DRFuzzer: Detector of Android APP inter-component vulnerability

Hongji song [a] and Hua Zhang [b]
Beijing University of Posts and Telecommunications, Beijing 100876, China.
[a]hongji606@gmail.com, [b]zhanghua_288@bupt.com

Abstract. In recent years, the Android system developed rapidly. It has become the leader of the smart phone operating system. The research of Android security mechanisms is significant. If an Android application could not protect its private components well in the process of inter-application communication, there would exist exposed component vulnerabilities. The current vulnerability detection methods cannot identify such vulnerabilities accurately. To solve this problem, we propose a new method combines Fuzzing with custom Android OS and design a vulnerability mining tool named DRFuzzer. Experimental results show that the tool can effective and comprehensive discover vulnerabilities in communication between components, and provide possibility for solving some deep problems.

Keywords: Fuzzing; custom Android OS; vulnerability; inter-component.

1. Introduction

Due to the low threshold for Android application development, developers do not pay enough attention to mobile security, leading to more and more malicious applications [1, 2]. Not only Android users face unprecedented security risks, the Android developers also faces great challenges.

The risk of components communication includes two cases. One case is that a developer exposes components those not need be exported, and the condition is common that will be found in this paper through a large number of experiments. The second is that there is no good control verification- related permissions and interaction data for shared components, so there will still be at risk. All these situations may cause application crashes, data leaking, and even lead to the extreme cases such as the Android system collapse.

In this paper, we propose a new method based on customized Android OS and Fuzzing technology [4]. The basic idea is to use a custom Android OS to obtain accurate information about the intent of communication components. Then structure accurate components communication Intent according to obtained information, and test every exposed component through the Fuzzing technology. In this paper, we implement a detection tools named DRFuzzer according to the method. Finally we verify the tool can effectively detect the risks associated with the app components and the components vulnerabilities of inter-component communication by testing more than 30 applications.

2. Related Work

Current methods for detecting Android application components communicate vulnerability can be divided into static and dynamic two categories. ComDroid [1] is a represent of static method which use Dedexer to reverse the apps and use static analysis methods to analyze the possible vulnerabilities in inter-component communication.

Static analysis methods can analyze potential security risk, but they can’t give the results when the apps are under attack. And currently apps would do a lot of reinforcement and confusion, decompile became more and more difficult. More important, many inter-component vulnerabilities only show in a particular context. So the static methods have many significant limitations.

Dynamic methods mainly use Fuzzing technology. Intent Fuzzer [5] is one of the representatives, but it cannot test Activities automatically. Even more critical is that it can only send blank messages, no log output and log analysis. JarJarBinks [6] is the improvement of Intent Fuzzer. It can construct random data for Fuzzing test. But it only can construct data using Android recommended Extras keys. Beyond that, the testing process is not fully automated, requires manual intervention.

3. DRFuzzer Design

DRFuzzer overall architecture is shown in Fig. 1, including one PC End and at least one Android device with one Android End. PC End includes Socket Server module, APK management module, task scheduling module, log analysis module, Extras information extraction module, vulnerability analysis module and vulnerability database modules. Android End include Socket Client module, component information management module, data structure module and Fuzzing test module.

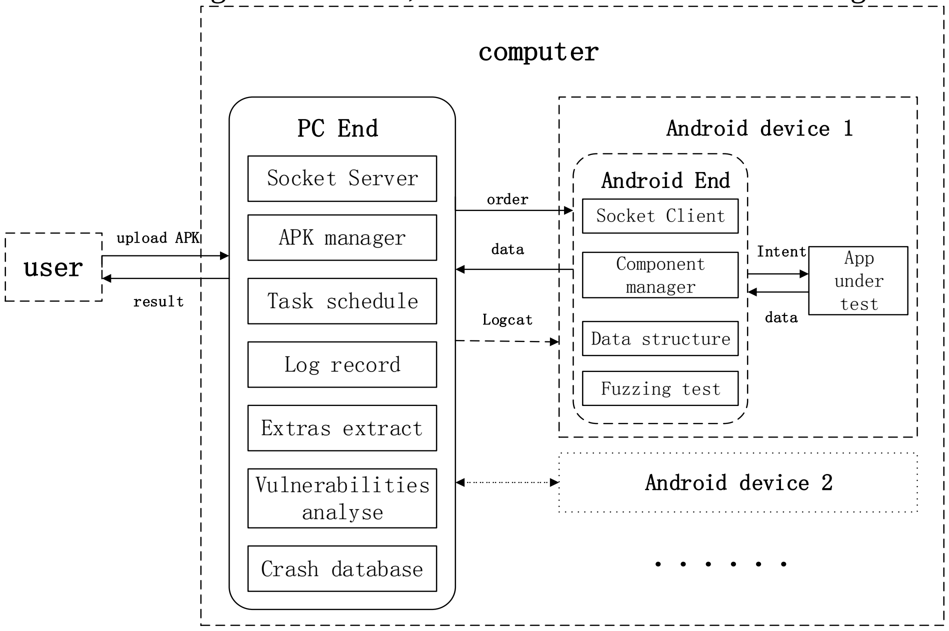

Fig. 1 Overall system architecture diagram

System data flow diagram is shown in Fig. 2. First of all, we custom Android system to output the extras information to the log; then construct intent data according to Android recommended extras keys and test every exposed component; then extract real extras from test log; finally, construct data according to the real extras to test all exposed components and record the test log.

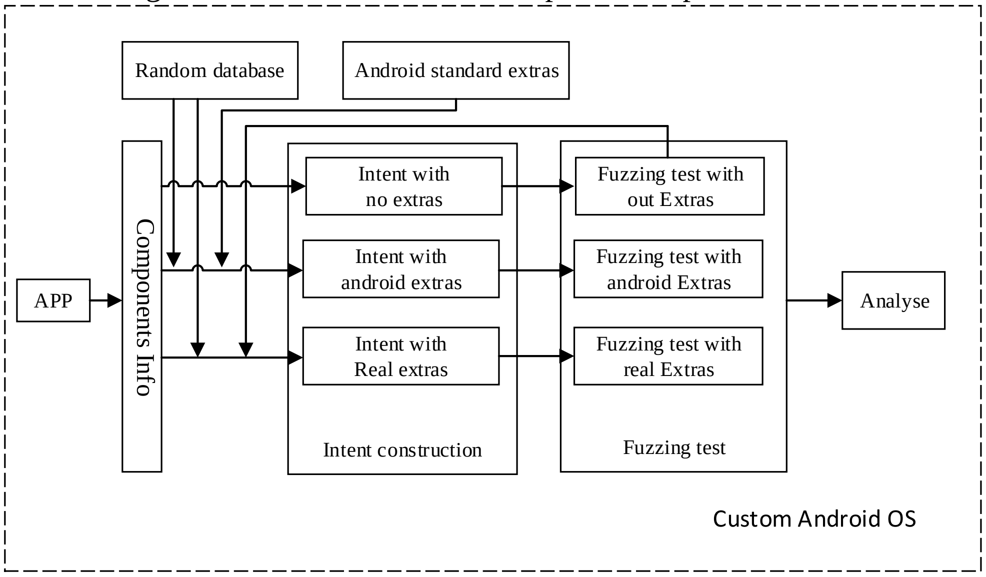

Fig. 2 System testing data flow diagram

3.1 Custom Android OS

The core of the system customization is mainly implemented by modifying the framework layer function associated with the Intent. The main purpose is to output Extras to the system log with a fixed format. Modification of Intent related functions can divided into two cases. The first, obtain the extras sent to Intent, the second is to obtain extras received from intent. The following mainly shows the first case, including three aspects.

I. Set Log format

Log format is shown in Fig. 3.

```
“&$&package_name&$&put(get)&$&type&$&key&$&value&$&”
"&$&" as the delimiter,
"package_name" means that the package name,
"put (get)" represents the Extras information Extras information about the current
component transmitted or acquired,
"key" indicates Extras bond, "value" Extras represents the value of information.
```

Fig. 3 Log format

II. Bind package name

All applications will take effect by modifying the Framework layer Intent related functions. The method of obtaining package name is implemented by modifying getIntent and getPackageName in Activity class and some other method in Intent class.

III. Modify intent correlation function

According to the format defined in I, output extras to the log by modifying some functions. Mainly related functions include getBooleanExtra, getByteExtra, getIntExtra and other functions.

3.2 Exposed Component Detection

Exposure component refers to the component that can be called directly by other apps. There may occur denial of service, components hijacking, data pollution and other vulnerability. There are two cases of the components can be determined be exposed components. One is the exported attribute is true and anther is the component contain intent-filter attribute.

3.3 Fuzzing Design

Fuzzing test includes three different processes, namely empty Fuzzing test, official Fuzzing test and real Fuzzing test. Empty Fuzzing test refers Fuzzing test not carry extras data; official Fuzzing test refers Fuzzing test based on Android official recommended extras; real Fuzzing is based on the real keys of intent extras.

3.3.1 Intent construction

We can construct intent for exposed components according to action, category and data obtained in Section 3.2. But the key is the keys and values construction of extras. Keys represent name of the extras data. Values represent values of the extras data. This paper constructs a random database used to store the values of data. When testing randomly select values from the database. The database values example shown in Table 1.

Table 1 Data example

Data Type	Data Content
int	0,1000, 1844674407, -999997203,
String	"1881266","好的", " ", "m=Uv|I$@ \"
ArrayList	"188112176",0.1f, "aksdgejph","9.9","
...	...

According to the above components and extras information, we can construct test intent as shown in Fig. 4.

```
ComponentName componentName = new
ComponentName( packageName,exposedComponent);
Intent intent = new Intent();
intent.setComponent(componentName);
intent.putExtra(key, value);
intent.setAction(action);
intent.addCategory(category);
```

Fig. 4 Intent construct code

3.3.2 Data Fuzzing

In this section, we use the intent constructed in the section 3.3.1 to Fuzzing test activities, services and broadcast receivers. For example, we use start Activity For Result to complete Fuzzing test with two advantages. One is that we can close the activity after test. The other is that we can record the data return from the activity being test. These data may contain privacy data, leading to data leakage.

4. Results

In this paper, in order to verify test results of DRFuzzer, we particular choice top seven apps of Peas app store as target apps. For each app, three tests are carried out as follows: empty Fuzzing test, official Fuzzing test and real Fuzzing test. In the following, results will show in three aspects: exposed components, three test comparison and crash type comparison.

4.1 Exposed Components

Table 2 shows exposed components including exported activities, exported services, exported receivers of top seven apps in Pea app store. As it can be seen from the results, the component exposure is a common phenomenon. Exists exposed components may lead to denial of service, components hijacking, data leakage and so on. So developers should pay more attention to the safety of components.

Table 2 Exposed components result

Package Name	Exported Activities	Exported Services	Exported Receivers
com.youku.phone	42	11	16
com.ss.android.article.news	155	15	24
com.thestore.main	10	5	2
com.achievo.vipshop	11	5	9
com.fastclean	2	3	8
com.Qunar	15	2	4
com.snda.wifilocating	36	3	11

4.2 Three Test Comparison

We get the following statistics results through three different Fuzzing tests. According to the results, we can find that the speed of empty Fuzzing test is the fastest, but the effect of it is the worst. The effect of official Fuzzing test is better than empty Fuzzing, but the time it used is too long. The real Fuzzing test is clearly the best with best results and short test time. This is the advantages of data construction using real extras. In this way, the data construction is highly targeted, and the vulnerability detection is highly coverage.

Table 3 Crash statistics of the test

Package Name	Item	Empty Fuzzing	Official Fuzzing	Real Fuzzing
com.youku.phone	crash(number)	3	4	9
	time(minute)	3	115	31
com.ss.android.article.news	crash(number)	0	4	5
	time(minute)	13	340	136
com.thestore.main	crash(number)	3	1	4
	time(minute)	1	10	20
com.achievo.vipshop	crash(number)	0	4	2
	time(minute)	2	18	30
com.fastclean	crash(number)	4	5	7
	time(minute)	0.2	2	1
com.Qunar	crash(number)	3	5	3
	time(minute)	1	8	7
com.snda.wifilocating	crash(number)	2	3	3
	time(minute)	2	30	50
average	crash(number)	2.2	4.1	5.7
	time(minute)	3.2	74.7	32.3

4.3 Crash Type Comparison

According to the general statistics of the three test results, we can obtain various crash proportion shown in Fig. 5. In addition, the system can give every detail of the crashes, helping developers to quickly locate and repair the hole. In this paper, we use dynamic method to test inter-component vulnerabilities, one of the biggest advantages is the ability to trigger vulnerabilities when the application is running, and record the details of vulnerabilities.

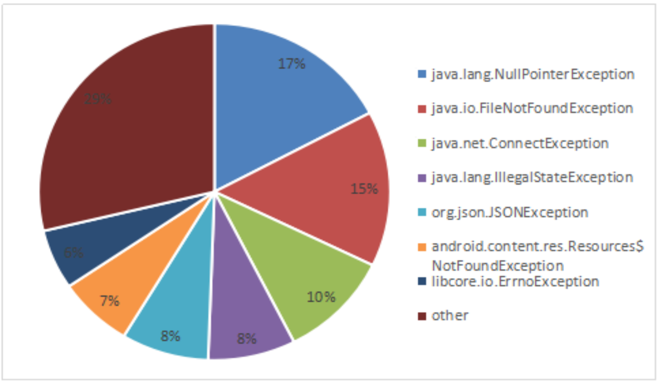

Fig. 5 Type of crash statistics

5. Conclusion

After a great deal of experiments, we find the following disadvantages. For the First, the vulnerability is hard to reproduce. Reproduce vulnerability is a proven difficulty for Fuzzing [7]. In addition, we can improve the data structure of the article in the future. For example, we can use a similar machine learning method to construct test data in order to obtain better test results.

Acknowledgments

This work is supported by NSFC (Grant Nos. 61300181, 61502044), the Fundamental Research Funds for the Central Universities (Grant No. 2015RC23).

References

[1] Chin E, Felt A P, Greenwood K, et al. Analyzing inter-application communication in Android [C]//Proceedings of the 9th International Conference on Mobile Systems, Applications, and Services. ACM, 2011: 239-252.

[2] Enck W, Ongtang M, McDaniel P D. Understanding Android Security [J]. IEEE Security & Privacy, 2009, 7(1): 50-57.

[3] Vennon M A T. Android malware: Spyware in the Android Market [R]. Technical Report, SMobile Systems, 2010.

[4] LIU Q, ZHANG Y. TFTP Vulnerability Exploiting Technique Based on Fuzzing [J]. Computer Engineering, 2007, 20: 051.

[5] nccgroup. Intent Fuzzer [EB/OL]. [2016-10-28] https://www.nccgroup.trust/us/-about-us/re-sources/ intent-fuzzer/

[6] Maji A K, Arshad F A, Bagchi S, et al. An empirical study of the robustness of inter-component communication in Android [C]//IEEE/IFIP International Conference on Dependable Systems and Networks (DSN 2012). IEEE, 2012: 1-12.

[7] Rebert A, Cha S K, Avgerinos T, et al. Optimizing seed selection for Fuzzing [C]//23rd USENIX Security Symposium (USENIX Security 14). 2014: 861-875.