

A study of vulnerabilities on Android systems



Author: Vicente Javier Mozos Pérez

Directors: Dr. David de Andrés Martínez DISCA-UPV, Spain

Dr. Jesús Frigonal López LAAS-CNRS, France

Dr. Juan Carlos Ruiz García DISCA-UPV, Spain

Master's thesis

Máster Universitario en Ingeniería de computadores

Departamento de Informática de Sistemas y Computadores

September 2013

Table of Contents

1 Introduction	8
2 Android in a nutshell.....	10
2.1 Android layers.....	10
2.2 Application Components.....	11
2.3 Communication between components.....	13
2.4 Android, a vulnerable system.....	14
2.5 Summary.....	19
3 Classification of security tools for Android	20
3.1 Dynamic tools.....	21
3.1.1 Androguard	21
3.1.2 APK-tool.....	22
3.1.3 APK Multi-tool	24
3.1.4 Mercury	26
3.1.5 ASEF-Android Security Evaluation Framework	28
3.2 Static tools.....	29
3.2.1 ComDroid	29
3.2.2 StowAway.....	30
3.2.3 Intent Fuzzer.....	31
3.2.4 Intent Sniffer.....	32
3.3 Tools summary.....	34
3.4 Why we choose Mercury	35
3.5 Summary.....	36
4 Methodology supported by Mercury.....	37
4.1 Common attack injection methodology.....	37
4.2 Attacks supported by Mercury	37
4.2.1 Malformed Intents.....	37
4.2.1.1 Surface attack.....	38
4.2.1.2 Injections.....	38
4.2.1.3 Observable output.....	39
4.2.2 Broadcasts.....	40
4.2.2.1 Attack surface.....	40
4.2.2.2 Injections	40
4.2.2.3 Observable output.....	41

4.2.3 SQL Injection.....	42
4.2.3.1 Attack surface.....	42
4.2.3.2 Injections.....	44
4.2.3.3 Observable output.....	46
4.2.4 Command Execution.....	47
4.2.4.1 Attack surface.....	47
4.2.4.2 Injections.....	47
4.2.4.3 Observable output.....	48
4.3 Summary.....	48
5 Case study.....	49
5.1 Target Devices.....	50
5.2 Work load.....	50
5.3 Experiments configuration.....	51
5.3.1 Relevant parameters.....	51
5.4 Results.....	52
5.4.1 Malformed Intents.....	53
5.4.2 Broadcasts.....	54
5.4.3 SQL Injection.....	56
5.4.4 Command execution.....	58
5.4.5 Curious data.....	59
6 Tips and tricks	61
6.1 Storing data.....	61
6.1.1 Using internal storage.....	62
6.1.2 Using external storage.....	62
6.1.3 Using content providers.....	62
6.2 Using Permissions.....	63
6.2.1 Requesting Permissions.....	63
6.2.2 Creating Permissions.....	64
6.3 Summary.....	65
7 Conclusion and further work.....	66
8 ITACA Research Day.....	67
9 References	69
10 Annex.....	73
10.1 Automating the experiments: Masquerade.....	73

10.1.1 Masquerade.....	73
10.1.1.1 Activities.....	74
10.1.1.2 Broadcasts.....	75
10.1.1.3 Package Menu.....	76
10.1.1.4 Providers.....	77
10.1.1.5 Services.....	78
10.1.1.6 Exploits menu	79
10.1.1.7 Information menu.....	79
10.1.1.8 Scanner menu.....	80
10.1.1.9 Generate general info	81
10.1.1.10 SQL Injection.....	82
11 Intents classification.....	83

List of figures

Figure 1: Android architecture layers.....	10
Figure 2: Android SO distribution.....	15
Figure 3: Android versions.....	16
Figure 4: Android threats.....	17
Figure 5: Android threats.....	18
Figure 6: APK-tool menu.....	23
Figure 7: APK-Multi tool menu.....	25
Figure 8: Mercury Menu.....	27
Figure 9: Drozen logo.....	27
Figure 10: ASEF diagram.....	29
Figure 11: Analysis example.....	31
Figure 12:: Intent fuzzer overview.....	32
Figure 13: Overview	33
Figure 14: Exported activities.....	38
Figure 15: Implicit and explicit.....	38
Figure 16: Force close.....	39
Figure 17: Providers search.....	40
Figure 18: Broadcast response.....	41
Figure 19: Web content resolver.....	42
Figure 20: vulnerable contents provider.....	43
Figure 21: SQL output example.....	46
Figure 22: Launching a shell.....	47
Figure 23: Content of the file boot.txt.....	48
Figure 24: User network information.....	60
Figure 25: Itaca poster.....	68
Figure 26: Masquerade menu.....	73
Figure 27: Activities output.....	74
Figure 28: Broadcasts output.....	75
Figure 29: Package sub-menu.....	76
Figure 30: Providers output.....	77
Figure 31: Services output.....	78
Figure 32: Exploits menu.....	79
Figure 33: Information sub- menu.....	79

Figure 34: Scanner sub-menu.....80
Figure 35: general info example.....81
Figure 36: SQL vulnerabilities.....82

Index of tables

Table 1: Androguard's features.....	22
Table 2: Tool's summary.....	35
Table 3: Results	44
Table 4: target devices.....	50
Table 5: Malformed Intents.....	53
Table 6: Access control problems.....	54
Table 7: Secure storage and encryption.....	56
Table 8: Insecure file operations.....	58

1 Introduction

During the last decade, we have seen the rise of a new generation of personal devices that has revolutionized our daily lives. The reduction of integration scales of computer components as well as the efforts to develop new communication technologies, and the advances to enlarge power battery lifetime, has led to an evolution of the functionalities traditionally offered by mobile phones. Such functionalities are addressed to offer new services that will redefine our relationship with other individuals thus gradually approaching that many have a deep impact on the security of mobile devices. Unfortunately these are just some examples showing the magnitude of the problem. Consequently, the provision of mechanisms to preserve the security in our daily mobile devices is as important as the need for methodologies and approaches to analyse the level of security they provide to such devices. Although their use is generalized in developed countries, and is increasingly common to find them in various forms, sizes or names (tablets, eye-wear, watches), their most important boom is taking place in developing countries, even overcoming the conventional PC, given the higher capabilities they offer at a lower cost.

The architecture of these devices, increasingly diverse and complex, is now more similar to personal computers. This fact is illustrated through the extended use of operating systems (OS) addressed to mobile devices. The pioneer in the field of mobile phones was Apple, with its iPhone, which incorporates IOS. This system captures much of the market share. Windows' proposal is called Windows Phone, but it lacks of enough acceptance. Finally we have the Android operating system, the most widespread. It has significant market share because it is freely available and each manufacturer can adapt it to their own mobile phone. In the last five years the Android operating system has experienced a tremendous growth, becoming the best-selling operating system for mobile devices. It is based on a robust Linux kernel, and is designed for touch-screen phones or tablets. Currently, Android has 75% of the market share. Part of its success is due to the facility of customization offered by the operating system. Unfortunately, this same reason has become Android in the main target of malware developers. Basically, developers try to exploit potential vulnerabilities resulting from modifications made by the device manufacturer, or the negligence of application developers, as they often do not bear in mind the recommendations set out to create a program.

Lately, we have seen more and more cases of serious vulnerabilities, such as the ability to remotely wipe a terminal (especially affecting Samsung devices) or run malicious codes [24]. Security is important for any device, but it is particularly important in personal devices in which we stored all kind of information concerning the private life of users, such as emails, photos, etc..

Currently there are almost barely tools to evaluate the Android system security. Although there have been various studies as those shown in [7] [28], [29], [30], [31], [32], and gradually new frameworks are emerging, such as Mercury, AndroGuard, ASEF, etc., there is still a long way to go in determining with rigour the degree of vulnerability of the system. One of the most important steps in the development of a new device and its potential applications, is the assessment of their vulnerabilities and that today, in the realm of Android, is a topic with a huge room for improvement.

Thus, the main objective of this master's thesis is to study the security issues of mobile devices based on the Android operating system. By covering this goal, we expect to answer questions such as:

- How to classify current applications for vulnerability assessment on Android?
- What kind of vulnerabilities can be exploited on Android systems?
- Is there any way to systematise the injection of attack on Android systems to show their security bottlenecks?
- What kind of results can be obtained from evaluating the security of real mainstream devices

To address these issues, the rent of this master's thesis is structured as follows: Chapter II introduces the Android systems and its most important security threats. Chapter III presents current security application, classified according to a novel taxonomy. Chapter IV proposes a methodology to apply the most interesting properties identified in Chapter III from an experimental viewpoint. At the chapter V we can see the case of study. Chapter VI shows the results of a case study where different mobile devices have been evaluated using our methodology. Finally, chapter VII concludes this master's thesis with the most important remarks. At the chapter IX we can see the Annex, with a tool created for us, to automate some tasks, and we can see too, the poster showed at ITACA work day.

2 Android in a nutshell

This section explains how Android works. We will briefly introduce all the elements that compound Android, especially focusing on the Android components and the communications among one another.

2.1 Android layers



Figure 1: Android architecture layers

Let's start off by taking a look at the overall system architecture of the Android stack in the *Figure 1* we have a graphical overview from the Android layers.

At the lowest level we find is the Linux Kernel. Android uses Linux for its device drivers, memory management, process management, and networking.

The next level up contains the Android native libraries. They are all written in C/C++ internally, but you'll be calling them through Java interfaces. In this layer you can find the Surface Manager (for compositing windows), 2D and 3D graphics, Media codecs (MPEG-4, H.264, MP3, etc.), the SQL database (SQLite), and a native web browser engine (WebKit).

Next is the Android runtime, including the Dalvik Virtual Machine. Dalvik runs dex files, which are converted at compile time from standard class and jar files. Dex files are more compact

and efficient than class files, an important consideration for the limited memory and battery powered devices that Android targets.

The core Java libraries are also part of the Android runtime. They are written in Java, as is everything above this layer. Here, Android provides a substantial subset of the Java 5 Standard Edition packages, including Collections, I/O, and so forth.

The next level up is the Application Framework layer. Parts of this tool-kit are provided by Google, and parts are extensions or services customized by the manufactures. The most important component of the framework is the Activity Manager, which manages the life cycle of applications and a common "back-stack" for user navigation.

Finally, the top layer is the Applications layer, where ours apps are typically located, such as the Phone and Web Browser utilities.

Since applications become an important security bottleneck we will pay especial attention to them in the rest of this chapter.

Android applications are built using essential components blocks, each of which exists as its own entity and plays a specific role, each item is a unique piece that helps define the overall behaviour of the application. It is noteworthy that some of these elements are the entry point for users to interact with the application and in many cases we see that depend on other elements.

There are four types of components in an Android application. Each has a purpose and a different life cycle that defines how it creates and destroys the component. Activities, Intents, Contents Providers, Services and Broadcasts receivers.

2.2 Application Components

As we can read in the Google developers guide, application components are the essential building blocks of an Android application. Each component is a different point through which the system can enter your application. Not all components are actual entry points for the user and some depend on each other, but each one exists as its own entity and plays a specific role—each one is a unique building block that helps define your application's overall behaviour. In this section we will link the components with the potential vulnerabilities.

Activities

An *activity* represents a single screen with a user interface. For example, an email application might have one activity that shows a list of new emails, another activity to compose an email,

and another activity for reading emails. Although the activities work together to form a cohesive user experience in the email application, each one is independent of the others. As such, a different application can start any one of these activities (if the email application allows it). For example, a camera application can start the activity in the email application that composes new mail, in order for the user to share a picture. Here we can find some vulnerabilities (Invalidated Input vulnerability) when we start an activity we have to do it through an Intent, in this Intents often, we have to include some fields like Integers, Strings or similar, the problem begin with the no validation of these fields by the developer of the activity, that receives the intent, a malicious app can launch a malformed intent an if we didn't filtered the fields, we can receive a String in the field where we expect an Integer for example, making the app crash.

Services

A *service* 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. For example, a service might play music in the background while the user is in a different application, or it might fetch data over the network without blocking user interaction with an activity. Another component, such as an activity, can start the service and let it run or bind to it in order to interact with it. In this component we can find the typical Denial of service (DoS) attack, for example we can launch a series of services that requires a lot of CPU, for example a complex algorithm, to reach the objective of exhaust the battery of the mobile device, the user of that mobile will not understand why the battery runs out so quickly, because the services in Android doesn't require a layout so it runs in background.

Content providers

A *content provider* manages a shared set of application data. You can store the data in the file system, an SQLite database, on the web, or any other persistent storage location your application can access. Through the content provider, other applications can query or even 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 application with the proper permissions can query part of the content provider (such as `ContactsContract.Data`) to read and write information about a particular person. Content providers are also useful for reading and writing data that is private to your application and not shared. For example, the Note Pad sample application, uses a content provider to save notes. In this case we can find a pair of vulnerabilities, first the acces-control, we can access to Contents Providers from

another apps or from the OS, without requiring a special permission, and the second vulnerability the Bypass, the Content Provider of an app is supposedly protected against unauthorized access by others apps by the sandbox, but it's proved that we can cross that sandbox easily[45], this is a big problem, because we're not sure if the information that we store in our Content Provider is secure or not.

Broadcast receivers

A *broadcast receiver* is a component that responds to system-wide broadcast announcements. 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. Applications can also initiate broadcasts—for example, to let other applications know that some data has been downloaded to the device and is available for them to use. 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. For instance, it might initiate a service to perform some work based on the event. In this component we can find another access-control vulnerability, we can launch a System Broadcast, that supposedly only the OS can launch, like `Dreaming_Started` or `Dreaming_Stopped`, launching this type of broadcasts we can modify the behaviour of a app that considers that broadcasts, like modifying their operations or just stopping it.

2.3 Communication between components

The communication between the Android components is essential. Android uses an IPC (Inter Process Communication) mechanisms, very similar to that used on IOS (Apple operative system), to allow the communication among the processes of the system. Interprocess communication (IPC) is a set of programming interfaces that allow a programmer to coordinate activities among different program processes that can run concurrently in an operating system. This allows a program to handle many user requests at the same time. Since even a single user request may result in multiple processes running in the operating system on the user's behalf, the processes need to communicate with each other. The IPC interfaces make this possible. Each IPC method has its own advantages and limitations so it is not unusual for a single program to use all of the IPC methods. There are three types of IPC mechanisms in Android:

- Intents, are messages between components. They are a mechanism to pass data between processes. Through intents one can starts services or activities, or invoke broadcast

receivers.

- Bundles, are entities to encapsulate data. Their use is similar to the concept of object serialisation, but faster in Android.
- Binders, are entities that allow the permissions to obtain a reference to another service as well as to send messages.

But we have some studies that the IPC is not as strength as we think, in [25] we can read that we are able to crash the Android runtime from unprivileged user processes, because the exception handling is a rarity between the Android developers.

2.4 Android, a vulnerable system

Even though our device contains vulnerabilities, they may not be exploitable, ie, although our system contains multiple vulnerabilities, this does not mean that this is vulnerable, as these can be not exploited, either because they are unaware of just no-one known to exist, although they are in our system, because they are not accessible, not having a tool that possibility us to access them. One of the consequences is that they simplify the attacks on our system, as it is much easier to try to exploit a weakness in our system to be tested potential vulnerabilities in our system, the means to exploit a vulnerability in the attack and subsequent consequences can be of any kind, theft of private information, introduction of erroneous data in the system, denial of services, remote control socket terminal or just, turn our mobile device to a brick.

Securing an open platform as Android requires a robust security architecture and rigorous security programs. Android was designed with multi-layered security that provides the flexibility required for an open platform, while providing protection for all users of the platform. Android considers the use of security controls to reduce damage of the bugs introduced by developers. Thus, even less skilled developers can work with and rely on flexible security controls. Unfortunately the design of these controls is vulnerable to malware, and attacks on third-party applications on Android. Android was designed to reduce both the probability of these attacks and greatly limit the impact of the attack in the event it was successful. However, despite such efforts Android is still a vulnerable system.

Some studies claim that in late 2013, the Android system, could be threatened by over 1 million threats as we can read in [26], as on Figure 4 shows. One of the main causes is that is that Google is much more flexible than Apple and therefore does not restrict the freedom of users when changing their mobile system. But delving a little more, the above answer that we have is a little short, if you look at the mechanism of use we found large differences as we can see in the *figure 2*.

On the one hand we have Google, that has delegated the control of the Android operative system to mobile producers, and the decision of when and how they will upgrade your operating system, so we are in a market, in which there are many versions of Android as manufacturers, for this reason, we have one of the biggest problems of Android, the fragmentation, on the other hand we have Apple, which is the leading version control and delivers these versions directly to users.

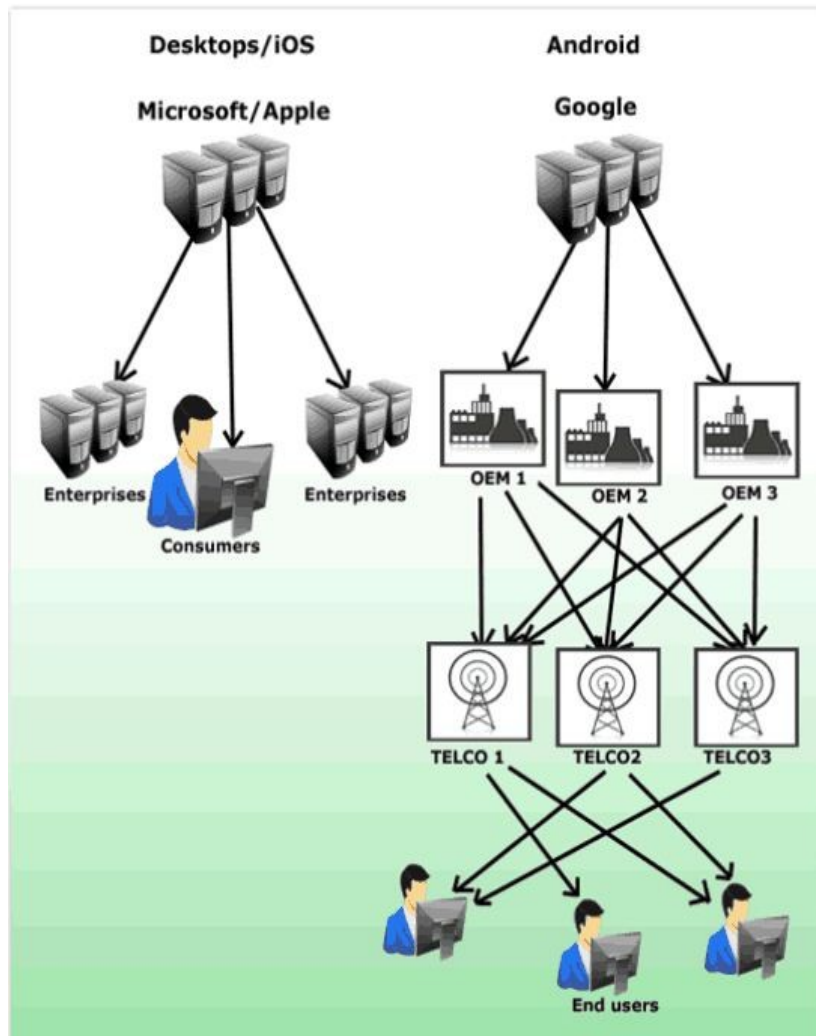


Figure 2: Android SO distribution

Talking about the fragmentation of Android, let us illustrate the below example, with some graphics, considering data from Google, the most common version is 2.3 (44%), as we can see in the figure 3, it was last updated in September 2011, while the 4.1 version only 14.9% used, and very likely to get forgotten, because the version called “Key Lime Pie” will be released soon. By contrast, Apple got that over 20% of its users were in version 6.1, after 36 hours of this release. We can read more about most used versions in [27]

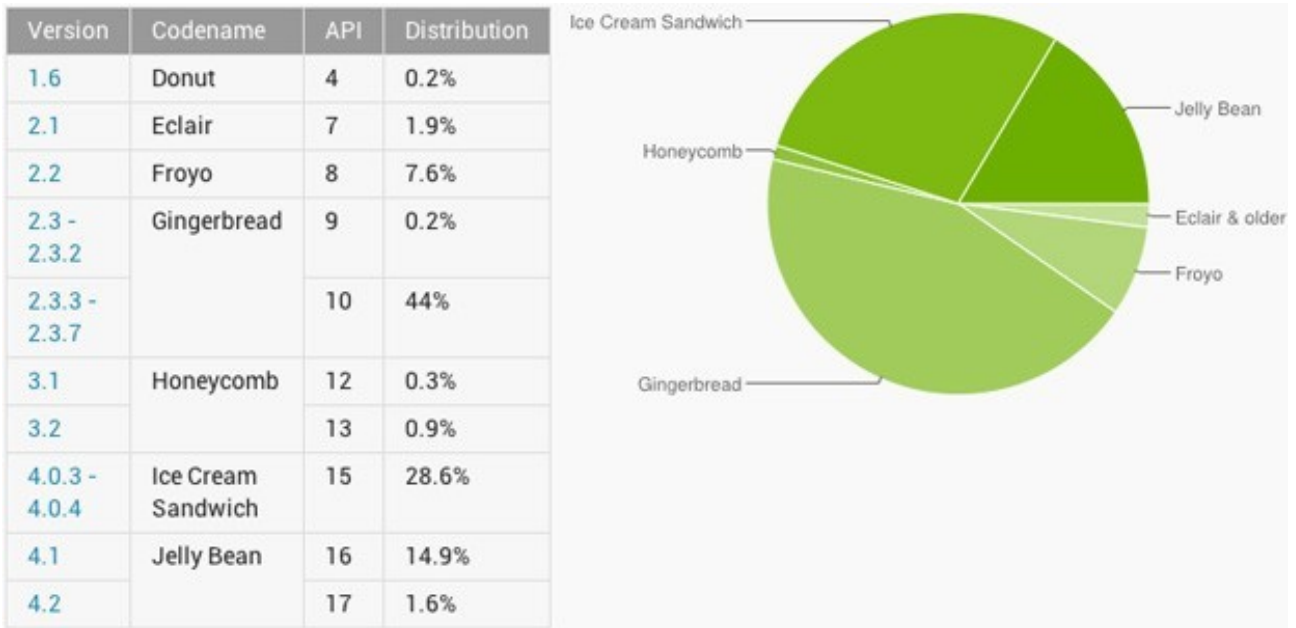


Figure 3: Android versions

For the end user, it may sound trivial, but it is not. The Android updates, has associated security enhancements and more features. The above cycle, makes updates slower, opening one window to the hackers, allowing them to create new malware to exploit the vulnerabilities discovered. And so the malware charts show disturbing increases.

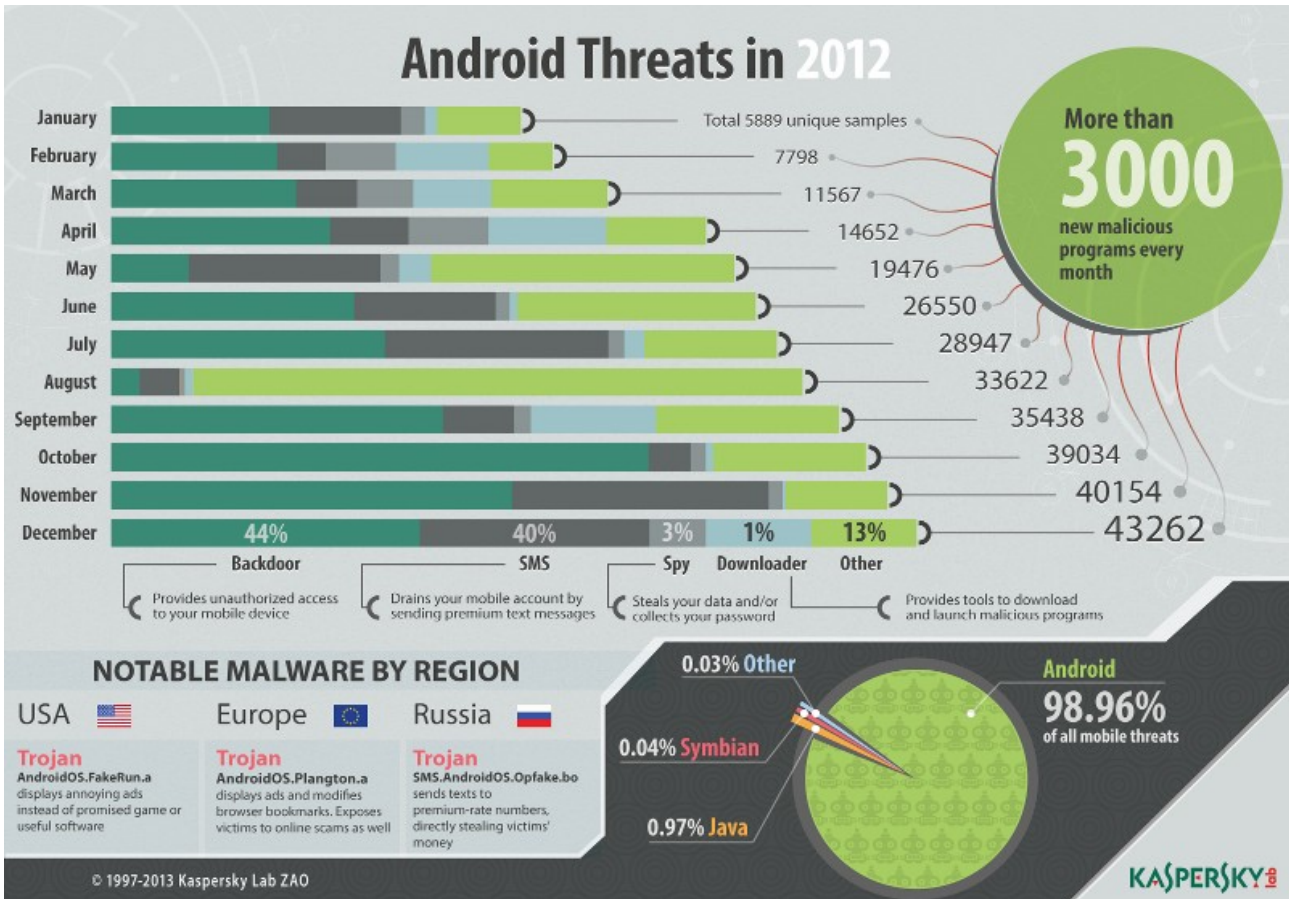


Figure 4: Android threats

At the above picture we can see the Android threats in 2012, as we can see the number of hazards are increasing monthly, illustrating that data we have the figure 5, and mainly affect android. That numbers are disturbing, we can see that in early 2012, the threats were only 3000, only in four months that threats were almost quadruplicate the initial number, closer to 11.000, but not only that, in the last 6 months of the year, the number has increased to the amazing amount of 129.000, 43 times more than in the beginning of the year! Where we will en up! If we don't stop that, that problem may will end with Android.



Figure 5: Android threats

Now, we will explain the most common vulnerabilities that we can find in our systems and mobile devices. Some of them are invalidated input, access-control problems, secure storage an encryption, insecure file operations, DoS, Bypass.

In the follow lines, we are going to explain the vulnerabilities that we can commonly find in Android systems.

- **Unvalidated input:** Information from another activity or Intent that it has not been validated, before being used by the receiver. Attackers can use these flaws to attack the components of our app.
- **Acces-control:** It is an error due to the lack of enforcement pertaining to users or functions that are permitted, or denied, access to an object or a resource.
- **Secure storage and encryption:** Is a common vulnerability that occurs when sensitive data is not stored securely. Insecure Cryptographic Storage isn't a single vulnerability, is a collection of vulnerabilities.
- **DoS:** Attackers can consume mobile phone resources, to achieve the goal of, prevent the normal use of the owner's mobile device. Attackers can also steal users accounts or even stop the mobile device.
- **Bypass:** Bypass, in general, means either to go around something by an external route rather than going through it. In network security, a bypass is a flaw in a security system that allows an attacker

to circumvent security mechanisms to get system or network access.

In conclusion, the security industry must be one step ahead of the hackers. As we see in previous graphics Android is under serious threat and may be, if we don't make anything, Android will disappear drowned by countless threats, malicious apps, malware etc. But this creates new answers from the community, for example the Cloud, which has created an app repository and a reputation service, that protects us from malicious or fraudulent apps, avoiding that the download to our device. We have too an evaluation by the users, voting the apps, causing the suspicious apps be blocked. Furthermore we can use an anti-virus but as we can see in [33] they are useless, because in Android, does not exist the typical pc virus, we have a new model of attacks.

2.5 Summary

In this chapter we have taken a quick look to the Android SO, we have seen its main components, layers, explaining its main components and how they interact between them. We have introduced the vulnerabilities related to Android, explaining why Android is a very threatened and vulnerable OS. But what is the next step? Now that we have focused the main threats, and the possible consequences, we have to find the way to avoid and detect such problems. In the next chapter we will characterize and classify an extensive collection of tools, as a result of an exhaustive search.

3 Classification of security tools for Android

Currently there is a wide variety of tools that help us to evaluate the security of the applications we have installed in our phones or tablets. However, given their heterogeneous nature, it is difficult to select which is the most adequate tool for a given system. Unfortunately, to this day there is an absence of the tools that can help us in determining the functionalities they offer

This chapter proposes a taxonomy to characterize the purpose of the security tools in the domain of Android.

The main features we look for in these tools, which help us in finding vulnerabilities are, that is a dynamic tool and allows us to interact with the device that we're looking for vulnerabilities, and be able to work with him, is a very important point in our search, to be able to work in real time, we are able to do experiments in various contexts and workloads, it is not the same find security flaws in a terminal, in actual use, than with an emulator. The actual device will always be more reliable, and we will get real results with an emulator.

It is also important to have a proper working environment, to make the work easier, we don't want to have to be introducing complicated orders, or spend too much time to learn how to use it. It is also valuable that already contains predefined orders to allow us to start working immediately, since we are interested in studying ourselves immediately, that is our goal. Another feature to appreciate is that, the tool has a community that gives us support when we have a problem, or when we find a bug, it's always important to have this type of aid to choose one option over another. It is also important that the tool in question this regularly updated, with input from other users, such as improvements or even with user's new contributions, that are not linked to the project, because our experience says that, an application or tool grows faster if you have a large community supporting it.

According to their way of working, the tools can be dynamic or static, depending on the type of the analysis that they perform of the system. The first, enables an interaction between user and system, while the second does not require the system executed nor the user interaction.

According to their deployments, tools may require the use of additional hardware or software, or the tools may be executed desolately in devices, may be we need more components to start working with the tool.

According to their purpose, we can find tools that just analyse the manifest of the installed apps, we have another's that decompile the apk packets, we have tools that can draw the logical flow of the application, we have applications that searches for atypical patters, we can find tools to modify the entire Android system, changing the icons of the battery, the sounds, and much more.

3.1 Dynamic tools

In this section we can find two types of security tools, on the one hand dynamic tools, why dynamic? because we can interact with them, we can change the inputs and get different outputs, in the other hand we have the static tools, we can not interact with them, we just wait for the results.

3.1.1 Androguard



Androguard [7, 12, 16, 20] is programmed in python, is not just a malware analysis tool on Android, actually it is a complete framework that allows us to interact directly with malicious code, read their resources, access the code, and even compare different threats, find similarities or differences in their methods, classes and resources. It is also possible to incorporate all the features of Androguard to make custom scripts in Python to obtain all the details in a simply way about a file.

This framework first available in the project website Androguard, or can be used through Santoku[34] distribution, which includes ans it's already installed. In either case, Androguard makes it possible for security experts a platform extremely complex, useful for analysing malicious code on mobile platforms. In addition, you can also download the necessary modules for inclusion within any personal tool for analysing malware[11, 13] on Android.

The reach that this framework provides for the analysis of Android malware is excellent and allows a better understanding of the threat as well as a better knowledge of its internal structure and functionalities. In addition, features Androguard file comparison tools, similarity search with other known threats[14], visualization capabilities and more.

The tool is available only for Linux, you can install it in various ways, we can get off the binaries and compile them on our own, on the other hand we can install it with the order apt-get install and following a few simple steps will have installed in minutes. Whether you choose an option as another, we have to overcome a dependency accounts, such as python 2.6 and optionally if we want all the functionality available Androguard, ipython, pygments (colour code), pyfuzzy (to calculate the risk of the. apk), mercury (to activate andromercury.py) psyco (to speed up Androguard). [15]

By default Androguard brings a series of tools and scripts to facilitate the work of the analyst and lets you interact in a simple and dynamic threats. One of the most practical options, and a good place to start learning about this framework is Androlyze, which allows us to start an interactive shell analysis.

Androguard's analysis is static, as the program runs on a computer with Linux not in the

mobile device. This analysis focuses on the apk packages. We can find more information in the Androguard's blog in [10]. In the *table 1* we have the main features from Androguard.

Main features

Format types supported	Differences similarity analysis	Static analysis	Support
APK	APK/DEX	Flow control graphs	Support sessions
DEX	Similarities between methods	Search for packages, methods, fields	Remember orders history
JAR	It detects plagiarism between apps		
Bytecodes (Dalvik)	Comparison based on compressors		
Byte-codes (Java)			

Table 1: Androguard's features

3.1.2 APK-tool



APK-tool[3] is a reverse engineering tool[18], designed to analyse third-party Android apps. Provides the ability to debug Smali[19] code step by step. Makes working with apps easier, APK-tool can decode source code to make it very close to the original, and rebuild after being modified, you can automate tasks like building, sign, align apks. It also offers the possibility of access to the app resources, such as icons or strings and can change them to need and return to recompile the app and install it. The code is available at [9]

Features

- Decode source code (including resources.arsc, XMLs and 9.png) and compile it.
- APK-tool, give us the option to debug Smali code.
- Give support for repetitive tasks.
- Sign an apk

APK-tool is available for Windows, MacOS and Linux, installation is as easy as unpacking the file we downloaded and run, we do no need to install additional packages or dependencies, so just have superuser permissions.

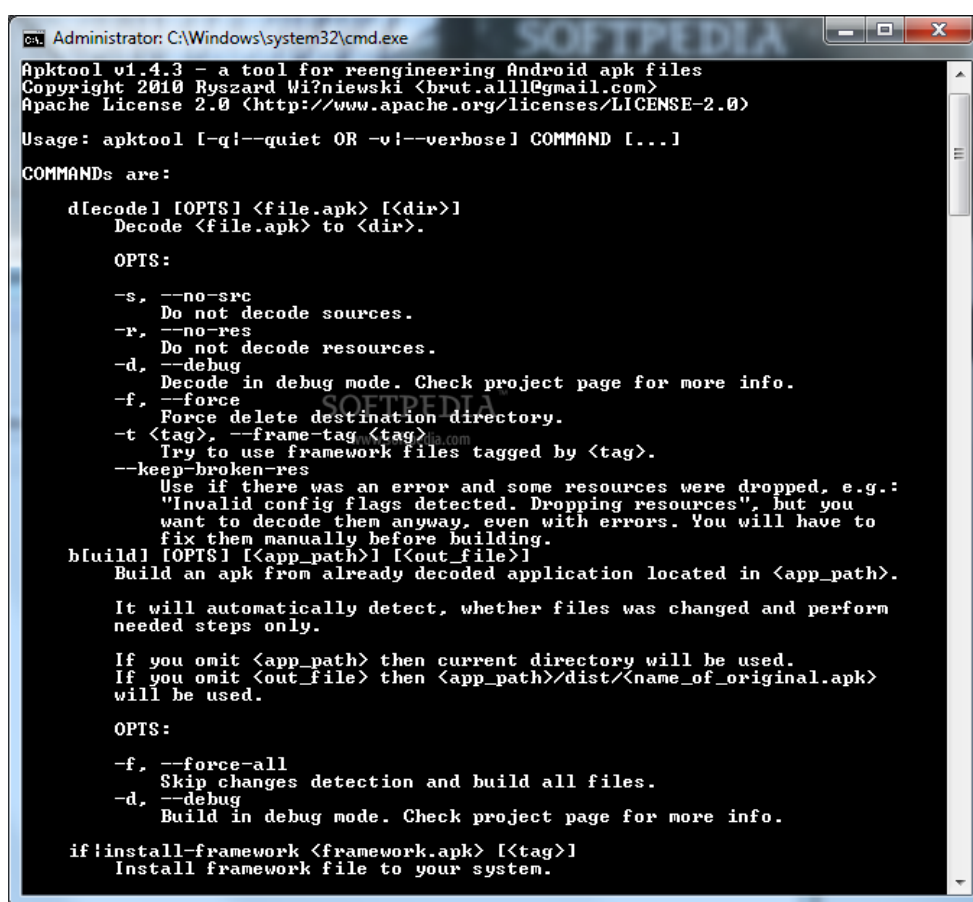
Apk-tool is a closed tool, we can not change anything, just have available the functionality that it offers, just have a series of commands accessible via *cmd* console or in case of Windows, with their respective options.

It is an open project we can find it in code.google, APK-tool is programmed in Python, we

can support the project by downloading the repository on GitHub and we can contribute to the project with our features or ideas.

APK-tool is a static analysis tool, that is restricted to analysing the contents of the package apk, running on the computer on which you have installed and not on the mobile device, so we don't need a real mobile device, that tool is focused on the analysis of the contents of the install application package.

In the *figure 6* we have an example from the available commands from APK-tool, we can find decoding orders, with the options from debug mode, decode source, deleting destination folder, and installing a framework.



```
Administrator: C:\Windows\system32\cmd.exe
Apktool v1.4.3 - a tool for reengineering Android apk files
Copyright 2010 Ryszard Wi?niewski <brut.all@gmail.com>
Apache License 2.0 <http://www.apache.org/licenses/LICENSE-2.0>

Usage: apktool [-q|--quiet OR -v|--verbose] COMMAND [...]

COMMANDs are:

  d[ecode] [OPTS] <file.apk> [<dir>]
    Decode <file.apk> to <dir>.

    OPTS:

    -s, --no-src
      Do not decode sources.
    -r, --no-res
      Do not decode resources.
    -d, --debug
      Decode in debug mode. Check project page for more info.
    -f, --force
      Force delete destination directory.
    -t <tag>, --frame-tag <tag>
      Try to use framework files tagged by <tag>.
    --keep-broken-res
      Use if there was an error and some resources were dropped, e.g.:
      "Invalid config flags detected. Dropping resources", but you
      want to decode them anyway, even with errors. You will have to
      fix them manually before building.
  b[uild] [OPTS] [<app_path>] [<out_file>]
    Build an apk from already decoded application located in <app_path>.

    It will automatically detect, whether files was changed and perform
    needed steps only.

    If you omit <app_path> then current directory will be used.
    If you omit <out_file> then <app_path>/dist/<name_of_original.apk>
    will be used.

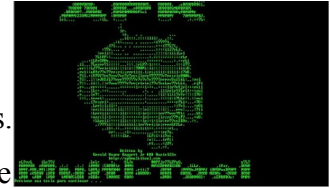
    OPTS:

    -f, --force-all
      Skip changes detection and build all files.
    -d, --debug
      Build in debug mode. Check project page for more info.

  i[install-framework] <framework.apk> [<tag>]
    Install framework file to your system.
```

Figure 6: APK-tool menu

3.1.3 APK Multi-tool



Decompiling tool which allows us to decompile the Android apps. APK Multi-tool [8] offers a wide range of options supported for simple tasks, like, removing a mobile device apk, optimizing images, sign an apk. It also supports more complex tasks, for example, compile part of the apk. Once decompiled the apk packet, you can modify the code of the application, although is not recommended because it is in a pseudo-code, it's quite difficult to understand if we don't have the suitable knowledge.

APK Multi-tool is only available for windows, is installed on your computer and the analysis is done from the it, you can access through the command line, and offers a menu from which you can use all functionalities. This tool can not be configured, is a closed tool, and we can only use the options offered. The analysis of that this tool is static, since its function is centred on apk packages, running on the computer and not on the mobile device.

Developed initially by Daneshm90, the code is not available.

Features:

- Check apk dependencies.
- Allows to modifying system apks.
- Apk optimization (Zipalign, OptiPNG)
- Provide log the events.
- Quick apk signature.
- Error detection, checks if errors have occurred while performing a task.

- Apks multi installation support.

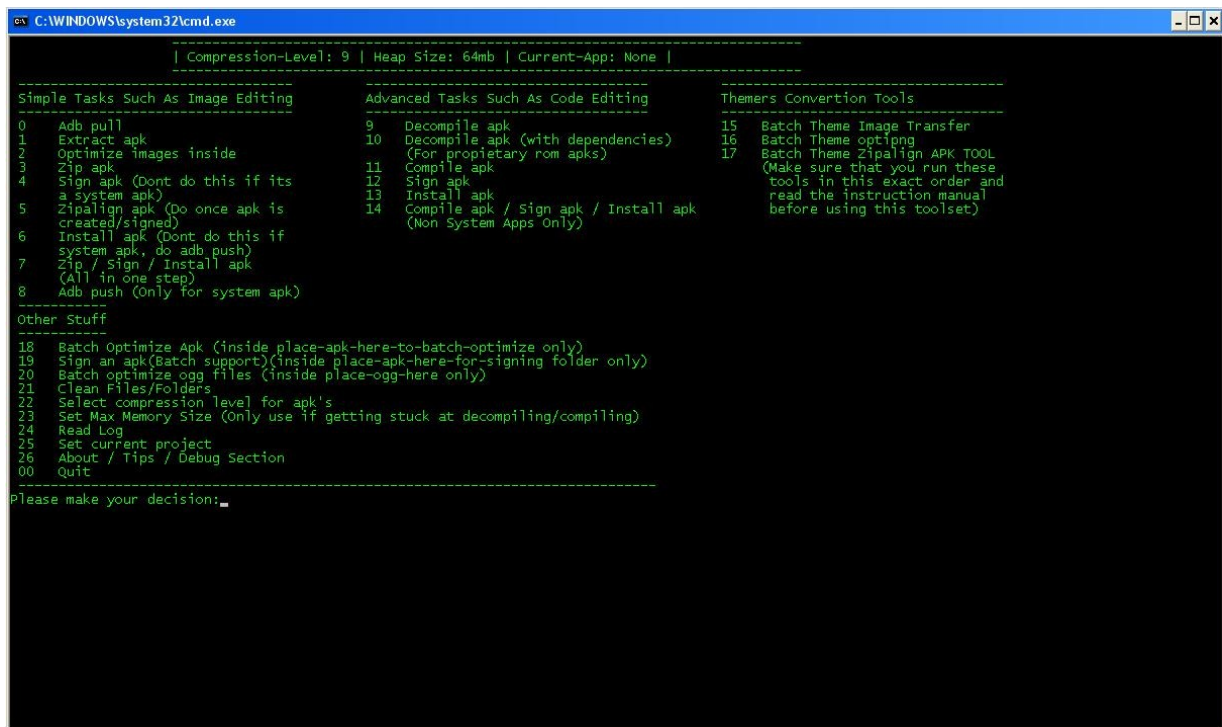


Figure 7: APK-Multi tool menu

At the figure 7 we can see the APK-Multi tool menu, here we can find all the offered features, as we can see it's very easy to access to the orders, just we have to insert the code, here we have too a short explanation of each command.

3.1.4 Mercury



Mercury[1,2,4,5,6] is a framework, written in Python and available only for Linux, that allows you to explore the Android platform, to find vulnerabilities and exploits to share with its developer community. This tool allows you to assume the role of a without Android app privileges and interact with other apps and with the system. It gives the user an interface where it can explore the environment of the mobile device, you can access the different components of the installed applications.

Mercury Installation is only available for Linux, is simple, we just have to get off the installation package, correct occasional dependence on Python, and have installed the ADB (Android Debug Bridge), once installed, we just need to execute a command on the command console and we're in Mercury console.

Mercury comes with a number of predefined modules, which are the main base of resources available to the tool, but you can add more modules written for us to meet the needs of the user, we can automate tasks by writing a script that uses the orders already defined in Mercury.

Mercury follows a client-server model, the client is formed by the console that is which sends the orders and showing the results on the screen, and the server that runs on the mobile device, so that all the results get real-time, so that we have a tool that performs dynamic analysis.

Mercury was born of an initiative MWRLabs a security company, the tool is freely available, we we can download and use freely. The source code is accessible via a GitHub project.

Allows you to add modules and create scripts to add new features.

Mercury allows you to:

- Interact with the 4 IPC endpoints-activities, broadcast receivers, content providers and services
- Use a proper shell that allows you to play with the underlying Linux OS from the point of view of an unprivileged application.
- Find information on installed packages with optional search filters to allow for better control
- Built-in commands that can check application attack vectors on installed applications .
- Transfer files between the Android device and your computer
- Create new modules [6] to exploit your latest finding on Android, and playing with those

that others have found.

```
mercury> list
app.activity.forintent    Find activities that can handle the given intent
app.activity.info        Gets information about exported activities.
app.activity.start        Start an Activity
app.broadcast.info        Get information about broadcast receivers
app.broadcast.send        Send broadcast using an intent
app.package.attacksurface Get attack surface of package
app.package.debuggable    Find debuggable packages
app.package.info          Get information about installed packages
app.package.launchintent  Get launch intent of package
app.package.list          List Packages
app.package.manifest      Get AndroidManifest.xml of package
app.package.shareduid      Look for packages with shared UIDs
app.provider.columns      List columns in content provider
app.provider.delete        Delete from a content provider
app.provider.download      Download a file from a content provider that
                           supports files
app.provider.finduri       Find referenced content URIs in a package
app.provider.info          Get information about exported content providers
app.provider.insert        Insert into a Content Provider
app.provider.query         Query a content provider
app.provider.read          Read from a content provider that supports files
app.provider.update        Update a record in a content provider
app.service.info          Get information about exported services
app.service.start          Start Service
app.service.stop           Stop Service
auxiliary.webcontentresolver
```

Figure 8: Mercury Menu

At the *figure 8* we can see the output of the Mercury command *ls*, here we can find a piece of the commands that we can find in Mercury.

Nowadays the Mercury protect is discontinued, the last version is the 2.2, MWR labs has decided to change Mercury for Drozer, at the *figure 9*, we can see the logo from the new MRW-Labs tool.



Figure 9: Drozer logo



3.1.5 ASEF-Android Security Evaluation Framework

Android Security Evaluation Framework (ASEF)[20], performs an analysis while applications are running on your mobile device, alerting us to the problems that may arise. It can warn us about installed apps that can expose vulnerable components, and can help us find and remove suspicious apps to their subsequent verification and removed if necessary. ASEF works this way, you choose a set of apps already installed or from a apk folder, will be migrated to a virtual device (AVD), where they pass a series of tests.

During the test cycles, the applications are installed on the AVD and are launched. ASEF causes certain behaviours by sending random or pre-set gestures and then uninstall the application automatically. It can capture logs, network traffic, kernel logs, memory dumps, running processes and other parameters at all stages of the life cycle of the app, later all these logs will be used by the ASEF parser. That attempt to determine for example the aggressive use of bandwidth, the interactions with any server that uses the API Google Safe Browsing, a permission assignments and known security failures. ASEF can be easily integrate with other open source tools to capture sensitive information such as SIM card numbers, phone numbers and other.

ASEF is only compatible with Linux and MacOS, can not be installed on another operating system, is installed on the computer and works with apks folders. We can install it by downloading a package, we need to fix some python dependencies, installing certain modules, `Getopt :: Std; URI :: Find; URI :: Encode`, once done already and we can access ASEF.

ASEF is an open source[21] tool for analysing the security of Android devices. Users can perform basic analysis using the scenarios that come in a default installation, or if you want to extract more information you can create or modify other scenarios that already brings the tool, you can even find other patterns of analysis. ASEF provides automated testing by providing a plug and play, making it possible to stay updated with the latest developments in the field of apps.

As we can see at the *figure 10*, ASEF works like a black box, we can introduce a folder with the apks packages, that we want to analyse and ASEF takes care of everything, it self makes analysis to detect malware, aggressive adware, we don't have to introduce a command or make a choice.

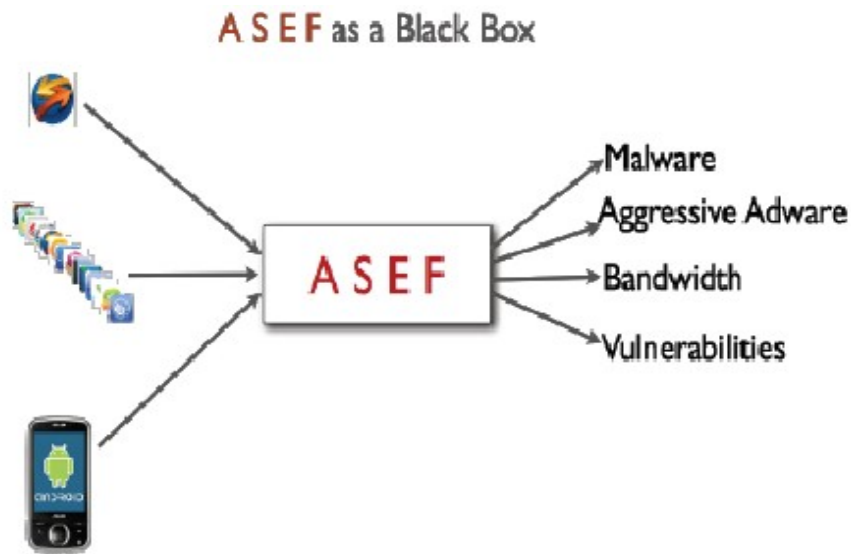


Figure 10: ASEF diagram

3.2 Static tools

Here we have all the tools that not interact with any device, they just are used to static analysis, these tools perform tests of apk packets, analysing for example, the manifest, the permissions manifest, or analysing the flow of the application for seek hidden features.

3.2.1 ComDroid

ComDroid[16] is an online tool that offers a static analysis of our app, we just need to have the apk package and upload it to the ComDroid web and after a brief analysis will show, that it allows us to see the apk vulnerabilities, the IPC communications will be emphasized, and other typical faults by errors of the programmer. Relies on a database of patterns vulnerabilities. Android supports communication between applications through the use of Intents. Unfortunately, these can make your application vulnerable to others. The content of Intents can be intercepted, modified, stolen, or replaced, compromising user privacy. Also, a malicious application can inject false or malicious Intents, which can access private user data or violating security policies of the application.

Comdroid bases its operation in search of patterns in our code, running on a remote machine, we assume that decompiles the apk package and analyses, therefore performs a static analysis.

This tool does not require any installation, just we access their website, we just upload our apk package and application and takes care of analysing our code, after this, shows us our

application vulnerabilities. Hence this application does not support any configuration.

ComDroid has been developed by the University of Berkeley and is freely accessible, we have no access to the code, but we do know is that ComDroid is based on the JarJarBinks program code.

Unauthorized Intent Receipt

- Broadcast Theft
- Activity Hijacking
- Service Hijacking

Intent Spoofing

- Malicious Broadcast Injection
- Malicious Activity Launch
- Malicious Service Launch

3.2.2 StowAway

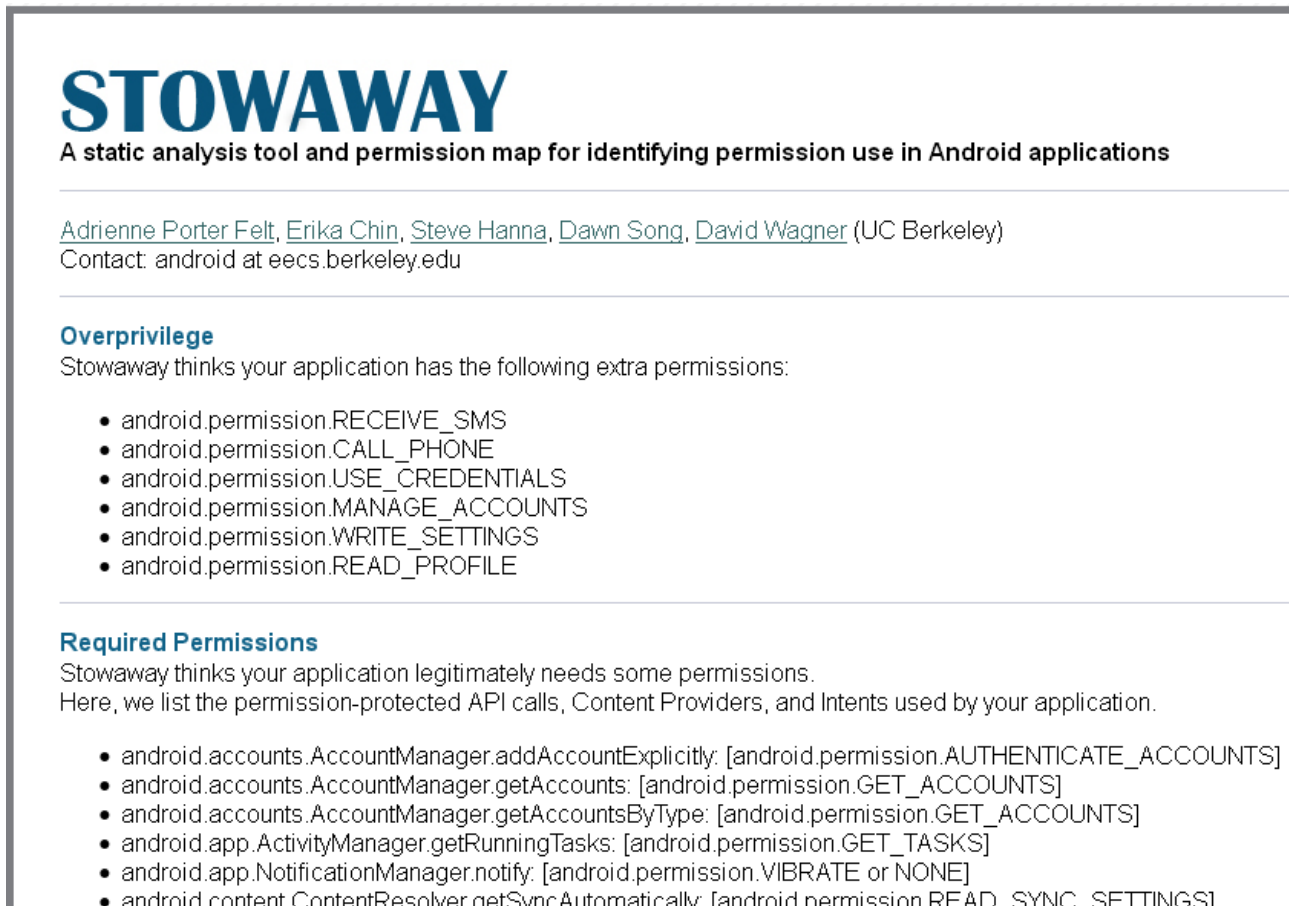
StowAway[17] is a static package analysis tool, that analyses our apk manifest, just have to upload our app to their website, and takes place a static analysis, returning a report on the permissions requested by the application. It also has a complete map of existing permissions on the Android API[22].

Certain services of the Android APIs are protected with permissions. To access the API calls, you need to include these permissions in the application manifest. But this can be dangerous, because if an app asks for more permissions than they actually need, this is an over-privileged application. It is important to consider when installing an application on your device, which permits asks. Common users typically do not consider which permissions an application asks, this can be a door to future theft information, and an increased vulnerability of our device. StowAway use automated tools to detect application permissions that analyses, returning a detailed report of the analysed application permissions.

This tool not need any installation, is a service via the Internet, we need only connect to the StowAway website and upload the package you want it to be analysed ,only accepts apk packets and after a few minutes we present a report of the analysis.

The tools doesn't accept any configuration because it is a web service, the type of analysis that performs this static type tool is because only analyses the apk package manifest also from the website assures us that do not stay back our app. This tool is free of charge, is a free service access, code itself is not accessible.

At the *figure 11* we have an analysis example output:



STOWAWAY
A static analysis tool and permission map for identifying permission use in Android applications

[Adrienne Porter Felt](#), [Erika Chin](#), [Steve Hanna](#), [Dawn Song](#), [David Wagner](#) (UC Berkeley)
Contact: android at eecs.berkeley.edu

Overprivilege
Stowaway thinks your application has the following extra permissions:

- android.permission.RECEIVE_SMS
- android.permission.CALL_PHONE
- android.permission.USE_CREDENTIALS
- android.permission.MANAGE_ACCOUNTS
- android.permission.WRITE_SETTINGS
- android.permission.READ_PROFILE

Required Permissions
Stowaway thinks your application legitimately needs some permissions.
Here, we list the permission-protected API calls, Content Providers, and Intents used by your application.

- android.accounts.AccountManager.addAccountExplicitly: [android.permission.AUTHENTICATE_ACCOUNTS]
- android.accounts.AccountManager.getAccounts: [android.permission.GET_ACCOUNTS]
- android.accounts.AccountManager.getAccountsByType: [android.permission.GET_ACCOUNTS]
- android.app.ActivityManager.getRunningTasks: [android.permission.GET_TASKS]
- android.app.NotificationManager.notify: [android.permission.VIBRATE or NONE]
- android.content.ContentResolver.getSyncAutomatically: [android.permission.READ_SYNC_SETTINGS]

Figure 11: Analysis example

3.2.3 Intent Fuzzer

Intent Fuzzer[23] is a tool that can be used on any device using the Google Android operating system (OS). Intent Fuzzer is exactly what it seems, which is a fuzzer. It often finds bugs that cause the system to crash or performance issues on the device. The tool can either fuzz a single component or all components. It works well on Broadcast receivers, and average on Services. For Activities, only single Activities can be fuzzed, not all them. Instrumentations can also be started using this interface, and content providers are listed, but are not an Intent based IPC mechanism.

This tool is only compatible with Android systems, that is mobile phones and tablets, but you can use it in the Android simulator. IntentFuzzer package is installed like any apk, analysis is dynamic, as it runs on the mobile device and in real time, since the test is performed while the device is running other tasks, so that the analysis can be influenced by the set of tasks that are running while the analysis.

IntentFuzzer supports some configuration, we can choose what kind of Intents are launched,

we can launch activities, broadcasts, providers, services or instrumentations, having chosen a category, we can choose specifically that activity, service etc, we want to launch the Intent.

Intent Fuzzer is freely available but the source code is not provided, but this is also easily accessible using the right tools.

At the *figure 12* we have the Intent Fuzzer application's overview, at the left side we have the initial menu, here we can choose the type and the component, we can choose to launch a series of malformed intents or a single intent, at the right side we have the output of a single intent.

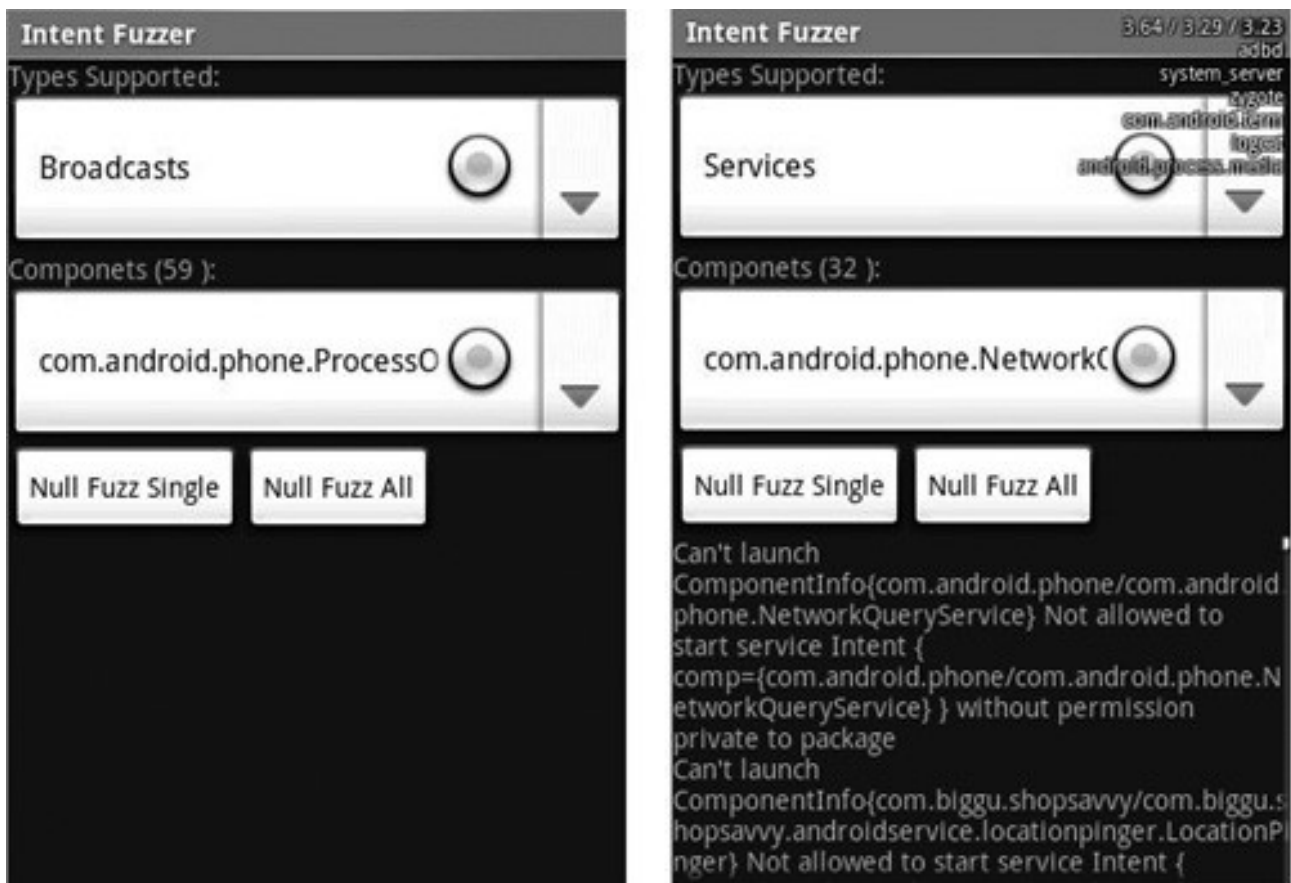


Figure 12:: Intent fuzzer overview

3.2.4 Intent Sniffer

Intent Sniffer is a tool that can be used on any device using the Google Android operating system (OS). On the Android OS, an Intent is description of an action to be performed, such as startService() to start a service. The Intent Sniffer tool performs monitoring of runtime routed broadcasts Intents. It does not see explicit broadcast Intents, but defaults to (mostly) unprivileged broadcasts. There is an option to see recent tasks Intents, as activity's Intents are visible when started and we can capture it and analyse aftherwards. The tool can also dynamically update.

Intent Sniffer is installed like any app, can be installed on any Android mobile device you have, does not need more to run. Supports certain level of configuration, you can choose what type of Inten can show on screen, performing a filter.

It makes a dynamic analysis, as this takes place while the other activities are running, so it may be that this is influenced by the environment in which the device is running so the analysis can be different even with the same mobile device.

ISEC Partners distributes freely this program, we should simply go to their website and download the apk package and install it, the code is not distributed, but is easily accessible.

At the *figure 13*, we have the output of an analysis, here we have selected to sniff the activities and the broadcasts and below of the refresh button we have the output of the sniffed activities and broadcasts.

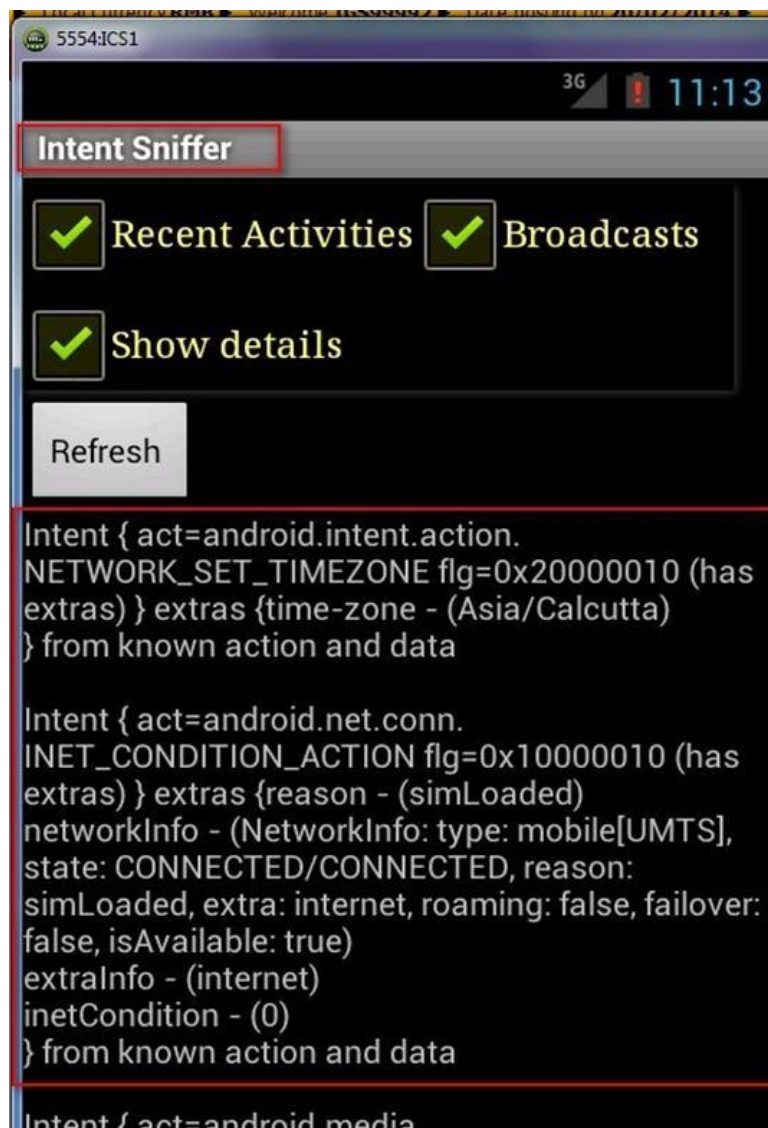


Figure 13: Overview

3.3 Tools summary

Here we have the tools summary table(*table 2*), here we have the main characteristics of the previous tools, we will comment some of them. On the left column we have the main features that we have considered the most important. The OS it's important because we don't have the same features in Windows, more restrictive than Linux, we prefer the last one because we can modify more parameter, in the offered tools we have more working on Linux than in Windows, and finally our computer will work better in Linux than in windows, when we perform the analysis. The available code is a good characteristic, because if we have it, we can adapt the tool to our necessities. The free access to the tool is important, because a free tool is always more accessible than a privative tool, we are not going to buy a tool just to test it, simply we proceed to the next. The support is an important point to consider, because if we have a problem we can have de help of the tool forum or the mailing list. The automating tasks it's important, to be able to find vulnerabilities automatically, that feature is important if we have a large number of devices to analyse. The need of a device it may be necessary but no essential, but its better if we have a real environment to make the tests, although is valuable if we can make the tests in an emulator, because the different versions of Android are more accessible.

Characteristics	Intent Fuzzer	Intent Sniffer	Manifest Explorer	Mercury	APK-tool	APK Multitool	Androguard	Comdroid
OS	Android	Android	Android	Linux	Linux	Windows	Html	Html
Emulator compatible	✓	✓	✓	✓				
Code available	✓ Github	✓ Github	✓ Github	✓ Github	✓ code.google		✓ code.google	
Free	✓	✓	✓	✓	✓	✓	✓	✓
Support	✓	✓	✓	✓				
Where is installed	Mobile	Mobile	Mobile	Pc	Pc	Pc	Pc	N/A
Type	Static	Static	Static	Dynamic	Static	Dynamic	Dynamic	Static
Interact with device	✓	✓		✓				
Written in	Java	Java	Java	Python	Python	?	Python	?
Find vulnerabilities	✓			✓				
SQL Injection				✓				
Needs a device	✓	✓		✓	✓	✓		
Compile, de-compile					✓	✓	✓	
Automating tasks				✓ scripts				

Table 2: Tool's summary

3.4 Why we choose Mercury

In this paragraph we will explain the election of the best tools for us, with that tool we will make the later analysis that we will explain in next chapters. We chose Mercury tool, because it is the most complete that, we have found from all we have been discussing, one of the most important features and that finally we decided on Mercury, is because Mercury is the only tool that allows us to work in real time with mobile devices such as tablets, mobile phones, etc. Therefore it's to work with the physical devices and obtain more realistic measures, with devices which are used in real life. Another feature which has made us decide for the option of Mercury is that this tool is in constant development (or it was in development, while we were carrying out that experience) since Mercury is part of a project, about security on mobile devices with the Android operating system. Also of note that Mercury has a whole community behind which supports the tool, through forums and emails. But Mercury is still in development, and some of their features are not completely finished, for example the intents orders are not completed and we can't launch them with all of

parameter types.

3.5 Summary

In this chapter we have explained the differences between a dynamic and static tool. We have taken a look at several tools that can help us to discover new vulnerabilities and find the known ones, how to test potential malicious apps, see the real permissions that an app requires and much more. We have explained the most important features of each tool that we have analysed and we have put all together in a comparing table, and at last we have justified the choice of Mercury as the tool to develop for that Master Thesis.

At the present day we can't find many tools that help us to find, maybe because Android is still too new, and the people are not greatly concerned about their mobile security, but we can find some initiatives that are starting to develop few tools to help us to find the weakness of our devices. But these tools do not cover all areas that we are looking for, so if we want to make a complete analysis we will need to combine some of them, because if we only focus on one tool we will not cover all the areas. The characteristics we look for in these tools are, that it will be based on Linux, because Android is a Linux based system. That we can use a device or an emulator, because it's so important for us that we can work with on a real environment, to find real vulnerabilities. It can be important if we can add our requirements, adding additional modules, we will adapt the tool to our needs. Although the cost of the tool is important, a non-payment may facilitate the diffusion and extending more easily. It's very important for a tool to have behind a community, because we can enlist the help of these people to fix problems or propose new features. It's very important that we can interact with the device, because not all devices behave the same way.

At this point, it's clear for us that Android is a vulnerable system, that needs tools to discover the threats, we have chosen the tool that will help us to carry out our job, but what's next? In the next chapter, we will explain the methodology that we follow to standardize the vulnerabilities search, because we need an efficient and an easy way to find them and the weakness of our mobile device, we need to define the steps to follow, for don't forget anything through all of the phases, in this way we will not forget anything.

4 Methodology supported by Mercury

This chapter presents a methodology to systematise the configuration, execution and analysis of experiments to detect vulnerabilities in Android. For this purpose of this master's thesis our methodology has been instantiated on Mercury.

4.1 Common attack injection methodology

In this section we will explain shortly the main attacks performed in our analysis. We have to explain, that, for develop an analysis, we have to follow three steps:

- Search for the potentials injection points, commonly know as attack surface[40] , once that we know the target vulnerable entries, we can start with the injection planning.
- Injection the selected attack, on the previous vulnerable entry target.
- Once the attack is running or has ended, we have to monitor the output, of the attacked device. Sometimes, the information what we get is confusing, so we need to format, to make easier the reading.

4.2 Attacks supported by Mercury

At this sub-chapter we will explain the steps to perform an attack, we will see how to get information, how we can use that information against that app, and what we can expect from that attack, all of these steps are common for all attacks.

4.2.1 Malformed Intents

For the Intents launch, we have to locate the path of the exported activity, for example, *com.android.browser*, the second is to know if that Intent accepts fields, for example, *--data-uri web www.upv.es*, it's important too to know the type of these fields, because it can be, that a field is a String or an Integer, we can know that if we are the programmer of the activity or we have the documentation of the application, if not, it can be a hard problem, to discover that. So we have focused on well know apps, like browser, search or similar. We have to emphasize that Mercury is a tool that is still in development, and we don't have access to all of the features. In these fields we have made changes, for example inserting an Integer in a String field or vice versa, we have inserted long numbers, negative numbers, large and short values, we have launched Intents with empty fields, etc.

4.2.1.1 Surface attack

First we have to get information about the target app. At the *figure 14* we have the exported modules at the package `com.android.browser`, we can see the exported components by the activity, we can read that, it has six, `BrowserActivity`, `ShortcutActivity`, `BrowserPreferencesPage`, `BookmarkSearch`, `AddBookmarkPage` and `Widget.BookmarkWidgetConfigure`. We can send intents to these components to launch activities, and we can send parameters too, and test the activity. Unfortunately, we can't know, which fields the activity waits for, if that activity is not ours, or we don't have the documentation.

```
$ mercury console connect f75640f67144d9a3
Mercury Console
mercury> run app.activity.info --package com.android.browser
Package: com.android.browser
  com.android.browser.BrowserActivity
  com.android.browser.ShortcutActivity
  com.android.browser.BrowserPreferencesPage
  com.android.browser.BookmarkSearch
  com.android.browser.AddBookmarkPage
  com.android.browser.widget.BookmarkWidgetConfigure
mercury> □
```

Figure 14: Exported activities

4.2.1.2 Injections

We can launch explicit intents or implicit intents, as we can see at the *figure 15*

```
run app.activity.start
  --component
    com.android.browser
    com.android.browser.BrowserActivity
  --flag ACTIVITY_NEW_TASK
  --data-uri http://google.com/
```

```
run app.activity.start
  --action android.intent.action.VIEW
  --data-uri http://google.com/
```

Figure 15: Implicit and explicit

At the following point we have an example of intent launch.

- `--flags ACTIVITY_NEW_TASK`: here we introduce the parameters, we can introduce more than one, another examples of parameters, `--data-uri "tel:123"`, `--extra string search "searchedword"`

Due to the impossibility to know which fields an activity expects, if that application is not public or if we don't have the documentation, we have to keep to the public Intents[46], because the optional fields are well known. Or this reason we can't test market's app, we would like to test, for

example the top 10 downloaded apps, but due to this reason, we couldn't.

Here we will show some of the test that we have launched with Mercury:

```
run app.activity.start --action android.intent.action.SEARCH --flags ACTIVITY_NEW_TASK
```

First we are going to explain the order:

- `app.activity.start`: That is the Mercury order to launch intents
- `--action`: that indicate that the following string it's the exported module to which we will launch the intent.

The next step, is to launch a series of intents to a determined activities, modifying the parameter, like change the type, sending large values, etc and expecting the results.

Due to the Mercury's limitations and the impossibility of know the parameters of the private applications we had to follow to the Android's developers Intent classification page, for these reasons we have obtained few results. We have to explain, that Mercury is a tool in continuous development and still is not finished, for this reason, Mercury is not full developed and we don't have access to all of the features that we need, for example we can't use all of the types in the intent fields.

4.2.1.3 Observable output

With the malformed intents we can expect an abnormal behaviour form the target app, maybe because the programmer of that app, doesn't check the fields. We can see an abort of the target app. In resume we have to wait the answer of the target app, to our malformed intents injection. At the *figure 16* we can see an example of bad response to a malformed intent.

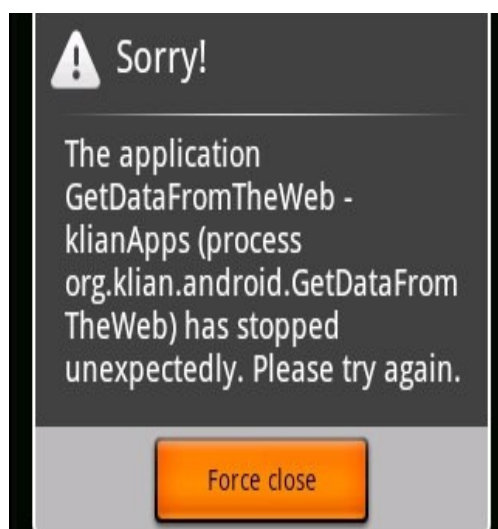


Figure 16: Force close

4.2.2 Broadcasts

Here due to the lack of information about the exported broadcasts launchers and the broadcasts listeners, we have to adjust to the well know broadcasts, like the system broadcasts. In that case we have to know the Intent and the fields that accepts and the type, (if any), once that we have that data, we just have to launch broadcasts changing the content of the fields and introducing wrong data in these fields, changing the type of them, wrong values, once we have launch the broadcasts, we have to wait to the mobile device behaviour, some times we have an output, because any application is waiting for that broadcast other times we don't have any response, because no application is listening for that broadcast.

4.2.2.1 Attack surface

We have to adjust to the system broadcasts(we have made a classification from them at annex), because we can't know the broadcasts that an app waits for, unless we have the documentation of the app, and the fields (if any) that an app waits for, so for the cause we have adjusted to the system defined broadcasts. So we have adjusted to the Android developers broadcast table. At the *figure 17* we can see that the sieve app, it doesn't have a broadcast. This is a way to know, which applications have broadcasts, but we can't know how do they work.

```
dz> run app.package.attacksurface com.mwr.example.sieve
Attack Surface:
 3 activities exported
 0 broadcast receivers exported
 2 content providers exported
 2 services exported
 is debuggable
```

Figure 17: Providers search

4.2.2.2 Injections

We have launched a series of system broadcasts, It's so difficult to know what is the influence of a broadcast in an app, may be it's not affected, or may be the application stops their work, but is so difficult to know that, if we don't have the app documentation. For this purpose we have programmed a simple app, that reply only to a broadcast that we have registered, the response of our app, is just a message shown at the device screen. At the following line we have an example of one broadcast launch.

```
mercury> run app.broadcast.send --action com.upv.intent0 --extra string parameter "string to display"
```


4.2.2.3 Observable output

Due to the nature of the broadcasts, it's very complicated to know, if any app is waiting for that broadcast, and if our launched broadcast affects any app, at the *figure 18* here we have an example of a broadcast's launch response. Maybe an app waits for a specific broadcast to disconnect himself from the internet, to consume less battery, but it's very difficult to know if an app has changed his behaviour due to a broadcast.

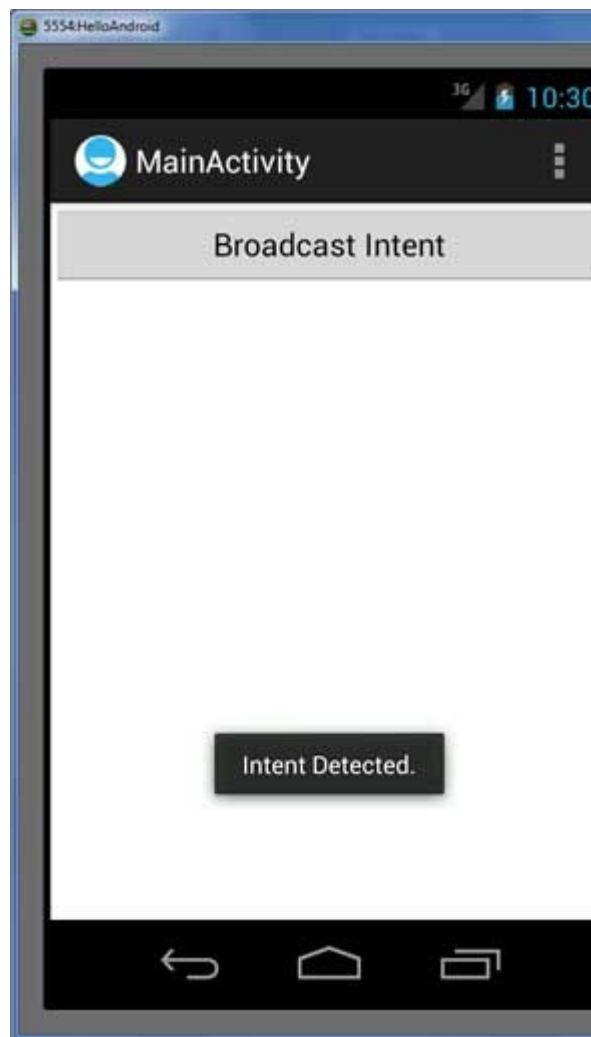


Figure 18: Broadcast response

4.2.3 SQL Injection

For that type of attack first we have to know if we have vulnerable content providers, this is an easy task, Mercury can do all the work, one we know the vulnerable content providers only remains to launch the SQL injection, if they are vulnerable we can ask to the table about the information, we can know the orders that made the table and the type of the fields. Test the content providers is easy just typing the SQL query, but we can also, insert into the table, and we can test the consistency of the table introducing wrong types into the table, unfortunately we can't test all the features due to Mercury development.

4.2.3.1 Attack surface

To test a Content provider must first explore whether there are and how they are configured, Mercury offers us an interesting tool, Webcontentresolver, this starts a service on port 8080 in our own machine, which offers a complete listing of all Content Providers of our device.

Package	Authorities	Read permission	Write permission
com.android.bluetooth	com.android.bluetooth.opp /btopp	null android.permission.ACCESS_BLUETOOTH_SHARE	null android.permission.ACCESS_BLUETOOTH_S
com.android.browser	com.android.browser browser /bookmarks/search_suggest_query	com.android.browser.permission.READ_HISTORY_BOOKMARKS android.permission.GLOBAL_SEARCH	com.android.browser.permission.WRITE_HI null
com.android.browser	com.android.browser.home	com.android.browser.permission.READ_HISTORY_BOOKMARKS	null
com.android.browser	com.android.browser.snapshots	null	null
com.android.calendar	com.android.calendar.CalendarRecentSuggestionsProvider	null	null
com.android.deskclock	com.android.deskclock	null	null
com.android.email	com.android.email.attachmentprovider	com.android.email.permission.READ_ATTACHMENT	null
com.android.email	com.android.email.provider com.android.email.notifier	com.android.email.permission.ACCESS_PROVIDER	com.android.email.permission.ACCESS_PRO
com.android.exchange	com.android.exchange.directory.provider	android.permission.READ_CONTACTS	null
com.android.gallery3d	com.google.android.gallery3d.provider	null	null
com.android.gallery3d	com.google.android.gallery3d.GooglePhotoProvider	null	null
com.android.htmlviewer	com.android.htmlfileprovider	null	null
com.android.mms	com.android.mms.SuggestionsProvider /search_suggest_query /search_suggest_shortcut	android.permission.READ_SMS android.permission.GLOBAL_SEARCH android.permission.GLOBAL_SEARCH	null null null
com.android.mms	mms_temp_file	null	null
com.android.mms	com.android.cm.mms	null	null
com.android.phone	icc	android.permission.READ_CONTACTS	android.permission.WRITE_CONTACTS
com.android.providers.applications	applications	null	null
com.android.providers.calendar	com.android.calendar	android.permission.READ_CALENDAR	android.permission.WRITE_CALENDAR
com.android.providers.contacts	contacts com.android.contacts /search_suggest_query /search_suggest_shortcut /contacts/*/photo	android.permission.READ_CONTACTS android.permission.GLOBAL_SEARCH android.permission.GLOBAL_SEARCH android.permission.GLOBAL_SEARCH	android.permission.WRITE_CONTACTS null null null
com.android.providers.contacts	call_log	android.permission.READ_CONTACTS	android.permission.WRITE_CONTACTS
com.android.providers.contacts	com.android.voicemail	com.android.voicemail.permission.ADD_VOICEMAIL	com.android.voicemail.permission.ADD_VOI
com.android.providers.contacts	com.android.social	android.permission.READ_CONTACTS	android.permission.WRITE_CONTACTS
com.android.providers.downloads	downloads /my_downloads /all_downloads /download	null android.permission.INTERNET android.permission.ACCESS_ALL_DOWNLOADS android.permission.INTERNET	null android.permission.INTERNET android.permission.ACCESS_ALL_DOWNLO android.permission.INTERNET
com.android.providers.drm	drm	null	null
com.android.providers.media	media	null	null
com.android.providers.settings	settings	null	android.permission.WRITE_SETTINGS
com.android.providers.telephony	telephony	null	null
com.android.providers.telephony	sms	android.permission.READ_SMS	android.permission.WRITE_SMS
com.android.providers.telephony	mms	android.permission.READ_SMS	android.permission.WRITE_SMS
com.android.providers.telephony	mms-sms	android.permission.READ_SMS	android.permission.WRITE_SMS

Figure 19: Web content resolver

At the *figure 19*, we can see a the Webcontentresolver list of content providers installed in our device, we can see their authority and their permissions (R / W,) we are looking for content providers, having either or both permits R/ W to null, allowing us to interrogate the database and write on it. For this same information can also use the order, *scanner.provider.injection*, and Mercury will return a list of Content providers vulnerable to SQL injection, here an example of a Samsung mobile:

```
Archivo Editar Ver Buscar Terminal Ayuda
content://com.google.android.apps.uploader/uploads
content://com.google.photos.provider.Album/albums
content://telephony/carriers/preferapn

Injection in Selection:
content://com.android.bluetooth.opp/live_folders/received
content://com.google.android.maps.TrafficAppWidgetProvider/
content://com.android.cm.mms/
content://com.tmobile.thememanager.themes/themes
content://media/external/video/media
content://com.android.cm.mms/templates
content://com.google.photos.provider.Album/
content://com.google.android.maps.LayerInfoProvider/
content://media/external/audio/media/
content://com.google.android.maps.TrafficAppWidgetProvider/appwidgets
content://com.android.bluetooth.opp/btopp
content://com.google.android.maps.LayerInfoProvider
content://media/external/audio/artists/
content://media/external/audio/albums/
content://com.google.settings/partner
content://com.google.android.apps.uploader/uploads
content://com.google.photos.provider.Album/albums
content://telephony/carriers/preferapn
mercury>
```

Figure 20: vulnerable contents provider

At the *figure 20* we have the output of the vulnerable content providers command, that command searches in all the providers the vulnerable ones, looking for the read and write permissions set to null.

4.2.3.2 Injections

Once we have Content providers vulnerable, we can test them, here an example:

```
run app.provider.query content://telephony/carriers/preferapn --vertical
```

and in the *table 3* we have the results of the previous order:

_id	71
name	Yoigo
Numeric	21404
mcc	214
mnc	04
apn	internet
user	null
User	null
server	null
password	null
proxy	010.008.000.036
port	8080
mmsproxy	null
mmsport	null
mmsc	null
authtype	-1
type	Default, supl
current	1
Protocol	IP
roaming_protocol	IP
carrier_enabled	1
bearer	0

Table 3: Results

Here we can find a table with multiple fields, `_id` (this field is common on all tables), `name`, `numeric`, `user` and much more, we can find that the last sim card was from Yoigo.

We can know too, internal information of the table, as the SQL statements that created it, the type of fields:

```
mercury> run app.provider.query content://telephony/carriers/preferapn --projection "*" from
sqlite_master--" --vertical
```

```
type table
```

```
name android_metadata
```

```
tbl_name android_metadata
```

```
rootpage 3
```

```
sql CREATE TABLE android_metadata (locale TEXT)
```

```
type table
```

```
name carriers
```

```
tbl_name carriers
```

```
rootpage 4
```

```
sql CREATE TABLE carriers(_id INTEGER PRIMARY KEY,name TEXT,numeric TEXT,mcc
TEXT,mnc TEXT,apn TEXT,user TEXT,server TEXT,password TEXT,proxy TEXT,port
TEXT,mmsproxy TEXT,mmsport TEXT,mmsc TEXT,authype INTEGER,type TEXT,current
INTEGER,protocol TEXT,roaming_protocol TEXT,carrier_enabled BOOLEAN,bearer INTEGER)
```

So far, we have obtained information from the contents of the table, we extracted additional information from the table, what's left? trying to insert data into the table, let's see. We will insert information into the content provider `//com.cyanogenmod.cmparts.provider.Settings/settings`

```
mercury> run app.provider.insert content://com.cyanogenmod.cmparts.provider.Settings/settings
--integer _id 1 --string key nth --integer value 10
```

In the above order, we have to indicate the full address of the content provider, we also have to indicate the names of the fields, their type and value, then we execute the order and if Mercury responds with *done*, it means that the insertion was successful.

```
mercury> run app.provider.query content://com.cyanogenmod.cmparts.provider.Settings/settings
```

```
| _id | key | value |
```

```
| 1 | nth | 10 |
```

In the above command we can see that the information is properly inserted.

After these experiments, we can see that we can access sensitive information, this app is capable without having any special permission to access all content providers that are not properly

configured, we must to remember that Mercury only requires the *Intnernet* permission.

4.2.3.3 Observable output

Here we can observe the access to the data base, we can see the content of the tables that store the information of the app, may be that information it can be sensitive, for example we can access to a password organizer, or to our emails. At the *figure 21* we can see an example of output from a SQL injection.

```
dz> run app.provider.query
content://com.mwr.example.sieve.DBContentProvider/Passwords/ --projection "*"
FROM SQLITE MASTER WHERE type='table';--"
| type | name | tbl_name | rootpage | sql |
| table | android metadata | android metadata | 3 | CREATE TABLE ... |
| table | Passwords | Passwords | 4 | CREATE TABLE ... |
| table | Key | Key | 5 | CREATE TABLE ... |
```

Figure 21: SQL output example

4.2.4 Command Execution

In this attack we only can launch a shell or a single command, here we can test all the command that Busybox offers, so we can test the access to the file-system directory, try to copy files and modify them with the *vi* or *vim* program, offered by Busybox. We can test the access to secure folders like */etc*, */mnt*, access to the SD card (if any).

4.2.4.1 Attack surface

We have to explain first that, if we want to launch commands like in a normal Linux OS, first we have to install the Busybox, because we will launch the orders that Busybox allows us, we don't need anything more, we don't need to find nothing, but with Mercury we can install BussyBox without the permission of the user, just we have to put a simple order like that:

```
Mercury> run tools.setup.busybox
```

And the user will never know that we have installed a new application on her mobile device, so yes it's possible to install an app with the knowledge of the user, a disturbing fact

4.2.4.2 Injections

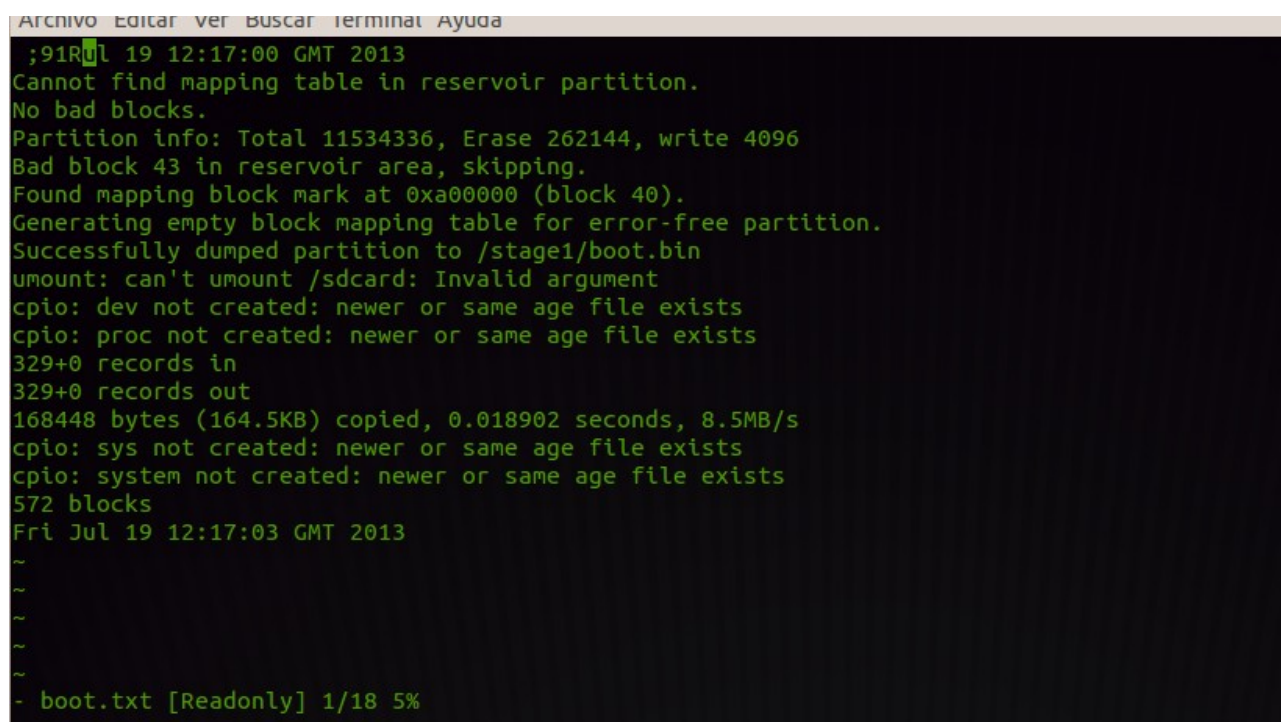
We can open a shell through Mercury and we can navigate across the file-system, as we can see on the below figure we can reach the *mnt/* file, for example we can copy all the content of that folder to a local path. We have an example at the *figure 22*, here we can see the output of a *ls* order.

```
data
datadata
default.prop
dev
efs
emmc
etc
init
init.aries.gps.rc
init.aries.rc
init.aries.usb.rc
init.goldfish.rc
init.rc
lpm.rc
mnt
proc
radio
sbin
sdcard
sys
system
ueventd.aries.rc
ueventd.goldfish.rc
ueventd.rc
vendor
app_76@android:/ $ cd mnt
app_76@android:/mnt $ ls
asec
emmc
obb
sdcard
secure
app_76@android:/mnt $
```

Figure 22: Launching a shell

4.2.4.3 Observable output

As a result of an vi command, we can see the content of the files, here we have an example from the boot.txt as we can see at the *figure 23*.



```
Archivo Editar Ver Buscar Terminal Ayuda
;91Rl 19 12:17:00 GMT 2013
Cannot find mapping table in reservoir partition.
No bad blocks.
Partition info: Total 11534336, Erase 262144, write 4096
Bad block 43 in reservoir area, skipping.
Found mapping block mark at 0xa00000 (block 40).
Generating empty block mapping table for error-free partition.
Successfully dumped partition to /stage1/boot.bin
umount: can't umount /sdcard: Invalid argument
cpio: dev not created: newer or same age file exists
cpio: proc not created: newer or same age file exists
329+0 records in
329+0 records out
168448 bytes (164.5KB) copied, 0.018902 seconds, 8.5MB/s
cpio: sys not created: newer or same age file exists
cpio: system not created: newer or same age file exists
572 blocks
Fri Jul 19 12:17:03 GMT 2013
~
~
~
~
~
- boot.txt [ReadOnly] 1/18 5%
```

Figure 23: Content of the file boot.txt

The most striking is the ease with which we can access the system and start throwing orders as if we were in a normal system, we don't have to forget that we are in a mobile device, and we are subject to Busybox availability.

4.3 Summary

In that chapter we have explained the sequence of steps to find vulnerabilities in an Android mobile device, we have explained in a simply way, the main vulnerabilities, the attack types that we will use in following steps, we have seen the configuration of the experiments. We have explained how to make the attacks, we have seen some examples and a few screen captures from the Mercury's output. In the next chapter we will explain the case of study, the target mobile devices, we will explain more deeply the attacks.

5 Case study

In this chapter we are going to explain how we did the experiments that, we have been developed, defining the targets devices, the experiment objectives, how we developed the experiments and the measures.

In this case, we will develop or experiences on a Linux environment, specifically in Ubuntu 12.10, but it also could be done in MacOS and Windows. We will use the Mercury version 2.2.1.

To prepare the experiment, first we have to prepare the working environment. We must first have a device that can launch the experiments, we can ask whether working in a real or emulated, for the first option we should have available any phone, better if is a real device, we will get better results. We can help with the emulator, if we install the Android Software Development Kit (Android SDK), with this tool we can create virtual devices with any version of Android. With the emulator gives us great versatility in being able to choose which version to realize. We can choose the emulator, it's a safe environment if we use unsafe or malicious apps and if anything goes wrong we can return to safe. But we have few cons, sometimes is very slow and is not an example of device that we can find in the real world. On the other hand, we can choose a real device, is faster and easy to monitor, we have a real usage example and we have full control if the device is rooted, but it can be damaged if we test malicious apps. We will choose the real device, because we have at our grasp, several devices, so we will use them.

We should install Mercury, in our computer, is simple, we just get off their latest version and follow some easy steps, additionally, for Mercury's correctly work we need to install Python. In addition we have to install the agent (the client) in our target mobile device, is as easy as install a normal app, we can do it thought the ADB. Before connect Mercury to the agent, we have to start it. Once installed all the need, simply we have to type in a terminal *mercury console connect*.

We have a couple of models of connection between the client(Mercury app, con the mobile device) and the server(Mercury on the computer), we can connect via USB or via WIFI, we have decided for the first option, via USB, is much faster and requires less infrastructure to run.

5.1 Target Devices

For our tests we have used several devices, in the *table 4*, we can see them, including mobile phones and tablets. We have tested two HTC WildFire A3333, one Samsung Galaxy S and two Archos 7o internet tablet,

Brand	Model	Operative System	Version	
HTC	WildFire A3333	Android - HTC Sense	2.1	Model A
HTC	WildFire A3333	Cyanogen Mod	7.2	Model B
Samsung	Galaxy S	Cyanogen Mod	9.2	Model C
Archos	7o Internet Tablet	Android 2.2.1	2.4.8.3	Model D
Archos	7o Internet Tablet	Android 2.2.1	2.4.8.1	Model E

Table 4: target devices

5.2 Work load

Here we are going to explain the work load, the installed apps, services, and the overall info about the target mobile.

- Archos versions 2.4.8.1 and 2.4.8.3, installed apps:
 - Adobe reader, Aldiko, Angry birds, AppsLibs, calculadora, Búsqueda de google, cámara, CineShowTime, ColorNote, Deezer, eBuddy, FindYourWay, Fring, Galeria, HubkapMobile, Música, navegador, Quiclopedia, Relog, Skype, Tuteur, ViewPagerIndicator, Youtube, contactos.
- HTC house Rom:
 - Adobe reader, Aldiko, BusyBox, calculadora, cámara, carHome, descargas, DevTools, DroidSheep, DSPManager, email, filemanager, gamería, ManifestExplorer, sms, MercuryAgent, Música, navegador, noticias, playStore, radioFM, torch, videos, superuser, TestGCM, contactos.
- HTC Cyanogen V7:
 - Galeria, contactos, SMS, Mensajes, Música, mercuryAgent, radioFM, videos, torch, BusyBox, noticias, DSPManager, playStore, MovieStudio, taskManager, Shark, filemanager, TestGCM, ViewPagerIndicator
- Samsung Cyanogen V9:
 - Ajustes, Boox, Búsqueda, calculadora, calendario, cámara, contactos, correo, descargas,

DevTools, drone, DSPmanager, Eventivities, Findme, galeria, Gmail, geoEvents, Hikinapp, mercuryAgent, Mensajes, MovieStudio, Música, navigation, PepeRoutes, PersonalFinance, PlayStore, Principal, Relog, Shark, taskManager, Shamir, Shark, superuser, valenparty, Youtube.

5.3 Experiments configuration

Here we are going to explain our experiments in numbers:

Experiments done:

Fault	N° of injections	Positive results
Malformed intents	75	54
Broadcasts	50	43
SQL Injections	45	30
Command	30	23

5.3.1 Relevant parameters

Here we will explain the most relevant parameters that we have introduced in our injections

- Malformed intents:
 - Integers: Empty fields, out of range (like one million), negative, malformed numbers, numbers with letters, numbers with punctuation marks.
 - Strings: Empty fields, strings with numbers, strings with capital letters, strings with spaces, strings with punctuation marks, long strings,
- Broadcasts: we have introduced, wrong uris, wrong paths, we have inserted in the optional fields unexpected parameters, like empty fields, malformed strings and integers.
- SQL Injections: we have inserted malformed parameters, like wrong integers, malformed strings, very similar to the malformed entry.
- Command: Here is impossible to insert parameters.

5.4 Results

In this chapter we are going to introduce the results that we have found in our experiments. Results are obtained from performing 200 experiments, and from these, in 150 we found vulnerabilities. As we will present, the most vulnerable devices are those which have installed a cooked Roms like Cyanogen [42]. If we install a modified rom, we have more possibilities adapt our mobile device to ours requirement, but the more possibilities we have to modify, the more vulnerable our mobile will be. The most surprising vulnerability was found in Archos devices, we found that the OS answer to the CALL an DIAL Intents, when the Archos devices are tablets without dial options, they are not phones. Following the advices of Android developers, we have explained some advices to avoid some of the vulnerabilities exposed in the previous chapters. But we have to emphasize that no all of the vulnerabilities are avoidable.

Most software security vulnerabilities fall into one of a small set of categories:

- Invalidated input.
- access-control problems.
- Secure Storage and Encryption.
- Insecure File Operations.
- DoS.
- Bypass.

But, we will classify then in four categories, malformed intent, broadcasts, SQL Injection and command execution.

5.4.1 Malformed Intents

Here we will introduce the fails that we have observed when we launched the malformed intents. At the *table 5*, we have the access-control and invalidated input vulnerabilities.

Device models	DELETE(1)	WEB_SEARCH (2)	SEARCH(3)
Model A			
Model B	✓	✓	✓
Model C	✓	✓	✓
Model D			
Model E			

Table 5: Malformed Intents

Delete: an activity action, we can delete de given data from its container. Input required.

Web_search: an activity action, action required to perform a web search. Input required.

Search: an activity action, perform a search. Input required.

(1) We were able to crash, the delete feature when we didn't insert any string in the target field.

(2) We stop the execution of the de Google Search when we launch the system call WEB_SEARCH.

(3) We were able to stop, the search function, when we didn't insert anything to search, because we suppose, that field is necessary.

Here we face to a system problem, because we were able to access to a some system features, like *Delete*, *Web search*, *search*, is supposedly we could not be able to access to that functionalities so we have an access-control problems, but we can access to them, may be the system doest not check the id of the process that launch that calls, or these features are not market like only-system-access. We have too, an invalidated input problem, because that features doesn't check the fields that the intent call pass, we can have a string in a field that supposedly it must be an integer. It can be very difficult to know the source of these problems.

We can exploit these vulnerabilities, leaving empty the required fields or inserting the wrong type, we can stop that feature or may be, depending on the Android OS, we can reboot the system, reaching a DOS (denial of service), so that vulnerabilities can be very harmful for the user, because it's impossible to see why that is happening, is transparent.

5.4.2 Broadcasts

In the *table 6*, we have the access to System Broadcasts, in the first row we have the tested broadcasts that supposedly are only accessible by the OS. Here mainly, we have the access-control the DOS and the ByPass vulnerabilities.

Device models	DREAMING STARTED (2)	DREAMING STOPPED (2)	EXTERNAL APPLICATIONS AVAILABLE(1) (2)	PACKAGE FIRST LAUNCH (2)	PACKAGE VERIFIED (2)	MANAGE NETWORK USAGE (2)	POWER USAGE SUMMARY (2)	MY PACKAGE REPLACED (2)	(*)
Model A	✓								
Model B	✓	✓	✓	✓		✓		✓	
Model C	✓	✓	✓	✓	✓	✓	✓		✓
Model D(**)	✓			✓	✓	✓	✓	✓	
Model E(**)	✓			✓	✓	✓	✓	✓	

Table 6: Access control problems

DREAMING_STARTED: Broadcast Action, Sent after the system starts dreaming.

DREAMING_STOPPED: Broadcast Action, Sent after the system stops dreaming.

EXTERNAL_APPLICATIONS_AVAILABLE: Broadcast Action, Resources for a set of packages (which were previously unavailable) are currently available since the media on which they exist is available.

PACKAGE_FIRST_LAUNCH: Broadcast Action, Sent to the installer package of an application when that application is first launched (that is the first time it is moved out of the stopped state). The data contains the name of the package.

PACKAGE_VERIFIED: Broadcast Action, Sent to the system package verifier when a package is verified. The data contains the package URI.

MANAGE_NETWORK_USAGE: Activity Action, Show settings for managing network data usage of a specific application. Applications should define an activity that offers options to control data usage.

POWER_USAGE_SUMMARY: Activity Action, Show power usage information to the user.

MY_PACKAGE_REPLACED: Broadcast Action, A new version of your application has been installed over an existing one. This is only sent to the application that was replaced. It does not contain any additional data; to receive it, just use an intent filter for this action.

(*) Access control vulnerability access to non specific functionality of the application DIAL

(**) We found too, that the Archos models answer to the CALL and DIAL broadcast, when these models don't have the phone options.

(1) Deny of service when we launch the broadcast EXTERNAL_APPLICATIONS_AVAILABLE we stop the apps Trebuchet[47] and Google Play.

(2) With these attacks, we didn't get any remarkable response, just we were able to launch that OS broadcasts.

Here we have the broadcasts that, supposedly are accessible only by the OS, we can see a ByPass vulnerability, so we found that many of these broadcasts that are accessible by any app, this discovery is disturbing, because this an Operative System fail, is not a vulnerability inserted by a programmer mistake or oversight, is an error from the own system, so the problem is worst than we thought. We can exploit, mainly that vulnerability, sending a series of broadcasts hoping to change the behaviour of the applications, that, are currently running in the system, some of them, will not take care of these broadcasts, but others will be listening and waiting for that broadcast, and will change their performance. Unfortunately we can't make anything against that problem, the origin of this is in the own system we hope, that problem will be solved in coming versions of Android.

5.4.3 SQL Injection

At the *table 7*, we have the content providers unauthorized access vulnerabilities and Secure Storage and Encryption vulnerabilities, at the first row we have the content providers that we were able to access, extract and modify information.

Models	Apn provider (1)	Settings provider (1)	phone contacts (1)	com.google.settings/partner (1)	control device secret codes (1)	com.google.android.maps.LayerInfo Provider (1)	com.android.cm.mms (1)(2)	com.android.cm.mms/templates (1)(2)
Model A	✓	✓			✓			
Model B	✓	✓	✓	✓	✓	✓	✓	✓
Model C	✓	✓	✓	✓	✓	✓	✓	✓
Model D	✓	✓			✓			
Model E	✓	✓			✓			

Table 7: Secure storage and encryption

Apn provider: Is the name of the gateway between the 3G, etc. mobile network and another computer network.

Settings provider: Here we can find the configuration of the 3G network, we can find, the name of the providers and several settings, (frequency, channel, etc.).

Phone contacts: The stored contacts on our mobile device.

com.google.settings/partner: Here we can find information about the settings from google applications, like the mail or similar.

Control device secret codes: Here we can find the factory secret codes, to access, for example, Wireless LAN tests, System dump mode, Immediate backup of all media files, etc.

com.google.android.maps.LayerInfoProvider: We have all the information about the given layer from Google Maps.

com.android.cm.mms: The mms, (multi-media messages), stored in our mobile phone.

com.android.cm.mms/templates: The mms templates stored in our mobile phone.

(1) We were able to access to the content of the tables, and to the SQL instructions to build the table.

(2) We were able to read and write on the content provider.

Here we have the content providers (similar to a table, from a database) that, are vulnerable to the access to their own data, we have to clear up that, an application can access to their own content providers or the content providers from others applications that have authorized it, but here we have that we are able to access, nearly to all of the content providers that have some mistakes on their configuration, if we don't want to make that mistake, we have to configure correctly the content provider permissions, in particular, write and read, if we don't make anything, these permissions will be set at null, and that is our gate to access to the contents. We can read and/or write, depending on the condition of the previous permissions. This is a big fail, mainly, from the programmers, sometimes they forget to set that parameters, with that mistake we are able to access to the data of others applications, may be, the mail, the sms, or the configuration of the wireless network, and of course, we can write too, we can insert new information on the content providers, we can test the integrity of the tables, because we can insert incorrect data, and monitor the behaviour of the applications when that access to that corrupt data, and perhaps, we can stop the application or restart the mobile device, or left the table in a weak state.

5.4.4 Command execution

In the *table 8* we have the access to file-system and bypass vulnerabilities, we were able to access to the whole SDcard file-system and to the device file-system and we were able to open a console on the target device.

Models	Total access to the SDcard file system	Access to the device file system	Code Execution (*)
Model A	✓	✓	✓
Model B	✓	✓	✓
Model C	✓	✓	✓
Model D	✓	✓	✓
Model E	✓	✓	✓

Table 8: Insecure file operations

(*) Code execution through the BusyBox app[71].

Through the busy box application we can access, almost, to the mail features of a Unix systems, like the most orders that we can find in a Linux command console. What will can do, if we can deploy all of these commands in an Android device (we don't have to forget, that is based on Linux), for example we can list the file-system directories or we can access to the SD-Card (if any) and we can copy that contents to our computer or send it thought Internet. We can extract the whole path to the file that we want to extract and with the Mercury order of copy files and we can download wherever we want.

First we have to say that, we are able to launch commands or launch because we have installed the Busy box app, that offers us all, almost, all features of a Unix system. We have to focus, that the system allows us to commands like in a computer, we know that Android runs over a Linux systems, but we are not in a computer, we are in a mobile device, we supposedly we could not access to some features, like list a system folder or see and modify systems files, we just a modify one of these files and turn the mobile device into an expensive brick.

5.4.5 Curious data

We have found some curious data in our experiments, we will show them here:

- In the Archos Internet tablets we found that the system allows us to launch a system broadcasts, like *android.intent.action.DREAMING_STOPPED*, *android.intent.action.PACKAGE_FIRST_LAUNCH*, *android.intent.action.PACKAGE_FULLY_REMOVED* and *android.intent.action.PACKAGE_VERIFIED*,
- In the Archos Internet tablets, we found that the system allows us to launch *android.intent.action.NEW_OUTGOING_CALL*, despite of that tables not have the function of call, we can't make a phone call at all.
- In the Samsung mobile, we have the same case as above, the system allows us to launch a system broadcasts, like *android.intent.action.DREAMING_STOPPED*, *android.intent.action.PACKAGE_FIRST_LAUNCH*, *android.intent.action.PACKAGE_FULLY_REMOVED* and *android.intent.action.PACKAGE_VERIFIED*
- In Samsung launching the broadcast *android.intent.action.EXTERNAL_APPLICATIONS_AVAILABLE*, we sopped the apps, **Trebuchet** and **Google Play**.
- In Samsung mobile, we stopped the Google search launching the *android.intent.action.WEB_SEARCH*, *without extra and with*, *android.intent.action.WEB_SEARCH --flags ACTIVITY_NEW_TASK --extra string search "wordtosearch"*
- In Samsung mobile, we can reboot the mobile inserting the manufacturer code in the dial, as follows ***##947322243*##**

- In Samsung mobile if we introduce the manufacturer code `***#4636***` in the dial, we can know the user and password from the connected wireless network as we can see at the *figure 24*.

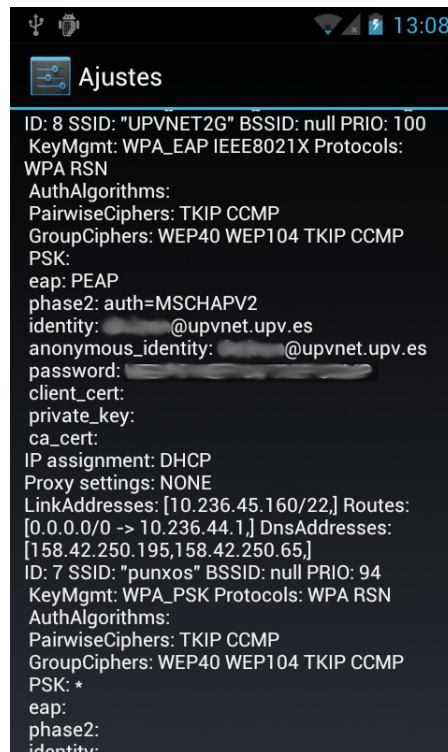


Figure 24: User network information

- In both HTC mobiles we found the same response to the launch of the `system.broadcastsandroid.intent.action.DREAMING_STOPPED`, `android.intent.action.PACKAGE_FIRST_LAUNCH`, `android.intent.action.CKAGE_FULLY_REMOVED` and `android.intent.action.PACKAGE_VERIFIED`, the system allows us to launch them.
- In both HTC, have ben able to insert data in the content provider `media/external/images/media/`, this is a table that contains images, when we insert a new row in the table, that query needs a photo, so we didn't insert a real photo in the query, but the system inserted an empty photo, with a size of 0 Kb.

6 Tips and tricks

In this chapter we will explain some tips and tricks[41] to make safer our application, just following these advices we will improve the security of our mobile device.

Android has some security features included into the operating system, that significantly reduces the risks of that phone information will be compromised. To help with this task, you can build your app with default system file permissions and avoid difficult decisions about security, but it's better if we adapt the security to our requirements.

Here we present the most relevant core advices:

- With the Sandbox from our Android application, we are able to avoid the code execution from outside of our app, from others apps.
- An application framework with robust implementations of common security functionality such as cryptography, permissions, and secure IPC.
- Technologies like ASLR, NX, ProPolice, safe_iop, OpenBSD dmalloc, OpenBSD calloc, and Linux mmap_min_addr to reduce the risks associated with the most common memory errors.
- We can use an encrypted file-system, with that we will able to protect our, if our mobile device or stolen.
- We can use a system of permissions to restrict the access to system features.
- Application-defined permissions to control application data on a per-app basis.

Notwithstanding, it is very important to build a robust app, that we be familiar with the Android security best practices. If we follow these practices we will reduce the likelihood of inadvertently introducing security issues that could make could make easier to steal information from our mobile.

6.1 Storing data

One of the most important concern for Android is, if the data that we save on the mobile is accessible to other apps that shares the device. We have three ways to store our information on the device.

6.1.1 Using internal storage

By default, files that we create on internal storage are accessible only to your app. This protection is implemented by Android and is sufficient for most applications.

We should generally avoid using the `MODE_WORLD_WRITEABLE` or `MODE_WORLD_READABLE` modes for IPC files because they do not provide the ability to limit data access to particular applications, nor do they provide any control on data format. If we want to share your data with other app processes, we might instead consider using a content provider, which offers read and write permissions to other apps and can make dynamic permission grants on a case-by-case basis.

To provide additional protection for sensitive data, we might choose to encrypt local files using a key that is not directly accessible to the application. But we have some problems to solve before encrypt our data, for example, choose the correct data, where to store the key, which algorithm we will use, we have many problems to solve as we can see in [37] before choose the proper way.

6.1.2 Using external storage

Files created on external storage, such as SD Cards, are readable and writable by everybody. Because external storage can be removed by the user and also modified by any application, we should not store sensitive information using external storage.

As with data from any untrusted source, we should perform input validation when handling data from external storage. If your app does retrieve executable files from external storage, the files must be signed and cryptographically verified prior to dynamic loading, in order to prevent possible infection by a potential virus.

6.1.3 Using content providers

Content providers offer a structured storage mechanism that can be limited to your own application or exported to allow access by other applications. If we do not intend to provide other applications with access to our `ContentProvider`, mark them as `android:exported=false` in the application manifest. Otherwise, set the `android:exported` attribute "true" to allow other apps to access the stored data. But that last option it's not recommended to share information with other apps, unless we can trust on these apps.

When creating a `ContentProvider` that will be exported for use by other applications, we can specify a single permission for reading and writing, or distinct permissions for reading and writing

within the manifest. We recommend that you limit your permissions to those required to accomplish the task at hand. We have to keep in mind that it's usually easier to add permissions later to expose new functionality than it is to take them away and break existing users.

If we are using a content provider for sharing data between only your own apps, it is preferable to use the `android:protectionLevel` attribute set to "signature" protection. Signature permissions do not require user confirmation, so they provide a better user experience and more controlled access to the content provider data when the apps accessing the data are signed with the same key.

Content providers can also provide more granular access by declaring the `android:grantUriPermissions` attribute and using the `FLAG_GRANT_READ_URI_PERMISSION` and `FLAG_GRANT_WRITE_URI_PERMISSION` flags in the Intent object that activates the component. The scope of these permissions can be further limited by the `<grant-uri-permission element>`.

When accessing a content provider, use parametrized query methods such as `query()`, `update()`, and `delete()` to avoid potential SQL injection from untrusted sources. Note that using parametrized methods is not sufficient if the selection argument is built by concatenating user data prior to submitting it to the method.

Do not have a false sense of security about the write permission. Consider that the write permission allows SQL statements which make it possible for some data to be confirmed using creative WHERE clauses and parsing the results. For example, an attacker might probe for presence of a specific phone number in a call-log by modifying a row only if that phone number already exists. If the content provider data has predictable structure, the write permission may be equivalent to providing both reading and writing.

6.2 Using Permissions

Because Android sandboxes applications from each other, applications must explicitly share resources and data. They do this by declaring the permissions they need for additional capabilities not provided by the basic sandbox, including access to device features such as the camera.

6.2.1 Requesting Permissions

We recommend minimizing the number of permissions that your app requests not having access to sensitive permissions reduces the risk of inadvertently misusing those permissions, can improve user adoption, and makes your app less for attackers. Generally, if a permission is not

required for your app to function, do not request it.

In addition to requesting permissions, your application can use the `<permissions>` to protect IPC that is security sensitive and will be exposed to other applications, such as a `ContentProvider`. In general, we recommend using access controls other than user confirmed permissions where possible because permissions can be confusing for users.

Do not leak permission-protected data. This occurs when your app exposes data over IPC that is only available because it has a specific permission, but does not require that permission of any clients of its IPC interface. More details on the potential impacts, and frequency of this type of problem is provided in this research paper published at USENIX: http://www.cs.berkeley.edu/~afelt/felt_usenixsec2011.pdf

6.2.2 Creating Permissions

Generally, we should strive to define as few permissions as possible while satisfying your security requirements. Creating a new permission is relatively uncommon for most applications, because the system-defined permissions cover many situations. Where appropriate, perform access checks using existing permissions.

If we must create a new permission, consider whether you can accomplish your task with a "signature" protection level. Signature permissions are transparent to the user and only allow access by applications signed by the same developer as application performing the permission check.

If we create a permission with the "dangerous" protection level, there are a number of complexities that you need to consider:

- The permission must have a string that concisely expresses to a user the security decision they will be required to make.
- The permission string must be localized to many different languages.
- Users may choose not to install an application because a permission is confusing or perceived as risky.
- Applications may request the permission when the creator of the permission has not been installed.

Each of these poses a significant non-technical challenge for you as the developer while also confusing your users, which is why we discourage the use of the "dangerous" permission level.

6.3 Summary

As conclusion we can say that if we invest a little more time, in the facets that are related with the security and best practices we will increase the safety of our application. Also the final user must have attention when install app from non-GooglePlay apps because we don't know the origin of these apps, we can be installing applications do not do what they really say. As well we have to have care with the GooglePlay apps, above all we have to keep in mid the apps permissions, because most of them are over-privilege closer to the 30%[43], in may cases we can found apps (like a torch[44]), that demanded an access to the GPS, check the identity and phone status, access to the camera, network connections access, USB access, awesome for a torch, that only access to the flash.

As we can see at the previous tables, the most insecure mobiles it's the model C, the Samsung Galaxy S with Cyanogen Mod 9.2, why? The principal difference between that mobile with the others, it's the installed rom, in one hand we have the official manufacturers Roms and in the other hand, we have the unofficial roms, that are made by amateurs programmers, took from an official rom or a rom made by another amateur, and they start changing and modifying the behaviour of the system, opening new ways to improve the system, like having more options to customize our mobiles, add new functions offering new ways to modify for example the parameters of the CPU, but, who can ensure that this rom it doesn't have a service monitoring our device or a trojan? Opposite of that, we have the official roms, made by the mobile manufactures, in these roms we have few possibilities to customize the system or some times we can not un-install the pre-installed manufacturer apps. We don't have any chance to modify the behaviour of the system. May be we have a safer rom, but nobody can claim that, our manufactures has included a tracker or similar.

7 Conclusion and further work

Based on what we've seen in this master's thesis, we know the potential risks that exist in our mobile devices, we know that a malicious application can not require many special permits. We know, what we can do with only a couple of permissions, extract information from our SD card and send it to their servers, for example.

But how can we prevent the attacks against Android by these malicious apps? In fact there is not a perfect formula to protect us from these attacks, we can affirm that is better if we don't install any third-party apps, is better if we install apps that have passes the filter of Google Play market, although some of these apps are malicious, nothing is certain.

Unfortunately, the current anti-virus that we have available for Android systems do not solve these problems, they can not do anything against this kind of attacks. We believe that what we really need is to improve the current Android sandbox. The security of the applications made for Android, should come from the programmers themselves, a major improvement would be that development tools incorporate options for detecting potential security flaws, implementing mechanisms to protect against SQL injection, directory traversal vulnerabilities and maybe, improve the control access to the SD card, defining the limits, which can read and can not read, for example limit the app access to the SD card, giving permission, to read and write, to the own folder of the application.

Mercury has shown us how easy, it can be for an application to access to the sensitive data from your mobile device, with only, and we have to empathise that, the Internet permission, so what it can be happening when we install an app that requires the permission of “access to accounts”, the access to SD Card and the Internet, who can claim, that, our information is not being sent to a third-party company? therefore, or mobile is far away from begin secure.

Currently, the development and distribution of software for Android is much faster than the security improvements, unless we begin to focus on our security applications, or in improving the security model of the system itself, soon, we may be facing one scenario where most applications are malicious, and perhaps it can be the end of Android.

8 ITACA Research Day

We have displayed our work in the ITACA Research Day, with the reference, J. Mozos, D. de Andrés, J. Friginal, J.-C. Ruiz, "A study of vulnerabilities on Android systems: supporting tools and best practices", Jornada de investigación del IUI ITACA, Valencia, Spain, June 28, 2013.

Due to the size of the image, we prefer to take up one page, to make easier it's reading, we can see it at *figure 25*.

A study of vulnerabilities on Android systems: supporting tools and best practices

Javier Mozos¹, David de Andrés¹, Jesús Frigal² and Juan-Carlos Ruiz¹

¹Instituto ITACA, Universidad Politécnica de Valencia, Camino de Vera s/n, 46022, Valencia, Spain

²LAAS-CNRS, 7 Avenue du colonel Roche, 31400 Toulouse, France

Abstract



Over the past two decades, we have witnessed significant technology advances in mobile devices, from the PDAs of the late 1990s and early 2000s to the ubiquitous and multifunctional smartphones of today. These devices are, nowadays, an essential part of our everyday life, providing a myriad of useful applications (apps) for both work and leisure. However, this has also increased the attractiveness of these kind of platforms as a target for attackers, as mobile devices store lots of sensitive information, enable the access to company's private network or e-banking services, etc. In order to prevent the malicious exploitation of existing vulnerabilities, it is necessary to i) develop suitable tools to identify them, so they can be removed from existing apps, and ii) define a set of best practices to avoid introducing new vulnerabilities during development. This work studies existing tools and provides a first insight of lessons learned to define a set of best practices.



Technical description of the result

Top mobile vulnerabilities and exploits of 2012

- **Twitter SMS Spoofing:** attackers could post messages or alter the account settings of a Twitter user
- **Dirty USSD:** hackers may remotely reset and wipe phones running versions earlier than Android 4.1.x
- **Android SSL/TLS Woes:** 8 percent of Android apps could be vulnerable to man-in-the-middle attacks due to poor SSL/TLS implementations
- **Android NFC Vulnerabilities:** attackers may take over a device using another device placed within a few centimeters of the phone under attack
- **Mobile Man-in-the-Middle Attacks Using Exchange:** attackers may access contact entries, and impersonate a corporate email server and erase all of the data on the device through push commands
- **Social and Sharing Authentication Flaws:** Authentication keys in Facebook, LinkedIn, and Dropbox were stored in unencrypted plain text files
- **Zitmo:** Zitmo masquerades as a banking activation application and eavesdrops on SMS messages in search of the mobile transaction authentication numbers banks send via text to their users



Android-based security tools

Existing tools, like Mercury, Android Guard and ASEF, automate the analysis of apps.

- **Static analysis** focuses on scanning the internal resources of an app matching the results of the scan with pre-defined patterns assimilated to well-known vulnerabilities. These resources include i) the source code, where common programming errors could be checked, ii) the manifest, where undesired permissions may be granted to the app, and iii) internal resources used by the app, like local databases, images, multimedia files, etc.
- **Dynamic analysis** executes the app on a real device or simulator to determine its behaviour in presence of accidental faults or attacks. Accidental faults include comprise any non-malicious phenomena that can affect the device, like battery exhaustion, whereas attacks include malicious attempts to exploit existing vulnerabilities, like incorrect input validation (launching Intents with out of range parameters) or non-authorised access to stored data (via SQL injection),



Figure 2. Mercury
The heavy metal that poisoned the droid



Figure 3. Android Guard
Reverse engineering, malware and
goodware analysis of Android apps



Figure 4. ASEF
Android Security Evaluation Framework

Applications

Device-based mobile application vulnerability identification

- **Identify application permissions:** Which access privileges is the app granted
- **Map application functionality:** How apps access network connections, data storage, user inputs and permissions
- **Monitor connections:** Monitor requests and responses through all interfaces
- **Review data handling:** Where is the sensitive information located and how it is protected
- **Decompile application:** Look for patterns, symptoms, and common programming errors leading to vulnerabilities

```
mercury: provider
mercury#provider: info -p null
Package name: com.sec.android.app.callsetting
Authority: com.sec.android.app.callsetting.allcalls
Required Permission - Read: null
Required Permission - Write: null
Grant URI Permissions: false
Multiprocess allowed: false

mercury#provider: finduri com.sec.android.app.callsetting
/system/app/CallSetting.apk:
Contains no classes.dex
/system/app/CallSetting.odex:
content://com.sec.android.app.callsetting.allcalls/prefix_num
content://com.sec.android.app.callsetting.allcalls/reject_msg
content://com.sec.android.app.callsetting.allcalls/reject_num

mercury#provider: query content://com.sec.android.app.callsetting.allcalls/reject_msg
_id | reject_message | edit_checked
....
1 | Sorry, I'm busy, call me later | 2131165242
2 | I'm in a meeting | 2131165243
3 | Call me at 123456789 | 2131165244
4 | Night out at George's | 2131165245
5 | Cu | 2131165246
```

Figure 5. Mercury exploiting ill-defined permissions to disclose private information

Lesson learned to build more robust apps

- **Using internal storage:** Avoid MODE_WORLD_WRITEABLE/READABLE. Encrypt files with a key not directly accessible to the app.
- **Using external storage:** Do not put sensitive data there. Check inputs for incoming data and files should be signed and verified previous loading.
- **Using Content Providers:** Define read/write permissions and do not share data with others apps unless trusted. Use parameterised query methods to prevent SQL injection.
- **Requesting permissions:** Limit requested permissions to the minimum. Do not leak permission-protected data.
- **Creating permissions:** Create new permission only when necessary or use signatures instead.

Additional comments

This work has been performed as part of Mr. Mozos' M.S. thesis. This work is partially supported by the Spanish project ARENES (TIN2012-38308-C02-01), the ANR French project AMORES (ANR-11-INSE-010), the Intel Doctoral Student Honour Programme 2012.

[1] M. Cinque, D. Cotroneo and A. Testa, "A logging framework for the on-line failure analysis of Android smart phones", *1st European Workshop on Approaches to Mobile/Ubiquitous Resilience*, Sibiu, Romania, 2012.
 [2] A.K. Maji, F.A. Arshad, S. Bagchi, and J.S. Rellermeier, "An empirical study of the robustness of inter-component communication in Android", *International Conference on Dependable Systems and Networks*, Boston, USA, 2012.
 [3] Mercury, [Online] Available: <http://labs.mwrinfosecurity.com/tools/2012/03/16/mercury/>
 [4] AndroGuard, [Online] Available: <http://code.google.com/p/androguard/>
 [5] ASEF, [Online] Available: <http://code.google.com/p/asef/>

Figure 25: Itaca poster

9 References

- [1] MWR Labs main page: <http://labs.mwrinfosecurity.com> (April – 2013) created: (unknown) author: unknown
- [2]MWR Labs Twitter: <https://twitter.com/droidhg> (April – 2013) created: (unknown) author: unknown
- [3]Apk-tool :<https://code.google.com/p/android-apktool/> (April – 2013) created: (unknown) author: unknown
- [4] Tools Yard:
http://toolsyard.thehackernews.com/2012/09/mercury-v11-android-vulnerability.html#_ (April – 2013) created: (March 2012) author: Mohit Kumar
- [5] Infoseckeny, Mercury modules: http://infoseckeny.blogspot.com.es/2012_07_01_archive.html (April – 2013) created (July 2012) author Sam Kihahu
- [6][GitHub: <https://github.com/mwrlabs/mercury/wiki> (April - 2013)
- [7]Simon Roses Femerling Blog, Exploring malware:
<http://www.simonroses.com/2011/07/exploring-android-malware/> (April – 2013) created: (July 2011) created: (unknown) author: Simon Roses Femerling
- [8]RootWiki, APK multi:-tool <http://rootzwiki.com/topic/31506-linuxutilitytool-apk-multi-tool/> (March -2013) created: (August 2012) author: Raziel23x
- [9][Google Code: <http://code.google.com/p/android-apktool/> (March -2013) created: (unknown) author: unknown author: APK-tool community.
- [10]AndroGuard BlogSopt: <http://androguard.blogspot.com.es/2012/03/androguard-mercury.html> (May -2013) created: (March 2012) author: Anthony Desnos
- [11]Analizando malware en Android:
<http://www.expresionbinaria.com/analizando-malware-en-android-con-androguard/> (May -2013) created: (unknown) author: unknown
- [12]Google Code: <http://code.google.com/p/androguard/#Description> (May -2013) created: (March 2012) author: unknown
- [13]Analizando malware en Android con Androguard
: <http://solucionavirus.blogspot.com.es/2013/02/analizando-malware-en-android-con.html> (May -2013) created: (February 2013) author: unknown

- [14]CyberPunk Blog: <http://n0where.net/androguard-1-9/> (May -2013) created: (December 2012) author: unknown
- [15]Google Code: <http://code.google.com/p/androguard/wiki/Usage?ts=1337957819&updated=Usage#Androrisk> (May -2013) created: (unknown) author: unknown
- [16]Comdroid: <http://www.comdroid.org/> (May -2013) created: (unknown) author: unknown
- [17]Stowaway: <http://www.android-permissions.org/> (May -2013) created: (unknown) author: unknown
- [18]CCC congress <http://events.ccc.de/congress/2012/Fahrplan/events/5123.en.html> (May -2013) created: (December - 2012) author: CCC community.
- [19]Gogle Code: <http://code.google.com/p/smali/> (May -2013) created: (unknown) author: unknown
- [20]Tools Yard web:
<http://toolsyard.thehackernews.com/2012/08/asef-android-security-evaluation.html> (May -2013) created: (unknown) author: unknown
- [21]Google Code: <http://code.google.com/p/asef/> (May -2013) created: (unknown) author: ASEF community
- [22]Android permissions: <http://www.android-permissions.org/permissionmap.html> (May -2013) created: (unknown) author: ASEF unknown
- [23]iSecpartners: www.isecpartners.com/storage/tools/mobile-security/IntentFuzzer.zip (May -2013) created: (unknown) author: iSECpartners
- [24]Wikipedia: <http://es.wikipedia.org/wiki/USSD> (May -2013) created: (unknown) author: Wikipedia community
- [25]Paper:
https://engineering.purdue.edu/dcs/publications/papers/2012/IntercomponentCommunication_DSN_2012.pdf (May -2013) created: (unknown) authors: Amiya K. Maji, Fahad A. Arshad, Saurabh Bagchi
- [26] MyComputerPro, ¿Por qué Android es más vulnerable al malware?
: <http://www.muycomputerpro.com/2013/05/18/por-que-android-es-mas-vulnerable-al-malware/> (July-2013) created (May - 2013) author: Maria Gillarte
- [27] Most used Android versions:
<http://www.engadget.com/2013/03/05/android-4-usage-finally-overtakes-gingerbread/> (July-2013)

created (Mar 2013) author: Jon Fingas

[28] International Workshop on State in Java program analysis:

<http://www.monperrus.net/martin/bibtexbrowser.php?key=Bartel2012&bib=monperrus.bib>

(July-2013) created (January - 2013) authors: Alexandre Bartel, Jacques Klein and Yves Le Traon.

[29]Automatically Securing Permission-Based Software by

Reducing the Attack Surface: An Application to

Android :http://hal.archives-ouvertes.fr/docs/00/70/00/74/PDF/article_ASE12.pdf (July-2013)

created (May 2012) authors: Alexandre Bartel, Jacques Klein and Yves Le Traon.

[30] Improving Privacy on Android Smartphones Through In-Vivo Bytecode Instrumentation:

<http://hal.inria.fr/hal-00700319/> created (May 2012) authors: Alexandre Bartel, Jacques Klein, Martin Monperrus, Kevin Allix and Yves Le Traon.

[31]An Empirical Study of the Robustness of Inter-component Communication in Android:

https://engineering.purdue.edu/dcs/publications/papers/2012/IntercomponentCommunication_DSN2012.pdf created (unknown) authors: Amiya K. Maji, Fahad A. Arshad and Saurabh Bagchi.

[32] Google Android: A Comprehensive Security Assessment: <http://dl.acm.org/citation.cfm?id=1804131>

(June-2013) created (March - 2010) authors: Asaf Shabtai, Yuval Fledel, Uri Kanonov, Yuval Elovici, Shlomi Dolev and Chanan Glezer.

[33]Los antivirus para Android no valen para nada:

<http://www.computerworld.es/movilidad/los-antivirus-para-android-no-valen-para-nada> (June – 2013) created (2013) author: ComputerWolrd

[34] Santoku distribution: <https://santoku-linux.com/> (June – 2013) created (unknown) author: unknown

[36]packpub CISSP:Vulnerability and Penetration Testing for Access Control :

<http://www.packtpub.com/article/cisssp-vulnerability-and-penetration-testing-for-access-control>

(June – 2013) created (November - 2009) author: M.L. Srinivasan.

[37]VeraCode, Insecure Cryptographic Storage: <http://www.veracode.com/security/insecure-crypto>

(June – 2013) created (March - 2012) author: Fergal Glynn

[38]Wikipedia, Denial-of-service attack : https://en.wikipedia.org/wiki/Denial-of-service_attack

(June – 2013) created (unknown) author: Wikipedia community

[39]Cisco, Communication Encryption Bypass Vulnerability:

<http://tools.cisco.com/security/center/content/CiscoSecurityNotice/CVE-2013-1209> (June – 2013)

Cisco

[40] OWASP, Attack Surface:

https://www.owasp.org/index.php/Attack_Surface_Analysis_Cheat_Sheet (June – 2013) last modified (July - 2013) authors: Jim Bird and Jim Manico.

[41] Security tips: <http://developer.android.com/training/articles/security-tips.html> (June - 2013) created (unknown) author: unknown

[42]Cyanogen Mod web: <http://www.cyanogenmod.org/> (July - 2013) created (unknown) authors: Steve Kondik, Abhisek Devkota, Koushik Dutta, Benji Hertel, Keyan Mobli, Jef Oliver, Ricardo Cerqueira and Chris Soyars.

[43]Over privileged apps: <http://blog.fortify.com/blog/2011/12/16/> (July-2013) created (December - 2011) author: Yoneil

[44]Google Play, Over-privilege app: https://play.google.com/store/apps/details?id=com.surpax.ledflashlight.panel&feature=search_result#?t=W251bGwsMSwyLDEsImNvbS5zdXJwYXgubGVkZmxhc2hsaWdodC5wYW5lbcJd (July-2013) created (unknown) author: Surpax technology

[45]how to hack the Android sandbox with a game:<http://www.youtube.com/watch?v=WwS6oLcw6wQ> (July-2013) created (March - 2013) author: Joau Manuel

[46] Android developers, Standard Intents:

http://developer.android.com/reference/android/content/Intent.html#EXTRA_CC (July-2013) created (unknown) author: Android community.

[47]Trebuchet launcher:

<http://www.xatakandroid.com/productividad-herramientas/trebuchet-el-launcher-oficial-de-cyanogenmod-9-ya-se-encuentra-disponible> (July-2013) created (December - 2011) author: Juan carlos.

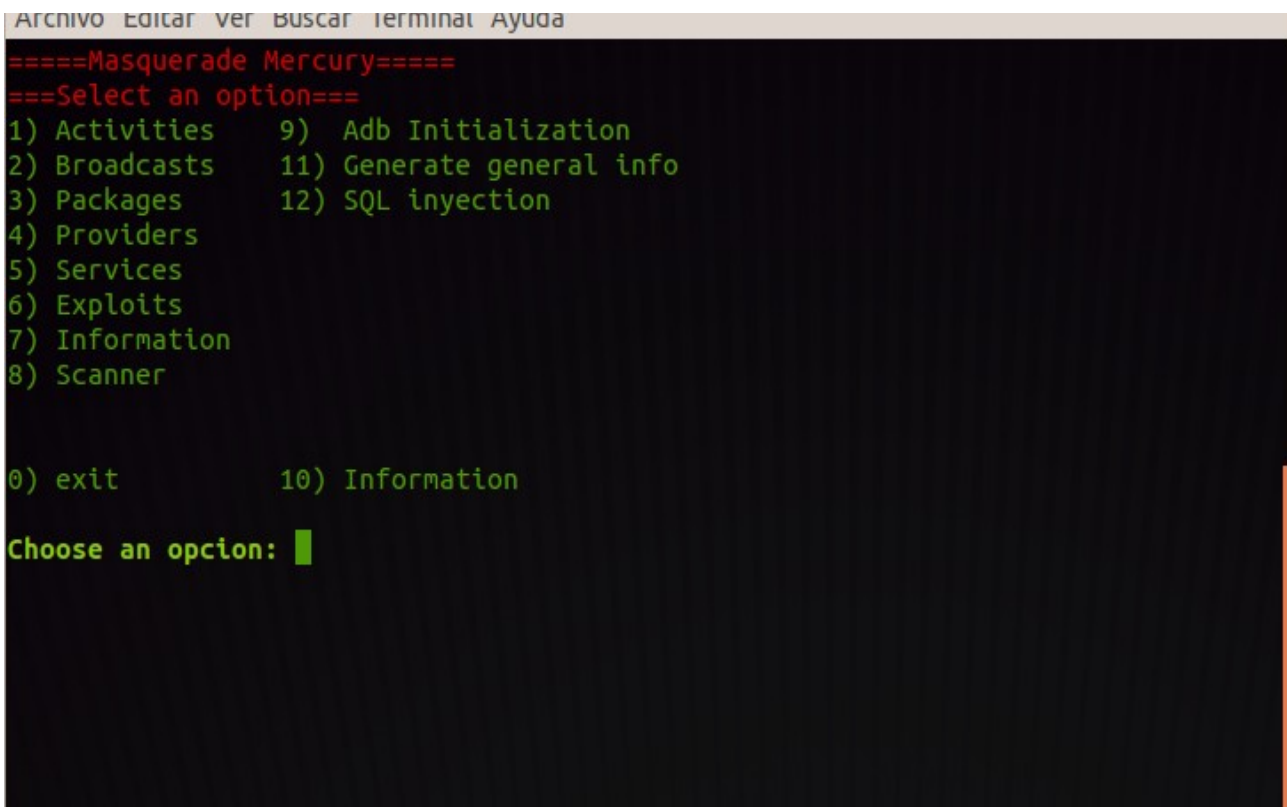
10 Annex

10.1 Automating the experiments: Masquerade

In this section we are going to explain how we automated a part of the experiments, we couldn't automate all, by Mercury's limitations and complexity.

10.1.1 Masquerade

Masquerade is what his name says, a mask on Mercury, we don't have to know how Mercury works, we can access to almost all features with a simple menu. We will take a look to all the sections at the *figure 26*.

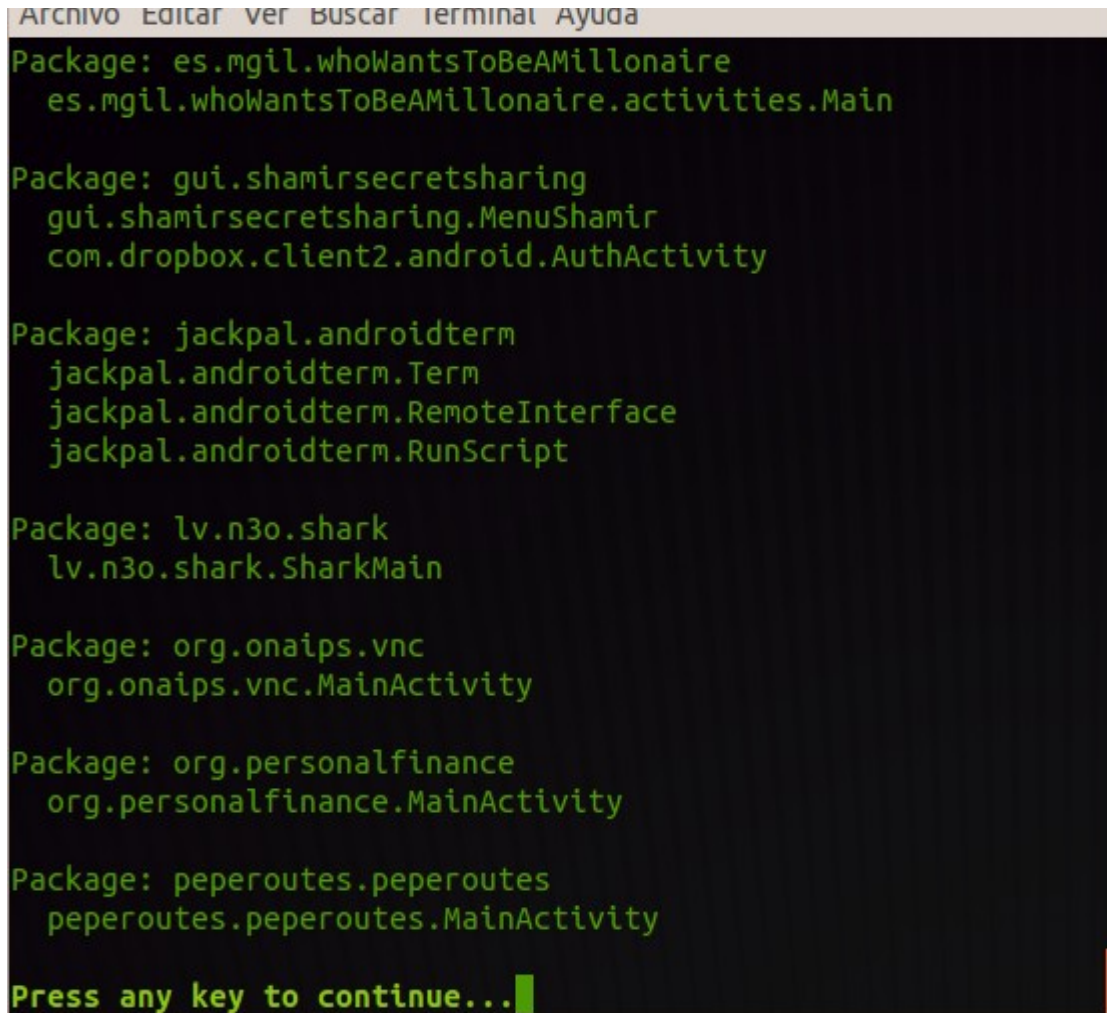
A screenshot of a terminal window showing the 'Masquerade Mercury' menu. The window title is 'Archivo Editar Ver Buscar Terminal Ayuda'. The menu text is as follows:

```
====Masquerade Mercury====  
===Select an option===  
1) Activities      9) Adb Initialization  
2) Broadcasts     11) Generate general info  
3) Packages       12) SQL inyection  
4) Providers  
5) Services  
6) Exploits  
7) Information  
8) Scanner  
  
0) exit           10) Information  
  
Choose an opcion: █
```

Figure 26: Masquerade menu

10.1.1.1 Activities

With the option *l) Activities*, we can find all the exported activities in our device, with the package name, at the *figure 27* we can see an output example, we can redirect the output to an txt file:



```
Archivo  Editar  Ver  Buscar  Terminal  Ayuda
Package: es.mgil.whoWantsToBeAMillionaire
       es.mgil.whoWantsToBeAMillionaire.activities.Main

Package: gui.shamirsecretsharing
       gui.shamirsecretsharing.MenuShamir
       com.dropbox.client2.android.AuthActivity

Package: jackpal.androidterm
       jackpal.androidterm.Term
       jackpal.androidterm.RemoteInterface
       jackpal.androidterm.RunScript

Package: lv.n3o.shark
       lv.n3o.shark.SharkMain

Package: org.onaips.vnc
       org.onaips.vnc.MainActivity

Package: org.personalfinance
       org.personalfinance.MainActivity

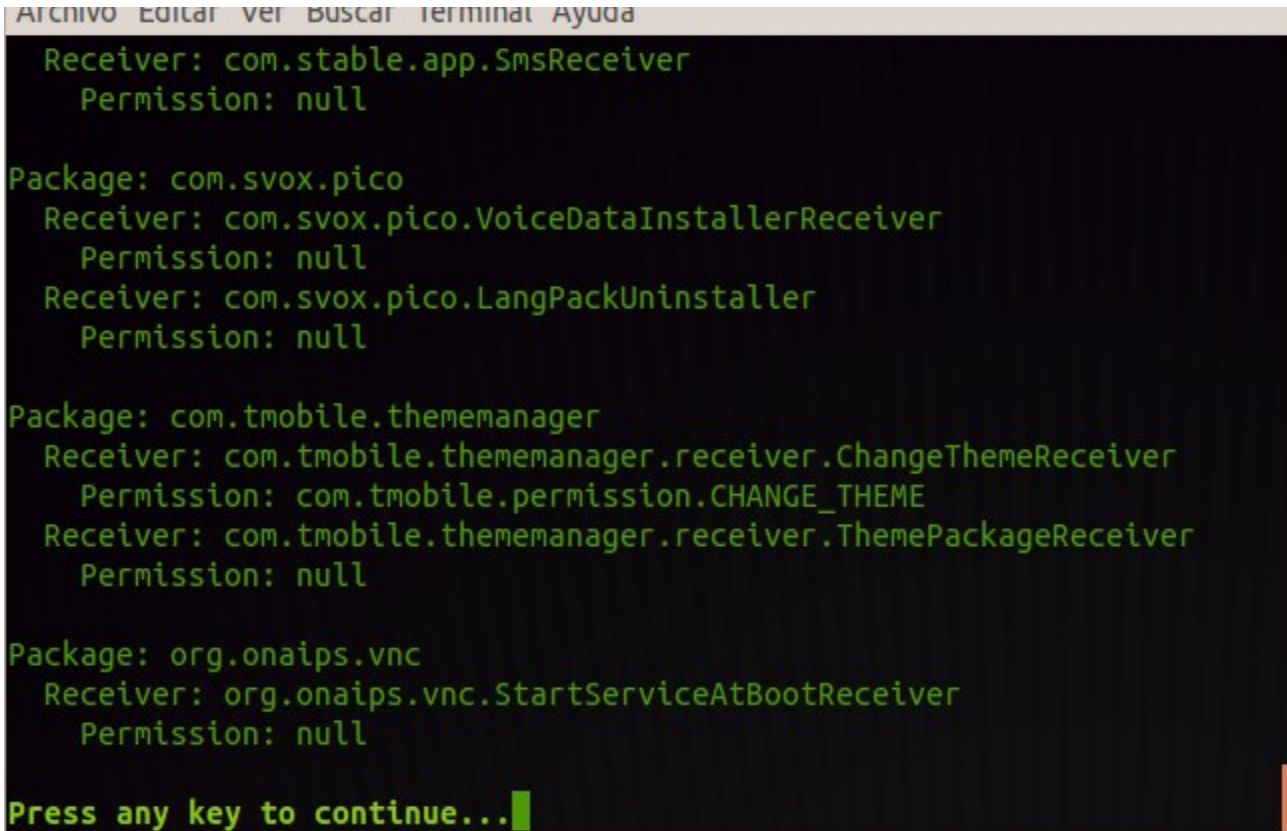
Package: peperoutes.peperoutes
       peperoutes.peperoutes.MainActivity

Press any key to continue...|
```

Figure 27: Activities output

10.1.1.2 Broadcasts

In this option we can, find all the exported broadcasts from the installed activities, we can redirect the output to an txt file. At the *figure 28* we can see an example of the output.



```
Archivo Editar Ver Buscar Terminal Ayuda
Receiver: com.stable.app.SmsReceiver
Permission: null

Package: com.svox.pico
Receiver: com.svox.pico.VoiceDataInstallerReceiver
Permission: null
Receiver: com.svox.pico.LangPackUninstaller
Permission: null

Package: com.tmobile.thememanager
Receiver: com.tmobile.thememanager.receiver.ChangeThemeReceiver
Permission: com.tmobile.permission.CHANGE_THEME
Receiver: com.tmobile.thememanager.receiver.ThemePackageReceiver
Permission: null

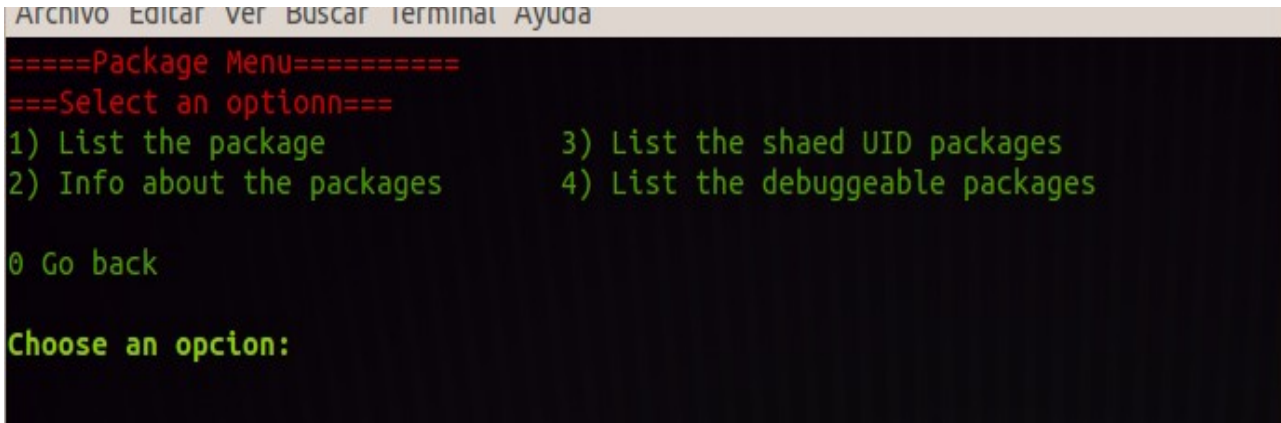
Package: org.onaips.vnc
Receiver: org.onaips.vnc.StartServiceAtBootReceiver
Permission: null

Press any key to continue...|
```

Figure 28: Broadcasts output

10.1.1.3 Package Menu

Here we have a sub-menu, because we can make several options with the packages, at the *image 29* we can see the options:



```
Archivo Editar Ver Buscar Terminal Ayuda
====Package Menu====
===Select an option===
1) List the package          3) List the shaed UID packages
2) Info about the packages  4) List the debuggeable packages
0 Go back
Choose an option:
```

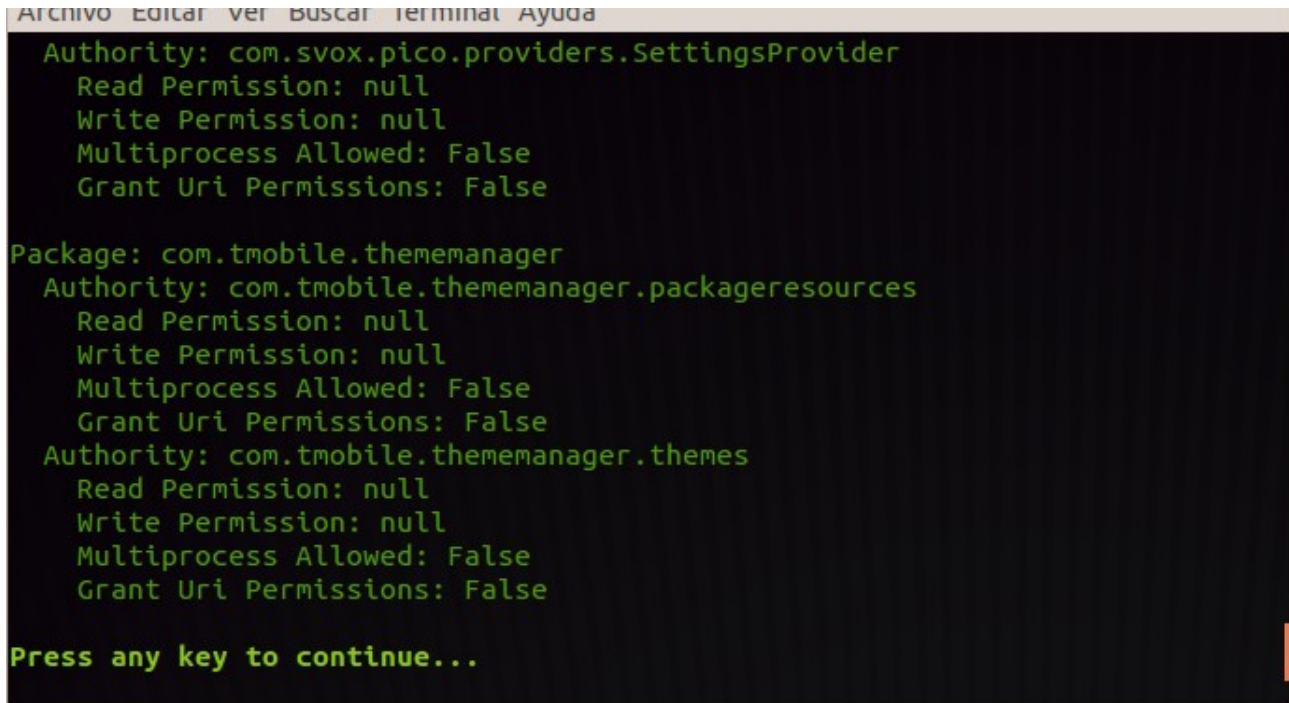
Figure 29: Package sub-menu

- Option 1): just a list of installed packages
- Option 2): we obtain information about the installed packages, with the UID, with the whole path, and the permissions that requires that package.
- Option 3): we obtain only the packages that shares their UIDs.
- Option 4): we obtain only the packages that are debuggeable.

As the other menus we can always redirect the output to an txt file.

10.1.1.4 Providers

Here we can search for all the providers installed in our mobile device as we can see at the *figure 30*, an example of the output, we can find:



```
Archivo  Editar  Ver  Buscar  Terminal  Ayuda
Authority: com.svox.pico.providers.SettingsProvider
Read Permission: null
Write Permission: null
Multiprocess Allowed: False
Grant Uri Permissions: False

Package: com.tmobile.themanager
Authority: com.tmobile.themanager.packageresources
Read Permission: null
Write Permission: null
Multiprocess Allowed: False
Grant Uri Permissions: False
Authority: com.tmobile.themanager.themes
Read Permission: null
Write Permission: null
Multiprocess Allowed: False
Grant Uri Permissions: False

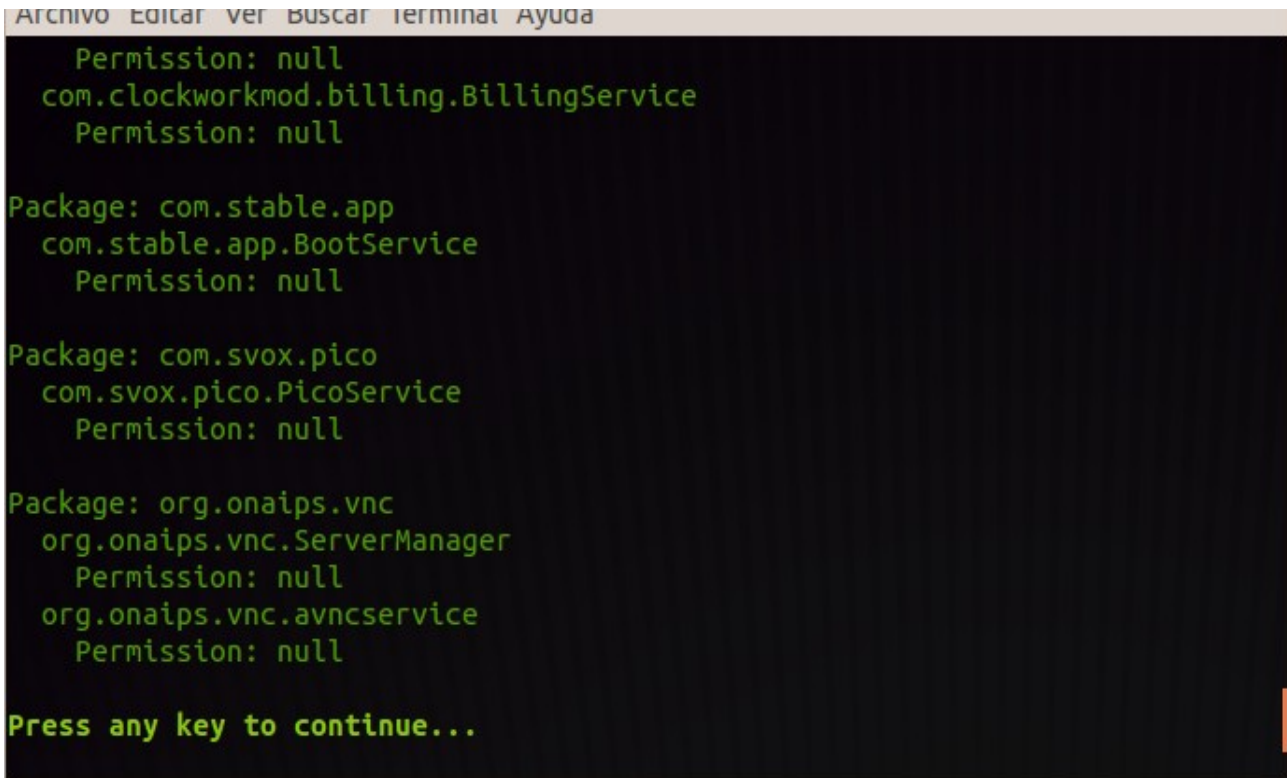
Press any key to continue...
```

Figure 30: Providers output

- The package name and path, *com.tmobile.themanager.com*.
- The owner authority, *com.tmobile.themanager.packageresources*.
- The read and write permissions.
- The multiprocess flag.
- The grant uri permissions flag.

10.1.1.5 Services

With that option we can find all the exported services in our mobile device, at the *figure 31* we can see an example of the output of this command, the services running in a real mobile device. We can see the package, the activity that starts the service and the permissions.



```
Archivo Editar Ver Buscar Terminal Ayuda
Permission: null
com.clockworkmod.billing.BillingService
Permission: null

Package: com.stable.app
com.stable.app.BootService
Permission: null

Package: com.svox.pico
com.svox.pico.PicoService
Permission: null

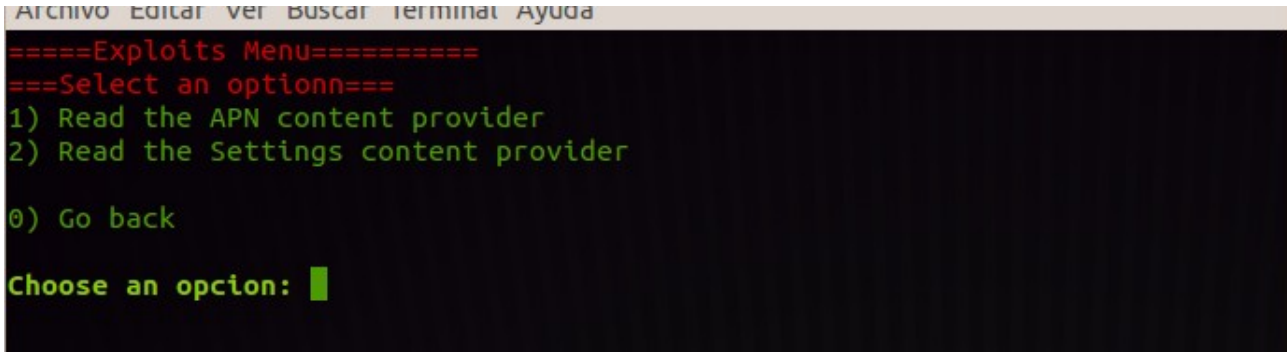
Package: org.onaips.vnc
org.onaips.vnc.ServerManager
Permission: null
org.onaips.vnc.avncservice
Permission: null

Press any key to continue...
```

Figure 31: Services output

10.1.1.6 Exploits menu

Here we can launch the pre-installed exploits in Mercury, we have two options, *read the APN content provider* and *read the settings content providers*, as we can see at the *figure 32*.

A terminal window with a title bar containing 'Archivo', 'Editar', 'Ver', 'Buscar', 'Terminal', and 'Ayuda'. The terminal content is as follows:

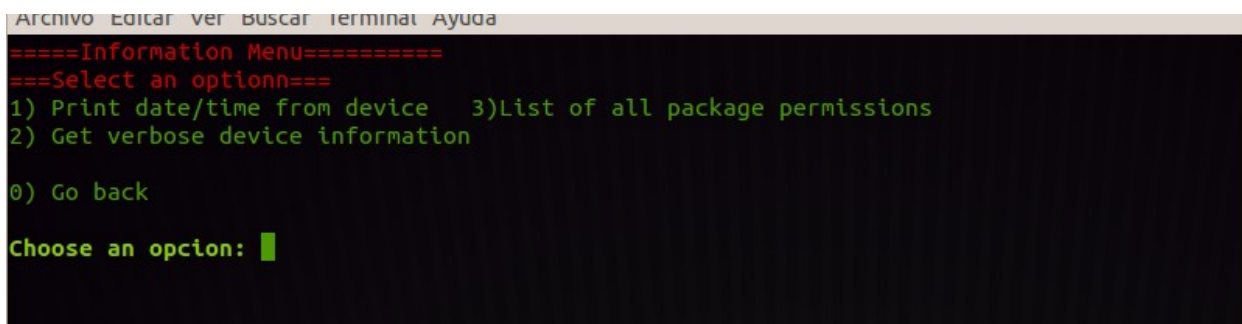
```
====Exploits Menu====  
===Select an optionn===  
1) Read the APN content provider  
2) Read the Settings content provider  
  
0) Go back  
  
Choose an opcion: █
```

Figure 32: Exploits menu

10.1.1.7 Information menu

In the *figure 33* we can see that menu, here can find the general information from the device,

- Print date/time from the device, we get the time in seconds
- Get verbose device information, we get the device, linux version, the /system/build.prop information.
- List all the package permissions, we get all the permissions from the installed packages in our device.

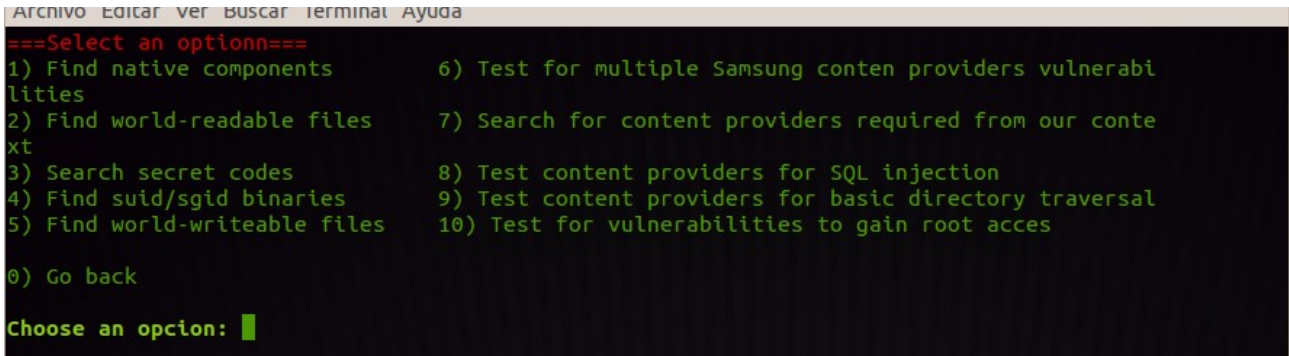
A terminal window with a title bar containing 'Archivo', 'Editar', 'Ver', 'Buscar', 'Terminal', and 'Ayuda'. The terminal content is as follows:

```
====Information Menu====  
===Select an optionn===  
1) Print date/time from device 3)List of all package permissions  
2) Get verbose device information  
  
0) Go back  
  
Choose an opcion: █
```

Figure 33: Information sub- menu

10.1.1.8 Scanner menu

Here we have the scanner sub-menu, here we can find the general searches, and the tests from know bugs or vulnerabilities. At the *figure 34* we can see the menu.



```
Archivo Editar Ver Buscar Terminal Ayuda
===Select an optionn===
1) Find native components          6) Test for multiple Samsung conten providers vulnerabi
lities
2) Find world-readable files       7) Search for content providers required from our conte
xt
3) Search secret codes             8) Test content providers for SQL injection
4) Find suid/sgid binaries         9) Test content providers for basic directory traversal
5) Find world-writeable files      10) Test for vulnerabilities to gain root acces

0) Go back

Choose an opcion: █
```

Figure 34: Scanner sub-menu

10.1.1.10 SQL Injection

Here we can find the option to find the vulnerable content providers vulnerable to the SQL injections, once we have the vulnerable contents we can launch selects to look for the information that they contain. In the *figure 36* we can find an example of total SQL analysis from a Samsung

```
sqlvul.txt x
Selecting b9c5d523b7701768 (samsung GT-I9000 4.0.4)
|_id| theme_package | theme_id | is_applied | author | is_drm | system | name | style_name
| wallpaper_name | wallpaper_uri | lock_wallpaper_name
| lock_wallpaper_uri | ringtone_name | ringtone_name_key | ringtone_uri | notif_ringtone_name |
notif_ringtone_name_key | notif_ringtone_uri | thumbnail_uri |
preview_uri | has_host_density |
has_theme_package_scope |
| 1 | | | 1 | Google | 0 | 1 | System | System
| Default | android.resource://android/drawable/default_wallpaper | null
| null | null | null | null | null | null |
null | null | null | null | android.resource://
com.tmobile.thememanager/drawable/default_theme_preview | 1 |
1 |
content://media/external/video/media

Selecting b9c5d523b7701768 (samsung GT-I9000 4.0.4)
|_id| _data | _display_name | _size |
mime_type | date_added | date_modified | title | duration | artist |
album | resolution | description | isprivate | tags | category | language |
mini_thumb_data | latitude | longitude | datetaken | mini_thumb_magic | bucket_id |
bucket_display_name | bookmark | width | height |
| 454 | /mnt/sdcard/Screencast/video_jul_05_2013_0.mp4 | video_jul_05_2013_0.mp4 | 606532 |
video/mp4 | 1373036770 | 1373036769 | video_jul_05_2013_0 | 20048 | <unknown> |
Screencast | null | null | null | null | null | null | null |
null | null | null | 1373036769000 | null | 37883566 |
Screencast | null | null | null |
content://com.android.cm.mms/templates

Selecting b9c5d523b7701768 (samsung GT-I9000 4.0.4)
```

Figure 36: SQL vulnerabilities

11 Intents classification

Here we have made a classification of the known Intents and broadcasts, depending on the input, output, only accessible by the system.

All actions are preceded by **android.intent.action**, information extracted from [Android Developers](#)

B= Broadcast Action A= Activity Action

No data input or output	With Input data	Only accessible by the system (2) (3)	No data entry with extra data	Without input data, with output data in, with extra data
ALL_APPS A	ATTACH_DATA (1)	BATTERY_LOW B	CALL (4) A	CHOOSER (6) A
ANSWER A	DELETE A (8)	BATTERY_CHANGED B	DOCK_EVENT B (10)	CREATE_SHORTCUT (7) A
APP_ERROR A	WEB_SEARCH A	BATTERY_OKAY B	HEADSET_PLUG B (14)	PROVIDER_ CHANGED B (29)
ASSIST A	EDIT A	BOOT_COMPLETED B		
BUG_REPORT A	GET_CONTENT A (13)	CONFIGURATION_CHANGED B		
CALL_BUTTON A	INSERT A (15)	DREAMING_STARTED B		
CAMERA_BUTTON (5) A	INSERT_OR_EDIT A (16)	DREAMING_STOPPED B		
SEARCH_LONG_PRESS A	INSTALL_PACKAGE A (17)	NEW_OUTGOING_CALL B (18)		
SENDTO A (33)	PASTE A (26)	PACKAGE_ADDED B (19)		
SET_WALLPAPER A	PICK A (27)	PACKAGE_CHANGED B (20)		
SYNC A	PICK_ACTIVITY A (28)	PACKAGE_DATA_CLEARED B (21)		
FACTORY_TEST A	RUN A (30)	PACKAGE_FIRST_LAUNCH B		
VOICE_COMMAND A	SEARCH A (31)	PACKAGE_FULLY_REMOVED B (22)		
MAIN A	SEND A (32)	PACKAGE_NEEDS_ VERIFICATION B		
MANAGE_NETWORK _USAGE A	ATTACH_DATA	PACKAGE_REMOVED B (23)		
POWER_USAGE_ SUMMARY A	SEND_MULTIPLE A (34)	PACKAGE_REPLACED B (24)		
GTALK_CONNECTED B	SYSTEM_TUTORIAL A (35)	PACKAGE_RESTARTED B (25)		

GTALK_DISCONNECTED	ACTION_UNINSTALL_PACKAGE A (37)	PACKAGE_VERIFIED B		
INPUT_METHOD_CHANGED B	VIEW A	ACTION_POWER_CONNECTED B		
MANAGE_PACKAGE_STORAGE B		ACTION_POWER_DISCONNECTED B		
MEDIA_BAD_REMOVAL B		REBOOT B		
MEDIA_BUTTON B		SCREEN_OFF B		
MEDIA_CHECKING B		SCREEN_ON B		
MEDIA_EJECT B		ACTION_SHUTDOWN B		
MEDIA_MOUNTED B		TIMEZONE_CHANGED B (36)		
MEDIA_NOFS B		TIME_TICK B		
MEDIA_REMOVED B		UID_REMOVED B		
MEDIA_SCANNER_FINISHED B		USER_PRESENT B		
MEDIA_SCANNER_SCAN_FILE B		EXTERNAL_APPLICATIONS_AVAILABLE B (11)		
MEDIA_SCANNER_STARTED B		EXTERNAL_APPLICATIONS_UNAVAILABLE B (12)		
MEDIA_SHARED B		MY_PACKAGE_REPLACED B		
MEDIA_UNMOUNTABLE B		DEVICE_STORAGE_OK B (9)		
MEDIA_UNMOUNTED B		DEVICE_STORAGE_LOW B		
QUICK_CLOCK		LOCALE_CHANGED B		
TIME_SET B				
USER_BACKGROUND				
USER_FOREGROUND				

(1) getData() is URI of data to be attached

(2) This is a protected intent that can only be sent by the system

(3) Sin datos de entrada

(4) If nothing, an empty dialer is started; else getData() is URI of a phone number to be dialed or a tel: URI of an explicit phone number

(5) Includes a single extra field, EXTRA_KEY_EVENT, containing the key event that caused the broadcast

(6) No data should be specified. `get*Extra` must have a `EXTRA_INTENT` field containing the Intent being executed, and can optionally have a `EXTRA_TITLE` field containing the title text to display in the chooser.

Output: Depends on the protocol of `EXTRA_INTENT`.

(7) Output: An Intent representing the shortcut. The intent must contain three extras: `SHORTCUT_INTENT` (value: Intent), `SHORTCUT_NAME` (value: String), and `SHORTCUT_ICON` (value: Bitmap) or `SHORTCUT_ICON_RESOURCE` (value: ShortcutIconResource).

(8) `getData()` is URI of data to be deleted.

(9) If nothing, an empty dialer is started; else `getData()` is URI of a phone number to be dialed or a tel: URI of an explicit phone number.

(10) `EXTRA_DOCK_STATE` - the current dock state, indicating which dock the device is physically in.

(11) `EXTRA_CHANGED_PACKAGE_LIST` is the set of packages whose resources (were previously unavailable) are currently available. `EXTRA_CHANGED_UID_LIST` is the set of uids of the packages whose resources (were previously unavailable) are currently available.

(12) `EXTRA_CHANGED_PACKAGE_LIST` is the set of packages whose resources are no longer available. `EXTRA_CHANGED_UID_LIST` is the set of packages whose resources are no longer available.

(13) Input: `getType()` is the desired MIME type to retrieve. Note that no URI is supplied in the intent, as there are no constraints on where the returned data originally comes from. You may also include the `CATEGORY_OPENABLE` if you can only accept data that can be opened as a stream. You may use `EXTRA_LOCAL_ONLY` to limit content selection to local data.

Output: The URI of the item that was picked. This must be a content: URI so that any receiver can access it.

(14) The intent will have the following extra values:

- `state` - 0 for unplugged, 1 for plugged.
- `name` - Headset type, human readable string
- `microphone` - 1 if headset has a microphone, 0 otherwise

(15) Input: `getData()` is URI of the directory (`vnd.android.cursor.dir/*`) in which to place the data.

Output: URI of the new data that was created.

(16) Input: `getType()` is the desired MIME type of the item to create or edit. The extras can contain type specific data to pass through to the editing/creating activity.

Output: The URI of the item that was picked. This must be a content: URI so that any receiver can access it.

(17) Input: The data must be a content: or file: URI at which the application can be retrieved. As of JELLY_BEAN_MR1, you can also use `"package:EXTRA_INSTALLER_PACKAGE_NAME, EXTRA_NOT_UNKNOWN_SOURCE, EXTRA_ALLOW_REPLACE, and EXTRA_RETURN_RESULT.`

Output: If `EXTRA_RETURN_RESULT`, returns whether the install succeeded.

(18) The Intent will have the following extra value:

- `EXTRA_PHONE_NUMBER` - the phone number originally intended to be dialed.

(19) May include the following extras:

- `EXTRA_UID` containing the integer uid assigned to the new package.
- `EXTRA_REPLACING` is set to true if this is following an `ACTION_PACKAGE_REMOVED` broadcast for the same package.

(20) The data contains the name of the package.

- `EXTRA_UID` containing the integer uid assigned to the package.
- `EXTRA_CHANGED_COMPONENT_NAME_LIST` containing the class name of the changed components.
- `EXTRA_DONT_KILL_APP` containing boolean field to override the default action of restarting the application.

(21) The data contains the name of the package.

- `EXTRA_UID` containing the integer uid assigned to the package.

(22) Broadcast Action: An existing application package has been completely removed from the device. The data contains the name of the package. This is like `ACTION_PACKAGE_REMOVED`, but only set when `EXTRA_DATA_REMOVED` is true and `EXTRA_REPLACING` is false of that broadcast.

- `EXTRA_UID` containing the integer uid previously assigned to the package.

(23) Broadcast Action: An existing application package has been removed from the device. The data contains the name of the package. The package that is being installed does *not* receive this Intent.

- EXTRA_UID containing the integer uid previously assigned to the package.
- EXTRA_DATA_REMOVED is set to true if the entire application -- data and code -- is being removed.
- EXTRA_REPLACING is set to true if this will be followed by an ACTION_PACKAGE_ADDED broadcast for the same package.

(24) May include the following extras:

- EXTRA_UID containing the integer uid assigned to the new package.

(25) The data contains the name of the package.

- EXTRA_UID containing the integer uid assigned to the package.

(26) Input: getData() is URI containing a directory of data (vnd.android.cursor.dir/*) from which to pick an item.

Output: The URI of the item that was picked.

(27) Input: getData() is URI containing a directory of data (vnd.android.cursor.dir/*) from which to pick an item.

Output: The URI of the item that was picked.

(28) Input: get*Extra field EXTRA_INTENT is an Intent used with queryIntentActivities(Intent, int) to determine the set of activities from which to pick.

Output: Class name of the activity that was selected.

(29) The intent will have the following extra values:

- *count* - The number of items in the data set. This is the same as the number of items in the cursor returned by querying the data URI.

(30) Input: ? (Note: this is currently specific to the test harness.)

(31) Input: getStringExtra(SearchManager.QUERY) is the text to search for. If empty, simply enter your search results Activity with the search UI activated.

(32) Input: getType() is the MIME type of the data being sent. get*Extra can have either a EXTRA_TEXT or EXTRA_STREAM field, containing the data to be sent. If using

EXTRA_TEXT, the MIME type should be "text/plain"; otherwise it should be the MIME type of the data in EXTRA_STREAM. Use */* if the MIME type is unknown (this will only allow senders that can handle generic data streams). If using EXTRA_TEXT, you can also optionally supply EXTRA_HTML_TEXT for clients to retrieve your text with HTML formatting.

(33) Input: getData() is URI describing the target.

(34) Input: getType() is the MIME type of the data being sent. get*ArrayListExtra can have either a EXTRA_TEXT or EXTRA_STREAM field, containing the data to be sent. If using EXTRA_TEXT, you can also optionally supply EXTRA_HTML_TEXT for clients to retrieve your text with HTML formatting.

(35) Input: getStringExtra(SearchManager.QUERY) is the text to search for. If empty, simply enter your search results Activity with the search UI activated.

(36) The intent will have the following extra values:

- *time-zone* - The java.util.TimeZone.getID() value identifying the new time zone.

(37) Input: The data must be a package: URI whose scheme specific part is the package name of the current installed package to be uninstalled. You can optionally supply EXTRA_RETURN_RESULT.

Output: If EXTRA_RETURN_RESULT, returns whether the install succeeded.

(38) Input: getData() is URI from which to retrieve data.

(39) Input: getStringExtra(SearchManager.QUERY) is the text to search for. If it is a url starts with http or https, the site will be opened. If it is plain text, Google search will be applied.