Android Fuzzing Tool
CSE509 Final Project Report

Haolong Ning
Yubin Lai
Ruonan Hao

December 17, 2013

Date Performed: November 20, 2013
Instructor: Professor Rob Johnson

1 Introduction

1.1 Background

App markets are stirring a paradigm shift in the way software is provisioned to the end users. However, it has also given rise to a new set of security challenges [1]. In parallel with the emergence of app markets, we have witnessed increased security threats that are exploiting the provisioning software. Arguably, this is nowhere more evident than in the Android market, where numerous cases of apps infected with malwares and spywares have been reported [2]. However, from a technical point of view, the key obstacle is the ability to rapidly assess the security of APIs. The problem is that security testing is generally a manual, expensive, and cumbersome process [1].

A form of automated security testing that does not require test case specification or significant upfront effort is fuzz testing, or simply fuzzing [3]. In short, fuzzing is a form of negative software testing that feeds malformed and unexpected input data to a program with the objective of revealing security vulnerabilities. Fuzz testing has two obvious benefits. First, employing unexpected inputs mean that rarely used code paths are tested. Second, the generation of random inputs and the tests themselves can be fully automated, so that a large number of tests can be quickly performed [4]. Programs that are used to create and examine fuzz tests are called fuzzers.

Fuzz testing has a history that stretches back to at least the 1980s, when fuzz testers were used to test command-line utilities. The history of system call fuzz testing is nearly as long. During his talk at linux.conf.au 2013, Dave Jones, the developer of Trinity, noted that the earliest system call fuzz tester that he had heard of was Tsys, which was created around 1991 for System V Release 4. Another early example was a fuzz tester developed at the University of Wisconsin in the mid-1990s that was run against a variety of kernels, including Linux.

Tsys was an example of a ”naive” fuzz tester. It simply generated random bit patterns, placed them in appropriate registers, and then executed a system call. About a decade later, the kg_crashme tool was developed to perform fuzz testing on Linux. Like Tsys, kg_crashme was a naive fuzz tester. Naive fuzz testers are capable of finding some kernel bugs, but the use of purely random inputs greatly limits their efficacy.

Compared with naive fuzz testers, trinity, a fuzzer for the linux kernel, performs intelligent fuzz testing by incorporating specific knowledge about each system call that is tested. The idea is to reduce the time spent running ”useless” tests, thereby reaching deeper into the tested code and increasing the chances of testing a more interesting case that may result in an unexpected error.

Additionally, an SMS protocol fuzzer [4] was recently shown to be highly effective in finding severe security vulnerabilities in all three major smartphone platforms, namely Android, iPhone, and Windows Mobile. In the case of Android, fuzzing found a security vulnerability triggered by simply receiving a particular type of SMS message, which not only kills the phones telephony process, but also kicks the target device off the network [5].

1.2 Objectives

In this project, we will follow the idea of intelligent fuzz testing (e.g., trinity) introduced in Section 1.1 to design a fuzzing tool to test the Android APIs. Our objective is to investigate any crashes or bugs provoked by the fuzzer and hopefully, report any real bugs we find to the Android developers.

The remainder of this report is organized as follows. Section 2 described the design of our fuzzer, which is followed by detailed analysis and evaluation of experiment results in Section 3. Finally, Section 4 concludes the report.

2 Methodology

This section presents the methodology we use to design the android fuzzer. Generally, it can be divided into three part. The framework is shown as follows in Fig. 1.

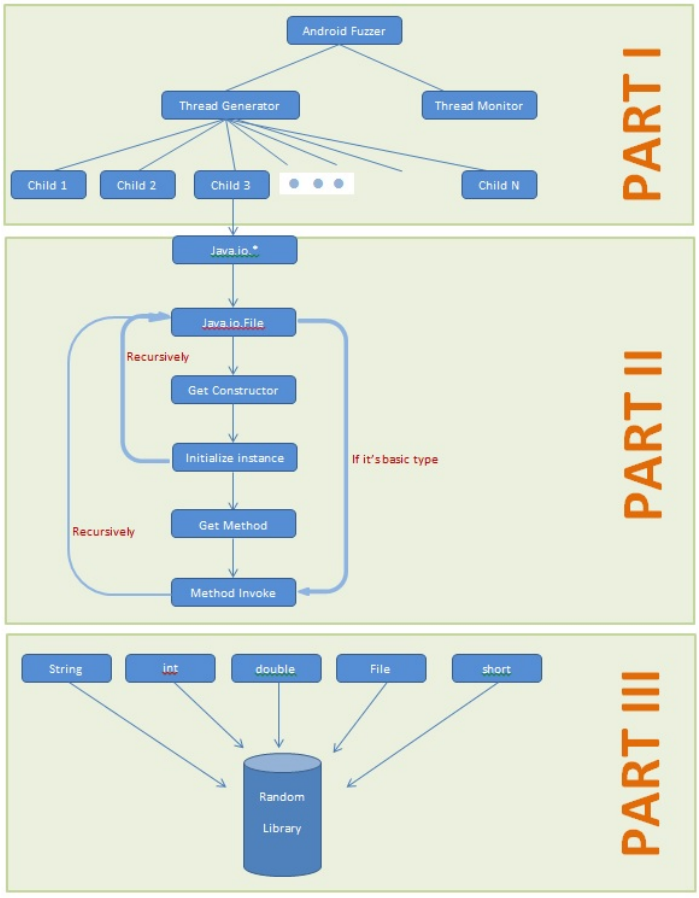

Figure 1: Framework of the Android Fuzzer

At the beginning, the fuzzer creates a multi-thread program that recursively tracks build inner object type by Java reflection technique. That is, the program starts from Part I, which will generate multiple thread to execute Part II and Part III.

2.1 Part I

The $ThreadGenerator$ process performs various initializations and then kicks off a certain number of child processes to perform the system call tests. Here, we set the maximum number of threads to be 20, as the speed of Android emulator tested in our environment is limited. Of course, the android fuzzer is initially designed for much larger number of threads (2000 or more) as long as the hardware resource is powerful enough.

The initial $AndroidFuzzer$ process uses a shared memory to record various information about child processes and global variables (e.g., open file descriptor numbers, the number of system calls that succeeded or failed, etc.).

The $ThreadMonitor$ process is used to ensure the fuzz testing system is operating correctly. Specifically, if a child is not progressing, the $ThreadMonitor$ process will kill it. Meanwhile, the $ThreadGenerator$ process will start a new child process to replace it whenever it detects a child has terminated. Additionally, the integrity of the memory region is also monitored by $ThreadMonitor$. This can effectively prevent memory corruption from child processes.

2.2 Part II

After program generate a thread, it will goes into second part (Part II). Each thread, will repeat Part II and Part III exactly the same. The more thread we generate, the more randomization we have, and more possible to find out bugs among API. For each thread will fall into following stages:

2.2.1 Stage I

Get a package name (API) from user. We designed in this way that it's user's responsibility to provide the correct API. Specifically, we chose **`Java.io.*`** as our testing package. We maintained a table for all the classes inside this package:

However, for further work, it should be available for any package. As the number of package increased, the size of table will also increased respectfully. Since the table of classes are maintained manually and time constrained, we first target on **`Java.io.File`** specifically. For completely version, It will contain all classes.

Java.io.BufferedInputStream	Java.io.BufferedOutputStream
Java.io.BufferedReader	Java.io.BufferedWriter
Java.io.ByteArrayInputStream	Java.io.ByteArrayOutputStream
Java.io.CharArrayReader	Java.io.CharArrayWriter
Java.io.Console	Java.io.DataInputStream
Java.io.DataOoutputStream	Java.io.File
Java.io.File	Java.io.FileDescriptor
Java.io.FileInputStream	Java.io.FileOutputStream
Java.io.FilePermission	Java.io.FileReader
Java.io.FileWriter	Java.io.InputStream
Java.io.InputStreamReader	Java.io.LineNumberInputStream
Java.io.LineNumberReader	Java.io.ObjectInputStream
Java.io.ObjectInputStream.GetField	Java.io.ObjectOutputStream
Java.io.ObjectOutputStream	Java.io.ObjectOutputStream.putField
Java.io.ObjectStreamClass	Java.io.ObjectStreamField
Java.io.OutputStream	Java.io.OutputStreamWriter
Java.io.PipedInputStream	Java.io.PipedOutputStream
Java.io.PipedReader	Java.io.PipedWriter
Java.io.PushBackInputStream	Java.io.PushBackReader
Java.io.RandomAccessFile	Java.io.Reader
Java.io.SequenceInputStream	Java.io.SeriallzeablePermission
Java.io.StreamTokenizer	Java.io.StringBufferInputStream
Java.io.StringReader	Java.io.StringWriter
Java.io.Writer	

2.2.2 Stage II

Randomly pick up a class that inside the selected package, take **Java.io.File** as example, create a the corresponding object for this class. Each class contains one or more than one constructors:

```
File(File parent, String child)
File(String pathname)
File(String parent, String child)
File(URI uri)
```

We used Java reflection to get all these constructors and randomly pick up one of them, Though this constructor, we initialize a instance (Object) for this class.

```
...
Class c = c.forName(Java.io.File);
Constructors cons = c.getConstructors();
Constructor constructor = cons[random(cons.size() - 1];
Object obj = constructor.getInstance();
...
```

After we initialized a instance of this class, we can first get all the methods associated with this class, and then randomly pick one to test:

```
...
Method[] methods = obj.getMethods();
Method method = methods[random(methods.size() - 1];
method.invoke(...);
...
```

upon this point, we finished Stage II, that pick up one path to test among multiple choice by reflection.

2.2.3 Stage III

It is very common that to initialize one class or one method need another class (object). We recursively repeat stage I and stage II to get instance of those class in order to invoke the method successfully. For example, a method under

```
Java.io.File:
int compareTo(File pathname)
```

To test **compareTo** method, we have to first pass a **File** object. Repeat Stage I and Stage II:

1) get constructors of **File** Class;
2) randomly pick up one among those constructors;
3) create a new instance through this constructor;
4) and finally pass this new instance to the method.

Since to get a real instance, we may have to keep repeating Stage I and Stage II multiple time, we build method recursively handle those steps until we hit the basic type. In order words, this recursively call won't terminate until we hit the basic type, and this will be the end of Part II.

2.3 Part III

So far, the program will be able to generate a thread to randomly pick a method to test, based on a randomly chosen class, and it will recursively initialize new instance type until it hit the basic type. After one thread hit the basic type, program will goes into another path, that pick up a value from pre-defined table and feed it back to the caller. Those basic type defined as following types: **String, int, double, float, short, byte, long, boolean, File, URI** for testing **Java.io.File** class, specifically. For numerical type, our random value will come from 5 kind of specific case: Maximum positive value, minimum positive value, 0, minimum negative value, maximum negative value. Take **double** as example, those values will be:

```
+Double.getMaxValue()   +Double.getMinValue()   0
-Double.getMinValue()   -Double.getMaxValue()
```

Above case including: **int, double, float, short, byte, long**. For type **String** we handled following cases:

```
// one
str = "1----/";
strList.add(str);

str = "$PATH$:/";
strList.add(str);

str = "\64\2\67\34/";
strList.add(str);

str = "--.-%DIR%-/";
strList.add(str);

str = "----/";
strList.add(str);

str = "./.";
strList.add(str);

str = "-/";
strList.add(str);

str = "\01\01\01\01\01\01\01\01\01\01\01";
strList.add(str);

str = "\00\00\00\00";
strList.add(str);

str = "\177\177\177\177\177\177\177\177\177\177";
strList.add(str);

str = "######";
strList.add(str);

str = "\140echo␣1";
strList.add(str);

str = "\140;echo␣1";
strList.add(str);

str = "ls␣*sjgl";
strList.add(str);

str = "&df␣";
strList.add(str);
```

```
str = "./---␣/";
strList.add(str);

str = "--";
strList.add(str);

str = "%n%s%d";
strList.add(str);

str = "--10101/";
strList.add(str);

str = "ls␣-";
strList.add(str);

str = "er␣&&␣";
strList.add(str);

str = "--../--.";
strList.add(str);

str = "||";
strList.add(str);

str = "&&for␣!";
strList.add(str);

str = "0XFFFFFFFF";
strList.add(str);

str = "0X00000000";
strList.add(str);

str = "0X--␣0X";
strList.add(str);

str = "0XFF0101110";
strList.add(str);

str = "0X&er111";
strList.add(str);
```

Since we focus on **Java.io.File** as our initial testing case, we also create treat **File** as our basic type. We generate lots of file at beginning and store them in a array. We build methods for help us handle this, each method will response for one type of file:

```
this.gen_LongNameFile();
this.gen_MutipleLevelFile();
this.gen_ReadonlyFile();
this.gen_WriteonlyFile();
this.gen_ExcuteonlyFile();
this.gen_noRWXFile();
this.gen_BigFile();
this.gen_EmptyFile();
this.gen_RootFile();
this.gen_LongNameDir();
this.gen_MutipleLevelDir();
this.gen_ReadonlyDir();
this.gen_WriteonlyDir();
this.gen_ExcuteonlyDir();
this.gen_noRWXDir();
this.gen_RootDir();
this.gen_WeirdNameDir();
this.gen_WeirdNameFile();
```

For most of them is easy to understand from their name, same explain below:

gen_MultipleLevelFile()\gen_MultipleLevelDir() These two methods will create a file or directory within 15 level:

```
level_1\level_2\...\level _15\file .
```

gen_WeiredNameFile()\gen_WeiredNameDir() These two methods will create a file or directory with weird name:

```
dir = absDir + "1----";
dir = absDir + "$PATH$:";
dir = absDir + "\64\2\67\34";
dir = absDir + "----";
dir = absDir + "--.-%DIR%-";
dir = absDir + "%DIR%-";
dir = absDir + ".";
dir = absDir + "-";
dir = absDir + "\01\01\01\01\01\01\01\01\01\01\01\01";
dir = absDir + "\00\00\00\00";
dir = absDir + "\177\177\177\177\177\177\177\177\177";
dir = absDir + "######";
dir = absDir + "\140echo␣1";
dir = absDir + "\140;echo␣1";
dir = absDir + "\140echo␣1";
```

3 Results & Evaluation

As evaluation, there are a few shortages need to point out. First, the program didn't test all the classes for all the API. We chose a **`Java.io.*`** as our minimum version to demo. However, as I mentioned before, as completely version of this program, it should be able to randomly pick up a package from entirely API and it should maintain all the classes. It will require sometime that developer manually input those data into the system. For long term's point's view, it should not be a problem. Of course, as project description mentioned, we are allowed to choose a specific API to test. Second, basic type is limited. Since we target on Java IO, there not too much basic type we can choose. We decided that **`File`** should a basic type and therefore program will stop reflection here. In addition, program will randomly pick up a existing file, which we created already, feed it back to the program. However, if we choose other package, for instance **`Socket`** to test, then **`Socket`** show to the basic type and developers should pre-created different wired versions of Socket to feed to program. Third, randomization is limited. In practice, we believe we should generate more than 200 threads to test, or even more, in order to see any real bug in android API. However, it's impossible to run 200 threads in our platforms as the Android Emulator will crash due to the limited memory issues.

In our fuzzer, if program crashes, it indicates that there exists bug in android API. However, we haven't found any system crash yet before Emulator stop. We did see **`NULLOjbectException`** few time and we decide it should be considered as normal. Overall, this android fuzzer satisfied a basic fuzzer needs and has all the fundamental functionalities.

4 Conclusion

In summary, we have built an multi-thread Android Fuzzer to test Android APIs. By using Java reflection, we built a little intelligence for the program for better understanding of the APIs. We defined the basic types that terminate Java reflection, and setup random values for those basic types to feed into APIs. Overall, it contains all the fundamental features a Fuzzer needs and it runs on background successfully. For further work, we should increase the capacity of the package to test, and comprehend basic types so that make Fuzzer more intelligent. Finally, we also should run this Java Fuzzer on more powerful machine to expect better results.

References

[1] S.Malek, N. Esfahani, T. Kacem, R. Mahmood, N. Mirzaei and A. Stavrou, "A Framework for Automated Security Testing of Android Applications on the Cloud", 2012 IEEE 6th International Conference on Software Security and Reliability Companion (SERE-C), pp. 20-22, June 2012.
[2] Malicious Mobile Threats Report 2010/2011, White paper, Juniper Networks Global Threat Center Research.

[3] A. Takanen, et al. Fuzzing for Software Security Testing and Quality Assurance. Artech House, Information Security and Privacy Series, Norwood, MA, 2008.
[4] LCA: The Trinity fuzz tester. From http://lwn.net/Articles/536173/
[5] C. Mulliner, and C. Miller. Fuzzing the Phone in your Phone. Black Hat, USA, July 2009.