CON54-CPP. Wrap functions that can spuriously wake up in a loop

The `wait()`, `wait_for()`, and `wait_until()` member functions of the `std::condition_variable` class 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 `notify_one()` or `notify_all()` member functions invoked by another thread.

The `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 `wait()`, `wait_for()`, `wait_until()`, or some other mechanism, and is resumed later, presumably when the condition predicate is true and the thread is notified.

```
#include <condition_variable>
#include <mutex>

extern bool until_finish(void);
extern std::mutex m;
extern std::condition_variable condition;

void func(void) {
  std::unique_lock<std::mutex> lk(m);

  while (until_finish()) {  // Predicate does not hold.
    condition.wait(lk);
  }

  // Resume when condition holds.
}
```

The notification mechanism notifies the waiting thread and allows it to check its condition predicate. The invocation of `notify_all()` 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 `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 `notify_all()`, which notifies all threads. Notification using `notify_all()` is frequently preferred over using `notify_one()`. (See CON55-CPP. 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 != nullptr)`, whereas the condition expression for the `while` loop condition is `(list->next == nullptr)`.

This noncompliant code example nests the call to `wait()` 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.

```
#include <condition_variable>
#include <mutex>

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

static Node list;
static std::mutex m;
static std::condition_variable condition;

void consume_list_element(std::condition_variable &condition) {
  std::unique_lock<std::mutex> lk(m);

  if (list.next == nullptr) {
    condition.wait(lk);
  }

  // Proceed when condition holds.
}
```

Compliant Solution (Explicit loop with predicate)

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

```
#include <condition_variable>
#include <mutex>

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

static Node list;
static std::mutex m;
static std::condition_variable condition;

void consume_list_element() {
  std::unique_lock<std::mutex> lk(m);

  while (list.next == nullptr) {
    condition.wait(lk);
  }

  // Proceed when condition holds.
}
```

Compliant Solution (Implicit loop with lambda predicate)

The `std::condition_variable::wait()` function has an overloaded form that accepts a function object representing the predicate. This form of `wait()` behaves as if it were implemented as `while (!pred()) wait(lock);`. This compliant solution uses a lambda as a predicate and passes it to the `wait()` function. The predicate is expected to return true when it is safe to proceed, which reverses the predicate logic from the compliant solution using an explicit loop predicate.

```
#include <condition_variable>
#include <mutex>

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

static Node list;
static std::mutex m;
static std::condition_variable condition;

void consume_list_element() {
  std::unique_lock<std::mutex> lk(m);

  condition.wait(lk, []{ return list.next; });
  // Proceed when condition holds.
}
```

Risk Assessment

Failure to enclose calls to the `wait()`, `wait_for()`, or `wait_until()` member functions inside a `while` loop can lead to indefinite blocking and denial of service (DoS).

Rule	Severity	Likelihood	Remediation Cost	Priority	Level
CON54-CPP	Low	Unlikely	Medium	**P2**	**L3**

Automated Detection

Tool	Version	Checker	Description
Parasoft C/C++test	10.4.2	**CERT_CPP-CON54-a**	Wrap functions that can spuriously wake up in a loop
Polyspace Bug Finder	R2019b	CERT C++: CON54-CPP	Checks for situations where functions that can spuriously wake up are not wrapped in loop
PRQA QA-C++	4.4	**5019**	

Related Vulnerabilities

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

Related Guidelines

CERT Oracle Secure Coding Standard for Java	THI03-J. Always invoke wait() and await() methods inside a loop
SEI CERT C Coding Standard	CON36-C. Wrap functions that can spuriously wake up in a loop
SEI CERT C++ Coding Standard	CON55-CPP. Preserve thread safety and liveness when using condition variables

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"