CON36-C. Wrap functions that can spuriously wake up in a loop

The `cnd_wait()` and `cnd_timedwait()` functions temporarily cede possession of a mutex so that other threads that may be requesting the mutex can proceed. These functions must always be called from code that is protected by locking a mutex. The waiting thread resumes execution only after it has been notified, generally as the result of the invocation of the `cnd_signal()` or `cnd_broadcast()` function invoked by another thread. The `cnd_wait()` function must be invoked from a loop that checks whether a condition predicate holds. A condition predicate is an expression constructed from the variables of a function that must be true for a thread to be allowed to continue execution. The thread pauses execution, via `cnd_wait()`, `cnd_timedwait()`, or some other mechanism, and is resumed later, presumably when the condition predicate is true and the thread is notified.

```c
#include <threads.h>
#include <stdbool.h>
extern bool until_finish(void);
extern mtx_t lock;
extern cnd_t condition;

void func(void) {
    if (thrd_success != mtx_lock(&lock)) {
        /* Handle error */
    }

    while (until_finish()) { /* Predicate does not hold */
        if (thrd_success != cnd_wait(&condition, &lock)) {
            /* Handle error */
        }
    }

    /* Resume when condition holds */
    if (thrd_success != mtx_unlock(&lock)) {
        /* Handle error */
    }
}
```

The notification mechanism notifies the waiting thread and allows it to check its condition predicate. The invocation of `cnd_broadcast()` in another thread cannot precisely determine which waiting thread will be resumed. Condition predicate statements allow notified threads to determine whether they should resume upon receiving the notification.

Noncompliant Code Example

This noncompliant code example monitors a linked list and assigns one thread to consume list elements when the list is nonempty.

This thread pauses execution using `cnd_wait()` and resumes when notified, presumably when the list has elements to be consumed. It is possible for the thread to be notified even if the list is still empty, perhaps because the notifying thread used `cnd_broadcast()`, which notifies all threads. Notification using `cnd_broadcast()` is frequently preferred over using `cnd_signal()` (See CON38-C. Preserve thread safety and liveness when using condition variables for more information.)

A condition predicate is typically the negation of the condition expression in the loop. In this noncompliant code example, the condition predicate for removing an element from a linked list is `!(list->next != NULL)`, whereas the condition expression for the while loop condition is `!(list->next == NULL)`.

This noncompliant code example nests the `cnd_wait()` function inside an `if` block and consequently fails to check the condition predicate after the notification is received. If the notification was spurious or malicious, the thread would wake up prematurely.
Compliant Solution

This compliant solution calls the `cnd_wait()` function from within a `while` loop to check the condition both before and after the call to `cnd_wait()`:

```c
#include <stddef.h>
#include <threads.h>

struct node_t {
    void *node;
    struct node_t *next;
};

struct node_t list;
static mtx_t lock;
static cnd_t condition;

void consume_list_element(void) {
    if (thrd_success != mtx_lock(&lock)) {
        /* Handle error */
    }

    while (list.next == NULL) {
        if (thrd_success != cnd_wait(&condition, &lock)) {
            /* Handle error */
        }
    }

    /* Proceed when condition holds */

    if (thrd_success != mtx_unlock(&lock)) {
        /* Handle error */
    }
}
```

Risk Assessment
Failure to enclose calls to the `cnd_wait()` or `cnd_timedwait()` functions inside a `while` loop can lead to indefinite blocking and denial of service (DoS).

<table>
<thead>
<tr>
<th>Rule</th>
<th>Severity</th>
<th>Likelihood</th>
<th>Remediation Cost</th>
<th>Priority</th>
<th>Level</th>
</tr>
</thead>
<tbody>
<tr>
<td>CON36-C</td>
<td>Low</td>
<td>Unlikely</td>
<td>Medium</td>
<td>P2</td>
<td>L3</td>
</tr>
</tbody>
</table>

### Automated Detection

<table>
<thead>
<tr>
<th>Tool</th>
<th>Version</th>
<th>Checker</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>Parasoft C/C++test</td>
<td>2020.2</td>
<td>CERT_C-CON36-a</td>
<td>Wrap functions that can spuriously wake up in a loop</td>
</tr>
<tr>
<td>Polyspace Bug Finder</td>
<td>R2020a</td>
<td>CERT C: Rule CON36-C</td>
<td>Checks for situations where functions that can spuriously wake up are not wrapped in loop (rule fully covered)</td>
</tr>
<tr>
<td>PRQA QA-C</td>
<td>9.7</td>
<td>2027</td>
<td></td>
</tr>
</tbody>
</table>

### 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)

<table>
<thead>
<tr>
<th>Taxonomy</th>
<th>Taxonomy Item</th>
<th>Relationship</th>
</tr>
</thead>
<tbody>
<tr>
<td>CERT Oracle Secure Coding Standard for Java</td>
<td>THI03-J. Always invoke wait() and await() methods inside a loop</td>
<td>Prior to 2018-01-12: CERT: Unspecified Relationship</td>
</tr>
</tbody>
</table>

### 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"