POS47-C. Do not use threads that can be canceled asynchronously

In threading, pthreads can optionally be set to cancel immediately or defer until a specific cancellation point. Canceling asynchronously (immediately) is dangerous, however, because most threads are in fact not safe to cancel immediately.

The IEEE standards page states that

```
only functions that are cancel-safe may be called from a thread that is asynchronously cancelable.
```

Canceling asynchronously would follow the same route as passing a signal into the thread to kill it, posing problems similar to those in CON37-C. Do not call signal() in a multithreaded program, which is strongly related to SIG02-C. Avoid using signals to implement normal functionality. POS44-C and SIG02-C expand on the dangers of canceling a thread suddenly, which can create a data race condition.

Noncompliant Code Example

In this noncompliant code example, the worker thread is doing something as simple as swapping a and b repeatedly.

This code uses one lock. The global_lock mutex ensures that the worker thread and main thread do not collide in accessing the a and b variables.

The worker thread repeatedly exchanges the values of a and b until it is canceled by the main thread. The main thread then prints out the current values of a and b. Ideally, one should be 5, and the other should be 10.
volatile int a = 5;
volatile int b = 10;

/* Lock to enable threads to access a and b safely */
pthread_mutex_t global_lock = PTHREAD_MUTEX_INITIALIZER;

void* worker_thread(void* dummy) {
    int i;
    int c;
    int result;
    if ((result = pthread_setcanceltype(PTHREAD_CANCEL_ASYNCHRONOUS,&i)) != 0) { /* handle error */
    }

    while (1) {
        if ((result = pthread_mutex_lock(&global_lock)) != 0) { /* handle error */
            c = b;
            b = a;
            a = c;
            if ((result = pthread_mutex_unlock(&global_lock)) != 0) { /* handle error */
        }
        return NULL;
    }

    int main(void) {
        int result;
        pthread_t worker;

        if ((result = pthread_create( &worker, NULL, worker_thread, NULL)) != 0) { /* handle error */
            /* .. Do stuff... meanwhile worker thread runs for some time */

            /* since we don't know when the character is read in, the program could continue at any time */
            if ((result = pthread_cancel(worker)) != 0) { /* handle error */
                /* pthread_join waits for the thread to finish up before continuing */
                if ((result = pthread_join(worker, 0)) != 0) { /* handle error */
                    if ((result = pthread_mutex_lock(&global_lock)) != 0) { /* handle error */
                        printf("a: %i | b: %i", a, b);
                        if ((result = pthread_mutex_unlock(&global_lock)) != 0) { /* handle error */
                            return 0;
                        }
                    }
                }
            }
        }
    }
Noncompliant Code Example

In this example, the worker thread arranges to release the global_lock mutex if it gets interrupted:

```c
void release_global_lock(void* dummy) {
    int result;
    if ((result = pthread_mutex_unlock(&global_lock)) != 0) {
        /* handle error */
    }
}

void* worker_thread(void* dummy) {
    int i;
    int c;
    int result;

    if ((result = pthread_setcanceltype(PTHREAD_CANCEL_ASYNCHRONOUS,&i)) != 0) {
        /* handle error */
    }

    while (1) {
        if ((result = pthread_mutex_lock(&global_lock)) != 0) {
            /* handle error */
        }
        pthread_cleanup_push( release_global_lock, NULL);
        c = b;
        b = a;
        a = c;
        pthread_cleanup_pop(1);
    }
    return NULL;
}
```

The global variables are still subject to a race condition because an asynchronous cancel can happen at any time. For instance, the worker thread could be canceled just before the last line (a = c) and thereby lose the old value of b. Consequently, the main thread might print that a and b have the same value.

The program is still subject to the race condition where the main thread cancels the worker thread before it has invoked pthread_setcanceltype(). If this happens, the cancelation will be delayed until the worker thread calls pthread_setcanceltype().

Furthermore, though less likely, the program can still deadlock if the worker thread gets canceled after the global_lock is acquired but before pthread_cleanup_push() is invoked. In this case, the worker thread is canceled while holding global_lock, and the program will deadlock.

Compliant Solution

From IEEE standards page:

> The cancelability state and type of any newly created threads, including the thread in which main() was first invoked, shall be PTHREAD_CANCEL_ENABLE and PTHREAD_CANCEL_DEFERRED respectively.

Because the default condition for POSIX, according to the IEEE standards, is PTHREAD_CANCEL_DEFERRED, it is not necessary to invoke pthread_setcanceltype() in the compliant solution:
void* worker_thread(void* dummy) {
    int c;
    int result;

    while (1) {
        if ((result = pthread_mutex_lock(&global_lock)) != 0) {
            /* handle error */
        }
        c = b;
        b = a;
        a = c;
        if ((result = pthread_mutex_unlock(&global_lock)) != 0) {
            /* handle error */
        }

        /* now we're safe to cancel, creating cancel point */
        pthread_testcancel();
    }
    return NULL;
}

Because this code limits cancellation of the worker thread to the end of the while loop, the worker thread can preserve the data invariant that $a \neq b$. Consequently, the program might print that $a$ is 5 and $b$ is 10 or that $a$ is 10 and $b$ is 5, but they will always be revealed to have different values when the worker thread is canceled.

The other race conditions that plague the noncompliant code examples are not possible here. Because the worker thread does not modify its cancel type, it cannot be canceled before being improperly initialized. And because it cannot be canceled while the global_lock mutex is held, there is no possibility of deadlock, and the worker thread does not need to register any cleanup handlers.

Risk Assessment

Incorrectly using threads that asynchronously cancel may result in silent corruption, resource leaks, and, in the worst case, unpredictable interactions.

<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>POS47-C</td>
<td>Medium</td>
<td>Probable</td>
<td>Low</td>
<td>P12</td>
<td>L1</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>Axivion Bauhaus Suite</td>
<td>6.9.0</td>
<td>CertC-POS47</td>
<td></td>
</tr>
<tr>
<td>Parasoft C/C++test</td>
<td>10.4.2</td>
<td>CERT_C-POS47-a</td>
<td>The function 'pthread_setcanceltype()' should not be called with 'PTHREAD_CANCELASYNCHRONOUS' argument</td>
</tr>
<tr>
<td>PROQA QA-C</td>
<td>9.7</td>
<td>5035</td>
<td></td>
</tr>
</tbody>
</table>

Bibliography

Related Vulnerabilities

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

Related Guidelines

SEI CERT Oracle Coding Standard for Java: THI05-J. Do not use Thread.stop() to terminate threads
In Java, similar reasoning resulted in the deprecation of Thread.stop().

Bibliography

[MKS] pthread_cancel() Man Page