An Empirical Security Study of the Native Code in the JDK

Gang Tan, Boston College ⇒ Lehigh University
Jason Croft, Boston College
Java Security

- Various holes identified and fixed
  - [Dean, Felten, Wallach 96]; [McGraw & Felten 99]; [Saraswat 97]; [Liang & Bracha 99]; …

- Formal models of various aspects of Java
  - Stack inspection [Wallach & Felton 98]
  - JVVML model [Freund & Mitchell 03] [Stata & Abadi 99]
  - …

- Machine-checked theorems and proofs [Klein & Nipkow 06]
What About Native Methods?

- A Java class can have native methods
  - Implemented in C/C++
  - Interact with Java through the Java Native Interface (JNI)
- Outside of the Java security model
  - No type safety
  - Outside of the Java sandbox
- By default, Java applets does not allow loading non-local native code
What About the Native code in the Java Development Kit (JDK)?

- `java.io.FileInputStream`
  - A Java wrapper for C code that invokes system libraries

- `java.util.zip.*`
  - Java wrappers that invoke the Zlib C compression/decompression library

- **The JDK’s native code is trusted by default**
How Large Is This Trust?

Java

C/C++

> 800 kloc in Sun’s JDK 1.6
The JDK’s Native Code: On the Increase

![Graph showing the increase in LOC of JDK's Native Code from JDK 1.4.2 to JDK 1.6.](image)
Triggering a Bug in the Native Code
An Obvious Example

class Vulnerable {
   public native void bcopy(byte[] arr);
   ...
}

Java code

void Java_Vulnerable_bcopy (…, jobject jarr) {
   char buffer[512];
jbyte *arr = GetByteArrayElements(jarr, 0);
strcpy(buffer, arr);
}

C code

Unbounded string copy!
An Empirical Security Study

- Folklore: bugs in the JDK’s native code is a threat to Java security
  - All 800,000 lines are too big to be trusted
- Problem: how to alleviate the threat?
- An empirical study is a first and important step
- Goals of the study:
  - Collect evidence that the native code is a realistic threat to Java security
  - Collect data to understand the extent
  - Characterize bug patterns
Approach to Characterizing Bug Patterns

- Static analysis tools + manual inspection
  - Common C vulnerabilities
    - Splint, ITS4, Flawfinder
  - Bug patterns particular to the JNI
    - Custom built scanners: grep-based scripts; CIL-based scanners
    - Bug patterns inferred from the JNI manual
  - Manual inspection to rule out false positives
    - An HTML interface for browsing the code: GNU Global source code tag system; htags
Approach and Scope of the Study

- **Pros**
  - Can cover many bug patterns
  - The scanning results are fairly complete: good for collecting empirical evidence

- **Cons**
  - Lots of manual work: cannot cover all 800,000 lines
  - Limiting the scope: target directories
    - Native code under share/native/java and solaris/native/java
    - They implement the native methods of the classes under java.*
    - 38,000 LOC of C code
A Taxonomy of Bugs in the Native Code of the JDK
# A Summary of the Bugs Identified

<table>
<thead>
<tr>
<th>Bugs</th>
<th>Security Critical</th>
<th>Tools used</th>
</tr>
</thead>
<tbody>
<tr>
<td>Mishandling JNI exceptions</td>
<td>11 Y</td>
<td>grep-based scripts</td>
</tr>
<tr>
<td>C pointers as Java integers</td>
<td>38 N</td>
<td>Our CIL scanner</td>
</tr>
<tr>
<td>Race conditions in file accesses</td>
<td>3 Y</td>
<td>ITS4, Flawfinder</td>
</tr>
<tr>
<td>Buffer Overflows</td>
<td>5* Y</td>
<td>Splint, ITS4, Flawfinder</td>
</tr>
<tr>
<td>Mem. Management Flaws</td>
<td>29 N</td>
<td>Splint, grep-based scripts</td>
</tr>
<tr>
<td>Insufficient error checking</td>
<td>40 Y</td>
<td>Splint, grep-based scripts</td>
</tr>
<tr>
<td><strong>TOTAL</strong></td>
<td><strong>126</strong></td>
<td><strong>59</strong></td>
</tr>
</tbody>
</table>
Java Exceptions

```java
try {
    if (checkFails()) {
        throw ...;
    }
    doSensitiveOp();
} catch (Exception e) {
    ...
}
```

- When an exception is thrown
  - The JVM transfers the control to the nearest enclosing catch statement
JNI Exceptions Are Different!

- The JNI exception won’t be thrown until the C method returns
Mishandling JNI Exceptions

- Things become more complicated when function calls are involved

```c
void c_fun (...) {
    util_fun(); // Might throw a JNI exception
    if (ExceptionOccurred()) {...; return;}
    {...};
}
```

- Our study found 11 cases of mishandling JNI Exceptions
  - Mismatch between the programming models
  - Blame the programmers or the API designers
Another Bug Pattern: C Pointers as Java Integers

- Often, need to store C pointers at the Java side
  - However, how to declare the types of the C pointers in Java?

- Commonly used pattern
  - Cast the C pointers to Java integers
  - When passed back to C, they are cast back to pointers

- Example:
  - Zlib maintains a z_stream struct for keeping state info
  - A Java Deflater object needs to store a pointer to this C struct
Bogus Pointers to C

- The pattern is unsafe if the Java side can inject arbitrary integers to C

- Example [Greenfieldboyce & Foster]: GTK
  ```java
  class GUILib {
    public native static void setFocus (int windowPtr);
    ...
  }
  ```
  - A public method that anybody can invoke with bogus pointers

- Some cases will enable reading/writing arbitrary memory locations
Bogus Pointers to C in the JDK

- The target directories in the JDK
  - 38 native methods that accept Java integers and cast them to pointers
  - Not security critical: they are declared as private
  - Attackers cannot invoke private methods, without Java Reflection

- Should still be fixed
  - Java Reflection: can invoke private methods
  - Java Reflection + C pointers as Java integers: read/write arbitrary memory locations

Still type safe

Type unsafe!
A Summary of Bug Patterns

- We found a range of bugs: buffer overflows, misusing JNI exceptions, ...
  - O(100) bugs in 38 kloc code
- Other bug patterns (we did not find violations)
  - Type misuses
  - Deadlocks
  - Violating the Java sandbox security model
Remedies, Limitations, and Future Directions
Remedy: Static Analysis

- Find and remove bugs
- The static tools used in the study do not scale
  - High proportions of false positives (FP)

<table>
<thead>
<tr>
<th>Off-the-shelf tools</th>
<th>FP rates</th>
</tr>
</thead>
<tbody>
<tr>
<td>ITS4 -c1</td>
<td>97.5%</td>
</tr>
<tr>
<td>Flawfinder</td>
<td>98.3%</td>
</tr>
<tr>
<td>Splint</td>
<td>99.8%</td>
</tr>
</tbody>
</table>

- Same story for our own scripts and scanners
- A large amount of time on manual inspection
  - Prone to human errors
Reducing False Positives

- Advanced static analysis techniques can help
  - Software model checking; abstract interpretation; type qualifiers; theorem proving techniques
- Mishandling JNI exceptions: dataflow analysis
  - How many more bugs can we expect to find?
    - 11 violations out of 337 Throws
    - 2471 Throws => \( \approx 80 \) violations
Reducing False Positives: Inter-Language Analysis

During our manual inspection, we often went back and forth between Java and C side to decide if a warning is a bug.

```c
jint deflatebytes(…, jarray b, jint len, jint off) {
    ...
    out_buf = (jbyte *) malloc (len);
    ...
    SetByteArrayRegion(b, off, len, out_buf)
    ...
}
```

Is this a buffer overrun?

No range checks on len and off!

Well, it depends on how the Java side invokes it.
Most existing source-code analysis tools are limited a priori to code written in a single language.

Extending the horizon of analysis:

- Saffire [Furr & Foster, PLDI ’05, ESOP ‘06]
- APLT [Zhang et al., ISSTA ’06]
- ILEA [Tan & Morrisett, OOPSLA ’07]
  - Enable Java analysis to also understand the behavior of C code.
Remedy: Dynamic Mechanisms

- SafeJNI [Tan et al. ISSSE ‘06]: dynamic checks + static pointer type system
  - Statically reject or dynamically stop ill-behaved C programs
  - Leverage CCured [Necula et al.] to provide internal memory safety to C code
  - Checkings at the boundary between Java and C
  - Performance slowdown: Microbenchmark: 14%-119%; Zlib: 74%
- Limitations: concurrency; efficiency
- Assembly level monitoring: SFI, XFI
Remedy: Rewrite the Native Code in Safer Languages

- Java
- Cyclone
- Better interfaces between Java and C
  - Jeannie [Hirzel and Grimm OOPSLA ‘07]
  - Janet
In Summary

- Native code in the JDK is a time bomb to Java security

- In the short term
  - Develop scalable static analysis tools to eliminate bugs
  - Efficient dynamic mechanisms

- In the long term
  - Most of the C code should be converted into Java code---CLASSPATH’s long term goal

- Same problem with .NET
The End