LCK00-J. Use private final lock objects to synchronize classes that may interact with untrusted code

There are two ways to synchronize access to shared mutable variables: method synchronization and block synchronization. Methods declared as synchronized and blocks that synchronize on the this reference both use the object as a monitor (that is, its intrinsic lock). An attacker can manipulate the system to trigger contention and deadlock by obtaining and indefinitely holding the intrinsic lock of an accessible class, consequently causing a denial of service (DoS).

One technique for preventing this vulnerability is the private lock object idiom [Bloch 2001]. This idiom uses the intrinsic lock associated with the instance of a private final java.lang.Object declared within the class instead of the intrinsic lock of the object itself. This idiom requires the use of synchronized blocks within the class's methods rather than the use of synchronized methods. Lock contention between the class's methods and those of a hostile class becomes impossible because the hostile class cannot access the private final lock object.

Static methods and state also share this vulnerability. When a static method is declared synchronized, it acquires the intrinsic lock of the class object before any statements in its body are executed, and it releases the intrinsic lock when the method completes. Untrusted code that has access to an object of the class, or of a subclass, can use the getClass() method to gain access to the class object and consequently manipulate the class object's intrinsic lock. Protect static data by locking on a private static final Object. Reducing the accessibility of the class to package-private provides further protection against untrusted callers.

The private lock object idiom is also suitable for classes that are designed for inheritance. When a superclass requests a lock on the object's monitor, a subclass can interfere with its operation. For example, a subclass may use the superclass object's intrinsic lock for performing unrelated operations, causing lock contention and deadlock. Separating the locking strategy of the superclass from that of the subclass ensures that they do not share a common lock and also permits fine-grained locking by supporting the use of multiple lock objects for unrelated operations. This increases the overall responsiveness of the application.

Objects that require synchronization must use the private lock object idiom rather than their own intrinsic lock in any case where untrusted code could:

- Subclass the class.
- Create an object of the class or of a subclass.
- Access or acquire an object instance of the class or of a subclass.

Subclasses whose superclasses use the private lock object idiom must themselves use the idiom. However, when a class uses intrinsic synchronization on the class object without documenting its locking policy, subclasses must not use intrinsic synchronization on their own class object. When the superclass documents its policy by stating that client-side locking is supported, the subclasses have the option to choose between intrinsic locking and using the private lock object idiom. Subclasses must document their locking policy regardless of which locking option is chosen. See rule TSM00-J. Do not override thread-safe methods with methods that are not thread-safe for related information.

When any of these restrictions are violated, the object's intrinsic lock cannot be trusted. But when these restrictions are obeyed, the private lock object idiom fails to add any additional security. Consequently, objects that comply with all of the restrictions are permitted to synchronize using their own intrinsic lock. However, block synchronization using the private lock object idiom is superior to method synchronization for methods that contain nonatomic operations that could either use a more fine-grained locking scheme involving multiple private final lock objects or that lack a requirement for synchronization. Nonatomic operations can be decoupled from those that require synchronization and can be executed outside the synchronized block. Both for this reason and for simplification of maintenance, block synchronization using the private lock object idiom is generally preferred over intrinsic synchronization.

Noncompliant Code Example (Method Synchronization)

This noncompliant code example exposes instances of the SomeObject class to untrusted code.
The untrusted code attempts to acquire a lock on the object's monitor and, upon succeeding, introduces an indefinite delay that prevents the synchronized `changeValue()` method from acquiring the same lock. Furthermore, the object locked is publicly available via the `lookup()` method.

Alternatively, an attacker could create a private `SomeObject` object and make it available to trusted code to use it before the attacker code grabs and holds the lock.

Note that in the untrusted code, the attacker intentionally violates rule LCK09-J. Do not perform operations that can block while holding a lock.

Noncompliant Code Example (Public Non-final Lock Object)

This noncompliant code example locks on a public nonfinal object in an attempt to use a lock other than `SomeObject`'s intrinsic lock.

```
public class SomeObject {
    public Object lock = new Object();

    public void changeValue() {
        synchronized (lock) {
            // ...
        }
    }
}
```

This change fails to protect against malicious code. For example, untrusted or malicious code could disrupt proper synchronization by changing the value of the lock object.

Noncompliant Code Example (Publicly Accessible Non-final Lock Object)

This noncompliant code example synchronizes on a publicly accessible but nonfinal field. The lock field is declared volatile so that changes are visible to other threads.

```
public class SomeObject {
    public Object lock = new Object();

    public void changeValue() {
        synchronized (lock) {
            // ...
        }
    }
}
```

This change fails to protect against malicious code. For example, untrusted or malicious code could disrupt proper synchronization by changing the value of the lock object.
public class SomeObject {
    private volatile Object lock = new Object();

    public void changeValue() {
        synchronized (lock) {
            // ...
        }
    }

    public void setLock(Object lockValue) {
        lock = lockValue;
    }
}

Any thread can modify the field’s value to refer to a different object in the presence of an accessor such as `setLock()`. That modification might cause two threads that intend to lock on the same object to lock on different objects, thereby permitting them to execute two critical sections in an unsafe manner. For example, if the lock were changed when one thread was in its critical section, a second thread would lock on the new object instead of the old one and would enter its critical section erroneously.

A class that lacks accessible methods to change the lock is secure against untrusted manipulation. However, it remains susceptible to inadvertent modification by the programmer.

**Noncompliant Code Example (Public Final Lock Object)**

This noncompliant code example uses a public final lock object.

```java
public class SomeObject {
    public final Object lock = new Object();

    public void changeValue() {
        synchronized (lock) {
            // ...
        }
    }
}
```

This noncompliant code example also violates rule **OBJ01-J. Limit accessibility of fields**.

**Compliant Solution (Private Final Lock Object)**

Thread-safe public classes that may interact with untrusted code must use a private final lock object. Existing classes that use intrinsic synchronization must be refactored to use block synchronization on such an object. In this compliant solution, calling `changeValue()` obtains a lock on a private final `Object` instance that is inaccessible to callers that are outside the class’s scope.

```java
public class SomeObject {
    private final Object lock = new Object(); // private final lock object

    public void changeValue() {
        synchronized (lock) { // Locks on the private Object
            // ...
        }
    }
}
```

A private final lock object can be used only with block synchronization. Block synchronization is preferred over method synchronization because operations without a requirement for synchronization can be moved outside the synchronized region, reducing lock contention and blocking. Note that it is unnecessary to declare the `lock` field `volatile` because of the strong visibility semantics of final fields. When granularity issues require the use of multiple locks, declare and use multiple private final lock objects to satisfy the granularity requirements rather than using a mutable reference to a lock object along with a setter method.

**Noncompliant Code Example (Static)**

This noncompliant code example exposes the class object of `SomeObject` to untrusted code.
The untrusted code attempts to acquire a lock on the class object's monitor and, upon succeeding, introduces an indefinite delay that prevents the synchronized `changeValue()` method from acquiring the same lock.

A compliant solution must also comply with rule LCK05-J. Synchronize access to static fields that can be modified by untrusted code. In the untrusted code, the attacker intentionally violates rule LCK09-J. Do not perform operations that can block while holding a lock.

Compliant Solution (Static)

Thread-safe public classes that both use intrinsic synchronization over the class object and may interact with untrusted code must be refactored to use a static private final lock object and block synchronization.

```java
public class SomeObject {
    private static final Object lock = new Object();

    public static void changeValue() {
        synchronized (lock) { // Locks on the private Object
            // ...
        }
    }
}
```

In this compliant solution, `changeValue()` obtains a lock on a private static `Object` that is inaccessible to the caller.

Exceptions

**LCK00-J-EX0**: A class may violate this rule when all of the following conditions are met:

- It sufficiently documents that callers must not pass objects of this class to untrusted code.
- The class cannot invoke methods, directly or indirectly, on objects of any untrusted classes that violate this rule.
- The synchronization policy of the class is documented properly.

Clients are permitted to use a class that violates this rule when all of the following conditions are met:

- Neither the client class nor any other class in the system passes objects of the violating class to untrusted code.
- The violating class cannot invoke methods, directly or indirectly, from untrusted classes that violate this rule.

**LCK00-J-EX1**: When a superclass of the class documents that it supports client-side locking and synchronizes on its class object, the class can support client-side locking in the same way and document this policy.

**LCK00-J-EX2**: Package-private classes that are never exposed to untrusted code are exempt from this rule because their accessibility protects against untrusted callers. However, use of this exemption should be documented explicitly to ensure that trusted code within the same package neither reuses the lock object nor changes the lock object inadvertently.

Risk Assessment

Exposing the lock object to untrusted code can result in DoS.

<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>LCK00-J</td>
<td>low</td>
<td>probable</td>
<td>medium</td>
<td>P4</td>
<td>L3</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>The Checker Framework</td>
<td>2.1.3</td>
<td>Lock Checker</td>
<td>Concurrency and lock errors (see Chapter 6)</td>
</tr>
<tr>
<td>Parasoft Jtest</td>
<td>2021.1</td>
<td>CERT.LCK00.SOPF</td>
<td>Do not synchronize on &quot;public&quot; fields since doing so may cause deadlocks</td>
</tr>
<tr>
<td>SonarQube</td>
<td>6.7</td>
<td>S2445</td>
<td></td>
</tr>
</tbody>
</table>

**Related Guidelines**

- MITRE CWE [CWE-412](#). Unrestricted externally accessible lock
- [CWE-413](#). Improper resource locking

**Bibliography**

[Bloch 2001](#) Item 52. Document Thread Safety