LCK08-J. Ensure actively held locks are released on exceptional conditions

An exceptional condition can circumvent the release of a lock, leading to deadlock. According to the Java API [API 2014]:

> *A `ReentrantLock` is owned by the thread last successfully locking, but not yet unlocking it. A thread invoking `lock` will return, successfully acquiring the lock, when the lock is not owned by another thread.*

Consequently, an unreleased lock in any thread will prevent other threads from acquiring the same lock. Programs must release all actively held locks on exceptional conditions. Intrinsic locks of class objects used for method and block synchronization are automatically released on exceptional conditions (such as abnormal thread termination).

This guideline is an instance of FIO04-J. Release resources when they are no longer needed. However, most Java lock objects are not closeable, so they cannot be automatically released using Java 7's `try`-with-resources feature.

Noncompliant Code Example (Checked Exception)

This noncompliant code example protects a resource, an open file, by using a `ReentrantLock`. However, the method fails to release the lock when an exception occurs while performing operations on the open file. When an exception is thrown, control transfers to the `catch` block and the call to `unlock()` never executes.

```
public final class Client {
  private final Lock lock = new ReentrantLock();

  public void doSomething(File file) {
    InputStream in = null;
    try {
      lock.lock();
      in = new FileInputStream(file);

      // Perform operations on the open file

      lock.unlock();
    } catch (FileNotFoundException x) {
      // Handle exception
    } finally {
      if (in != null) {
        try {
          in.close();
        } catch (IOException x) {
          // Handle exception
        }
      }
    }
  }
}
```

Noncompliant Code Example (`finally` Block)

This noncompliant code example attempts to rectify the problem of the lock not being released by invoking `Lock.unlock()` in the `finally` block. This code ensures that the lock is released regardless of whether or not an exception occurs. However, it does not acquire the lock until after trying to open the file. If the file cannot be opened, the lock may be unlocked without ever being locked in the first place.

```
public final class Client {
  private final Lock lock = new ReentrantLock();

  public void doSomething(File file) {
    InputStream in = null;
    try {
      in = new FileInputStream(file);
      lock.lock();
      // Perform operations on the open file
    } catch (FileNotFoundException fnf) {
      // Forward to handler
    } finally {
      lock.unlock();
      if (in != null) {
        try {
          in.close();
        } catch (IOException e) {
          // Forward to handler
        }
      }
    }
  }
}
```

Compliant Solution (`finally` Block)

This compliant solution encapsulates operations that could throw an exception in a `try` block immediately after acquiring the lock (which cannot throw). The lock is acquired just before the `try` block, which guarantees that it is held when the `finally` block executes.

```
public final class Client {
  private final Lock lock = new ReentrantLock();

  public void doSomething(File file) {
    InputStream in = null;
    lock.lock();
    try {
      in = new FileInputStream(file);
      // Perform operations on the open file
    } catch (FileNotFoundException fnf) {
      // Forward to handler
    } finally {
      lock.unlock();

      if (in != null) {
        try {
          in.close();
        } catch (IOException e) {
          // Forward to handler
        }
      }
    }
  }
}
```

Compliant Solution (Execute-Around Idiom)

The execute-around idiom provides a generic mechanism to perform resource allocation and cleanup operations so that the client can focus on specifying only the required functionality. This idiom reduces clutter in client code and provides a secure mechanism for resource management.

In this compliant solution, the client's `doSomething()` method provides only the required functionality by implementing the `doSomethingWithFile()` method of the `LockAction` interface without having to manage the acquisition and release of locks or the open and close operations of files. The `ReentrantLockAction` class encapsulates all resource management actions.

```
public interface LockAction {
  void doSomethingWithFile(InputStream in);
}

public final class ReentrantLockAction {
  private static final Lock lock = new ReentrantLock();

  public static void doSomething(File file, LockAction action)  {
    InputStream in = null;
    lock.lock();
    try {
      in = new FileInputStream(file);
      action.doSomethingWithFile(in);
    } catch (FileNotFoundException fnf) {
      // Forward to handler
    } finally {
      lock.unlock();

      if (in != null) {
        try {
          in.close();
        } catch (IOException e) {
          // Forward to handler
        }
      }
    }
  }
}

public final class Client {
  public void doSomething(File file) {
    ReentrantLockAction.doSomething(file, new LockAction() {
      public void doSomethingWithFile(InputStream in) {
        // Perform operations on the open file
      }
    });
  }
}
```

Noncompliant Code Example (Unchecked Exception)

This noncompliant code example uses a `ReentrantLock` to protect a `java.util.Date` instance—recall that `java.util.Date` is thread-unsafe by design.

```
final class DateHandler {

  private final Date date = new Date();

  private final Lock lock = new ReentrantLock();

  // str could be null
  public void doSomething(String str) {
    lock.lock();
    String dateString = date.toString();
    if (str.equals(dateString)) {
      // ...
    }
    // ...

    lock.unlock();
  }
}
```

A runtime exception can occur because the `doSomething()` method fails to check whether `str` is a null reference, preventing the lock from being released.

Compliant Solution (`finally` Block)

This compliant solution encapsulates all operations that can throw an exception in a `try` block and releases the lock in the associated `finally` block. Consequently, the lock is released even in the event of a runtime exception.

```
final class DateHandler {

  private final Date date = new Date();

  private final Lock lock = new ReentrantLock();

  // str could be null
  public void doSomething(String str) {
    lock.lock();
    try {
      String dateString = date.toString();
      if (str != null && str.equals(dateString)) {
        // ...
      }
      // ...

    } finally {
      lock.unlock();
    }
  }
}
```

The `doSomething()` method also avoids throwing a `NullPointerException` by ensuring that the string does not contain a null reference.

Risk Assessment

Failure to release locks on exceptional conditions could lead to thread starvation and deadlock.

Rule	Severity	Likelihood	Remediation Cost	Priority	Level
LCK08-J	Low	Likely	Low	P9	L2

Automated Detection

Some static analysis tools are capable of detecting violations of this rule.

Tool	Version	Checker	Description
Parasoft Jtest	10.3	**TRS.RLF, BD.TRS.LOCK**	Implemented
ThreadSafe	1.3	**CCE_LK_UNRELEASED_ON_EXN**	Implemented

Related Vulnerabilities

The GERONIMO-2234 issue report describes a vulnerability in the Geronimo application server. If the user single-clicks the keystore portlet, the user will lock the default keystore without warning. This causes a crash and stack trace to be produced. Furthermore, the server cannot be restarted because the lock is never cleared.

Related Guidelines

MITRE CWE	CWE-883, Deadlock

Bibliography

[API 2014]	Class `ReentrantLock`