
Automatic Android Malware Analysis

Contents

1	Introduction	2
1.1	Android application fundamentals	2
1.1.1	Application components	2
1.1.2	Intents messages	2
1.2	APK file	3
1.2.1	APK file contents	3
1.3	Dex file	5
1.3.1	Dex file format	5
1.3.2	Multiple Dex files in single APK	5
1.4	Android Application Analysis	5
2	Static Analysis	8
2.1	Androguard	8
2.1.1	Androguard usage example	8
2.2	Information Extracted from an APK using Androguard	12
2.2.1	Example usage	14
3	Dynamic Analysis: CuckooDroid based on Cuckoo sandbox	17
3.1	CuckooDroid architecture	17
3.2	CuckooDroid Installation	18
3.3	CuckooDroid required patching for Android 4.1	19
3.3.1	Android 4.1 Persistent Rooting	19
3.3.2	Bug fixes in Analyzer code	19
3.4	CuckooDroid with Android 5.1	20
3.4.1	Android 5.1 AVD persistent rooting procedure	20
3.4.2	Python 2.7 on Android 5.1	22
3.5	CuckooDroid with Android 7.0	25
3.6	Chapter Conclusion	25
4	Dynamic Analysis: Anti-Emulator Detection	26

1 Introduction

Write introductory paragraph here

1.1 Android application fundamentals

Android application are mostly written in Java. The Android SDK tool compile this code along with any data and resources files into an APK, an Android Package. One APK file contain all contents of an Android app and is the file that Android devices use to install the application [1]. We will discuss the structure of an APK file in section 1.2. In this section we will discuss some basic parts of an Android application.

1.1.1 Application components

The essential building blocks of an Android application are called App components. Each component is an entry point to the application. Each type serves a distinct purpose and has a distinct life-cycle that defines how the component is created and destroyed. The communication between these components (except Content providers) is done using messages called "Intents" (section 1.1.2). It is also important to note that all of these components needs to be listed in AndroidManifest.xml file, for more detailed description of this file please look at section 1.2.1. There are four different types of app components:

- **Activities** An activity is the entry point for interacting with the user. It represents a single screen with a user interface. Each activity is independent from others [1].
- **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[1].
- **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. Because broadcast receivers are another well-defined entry into the app, the system can deliver broadcasts even to apps that aren't currently running. Although broadcast receivers don't display a user interface, they may create a status bar notification to alert the user when a broadcast event occurs. More commonly, though, a broadcast receiver is just a gateway to other components and is intended to do a very minimal amount of work [1].
- **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. As such, any app with the proper permissions can query the content provider, such as `ContactsContract.Data`, to read and write information about a particular person [1].

1.1.2 Intents messages

Three of the four component typesactivities, services, and broadcast receiversare activated by an asynchronous message called an intent. Intents bind individual components to each other at runtime.

You can think of them as the messengers that request an action from other components, whether the component belongs to your app or another [1]. Although intents facilitate communication between components in several ways, there are three fundamental use cases:

- Starting an activity
- Starting a service
- Delivering a broadcast

Readers more interested in this topic are recommend to have a look at [2].

Add ref to section describing the use of am and broadcast to fire activities and intents

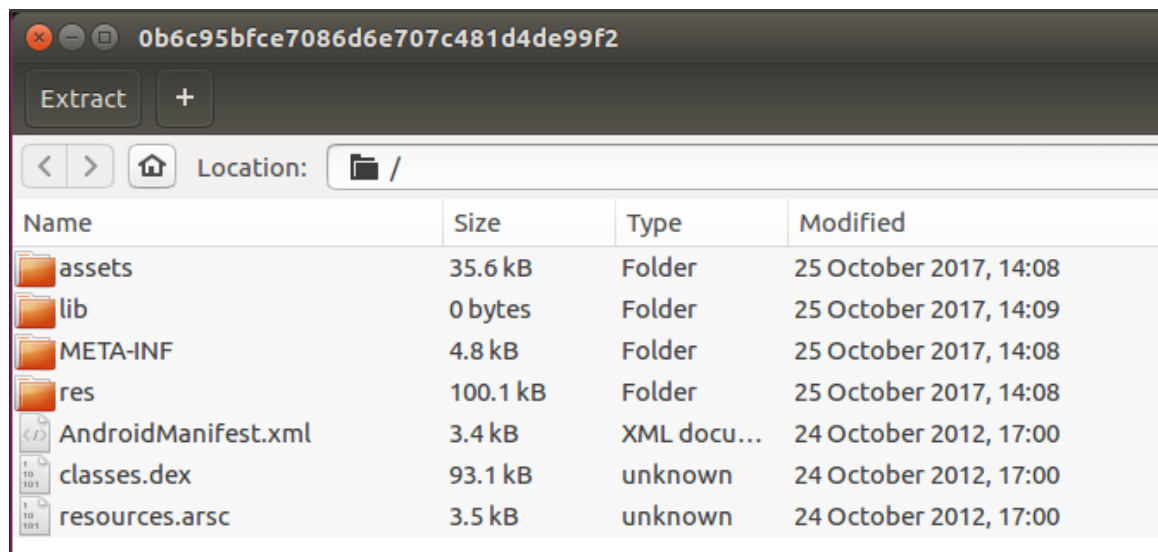
1.2 APK file

Android Application Package(APK) is the file format used for an android application. It contains all the resources required for an application to run on android operating system. Its basically a zip file or a jar file with extension of ".apk"[3].

1.2.1 APK file contents

Normally an apk file contains following files or folders:

Add captions and made the picture available in list of figures



Name	Size	Type	Modified
assets	35.6 kB	Folder	25 October 2017, 14:08
lib	0 bytes	Folder	25 October 2017, 14:09
META-INF	4.8 kB	Folder	25 October 2017, 14:08
res	100.1 kB	Folder	25 October 2017, 14:08
AndroidManifest.xml	3.4 kB	XML docu...	24 October 2012, 17:00
classes.dex	93.1 kB	unknown	24 October 2012, 17:00
resources.arsc	3.5 kB	unknown	24 October 2012, 17:00

Figure 1: Files inside an APK

- **assets/**: It provides a way to include arbitrary files like text, xml, fonts, music and video in your application and allow you to access your data raw/untouched. AssetManager is used to read this data[4]. Due to raw access sometimes this directory contains executable payloads and

dynamically loaded code. One interesting usage is storing Dex files in it to avoid its reverse engineering. [5]

- **lib/**: This directory is for natively compiled code. This directory contains a subdirectory for each platform type, like armeabi, armeabi-v7a, arm64-v8a, x86, x86_64, and mips [3]. This code is run directly on CPU and have access to android API using Java Native Interface(JNI). Natively compiled code is more suitable for CPU intensive jobs because of less overhead and good performance of programming language like c/c++. Most of the android static analysis tools work on Java level- that is, they process either the decompiled Java source code or Dalvik Byte Code[6]. This rises several interesting scenarios in which malware authors can avoid detection, can redistributing benign applications with malicious injections or completely modifying behavior of an application. Readers interested in this topic are encouraged to have a look at [6]. Android NDK can be used to compile native code for android.

compile hello world in c for android in appendix

- **META-INF/**: This directory contains the following three files:
 1. **MANIFEST.MF**: Its a text file and contains a list and base64 encoded SHA-1 hashes of all files included in the APK.
 2. **CERT.SF**: This file again contain a list of all files but this time with the base64 encoded SHA-1 hashes of the corresponding lines in the MANIFEST.MF file. It also contain based64 encoded SHA-1 hash of MANIFEST.MF file.
 3. **CERT.RSA**: It contains developers public signature, used for validation of upgrades. Its basically signed content of CERT.SF file along with public key to validate the contents.
- **res/**: This directory contain resource which are not compiled into "resources.arsc" (see below) [3]. These resources can be accessed from inside the application code using resource ID. All resource IDs are defined in "R" class of the project. Application developers can specify alternate resources to support specific device configurations e.g, alternative drawable resources for different screen sizes, alternative strings for different languages etc.
- **AndroidManifest.xml**: Every application must have an AndroidManifest.xml file. This file provide essential information about the application like entry points, package name, components, permissions, minimum level of Android API, libraries, intents etc. For static analysis purposes a lot of information can be extracted from this file.
- **classes.dex**: This is the most important file insude an apk. It contains classes compiled in the DEX file format which can be understood by the Dalvik/ART virtual machine [3]. In the next section we will describe this file in more details.
- **resources.arsc**: This file contain compiled resources. This file contains the XML content from all configurations of the res/values/ folder. The packaging tool extracts this XML content, compiles it to binary form, and archives the content. This content includes language strings and styles, as well as paths to content that is not included directly in the resources.arsc file, such as layout files and images [3]. These resources can also be accessed using the "R" class.

1.3 Dex file

Dex file is the heart of an android application. First Java source code of an application is compiled to Java byte code (".class" extension). Then this Java byte code is compiled to Dalvik Byte Code or Dalvik Executable(DEX) using Dex-compiler or dexter tool. This code is then executed on Dalvik Virtual Machine (deprectated) or in case of Android Runtime (ART), this code is compiled at install time to the native code.

1.3.1 Dex file format

In this section we will briefly discuss the file format for dex files. For more in depth and up to date specifications readers are encouraged to have a look at android official documentation on dex format [7]. A more graphical representation of dex file is shown in Figure 2. In 2 we had shown that how one element of proto_ids points to different locations in a dex file with the help of lines.

structure of Dex file

1.3.2 Multiple Dex files in single APK

Android app (APK) files contain executable bytecode files in the form of Dalvik Executable (DEX) files, which contain the compiled code used to run your app. The Dalvik Executable specification limits the total number of methods that can be referenced within a single DEX file to 65,536 including Android framework methods, library methods, and methods in your own code. This limit is referred to as the '64K reference limit'

Cite <https://developer.android.com/studio/build/multidex.html>

. [9]

Versions of the platform prior to Android 5.0 (API level 21) use the Dalvik runtime for executing app code. By default, Dalvik limits apps to a single classes.dex bytecode file per APK. Multidex support library can be used to workaround this limitation. Android 5.0 (API level 21) and higher uses a runtime called ART which natively supports loading multiple DEX files from APK files

Cite <https://developer.android.com/studio/build/multidex.html>

. Because of this support its not uncommon these days to come across APKs that contain multiple dex files e.g, Facebook, instagram etc. [9]

1.4 Android Application Analysis

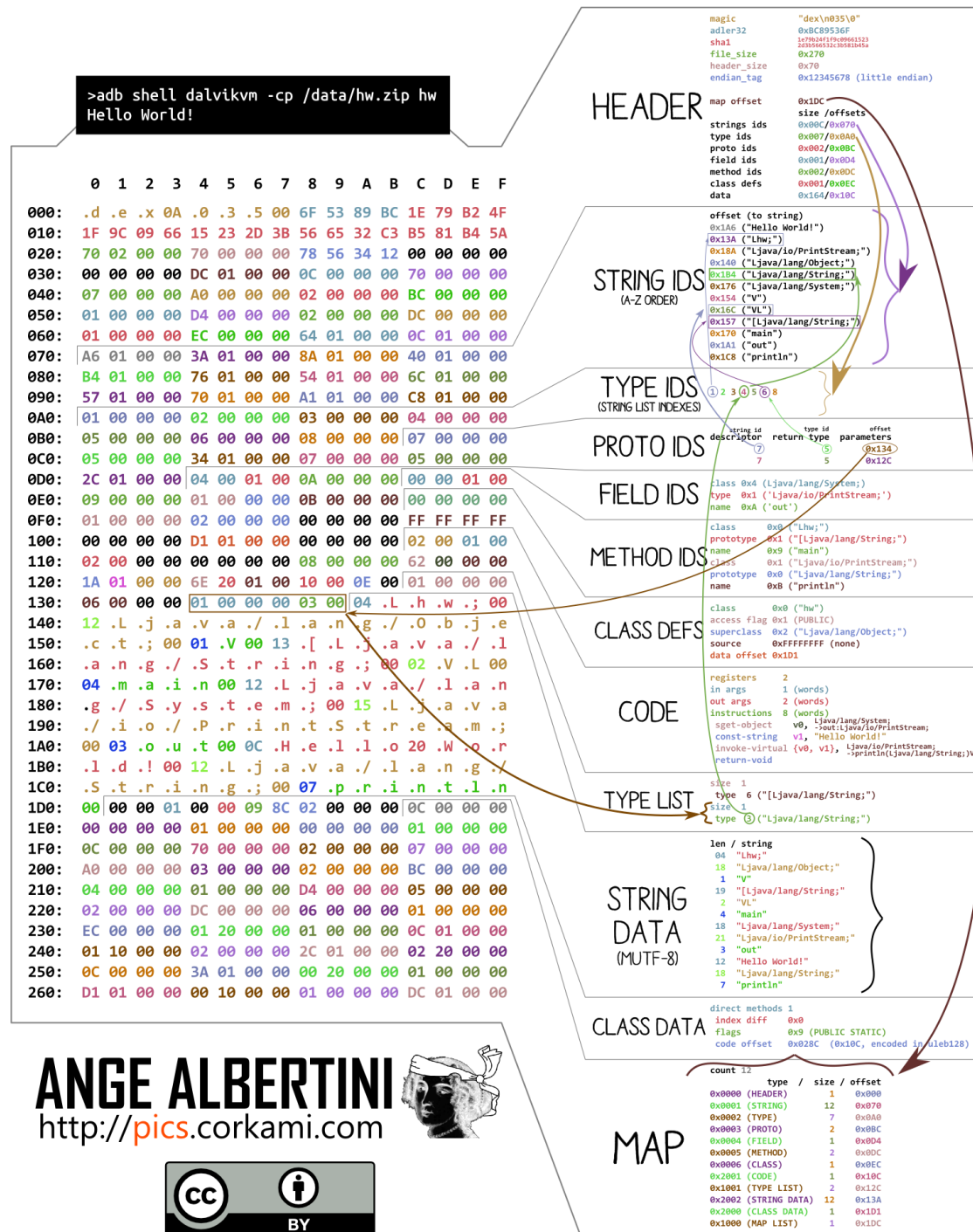
TODO: Write introduction section after the significant part of report is done and the structure is more clear

To be done later, In this chapter we include the problem statement, See fh kiel project report structure for missing parts.

ANGE ALBERTINI
http://pics.corkami.com



Figure 2: Dex file format [8]



change table to multipage table

Add information about Janus vulnerability to make the reading interesting, also about the latest news about its exploitation in the wild

Name	Format	Description
header	header_item	The header contain information about how the dex file is organized, sizes of different sections inside the dex file, size of dex file, size of data section, version of dex format etc.
string_ids	list of string_id_items	Its a list of string identifiers. These are identifiers for all the strings used by this file e.g, class names, method names, constant objects. Each item points to a location in data section (see below) where the original string is stored.
type_ids	list of type_id_items	This list contain type identifiers for all types (classes, arrays or primitive types) referred to by this file, whether defined in the file or not. The actual identifier string is stored in data section. Items in this list points to items in string_ids list and which in turn points to type identifier string stored in data section.
proto_ids	list of proto_id_items	Its a method prototype identifier list. Each item of this list contain three elements: <ul style="list-style-type: none">• shorty_idx Points to string_id_item of shorty descriptor for this prototype• return_type_id Specify return type by pointing to corresponding type_id_item• parameter_off Offset from start of file to the list of parameter types for this prototype. It must point to location in data section. The data there should be in "type_list" format. This value would be zero in case no parameters.
field_ids	list of field_id_items	These are identifiers for all fields referred to by this file, whether defined in the file or not.
method_ids	list of method_id_items	These are identifiers for all methods referred to by this file, whether defined in the file or not.
class_defs	list of class_def_items	The classes must be ordered such that a given class's superclass and implemented interfaces appear in the list earlier than the referring class. Furthermore, it is invalid for a definition for the same-named class to appear more than once in the list.
call_site_ids	list of call_site_id_items	These are identifiers for all call sites referred to by this file, whether defined in the file or not.
method_handles	list of method_handle_items	A list of all method handles referred to by this file, whether defined in the file or not. This list is not sorted and may contain duplicates which will logically correspond to different method handle instances.
data	unsigned bytes	Containing all the support data for the tables listed above. Different items have different alignment requirements, and padding bytes are inserted before each item if necessary to achieve proper alignment.
link_data	unsigned bytes	The format of the data in this section is left

2 Static Analysis

Add info from tutorials on <http://www.fasteque.com/android-reverse-engineering-101-part-5/>

There are several static analysis tools available for APKs, each one having its own strengths and weaknesses. We will be using androguard because it can be easily automated to analyze large amount of applications and store the result in a database. Other popular static analysis tools include Smali/Backsmali, apktool, JADX-Dex to java decompiler, Dex2Jar, Jd-gui etc.

2.1 Androguard

Introduce androguard

MalloDroid, extension of androguard <https://www.dfn-cert.de/dokumente/workshop/2013/FolienSmith.pdf>

Androguard used in http://lilicoding.github.io/SA3Repo/papers/2013_guo2013characterizing.pdf

Androguard is an open source tool written in python for analyzing android applications. Its been used in several tools including Virustotal and Cuckoodroid among others. It can process APK files, dex files or odex files. It can disassemble Dex/Odex files to smali code and can decompile Dex/Odex to Java code. Being python based and open source it allows for automating most the analysis process and one can make desired improvements. The analysis is done on need-to basis which means we only use our CPU for the stuff that we need. It has the also the option to be used in interactive mode using androlyze.py script. We will give more information about how to use androgaurd in 2.1.1.

Since its an Open source project and as is the case with most open source projects, the available documentation is not that much detailed and . For easier understanding and use, we can generalize the classes androgaurd contain into two groups as shown in table 1.

2.1.1 Androguard usage example

As we mentioned above, androgaurd can be used in two way, its a good idea to start with interactive CLI mode of androguard just to get some idea of what it is capable of. You can use the below command in your terminal to start the interactive shell. It will import all the required androguard packages and you can start analyzing APKs right away. Depending on how you installed androguard, you may need to execute androlyze.py or just androlyze.

```
androlyze -s
```

This will open a python interactive shell and you can now specify the APK file as show below:

```
apk, dvm, dx = AnalyzeAPK("path/to/app.apk")
```

Fix the position of table

Classes for Parsing	Classes for Analysis
<ul style="list-style-type: none"> • APK Used for accessing all elements inside an APK, including information from Manifest.xml like permissions, activities etc. • DalvikVMFormat It parses the dex file and gives access classes, methods, strings etc. defined inside the dex file. • ClassDefItem Class for interacting with class information inside the dex file. • EncodedMethod Class for interacting with method information inside the dex file. • Instuction Class for interacting with instructions, it contains mnem, opcodes etc. Its a base class and a androguard derive a class for each instruction format from this class. 	<ul style="list-style-type: none"> • Analysis Its the main analysis class and contain instances of all other analysis classes discussed below. create_xref() method needs to be called after an instance of this class is created to populate all defined fields in this class. • ClassAnalysis This class contain analysis data of a class like cross references and external methods etc. • MethodAnalysis Contain analysis information of a method like the basic blocks it is composed of etc. • DvmBasicBlock Represents a simple basic block of a method. It contains information about that basic block like its parents, children etc.

Table 1: Some classes of androguard and their description

The analyzeAPK method returned three objects. These classes are also mentioned in 1. The first returned class is an object of APK class. This one can be used to get information about the APK most of which are extracted from manifest.xml file. Below is the example on how to extract some information:

```
user@workstation:~$ androlyze.py -s
Androguard version 3.1.0
In [1]: pwd
Out[1]: '/home/user'

In [2]: apk, dvm, dx = AnalyzeAPK("/home/user/workspace/apps/evasion.apk")

In [3]: apk.get_activities()
Out[3]: [b'com.ouz.evasion.EvasionMainActivity']

In [4]: apk.get_permissions()
Out[4]: []

In [5]: apk.get_libraries()
Out[5]: []

In [6]: apk.get_package()
Out[6]: 'com.ouz.evasion'

In [7]: apk.get_max_sdk_version()

In [8]: apk.get_min_sdk_version()
Out[8]: '14'

In [9]: apk.get_files()
Out[9]:
['res/layout/activity_evasion_main.xml',
'res/menu/evasion_main.xml',
'AndroidManifest.xml',
'resources.arsc',
'res/drawable-hdpi/ic_launcher.png',
'res/drawable-mdpi/ic_launcher.png',
'res/drawable-xhdpi/ic_launcher.png',
'res/drawable-xxhdpi/ic_launcher.png',
'classes.dex',
'META-INF/MANIFEST.MF',
'META-INF/CERT.SF',
'META-INF/CERT.RSA']
```

There are a lot of methods that can be called on the APK object, to see them just type "apk." in the CLI and then press tab. The auto-complete feature will show you all the possible methods you can call. For more details you can have a look at the source code or the documentation for this class.

The dvm class represent DalvikVMFormat class, which basically parse the dex file contained inside the apk and gives us information about the classes, methods, Basic Blocks and instructions.

```
In [12]: type(dvm)
Out[12]: androguard.core.bytecodes.dvm.DalvikVMFormat
```

In below script we will demonstrate how to get name of some of the methods using this dvm object:

```
In [13]: methods = dvm.get_methods()[10:15]
```

```
In [14]: for method in methods:
...:     print(method.get_name())
...:
getSettingsActivityName
<clinit>
<init>
capabilityToString
feedbackTypeToString
```

The auto-complete feature, documentation or source code can be used in similar way as mentioned above to discover more features more methods or fields of this objects. The dx object is an instance of Analysis class which also provide access to analysis parts classes and methods like cross references, string analysis as shown below:

```
In [23]: for key in dx.get_strings_analysis():
...:     print(key)
...:
Unable to create files subdir

mCommitted=
mDeliveredData=
Resolving type
No fake drag in progress. Call beginFakeDrag first.
android.app.Notification$Action
mAvailIndices:
; boundsInScreen:
RemoteInput
Can't change tag of fragment
mAvailBackStackIndices:
Couldn't fetch mRecreateDisplayList field; dimming will be slow.
Operations:
is not a sliding drawer
NULL
mRetaining=
command cannot be null or empty
mAccessibilityDelegate
mProcessingChange=
op #
Callback may not be null
onCreateView: id=0x
.
.
.
```

```
.
=>
:
Unable to instantiate fragment
onLoadComplete:
android:fragment:
icon
.
, tag
Can not perform this action after onSaveInstanceState
mediaSession is not a valid MediaSession object
mMenuVisible=
tag must not be null or empty
```

Below script demonstrates how to further examine the results provided by Analysis class object dx.

```
In [30]: string_analysis = list(dx.get_strings_analysis().values())[10]

In [31]: string_analysis.get_orig_value
Out[31]: <bound method StringAnalysis.get_orig_value of <androguard.core.
        ↪ analysis.analysis.StringAnalysis object at 0x7fd6247ff7b8>>

In [32]: string_analysis.get_orig_value()
Out[32]: "Can't change tag of fragment "

In [33]: string_analysis.get_xref_from()
Out[33]:
{(<androguard.core.analysis.analysis.ClassAnalysis at 0x7fd62588d320>,
<androguard.core.bytecodes.dvm.EncodedMethod at 0x7fd626197b00>)}

In [35]: string_analysis.get_value()
Out[35]: "Can't change tag of fragment "
```

We encourage reader to play with the androguard interactive CLI. After you have some idea of what can you do, then you proceed to write some python scripts and automate some of the tasks. In section 2.2 we will talk about the information that we extracted from APK files using androgaurd.

2.2 Information Extracted from an APK using Androguard

Before getting into details, we would like to mention that the information we extract will be stored in a database and similarity search would be performed on this data to figure out its relations other malware or goodware samples. The more information we have about an APK, more relations we can figure out.

Now coming towards, the extracted information, we divide it into different groups as shown in figure 3.

- **APK information** This contain general information about the APK like Permissions, Package name, Libraries, Certificates, components (activities, services, receivers) and information about

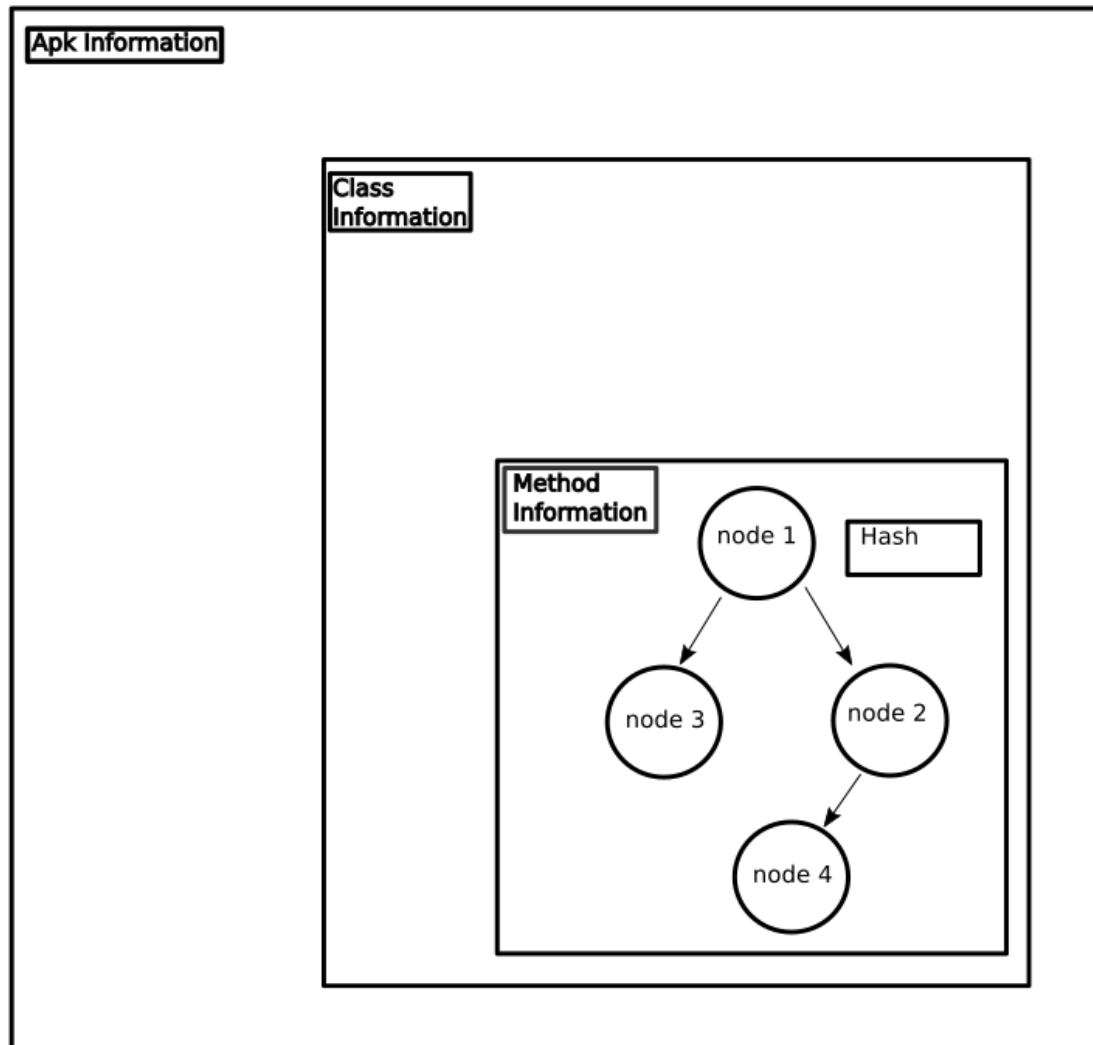


Figure 3: Groups of information extracted using APK

all classes contained in this APK. We also wrote some code to extract strings from layout files and other binary xml files contained in the APK. Most of Malwares bitcoin wallet addresses were hard coded in the those files.

- **Class information** It represent a single element of the class_dictionary contained in APK information. It contains information like fields in the class, its access flags, name, superclass name, number of internal methods, inherited methods etc. It also contain information about methods which are part of this class in the form of a list.
- **Method information** A we said above, class_information contain a list of methods_info. Each element of that list contain information about a method such as method name, class name, address, method descriptor, length of method, its cross references, shorty descriptor, Java decompiled source code for that method, Control flow graph of that method and the calculated hash for this method.
- **Control Flow Graph** Control flow graph contain edges and nodes. Nodes are basic blocks and are basically a list of instructions.
- **Canonical Certificate** In order to compare two control flow graphs, we need to find the Canonical form of this Control flow graph. Fore more details about the topic readers are encouraged to search the Internet for "Graph canonization" and also have a look at the webpage of "bliss" [10]. We used pybliss python library [10] [11] to get the canonical certificate of this graph. Before computing this certificate, the smali instructions are normalized according to a specific criteria so that compilation specific changes are ignored like offsets etc.

2.2.1 Example usage

Just to make the usage of this extracted data more clear. In this example we process some of the sonicspy samples and tried to figure out how much code they share. We analyzed 16 samples

Add hashes, probably redo the whole analysis

and the result is shown in figure 4.

Line 23 in figure prints the result, in this dictionary keys represent frequency or number of times a method is been reused. Value represent numbers i.e, number of methods reused a specific number of times. From this analysis we can see that a large portion of code is common between these samples but a large part of this code is not malicious. Most of it standard android API methods and non-malicious general purpose methods like wrappers etc. It would be very interesting to identify and separate API code from this chunk. It can be topic for further research to identify common non-malicious pieces of code to make analysis easier.

In the code snippet, "jp" is an object of a class JasonParser which we wrote just to verify information extracted from APKs. In line 24 we get the hash of a specific method and in line 25 and 26, we print its Java source code.

TODO: Do androguard basic usage examples

Discuss the changes we made including normalization, canonical hasing for similarity search

Discuss the info we are extracting from apks for platform

```

In [23]: jp.freq_methods_nb
Out[23]:
{1: 2,
 2: 1882,
 3: 533,
 4: 225,
 5: 3350,
 7: 42,
 9: 14,
11: 831,
13: 34,
14: 124,
16: 910}

In [24]: hash_ = jp.freq_methods[16][3]
In [25]: code = jp.get_method_code(hash_)
In [26]: print(code)

    public boolean dispatchKeyEvent(android.view.KeyEvent p2)
    {
        if ((!super.dispatchKeyEvent(p2)) && (!this.executeKeyEvent(p2))) {
            int v0_2 = 0;
        } else {
            v0_2 = 1;
        }
        return v0_2;
    }

In [27]: █

```

Figure 4: Reused methods in sonicspy variants

TODO: Do androguard comparison apks to see how many functions has added and how many removed, make a table out of it

TODO: Find reused code section in sonicspy or bankbots or lokibot

Usage of androguard for extracting features for AI/ML, prepare for talk in AIOLI-FFM group

Ask lukas for some results from platform

Improvements in androguard

3 Dynamic Analysis: CuckooDroid based on Cuckoo sandbox

CuckooDroid is an extension of Cuckoo Sandbox the Open Source software for automating analysis of suspicious files. CuckooDroid brings to cuckoo the capabilities of execution and analysis of android application [12]. For more information about the cuckoo sandbox, readers are encouraged to visit the cuckoo sandbox website [13]. Currently, CuckooDroid only supports Android 4.1.

CuckooDroid can be downloaded from the CuckooDroid github repository [14] by following the guidelines provided there. For more step-by-step installation guide, readers are encouraged to have a look at CuckooDroid documentation [12]. Because of changes in android emulator (goldfish) and android SDK the CuckooDroid documentation are not precisely accurate and some deviations are required from it in order to make the CuckooDroid work. By following the CuckooDroid documentation with a few changes discussed later in this chapter, it should be fairly easy for reader to configure his CuckooDroid setup.

3.1 CuckooDroid architecture

In this section we will describe the architecture of CuckooDroid. There are main two parts a "Host" and a "Guest". Below is an excerpt from the CuckooDroid documentation [12]:

"This documentation refers to Host as the underlying operating systems on which you are running Cuckoo (generally being a GNU/Linux distribution) and to Guest as the Windows virtual machine used to run the isolated analysis."

We will be configuring CuckooDroid with Android Emulator (Goldfish) and figure 5 shows the architecture CuckooDroid with Android Emulator. As it can be seen in the figure 5 that there are two main parts, Cuckoo Sandbox and Android Emulator.

Cuckoo Sandbox is responsible for managing the android emulator and generating report at the end of analysis. Android Emulator executes the application, gather some information from it and reports it back to Cuckoo Sandbox. Below is the description of some of the main parts shown in figure 5

- **Python Agent** Executed on AVD and is responsible for receiving APK file, Analysis code, configuration and executing analysis. It also provides constant status updates to Host.
- **Python Analyzer** Android analyzer component that is sent to the guest machine at the beginning of the analysis. This is the main part that executes application, send dropped files back to host, send screenshots back to host, interact with the application if required. It is also responsible for ending the analysis and sending back some log files back to host. It has modular structure and new modules can be added very easily.
- **Xposed Framework** A framework for modules that can change the behavior of the system and apps without affecting any APKs. The version in use only work up to Android 4.1.2 (API 16). In CuckooDroid two modules are used with this Framework:
 1. **Droidmon:** Dalvik API Call monitoring module, it hooks up API calls and prints it into logcat, analysis code takes it from logcat and store it in a log file which is sent back to host at the end of analysis.

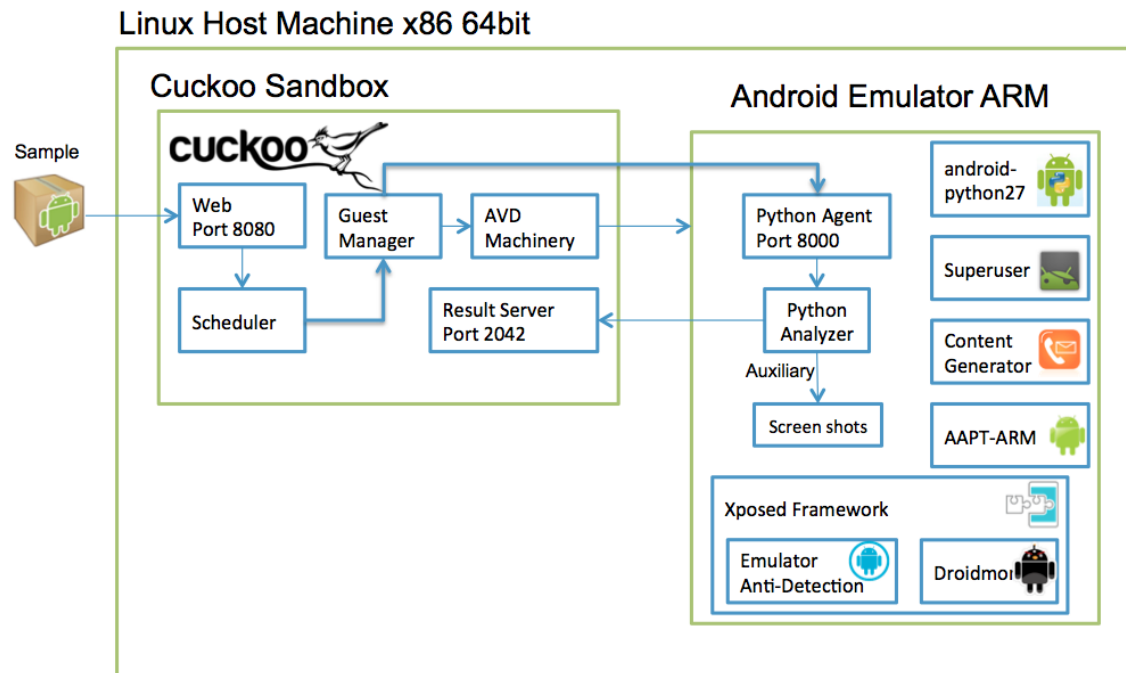


Figure 5: CuckooDroid architecture with AVD

2. **Emulator Anti-Detection:** Implements some know Emulator Anti-detection techniques (For more details on this topic see Chapter 4)

In the rest of this chapter we will walk through configuring CuckooDroid and some of the changes which are necessary to make it work with latest Android Emulator and latest Android versions.

3.2 CuckooDroid Installation

Installation of CuckooDroid is pretty straight forward and can be easily done by following the Easy integration script available on the CuckooDroid github repository [14]. For the configuration and requirements follow the instruction in the CuckooDroid documentations [12]. In the next two paragraphs we will give a bit more intuitive description of how CuckooDroid works which will help in understanding some of steps in installation and configuration step. Please note that there are some changes which are required to make CuckooDroid work and those are discussed in section 3.3.

The basic components of CuckooDroid can be seen in figure 5. There are two basic parts of CuckooDroid host and guest. We run the cuckoo sandbox on host which handles the emulator. In order to do analysis our guest needs to be a rooted AVD with Xposed Framework [15] two of its modules Droidmon and Emulator Anti-Detection. We also need Python 2.7 to run the python agent and analyzer code on our guest.

Once we run the Cuckoo sandbox and submit a sample to it, it makes a new copy of our specified AVD and launches it. Once the machine boots, the cuckoo sandbox runs the python agent using Android Debugger (adb) and sends the APK file, configuration file, and analysis code to it and orders it to start executing the analysis code. The analysis code uses the apk module to install the sample and executes it. The DroidMon module of xposed Framework monitors the specified function calls using hooks and prints it on the logcat. Screenshot modules takes screenshots of the android screen and sends it to host. Once the analysis is completed, the analyzer sends all the required files back to host and terminates. Then host kills the emulator and starts making a report out of these files.

3.3 CuckooDroid required patching for Android 4.1

In this section we will talk about some of the changes which were required to make CuckooDroid work with Android 4.1. There are two main steps, one is the persistent root problem and the second one is some bug fixes in python analyzer code.

3.3.1 Android 4.1 Persistent Rooting

Now coming towards the first fix which was that the root wasn't staying persistent across reboots. It was the result of changes in Android emulator. More details and solutions about this problem can be found in issue 4 of CuckooDroid github repository [16]. The solution can be found in the same issue, according to which we needed to copy the system.img file from the android SDK directory to the AVD directory. The following command can be used to do that, make sure you replace the paths according to the location of your android SDK and AVD.

```
cp ~/Android/Sdk/system-images/android-16/default/armeabi-v7a/system.img ~/.  
↪ android/avd/aosx.avd/system-qemu.img
```

3.3.2 Bug fixes in Analyzer code

The second fix is in the python code that is executed on Android. The CuckooDroid analyzer and agent code heavily relies on Subprocess Management library [17] (specifically subprocess.Popen() method). During debugging we noticed that Most of the calls to subprocess.Popen() method never returns resulting in analysis critical time out. Upon further research we found an issue relating to a similar problem in python 3.7 [18], which seems to be the case here. The subprocess.Popen function uses /bin/sh in Unix environments. Android is detected as a Unix environment, but has moved that executable to /system/bin/sh. This can be worked around by adding a parameter "executable='/system/bin/sh'" to all the subprocess.Popen calls. One related problem was that in some parts of the analyzer, like in adb.py os.popen was used to execute shell commands. This function was also failing to return so we have to replace it with subprocess.Popen.

The next limitation that we faced was that CuckooDroid only supports Android upto 4.1, which is quite older and in order to analyze applications that are targeting higher version of androids we need to find a way to run CuckooDroid on higher version of Androids. In the next sections we will describe the steps and results of our attempts to upgrade to Android 5.1 and Android 7.0.

3.4 CuckooDroid with Android 5.1

Just like with Android 4.1, here we again faced the persistent root problem. Another problem was that we couldn't use the python binaries for android that comes with CuckooDroid because from this version Android make it a requirement for binaries to support PIE mode. In the following subsections we will describe in more detail about how did we solved these problems. Here we used the x86 image just to save time during testing. This rooting procedure works on arm image too but that speed of arm image is very slow and was wasting a lot of time during the development and debugging.

3.4.1 Android 5.1 AVD persistent rooting procedure

The rooting procedure for Android 5.1 is a bit different than that of Android 4.1 in the sense that we needed updated version of tools like SuperSU, XposedFramework etc. Below are the required steps to root Android 5.1 AVD and keep the root persistent.

1. First copy the system.img to your avd directory and rename it as system-qemu.img: For example for Nexus One API 22 x86:

```
cp ~/Android/Sdk/system-images/android-22/default/x86/system.img ~/.
↪ android/avd/Nexus_One_API_22_x86.avd/system-qemu.img
```

2. Then start the AVD Machine with

```
emulator @Nexus_One_API_22_x86 -verbose -writable-system
```

3. Run the script located in utils/android_emulator_creator and named create_guest_avd_x86/arm.5.1+.sh. Below is a snippet from one of the scripts which was made with the help of stackexchange answer by a user with the name xavier_fakerat [19]

```
#!/usr/bin/env bash
# Inspired by: https://android.stackexchange.com/questions/171442/root-
↪ android-virtual-device-with-android-7-1-1/176447
#this script is meant for easy creation on an analysis machine for
↪ android emulator avd

#Path to the local installation of the adb - android debug bridge
↪ utility.
#ADB=/home/wra/Andriod/Sdk/platform-tools/adb
ADB=$(which adb)
if [ ! -f $ADB ]
then
echo -e "\n-Error: adb path is not valid."
exit
fi
echo -e "\n-adb has been found."
# First we root the emulator using Superuser
echo -e "\n-Pushing /system/xbin/su binary"
$ADB root
```

```

$ADB remount
# one of these is correct, we do both just to be on safe side
$ADB push nougat/SuperSU-v2.82-201705271822/x86/su.pie /system/xbin/su
$ADB shell chmod 06755 /system/xbin/su
$ADB push nougat/SuperSU-v2.82-201705271822/x86/su.pie /system/bin/su
$ADB shell chmod 06755 /system/bin/su
echo -e "\n-Installing application Superuser"
$ADB install nougat/eu.chainfire.supersu_2.82.apk
# Install Xposed Application
echo -e "\n-Installing Xposed Application"
$ADB install nougat/XposedInstaller_3.1.4.apk
# Install Droidmon Application
echo -e "\n-Installing Droidmon Application"
$ADB install hooking/Droidmon.apk
# Install Termux application
$ADB install nougat/com.termux.apk
# Install Anti Emulator Detection Application
echo -e "\n-Installing Anti Emulator Detection Application"
$ADB install hooking/EmulatorAntiDetect.apk
$ADB push anti-vm/fake-build.prop /data/local/tmp/
$ADB push anti-vm/fake-cpuinfo /data/local/tmp/
$ADB push anti-vm/fake-drivers /data/local/tmp/
# Install Content Generator
echo -e "\n-Installing Content Generator"
$ADB install apps/ImportContacts.apk
# Install Cuckoo Agent and Python for ARM
echo -e "\n-Pushing Agent executing scripts"
$ADB push ../../agent/android/python_agent/. /data/local/
$ADB shell chmod 06755 /data/local/aapt
$ADB shell chmod 06755 /data/local/get_terminal.sh
$ADB shell chmod 06755 /data/local/run_agent.sh
echo -e "\n***** Run the following commands and you are good to go*****\n"
↪ n"
echo "adb shell"
echo "cd /system/bin/"
echo "su root"
echo "su --install"
echo "su --daemon&"
echo "setenforce 0"
echo -e "\n"
echo -e "*NOTE"
echo " To make sure everything is working fine after the rooting is done
↪ , do the following:"
echo " Open android emulator and run termux. Then type python2 to make
↪ sure to that python is installed"
echo " Then open adb shell Run the get_terminal.sh script and then type
↪ python2 to verify that you "
echo " that you can run python with root privileges. After that run
↪ run_agent.sh and verify that agent is running."

```

4. Run the SuperSu app and if the device is rooted, it will ask for update, update the app, don't reboot yet.

5. After update, open Xposedframework app and click install there, it will ask for administrator permission, click grant.
6. Enable Droidmon and anti-emulator modules.
7. Soft reboot the machine, (It will be somewhere in xposed app)
8. After reboot verify root by opening SuperSU app and it doesn't give you an error and show xposedframework app, the device is rooted.
9. Close the machine, (The emulator 27.0.1 will save the snapshot of the machine and will load it at next start)
10. start the machine with -writable-system option (without it the loaded machine is not rooted)and it will automatically load the previously save snapshot, that means it will be up and running in no time. (verify root by opening SuperSu app), then close the machine.
11. start the machine again with adding one more option to above command, that is -no-snapshot, this time the machine is boot properly and verify the root by running SuperSU app.

3.4.2 Python 2.7 on Android 5.1

In order to do the analysis we need to have python 2.7 in our Android 5.1 guest. CuckooDroid comes with python 2.7 binaries which are compiled for Android 4.1, which doesn't work with Android 5.1 because with the release of Android 5.1 Android requires all dynamically linked executables to support PIE (position-independent executables). So we either need python compiled with PIE support. We can either compile it ourselves or we can use pre-compiled python with PIE support. We opted for the pre-compiled solution because of simplicity. Below is an example on how to compile a C program with PIE support using Android Native Development Kit (NDK).

Compiling C code for Android 5.1+

1. Download and extract NDK (My NDK directory is /android-ndk-r16)
2. Create a new standalone toolchain using the script /android-ndk-r16/build/tools/make_standalone_toolchain.py


```
python ~/android-ndk-r16/build/tools/make_standalone_toolchain.py --arch
  ↪ x86 --api 22 --install-dir ~/android-ndk-r16/toolchains/x86-22
```
3. Write your c program, compile it and link it separately with required flags for PIE.


```
cd ~/android-ndk-r16/toolchains/x86-22/bin
touch main.c
cat > main.c

#include<stdio.h>
int main()
{
printf("Hello world!\n");
return 0;
}

./i686-linux-android-gcc -c main.c -fPIE
./i686-linux-android-gcc -lm main.o -pie
```

4. Start emulator, copy file to it, make it executable and run it

```
emulator @Nexus_One_API_22_x86 -writable-system
adb push a.out /data/local
adb shell chmod 06755 /data/local/a.o
adb shell /data/local/a.o
WARNING: linker: /data/local/a.out: unused DT entry: type 0x6ffffffe arg
↪ 0x32c
WARNING: linker: /data/local/a.out: unused DT entry: type 0x6fffffff arg
↪ 0x1
Hello world!
```

Pre-Compiled python for Android 5.1+

Now, as we said above we used the pre-compiled python 2.7 that can be installed inside the Termux application [20]. Termux is an Android terminal emulator and Linux environment application that provides several linux command line tools an android making it really powerful. The app doesn't require root permissions which means it can't access or modify files outside of application. We needed a workaround so that we can run termux shell as root and have access to all system. Below are steps required to achieve that:

1. Download Termux apk from google playstore or apkmirror and install it to rooted emulator
2. Open the installed app and type "pkg search python"
3. Install python2 using pkg install python2 (take help from step2 to determine exact command)
4. Now type python2 and you will have a working python. But this only works inside the app and can't be run as sudo/root.
5. To be able to run python as root from adb, we will need to download the script <https://github.com/cswl/tsu> (A fork of this which support optional command is here <https://github.com/kiney/tsu>)
6. Push this script to your emulator and run it with -e option.
7. It will give you a bash shell (whatever) which support most of linux commands and others can be installed as well. (You need to have rooted device to run this script)
8. Now we need to add the code inside the script so that instead of launching a shell it run our agent.py script.
9. To do that go to line 100 which is:

```
exec "$s" --preserve-environment -c "LD_LIBRARY_PATH=$LD_LIBRARY_PATH
↪ $ROOT_SHELL"
```

Step 10: Edit the line as follows:

```
exec "$s" --preserve-environment -c "LD_LIBRARY_PATH=$LD_LIBRARY_PATH
↪ python2 /data/local/agent.py"
```

10. Now run this script with "-e" and you will see that our agent.py script is been fired.
11. Now we need to change part of the code which fires up agent.sh

12. Due to some unknown reasons we ran into some permissions problems like not being able to read hooks.json file. In logcat droidmon was complaining about not having permission to read hooks.json. This can be fixed by setting the global read flag on hooks.json after it is copied in cuckoodroid/analyzer/android/analyzer.py Below is the snippet of code:

```
def prepare(self):
    # Initialize logging.
    init_logging()

    # Parse the analysis configuration file generated by the agent.
    self.config = Config(cfg="analysis.conf")

    # We update the target according to its category. If it's a file
    ↪ , then
    # we store the path.
    if self.config.category == "file":
        self.target = os.path.join("/data/local/tmp", str(self.config.
            ↪ file_name))
        shutil.copyfile("config/hooks.json", "/data/local/tmp/hooks.json
            ↪ ")
        os.chmod("/data/local/tmp/hooks.json", 0754) # Give set global
            ↪ read flag on hooks.json
    # If it's a URL, well.. we store the URL.
    else:
        self.target = self.config.target
```

13. Another problem was in the screenshot module, it wasn't able to read to screenshots taken and this can be solved by using the same method as above i.e, setting global read at the time the file is created and also changing the location of file in this case sdcard was readonly and this function wasn't able to save the image there. Below is the modified code of take_screenshot() method in adb file:

```
def take_screenshot(filename):
    proc1= subprocess.Popen("/system/bin/screencap -p /data/local/
        ↪ screenshots/"+filename, # "/data/local/screenshots/" Fix
        ↪ for /sdcard read only
    stdout=subprocess.PIPE,
    shell=True,
    stderr=subprocess.PIPE,
    executable='/system/bin/sh')
    stdout, stderr = proc1.communicate()

    if len(stdout)>0:
        log.info("take_screenshot stdout: %s", stdout)
    if len(stderr)>0:
        log.info("take_screenshot stderr: %s", stderr)

    os.chmod("/data/local/screenshots/"+filename, 0744)
    return "/data/local/screenshots/"+filename
```

3.5 CuckooDroid with Android 7.0

We didn't have much success with Android 7.0. We couldn't pass the persistent rooting problem and the machine wouldn't boot after the rooting procedure as described for Android 5.1.

3.6 Chapter Conclusion

After the work described in this chapter we came to some conclusions which will be described in this section.

Emulator Anti Detection

Emulator anti detection

Although during our dynamic analysis we didn't analyzed enough malicious samples make an estimate the usage of emulation detection techniques in malicious samples, but a little search on the Internet revealed a several techniques that were very easy to implement and or integrate into an application.

Frida good because can adapt quickly to new threats and good support for higher version of android

Support for higher version android

Slow emulator performance

Slow android emulator

4 Dynamic Analysis: Anti-Emulator Detection

Common methods employed for emulator detection, some literature

Good and bad uses of anti-emulator detection

Testing results of cuckoodroid against common emulator detection methods

Adding some new anti-emulator detection features to cuckoodroid

result of analysis before and after

References

- [1] “Android application fundamentals.”
- [2] “Intents.”
- [3] “Reduce the apk size.”
- [4] “Using android assets.”
- [5] K. Lim, Y. Jeong, S.-j. Cho, M. Park, and S. Han, “An android application protection scheme against dynamic reverse engineering attacks,” *JoWUA*, vol. 7, no. 3, pp. 40–52, 2016.
- [6] V. M. Afonso, P. L. de Geus, A. Bianchi, Y. Fratantonio, C. Kruegel, G. Vigna, A. Doupé, and M. Polino, “Going native: Using a large-scale analysis of android apps to create a practical native-code sandboxing policy,” in *NDSS*, 2016.
- [7] “Dalvik executable format.”
- [8] “Dalvik executable picture by ange albertini.”
- [9] “Enable multidex for apps with over 64k methods.”
- [10] “bliss: A tool for computing automorphism groups and canonical labelings of graphs.”
- [11] T. Junttila and P. Kaski, “Engineering an efficient canonical labeling tool for large and sparse graphs,” in *2007 Proceedings of the Ninth Workshop on Algorithm Engineering and Experiments (ALENEX)*, pp. 135–149, SIAM, 2007.
- [12] “Cuckoo-droid documentation.”
- [13] “Cuckoo website.”
- [14] “Cuckoo-droid github.”
- [15] “Xposed module repository.”
- [16] “Android avd root access fail.”
- [17] “Subprocess management python 2.7.”
- [18] “Subprocess.popen issue 16255.”
- [19] “Android emulator 7.1 rooting procedure.”
- [20] “Termux website.”