A Heuristic Fuzz Test Generator for Java Native Interface

National Key Laboratory of Science and Technology on Information System Security
Beijing, China

Jinjing Zhao
misszhaojinjing@hotmail.com
INTRODUCTION

JNI is the Java native interface between the Host JVM and native libraries, by which Java program can invoke the system functions implemented by C/C++. JNI allows type-safe Java code to interact with unsafe C code, then the overall application becomes unsafe.

Java Code

```java
class Vulnerable{
    // Declare a native method
    private native void bcopy(byte[] arr);
    public void byteCopy(byte[] arr){
        // Call the native method
        bcopy(arr);
    }
    static{
        System.loadLibrary("Vulnerable");
    }
}
```

C Code

```c
#include <jni.h>
#include "Vulnerable.h"
JNIEXPORT void JNICALL Java_Vulnerable_bcopy
(JNIEnv *env, jobject obj, jobject arr)
{
    char buffer[512];
    jbyte *carr;
    carr = (*env)->GetByteArrayElements(env, arr, 0);
    // Unbounded string copy to a local buffer
    strcpy(buffer, carr);
    (*env)->ReleaseByteArrayElements(env, arr, carr, 0);
}
```
INTRODUCTION

While any security guarantees provided by Java might be invalidated by the native methods. Any vulnerability in this trusted native code can compromise the security of the Java program.

So it is very important to the whole program for the JNIs are used properly and safely. Examining and ensuring its security is of great practical value.

In this paper, we present a new fuzzing framework on JNI vulnerability detection. Based on the branch predication information of program, our method can accurately aim at the JNI functions to generate the fuzzer inputs. Furthermore, the fuzzing process could test the native code through JNI to find the vulnerabilities may be caused by invoking the high-risk system APIs.
Security Problems in JNI

A java function can invoke the JNI to get a system call to control the hardware.

Some studies on the security of JNI identified the types of bugs in JNI:
- **Direct access through Java references**
- **Wrong Use of Interface pointers**
- **Out-of-bounds array access**
- **Violating access control rules**
- **Manual memory management**
- **Arguments of wrong classes**
- **Calling wrong methods**
- **Exception handling**
- **Bypassing the security manager**
DESIGN OF AIMING-JNI SYSTEM

1. Pre-disposal process
2. Static analysis process
3. Dynamic analysis process

![Diagram of AIMING-JNI System]

- **Pre-disposal process**
  - Select the Windows sensitive system API
  - Sensitive API Database
  - Analysis *.dll file, search the JNI
  - Relations between JNI and Windows API
  - Decompiled the definition file of sensitive JNI

- **Static Analysis process**
  - Static Analysis
    - CFG
    - BPT
    - BPIT
    - PIT
    - Relation Function

- **Dynamic Analysis process**
  - Dynamic Analysis
    - Corresponding Input
    - Fuzzing Test
    - Program Bugs
1. Pre-disposal process

Some of the functions in the DLL files are system calls which can cause the system interruptions. We define these 284 functions as the sensitive APIs.

Next, follow the JNI invoking rules, we can find the JNI definition and decompile the corresponding .dll files to get its implementation and all the invoked Windows system API.
2. Static analysis process

1) Draw the call flow graph (CFG) of the program;
2) Build the branch path tree (BPT) of the program from the CFG;
3) Slice the program with every branch predication variable in branch lines and build the 
   \textit{BPIT}.
   The item in \textit{BPIT} is formed with four terms as \{Line Number, Branch Predication, 
   Corresponding Input, Relation Function\};
4) According to the BPT and BPIT, deduce the information on every possible path and 
   analyze the relation between paths and the inputs to build the PIT.
   The \textit{PIT} is composed of five properties as \{ID, Path, Branch Point, Corresponding Input, 
   Relation Function\}.

Then, we can get the \textit{relations between the execution path with the branch predication}s.
## 2. Static analysis process

### The Relation Between Branch Predication and the Generated Inputs

<table>
<thead>
<tr>
<th>Branch Predication</th>
<th>Generated Inputs</th>
</tr>
</thead>
<tbody>
<tr>
<td>$n_1 + n_2 + \cdots + n_i &gt; m$</td>
<td>$n_1 = N, n_2 = N+1, \ldots, n_i = N+i-1, m = -1+i(2N+i-1)/2$</td>
</tr>
<tr>
<td>$n_1 + n_2 + \cdots + n_i &lt; m$</td>
<td>$n_1 = N, n_2 = N+1, \ldots, n_i = N+i-1, m = 1+i(2N+i-1)/2$</td>
</tr>
<tr>
<td>$n_1 + n_2 + \cdots + n_i = m$</td>
<td>$n_1 = N, n_2 = N+1, \ldots, n_i = N+i-1, m = i(2N+i)/2$</td>
</tr>
<tr>
<td>$n_1 + n_2 + \cdots + n_i \neq m$</td>
<td>$n_1 = N, n_2 = N+1, \ldots, n_i = N+i-1, m \neq i(2N+i-1)/2$</td>
</tr>
</tbody>
</table>

Establish dimensional rectangular coordinate system, taking $a$, $b$ for the axis, and choose the points surrounded by three lines.

$a \neq b \land a \neq M$ Establish dimensional rectangular coordinate system, taking $a$, $b$ for the axis, and choose the points outside the two lines.

$a \neq a+b+c=d \land a=M+1, b=N, c=M, d=M+1, a=N, b=N+1, c=M+1, d=M, a=N+1, b=N, c=M+1, d=M$. 

$a \neq N, b=\neq N, c=M, d=M, a=N, b=N, c=M, d=M, a \neq N, b=\neq N, c=M, d=M$. 

$a \neq N, b=\neq N, c=M, d=M, a=N, b=N, c=M, d=M, a \neq N, b=\neq N, c=M, d=M$. 

$a \neq N, b=\neq N, c=M, d=M, a=N, b=N, c=M, d=M, a \neq N, b=\neq N, c=M, d=M$. 

$a \neq N, b=\neq N, c=M, d=M, a=N, b=N, c=M, d=M, a \neq N, b=\neq N, c=M, d=M$. 

$a \neq N, b=\neq N, c=M, d=M, a=N, b=N, c=M, d=M, a \neq N, b=\neq N, c=M, d=M$. 

$a \neq N, b=\neq N, c=M, d=M, a=N, b=N, c=M, d=M, a \neq N, b=\neq N, c=M, d=M$. 

$a \neq N, b=\neq N, c=M, d=M, a=N, b=N, c=M, d=M, a \neq N, b=\neq N, c=M, d=M$. 

$a \neq N, b=\neq N, c=M, d=M, a=N, b=N, c=M, d=M, a \neq N, b=\neq N, c=M, d=M$. 

$a \neq N, b=\neq N, c=M, d=M, a=N, b=N, c=M, d=M, a \neq N, b=\neq N, c=M, d=M$. 

$a \neq N, b=\neq N, c=M, d=M, a=N, b=N, c=M, d=M, a \neq N, b=\neq N, c=M, d=M$.

$a \neq N, b=\neq N, c=M, d=M, a=N, b=N, c=M, d=M, a \neq N, b=\neq N, c=M, d=M$.
3. Dynamic analysis process

1) Sensitive APIs can be found in the CFG according to the sensitive API database that we defined in the pre-disposal process.
2) Then we can draw the sensitive execution paths from the inputs to the sensitive APIs.
3) The last step is to generate the fuzzer with the PIT table.

In order to elevate the efficiency of the fuzzing process, we make the following rules:

*Rule 1: Maximum Iteration Times Rule.*
*R​ule 2: The Minimum Variable Changed Rule.*
*R​ule 3: DFS and Nearest Rule.*
PEInsight, which can find the sensitive JNI and construct their CFGs.

The native function \_JVM\_GetClassSignature@8 is selected to be analyzed.
We compared the fuzzing times and sensitive paths coverage between AIMING-JNI and JPF. The results show that AIMING-JNI can use less fuzzing times to reach the sensitive windows APIs.
Our research makes the following contributions:

- We analyze all 2280 functions in the three most important Windows XP system DLL files and pick up all the 284 system interruption related functions as the high-risk system APIs. Aiming at those java functions who invoke these APIs as the fuzz targets can efficiently reduce the test range and cost;
- An automatic static analysis tool for searching the high-risk functions and their execution paths in the DLL files has been implemented, which is called PEInsight;
- An improved fuzzer is presented based on JPF, which is a very popular fuzzing system to verify executable Java bytecode programs. The relationship between program branch predications and inputs plays very important roles in it.

In the future, we plan to improve the efficiency and reduce the expense for large-scale program fuzz.
THANKS!