CON41-C. Wrap functions that can fail spuriously in a loop

Functions that can fail spuriously should be wrapped in a loop. The `atomic_compare_exchange_weak()` and `atomic_compare_exchange_weak_explicit()` functions both attempt to set an atomic variable to a new value but only if it currently possesses a known old value. Unlike the related functions `atomic_compare_exchange_strong()` and `atomic_compare_exchange_strong_explicit()`, these functions are permitted to *fail spuriously*. This makes these functions faster on some platforms—for example, on architectures that implement compare-and-exchange using load-linked/store-conditional instructions, such as Alpha, ARM, MIPS, and PowerPC. The C Standard, 7.17.7.4, paragraph 4 [ISO/IEC 9899:2011], describes this behavior:

> *A weak compare-and-exchange operation may fail spuriously. That is, even when the contents of memory referred to by `expected` and `object` are equal, it may return zero and store back to `expected` the same memory contents that were originally there.*

Noncompliant Code Example

In this noncompliant code example, `reorganize_data_structure()` is to be used as an argument to `thrd_create()`. After reorganizing, the function attempts to replace the head pointer so that it points to the new version. If no other thread has changed the head pointer since it was originally loaded, `reorganize_data_structure()` is intended to exit the thread with a result of `true`, indicating success. Otherwise, the new reorganization attempt is discarded and the thread is exited with a result of `false`. However, `atomic_compare_exchange_weak()` may fail even when the head pointer has not changed. Therefore, `reorganize_data_structure()` may perform the work and then discard it unnecessarily.

```
#include <stdatomic.h>
#include <stdbool.h>

struct data {
  struct data *next;
  /* ... */
};

extern void cleanup_data_structure(struct data *head);

int reorganize_data_structure(void *thread_arg) {
  struct data *_Atomic *ptr_to_head = thread_arg;
  struct data *old_head = atomic_load(ptr_to_head);
  struct data *new_head;
  bool success;

  /* ... Reorganize the data structure ... */

  success = atomic_compare_exchange_weak(ptr_to_head,
                                         &old_head, new_head);
  if (!success) {
    cleanup_data_structure(new_head);
  }
  return success; /* Exit the thread */
}
```

Compliant Solution (`atomic_compare_exchange_weak()`)

To recover from spurious failures, a loop must be used. However, `atomic_compare_exchange_weak()` might fail because the head pointer changed, or the failure may be spurious. In either case, the thread must perform the work repeatedly until the compare-and-exchange succeeds, as shown in this compliant solution:

```
#include <stdatomic.h>
#include <stdbool.h>
#include <stddef.h>

struct data {
  struct data *next;
  /* ... */
};

extern void cleanup_data_structure(struct data *head);

int reorganize_data_structure(void *thread_arg) {
  struct data *_Atomic *ptr_to_head = thread_arg;
  struct data *old_head = atomic_load(ptr_to_head);
  struct data *new_head = NULL;
  struct data *saved_old_head;
  bool success;

  do {
    if (new_head != NULL) {
      cleanup_data_structure(new_head);
    }
    saved_old_head = old_head;

  /* ... Reorganize the data structure ... */

  } while (!(success = atomic_compare_exchange_weak(
               ptr_to_head, &old_head, new_head
             )) && old_head == saved_old_head);
  return success; /* Exit the thread */
}
```

This loop could also be part of a larger control flow; for example, the thread from the noncompliant code example could be retried if it returns `false`.

Compliant Solution (`atomic_compare_exchange_strong()`)

When a weak compare-and-exchange would require a loop and a strong one would not, the strong one is preferable, as in this compliant solution:

```
#include <stdatomic.h>
#include <stdbool.h>

struct data {
  struct data *next;
  /* ... */
};

extern void cleanup_data_structure(struct data *head);

int reorganize_data_structure(void *thread_arg) {
  struct data *_Atomic *ptr_to_head = thread_arg;
  struct data *old_head = atomic_load(ptr_to_head);
  struct data *new_head;
  bool success;

  /* ... Reorganize the data structure ... */

  success = atomic_compare_exchange_strong(
    ptr_to_head, &old_head, new_head
  );
  if (!success) {
    cleanup_data_structure(new_head);
  }
  return success; /* Exit the thread */
}
```

Risk Assessment

Failing to wrap the `atomic_compare_exchange_weak()` and `atomic_compare_exchange_weak_explicit()` functions in a loop can result in incorrect values and control flow.

Rule	Severity	Likelihood	Remediation Cost	Priority	Level
CON41-C	Low	Unlikely	Medium	**P2**	**L3**

Automated Detection

Tool	Version	Checker	Description
Coverity	2017.07	**BAD_CHECK_OF_WAIT_COND**	Implemented
Parasoft C/C++test	10.4.2	**CERT_C-CON41-a**	Wrap functions that can fail spuriously in a loop
Polyspace Bug Finder	R2019b	CERT C: Rule CON41-C	Checks for situations where functions that can spuriously fail are not wrapped in loop (rule fully covered)
PRQA QA-C	9.7	**2026**	
PRQA QA-C++	4.4	**5023**	

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 Oracle Secure Coding Standard for Java	THI03-J. Always invoke wait() and await() methods inside a loop	Prior to 2018-01-12: CERT: Unspecified Relationship

Bibliography

[ISO/IEC 9899:2011]	7.17.7.4, "The `atomic_compare_exchange` Generic Functions"
[Lea 2000]	1.3.2, "Liveness" 3.2.2, "Monitor Mechanics"