Sicurezza delle Reti e dei Sistemi Software

a.a. 2016-2017

DoApp – Denial of App

https://github.com/lmartire/DoApp

The Team

- Marta Catillo
- Antonio Farina
- Luigi Martire

Supervisor

Prof. Aaron Visaggio

Team Leader

Dott. Ing. Antonio Pirozzi

SUMMARY

1. INTRODUCTION

The ever-growing popularity of Android and its market share for mobile smartphones constantly attract new developers and businesses that target the huge user base. At the same time, sensitive data stored on devices induce attackers to search for new ways to compromise the system and steal data. In particular, the presence of malware apps indicates the existence of a global market for users' private data. A major focus of Android security research is to analyse apps for malicious behaviour [1]. The ubiquitous risk of privacy leaks demands rigorous techniques to detect early both intended and unintended exposure of users' private data, such as location, contacts, and photos, to third parties. Many Android apps export their components to unprivileged apps in the system, and so any app could explicitly trigger the exported component's execution via an intent. Consequently, the processing of intent's structured data at the app's trust boundary represents a security risk that has to be carefully addressed.

1.1 Android Environment

All Android Apps are built by components, the essential blocks of Android [2].

There are four different types of app components; each type serves a special purpose and has a distinct lifecycle that defines how the component is created and destroyed:

- Activities
  An *activity* is the entry point for interacting with the user. It represents a single screen with a user interface. Although the activities work together to form a cohesive user experience, each one is independent of the others. As such, a different app can start any one of these activities, if the app allows it.
- Services
  A *service* is a general-purpose entry point for keeping an app running in the background for all kinds of reasons. It is a component that runs in the background to perform long-running operations or to perform work for remote processes. A service does not provide a user interface. Another component, such as an activity, can start the service and let it run or bind to it in order to interact with it.

- Content providers

A *content provider* manages a shared set of app data that you can store in the file system, in a SQLite database, on the web, or on any other persistent storage location that your app can access. Through the content provider, other apps can query or modify the data if the content provider allows it. For example, the Android system provides a content provider that manages the user's contact information. In general, a content provider is an abstraction on a database, because there are a lot of APIs and support built in them for that common case. However, they have a different core purpose from a system-design perspective. For the system, a content provider is an entry point into an app for publishing named data items, identified by a URI scheme. Thus an app can decide how it wants to map the data it contains to a URI namespace, handing out those URIs to other entities, which can in turn use them to access the data.

- Broadcast receivers
  A *broadcast receiver* is a component that enables the system to deliver events to the app outside of a regular user flow, allowing the app to respond to system-wide broadcast announcements. Since broadcast receivers are another well-defined entry into the app, the system can deliver broadcasts even to apps that are not currently running. Many broadcasts originate from the system—for example, a broadcast announcing that the screen has turned off, the battery is low, or a picture was captured. Apps can also initiate broadcasts—for example, to let other apps know that some data have been downloaded to the device and are available for them to be used. Although broadcast receivers do not display a user interface, they may create a status bar notification to alert the user when a broadcast event occurs.

Every component must be declared in the Android Manifest, the syntetic document that Android Environment uses to describe to whole properties of the app. It is an XML document that declares Components, Permissions, Namespaces and other basic information, coded through XML-tags.

1.2 Role of the intent in IPC

Android components and Apps comunicate with each other using an IPC Mechanism called Intent. An intent is an abstract description of an operation to be performed. It can be used, by suitable Android methods, to launch an activity (startActivity(Intent)), to wake up a Broadcast Receiver (sendBroadcast(Intent)) or to communicate with a background service (startService(Intent)).

There are two forms of intents:

- Explicit Intents: have a specified component (set through the setComponent() method) which provides the class to be run;
- Implicit Intents: have not a specified component; instead, they must include enough information for the system to determine which of the available components is the best to response to that intent. These intents are managed by the Intent Resolution process, which sends them to a component that can handle them.

The Intent Resolution process is based on matching between the Intent's information and all of the <intent-filter> declared in the manifest by the installed application.

Each component that would be called for an Intent management, must declare an <intent-filter> in the manifest using suitable tags, as follows:

- <action> : a string that specifies the generic action to perform;
- <data> : the data type managed by the component; it can be a mimetype or an information about an Uri which addresses to a specific object that must be handled by the component (Uri's information can be a scheme, i.e. 'http', an host, a path, a pathPrefix or a pathPattern).
- <category> : a string containing additional information about the action.

Using these information it is possibile to send, through an explicit intent, malicious content to a component that exports a particular <intent-filter> causing app's anomalies, such as its crash. In fact, many Android apps export their components to unprivileged apps in the system, i.e., any app can explicitly trigger the exported component's execution via an intent. This is an unpleasant limit for the security in the whole Android Environment.

Hence, we want to start from this leak and create an App for testing other Apps by sending them malformed intents addressed to all their components that export intent-filters, declared in their Android Manifest.

2. STATE OF THE ART

2.1 Fuzzers in the Literature

Since Android 2.3 GingerBread, Android developers and researchers started to study the IPC security in the Android components. They used to test apps through the fuzzing approach in order to submit malformed and unexpected inputs.

Some of their tools generate the inputs without paying attention to the Manifest of the target app, using a sort of blind-test approach. One of these apps is the Intent Fuzzer developed by iSEC, which can fuzz test all the components of the target app, although using the only input test data is NULL. This approach weakens the ability to discover bugs.

Another tool, called DroidFuzzer, makes it possible to test only the activities of a target app knowing only the manifest. The DroidFuzzer developers analysed the MIMEtype data of all the activities and generated malformed URIs to send them [3].

Fuzzinozer uses the same approach as DroidFuzzer, but it is a Python module integrated in Drozer Framework. It allows to send fuzzed intents to the target app through the command line. As the previously mentioned tools, it generates a crash report after parsing the android system logcat [4].

Bifuz (Broadcast Intent Fuzzer), another Python tool, can send Broadcast Intents or Fuzzed Intents to the target app (specified by its package name). The error logs are stored in separate folders, based on the device id [5].

There are other solutions at lower level to make fuzz tests. This category of fuzzers is used to test the system applications: a representative is the American Fuzzy Lop (AFL). It is an instrumentation-based fuzzing tool that can be used against binaries that consume multiple file formats as input. The target binaries need to be compiled with afl-gcc, in order to enable the instrumentation of the binaries. This tool is used to test the vulnerabilities of Stagefright Framework, the environment used by Android to manage the multimedia files [6].

2.2 DoApp Objective

The DoApp goal is to create an Android standalone application that makes it possible perform a deep test of a target application. Analysing the manifest of the target application, DoApp is able to stress each component (Activities, Services and BroadcastReceivers) of the application. Through fuzzing and an ad-hoc heuristic, DoApp generates a set of malformed inputs in order to test if the application is crash-proof. Once the test is completed, DoApp produces a report that allows to individuate cause of fault in the target application.

This app is mainly designed for:

- Developers and testers, to lead them during the developing phase. It's, in fact, useful to find crash causes in their apps and to drive them towards security and stability improvement.
- Researchers, to help them in their studies about the Android Framework and its leaks.
- Pentesters, in order to support them in their jobs and tests. This helps them in the deep analysis of security issues, intent vulnerabilities, DOS attacks and data leakage.

The project is born after an evaluation of the existing tools in the same application domain. Our massive idea is studying the weaknesses of these tools and fix them in order to build a complete tool for the analysis and testing of the vulnerabilities in the Android components communication.

3. DOAPP DESIGN

Doapp has been designed in order to operate through the following five main phases:

- Retrieve the info about all installed apps on the system and choice by the user of the target app to be tested;
- Analysis of the choosen app manifest and extraction of all data fields that represent the inputs accepted by the app;
- Generation of ad-hoc intents for all the app components;
- Sending Intents to components;
- LogCat analysis and report about possible app crashes.

The architecture of the tool is presented in Fig. 1.

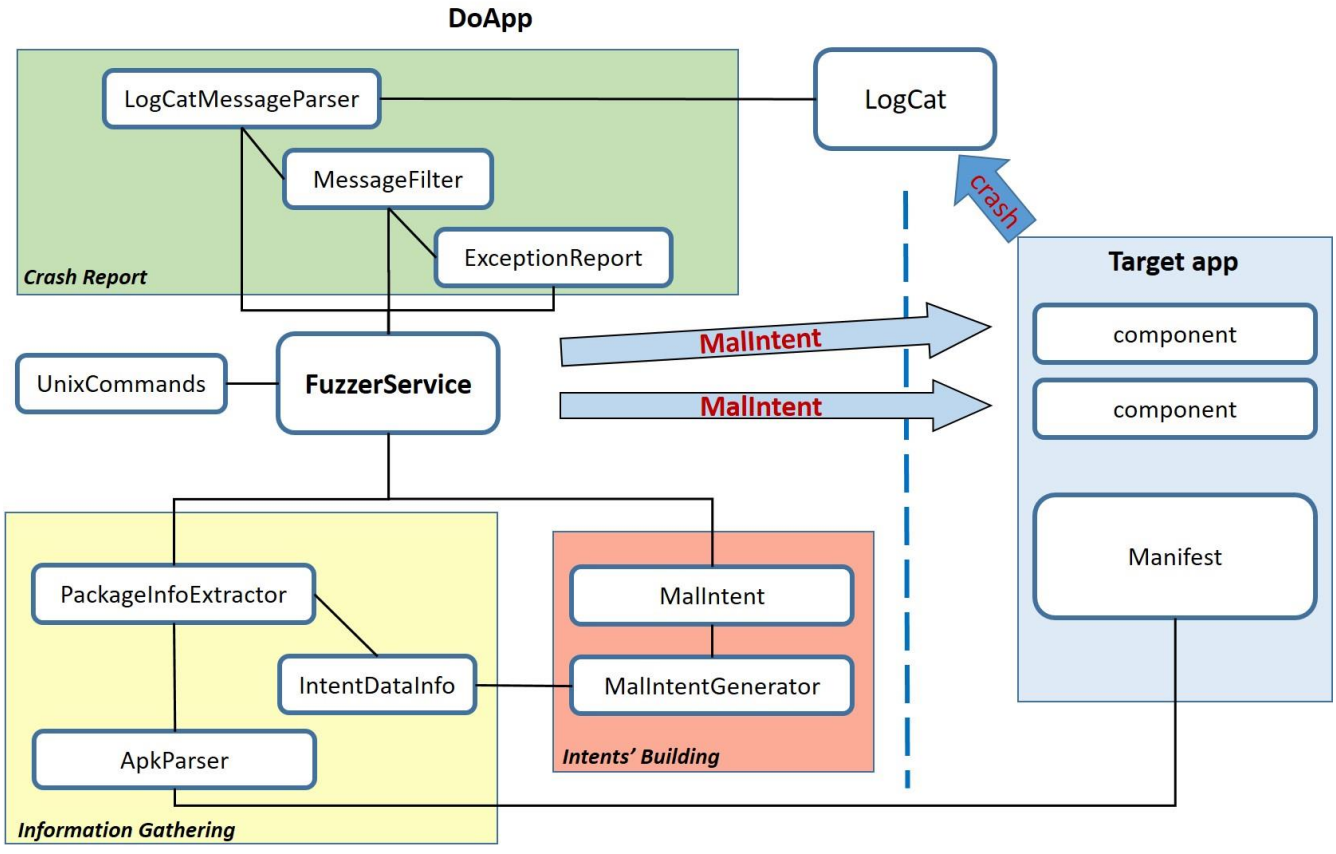

Figure 1: General Architecture of DoApp

3.1 Information Retrieving of Installed Apps

The search of the installed apps relies on the use of the Android Package Manager, using the method (with appropriate flags):

```
context.getPackageManager().getInstalledPackages(
                    PackageManager.GET_DISABLED_COMPONENTS
                  | PackageManager.GET_ACTIVITIES
                  | PackageManager.GET_RECEIVERS
                  | PackageManager.GET_SERVICES
                  );
```

Among all the installed apps we consider only NON-SYSTEM apps, because of the intended use of the app. In fact, the idea behind DoApp is give a good testing tool to the independent Android developers to improve the quality of their product.

```
if((packageInfo.applicationInfo.flags & ApplicationInfo.FLAG_SYSTEM) == 0)
        pkgInfoList.add();
```

All the collected packageInfos are triggered on a ListView, which allows the user to choice which app wants to test.

3.2 Manifest Analysis of the target app

Once the target app has been chosen, its Manifest is read in order to gather all the components' names, which declare intent-filters. In order to do so, we have used an external library (apk-parser-1.0.2)[7] that allows the extraction of the manifest from the app's apk and its parsing.

```
manifestParser = ApkParser.create(context.getPackageManager(), pkgname);
AndroidManifest manifest = manifestParser.getAndroidManifest();
```

The next step is to get all components with the suitable method:

```
components = manifest.getComponents();
```

However, this method returns all components, regardless of the lack of intent-filter and not taking into account that the components could not be exportable (if the attribute of the XML tag is set to false).

For this reason, we need to select only the components that are exportable and have declared intent-filters

```
for (AndroidComponent component : components)
    if (!component.intentFilters.isEmpty())
          // select this component
```

For each intent-filter declared by all components, we need to extract the data fields, with which we will use to construct the IntentDataInfo Object, containing all the info necessary to create the right Intents.

The IntentDataInfo class stores not only the data attributes (scheme, host, port, path, pathPrefix, pathPattern) but also keeps track of the components and the relative intent-filter in which the data field is declared.

```
public IntentDataInfo(IntentFilter.IntentData data, AndroidComponent
        component, String packageName, IntentFilter filter)
```

Since an intent-filter could have more action tags, it is necessary to create a new Object for each data field present in the same intent filter and for each possible action. Therefore, we need to preserve the AndroidComponent, the considered intent-filter and the current data field.

The figure 2 synthetizes how, by reading the Manifest, we generate the Objects IntentDataInfo, which are the starting point for MalIntent analysis and their generation.

```
<manifest package = "com.example.activities">
    <activity
        android:name = "activity1">
        <intent-filter>
            <action android:name = "android.intent.action.VIEW"/>
            <action android:name = "android.intent.action.EDIT"/>
            <data android:mimetype = "image/jpeg"/>
            <data android:mimetype = "image/png"/>
        </intent-filter>
        <intent-filter>
            <action android:name = "android.intent.action.ACTION_WEB_SEARCH"/>
            <data android:mimetype = "text/plain"/>
            <data android:scheme = "http"/>
            <data android:scheme = "https"/>
        </intent-filter>
    </activity>
</manifest>
```

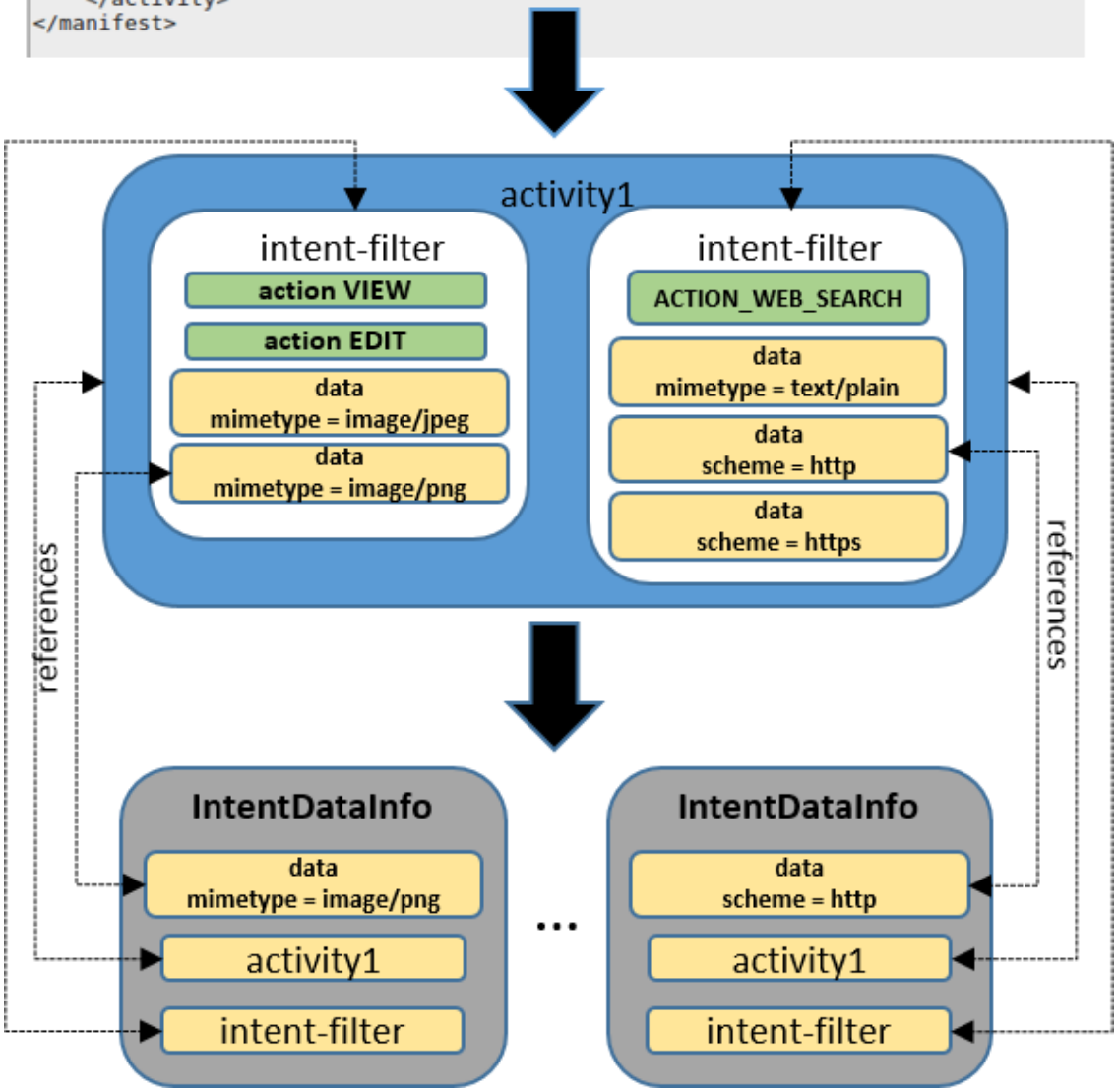

Figure 2: Android System for the intent-filters.

3.3 Test case generation

In this phase it is necessary to create an extension of the Intent class (Android native), MalIntent, which contains also the reference to the target Android component (using the class of the apk-parser-1.0.2 library). The purpose of this step is to generate suitable inputs from the data tags declared in the manifest, in order to try to find a leak in the target app's components.

For this reason, we need to keep track of the architecture of the data field in Android environment. Android developers can specify the data input accepted by their applications in several ways:

- Using only the MIME type

  <data android:mimeType = "text/plain" />

- Using the URI's fields:

  scheme://host:port/path OR pathPrefix OR pathPattern

  - <data android:scheme = "http" />
  - <data android:scheme = "text/plain"
    android:host = "ing.unisannio.it"/>
  - <data android:scheme = "text/plain"
    android:host = "ing.unisannio.it"
    android:port = "9090"/>
  - <data android:scheme = "text/plain"
    android:host = "ing.unisannio.it"
    android:port = "9090"
    android:path = "/didattica/user"
    />
  - <data android:scheme = "text/plain"
    android:host = "ing.unisannio.it"
    android:port = "9090"
    android:pathPrefix = "/didattica/"
    />
  - <data android:scheme = "text/plain"
    android:host = "ing.unisannio.it"
    android:port = "9090"
    android:pathPattern = "/.*/user"
    />

In the above example, we consider all the attributes in a unique data tag, but the real world is a bit different, because they could be spread out in different data tags.

According to these considerations, we have to generate some ad-hoc MalIntents.

In table 1 we synthetize all the possible combinations of the attributes that will be generated for each declared ACTION.

Attributes in <data>	Input generated
mimeType = "text/plain"	▪ NULL input without setting data type ▪ NULL input setting data type ▪ Random text (EXTRA_TEXT) setting data type
mimeType : all the other ones	▪ NULL input without setting data type ▪ NULL input setting data type ▪ Random URI (EXTRA_STREAM) setting data type ▪ Semivalid URI (EXTRA_STREAM) setting data type
scheme	▪ scheme://RANDOM
scheme + host	▪ scheme://host/RANDOM
scheme + host + port	▪ scheme://host:port/RANDOM
scheme + host + port + path	▪ scheme://host:port/path/RANDOM
scheme + host + port + pathPrefix	▪ scheme://host:port/pathPrefix/RANDOM
scheme + host + port + pathPattern	▪ scheme://host:port/MODIFIED_PATH_PATTERN
scheme + host + path	▪ scheme://host/path/RANDOM
scheme + host + pathPrefix	▪ scheme://host/pathPrefix/RANDOM
scheme + host + pathPattern	▪ scheme://host/MODIFIED_PATH_PATTERN

Table 1: Combination of attributes in <data> tag.

3.4 Fuzzer service-Target component interaction

The heart of the system is an Android service that triggers all the generated MalIntents on the target app component.

Before discussing the interaction between service and components, we need to explain the issues regarding root permissions. In fact, the service needs to kill the tested app whenever a MalIntent is sent to the component and it reacts doing something. Killing the app is necessary not only to make possible a new execution of the app, but also to ensure that the test would be independent of the other ones (stateless test). The root permission problem does not allow to execute correctly DoApp on a non-rooted device.

FuzzerService has to clean the logcat before sending MalIntents, to facilitate the search of the app crashes. For each generated MalIntent, the service must send it to the right component (it is necessary to consider the type of the component, if it is an Activity, a Service or a BroadcastReceiver). The next step is to understand whether the application is crashed. The only way to acknowledge this phenomenon is to read the logcat. For this purpose we have to use a parser based on a framework Android's class (LogCatMessageParser) [8]. Whenever the logcat parser finds an exception (after verifying that the exception belongs to the process in question), it is added to a list that contains all the collected exception.

Once all the MalIntents have been sent, the crash dumps are collected in a report file.

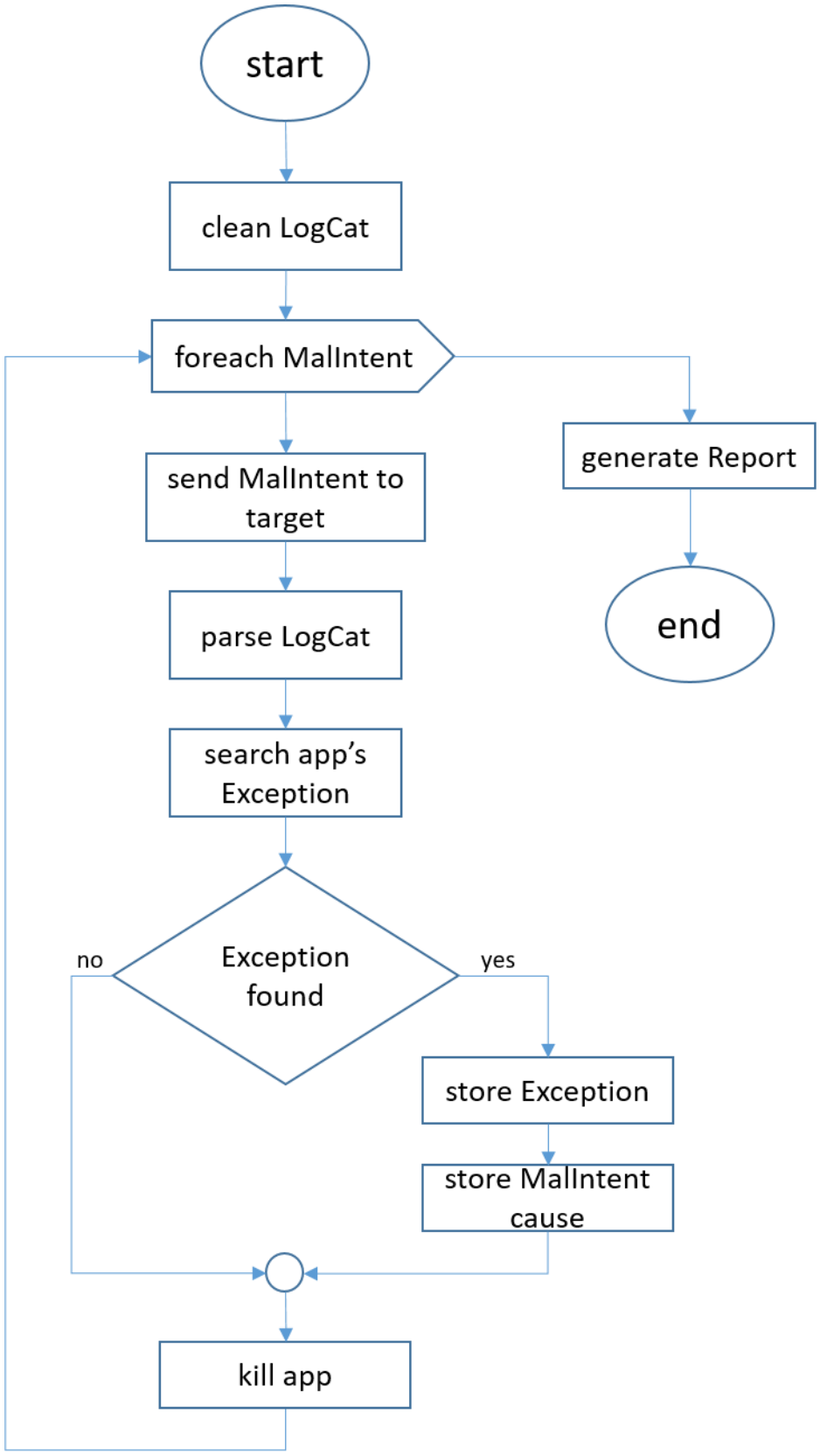

Figure 3: Cycle of execution of FuzzerService.

3.5 Report

The last point to highlight is the identification of app crash using the logcat outputs. As mentioned above, we used a parser that is able to create a LogCatMessage object for each information given by logcat. Using these objects, we can intercept an app FATAL EXCEPTION, but we must verify that it is relative to the current tested process. For this purpose, we try a match between the process' PID extracted from logcat and the process PID returned by Android environment through the Unix command pidof.

Next we have to define a method that allows us to merge the MalIntents that generate the same crash dump. Therefore, we compare the stack traces generated by the logcat, according to the following definition of equality between them: "*Two exception reports are equal when they throw the same exception and the relative stack trace is the same*". In this way, we collect the MalIntents, which generate the same exception report (keeping track of the original MalIntens).

4. EXPERIMENTAL RESULTS

4.1 Operational Conditions

We used Android Studio 2.2.3 for the code devvelopment, because in a multiplatform development environment and it is the official framework by Google. We used the Android API 22 as target API.

The developed app has been used to test other apps and the relative results are illustrated below.

We used an Android emulator to have a sandbox in order to make our tests in security. For this purpose, we used Genymotion, a powerful android emulator, which guarantees root permissions by default (necessary, as already mentioned). Genymotion uses an x86 Android image, so we had some limitations to test the arm native apps.

In the testing phase, we collected 65 apps. These apps are quite relevant as for their international diffusion. We tested all of them in black-box mode. Therefore, applications that require a login step have been neglected.

4.2 Results

From the initial 65 apps, more than one third of them crashed more than once.

We report the results in Table 2, showing also the thrown exception type:

APPNAME	VERSION	Exception Types
Microsoft Onedrive	4.9	NullPointerException
Whatsapp	2.17.32	NullPointerException
Adobe - Acrobat Reader	17.90	NullPointerException
Dropbox	32.2.4	• NullPointerException • IndexOutOfBoundsException
Shazam	6.7.0	NullPointerException
Twitter	6.31.0	NullPointerException
Microsoft Outlook	2.1.138	NullPointerException
Dubsmash	2.21.2	NullPointerException
Tinder	6.7.2	• NullPointerException • RuntimeException
Badoo	4.59.0	NullPointerException
Pinterest	6.7.0	IllegalArgumentException
The Fork	8.5.1	NullPointerException
Box	4.3.615	NullPointerException
vk	4.7.2	ArrayIndexOutOfBoundsException
Wire private messenger	2.26.309	NullPointerException
SoundCloud	2017.01.24	• NullPointerException • IllegalStateExceptipon
Asus Task	2.15.0.20	NullPointerException
Microsoft Word	16.0.7766.4775	NullPointerException
Microsoft Office Lens	16.0.7820.3002	NullPointerException
Microsoft Powerpoint	16.07.7766.4272	NullPointerException
Microsoft Excel	16.0.7766.5022	NullPointerException
Asus File Manager	2.0.0.355	NullPointerException
Signal - private messenger	3.28.1	NullPointerException
Expedia	8.0.1	NullPointerException

Table 2: Test's report

As can be seen from the table, the main problem not considered by the app developers is the check on null data. It is also interesting to consider the (few) problems found relative to the missing check about the index position.

5. CONCLUSIONS AND FUTURE DEVELOPMENT

We can say that DoApp is certainly a useful tool for Android developers to lead them to make their apps safer and more stable. It was a surprise to discover that also apps developed by international corporations show some points of failure. The most recurrent spotted problem is NullPointerException. This means that many developers do not care about checking the null references. Another interesting failure is IndexOutOfBoundException, which could be also dangerous, because it might be used to exploit the injection of malicious payloads (but this is not an object of our study).

DoApp is a starting point for the development of a testing framework useful for the pre-deployment phase. In addition, DoApp represents a knowledge base to get insight on the IPC security problem in Android Framework, using the found results for keeping forward the pentest.

The next steps in the development of DoApp will be:

- To make more efficient and user-friendly DoApp, in order to help also the novice programmers to test their apps.
- To deepen tests against the Broadcast Receiver components, in order to understand the right malicious inputs to make them crash.

6. REFERENCES

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

[2] https://developer.android.com

[3] DroidFuzzer: Fuzzing the Android Apps with Intent-Filter Tag - Hui Ye1, Shaoyin Cheng1, Lanbo Zhang2, Fan Jiang.

[4] Android Intent Fuzzing Module for Drozer - Razvan Ionescu, Stefania Popescu.

[5] BIFUZ Broadcast Intent Fuzzing Framework for Android - Ionescu Răzvan-Costin, Proca Andreea Brînduşa.

[6] Fuzzing Android: a recipe for uncovering vulnerabilities inside system components in Android - Alexandru Blanda.

[7] https://github.com/jaredrummler/APKParser

[8]https://android.googlesource.com/platform/tools/base/+/0c58ef3a3a25e7eb4813825899f2758e035039a5/ddmlib/src/main/java/com/android/ddmlib/logcat