CON07-C. Ensure that compound operations on shared variables are atomic

Compound operations are operations that consist of more than one discrete operation. Expressions that include postfix or prefix increment (++), postfix or prefix decrement (`--`), or compound assignment operators always result in compound operations. Compound assignment expressions use operators such as `*=`, `/=`, `%=`, `+=`, `-=`, `<<=`, `>>=`, `^=`, and `|=`. Compound operations on shared variables must be performed atomically to prevent data races.

Noncompliant Code Example (Logical Negation)

This noncompliant code example declares a shared `_Bool flag` variable and provides a `toggle_flag()` method that negates the current value of `flag`:

```
#include <stdbool.h>

static bool flag = false;

void toggle_flag(void) {
  flag = !flag;
}

bool get_flag(void) {
  return flag;
}
```

Execution of this code may result in a data race because the value of `flag` is read, negated, and written back.

Consider, for example, two threads that call `toggle_flag()`. The expected effect of toggling `flag` twice is that it is restored to its original value. However, the following scenario leaves `flag` in the incorrect state:

Time	flag=	Thread	Action
1	true	t_1	Reads the current value of `flag`, `true`, into a cache
2	true	t_2	Reads the current value of `flag`, (still) `true`, into a different cache
3	true	t_1	Toggles the temporary variable in the cache to `false`
4	true	t_2	Toggles the temporary variable in the different cache to `false`
5	false	t_1	Writes the cache variable's value to `flag`
6	false	t_2	Writes the different cache variable's value to `flag`

As a result, the effect of the call by t_2 is not reflected in `flag`; the program behaves as if `toggle_flag()` was called only once, not twice.

Compliant Solution (Mutex)

This compliant solution restricts access to `flag` under a mutex lock:

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

static bool flag = false;
mtx_t flag_mutex;

/* Initialize flag_mutex */
bool init_mutex(int type) {
  /* Check mutex type */
  if (thrd_success != mtx_init(&flag_mutex, type)) {
    return false;  /* Report error */
  }
  return true;
}

void toggle_flag(void) {
  if (thrd_success != mtx_lock(&flag_mutex)) {
    /* Handle error */
  }
  flag = !flag;
  if (thrd_success != mtx_unlock(&flag_mutex)) {
    /* Handle error */
  }
}

bool get_flag(void) {
  bool temp_flag;
  if (thrd_success != mtx_lock(&flag_mutex)) {
    /* Handle error */
  }
  temp_flag = flag;
  if (thrd_success != mtx_unlock(&flag_mutex)) {
    /* Handle error */
  }
  return temp_flag;
}
```

This solution guards reads and writes to the `flag` field with a lock on the `flag_mutex`. This lock ensures that changes to `flag` are visible to all threads. Now, only two execution orders are possible. In one execution order, t_1 obtains the mutex and completes the operation before t_2 can acquire the mutex, as shown here:

Time	`flag=`	Thread	Action
1	`true`	t_1	Reads the current value of `flag`, `true`, into a cache variable
2	`true`	t_1	Toggles the cache variable to `false`
3	`false`	t_1	Writes the cache variable's value to `flag`
4	`false`	t_2	Reads the current value of `flag`, `false`, into a different cache variable
5	`false`	t_2	Toggles the different cache variable to `true`
6	`true`	t_2	Writes the different cache variable's value to `flag`

The other execution order is similar, except that t_2 starts and finishes before t_1.

Compliant Solution (`atomic_compare_exchange_weak()`)

This compliant solution uses atomic variables and a compare-and-exchange operation to guarantee that the correct value is stored in `flag`. All updates are visible to other threads.

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

static atomic_bool flag;

void init_flag(void) {
  atomic_init(&flag, false);
}
void toggle_flag(void) {
  bool old_flag = atomic_load(&flag);
  bool new_flag;
  do {
    new_flag = !old_flag;
  } while (!atomic_compare_exchange_weak(&flag, &old_flag, new_flag));
}

bool get_flag(void) {
  return atomic_load(&flag);
}
```

An alternative solution is to use the `atomic_flag` data type for managing Boolean values atomically.

Noncompliant Code Example (Addition of Primitives)

In this noncompliant code example, multiple threads can invoke the `set_values()` method to set the a and b fields. Because this code fails to test for integer overflow, users of this code must also ensure that the arguments to the `set_values()` method can be added without overflow (see INT32-C. Ensure that operations on signed integers do not result in overflow for more information).

```
static int a;
static int b;

int get_sum(void) {
  return a + b;
}

void set_values(int new_a, int new_b) {
  a = new_a;
  b = new_b;
}
```

The `get_sum()` method contains a race condition. For example, when a and b currently have the values `0` and `INT_MAX`, respectively, and one thread calls `get_sum()` while another calls `set_values(INT_MAX, 0)`, the `get_sum()` method might return either `0` or `INT_MAX`, or it might overflow. Overflow will occur when the first thread reads a and b after the second thread has set the value of a to `INT_MAX` but before it has set the value of b to `0`.

Noncompliant Code Example (Addition of Atomic Integers)

In this noncompliant code example, a and b are replaced with atomic integers.

```
#include <stdatomic.h>

static atomic_int a;
static atomic_int b;

void init_ab(void) {
  atomic_init(&a, 0);
  atomic_init(&b, 0);
}

int get_sum(void) {
  return atomic_load(&a) + atomic_load(&b);
}

void set_values(int new_a, int new_b) {
  atomic_store(&a, new_a);
  atomic_store(&b, new_b);
}
```

The simple replacement of the two `int` fields with atomic integers fails to eliminate the race condition in the sum because the compound operation `a.get() + b.get()` is still non-atomic. While a sum of some value of a and some value of b will be returned, there is no guarantee that this value represents the sum of the values of a and b at any particular moment.

Compliant Solution (Addition)

This compliant solution protects the `set_values()` and `get_sum()` methods with a mutex to ensure atomicity:

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

static atomic_int a;
static atomic_int b;
mtx_t flag_mutex;

/* Initialize everything */
bool init_all(int type) {
  /* Check mutex type */
  atomic_init(&a, 0);
  atomic_init(&b, 0);
  if (thrd_success != mtx_init(&flag_mutex, type)) {
    return false;  /* Report error */
  }
  return true;
}

int get_sum(void) {
  if (thrd_success != mtx_lock(&flag_mutex)) {
    /* Handle error */
  }
  int sum = atomic_load(&a) + atomic_load(&b);
  if (thrd_success != mtx_unlock(&flag_mutex)) {
    /* Handle error */
  }
  return sum;
}

void set_values(int new_a, int new_b) {
  if (thrd_success != mtx_lock(&flag_mutex)) {
    /* Handle error */
  }
  atomic_store(&a, new_a);
  atomic_store(&b, new_b);
  if (thrd_success != mtx_unlock(&flag_mutex)) {
    /* Handle error */
  }
}
```

Thanks to the mutex, it is now possible to add overflow checking to the `get_sum()` function without introducing the possibility of a race condition.

Risk Assessment

When operations on shared variables are not atomic, unexpected results can be produced. For example, information can be disclosed inadvertently because one user can receive information about other users.

Rule	Severity	Likelihood	Remediation Cost	Priority	Level
CON07-C	Medium	Probable	Medium	P8	L2

Related Guidelines

CERT Oracle Secure Coding Standard for Java	VNA02-J. Ensure that compound operations on shared variables are atomic
MITRE CWE	CWE-366, Race condition within a thread CWE-413, Improper resource locking CWE-567, Unsynchronized access to shared data in a multithreaded context CWE-667, Improper locking

Bibliography

[ISO/IEC 14882:2011]	Subclause 7.17, "Atomics"