ERR32-C. Do not rely on indeterminate values of errno

According to the C Standard [ISO/IEC 9899:2011], the behavior of a program is undefined when

> *the value of `errno` is referred to after a signal occurred other than as the result of calling the `abort` or `raise` function and the corresponding signal handler obtained a `SIG_ERR` return from a call to the `signal` function.*

See undefined behavior 133.

A signal handler is allowed to call `signal();` if that fails, `signal()` returns `SIG_ERR` and sets `errno` to a positive value. However, if the event that caused a signal was external (not the result of the program calling `abort()` or `raise()`), the only functions the signal handler may call are `_Exit()` or `abort()`, or it may call `signal()` on the signal currently being handled; if `signal()` fails, the value of `errno` is indeterminate.

This rule is also a special case of SIG31-C. Do not access shared objects in signal handlers. The object designated by `errno` is of static storage duration and is not a `volatile sig_atomic_t`. As a result, performing any action that would require `errno` to be set would normally cause undefined behavior. The C Standard, 7.14.1.1, paragraph 5, makes a special exception for `errno` in this case, allowing `errno` to take on an indeterminate value but specifying that there is no other undefined behavior. This special exception makes it possible to call `signal()` from within a signal handler without risking undefined behavior, but the handler, and any code executed after the handler returns, must not depend on the value of `errno` being meaningful.

Noncompliant Code Example

The `handler()` function in this noncompliant code example attempts to restore default handling for the signal indicated by `signum`. If the request to set the signal to default can be honored, the `signal()` function returns the value of the signal handler for the most recent successful call to the `signal()` function for the specified signal. Otherwise, a value of `SIG_ERR` is returned and a positive value is stored in `errno`. Unfortunately, the value of `errno` is indeterminate because the `handler()` function is called when an external signal is raised, so any attempt to read `errno` (for example, by the `perror()` function) is undefined behavior:

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

typedef void (*pfv)(int);

void handler(int signum) {
  pfv old_handler = signal(signum, SIG_DFL);
  if (old_handler == SIG_ERR) {
    perror("SIGINT handler"); /* Undefined behavior */
    /* Handle error */
  }
}

int main(void) {
  pfv old_handler = signal(SIGINT, handler);
  if (old_handler == SIG_ERR) {
    perror("SIGINT handler");
    /* Handle error */
  }

  /* Main code loop */

  return EXIT_SUCCESS;
}
```

The call to `perror()` from `handler()` also violates SIG30-C. Call only asynchronous-safe functions within signal handlers.

Compliant Solution

This compliant solution does not reference `errno` and does not return from the signal handler if the `signal()` call fails:

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

typedef void (*pfv)(int);

void handler(int signum) {
  pfv old_handler = signal(signum, SIG_DFL);
  if (old_handler == SIG_ERR) {
    abort();
  }
}

int main(void) {
  pfv old_handler = signal(SIGINT, handler);
  if (old_handler == SIG_ERR) {
    perror("SIGINT handler");
    /* Handle error */
  }

  /* Main code loop */

  return EXIT_SUCCESS;
}
```

Noncompliant Code Example (POSIX)

POSIX is less restrictive than C about what applications can do in signal handlers. It has a long list of asynchronous-safe functions that can be called. (See SIG30-C. Call only asynchronous-safe functions within signal handlers.) Many of these functions set `errno` on error, which can lead to a signal handler being executed between a call to a failed function and the subsequent inspection of `errno`. Consequently, the value inspected is not the one set by that function but the one set by a function call in the signal handler. POSIX applications can avoid this problem by ensuring that signal handlers containing code that might alter `errno`; always save the value of `errno` on entry and restore it before returning.

The signal handler in this noncompliant code example alters the value of `errno`. As a result, it can cause incorrect error handling if executed between a failed function call and the subsequent inspection of `errno`:

```
#include <signal.h>
#include <stdlib.h>
#include <errno.h>
#include <sys/wait.h>

void reaper(int signum) {
  errno = 0;
  for (;;) {
    int rc = waitpid(-1, NULL, WNOHANG);
    if ((0 == rc) || (-1 == rc && EINTR != errno)) {
      break;
    }
  }
  if (ECHILD != errno) {
    /* Handle error */
  }
}

int main(void) {
  struct sigaction act;
  act.sa_handler = reaper;
  act.sa_flags = 0;
  if (sigemptyset(&act.sa_mask) != 0) {
    /* Handle error */
  }
  if (sigaction(SIGCHLD, &act, NULL) != 0) {
    /* Handle error */
  }

  /* ... */

  return EXIT_SUCCESS;
}
```

Compliant Solution (POSIX)

This compliant solution saves and restores the value of `errno` in the signal handler:

```
#include <signal.h>
#include <stdlib.h>
#include <errno.h>
#include <sys/wait.h>

void reaper(int signum) {
  errno_t save_errno = errno;
  errno = 0;
  for (;;) {
    int rc = waitpid(-1, NULL, WNOHANG);
    if ((0 == rc) || (-1 == rc && EINTR != errno)) {
      break;
    }
  }
  if (ECHILD != errno) {
    /* Handle error */
  }
  errno = save_errno;
}

int main(void) {
  struct sigaction act;
  act.sa_handler = reaper;
  act.sa_flags = 0;
  if (sigemptyset(&act.sa_mask) != 0) {
    /* Handle error */
  }
  if (sigaction(SIGCHLD, &act, NULL) != 0) {
    /* Handle error */
  }

  /* ... */

  return EXIT_SUCCESS;
}
```

Risk Assessment

Referencing indeterminate values of `errno` is undefined behavior.

Rule	Severity	Likelihood	Remediation Cost	Priority	Level
ERR32-C	Low	Unlikely	Low	**P3**	**L3**

Automated Detection

Tool	Version	Checker	Description
Axivion Bauhaus Suite	6.9.0	**CertC-ERR32**	
Compass/ROSE			Could detect violations of this rule by looking for signal handlers that themselves call `signal()`. A violation is reported if the call fails and the handler therefore checks `errno`. A violation also exists if the signal handler modifies `errno` without first copying its value elsewhere
Coverity	2017.07	**MISRA C 2012 Rule 22.8** **MISRA C 2012 Rule 22.9** **MISRA C 2012 Rule 22.10**	Implemented
LDRA tool suite	9.7.1	44 S	Enhanced enforcement

Parasoft C/C++test	10.4.2	**CERT_C-ERR32-a**	Properly use errno value
Polyspace Bug Finder	R2019b	CERT C: Rule ERR32-C	Checks for misuse of errno in a signal handler (rule fully covered)
PRQA QA-C	9.7	**2031**	

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
CERT C Secure Coding Standard	SIG30-C. Call only asynchronous-safe functions within signal handlers	Prior to 2018-01-12: CERT: Unspecified Relationship
CERT C Secure Coding Standard	SIG31-C. Do not access shared objects in signal handlers	Prior to 2018-01-12: CERT: Unspecified Relationship

Bibliography

[ISO/IEC 9899:2011]	Subclause 7.14.1.1, "The `signal` Function"

← ↑ →