VNA05-J. Ensure atomicity when reading and writing 64-bit values

The Java Language Specification (JLS) allows 64-bit long and double values to be treated as two 32-bit values. For example, a 64-bit write operation could be performed as two separate 32-bit operations.

According to the JLS, §17.7, "Non-Atomic Treatment of double and long" [JLS 2015]:

> For the purposes of the Java programming language memory model, a single write to a non-volatile long or double value is treated as two separate writes: one to each 32-bit half. This can result in a situation where a thread sees the first 32 bits of a 64-bit value from one write, and the second 32 bits from another write.

> Some implementations may find it convenient to divide a single write action on a 64-bit long or double value into two write actions on adjacent 32-bit values. For efficiency's sake, this behavior is implementation-specific; an implementation of the Java Virtual Machine is free to perform writes to long and double values atomically or in two parts.

This behavior can result in indeterminate values being read in code that is required to be thread-safe. Consequently, multithreaded programs must ensure atomicity when reading or writing 64-bit values.

Noncompliant Code Example

In this noncompliant code example, if one thread repeatedly calls the assignValue() method and another thread repeatedly calls the printLong() method, the printLong() method could occasionally print a value of i that is neither zero nor the value of the j argument:

```java
class LongContainer {
    private long i = 0;
    void assignValue(long j) {
        i = j;
    }
    void printLong() {
        System.out.println("i = " + i);
    }
}
```

A similar problem can occur when i is declared double.

Compliant Solution (Volatile)

This compliant solution declares i volatile. Writes and reads of long and double volatile values are always atomic.

```java
class LongContainer {
    private volatile long i = 0;
    void assignValue(long j) {
        i = j;
    }
    void printLong() {
        System.out.println("i = " + i);
    }
}
```

It is important to ensure that the argument to the assignValue() method is obtained from a volatile variable or obtained as the result of an atomic read. Otherwise, a read of the variable argument can itself expose a vulnerability.

The semantics of volatile explicitly exclude any guarantee of the atomicity of compound operations that involve read-modify-write sequences such as incrementing a value (see VNA02-J. Ensure that compound operations on shared variables are atomic for more information).

Exceptions
VNA05-J-EX0: If all reads and writes of 64-bit long and double values occur within a synchronized region, the atomicity of the read/write is guaranteed. This requires both that the value is exposed only through synchronized methods in the class and that the value is inaccessible from other code (whether directly or indirectly). For more information, see VNA02-J. Ensure that compound operations on shared variables are atomic.

VNA05-J-EX1: This rule can be ignored for platforms that guarantee that 64-bit long and double values are read and written as atomic operations. Note, however, that such guarantees are not portable across different platforms.

Risk Assessment

Failure to ensure the atomicity of operations involving 64-bit values in multithreaded applications can result in reading and writing indeterminate values. However, many Java Virtual Machines read and write 64-bit values atomically even though the specification does not require them to.

<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>VNA05-J</td>
<td>Low</td>
<td>Unlikely</td>
<td>Medium</td>
<td>P2</td>
<td>L3</td>
</tr>
</tbody>
</table>

Automated Detection

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

<table>
<thead>
<tr>
<th>Tool</th>
<th>Version</th>
<th>Checker</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>ThreadSafe</td>
<td>1.3</td>
<td>CCE_SL_INCONSISTENT CCE_CC_CALLBACK_ACCESS CCE_SL_MIXED CCE_SL_INCONSISTENT_COL CCE_SL_MIXED_COL CCE_CC_UNSAFE_CONTENT CCE_FF_VOLATILE</td>
<td>Implemented</td>
</tr>
</tbody>
</table>

Related Guidelines

MITRE CWE CWE-667, Improper Locking

Bibliography

[Goetz 2006] Section 3.1.2, "Non-atomic 64-Bit Operations"
[Goetz 2004c]
[JLS 2015] "Non-Atomic Treatment of double and long"