CON40-C. Do not refer to an atomic variable twice in an expression

A consistent locking policy guarantees that multiple threads cannot simultaneously access or modify shared data. Atomic variables eliminate the need for locks by guaranteeing thread safety when certain operations are performed on them. The thread-safe operations on atomic variables are specified in the C Standard, subclauses 7.17.7 and 7.17.8 [ISO/IEC 9899:2011]. While atomic operations can be combined, combined operations do not provide the thread safety provided by individual atomic operations.

Every time an atomic variable appears on the left side of an assignment operator, including a compound assignment operator such as *=, an atomic write is performed on the variable. The use of the increment (++) or decrement (--) operators on an atomic variable constitutes an atomic read-and-write operation and is consequently thread-safe. Any reference of an atomic variable anywhere else in an expression indicates a distinct atomic read on the variable.

If the same atomic variable appears twice in an expression, then two atomic reads, or an atomic read and an atomic write, are required. Such a pair of atomic operations is not thread-safe, as another thread can modify the atomic variable between the two operations. Consequently, an atomic variable must not be referenced twice in the same expression.

Noncompliant Code Example (atomic_bool)

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

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

static atomic_bool flag = ATOMIC_VAR_INIT(false);

void init_flag(void) {
    atomic_init(&flag, false);
}

void toggle_flag(void) {
    bool temp_flag = atomic_load(&flag);
    temp_flag = !temp_flag;
    atomic_store(&flag, temp_flag);
}

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

Execution of this code may result in unexpected behavior because the value of flag is read, negated, and written back. This occurs even though the read and write are both atomic.

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 scenario in the following table leaves flag in the incorrect state.

<table>
<thead>
<tr>
<th>Time</th>
<th>flag</th>
<th>Thread</th>
<th>Action</th>
</tr>
</thead>
<tbody>
<tr>
<td>1</td>
<td>true</td>
<td>t₁</td>
<td>Reads the current value of flag, which is true, into a cache</td>
</tr>
<tr>
<td>2</td>
<td>true</td>
<td>t₂</td>
<td>Reads the current value of flag, which is still true, into a different cache</td>
</tr>
<tr>
<td>3</td>
<td>true</td>
<td>t₁</td>
<td>Toggles the temporary variable in the cache to false</td>
</tr>
<tr>
<td>4</td>
<td>true</td>
<td>t₂</td>
<td>Toggles the temporary variable in the different cache to false</td>
</tr>
<tr>
<td>5</td>
<td>false</td>
<td>t₁</td>
<td>Writes the cache variable's value to flag</td>
</tr>
<tr>
<td>6</td>
<td>false</td>
<td>t₂</td>
<td>Writes the different cache variable's value to flag</td>
</tr>
</tbody>
</table>

As a result, the effect of the call by t₂ is not reflected in flag; the program behaves as if toggle_flag() was called only once, not twice.

Compliant Solution (atomic_compare_exchange_weak() )

This compliant solution uses a compare-and-exchange to guarantee that the correct value is stored in flag. All updates are visible to other threads. The call to atomic_compare_exchange_weak() is in a loop in conformance with CON41-C. Wrap functions that can fail spuriously in a loop.
An alternative solution is to use the `atomic_flag` data type for managing Boolean values atomically. However, `atomic_flag` does not support a toggle operation.

**Compliant Solution (Compound Assignment)**

This compliant solution uses the `^=` assignment operation to toggle `flag`. This operation is guaranteed to be atomic, according to the C Standard, 6.5.16.2, paragraph 3. This operation performs a bitwise-exclusive-or between its arguments, but for Boolean arguments, this is equivalent to negation.

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

static atomic_bool flag = ATOMIC_VAR_INIT(false);

void toggle_flag(void) {
    flag ^= 1;
}

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

An alternative solution is to use a mutex to protect the atomic operation, but this solution loses the performance benefits of atomic variables.

**Noncompliant Code Example**

This noncompliant code example takes an atomic global variable `n` and computes $n + (n - 1) + (n - 2) + ... + 1$, using the formula $n \times (n + 1) / 2$:

```c
#include <stdatomic.h>

atomic_int n = ATOMIC_VAR_INIT(0);

int compute_sum(void) {
    return n * (n + 1) / 2;
}
```

The value of $n$ may change between the two atomic reads of $n$ in the expression, yielding an incorrect result.

**Compliant Solution**

This compliant solution passes the atomic variable as a function argument, forcing the variable to be copied and guaranteeing a correct result. Note that the function’s formal parameter need not be atomic, and the atomic variable can still be passed as an actual argument.

```c
#include <stdatomic.h>

int compute_sum(void) {
    return n * (n + 1) / 2;
}
```
#include <stdatomic.h>

```c
int compute_sum(int n) {
    return n * (n + 1) / 2;
}
```

## Risk Assessment

When operations on atomic variables are assumed to be atomic, but are not atomic, surprising data races can occur, leading to corrupted data and invalid control flow.

<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>CON40-C</td>
<td>Medium</td>
<td>Probable</td>
<td>Medium</td>
<td>P8</td>
<td>L2</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>Astére</td>
<td>19.04</td>
<td>multiple-atomic-accesses</td>
<td>Partially checked</td>
</tr>
<tr>
<td>Axivion Bauhaus Suite</td>
<td>6.9.0</td>
<td>Cert-C-CON40</td>
<td></td>
</tr>
<tr>
<td>Coverity</td>
<td>2017.07</td>
<td>EVALUATION_ORDER (partial) MISRA 2012 Rule 13.2 VOLATILE_ATOMICITY (possible)</td>
<td>Implemented</td>
</tr>
<tr>
<td>Parasoft C/C++test</td>
<td>10.4.2</td>
<td>CERT_C-CON40-a</td>
<td>Do not refer to an atomic variable twice in an expression</td>
</tr>
<tr>
<td>Polyspace Bug Finder</td>
<td>R2019b</td>
<td>CERT C: Rule CON40-C</td>
<td>Checks for:</td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td>* Atomic variable accessed twice in an expression</td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td>* Atomic load and store sequence not atomic</td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td>Rule fully covered.</td>
</tr>
<tr>
<td>PRQA QA-C</td>
<td>9.7</td>
<td>1114, 1115, 1116</td>
<td></td>
</tr>
<tr>
<td>RuleChecker</td>
<td>19.04</td>
<td>multiple-atomic-accesses</td>
<td>Partially checked</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>CWE 2.11</td>
<td>CWE-366, Race Condition within a Thread</td>
<td>2017-07-07: CERT: Rule subset of CWE</td>
</tr>
</tbody>
</table>

## CERT-CWE Mapping Notes

[Key here](#) for mapping notes

### CWE-366 and CON40-C

CON40-C = Subset( CON43-C) Intersection( CON32-C, CON40-C) = Ø

CWE-366 = Union( CON40-C, list) where list =

* C data races that do not involve an atomic variable used twice within an expression
Bibliography

| ISO/IEC 9899:2011 | 6.5.16.2, "Compound Assignment"  
7.17, "Atomics" |