal handlers can result in race conditions that can leave data in an inconsistent state. The two exceptions (C Standard, 5.1.2.3, paragraph 5) to this rule are the ability to read from and write to lock-free atomic objects and variables of type `volatile sig_atomic_t`. Accessing any other type of object from a signal handler is undefined behavior. (See undefined behavior 131.)

The need for the `volatile` keyword is described in DCL22-C. Use volatile for data that cannot be cached.

The type `sig_atomic_t` is the integer type of an object that can be accessed as an atomic entity even in the presence of asynchronous interrupts. The type of `sig_atomic_t` is implementation-defined, though it provides some guarantees. Integer values ranging from `SIG_ATOMIC_MIN` through `SIG_ATOMIC_MAX`, inclusive, may be safely stored to a variable of the type. In addition, when `sig_atomic_t` is a signed integer type, `SIG_ATOMIC_MIN` must be no greater than `127` and `SIG_ATOMIC_MAX` no less than `127`. Otherwise, `SIG_ATOMIC_MIN` must be `0` and `SIG_ATOMIC_MAX` must be no less than `255`. The macros `SIG_ATOMIC_MIN` and `SIG_ATOMIC_MAX` are defined in the header `<stdint.h>`.

According to the C99 Rationale [C99 Rationale 2003], other than calling a limited, prescribed set of library functions,

> *the C89 Committee concluded that about the only thing a strictly conforming program can do in a signal handler is to assign a value to a `volatile static` variable which can be written uninterruptedly and promptly return.*

However, this issue was discussed at the April 2008 meeting of ISO/IEC WG14, and it was agreed that there are no known implementations in which it would be an error to read a value from a `volatile sig_atomic_t` variable, and the original intent of the committee was that both reading and writing variables of `volatile sig_atomic_t` would be strictly conforming.

The signal handler may also call a handful of functions, including `abort().` (See SIG30-C. Call only asynchronous-safe functions within signal handlers for more information.)

Noncompliant Code Example

In this noncompliant code example, `err_msg` is updated to indicate that the `SIGINT` signal was delivered. The `err_msg` variable is a character pointer and not a variable of type `volatile sig_atomic_t`.

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

enum { MAX_MSG_SIZE = 24 };
char *err_msg;

void handler(int signum) {
  strcpy(err_msg, "SIGINT encountered.");
}

int main(void) {
  signal(SIGINT, handler);

  err_msg = (char *)malloc(MAX_MSG_SIZE);
  if (err_msg == NULL) {
    /* Handle error */
  }
  strcpy(err_msg, "No errors yet.");
  /* Main code loop */
  return 0;
}
```

Compliant Solution (Writing `volatile sig_atomic_t`)

For maximum portability, signal handlers should only unconditionally set a variable of type `volatile sig_atomic_t` and return, as in this compliant solution:

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

enum { MAX_MSG_SIZE = 24 };
volatile sig_atomic_t e_flag = 0;

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

int main(void) {
  char *err_msg = (char *)malloc(MAX_MSG_SIZE);
  if (err_msg == NULL) {
    /* Handle error */
  }

  signal(SIGINT, handler);
  strcpy(err_msg, "No errors yet.");
  /* Main code loop */
  if (e_flag) {
    strcpy(err_msg, "SIGINT received.");
  }
  return 0;
}
```

Compliant Solution (Lock-Free Atomic Access)

Signal handlers can refer to objects with static or thread storage durations that are lock-free atomic objects, as in this compliant solution:

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

#ifdef __STDC_NO_ATOMICS__
#error "Atomics are not supported"
#elif ATOMIC_INT_LOCK_FREE == 0
#error "int is never lock-free"
#endif

atomic_int e_flag = ATOMIC_VAR_INIT(0);

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

int main(void) {
  enum { MAX_MSG_SIZE = 24 };
  char err_msg[MAX_MSG_SIZE];
#if ATOMIC_INT_LOCK_FREE == 1
  if (!atomic_is_lock_free(&e_flag)) {
    return EXIT_FAILURE;
  }
#endif
  if (signal(SIGINT, handler) == SIG_ERR) {
    return EXIT_FAILURE;
  }
  strcpy(err_msg, "No errors yet.");
  /* Main code loop */
  if (e_flag) {
    strcpy(err_msg, "SIGINT received.");
  }
  return EXIT_SUCCESS;
}
```

Exceptions

SIG31-C-EX1: The C Standard, 7.14.1.1 paragraph 5 [ISO/IEC 9899:2011], makes a special exception for `errno` when a valid call to the `signal()` function results in a `SIG_ERR` return, allowing `errno` to take an indeterminate value. (See ERR32-C. Do not rely on indeterminate values of errno.)

Risk Assessment

Accessing or modifying shared objects in signal handlers can result in accessing data in an inconsistent state. Michal Zalewski's paper "Delivering Signals for Fun and Profit" [Zalewski 2001] provides some examples of vulnerabilities that can result from violating this and other signal-handling rules.

Rule	Severity	Likelihood	Remediation Cost	Priority	Level
SIG31-C	High	Likely	High	**P9**	**L2**

Automated Detection

Tool	Version	Checker	Description
Astrée	19.04	**signal-handler-shared-access**	Partially checked
Axivion Bauhaus Suite	6.9.0	**CertC-SIG31**	
CodeSonar	5.2p0	**CONCURRENCY.DATARACE**	Data race
Compass/ROSE			Can detect violations of this rule for single-file programs
LDRA tool suite	9.7.1	**87 D**	Fully implemented
Parasoft C/C++test	10.4.2	**CERT_C-SIG31-a**	Properly define signal handlers
Polyspace Bug Finder	R2019b	CERT C: Rule SIG31-C	Checks for shared data access within signal handler (rule partially covered)
PRQA QA-C	9.7	**2029, 2030**	
RuleChecker	19.04	**signal-handler-shared-access**	Partially checked

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)

Taxonomy	Taxonomy item	Relationship
ISO/IEC TS 17961:2013	Accessing shared objects in signal handlers [accsig]	Prior to 2018-01-12: CERT: Unspecified Relationship
CWE 2.11	CWE-662, Improper Synchronization	2017-07-10: CERT: Rule subset of CWE
CWE 2.11	CWE-828, Signal Handler with Functionality that is not Asynchronous-Safe	2017-10-30:MITRE:Unspecified Relationship 2018-10-19:CERT:Rule subset of CWE

CERT-CWE Mapping Notes

Key here for mapping notes

CWE-662 and SIG31-C

CWE-662 = Union(SIG31-C, list) where list =

- Improper synchronization of shared objects between threads
- Improper synchronization of files between programs (enabling TOCTOU race conditions

CWE-828 and SIG31-C

CWE-828 = SIG31-C + non-async-safe things besides shared objects.

Bibliography

[C99 Rationale 2003]	5.2.3, "Signals and Interrupts"
[ISO/IEC 9899:2011]	Subclause 7.14.1.1, "The `signal` Function"
[Zalewski 2001]	