**CON06-C. Ensure that every mutex outlives the data it protects**

********** Text below this note not yet converted from Java to C! **********

Programs must not use instance locks to protect static shared data because instance locks are ineffective when two or more instances of the class are created. Consequently, failure to use a static lock object leaves the shared state unprotected against concurrent access. Lock objects for classes that can interact with untrusted code must also be private and final, as shown in rule LCK00-J. Use private final lock objects to synchronize classes that may interact with untrusted code.

**Noncompliant Code Example (Nonstatic Lock Object for Static Data)**

This noncompliant code example attempts to guard access to the static `counter` field using a non-static lock object. When two `Runnable` tasks are started, they create two instances of the lock object and lock on each instance separately.

```java
public final class CountBoxes implements Runnable {
    private static volatile int counter;
    // ...
    private final Object lock = new Object();

    @Override
    public void run() {
        synchronized (lock) {
            counter++;
            // ...
        }
    }

    public static void main(String[] args) {
        for (int i = 0; i < 2; i++) {
            new Thread(new CountBoxes()).start();
        }
    }
}
```

This example fails to prevent either thread from observing an inconsistent value of `counter` because the increment operation on volatile fields fails to be atomic in the absence of proper synchronization. (See rule VNA02-J. Ensure that compound operations on shared variables are atomic for more information.)

**Noncompliant Code Example (Method Synchronization for Static Data)**

This noncompliant code example uses method synchronization to protect access to a static `counter` field.

```java
public final class CountBoxes implements Runnable {
    private static volatile int counter;
    // ...

    public synchronized void run() {
        counter++;
        // ...
    }
    // ...
}
```

In this case, the method synchronization uses the intrinsic lock associated with each instance of the class rather than the intrinsic lock associated with the class itself. Consequently, threads constructed using different `Runnable` instances may observe inconsistent values of `counter`.

**Compliant Solution (Static Lock Object)**

This compliant solution ensures the atomicity of the increment operation by locking on a static object.
public class CountBoxes implements Runnable {
    private static int counter;
    // ...
    private static final Object lock = new Object();

    public void run() {
        synchronized (lock) {
            counter++;
            // ...
        }
        // ...
    }
}

It is unnecessary to declare the `counter` variable volatile when using synchronization.

**Risk Assessment**

Using an instance lock to protect static shared data can result in nondeterministic behavior.

<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>CON06-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>Astrée</td>
<td>19.04</td>
<td></td>
<td>Supported, but no explicit checker</td>
</tr>
</tbody>
</table>

**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-667, Improper Locking</td>
<td>Prior to 2018-01-12: CERT:</td>
</tr>
</tbody>
</table>

**Bibliography**

[API 2006]