THI05-J. Do not use `Thread.stop()` to terminate threads

Threads preserve class **invariants** when they are allowed to exit normally. Programmers often attempt to terminate threads abruptly when they believe the task is complete, the request has been canceled, or the program or Java Virtual Machine (JVM) must shut down expeditiously.

Certain thread APIs were introduced to facilitate thread suspension, resumption, and termination but were later deprecated because of inherent design weaknesses. For example, the `Thread.stop()` method causes the thread to immediately throw a `ThreadDeath` exception, which usually stops the thread. More information about deprecated methods is available in MET02-J. Do not use deprecated or obsolete classes or methods.

Invoking `Thread.stop()` results in the release of all locks a thread has acquired, potentially exposing the objects protected by those locks when those objects are in an inconsistent state. The thread might catch the `ThreadDeath` exception and use a `finally` block in an attempt to repair the inconsistent object or objects. However, doing so requires careful inspection of all synchronized methods and blocks because a `ThreadDeath` exception can be thrown at any point during the thread’s execution. Furthermore, code must be protected from `ThreadDeath` exceptions that might occur while executing `catch` or `finally` blocks [Sun 1999]. Consequently, programs must not invoke `Thread.stop()`.

Removing the `java.lang.RuntimePermission stopThread` permission from the security policy file prevents threads from being stopped using the `Thread.stop()` method. Although this approach guarantees that the program cannot use the `Thread.stop()` method, it is nevertheless strongly discouraged. Existing trusted, custom-developed code that uses the `Thread.stop()` method presumably depends on the ability of the system to perform this action. Furthermore, the system might fail to correctly handle the resulting security exception. Additionally, third-party libraries may also depend on use of the `Thread.stop()` method.

Refer to ERR09-J. Do not allow untrusted code to terminate the JVM for information on preventing data corruption when the JVM is abruptly shut down.

Noncompliant Code Example (Deprecation `Thread.stop()`)

This noncompliant code example shows a thread that fills a vector with pseudorandom numbers. The thread is forcefully stopped after a given amount of time.

```java
public final class Container implements Runnable {
    private final Vector<Integer> vector = new Vector<Integer>(1000);

    public Vector<Integer> getVector() {
        return vector;
    }

    @Override public synchronized void run() {
        Random number = new Random(123L);
        int i = vector.capacity();
        while (i > 0) {
            vector.add(number.nextInt(100));
            i--;
        }
    }

    public static void main(String[] args) throws InterruptedException {
        Thread thread = new Thread(new Container());
        thread.start();
        Thread.sleep(5000);
        thread.stop();
    }
}
```

Because the `Vector` class is **thread-safe**, operations performed by multiple threads on its shared instance are expected to leave it in a consistent state. For instance, the `Vector.size()` method always returns the correct number of elements in the vector, even after concurrent changes to the vector, because the vector instance uses its own intrinsic lock to prevent other threads from accessing it while its state is temporarily inconsistent.

However, the `Thread.stop()` method causes the thread to stop what it is doing and throw a `ThreadDeath` exception. All acquired locks are subsequently released [API 2014]. If the thread were in the process of adding a new integer to the vector when it was stopped, the vector would become accessible while it is in an inconsistent state. For example, this could result in `Vector.size()` returning an incorrect element count because the element count is incremented after adding the element.

Compliant Solution (volatile flag)

This compliant solution uses a volatile flag to request thread termination. The `shutdown()` accessor method is used to set the flag to true. The thread's `run()` method polls the `done` flag and terminates when it is set.
public final class Container implements Runnable {
    private final Vector<Integer> vector = new Vector<Integer>(1000);
    private volatile boolean done = false;

    public Vector<Integer> getVector() {
        return vector;
    }

    public void shutdown() {
        done = true;
    }

    @Override public synchronized void run() {
        Random number = new Random(123L);
        int i = vector.capacity();
        while (!done && i > 0) {
            vector.add(number.nextInt(100));
            i--;
        }
    }

    public static void main(String[] args) throws InterruptedException {
        Container container = new Container();
        Thread thread = new Thread(container);
        thread.start();
        Thread.sleep(5000);
        container.shutdown();
    }
}

Compliant Solution (Interruptible)

In this compliant solution, the Thread.interrupt() method is called from main() to terminate the thread. Invoking Thread.interrupt() sets an internal interrupt status flag. The thread polls that flag using the Thread.interrupted() method, which both returns true if the current thread has been interrupted and clears the interrupt status flag.

public final class Container implements Runnable {
    private final Vector<Integer> vector = new Vector<Integer>(1000);

    public Vector<Integer> getVector() {
        return vector;
    }

    @Override public synchronized void run() {
        Random number = new Random(123L);
        int i = vector.capacity();
        while (!Thread.interrupted() && i > 0) {
            vector.add(number.nextInt(100));
            i--;
        }
    }

    public static void main(String[] args) throws InterruptedException {
        Container container = new Container();
        Thread thread = new Thread(container);
        thread.start();
        Thread.sleep(5000);
        thread.interrupt();
    }
}

A thread may use interruption for performing tasks other than cancellation and shutdown. Consequently, a thread should be interrupted only when its interruption policy is known in advance. Failure to do so can result in failed interruption requests.

Risk Assessment
Forcing a thread to stop can result in inconsistent object state. Critical resources could also leak if cleanup operations are not carried out as required.

<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>THI05-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>Parasoft Jtest</td>
<td>10.3</td>
<td>TRS.THRD</td>
<td>Implemented</td>
</tr>
</tbody>
</table>

**Related Guidelines**

- SEI CERT C Coding Standard: POS47-C. Do not use threads that can be canceled asynchronously
- MITRE CWE: CWE-705, Incorrect Control Flow Scoping

**Android Implementation Details**

On Android, `Thread.stop()` was deprecated in API level 1.

**Bibliography**

- [API 2006]: Class `Thread`, Method `stop`
- [Darwin 2004]: Section 24.3, "Stopping a Thread"
- [Goetz 2006]: Chapter 7, "Cancellation and Shutdown"
- [JavaThreads 2004]: Section 2.4, "Two Approaches to Stopping a Thread"
- [JDK7 2008]: Concurrency Utilities, More information: Java Thread Primitive Deprecation
- [JPL 2006]: Section 14.12.1, "Don't Stop"
  Section 23.3.3, "Shutdown Strategies"
- [Sun 1999]