**BIFUZ – Broadcast Intent Fuzzing Framework**

**for Android**

|  |  |
| --- | --- |
| Ionescu Răzvan-Costin  Intel Software Development  Bucharest, Romania  razvan.ionescu@intel.com | Proca Andreea Brînduşa  Intel Software Development  Bucharest, Romania  andreea.brindusa.proca@intel.com |

**Keywords**: Android, Intent, IPC, Fuzzing, Security, BroadcastReceiver

**ABSTRACT**

**Introduction**

We have designed and implemented an Intent Fuzzing Framework for Android.

Intents are one of the most important ways used by applications to communicate. They benefit also for a very high level of trust inside the Android OS, so if they are not validated appropriate, they might create an unwanted damage, or might even compromise a mobile device, from Security perspective.

As a term, fuzzing implies manipulating input data, in order to validate it through the mechanism or device under test. It is usually a black-box, negative testing technique, but we have used it as a grey-box method, also.

Knowing how Intents are built, and which type of parameters they accept and expect, we have been able to craft fuzzed Intents, in order to find Security vulnerabilities in the Inter Process Communication protocol.

**Content**

Our Fuzzing Framework is oriented mainly in two large directions: Intents and Broadcast Receivers.

According to Android developers’ documentation, the primary pieces of information in an Intent are:

* action - The general action to be performed, such as ACTION\_VIEW, ACTION\_MAIN, etc.
* data - The data to operate on, such as a person record in the contacts database, expressed as a Uri.

In addition to these primary attributes, there are a number of secondary attributes that you can also include with an intent:

* category - Gives additional information about the action to execute.
* type - Specifies an explicit type (a MIME type) of the intent data.
* component - Specifies an explicit name of a component class to use for the intent.
* extras - This is a Bundle of any additional information. They are a tuple containing a type, a key and a value.

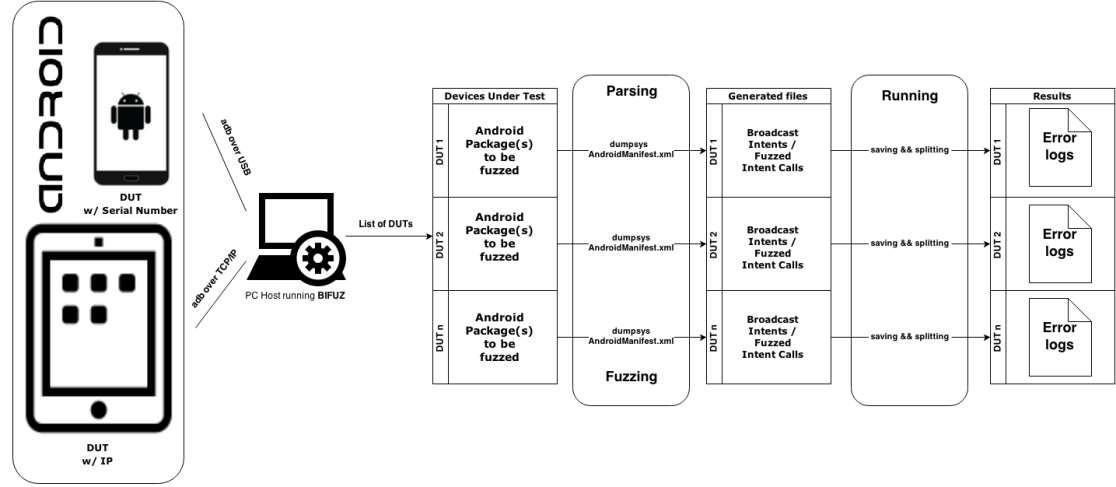


Figure 1 - BIFUZ Overview Architecture

The BIFUZ’s flow is depicted in Figure 1. Basically, any Android Device may be connected to the host running BIFUZ. The Device Under Test (DUT) is scanned and all the important data coming either from dumpsys, or from Android Manifests is saved. Then, the fuzzing process begins and the Intent Calls are generated.

At the users’ choice, those Intents may be crafted for a specific component, or for all the apps installed on the DUT.

**Example**

A Fuzzed Intent might have the following parameters:

|  |  |  |
| --- | --- | --- |
| **action** | | android.intent.action.MAIN |
| **category** | | android.intent.category.DEFAULT |
| **package/component** | | com.google.android.deskclock/com.android.deskclock.DeskClock |
| **data-uri** | | http://7WPIR8PGKYVBF1US7TYTBZ11PCM255YPX5Q1ECIAE6XUUS4R1TLPJ39GX45INCHK5QOOYF2M0TGZ9WF1L2EBQDD5K.org |
| **flags** | | ACTIVITY\_NEW\_TASK ACTIVITY\_NO\_HISTORY |
| **extras** | **type** | string |
| **key** | android.intent.extra.ALARM\_COUNT |
| **value** | TSUDY4FE1ELT16E9PA4G74AV3TOP4W5JO08DVKVRQ96W5RX5D9STOEQ77A5D09UJ8SH3GC3BFA0 |

Table 1- Fuzzed Intent sample

In Table 1 you may see that the data-uri and the value contains the fuzzed data. From negative testing perspective, the parameters which may be fuzzed are the data and the extra values. All the other components are predefined. As a future work we will improve the fuzzed Intents, such that they will send fuzzed files among Apps.

Within Android there are multiple IPC endpoints like Services, Activities, BroadcastReceivers and ContentProviders. They are considered potential attack vectors because they are the core of the system. All the interactions between data sources and pools are controlled using those IPC endpoints. For this matter, BIFUZ project generates also Broadcast Intents. These are generated based on the AndroidManifest file for each scanned package or based on the dumpsys output.

A Broadcast Intent example specific to **com.google.android.deskclock** package looks like this:

adb shell am broadcast -n com.google.android.deskclock/com.android.deskclock.AlarmInitReceiver

Once the Broadcast Intents are sent, the reaction of all the Apps which have registered Broadcast Receivers for that Intent is analysed. All the data logs are saved and the Broadcast Intents which caused the errors are stored.

Same thing happens for the fuzzed intents, also. Error types which may be found consist in „java null pointer exceptions, java class not found exception” and so on.

As an example, an error log might look like below:

--------- beginning of main

F/FUZZING\_BROADCAST(13863): adb -s **SERIAL\_NUMBER** shell am broadcast -n com.android.providers.contacts/com.android.providers.contacts.PackageIntentReceiver

D/AndroidRuntime(13815): Shutting down VM

--------- beginning of crash

E/AndroidRuntime(13815): FATAL EXCEPTION: main

E/AndroidRuntime(13815): Process: android.process.acore, PID: **13815**

E/AndroidRuntime(13815): java.lang.RuntimeException: **Unable to start receiver com.android.providers.contacts.PackageIntentReceiver: java.lang.NullPointerException: Attempt to invoke virtual method 'java.lang.String android.net.Uri.getSchemeSpecificPart()' on a null object reference**

E/AndroidRuntime(13815): at android.app.ActivityThread.handleReceiver(ActivityThread.java:2586)

E/AndroidRuntime(13815): at android.app.ActivityThread.access$1700(ActivityThread.java:144)

E/AndroidRuntime(13815): at android.app.ActivityThread$H.handleMessage(ActivityThread.java:1355)

E/AndroidRuntime(13815): at android.os.Handler.dispatchMessage(Handler.java:102)

E/AndroidRuntime(13815): at android.os.Looper.loop(Looper.java:135)

E/AndroidRuntime(13815): at android.app.ActivityThread.main(ActivityThread.java:5221)

E/AndroidRuntime(13815): at java.lang.reflect.Method.invoke(Native Method)

E/AndroidRuntime(13815): at java.lang.reflect.Method.invoke(Method.java:372)

E/AndroidRuntime(13815): at com.android.internal.os.ZygoteInit$MethodAndArgsCaller.run(ZygoteInit.java:899)

E/AndroidRuntime(13815): at com.android.internal.os.ZygoteInit.main(ZygoteInit.java:694)

E/AndroidRuntime(13815): Caused by: java.lang.NullPointerException: Attempt to invoke virtual method 'java.lang.String android.net.Uri.getSchemeSpecificPart()' on a null object reference

E/AndroidRuntime(13815): at com.android.providers.contacts.PackageIntentReceiver.onReceive(PackageIntentReceiver.java:36)

E/AndroidRuntime(13815): at android.app.ActivityThread.handleReceiver(ActivityThread.java:2579)

E/AndroidRuntime(13815): ... 9 more

I/Process (13815): Sending signal. PID: 13815 SIG: 9

--------- beginning of system

I/ActivityManager( 3051): Process android.process.acore (pid 13815) has died

We are logging each call of the broadcasted Intent in logcat as Fatal (F/) messages in order to be easily observed.

The logging line contains the tag „FUZZING\_BROADCAST” and the Broadcast Intent call:

**adb -s SERIAL\_NUMBER shell am broadcast -n com.android.providers.contacts/com.android.providers.contacts.**

**PackageIntentReceiver**

The **SERIAL\_NUMBER** might be a phone/tablet’s serial number or an IP, depending on the connection type between the Android Device and the Host. The activity manager (**am**) is a tool used within an adb shell to perform various system actions. We are using it to broadcast an Intent.

Every error log will contain in its name the package/broadcast intent which caused the crash and the error type. All the error logs specific to one device will be saved in a particular folder.

**Next steps**

We are planning two further development stages for BIFUZ.

During the first stage we want to improve this tool such that it will display a command menu to the user. Among the menu options, the following will be available:

1. Select testing devices. Even if multiple devices may be connected to the host, the user might choose specific devices to be tested on.

2. Choose between intent fuzzing and broadcast intent fuzzing.

3. Generate a delta report between two fuzzing sessions. The user might be interested in, which (broadcast) intents generate errors specific to Intel devices. This way he may compare 2 fuzzing sessions: the first one run against a Nexus device and the second one run against Intel device(s).

In order to perform the third improvement easier to be implemented, BIFUZ will generate for each fuzzing session one output file containing only the (broadcast) intent calls which have failed.

Before starting a fuzzing session, BIFUZ will provide a rough running time estimation based on the number of generated Intent calls. This way the user may take the decision of starting or not starting the fuzzing campaign.

In the second stage, the tool will be able to automatically generate sign apks based on a fuzzing campaign. Those apks will be basically small exploits, which will crash the „vulnerable” apps. They would be used by testers in order to verify their patches and to generate usefull logs.

**Conclusion**

Many Security bugs could be discovered while sending malformed Intents to an Android App which are not built to handle unexpected data. Those bugs may be further exploited, leading to execution of unwanted code, or to DoS attacks (the system may become unresponsive, or even reboot).

**References**

[1] **Intent Fuzzer: Crafting Intents of Death** – Raimondas Sasnauskas, John Regehr

[2] **A Study of Android Application Security** – William Enck, Damien Octeau, Patrick McDaniel, Swarat Chaudhuri

[3] **A Framework for Automated Security Testing of Android Applications on the Cloud** – Sam Malek, Naeem Esfahani, Thabet Kacem, Riyadh Mahmood, Nariman Mirzaei, Angelos Stavrou

[4] **DroidFuzzer: Fuzzing the Android Apps with Intent-Filter Tag** – Hui Ye, Shaoyin Cheng, Lanbo Zhang, Fan Jiang

[5] **An Empirical Study of the Robustness of Inter-component Communication in** **Android** – Amiya K. Maji, Fahad A. Arshad, Saurabh Bagchi, Jan S. Rellermeyer

[6] **A Fuzzing Framework for the Security Evaluation of NDEF Message Format** – Pasquale Stirparo

[7] **Android Hacker’s Handbook**, Joshua J. Drake, Pau Oliva Fora, Zach Lanier, Collin Mulliner, Stephen A. Ridley, Georg Wicherski

[8] **Android Developer Tools** - http://developer.android.com/