Security Analysis of the Native Code in Sun's JDK

Jason Croft Gang Tan
Department of Computer Science, Boston College

1 Introduction

Most real software systems are *multilingual*; that is, they consist of components developed in different programming languages. Multilingual software systems are convenient in practice. Developers can reuse existing code modules, and can also mix and match strengths of different languages.

However, multilingual software can have strong security implications. For example, the Java Native Interface (JNI) allows type-safe Java code to interact with unsafe C code. When a type-safe language interacts with an unsafe language in the same address space, in general, the overall application becomes unsafe. In addition, it is difficult to write safe and reliable multilingual software, as past research has shown [6, 7, 4, 5]. It usually requires programmers to carefully take into account the discrepancies between languages on issues such as language features, data layout, memory management, safety/security assumptions, and many others.

Figure 1 shows a contrived example of a multilingual program developed through the JNI. The Java class "Vulnerable" contains a native method, which is realized by a C function. The C function is susceptible to a buffer overflow as it performs an unbounded string copy to a 512-byte buffer. Consequently, an adversary can craft malicious inputs to the public Java byteCopy() method, and overtake the JVM. This example demonstrates that, although the JVM provides various kinds of mechanisms to ensure safety and security, the native C code in Java applications could render the JVM unsafe.

Due to the fundamental insecurity of native C/C++ code, the default policy of the JVM is to reject non-local native code. Nonetheless, there is already a large amount of trusted native code that comprises a significant portion of Sun's Java Development Kit (JDK). For instance, the classes under java.util.zip are just wrappers that invoke the popular Zlib C library. Of the 2.2 million lines of code in JDK 1.6, there are nearly 700,000 lines of C/C++ code, accounting for almost 25% of the JDK's source code. Any vulnerability in this trusted native code can compromise the security of the JVM. Several vulnerabilities have already been discovered in this code [4, 7].

2 Objectives

Since the native code inside the JDK is critical to Java security, examining and ensuring its security is of great practical value. Prior work [5, 8] has addressed some safety issues inherent in the JNI, but none have scrutinized the JDK's native code. In this research, we are performing a systematic security analysis of this large and security-critical code. We hope to achieve the following objectives:

- By collecting empirical evidence, we hope to expose patterns of bugs. The patterns can guide the solutions that we will propose to mediate them.
- The results of our security analysis will help to strengthen the overall security infrastructure of Sun's JVM platform. We plan to report all discovered bugs to Sun.
- We believe inspection of Java's code will give insight into a more extensive view of the security issues related to multilingual software environment, by uncovering previously unknown issues. We will be interested in collecting evidence that shows the multilingual environment increases the types and frequency of bugs.

Given the large amount of trusted native code in the JDK, bugs are likely to exist. Our objective is not to discover all bugs or vulnerabilities in this code. Rather, we would like to compile enough evidence to effectively conclude the common types of bugs particular to such types of multilingual software. After enough evidence has been collected, we plan to begin developing tools and methods to discover and mediate the types of problems we encounter.

3 Methodology

Discovering vulnerabilities in the source code is itself a difficult and time consuming task. As no general methodology exists to find all bugs in a program, part of the success of our endeavor relies in our approach to this problem. One option is to systematically examine the code using static

Java code

```
class Vulnerable {
  /* declare a native method */
  private native int bcopy(byte arr[]);
  public void byteCopy(byte[] arr) {
    /* call the native method */
    bcopy(arr);
  }
  static {
    /* load the shared library that implements
     the native method */
    System.loadLibrary("Vulnerable");
  }
}
```

C code

```
#include <jni.h>
#include "Vulnerable.h"
JNIEXPORT jint JNICALL
Java_Vulnerable_bcopy
(JNIEnv *env, jobject *obj, jobject *arr)
/* env is an interface pointer through which a JNI API
   function can be called.
   obj is the reference to the object on which the method is invoked.
   arr is the reference to the array. */
{
  char buffer[512];
  jbyte *carr;
  car = (*env)->GetByteArrayElements
                (env, arr, NULL);
  strcpy(buffer, carr);
}
```

Figure 1. Vulnerable JNI Code. An array of bytes is passed from Java code to C code. The buffer in the C code can be overflowed from malicious inputs to the Java function byteCopy().

analysis tools. These tools, however, are generally unsound and incomplete, resulting in a large number of false positives and false negatives. Nonetheless, we have chosen to employ this method, despite its drawbacks, as it is much more efficient than manual auditing and can automate much of the process. We used a combination of Splint [3], Cigital's ITS4 [2], and Flawfinder [1].

We characterize the results from these tools as such:

- Those warnings that are not bugs (*i.e.*, false positives).
- Those warnings that are real bugs in the native C/C++ code, but cannot be triggered by an attacker by sending malicious inputs to Java functions. These bugs are worth noting in our work, but pose no threat to the JVM. For example, bugs located in those C functions that are not used by Java belongs to this category.
- Those warnings that are real bugs and can also be triggered by an attacker from Java's side. These are the most critical flaws, as they may be exploitable vulnerabilities.

We also characterize vulnerabilities according to where they occur. The C code in the JDK are broken into two categories: those native C libraries, such as the ZLib C library, and the C "glue code" that serves as interfaces between Java functions and the underlying C libraries. We expect that C code in the second category has a higher vulnerability rate, because it is easy to make mistakes in the interface code, and because it is more likely that an attacker can trigger the vulnerability from Java's side.

We have already scanned several hundred source files and thousands of lines of code thus far. Additionally, we have organized our reportings based on our classifications and have examples from each category.

4 Work in Progress

Our plan involves continued analysis of C/C++ code in the JDK until adequate evidence has been collected. All the bugs we identify will be reported to Sun. Based on the evidence we collect, we will develop tools for finding/fixing particular kinds of bugs in the JNI. We also plan to extend our work beyond the JNI to include other types of multilingual environments, such the Microsoft Common Language Runtime (CLR).

References

[1] Flawfinder. `http://www.dwheeler.com/flawfinder/`.

[2] ITS4. `http://www.cigital.com/its4/`.

[3] Splint. `http://www.splint.org/`.

[4] Chris Evans. CESA-2006-004 - rev 2. `http://scary.beasts.org/security/CESA-2006-004.html`.

[5] Michael Furr and Jeffrey S. Foster. Polymorphic type inference for the JNI. In *European Symposium on Programming (ESOP '06)*, pages 309–324, 2006.

[6] Sheng Liang. *Java Native Interface: Programmer's Guide and Reference*, chapter 10. Addison-Wesley Longman Publishing Co., Inc., 1999.

[7] Marc Schönefeld. Hunting Flaws in JDK. In *Blackhat Europe 2003*, May 2003.

[8] Gang Tan, Andrew W. Appel, Srimat Chakradhar, Anand Raghunathan, Srivaths Ravi, and Daniel Wang. Safe Java Native Interface. In *Proceedings of IEEE International Symposium on Secure Software Engineering*, pages 97–106, 2006.