



universidad
de león

Departamento de Matemáticas

MÁSTER UNIVERSITARIO EN INVESTIGACIÓN EN CIBERSEGURIDAD

Trabajo de Fin de Master

“PLATAFORMA DE APRENDIZAJE Y PRÁCTICA PARA
CONCIENCIAR A LA POBLACIÓN ACERCA DE LAS
PRINCIPALES AMENAZAS EN LAS TECNOLOGÍAS
MÓVILES”

“LEARNING AND PRACTICAL PLATFORM FOR RAISING
PUBLIC AWARENESS ABOUT THE MAIN THREATS IN
MOBILE TECHNOLOGIES”

Autor: D. Alejandro Cueto Prieto

Tutor Académico: Dra. D^a Noemí de Castro García
Tutor Profesional: D. Álvaro Botas Muñoz

(Diciembre, 2017)

UNIVERSIDAD DE LEÓN
Departamento de Matemáticas
MÁSTER UNIVERSITARIO EN INVESTIGACIÓN EN
CIBERSEGURIDAD
Trabajo de Fin de Máster

ALUMNO: D. Alejandro Cueto Prieto

TUTOR ACADÉMICO: Dra. D^a Noemí de Castro García

TUTOR PROFESIONAL: D. Álvaro Botas Muñoz

TÍTULO: Plataforma de Aprendizaje y Práctica para concienciar a la población acerca de las principales amenazas en las tecnologías móviles

TITLE: Learning and practical platform for raising public awareness about the main threats in mobile technologies

CONVOCATORIA: Diciembre, 2017

RESUMEN:

Debido a la evolución tecnológica, los dispositivos móviles están empezando a sustituir a las soluciones de escritorio, predominantes hasta hace unos años, gracias a la movilidad, portabilidad, y potencia que proveen sus últimos avances. Día a día, los desarrolladores implementan nuevas aplicaciones, que permiten que servicios que hasta ese momento eran impensables, sean accesibles vía Smartphone o Tablet, aumentando las capacidades de estos, pero a su vez convirtiéndolos en objetivo para los hackers. Este proyecto, centrado en los dos principales Sistemas Operativos Móviles en la actualidad: iOS, y Android, trata de concienciar a los usuarios sobre los principales riesgos, en materia de seguridad y privacidad, que entraña este nuevo paradigma tecnológico, así como para presentar un taller que disponga de las herramientas necesarias que permitan a los profesionales realizar cómodamente análisis de seguridad sobre este ecosistema, así como a los usuarios sin conocimientos técnicos, les sirva como plataforma de aprendizaje y apoyo.

Palabras clave: iOS, Android, Security, Privacy, Mobile.

Firma del alumno:

VºBº Tutor:

VºBº Tutor 2:

Resumen

Debido a la evolución tecnológica, los dispositivos móviles están empezando a sustituir a las soluciones de escritorio, predominantes hasta hace unos años, gracias a la movilidad, portabilidad, y potencia que proveen sus últimos avances. Día a día, los desarrolladores implementan nuevas aplicaciones, que permiten que servicios que hasta ese momento eran impensables, sean accesibles vía Smartphone o Tablet, aumentando las capacidades de estos, pero a su vez convirtiéndolos en objetivo para los hackers. Este proyecto, centrado en los dos principales Sistemas Operativos Móviles en la actualidad: iOS, y Android, trata de concienciar a los usuarios sobre los principales riesgos, en materia de seguridad y privacidad, que entraña este nuevo paradigma tecnológico, así como para presentar un taller que disponga de las herramientas necesarias que permitan a los profesionales realizar cómodamente análisis de seguridad sobre este ecosistema, así como a los usuarios sin conocimientos técnicos, les sirva como plataforma de aprendizaje y apoyo.

Abstract

As a result of technological evolution, mobile devices are beginning to replace desktop solutions that were prevalent until a few years ago, thanks to the mobility, portability, and power of their latest advances. Day by day, developers are implementing new applications, which allow services that were unthinkable until then, to be accessible via Smartphone or Tablet, increasing their capabilities, but at the same time making them a target for hackers. This project, focused on the two main Mobile Operating Systems at present: iOS, and Android, and it aims to make users aware of the main security and privacy risks involved in this new technological paradigm, as well as to present a workshop with the necessary tools to enable professionals to carry out security analysis on this ecosystem, as well as users without technical knowledge, serve as a learning and support platform.

Outline

Index of figures.....	3
Index of tables	5
Glossary	6
1. Introduction	8
2. Objectives	11
2.1. Main objectives	11
2.2. Secondary Objectives	11
2.3. Transversal Objective	11
3. Methodology and Materials.....	13
4. Related work.....	17
4.1. Mobile Operating Systems	17
4.1.1. Security Mechanisms on Mobile Platforms	17
4.2. iOS Security Infraestructure	21
4.2.1 File System	23
4.2.2. Structure of an iOS Application	24
4.3. Android Security Infraestructure.....	26
4.3.1. File System	28
4.3.2 Structure of an Android Application	29
4.4. Some typical flaws on Mobile Environment	30
4.4.1. OWASP Mobile Risks last years.....	31
4.4.2. Types of threats: Inducted by the user or Triggered.....	34
4.5. Latest Threats on Mobile Platforms	37
4.5.1. Last Threats on iOS infraestructure.....	37
4.5.2. Main Threats on Android in the last years	40
4.6. Vulnerabilities on both platforms	43
4.6.1. Vulnerabilities on iOS	43
4.6.2. Vulnerabilities on Android	45
4.7. Need of a Security and Forensic Analysis.....	46
4.7.1. Security Analysis on Mobile Platforms.....	46
4.7.2. Forensics Analysis on Mobile Platforms.....	49
4.8. Procedure to perform an Analysis	51
4.8.1. Obtaining the Source Code.....	52
4.8.2. Repackaging	54
4.8.3. Analysis of Permission System	56
4.8.4. Evaluating Network Connectivity	59
4.8.5. Additional Elements	61
4.8.6. Extracting data on Mobile Platforms.....	63
4.8.7. Extraction of information stored in RAM memory	66
4.8.8. Analysis over Sensitive Information	67
4.9. Security Mobile Analysis tools	75
4.9.1. Utilities for Android	75
4.9.2. Utilities for iOS	76
4.9.3. Tools for Network connection in both platforms	77
4.9.4. Database Managers.....	79
4.9.5. Tools for a Mobile Forensic Analysis	80
5. Results: Security Analysis Workshop (SAW).....	82

5.1. Proposal	82
5.2. Minimum Requirements	82
5.3. How to install “Security Analysis Workshop”?	83
5.3.1. How to login?	84
5.4 Tools installed in “Software Analysis Workshop”	84
5.4.1. Integrated tools	87
5.4.2. Tools developed for this Master Thesis.....	87
5.5. Metrics	103
5.5.1. Limitations found	104
5.6. Real Cases of Security Analysis	104
5.6.1. Security Analysis over iOS app: DVIA.....	104
5.6.2. Metadata Analysis over Multimedia Assets	109
5.6.3. Downloading automatically applications	115
5.6.4. Permission classification from a Manifest file	117
5.6.5. Updating apk_tool easily	119
5.7. Limitations.....	120
5.8. Budget.....	120
6. Conclusions	121
References	122
Annexes	141
A. Usage on both platforms	141
B. Android Permissions Table.....	142
Acronyms	149

Index of figures

Figure 4.1. iOS Secure Boot, InfoSec (2003-2017), USA.....	20
http://resources.infosecinstitute.com/understanding-ios-security-part-1/	
Accessed: 29/11/2017	
Figure 4.2. iOS Security Architecture, Apple (1987-2017), USA	22
https://www.apple.com/business/docs/iOS_Security_Guide.pdf	
Accessed: 29/11/2017	
Figure 4.3. Plist configuration file.....	25
Figure 4.4. Android Framework, Google (1997-2017), USA.....	27
https://source.android.com/security/	
Accessed: 29/11/2017	
Figure 4.5. Main Android app components, Google4Tech (2013-2017), USA.....	29
http://www.google4tech.com/2015/05/android-components-activitieservicesbr.html	
Accessed: 29/11/2017	
Figure 4.6. OWASP Mobile Risks 2016.....	32
Figure 4.7. Comparative between OWASP Mobile Risks 2014 vs. OWASP Mobile Risks 2016, Security Innovation (2003-2017), USA.....	34
https://blog.securityinnovation.com/page/3	
Accessed: 29/11/2017	
Figure 4.8. iOS vulnerabilities from 2007 to 2017, CVE Details (2003-2017), USA.....	44
http://www.cvedetails.com/product/15556/Apple-Iphone-Os.html?vendor_id=49	
Accessed: 29/11/2017	
Figure 4.9. Android vulnerabilities from 2009 to 2017, CVE Details (2003-2017), USA.....	45
http://www.cvedetails.com/product/19997/Google-Android.html?vendor_id=1224	
Accessed: 29/11/2017	
Figure 4.10. Hopper Disassembler used in iOS analysis.....	54
Figure 4.11. Wireshark, Linux Secrets (2003-2017), USA.....	78
https://www.linuxsecrets.com/home/304-top-10-security-tools-for-penetration-testing	
Accessed: 29/11/2017	
Figure 4.12. Burp Suite.....	79
Figure 5.1. "Security Analysis Workwhop" first glance.....	84
Figure 5.2. Organizational Chart for CheckAPKTool.....	91
Figure 5.3. check_apktool execution.....	91
Figure 5.4. Organization chart for APKDownloader	94

Figure 5.5 apk_downloader execution	94
Figure 5.6 Organizational Chart for Manifest Interpreter	98
Figure 5.7 Manifest Interpreter running	98
Figure 5.8 Organizational chart for Metadata Extractor	102
Figure 5.9 Metadata extractor collecting private information	102
Figure 5.10 Exiftool extracting metadata from a certain picture information.....	110
Figure 5.11. List of exiftool parameters	111
Figure 5.12. Searching a place by its GPS coordinates	113
Figure 5.13 Execution of metadata_extractor	114
Figure 5.14. Results of metadata_extractor	115
Figure 5.15. Downloading an app via apk_downloader.....	116
Figure 5.16 Initial execution of manifest_interpreter	118
Figure 5.17. Results of manifest_interpreter	118
Figure 5.18. check_apktool executed with normal user privileges	119
Figure 5.19 check_apktool executed successfully	120
Figure 5.20. Latest apk_tool version already installed	120
Figure A.1. Android Distribution on November 2017, Google (1997-2017), USA	141
https://developer.android.com/about/dashboards/index.html?hl=es-419	
Accessed: 29/11/2017	

Index of tables

Table 3.1. iPhone 4S specifications	15
Table 3.2. iPad Air 2 specifications	15
Table 3.3. Motorola Moto G4 specifications	15
Table 3.4. Sony Xperia Tipo specifications	16
Table 4.1. Distribution of iOS vulnerabilities	44
Table 4.2. Distribution of Android vulnerabilities	45
Table 5.1. “Security Analysis Workshop” Minimum Requirements	82
Table 5.2. Tools instegrated in “Security Analysis Workshop”	86
Table 5.3. Source code snippet from check_apktool	89
Table 5.4. Source code snippet from apk_downloader	92
Table 5.5. Source code snippet from manifest interpreter	96
Table 5.6. Source code snippet from metadata_extractor	99
Table A.1 iOS Distribution (December 2017)	141
Table B.1. Android Permissions Classification	142

Glossary of terms

Jailbreak: is the name of the mechanism to unlock iOS devices, specifically iPhone. Generally, the user who wants to jailbreak his/her cell phone, He/She needs to download and next install a tool to his/her personal computer. When iOS dispositive is connected, the software detects it and starts a process of installation of the jailbreak which exploits some bugs found by researchers. Although, the iOS device now is capable of installing app from third-party stores, and even pirate software, some investigators claim this procedure is dangerous, and make your iPhone/iPad more vulnerable and unsafe to future attacks.

Root: a process in Android devices to get root access (super-user privileges), and unlock completely the cell phone or tablet. Users need to download and install a software in their computer, and when connecting the Android dispositive, following a few simple steps, they achieve to root the system. When system is rooted, users can install apps from third-party stores, which sometimes produces security issues, and They can even install a new and customized OS for their device, also named: ROM. In some terminals, users would be able to unlock their sim card, using their dispositives in any OSP (On-Line Service Provider), they choose.

1. Introduction

Nowadays, we are living in a world where developments in mobile technologies are marking our day-to-day. Without leaving aside the desktop environment, the mobility and easy-to-use features which provide modern devices like: smartphones, tablets, and smartwatches, they have created a new business model, and a brand-new approach to the IT (“Information Technology”) scenario. With all of this in mind, the number of different sources of information to consider, it has been increased substantially.

Day after day, the number of these dispositives are growing exponentially, bringing to the public all-new abilities to the palm of their hands, like for example: instant messaging applications to communicate with other people around the world, cloud computing solutions, even currently, they are deploying a way for making online payments through mobile platforms, among other new ideas under development. All this is possible, because cell phones, and portable gadgets have evolved, turning into a kind of small pocket computers, incrementing the complexity of the software inside them, and creating a new way to surf on the Internet.

In order to put in situation about this topic, we need to stand out the dominant mobile technologies which are ruling today, which are: iOS, and Android. These two Operating System (from now on OS), monopolize more than 95% (See smartphone market: [11](#)) of the market place in 2017, and are meant to maintain this position in future years. Lately, the companies behind these systems have incremented their area of action, introducing new state-of-art technologies like: Home Automation (also known as: Domotic), Augmented Reality (AR), or Online Banking.

Therefore, the nature of information present in these sort of dispositives is very diverse, so it needs to be treated differently according its origin, for preserving the privacy of the user. Like every piece of software, iOS and Android have bugs and different issues which may use by crackers and other malicious users to compromise these systems, exploiting vulnerabilities, and collecting sensitive data for different purposes. Furthermore, previously written, sometimes, these information leakages are produced by neglect, or deception techniques in order to get some private information, which is used to extort the user.

Hence, if we use these devices to save our personal information, and communicate with each other, it is paramount to improve these systems to achieve what was previously mentioned.

In order to assess the main risks that threaten these mobile platforms, there is a set of tools on the market that can be used by users to carry out these actions. Among the main existing utilities, we can highlight: apktool which allows the user to decompile mobile apps for Android, dex2jar that transforms the. dex files typical of Google Operating System into a. jar container for further processing, JD-GUI a graphical solution that provides the source code of the Java decompiled class, wireshark whose aim is to analyse network traffic, among others.

The problem with these software solutions is that most of them require a high level of technical knowledge in order to be able to use them, and their usage in many cases, even for professionals, is tedious and confusing. If global awareness is to be improved, a set of tools must be available that are accessible to novice users, while at the same time provides some kind of advantage to those with technical knowledge, allowing them to automate certain tasks or speed up their technical procedures

For all the above, we propose the creation of a workshop/platform called: "Security Analysis Workshop" that allows the user of these mobile devices, evaluate, test, and even learn about the main threats that put these platforms at risk, providing a free virtual unified environment thanks to the benefits of the Linux distributions, and in which will be installed by default a set of advanced tools that are used by the main analysts in security throughout the world, as well as a compendium of others developed for this project, whose main function is to automate certain processes that could be tedious and complicated for a user without technical knowledge, in addition to presenting in some cases the resulting information in a more visual and closer way to the public.

The existing tools mentioned above, among many others which will be described when the platform will be presented, they have been included to supply the user with a complete experience, which will provide the best and most used solutions on the market, in addition to serving as a professional platform for experts in security and privacy analysis on mobile platforms.

One of the limitations of this work is: the set of tools incorporated in the workshop for iOS analysis is smaller because Apple's system is much more closed than Google's, and also many of the utilities available on the market, are designed for the Desktop Operating System: macOS, so they are outside the scope of this project. However, a list of them, and their use is attached in the fourth section of this writing, corresponding to the current state-of-the-art, and entitled: "Related Work".

Moreover, due to one of the objectives of this project is to make the general public aware of the risks involved in the use of mobile technologies, an attempt has been made to develop a series of tools that facilitate, speed up, and even support the learning process for those users who choose to use them, showing the information in a clearer and more structured way than some of the tools available so far.

Therefore, it will be presented: `manifest_interpreter`. a solution that allows classifying the permissions associated with an Android app according to their degree of danger, showing clear descriptions of what each of them implies, as well as a more structured visualization than other tools. Next, we will introduce: `check_apktool` whose main function is to keep the Android analysis software updated to the latest version, avoiding the tedious process of doing it manually. The utility: `apk_downloader` will allow the user, or technician, to download an Android application for a later analysis through an external repository and in a more automated way than the options currently available. To end the section of tools specially developed for this project, it will be included: `metadata_extractor`, which aims to extract and export sensitive information associated with certain multimedia resources created on these mobile devices. Every of these tools will be equally integrated in the platform developed for this Master thesis.

With all this, the goal is to build a complete and robust platform, which allows security analysis to be carried out on mobile platforms with a guarantee of success.

Finally, it is important to note that this work will be organised as follows: in Section 2 will be presented main and secondary objectives of this work in addition to some objectives achieved transversally. Next in section 3, we are going to list the different materials, bibliography, and other resources used for this Master Thesis. In section 4, it will be developed the Related Work about the two mobile platforms to study: iOS and Android. In this long section, we will relate the architecture of both systems, apart from the main threats and vulnerabilities found in mobile ecosystem. Besides we will explain why a security and forensic analysis is necessary for this work, and what kind of tools we propose in order to perform this sort of analysis. Furthermore, in section 5 we will describe the platform created for this Master Thesis called: “Security Analysis Workshop”, and tools integrated and developed within it. As well, in this same section we will show some real cases in order to enlighten the user about the usage of the tools included, and we will introduce some metrics in order to allow users to evaluate, whether or not the platform fulfills its objectives. Finally, the last section of this document is for the conclusions reached after the completion of the work.

2. Objectives

In this section of this document, we will describe the main and secondary objectives associated with the accomplishment of the former, which are expected to be achieved with the realization of this work.

2.1. Main objectives

1. To elaborate a system (Linux distribution) that has the main tools used to carry out a security and privacy analysis in the mobile ecosystem, as well as others developed especially for this project, whose objective is to automate and speed up certain areas that, due to their complexity or because they are tedious, can slow down or make it impossible to develop a security analysis for the main mobile platforms.
2. To raise awareness among users and professionals of current mobile platforms of the risks involved in their use, providing them with the necessary resources (documents, multimedia content, other types of tools), to enable them to learn how security and privacy analysis is carried out in a mobile ecosystem, as well as to assess the potential threats to which they are exposed.

With the achievement of these objectives, there is also a set of milestones, directly related to them, which enrich and broaden the final scope of the Master's thesis.

2.2. Secondary Objectives

1. Develop a state of the art of the current situation, in terms of security on the most used mobile platforms: iOS and Android.
2. To make a list of the main software solutions present in the market.
3. Think about the features and other resources that can increase users' awareness of the main threats that mobile devices can pose.
4. Develop new tools which are able to improve the experience for professionals and novice users.
5. Implement every tool into the platform.
6. Check if the evaluation metrics are met.
7. Implementation of a series of real cases of use of the tools integrated and developed for the platform.

2.3. Transversal Objective

After all these goals accomplished, other types of them related to the implementation of this work could be triggered such as:

1. To create a unified environment that allows security professionals to perform their work comfortably and quickly
2. In addition to acting as a learning platform for all those who want to get into the intricacies of security in iOS and Android systems.

3. Methodology and Materials

This work can be framed in the field of research, and experimentation since it includes an initial part which is the collection, summary, and explanation of the state-of-the-art in both mobile platforms, as well as material to instruct users on how to perform a security and privacy analysis, but also provides added value as a workshop that can not only serve as a learning platform, but also as a useful solution for professional users, integrating some tools which help them to carry out their job.

Timing

- Selection of the topic (later October 2016): at the end of this month, the subject of "iOS & Android: Security and Privacy in Mobile Platforms" was selected, because it was considered to be a hot topic, very attractive, and of which we wanted to learn and research.

-Previous Documentation (November 2016): research is beginning on the current situation of these two platforms, in order to be able to elaborate a state-of-the-art in security and privacy matters of both iOS and Android, which can serve as an introduction to the Final Master's Thesis to be submitted.

-Elaboration of a Paper (December 2016): a Paper is developed on the risks that exist in the mobile platforms under study, trying to formalize the state-of-the-art developed in previous phases.

-Presentation of the Paper (early January 2017): the Paper is presented at the University of León, making a brief presentation in which the main risks posed to users by the use of these mobile devices are listed.

-Brainstorming (later January 2017 - later February 2017): these months are dedicated to thinking about what we want to do as a Final Master's work. It is clear that we want to do something to raise awareness among the general public, since the user is always considered to be the weakest link, but at the same time we do not want to leave the user with technical knowledge outside the scope of this project.

- Initial Analysis (March 2017): we want to make a thorough documentation of the current security and privacy situation in iOS and Android, therefore it is decided that in order to be oriented to: non-technical public and professionals, we have to try to develop a series of sections that serve to instruct on how to perform a security and privacy analysis for non-experienced users, but at the same time we have to look for a way which allows them to test what they have learned.

-Structure of the Work (April 2017 - early May 2017): in order to solve the lack of practical content, we are thinking of developing a workshop that will serve not only to practice for users who wish to use it as a learning platform, but also serve as a solution that combines the main tools in security that exist today, and therefore be a quality application for the most technical users.

- Collection of Information (Later May 2017 – early July 2017): once it has been decided, what will be developed in the Final Master's work, we proceed to look for documentation

via: books, websites, papers among others, in order to offer a quality and updated information to the readers of this work.

- Filtering of Information (mid July 2017): all information considered outdated or not directly related to the work is rejected.

- Workshop Infrastructure (later July 2017): it is decided that in order to develop a platform that will serve for both learning and technical use, a distro Linux (Ubuntu) will be used, which will allow its free distribution, as well as ensuring that it can act as a basis for the installation of the main tools that currently exist for analysis on mobile ecosystem.

- Writing Related Work (August 2017): we begin writing what is considered "Related work", or state-of-the-art in iOS and Android, besides more research on how to analyze both environments.

-Development of the Workshop (early September 2017): during the initial process of installing the tools, we realized that some of them, due to their start-up or display of the information, they could be complicated to use by users without knowledge, so we thought about a method to resolve this important issue.

-Programming of new tools (later September 2017 - mid October 2017): in these months the new tools specially created for this work are developed, with the aim of facilitating their use by novice users, as well as speeding up certain tasks that could be tedious for more experienced users.

-Documentation of what was done in the workshop, and correction (November 2017): this last month, prior to the presentation, it was dedicated to document the tools developed, in addition to correcting the structure of the documentation, apart from debugging on the programs and scripts integrated in the platform.

-Delivery and presentation (December 2017): finally, the work is deposited and presented.

Materials

Next, we will list the devices I have used throughout this master thesis to conduct my research on major threats to mobile platform security and privacy. Because this Final Master's Work focuses on the two currently most used Mobile Operating Systems (iOS and Android), I have needed the use of a series of devices, in order to be able to carry out the study with guarantees that will be reported throughout the rest of the sections of this document.

For iOS ecosystem [2] research, I've needed the materials described in Table 3.1 and 3.2:

Smartphone		iPhone 4S
Operating System	iOS v9.3.5	
CPU	Apple A5 2x800MHz	
RAM	512MB DDR2	
Storage	32GB	



Table 3.1. iPhone 4S specifications

Tablet		iPad Air 2 [3]
Operating System	iOS v11.0.3	
CPU	Apple A8X 3x1.5GHz	
RAM	2GB	
Storage	128GB	



Table 3.2. iPad Air 2 specifications

When we made the research for the section dedicated to Android we used the materials showed in Tables 3.3 and 3.4.

Smartphone		Motorola Moto G4 [4]
Operating System	Android v7.0 (Marshmallow)	
CPU	Qualcomm Snapdragon 617	
RAM	1GB	
Storage	32GB	




Table 3.3 Motorola Moto G4 specifications

Smartphone	Sony Xperia Tipo [5]
Operating System	Android v4.0.4 (Froyo)
CPU	Qualcomm Snapdragon 51
RAM	512MB
Storage	8GB




Table 3.4. Sony Xperia Tipo specifications

Besides this smartphones, and tablet we used some books in order to improve our knowledge about the platforms to study. This set of books could be seen in the Section: References, in the part entitled: Bibliography.

Furthermore, in Section: 5 and subsections, we will list the minimum requirements need for installing and deploying the platform developed for this work: "Security Analysis Workshop".

With all these dispositives, in addition to a computer to install the image of "Security Analysis Workshop" described in next section, a research will be developed to show the main tools used by a security researcher in mobile environments, in order to analyze and evaluate the security architecture of the platforms under study, through the analysis of some apps present in their official stores.

4. Related work

Throughout this section, there is a detailed description of the current state of the art in the two main mobile platforms: iOS and Android, describing the main security flaws, explaining the reasons that lead to the need for a security and forensic analysis in these environments, as well as available tools to carry out these jobs, along with a detailed description of its functionality and usage.

4.1. Mobile Operating Systems

Nowadays, computer systems are moving from the desktop to the mobile environment, which is why making a presentation of today's main mobile operating systems is essential, in order to know the fundamentals on which this project is based. Due to the increase of capabilities of the new smartphones, day after day these mobile devices are becoming like a pocket computer which allows the user to perform a lot of actions which were impossible before. This increase in possibilities has meant that nowadays much of the personal information of each user is stored on their smartphone or tablet, making it a very valuable target for hackers and other malicious users. This is why, first of all, we will mention the main security mechanisms offered by mobile operating systems in order to guarantee the security and privacy of users, and then provide a detailed description of the main current mobile operating systems (iOS and Android), along with the main features that determine each one.

4.1.1. Security Mechanisms on Mobile Platforms

From the beginning, when both Google and Apple decided to design their Operating Systems for mobile platforms, they decided that one of the pillars on which they would build their entire infrastructure would be: security above all. Every day smartphones are growing in number, and their capabilities and possibilities seem to make these new solutions that provide mobility and portability to users move more and respect the desktop architecture that has prevailed until now. Because of the need to store personal information on mobile devices, it is necessary to provide hardware and software with mechanisms to safeguard it from computer attacks, which due to the success of these platforms are increasingly targeted for crackers.

In order to guarantee the security of its users, today's main mobile operating systems (iOS and Android), provide a series of features aimed at protecting them against the main threats that exist, from those caused by software vulnerabilities of the systems themselves, to everything that has to do with infections through malware, and loss or theft of the device.

In order to achieve all of the above, although each system performs a different implementation, in general, the series of mechanisms offered to protect mobile users are (See more information in [\[6\]](#)):

- Sandboxing of Applications.
- System of Permissions.
- Access Control.

- Signed Apps.
- Safe Booting.
- Data encryption for apps.
- Remote Wipe and Block.

Next, we will briefly describe how each of the systems implements these features, to give an overview of how companies not only have configured their security architecture, but also know how our mobile OS protect us against the main risks to which we are exposed day after day.

a) Sandboxing

To improve the intrinsic security of the system, and prevent possible intrusions make a privilege escalation, getting access to system assets that should be restricted by default, the main Mobile Operating Systems, work by assigning users an account with very limited privileges, in order to control the degree of action of these users, and the impact they may cause the system.

This is why, in order to access of the system's resources, each OS uses a permissions system that determines if something is accessible or not for a certain user. In addition of improving the overall security of each of the system's applications, and its scope, the system establishes a kind of software cage called: sandbox (a detailed description in [7]), which prevents other apps from accessing its resources, as well as limiting its range of action to the sandbox in which it is enclosed.

For iOS deployment (a more detailed information about iOS sandboxing in: [8]), this feature causes each of the applications that the user installs on the system to install a directory with a random name that is generated through the unique UDID (Unique Device Identifier) that Apple associates with each device during its manufacturing process. In addition, in order to limit the user's impact on the system, it assigns a virtually unprivileged user, called "mobile". From the internal point of view, the app is only able to access everything that is inside its directory, and due to checks carried out by the system on the application signature, the access of each app is also limited to its area of action, which due to their importance, are restricted for the user's access.

In the case of Android, and due to their Linux nature, applications are run inside a virtual machine, whose user has a certain id, as happens in desktop distributions based on the Linux kernel. Applications, like Apple's system, only have access to their root directory, and not to the resources of other apps installed on the system. Since a few years ago, and after the arrival of Android 4.3, another component aimed at improving the security of the Linux world was included: SELinux (further information about this feature in: [9]), a system that provides a mechanism for user access control, through the implementation of various security policies, in order to restrict the actions that each user can perform.

b) Permission System

Complementarily, to the deployment of sandboxing in the Mobile Operating Systems, a system of permissions (information about UNIX permissions: [10]) has been implemented, which guarantees that a user cannot access system resources, which have

access privileges. Due to the common origin of both systems (UNIX), each system file has three types of permission: read, write and execute, thus controlling the actions that the owner of the same, and the rest of users of the system can do with them. But not only the permissions system affects software, because it is basically aimed at controlling the resources and hardware sensors that populate our mobile devices today.

Hence, the system provides the user with the ability to control that other apps of the system have access to the camera, its list of contacts and messages, as well as photographs or videos stored in its multimedia library, giving the owner of the mobile device, the ability to control the access that other applications have to the set of resources of your mobile device. In the case of both Android, and iOS, the responsibility for providing the ability to configure these permissions, comes from the developers of the applications, but while for many years Apple's system had a control center that allowed the owner of the mobile device, enable or disable individually each permission that requested every app installed on the system, in the case of Android was not until version 6.0 when it was provided. For a long time, the lack of this mechanism was the entry point for many malware, and other malicious software to Google's Operating System.

c) Access Control

To prevent other users from accessing the user's device, both iOS and Android implement a set of locking mechanisms (further information about “Unlocking mechanisms in [11]) that allow the device's content to be protected from external access.

The main methods used, chronologically described from oldest to most modern are:

- Four-digit or more code for unlocking the smartphone.
- Alphanumeric password.
- Graphic pattern, used as an alternative to the Android system.
- Fingerprint unlocking.
- Iris' scanner.
- Facial recognition.

All of these mechanisms are susceptible to being violated by malicious users, and although methods such as fingerprints or facial recognition seem to be the safest a priori, in recent times it has been shown that with the use of latex prostheses with the imprint of the owner of the cited device, or in the case of facial recognition, with a photo of the owner of the smartphone, has been able to infringe these security mechanisms, which seemed to provide so many guarantees.

d) Signed Apps

In order to upload the applications to the official store, as well as to be used by users of each of the platforms under study, developers must sign (details about signing Android apps in [12]) them beforehand. This mechanism makes it possible to identify not only the developer, but also the platform where it is installed, and prevent other unofficial stores from impersonating a legitimate entity, deceiving the user to install apps that include malware and may violate their security.

In the case of the iOS operating system (further information about iOS signing in [13]), it takes this feature a little further, not allowing apps to be installed on the user's device that have not been approved with the Apple certificate. Although this policy may seem very restrictive, it's one of the main reasons why malware is not as extensive on the Apple platform as it is on Google.

e) Safe Boot

One of the main characteristics that determine the security of an electronic system is to guarantee the integrity of all the information stored in it. In order to do this, the Operating Systems divide the boot process of the system into a set of stages that make up what is called: Safe Boot. This set of steps carried out inside the system, are performed with the sole objective of ensuring that the code executed inside the system, has not been tampered in any way during the boot process, and can alter the normal execution when the entire OS is loaded.

If by any chance, this procedure determines that the code has been modified in some way, the devices usually enter a mode called: "Recovery Mode", which requires the device to be restored so that it can be reused.

To illustrate this important safety feature implemented in mobile systems, we can cite the Secure Boot (more information about this procedure in [14]) by iOS devices, which is made up of this set of different steps in order to guarantee the security in all the process of booting the system:

Finally, we show a Figure 4.1 which summarize the process of booting in an iOS mobile device.



Figure 4.1. iOS Secure Boo. Source Infosec

f) Data Encryption

Due to the enormous amount of various types of information stored on our mobile devices, in order to ensure that it is protected against any intrusion, it is encrypted so that only the owner of the device can access it when he/she unlocks his/her mobile device.

Initially, encryption was an optional feature on the Android system, but as time has passed and the importance of this mechanism has become apparent, it has become mandatory (Android 5.0 onwards). Currently, Google's system uses a symmetrical-key encryption method called: AES (Advanced Encryption Protocol) 128 CBC (Cypher Block Chaining) (further information about this encrypt algorithm in [15]). The encryption key that is used by the system is created randomly, and is used to sign all other encryption operations in the system. In this way, all system data remains encrypted, or at least those

related to the internal memory of the mobile device, since in the case of external memory such as SD cards (Secure Digital cards), their contents are not encrypted by default.

In the case of the Apple Operating System, encryption has been a feature provided almost since its inception. Currently, all information stored in flash storage is encrypted to: AES 256 (more detailed information about iOS encryption in [16]). In addition, as a special feature, the iDevices incorporate an internal co-processor optimized to perform all encryption/decryption operations within the system, which in fact saves the encryption key of the device, which is generated during the manufacturing process of the device, and which is protected so that other system resources do not have access to it.

Additionally, iOS allows each app separately to be encrypted with a key that is created randomly when installed. This key is stored, and only allows access with the owner of the mobile device unlocks their system.

g) Remote Wipe and Block

Due to the possibility of loss or theft of these mobile devices by thieves, both mobile Operating Systems provide features that allow the system to be blocked remotely to prevent criminals from taking advantage of it. Once the deletion or blocking procedure (information about this process in [17]) is performed, a completely new encryption key is generated in the system, preventing the information that is currently stored in the system from being accessible by malicious users.

Both Apple, and Android provide a default application called "Find my iPhone", and "Find my Device" (an overview of these apps in [18],[19]) respectively, which provide an interface from which if you have the location activated, the user can geo-locate approximately the location of their stolen device, in addition to being able to perform a set of quick actions, such as: remote deletion, and the complete blocking of the device. Access to these capabilities is not only accessible from another device to which the lost device is linked and synchronized, but also allows access via cloud services through a web browser. Once the user has been identified with their credentials, he or she will be able to take the same set of actions in the event of an emergency, and thus have another mechanism to protect his or her privacy, when this is clearly violated.

4.2. iOS Security Infrastructure

As I mentioned in previous chapters, iOS mobile infrastructure is based on a close-source model, therefore sometimes it is really complicated to explain every part of this system in detail, because although the inner procedures are explained, some parts of themselves are unknown, and its code is not accessible to the public.

If we read carefully the security model used by Apple ("iOS Security Guide" [20]) when they created the OS, we can see, iOS is a system with a well foundation focus on security, not only the system but also the information stored in the OS itself. One of the main goals for Apple engineers is: to protect user information, thus privacy become in one of the principal pillars for iOS. To achieve these objectives, Apple used encryption to ensure chiefly the structure of system which is divided in several layers to improve security, but they also use encryption protocols (AES [21]) to protect all the information

saved in the dispositive powered by iOS, because they know: smartphones, tablets, and the rest of mobile devices have become in a sort of pockets computers, so they stored a lot of different information which cannot be compromised.

To improve overall performance in these dispositives, mostly of Apple chips has a particular **co-processor** dedicated to make all encryption operations (Security Enclave), and generate random numbers which are necessary to operate with these algorithms, besides a memory to collect all this information generated, and work with it. In Figure 4.2, we attach an overview about what it has been previously mentioned.

Since 2013, they have also added some improvements to their dispositives, in order to increment the overall security of these kind of systems. For instance, since iPhone 5S and newer, they include a fingerprint recognition sensor which is called: TouchID (a detailed description about this sensor is presented in: [22]). This characteristic works like a master password

which can be used like a way of univocal authentication by user, because his/her fingerprint is a personal feature which can be useful for purchases, log in services, and unlocking the device itself. Like user fingerprint is a something private, and its theft could probably lead to violate user rights, it is encrypted like practically every place and every data inside the OS.

We should also mention the inclusion of two new technologies by Apple, in their new terminals presented in September 2017, and that were: iPhone 8/8 Plus and iPhone X. These devices mount a new processor called: "A11 Bionic", which has a neural engine capable of executing up to 6000 million operations per second, and be responsible for controlling the new machine learning procedures implemented in the system that will now serve to improve the processing of natural language by the system, improving and processing images, in addition to learning from the habits of the user. In addition to all that has been specified, this new engine is responsible for controlling FaceID (further information about this new method of unlocking in [23]), included in the iPhone X, which is the new method for unlocking the terminal, and payment through mobile terminals included in the platform, and which becomes the substitute for TouchID, which was the solution used by Apple's devices until now. This new technology has eight sensors including an IR camera, a dot projector to create a virtual map of the user's face, as well as a light emitter to identify the carrier of the mobile phone, even in low visibility or no visibility conditions. All the information processed by these sensors allows to identify the face of the owner of the device, and claims to be more secure than the TouchID, and as it has been used with the latter, does not process its information in the cloud, or on external

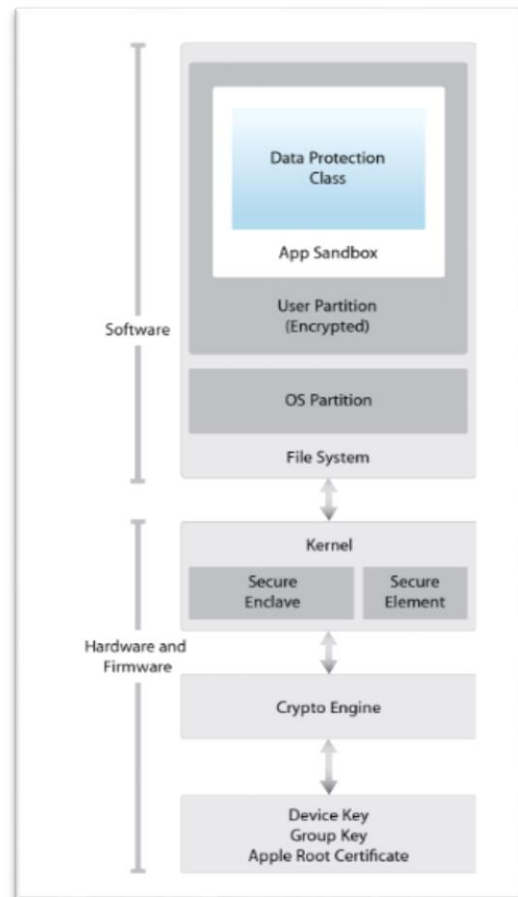


Figure 4.2. iOS Security Architecture. Source: iOS Security Guide.

servers, which endanger the privacy of the user, but that all this information is stored in the "Security Enclave" described above, it is therefore stored locally on the user's device, and this personal information is kept encrypted, to preserve the security and privacy of the user. For more information about "iOS Authentication", see: [\[154\]](#).

We cannot end this section without speaking about the AppStore ("AppStore Guidelines" [\[24\]](#)), surely another pillar of iOS security infrastructure. The AppStore is a central virtual market specially designed for Apple's products, which contains around 2200 millions of applications (regarding sources from June 2017 [\[25\]](#)). This number is increasing every day, despite the latest Apple policy of removing apps with no 64 bits' support, forcing developers of upgrading their apps to support the newest SDK (Software Development Kit) launched by the company. Until now, nothing is different respect other stores like: Google Play in Android, but there is one thing which makes a difference: The control. After a developer who enrolled in Apple developer program upload an application, a team inside Apple itself tests the app sent by the app developer, searching some flaw, bugs, vulnerability, and even malware, and it is detected they reject the app until the developer solves the issue. This process is not immediate, it takes around 2-3 days depending on the complexity of the app, but it has become in a great way to avoid the presence of malware and other malicious software in Apple devices, and so far, it works. To see more information about "iOS Security" check out: [\[155\]](#).

4.2.1 File System

Apple's system, keeps a tree structure similar to Android's, as both mimic the structure implemented by UNIX decades ago, but despite this the way of organizing its content, in addition to the format in which this information is stored on the system differs greatly from Google's system.

Until years ago, iOS used its own proprietary file system called: HFS+ (Hierarchical File System plus) (more information about Apple former file management: [\[26\]](#)), which was an adaptation of its counterpart for its desktop systems (macOS). Due to the need to adapt to new technologies, in 2017 they included a new file system called: APFS (Apple File System) (further information about this new file system in [\[27\]](#)), with the release of version 10.3 of iOS. APFS has a special feature with respect to the previous system, which is optimized for flash and solid-state drive storage, and which has as its primary focus the encryption of the information stored in them, an aspect that since its beginnings, Apple has made an effort to polish. As it was the case with its predecessor, APFS has a file structure similar to the one implemented by UNIX although it has its own differences, and in it, the main directories that compose it are:

-/Applications: is the main directory where the applications are installed by default on Apple's mobile devices.

-/private/var: is a partition oriented to store data referring to the user of the device.

-/private/var/mobile/Applications: directory where the rest of third-parties' applications are stored, which the user can download from the AppStore.

-/boot: place where OS updates are saved.

-/private/etc: this directory stores the entire compendium of system configuration files.

-/Library: The software libraries that are used by the different apps installed on the mobile device.

-/Developer: stores the development libraries, used during the programming of apps for the Apple platform, through the development IDE: Xcode.

As with Google Android, in Apple to improve the overall security of the system, each app is installed in a particular directory with a random name derived from the UDID of the device, and that by means of a sandboxing mechanism, isolates the scope of each of the applications installed on the device, to the internal environment of the directory where they are installed.

4.2.2. Structure of an iOS Application

Like every Desktop Operating System, iOS has its compendium of software which is designed to be installed in the Apple Mobile OS. As with Android, Apple provides an SDK (Software Development Kit), which has a series of software components that allow developers to build applications for their system, quickly and conveniently, through their own development IDE (Integrated Development Environment), in this case: Xcode.

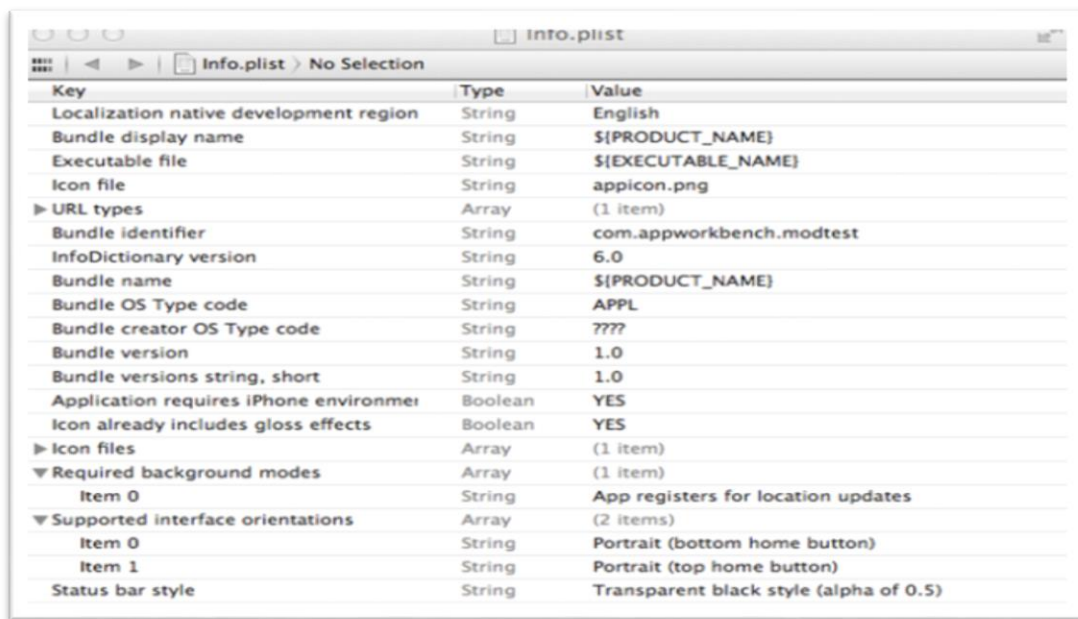
For iOS, the software components are called: Views, specifically UIView (more information about UIView in [28]), which is a class from which all the other subcomponents with which a programmer can create the application under development inherit. Once we have introduced the UIView class, they inherit from it another type of classes such as: UIWindow which, as its own name indicates, allows to build the window itself where the set of subcomponents that compose a view will be located, in addition to another more common type of objects such as: UITextView, UIButton, UISpinner among others. As can be seen, each and every view has the predicted UI (User Interface), which is a way for the iOS SDK to identify that effectively the specified set of components is intended to build a user interface. iOS's framework for designing UIs in iOS is called: Cocoa, and is specially designed to work optimally on mobile devices with a 64-bit architecture.

What has been mentioned so far is only the way to build graphical interfaces, but apart from UIs, developers must be provided with a way to generate the logic that developed each mobile application. In iOS, the programming languages that allow this are: Objective-C (description about former programming language in Apple system in [29]), and Swift. Until three years ago, the only alternative to develop on Apple's system was Objective-C, a rather outdated C++ extension despite its upgrades, which although it complied efficiently with the development of apps, had remained a little outdated in the present times. That's why Apple decided to create a modern language called Swift, which is also open-source, and allows applications to be developed more quickly and with a much cleaner syntax.

As with the Google Operating System, we have to cite the configuration files, which in the case of iOS, are called: plist configuration files, and which are a fundamental

part when defining the global configuration that will have a certain development in Apple's mobile platform.

Plist configuration files (further information about these files in [30]), is a structured text file, as we can see in Figure 4.3, that contains essential information for the binary that makes up the mobile application. Internally, it is encoded in UTF-8 (Unicode Transformation Format), and its structure is a dictionary created from XML (eXtensible Markup Language), in which there is therefore a set of key-value tuples. When it is generated, by convention is always associated with the name of: Info.plist, and is hosted in the contents folder of each application, which is automatically generated as it is developed from the official IDE (Integrated Development Environment) of the platform: Xcode.



Key	Type	Value
Localization native development region	String	English
Bundle display name	String	\$(PRODUCT_NAME)
Executable file	String	\$(EXECUTABLE_NAME)
Icon file	String	appicon.png
URL types	Array	(1 item)
Bundle identifier	String	com.appworkbench.modtest
InfoDictionary version	String	6.0
Bundle name	String	\$(PRODUCT_NAME)
Bundle OS Type code	String	APPL
Bundle creator OS Type code	String	????
Bundle version	String	1.0
Bundle versions string, short	String	1.0
Application requires iPhone environment	Boolean	YES
Icon already includes gloss effects	Boolean	YES
Icon files	Array	(1 item)
Required background modes	Array	(1 item)
Item 0	String	App registers for location updates
Supported interface orientations	Array	(2 items)
Item 0	String	Portrait (bottom home button)
Item 1	String	Portrait (top home button)
Status bar style	String	Transparent black style (alpha of 0.5)

Figure 4.3. Plist configuration file

This file includes, among other things, a description and how the set of components that make up the app in question of behavior, as well as being the place selected by iOS to store the path of possible extensions that may be added to the app under development, since it specifies some usual addons on the Apple platform such as the "Share" tab, or the integration with the company's official smartwatch: Apple Watch.

Moreover, this small configuration file is also in charge of providing a URI (Uniform Resource Identifier) through which the mobile application will be able to communicate with others, because through it will be able to receive and interpret all types of data that have been sent by other applications of the system, which greatly enriches its functionality, and manages to jump somehow one of the main security measures of the platform: Sandboxing, which did not enable the apps themselves to access. This mechanism is used to ensure that the app does not remain completely isolated from what is happening outside.

4.3. Android Security Infrastructure

As Apple does, Android's internal architecture is built to provide the highest level of security to its users, since currently the information stored by them on mobile platforms is private, and therefore requires a number of mechanisms to ensure their privacy.

From the beginning, Android has been designed to be an open platform that is enriched every day. Specifically, its architecture is based on a multi-layered model, which is directly created so that developers can implement the security controls provided by

Google's system. In turn, there is a team called: Android security team in charge of searching for potential vulnerabilities in apps, and ways to correct them. For example, updates were recently provided for the latest OpenSSL (Secure Socket Layer) issues, making system users secure against emerging threats using this Internet-based communications technology.

In addition, from Android 6.0 Marshmallow onwards, Google allows its users to control the permissions requested by all apps when installed on the system, thus ensuring that the user has full control over the information that accesses each of the apps installed on their device, and therefore minimizing the impact that may have some malicious applications, plus infecting the device with malware that affects overall system performance.

As it can be seen in Figure 4.4, the Android Security Stack is divided into different layers, each of which assumes that all components below it is properly secured. In addition, being an operating system built on the Linux kernel, ensures that only a small number of internal processes run with root privileges, while the rest work under a sandbox whose permissions are very restricted, to prevent an application can affect the system if a security problem is discovered, which does not yet have a patch.

Some of the main features that define the Android operating system (much more information about “Android Framework in [31]), are as follows:

- Hardware of the device: Android is processor-agnostic, although it takes advantage of the ARM (Advanced RISC Machine) architecture, which allows it to work in multiple devices, among which we can highlight: smartphones, tablets, TV, among others.

- Android runs on a custom Linux kernel, which is responsible for controlling each of the various sensors and technologies of the devices governed by the system.

- The majority of Android applications are programmed in Java, and work thanks to ART (Android Runtime), although there are also other types of native applications that are programmed in C/C++, and coexist with the rest of the system apps contained in a sandbox, thus maintaining the same security controls as the rest. This type of applications, have a private space in the file system, to write in their databases, and store important information.

In order to be able to extend the bulk of system applications, Android has two fronts:

- Pre-installed set of applications: these depend on the OS version itself, and even on the provider of the device (Android, unlike Apple, is used as OS of several brands, among which are: Samsung, LG, HTC or even Xiaomi). In addition, in order to differentiate their devices from the competition, many providers install a User Interface (UI) independent from Holo (Google's official UI), as Samsung does with TouchWiz, or Xiaomi with: MIUI.

- User applications: applications that can be installed from official or non-official markets, and that make the number of apps in the system grow to hundreds of thousands.

In addition to all of the above, Google provides a number of cloud-based services, which are included in a multitude of devices that work with Android. Some of them are:

- **Google Play** [32] (also called Play Store), includes the set of mobile platform applications classified by type. It is the main platform that developers have available to sell their applications, and has security mechanisms as well as other stores to prevent the inclusion of malware, and other malicious software, which can harm users.

- **Android updates:** through OTA (On-The-Air) updates, Google provides the latest changes of its system to the devices that officially support the company. It should be noted that in this case, each hardware provider provides the updates it considers necessary to users, which causes one of the major problems of the Google system, and that is **fragmentation**, which consists of a wide variety of versions of the same operating system, many of which are outdated.

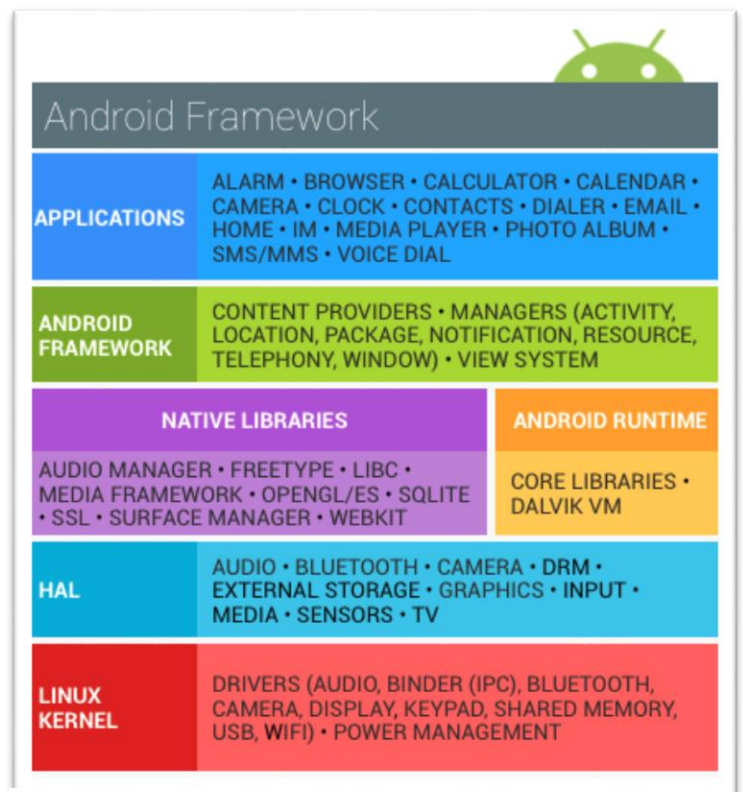


Figure 4.4. Android Framework (Source: Android Security)

- **Application services:** provides backup service for applications that implement this framework.

- **Apps verification services** to prevent malicious content from passing through the official store's security barriers.

- **SafetyNet:** A privacy preserving intrusion detection system to assist Google tracking and mitigating known security threats in addition to identifying new security risks.

- **Android Device Manager:** a mobile and web app, which can locate a lost or stolen Android devices.

Finally, with the release of the latest version of Android (Android 8.0 Oreo) (what's new in the latest Android version [33]), have been included new features to improve user security in this system. Some of these new improvements are: removal of SSLv3, in favor of the protocol TLS, a new restrictive permission called: TYPE_APPLICATION_OVERLAY, which prevents other applications pop-up to deceive the user, and therefore cause some security error. the inclusion of a new "Google Play Protect" service, which verifies and analyzes the third-party apps downloaded from the store, and in order to control the source from which the apps are installed, the "Allow from unknown sources" option has been removed, becoming a special permission to be implemented by the developers, and then accepting by users who want to install application from different stores respect from Google Play.

4.3.1. File System

Android is an operating system based on the Linux kernel, which gives it an internal architecture when dealing with files and directories, which comes directly from UNIX. It follows a hierarchical structure in tree, which has a main or root node, and from which hang the rest of files and folders of the system. One of the fundamental characteristics of this model is that everything is a file from the point of view of internal representation in the system, which confers a certain degree of homogeneity when it comes to accessing the resources provided by today's mobile devices.

Google Android supports several file systems (a detailed overview about Android file system in [34]), such as: ext4, or JFFS2, and it is generally the responsibility of manufacturers to choose one or the other. Although ext4 is the choice when mounting desktop systems, in general for the mobile panorama is usually used by default, the so-called: Journal Flash File System 2 (JFFS2).

Once the format for organizing the data stored in the internal storage of the mobile device has been decided, the internal structure in which the most important directories of the system are distributed, although with some variations depending on the manufacturer, usually follows this model.

-/system: This is the main directory where the operating system is stored. It is characterized by being read-only, in order to avoid its modification for malicious purposes, which could endanger the security of the user, as well as the integrity of the system itself.

-/proc: saves information about the main running processes.

-/mnt: is the single mounting point for the different types of removable media that exist.

-/sdcard: is where the SD card is mounted which is used in some models to expand the main memory.

-/cache: The application cache is stored, and the system itself.

-/data: Probably one of the most important directories, and one that any mobile security researcher has to research with special care, since the apps installed by the user are stored in it. Each one of them, is installed in a separate directory, which through sandboxing mechanisms as explained in previous sections, allows to isolate the scope of each app, to each of the system resources.

4.3.2 Structure of an Android Application

So far we have talked about the security infrastructure in Android and its file system, but now it is really paramount to describe in detail one of the main pillars that take part of any Operating System allowing it to increase its functionalities, we mean: the software. The software in mobile applications is called: apps and it is important to know its structure in depth in order to be able to carry out a security analysis with sufficient guarantees of success.

In the following paragraphs, it will be detailed the set of main parts that compose an application for the Operating Systems: iOS, and Android in addition to developing the main tasks to be performed in any analysis of source code. That parts are cited next:

-To determine the set of software structures that compose the mobile application under study.

-Research, discover, and list the entry points available to that application, and how the information is processed, and stored on the mobile device.

-Locate, and limit the main risks caused either by a poor development of the application under study, or by negligence and misuse due to the security mechanisms provided by the two mobile platforms under study.

The internal architecture or structure that imposes an Android application, allows application developers to make attractive apps for the rest of the users, that perform a certain functionality, and improve the capabilities of their mobile devices. In order to describe how an Android app is implemented, we can say that it consists of a well differentiated series of blocks, which are called: components. There are five different types (Figure 4.5) of components, which are:

- Activities.
- Services.
- Broadcast Receivers.
- Intent
- Content Providers.

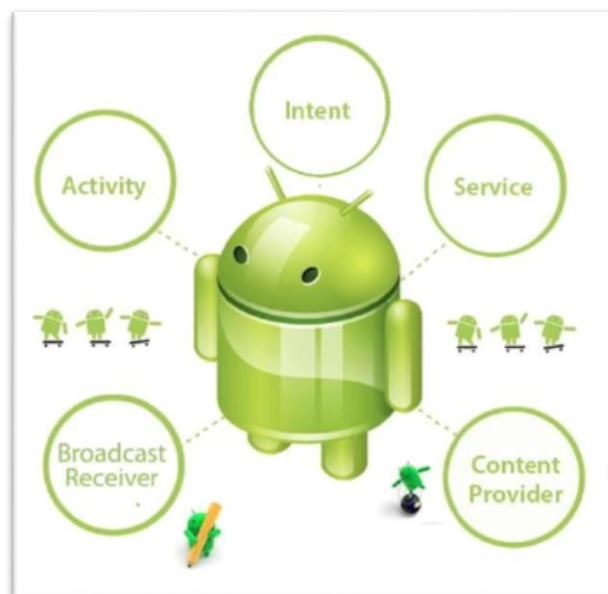


Figure 4.5. Main Android app components. Source: Google4Tech

- Manifest File

Each and every one of them has a specific purpose, and together they form an inner structure that communicates with each other, and allows to develop the set of actions that have been encapsulated in the internal logic of the application (for more information about these system components: [\[35\]](#)).

4.4. Some typical flaws on Mobile Environment

As we mentioned in Introduction, every software has its bugs, and vulnerabilities which a malicious user may exploit or violate, sometimes trying to disturb the user (i.e. DoS “Denial of Service”), but most of the time to get different kind of user information destined to other purposes. Because we live totally connected to the Internet, and with much of our information stored in our personal computers, and mobile devices. Due to the necessity to protect our personal data, it is very important to know the possible attacks, to prevent them. Next, we are going to summarize the most common threats on the mobile ecosystem in our time (for a further research about this, check out: [\[36\]](#)).

- AdWare: is probably the most usual menace in our days, in desktop and mobile environment. It consists in advertisement included in apps or other software installed in our devices, in which sometimes there are a sort of fake apps which are capable to collect personal information from our devices. This information contains not only sensitive data, but also our browser default configuration, and other settings of our dispositive.

- Trojan-SMS: is a fraudulent way used by hackers or other types of users to hide a method of sending SMS (Short Message Service), generally to unknown services created by malicious users and with fees imposed. These Trojans are usually concealed inside legitimate apps and not always share only our mobile phone number but also other usual information saved in our mobile dispositives. It affects particularly to Android devices. This risk vector is also known as: Premium SMS.

- Cloud-based services: used today to upload our personal data like: pictures, documents, calendars, etc to a remote server, freeing up our mobile storage and allowing the synchronization among other sort of devices. Many times, these servers have some kind of bugs, or vulnerability which is used by hackers to get personal information of the clients of that cloud service.

- Zero-day: is probably the most dangerous threats, because it is a recently discovered bug in software which has not been fixed yet, therefore it may be exploited (if it is critical), to obtain user information in a fraudulent way.

- Weak-encryption: is not a bug at all, but it is a negligent way to store information, or communicate with each other through Internet. Sometimes, companies which create software use weak functions to encrypt our personal information, or even methods currently unsafe, because the algorithm is easily cracked. This fact may possibly take advantage by hackers to violate our privacy.

- Phishing: likely, it could be a type of “Social Engineering”, but it is necessary to emphasize in it, because it is very popular above all in desktop panorama, and of course today is spreading to mobile environment swiftly. It is an attempt to get personal

information like: names, credit cards among others, masquerading all this process through a trustworthy electronic entity. This threat is present in desktop and mobile ecosystem.

- Social Engineering: probably one of the most famous attacks used today by hackers to exploit both systems: desktop and mobile architecture, it consists in the use of techniques orientated to imitate certain resource, with the interest of deceiving the users, for collecting private data like: their user id, name, passwords, or even more private information. It's really difficult to detect because user himself/herself granted access of this information to the attacker.

- Botnets: Through a Command & Control server (C&C), they allow remote execution of commands on those infected devices. On many occasions, exploiting vulnerabilities on the affected devices, they even allow attackers to access personal information.

- Ransomware: it consists in encrypting files stored in the device and after doing that, an attacker demands a ransom in order to decrypt the content of the affected mobile device. Generally, this transfer is carried out via untraceable transfers through: Bitcoin, or another crypto-currency platform.

- Applications Vulnerabilities: are bugs that affect libraries, or software that is shared between mobile platforms. They are usually very dangerous, since by their nature they are usually multiplatform. One of the best-known cases, he was informed by the company: "TrendLabs" was the vulnerability of UPnP (Universal Plug and Play) library that is used for video streaming between different devices. This vulnerability allowed the remote execution of code on a device that had an app that used this library.

See more information about main threats on mobile platforms in: [\[153\]](#)

4.4.1. OWASP Mobile Risks last years

Practically every end of the year, OWASP (Open Web Application Security Project) compiles a ranking of the main vulnerabilities found in the web landscape. For a few years now, (specifically in 2014), they have been developing a list of the main threats that have occurred in the mobile ecosystem, on their dominant platforms. In the ranking, OWASP shown in Figure 4.6 [\[37\]](#) includes the risks to which users of these technologies are exposed, along with a brief explanation of the danger of this incident, and why it is included in this classification. During 2016, the main threats to Apple's Operating Systems and Google's were:

- M1 - Improper Platform Usage: in this category is covered the misuse of a platform feature, in order of using the different security controls of the system. These controls include: Android intents, all kind of permissions, TouchID neglect, Keychain among others. It is a security risk very versatile, and it could appear in many situations.

- M2 - Insecure Data Storage: is a combination of category M2 and M4 respect the previous OWASP Mobile Security Risks in 2014. Besides, insecure data storage, it includes the unintended data leakage.

- M3 - Insecure Communication: this section includes: improper or poor handshaking process, incorrect SSL versions, weak negotiation between emitter/receiver of communication, send sensitive information in clear text, and others.

- M4 - Insecure Authentication: is about insecure negotiation with the end-user or managing the session improperly. Some of typical situations in this case are: failing in identifying the user when it required, failure maintaining user identification, or weakness in session management.

- M5 - Insufficient Cryptography: in general, it is recommended to apply cryptography in every communication in the Internet, but sometimes this mechanism is applied wrongly, or to encrypt the assets is used a weak cryptography function.

- M6 - Insecure Authorization: this category is about: failure in the process of authorization by a user. For instance, an app is not authorizing users how it should be, granting access of anonymous when it should reject the connection. This is the main difference regarding the M4 - Insecure Authentication.

- M7 - Client Code Quality: Although It could seem one of the least dangerous categories so far, but it is one of main indicatives of bad coding in mobile platforms. It is completely different of server-code issues. This section includes: buffer overflows, format string vulnerabilities, and other kind of mistakes while coding which require a refactoring process to correct them.

- M8 - Code Tampering: when an app is installed in a mobile device, its code and data is stored within it. If an attacker modifies the code, change the memory dynamically, change or replace the APIs used by the system, or modify the resources used but certain application, then this malicious user could change the normal behavior of the software, and even use it for monetary gain. This category includes: binary patching, local resource modification, method hooking, dynamic memory modification and method swizzling.

- M9 - Reverse Engineering: it consists in analyzing the binary through special tools like: IDA Pro, Hopper, radare2, otools and others, in order to find new vulnerabilities without a fix (Zero-day), gather information about server used for communication, or even find some strings, or private keys in order to decipher the information send by the app. Another objective of reverse engineering is to know how an app is built (external libraries, resources like: images, scripts, among others).

- M10 - Extraneous Functionality: is the last category and is about the inclusion by some developers of backdoors, or other sort of functionalities which should not be released when the app reaches the production process. For example: comment string not deleted from the code which are passwords, not enable two-factor authentication, or even not removing server

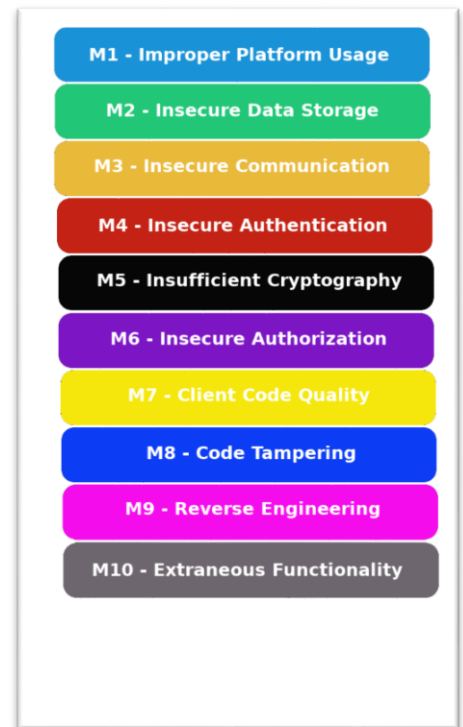


Figure 4.6. OWASP Mobile Risks 2016

If we make a review of the risks that according to the OWASP classification existed in 2014, to see how threats have evolved in two years for the mobile sector, we find that although some of the threats have increased or decreased in importance over the years, generally two years ahead we find the same types of risks that could be found in 2014.

The TOP 10 2014 OWASP [\[38\]](#) ranking consisted in this group of risks:

- M1 - Weak Server Side Controls
- M2 - Insecure Data Storage
- M3 - Insufficient Transport Layer Protection
- M4 - Unintended Data Leakage
- M5 - Poor Authorization and Authentication
- M6 - Broken Cryptography
- M7 - Client Side Injection
- M8 - Security Decisions via Untrusted Inputs
- M9 - Improper Session Handling
- M10 - Lack of Binary Protections

A comparison between the OWASP Mobile Risk 2014 which is shown in Figure 4.7, and the current 2016 ranking can be seen in the image below:

Despite all appearances, the position of some risks has changed its name and position, but with the exception of the former (M1 - Improper Platform Usage) and the

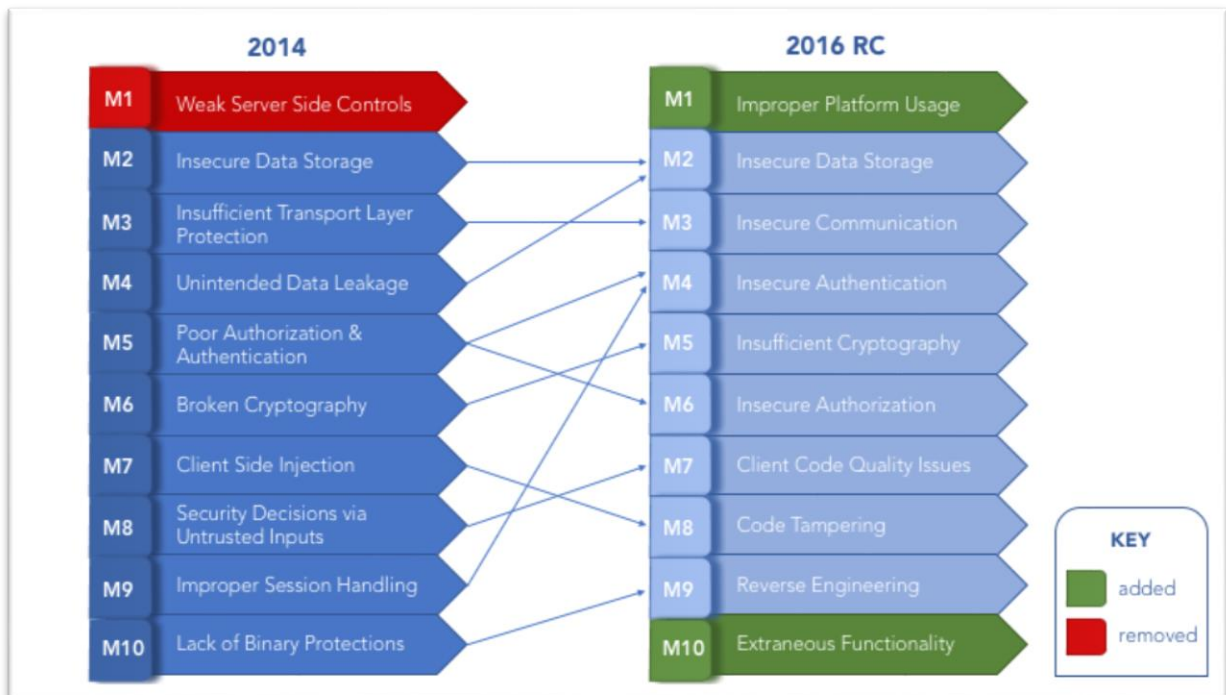


Figure 4.7. Comparative between OWASP Mobile Risks 2014 vs. OWASP Mobile Risks 2016 (Source: Security Innovation)

latter (Extraneous Functionality), the rest are the same as those specified two years ago. The most noteworthy aspect of this new classification is to see how the incorrect management of the platform's resources has become the most dangerous from the point of view of mobile threats, which indicates that in many cases the user, or developers are unaware of the capabilities to safeguard the security provided by the platform, and negligently acts to cause the risks and /or threats that may increase exponentially.

Apart from the new risks, it should be noted that threats such as "Insecure Authentication" have significantly increased their activity, becoming one of the main ways by which an attack can obtain information from mobile platform users. Therefore, it is necessary, a quick action on the part of the developers, to implement the latest technologies in security matters, which provide the necessary guarantees to protect the communications of users of their applications.

Last, but not least, the increase in the risk hazard of: "Insecure Data Storage" again denotes that in the field of development, programmers must implement the latest and most secure cryptographic methods to encrypt the information that their applications store on devices, for each day the data stored on smartphones are more personal, and therefore require some kind of safeguard.

4.4.2. Types of threats: Inducted by the user or Triggered

The type of risks that can be faced by all users of any mobile platform can be of several types: those that are caused by ignorance or inexperience of the user, due to a lack of knowledge of the platform itself which conditions that can be deceived by malicious users who want to take advantage of it, or those that either by physical or remote access

allow an attacker to violate the security infrastructure of a particular device, in the interest of some criminal activity. Next, the previous problems will be described in more detail.

a) User inducted threats

As has been demonstrated on many occasions, the weakest link in the chain is always: the user, and this makes it one of the main targets of hackers, and other malicious users, when they want to breach security on mobile devices. That is why, nowadays, one of the most used attacks is the so-called "Social Engineering" that seeks to deceive the victim, so that without needing to violate the security of their electronic device, users themselves gives access to their personal information.

In addition to this type of techniques so used today, another of the main risks caused by the user himself, is the ignorance of the security mechanisms offered by each of the mobile platforms, resulting in misuse of mobile devices.

Some examples of this behavior are: having the automatic blocking of your mobile device disabled, using your smartphone in unprotected public WiFi networks, downloading applications from unofficial stores or unknown sources, storing private information in memory, or not even activating the blocking and remote deletion in case of theft. All this set of risks can be remedied by educating users about the facilities provided by the different platforms, besides: restricting those activities that can be carried out by the user to avoid possible disasters.

Another of the risks caused by the user is when he/she accepts the call: BYOD (Bring Your Own Device), in the company where he/she works. Once your device is used to store corporate information, such information can be filtered through coexistence with other user-specific means. For example, the user could use corporate emails by mistake, or even if infected by any malware, cause important company information to be leaked to the network.

Within the iOS ecosystem, and also in Android you can find other types of threats called: Jailbreak, and Root (information about these two procedures in [39],[40]) respectively. In both cases, the user uses a software that takes advantage of vulnerabilities found in the Operating Systems of their devices, to make an escalation of privileges, and allow to install applications, or access certain functions that are blocked from the factory. Many times, this procedure is performed to install applications through unofficial stores, or for purposes such as piracy, but despite the benefits that may seem to have unlocked your smartphone, many times the user himself only gets to create an entry path for malware, infecting your mobile device.

In addition to all of the above, it can also become a risk for the user to install applications for recording telephone conversations. If you use the Google Operating System, this type of apps requires the user's permission to access the microphone on your smartphone, while iOS requires the device to have jailbreak. When these applications are installed, they store user-conducted conversations in the storage of the mobile device, and if the application is malicious it may use these data for criminal purposes. In addition to the above mentioned, in many occasions an attack can even listen to the communications carried out by user, by means of "Phone Cloning" methods, which consist of cloning the victim's SIM (Subscriber Identity Module) card, and so on, provided that if the user is

connected to a 2G/3G network, an attack can be carried out that allows the interception of their calls, SMS, and all type of information sent through their mobile device.

2. Attacks with physical access to the device

In general, when we talk about the main threats on mobile devices, people tend to think of attacks remotely carried out by hackers, or malicious users, but on many occasions these attacks are physically perpetrated and can even become much more harmful than those carried out over the network.

Some examples of attacks of this kind in Android and iOS are the following:

- Android Cool Boot Attack [41]: it consists of taking advantage of how RAM works in this type of mobile devices, and in general in any computer. After switching off the device, the electrical current stops flowing and the data is then gradually removed. Temperature is a fundamental factor in the speed at which this information is erased, gradually increasing the amount of time that this information is lost as the colder the temperature is in the RAM memory of the mobile device. You know this, an attacker can place the Android smartphone in the fridge for an hour, and then connect it to a computer with a Linux distribution via USB. During the loading process of the Operating System, a module containing RAM memory information is loaded, thus enabling the retrieval of all types of information (including user credentials) that are stored in the memory.

-Although security at WhatsApp, probably the most widely used messaging app in Spanish-speaking countries has improved its security in recent years, adding peer-to-peer encryption to communications over the Signal protocol, and adding encryption to the local database where user conversations are stored, some critical vulnerabilities to user security and privacy have been discovered in recent years. One of these vulnerabilities is that even though the conversations database is encrypted, the user's database encryption key is created during the app's first run and stored as a parameter within the program. This key can be extracted by programs created by some developers, thus allowing the user to have physical access to the phone, or get the user by Social Engineering methods to pass their conversation database, the ability to decrypt and read communications that have been established on the victim's mobile phone.

Another of the best-known attacks that were carried out until the appearance of the methods of mobile unlocking by fingerprint, were: the obtaining of the code through brute force, or attacks called "shoulder surfing" to discover the pattern of unlocking the device, and so if there was physical access to it, be able to extract their information easily, bypassing that safety barrier. Specifically, within the Apple platform appeared two solutions (one hardware and another software) that allowed to discover the unlocking code of the iPhone, in order to unlock it. The distressing hardware called: ScreenLock (more information about these device [42]), tested four-digit combinations when connecting the device via USB. If ScreenLock detected that the code was wrong, it would turn off the iPhone to prevent the failed attempts counter from having any effect, and then try another code again until it found the correct one. Although the attack could last more than a hundred hours, in the end it succeeded in successfully violating phone security. In the case of the software application, it had a similar operation, but as the only disadvantage it was indispensable that the device had Jailbreak.

In short, as has been demonstrated, threats on mobile platforms can come from several fronts, and physical access is certainly not one that should be underestimated.

4.5. Latest Threats on Mobile Platforms

Once it has been stated that today's threats on mobile platforms exist, and affect users, due to the nature of this project which focuses on the two main OS that dominate the mobile ecosystem today, the latest and most dangerous attacks suffered by users on these platforms will be described.

4.5.1. Last Threats on iOS infrastructure

There has been much activity in recent years on iOS devices, with some cases with great impact and very commented in press and some forums, and other cases lesser known but equally important. These issues have compromised the personal information of many users in the world, have received generally a lot of attention, and in the case of Apple have been fixed relatively quick, but despite of it, the sensation of insecurity while using these mobile devices to save our personal data remains. Next, we are going to relate chronologically the troubles suffered in this platform in the latter years.

We started the year 2014, with a study conducted by IOActive (see [43]), on the status of major mobile banking apps, and came to very discouraging conclusions for the privacy of its users, as it was found that half of the apps analyzed were vulnerable to Cross-Site Scripting (XSS) attacks on the client side, which a quarter had hardcoded in the code of the app, almost half had hardcoded credentials in the code of the app, almost half were storing, almost half of them stored sensitive information in their logs, and that slightly more than a quarter of the apps analyzed either did not use SSL certificates to protect their communications, or they did not encrypt sensitive information.

But the most famous case happened in August 31st, 2014 and It was called soon: “The Celebgate scandal” [44]. It consisted in the leak of more than 500 private pictures from celebrities all over the world but specially actresses from U.S.A. This incident was committed by a little group of hackers who hacked into celebrities iCloud accounts revealing during the last day of August, and first of September a large amount of photographs uploaded in Apple cloud system. At the beginning, many people blame iCloud for not having enough security to store user files, but finally after an investigation, Apple claimed that the problem was the weakness of the passwords used by the affected users. That proved the need to strengthen passwords for our online accounts, to prevent future hacks.

In November 2014, FireEye a mobile security research team, reported to Apple a new vulnerability called: Masque attack [45]. which allowed an attacker to bypass the security system designed by Apple, installing malicious software, pass itself off as legitimate. This substitution may provoke the exfiltration of personal information, saved by the user in his/her device, if this attack was successful. Generally, this issue affected users with iOS developer accounts who had special permissions to install certain apps for debugging and developing.

Also in late 2014, an advanced Chinese malware called: Xsser mRat [46] appeared, which was reported by Lagoon Mobile Security researchers. This dangerous Trojan horse

was theoretically able to obtain any kind of information stored on the mobile device, including: phone contacts, precise location data of the user, instant messages and SMS, as well as logs and credentials stored in the device's internal memory. Using "Social Engineering" methods, the attackers used messages for the distribution of this malware through WhatsApp, which included a link that would be clicked by the user in question, downloaded an app that if installed on a jailbreak device, the smartphone was then infected.

Speaking of malware, 2015 was a very busy year for iOS. Firstly, it was the ability to access the contacts number from the lock screen, simply commanding Siri to tweet through the official Twitter app. In September 2015, a malware named: YiSpecter [47] which affected users from China and Taiwan. This software changed the settings of the user dispositives, installed apps without asking for permission, and sent personal information to third-party servers. Almost a month after this issue, another malware named: XcodeGhost [48] showed up. This malicious code was provoked by the installation of a modified Xcode (IDE to compile iOS apps) from unknown sources. That version was modified and when developers created an app, it contained a code which was able to send personal information of the user like: IP address, credentials from the device and other kind of data. The worst thing about this problem was these apps managed to overcome the limitations of the AppStore.

In 2015, there was also a significant leakage of information from some mobile applications, specifically one of the most notorious was the obtaining of credentials and location of some users using parking payment applications in the United Kingdom. The protocol used by these apps, despite being TLS, did not check the certificate used by the server, which allowed malicious users to make a Man-in-the-Middle attack, and therefore intercept all data sent by users of those mobile applications.

Another problem which accompanies iOS practically since the first version is the process to unlock the device called: Jailbreak (See "Glossary") (a more detailed current situation about jailbreak can be seen in [49]). Thanks to this procedure, users manage to install apps from different sources other than AppStore itself. Sometimes, if the origin is not reliable, it brings some troubles like: the installation of malware, trojans or another annoyance to the system. Nowadays, there is no jailbreak for the last version of iOS (version 11.0.1), but the last version of the jailbreak which works on iOS 10.x was developed by a Chinese group called: Pangu team. The latest versions of iOS that can be used for Jailbreak are: 10.3 and 10.3.1. Once again, the procedure was carried out by Chinese researches: Pangu Team and TaG team, and is compatible for iPhone 5S and higher, and for iPad Air and upper tablets. Although some security researchers reported problems found in later versions of iOS, such as vulnerability to processing large text strings (CVE-2017-7047), no tools were released that could exploit these issues with guarantees. One of the latest milestones in the jailbreak panorama for Apple devices was shown at the "Def Con 25" in July 2017, where a jailbreak for the smartwatch Apple Watch was presented, which would be functional in watchOS3 and above. Following the release of the latest version of the iOS 11 operating system in September 2017, there is no news of a jailbreak release adapted to this latest compilation soon.

Following with the threats, it was reported the appearance of a new malware called: AceDeceiver [50], which unlike other malware that originated in the installation of applications from unofficial stores like Cydia, this affected non-jailbroken devices. To

access these types of devices, malware used a desktop application that exploited a design flaw in the Apple-implemented DRM (Digital Rights Management) to download a malicious app from the AppStore. What was really novel about AceDeceiver was that even after it had been removed from the system, it was able to download that malicious application from the official store, and it didn't even require negligence in the use of enterprise certificates, which had been the gateway to infections on other occasions.

In early September 2016, when Apple released its final revisions for its current mobile OS, one of the biggest problem in the iOS ecosystem ever, appeared. The malware in question is called: Pegasus [51], which was presumably developed by an Israeli company named: NSO Group. This group claimed that its mission is: to create advanced tools to help authorized government, to fight against crime and terrorism. It is believed that this malware works since iOS 7, that is two years ago, and It allows to the hacker: to record sounds, collect passwords, track the device, read messages, e-mails, contacts, and even the call registry. Therefore, Pegasus is a high-level spyware that takes advantage of three iOS vulnerabilities which were fixed in the latest version of nine series: iOS 9.3.5.

More recently, in late September 2016 when the final version of iOS 10 came out, Elcomsoft a russian company specialized in security, discovered that the encryption for backups in iOS got worse respect the previous version [52]. Specially, the current method of encryption in iOS 10 uses: SHA256 (Secure Hash Algorithm), a cryptographic hash function which is a lot easier than PBKDF2 (Password-Based Key Derivation Function 2) algorithm used in previous version of the OS. Researchers said: if the iOS device backup in iTunes is protected with a password, it is approximately 2500 times faster to find it out, compared with the old mechanism. It may look like very dangerous, but if an attacker had physical access to the machine, or got the iTunes backup, doing brute-force **attacks** with a pen-testing tool, then it might recover the original password, decrypt the backup and retrieve all the information stored inside it. After the knowledge of this fact, Apple pronounced saying that: They are aware of this situation, and promising a coming update to solve it. When the release in later October of iOS v10.1 [53], Apple solved this issue, removing the weak hash method used, and replacing it with another hash function not vulnerable to this type of attack.

Following the trend imposed in 2016, 2017 has also been a year of security incidents on the Apple platform. As reported by Motherboard in February 2017, Cellebrite [54], an Israeli firm specializing in mobile phone information extraction, for law enforcements agencies. Cellebrite managed to extract more than 900GB of information from smartphones, making use of exploits that were used when developing tools to perform jailbreak on iOS devices such as: limeraln or QuickPwn, with some additions such as the addition of code to perform brute force attacks on the PIN (Personal Identification Number) code of the mobile device in question. This exfiltration of information, reminiscent of the incident with Pegasus in September last year, shows how exposed users are to this type of attack, and the need to keep mobile devices updated, in order to solve the possible vulnerabilities that have been found in them.

Later, in March 2017, as reported by Lookout, they warned that some scammers had taken advantage of the possibility of showing pop-up windows in the Safari browser, to block the web browser and prevent users from using it in the future. The attack in question demanded money from the user, in the form of iTunes Gift Cards [55], if he/she wanted to see the browser unlocked, and be able to use it again, threateningly and trying

to scare the affected user, so that he would pay the imposed fee. As often happens on these occasions, an Internet user succeeded in unlocking Safari by simply deleting the browser cache from the iOS settings, demonstrating that this annoying attack does not encrypt any information as it does with ransomware, and was simply meant to annoy the attack victim. However, with the output of iOS v10.3, Apple changed the behavior of pop-up JavaScript windows, isolating their action per tab, to prevent them from collapsing the web browser.

In addition to the previous paragraph, in March of this year, Wikileaks released some CIA (Central Intelligence Agency) documents called Vault7 [56], dated from 2013 to 2017, which among many other systems, included information on tools for hacking devices with iOS operating systems. The filtered documentation included research to discover new ways of exploiting devices with iOS, as well as a list of exploits classified according to the version of Apple's system they affected.

One of the most recent vulnerabilities, which in fact was corrected in the last update of the iOS 10 series (v10.3.3), was the possibility that an attacker within the WiFi reception radio was able to execute arbitrary code by corrupting the device's memory, and was corrected by improving the system's memory management. Although it was a hardware vulnerability, this time caused by Broadcom's WiFi chip [57], hackers were able to bypass chip control and access the system, which proves once again the ingenuity and variety of ways threats can be presented today.

With all these examples, you can see that even though Apple's system has always had a reputation for being safer than Android, and in fact establishes more restrictive security controls than Android. is not free from threats, and these can be equally harmful to their users, since they have proven to be advanced enough to exfiltrate personal information from the infected devices, and affect the privacy and security of their mobile devices.

4.5.2. Main Threats on Android in the last years

Because it is the most widely used mobile operating system in our days, the proliferation of malware and other malicious software for Android, has increased substantially in recent years, and today the Google system is one of the main objectives of hackers, surpassing Windows that had been, by its volume of users, the main focus of attacks in recent decades.

One of the first cases of security flaws in Android occurred in 2013, specifically to the company Moonpig [58], a greeting card vendor. In particular, the API (Application Programming Interface) used by the company in Android app, does not verify credentials by allowing access to personal information, to any user. Although the failure was reported in 2013, it was not resolved until 2015, which shows the lack of interest that some companies put in security, and the privacy of their users, and also its users were not informed of this serious security breach that compromised their personal data.

Since its very beginning, due to a more permissive control over apps that were accessible from the Play Store, Google has had problems because some malicious users have managed to break the barriers imposed by the company, uploading malware to the official store of Android. In fact, one of the most dangerous and most damaging

vulnerabilities (affected nearly 95% of Android users by the end of 2015) was the malware called: "Stagefright" [59], which allowed a malicious user to take control of the victim's terminal remotely, unnoticed by the victim. Although Google quickly offered a patch for this vulnerability, and most manufacturers were quick to adopt it, it showed how vulnerable mobile users are today.

Already in 2016, according to a study carried out by TrendLabs Security Intelligence in July, there are more than 400 potentially dangerous apps that could affect our device. The main cause of this massive infection was a virus called: DressCode [60], whose primary objective was to obtain clicks and benefits in advertising websites, but according to the researchers, it could not be ruled out that its use could spread to more harmful purposes for the user. Google opened an investigation, and DressCode was removed from the store, but many users were infected in that time.

But before the end of 2016, Android was again hit by a new malware called: Gooligan [61], which affected all versions of the mobile system until: Lollipop. Unlike other malware, Gooligan took advantage of hacked devices not to collect personal information from them, but to download third-party applications that could generate advertising revenue. The main danger of this malicious software is that once it was installed, it gained full access to the device, becoming a great danger for privacy and security for the user.

In addition to the Play Store, another of the main malware gateways in Android, has been by terms of malicious software installed in some cheap mobile phone manufacturers, which specifically came from China. The ability to install the system on other dispositives not controlled by Google, has always been one of the main weaknesses of the platform, not only causing the usual fragmentation of the system, but also causing the installation of unwanted content in the system, which is aimed at infecting the user's device. According to DrWeb research, many of the Chinese mobile firmwares, including brands such as: Lenovo, include malwares and/or Trojans whose only goal is to exfiltrate personal information from users. The Trojan was installed under the name of: Android.Downloader.473.origin or Android.Sprovider.7, and after its execution proceeded to download and install apps loaded with ads, which worsened the user experience, and that in many occasions gets harder the process of uninstallation, and removal of the system.

Another of the main threats within the Android panorama is phishing, which is the same thing to supplant the identity of an entity, in order to obtain personal data of the users to whom this dangerous attack is destined. In general, phishing is aimed at supplanting financial organizations, and usually done via e-mail, but lately with the proliferation of smartphones, apps like WhatsApp are used to send files with malicious content, which imitate the entities, ask for personal information from users as their PIN number, which is then used not only for information theft, but also for scams.

Along with all that described above, another of the main sources of infection in Android, is the installation of apps from non-official store, which do not have the control that Google has over its official store, and can therefore include malware whose goal is to infect the user's mobile devices. Although 2016 was a year full of security and privacy issues in Google's system, 2017 did not start better.

In early 2017, a dangerous Trojan named: Tordow [62] began infecting certain Android terminals. The Tordow's modus operandi consists on identifying the terminal's brand, and then using a series of exploits discovered in Android, to achieve root privileges, to have total access within the system, and steal data bank information. Some research on this malware also suggested that it might even act as: ransomware, a threat increasingly present in the mobile world. This proves once again, that in order to protect yourself on any mobile platform, a fundamental recommendation is to download apps from the official store, because other alternative stores may include fake apps, including viruses, or any other type of malicious software.

But problems of Android had only just started, because also at the beginning of this year, appeared a new malware called: Skyfin [63], which was able to download and install applications, without asking the user's permission, which were usually apps full of advertisements, so that the authors of the malware described would benefit by clicking on such ads.

January was a month full of threats, and at the end of January 2017, it was released a ransomware hosted in an app called "Energy Rescue" [64], which aimed to improve the battery life of the mobile device. Once the application was downloaded, it was able to access the contacts, and SMS sent by the user, and after doing this it proceeded to block the device, and display a message, warning the user, that if he did not pay, his information would be sold to the black market, and could not unblock his smartphone.

And more or less at the same time as the end of this threat, an evolution of an ancient Android malware called: HummingBad, made its appearance under the name of: HummingWhale [65]. According to the security company: CheckPoint, HummingWhale infected more than 20 apps inside the Play Store, proving once again that even the official store is not safe. The malware in question used a complex procedure that consisted of creating a kind of virtual copy of the installed application, which could benefit the creators of the software through advertising. In addition, without user intervention, I was able to write positive ratings in the official Android store, to encourage other users to download the apps, and thus spread the malware more quickly.

In mid-March 2017, again the company specialized in security: CheckPoint, warned of one of the most advanced malwares that have affected the Google platform, which was called: Skinner [66]. This malware would be able to track the user's location, as well as log all actions performed by the user whose device is infected by Skinner, and allow remote control of the infected device. Following CheckPoint's warning, Google removed the affected app from the Play Store, but did not prevent it from already infecting more than 10,000 users. Once installed, Skinner used binary obfuscation methods to prevent system antivirus software from identifying it as a potential threat. After running the app, the malware sent information about the user in question, as well as information about which applications were open at that time, to a remote server. Skinner then deploys ads on the app, in order to obtain benefits for the malware authors.

In April 2017, a rare case of iOS malware evolved into a variant directed at the Android system, which was called: Chryasor [67]. This dangerous malware was able to spy on calls, as well as send information about the user's location, read SMS messages on the device, record through the device's microphone, and even turn on the camera. This

software was created by NSO Group as well as Pegasus malware in iOS, and was so advanced that it could also be deleted itself to avoid detection.

After a few months of peace and quiet, in June 2017 two more threats appeared in the Android ecosystem. The first was a malware called: Xavier [68], which according to the security company TrendsLab, was able to collect personal information from users. In that same month, another malware called: GhostCtrl [69], discovered by Trends Micro researchers, was able to collect information through the microphone of the mobile device and its cameras. This malware camouflaged itself in trusted and popular applications such as WhatsApp, or Pokémon Go, and after being run by the user, it acquired super-user permissions, and had the ability to block it completely.

Finally, in September 2017, a new multiplatform malware detected by Armis Labs appeared, capable of affecting several systems, among which it is necessary to stand out: Android, Windows, iOS, macOS and Linux. By taking advantage of a Bluetooth vulnerability, it allows a malicious user to connect to and remotely control the device that has it turned on, without the need for the user to be connected to the Internet. Although this extremely serious vulnerability has already been corrected in the latest versions of the system, the fragmentation present in the platform, makes billions of devices that do not have access to the latest versions of the OS, and therefore they are exposed to infection irremediably. Also in September 2017, a new and dangerous malware has appeared in environments: Android, called: "BankBot" [70], which camouflaged in a game: "Jewel Star Classic", gets through Social Engineering techniques to collect the bank details of the app user. To achieve the deception, the application requests after a while playing the game asks the user to enable access to accessibility options by impersonating Google service, and allowing from that moment onwards to install applications without the user's consent. The next time the user access the official Android store, he/she will be prompted for his/her credit card details, which will go directly to the malware-enabled servers.

After everything described in this section, it is clear that threats on mobile platforms such as Android are increasingly present, and that they require a rigorous care on the part of users of this platform.

4.6. Vulnerabilities on both platforms

We use the web page of CVE Details [71], which provides information about the number of vulnerabilities on desktop and mobile platforms to enumerate, and show an evolution about the software issues on iOS and Android during the last years.

4.6.1. Vulnerabilities on iOS

According to CVE Details, and as shown in picture below, vulnerabilities on iOS have surpassed last year's vulnerabilities, which is not only a worrying fact, but also indicates the enormous popularity of today's mobile devices

Year	# of Vulnerabilities	DoS	Code Execution	Overflow	Memory Corruption	Sql Injection	XSS	Directory Traversal	HTTP Response Splitting	Bypass something	Gain Information	Gain Privileges	CSRF	File Inclusion	# of exploits
2007	1		1	1											
2008	9	3	2		1						2				
2009	27	10	6	2	4		2			3	7				
2010	32	14	14	9	6					5	3	2			
2011	37	13	10	5	6		3			2	11	1			
2012	112	74	69	64	60		7			13	9	1			
2013	96	58	50	42	47		4			17	9	1			
2014	122	50	51	35	33		1	1		20	25	4			
2015	387	232	211	183	191			5		44	63	13	1		
2016	161	107	78	85	75		3			8	39	11			
2017	290	177	169	151	140		12			29	47	4			
Total	1274	738	661	577	563		32	6		141	215	37	1		
% Of All		57.9	51.9	45.3	44.2	0.0	2.5	0.5	0.0	11.1	16.9	2.9	0.1	0.0	

Figure 4.8. iOS vulnerabilities from 2007 to 2017 (Source: CVE Details) [72]

Regarding to this table in Figure 4.8 and Table 4.1, the distribution of vulnerabilities of iOS, from its first version until now, is as follows:

iOS	
Version	Number of vulnerabilities
1.x	1943
2.x	1458
3.x	2300
4.x	2992
5.x	938
6.x	592
7.x	879
8.x	334
9.x	274
10.x	287
11.x	TBD
Total:	11997

Table 4.1. Distribution of iOS vulnerabilities [73]

When iOS was the first OS for smartphones and mobile devices of new generation, we can see the number of vulnerabilities are about a thousand, but as time goes by, and Android devices appear, the number of issues has been decreasing, until being around a two hundred per year approximately.

4.6.2. Vulnerabilities on Android

In the case of Android, we observed an even higher increase, becoming the most vulnerable mobile phone of the moment, due not only to a security strategy more permissive than the Apple System, but also to its greater number of users, and its degree of fragmentation, which means that most devices that use it are not updated with the latest security corrections. In the Figure 4.9 and Table 4.2, we can see a brief summary.

Year	# of Vulnerabilities	DoS	Code Execution	Overflow	Memory Corruption	Sql Injection	XSS	Directory Traversal	http Response Splitting	Bypass something	Gain Information	Gain Privileges	CSRF	File Inclusion	# of exploits
2009	5	3								1					
2010	1	1	1												
2011	9	1	1		1					3	2	3			
2012	8	5	4	2							1				
2013	7	1	2	2	2					1	1	3			
2014	13	2	4	1		1				1	2	2			
2015	125	56	70	63	46					20	19	17			
2016	523	104	73	92	38					48	99	250			
2017	616	76	186	104	29			1		30	75	34			
Total	1307	249	341	264	116	1		1		104	199	309			
% Of All		19.1	26.1	20.2	8.9	0.1	0.0	0.1	0.0	8.0	15.2	23.6	0.0	0.0	

Figure 4.9. Android vulnerabilities from 2009 to 2017. (Source: CVE Details) [74]

As same as the iOS section, Android’s vulnerabilities since its inception is:

Android	
Version	Number of vulnerabilities
1.x	50
2.x	241
3.x	87
4.x	3601
5.x	1536
6.x	1090
7.x	1052
8.x	20
Total:	7677

Table 4.2. Distribution of vulnerabilities in Android [75]

The distribution of vulnerabilities on Android (Figure 12) is completely the opposite of iOS. It started with a half of a hundred issues, but year after year its number have increased, being today around a thousand of vulnerabilities. The reason for that, it is because today Android is the most used Mobile Operating System in the world, and therefore is the main target for hackers and other malicious users. One more thing, we can see in this table is also the fragmentation on Android ecosystem, because the latest and more up-to-date version have less vulnerabilities than an obsolete, but one of the most used version on Android devices is: version 4.0 (Froyo).

4.7. Need of a Security and Forensic Analysis

After presenting in previous chapters, the internal architecture of the main Mobile Operating Systems, and as in spite of all the mechanisms that are available to developers and users, issues continue appearing which endanger the security of global users; because one of the main objectives of this project is to make users aware of the main threats that are present in the mobile ecosystem, we will proceed to create an infrastructure that can allow the user, whether it has technical knowledge or not, to perform security analysis on mobile applications, in order to assess whether there are threats that endanger the user of these applications or not.

To this end, throughout this extensive section, the concepts of Security Analysis and Forensic Analysis will be developed, as well as the types in which both can be classified, to show how, when combined, they can help a user to determine the main risks to which they are exposed using mobile platforms.

4.7.1. Security Analysis on Mobile Platforms

Mobile security analysis is a procedure by means of which a security researcher determines and checks the security mechanisms available in a mobile platform, in order to discover whether the implementation of these security methods is achieved, for protecting its users.

The requirements that are demanded, vary substantially depending on the type of platform, the group of people to whom it is directed, and even in the current legislation of each country. This is why, before carrying out the analysis itself, any researcher must make a detailed assessment of all the aspects mentioned above, as well as an exhaustive study of the environment in which he or she is going to work. Only a detailed knowledge of the hardware and software to be worked on provides the necessary guarantees, that the job will be carried out successfully.

In addition, care should be taken to perform the analysis on final software versions and to avoid alphas, betas or other early development software stages at all costs. Once this analysis is completed, it should not be considered that the procedure has been completed, as due to the current diversity of the software and its complexity, it is extremely complicated to perform a seamless analysis. Therefore, priority should be given to the use of automated tools, which save time and avoid errors for mobile security researchers, and it is also highly recommended that all professional analyses carried out on any software platform, whether desktop or mobile, be carried out by several people, so that they can help debug the possible errors that their respective investigations may entail.

This section of the project cannot be completed without listing and briefly explains the different existing techniques for developing a mobile device analysis, which are equivalent to the methods used in desktop environments, and which provide information about the intrinsic security of the platform, as well as possible errors that may have occurred during its development stage.

Classified according to the information accessible by the researcher, we can highlight (see [76]):

a) White box testing: source code and some documentation of the system to be analyzed are available. Whenever there is the possibility of accessing the source code of the application, an exhaustive review should be made in order to search for errors and/or vulnerabilities that could endanger the security of users. In order to develop this, with guarantees, it is necessary to determine precisely what the internal structure of the app is, its functionalities, what elements of the system it affects, and which errors during its development, can be the gateway to attacks by malicious users.

In order to look for vulnerabilities of any kind, the investigator should focus on all those that according to the main standards, and specialized entities are more harmful to the user, such as: lack of validation of input parameters, **weak encryption** algorithm in communications, lack of encryption in personal data stored within the system, among others. Even with this set of procedures, it should never be forgotten that many of the errors can occur during the execution phase of the software to be analyzed, so it should not be concluded in any case after all the actions carried out in this phase, that the software has a specific number of issues, since in later phases this number should vary considerably.

b) Black box testing: although there is a final version of the software available, the documentation provided to the analyst is certainly very limited, which does not guarantee that the researcher has full control over the system to analyze. Using other techniques such as Reverse Engineering, can access the source code of the application, which could help greatly, to do a more rigorous analysis.

This type of analysis without knowing information, generally, will lead to the performance of a penetration test, also called: Pentesting [77], which implies that the researcher since there is a partial or total ignorance of the system, decides to test it from the outside. In order to guarantee this procedure, automated tools are usually used to carry out a battery of tests on the system under analysis, as well as in some cases, and whenever possible (the code is not obfuscated), the researcher could use techniques of "Reverse Engineering", which, in the absence of detailed knowledge of the application, can help to improve their knowledge about its internal functioning, and even to discover vulnerabilities in it, that can enrich the overall performance of the application.

In addition, and due to the great variability and portability of the mobile environment, compared to the desktop ecosystem that has been the predominant one until a few years ago, it should be taken into consideration that both the entry routes: wireless connections, as well as Bluetooth, NFC (Near Field Communication) among others can be the entry point for many malicious users, who can take advantage of exploiting their vulnerabilities to violate the system, such as the information stored by this set of sensors, which must be taken into account, to do a rigorous security analysis.

Therefore, in order to deal with this varied ecosystem, the main resources to be investigated will be: how these devices are transmitted through and if they are properly protected, research into data leaks risks due to the use of certain applications installed in the system, as well as permissions and what resources the set of system apps have access to, in order to limit the scope and impact of the them.

In order to perform this set of actions in the appropriate way, there are two types of security analysis [78]: Static Analysis (which is performed without the application

running on the system), and Dynamic Analysis (which cannot always be performed, and is the one that gives more information because it consists of evaluating running applications).

In the next sections, we will look at what they are, and the main features of these two important methods of analysis in mobile technologies.

When performing a mobile security analysis, as in the rest of the environments, there are two different types depending on the characteristics and status of the application to be analyzed by the researcher. These types are static analysis [79], and dynamic analysis [80]. Each of them are described in detail below.

a) Static Analysis

This first type of analysis, is the most limited and from the point of view of an investigation the one that less information can give to a security researcher, since it implies to carry out an evaluation on the software without executing, simply using the source code (if it is available, since many times it is not open-source), and if not, using techniques of "Reverse Engineering", in order to study the binary of the application, and to be able to obtain a representation of it.

In order to be able to carry out this type of analysis, the analyst must take into account not only the source code mentioned above, but also and more important in our days that most of the applications are connected to the network, in the set of metadata that these apps send when they are communicated in a normal way in order to be able to carry out the task for which they have been programmed, since many times in this information there are private and very sensitive data of the user, that if they are not handled with the minimum guarantees of security, could end up in a serious risk for the security of the user of a mobile platform.

In addition, both in iOS (configuration files) and Android (Manifest files), special care must be taken when analyzing, as these files store the routes to the main resources consumed by the app itself, such as: databases of sensitive information or not, stored by the user, set of resources: images, audio, or other multimedia content, as well as other information, and other types of data, that after a rigorous analysis can constitute for a researcher if it is done with good ends, or for a malicious user if it does with bad, a great source of information when it comes to knowing how a certain piece of software stores its data.

b) Dynamic Analysis

This second type of analysis is a way to be able to identify the behavior of a mobile application in a more real way, since the main difference with respect to static analysis is that it is done on the app while it is running in the system. This mainly provides a very valuable set of information, since the analyst does not have to infer how it acts, nor what the role of an app is, but observing it, analyzing it, and collecting this information in real time, which is a great advantage over the previous type of analysis, since it is possible to do a more legitimate analysis, in spite of the fact that the security analyst does not have the source code of that binary.

The main characteristics of the dynamic analysis are that they allow the researcher to obtain information from the memory of the system when the application is loaded into it, being one of the main reasons why the information obtained in this analysis is of great value, since the content that an application stores in RAM memory, once it is dumped, if the information has not been sufficiently protected, can be one of the main sources of information that a researcher uses when carrying out the analysis.

As if this were not enough, the dynamic analysis gives other advantages more, such as: the study of the program's execution flow, as well as its communications through the Internet, so it can give a global view of the activity performed by a certain app in the system, allowing to discern if it performs some kind of strange behavior or activity, when the user decides to run it on his smartphone or tablet.

But not everything could be advantageous, since in order to be able to carry out this type of analysis, the security researcher requires a hardware device or an emulator as the first essential requirement, in order to be able to execute the set of applications under study.

Moreover, in Apple's platform, this type of analysis is often not possible on a conventional iOS device due to the level of encryption of the platform, but Jailbreak must be applied in order to install some software in the tool, which allows to carry out the procedures that this analysis requires. In the case of Android, many times it is also needed a device with root, so as a first approximation, although the benefits of dynamic analysis are many, we show a scenario that is often very complicated to replicate. However, once these barriers are resolved, the researcher can access practically all the information that could be obtained in its static counterpart.

4.7.2. Forensics Analysis on Mobile Platforms

Although it may seem that forensic analysis (for a further see on [81]), is not one of the objectives of this project based on security and privacy in the main mobile platforms of today: iOS and Android, during the course of this chapter will explain the reason for the decision to include it in this Master Thesis, in addition to presenting a series of tools that will help the security analyst during its research on a mobile environment, besides to understand why these tools will help to substantially enrich the security analyst.

Forensic analysis within the world of computer science is a work methodology that allows us to investigate by obtaining information and evidence of an incident, the causes of it as well as what or who originated it.

The main purposes of forensic analysis, which make it a very attractive mechanism to include in any research project, and which make it a fundamental part of this thesis are:

- It allows to collect the information left behind after a criminal activity in order to be able to present it to the interested parts. In this case, the target audience of this process will be all users of the mobile platforms under analysis, since one of the main objectives of this thesis is to raise awareness about the risks in these environments.

- Due to the collection of evidence, it is useful to classify the degree of damage and action that a given attack has taken, also allowing the identification of the source that

caused the incident, whether a person or group of them (criminal activity), or a botnet or other type of digital threat.

-To extract information from a given device, in order to observe the method of storage with which it has been stored in the system under study, as well as whether the most appropriate encryption mechanisms have been applied to preserve the privacy of users of that mobile platform.

-Identify the set of errors, problems, and other vulnerabilities present in a system, as well as categorizing them in order to determine their dangerousness, and thus be able to deploy the possible methods of action, in order to mitigate them.

-Determine whether the developer of a given mobile application has implemented appropriate security mechanisms to preserve the security and privacy of its users, as well as whether it correctly uses the mechanisms provided by a given operating system, to ensure and facilitate that application developers can implement these features in their development.

-To locate the source or origin of an incident is a person, or software issue in order to elaborate a strategy that allows to prevent future problems originated by this cause, besides that in the case that it is a vulnerability of software type, to clarify its origin, and to help to the investigation and implementation of a patch that corrects it.

Forensic analysis in information systems inherits the main steps with which it must be deployed from conventional forensic analysis, so the fundamental characteristics [82] that identify and determine any analysis of this type will be:

-Repeatability: this is the ability to recreate the same results, under the same conditions, by another researcher or group of analysts, provided that the same methods are scrupulously followed, and the same tools used by the security analyst who has performed this procedure are used.

-Reproducibility: is the ability to obtain the same results under the same conditions. In other words, if another forensic analyst were to use other tools, and other methods for analyzing the information obtained from a given electronic device, he/she should be able to obtain results as close as possible to those highlighted in the analysis presented by a given forensic analyst.

-Integrity: the last but perhaps the most important characteristic of any forensic analysis is that it must be ensured that all those resources that have been manipulated in the process of conducting a certain investigation, have not undergone any change, over their initial state before starting the research, and their final state after presenting the results of it, remains unchanged.

Barriers on Mobile Forensic Analysis

Because one of the main fundamentals of the architecture design of today's main mobile operating systems has been to protect the **security** of its users, the extraction of information referring to personal data (credentials) as well as other types of data stored in the devices under study becomes really complicated, due to mechanisms such as encryption and the need for authentication.

In order, the main problems that a security researcher will encounter during the commissioning of a forensic analysis are as follows:

-Locking mechanism: since from the most primitive mobile devices that used multi-digit numerical codes, or graphical patterns for unlocking the system, such as the newest ones that use methods such as fingerprint or facial recognition, the main stumbling block that every mobile platform researcher must overcome when having physical access to a smartphone or tablet, is discovering how to unlock the system in order to be able to interact with it.

-Internal encryption: since both iOS (almost from its inception) and Android (from version 5.0 onwards), priority is the encryption of information stored on the internal storage of the mobile device in order to protect its content by ensuring its security, integrity and helping to protect the privacy of users of its platform. Therefore, in order to circumvent this phase, software tools must be available to help decipher the information gathered in the early stages of forensic analysis.

-Remote erase and blocking: iOS and Android, both systems provide the ability to allow the user of this device, in case of theft or loss, to be able to perform a blocking of the smartphone remotely through an application intended for this purpose, or by identifying the user to a web page developed to allow displaying this functionality when required. In addition to the resulting blocking, the user may also proceed to erase the information stored instantaneously, in order to protect their privacy, and that the set of personal data or not, saved cannot be used for malicious purposes by criminals, or other types of malicious users. For this reason, it is necessary that when performing any forensic analysis on a mobile platform, the first thing to do is to put the "Airplane Mode" on the smartphone or tablet seized, in order to avoid that through network connectivity, any of these procedures can be carried out, which would make the development of security analysis by the security researcher, almost impossible.

-Portability: since what is a benefit for users, which makes these portable devices so that they can be transported comfortably to any place, makes it a serious inconvenience for any analyst, because to ensure this fundamental feature coupled with any mobile device, there is a battery.

Because one of the main resources inspected during a security analysis is the memory of the device, and because it is volatile, it must be avoided that under no circumstances will the mobile device shut down or run out of battery power, since the data stored in memory would be irreparably lost. To this end, all safety analysts should carry a charger or external batteries to prevent the devices to be analyzed from becoming inactive.

4.8. Procedure to perform an Analysis

When carrying out a security investigation, it is essential for the investigator to make an adequate structure of the steps into which he or she will divide it, in order to have the maximum guarantee of success. To do this, the strategies to be followed must be classified according to the type of analysis, whether dynamic or static, that he/she wish to undertake. If the type of analysis is static, the analyst should follow the following procedures in order to perform a proper analysis:

- If the accessible source code of the application is available, it will be necessary to use a development IDE (in the case of Android: Android Studio, and for iOS: Xcode),

to display the available code so that it can be treated, evaluated and studied comfortably by the security analyst. This first procedure corresponds to a white box strategy, since we have the source code of the application, which is one of the most important assets to evaluate any type of software.

-If the source code is not available, the necessary "Reverse Engineering" methods must be used, in order to extract the source code from the application and analyze it. Once extracted, this code will be imported as a development project, imitating what was described in the previous step. In this case, since we don't know the source code and have to extract it using another type of methodology, we could include this procedure within the black box techniques.

-Finally, to see the structure of the app once installed, it should be installed on a mobile device, although it would not be executed since it is not a primary objective of this type of strategy, to evaluate the performance of the application in execution.

4.8.1. Obtaining the Source Code

Even though this set of stages is well differentiated, when starting the analysis, a researcher usually encounters a rather well-defined problem, and that is both in iOS and Android, their apps are usually encrypted. In order to solve this problem, the security analyst will have to unpack and decrypt these applications, which in the case of Android involves decoding the java classes from its encrypted and signed dex files [104], and then taking into account the particularities of the programming language, through a decompilation process to obtain the set of classes that make up the app under study.

For iOS, the process is often much more complicated, as their apps are usually compiled in languages such as Objective-C and Swift [105], and then encrypted using a unique key associated with each device. Most of the time this is usually an insurmountable hurdle when it comes to conducting an investigation, so unless Jailbreak is available on the mobile device, the steps that the analyst can take from this moment on the Apple platform, are very limited.

a) Android Procedure

There are many tools, which allow the mobile security researcher to perform the process of extracting source code from an app in Android, but the best known are:

-dex2jar: that allows to convert the app signed for the Android device into a jar container that includes the set of java classes that compose the application under study. To do that, it is only necessary to run this command: `sh d2j-dex2jar.sh classes.dex`

Once you get the java classes, you can use:

-JD-GUI: which is a tool that has a graphical interface that allows, comfortably decompile the set of java classes, to see an approximation as real as possible to the source code that has been programmed during its development.

b) iOS Procedure

As explained in previous sections, once the applications are downloaded from the AppStore (official store), they are encrypted. In order to access this source code, the

researcher must find a method to extract this app from the mobile device once it has been installed, which requires Jailbreak.

One of the main features that allows the Jailbreak in Apple devices, is the installation of an alternative application repository to the AppStore, which contains all those apps that for their purpose, or features are not available in the official store, but that in the case of a mobile security analysis, are strictly necessary to carry out it with guarantees.

This store is called: Cydia [106], and once installed it gives the mobile user access to a set of applications of different types (sometimes it can include malware), through a simple installation.

The first package to be installed is called: OpenSSH [107]. This software allows you to install an SSH (Secure Shell) server on your Apple device, so that once your credentials are entered over the Internet, they can be accessed remotely from another device that acts as an SSH client. All these procedures maintain encrypted communications, so that their security (provided by this protocol), is guaranteed.

Once installed this useful tool, it is highly recommended to give access to a repository called: BigBoss (<http://apt.thebigboss.org/repofiles/cydia/>), which includes useful command-line tools for any hacker or security researcher, and that will greatly facilitate interaction with the Apple device in future stages of research.

Following these simple steps, the researcher needs to discover the IP address through which the iOS mobile device connects to the Internet. To do this, go to: Settings>WiFi and look at the IP address specified there. Once this address has been determined, the machine with an SSH client will proceed to identify itself on the SSH server installed on the iOS device. The default authentication credentials (which we recommend changing) are:

```
-username: root  
-password: alpine
```

After accessing the device remotely, we will have access to the file structure itself, so we can install and all thanks to having it with jailbreak, tools such as: Clutch, whose github repository is the following:

<https://github.com/KJCracks/Clutch>

This software, allows from command line, to list the set of applications installed in the mobile device [108] (Clutch -i), which shows a menu ordered of the apps of the iOS device to analyze, and once decided that app is to be decrypted you must execute the command (Clutch -d number_of_selection), which will carry out the decryption process, dumping the content of the application. This tool is a great help for any security researcher, since we should remember that apps in iOS are not saved with the name of the application itself, but are installed in a directory whose name is a random identifier obtained from the key of each mobile device, so that their identification becomes really complicated, if not for tools of this type that improve the experience of a mobile security researcher.

As we have described, when apps are compiled and encrypted, after deciphering the app in the previous step, and by the structure of the programming languages: Objective-C and Swift, many times and in order to check how the mobile application has been developed, we must use techniques of "Reverse Engineering" [109], which will help us to know the structure of the source code used during the compilation of the software to be analyzed.

One of the most popular tools used by reverse engineers on Mac platforms is the disassembler Hopper shown in Figure 4.10 [110]. Hopper is a shareware tool, which allows a limited number of tests, but for the topic of analyzing an application, which are usually relatively small pieces of software compared to their desktop counterparts, it will be enough to proceed to study the source code of an iOS application.

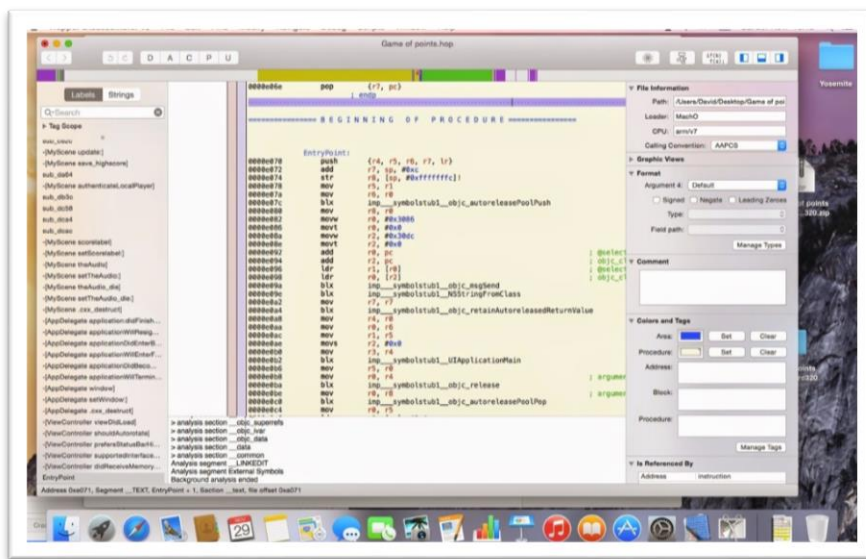


Figure 4.10. Hopper Disassembler used in iOS analysis.

4.8.2. Repackaging

Another of the processes most performed by any security analyst, once it has the source code of the applications to be analyzed, is what is called the: repackaging of the application, which has often been modified by the researcher himself to evaluate some theoretical issues during the analysis of a certain mobile application.

The reasons that can lead a security analyst to want to modify the code of an application, are usually very different, but among the most common include: exploitation of some security vulnerability of the application itself, which with the right knowledge could lead to obtaining personal information from the user of a mobile device, add other libraries software that expand or add some functionalities presents in the mobile app, remove all anti-piracy methods, anti-copy, and DRM, or even develop some mechanism that allows them to log and monitor the network communications carried out by that software once it is installed on a mobile device.

After performing the repacking process, it is important to remember for both iOS and Android that it will be necessary to sign the app with the certificate provided by Apple and Google respectively to install on mobile devices under study. In turn, and due to what

has been described in previous sections, it will also be necessary to encrypt the application, since it is worth mentioning that once installed, the content of the apps remains encrypted in the mobile device for security reasons.

a) Android Procedure

In order to carry out the repacking procedure on the Android platform, security researchers often use the tool called: apktool. This software application allows whenever the following command is executed [111]: `apktool b folder_name`, to repack the contents of this directory in an application that can be used on any Android device.

But before you can include it, and install it on a smartphone or tablet powered by Operating System: Android, you must sign the application obtained, using another tool supplied again by the Google Android SDK. This software tool is called: jarsigner, in addition to a set of public and private keys that the developer will specify when performing the signing process. The command required to sign an application is this:

```
jarsigner -verbose -sigalg SHA1withRSA digestalg SHA1 -keystore  
keystore_path app_name.apk app_name.apk (Lately, SHA1 hash function has been  
compromised, and researches has found collisions [112], therefore developers are  
switching this hash function for other stronger like: SHA256).
```

After executing this command, the user will be asked to enter the passphrase that unlocks his or her keystore, and if it does correctly, the application's signature process will have been successfully completed. Although many people prefer to use this method from command line, since the appearance of the IDE for developing apps for Android (Android Studio), there is a toolbar access to everything which process it, in a comfortable and fast way.

Optionally, and in order to improve the performance of the repacked app and thus optimize its RAM memory consumption in the mobile device, the researcher can choose to align the arrangement of the data in memory. To do this, the tool must be invoked: `zipalign` [113], as follows:

```
zipalign -v 4 original_app.apk aligned_app.apk (v indicates the number of bytes  
in which the information will be aligned. Default: 4 bytes).
```

b) iOS Procedure

As with the procedure for obtaining the source code, and decrypting the binary, the analog procedure for repacking apps for iOS is much more complicated. First, it imposes a series of new requirements, among which it emphasizes that the recipient device of these modified apps must have jailbreak applied.

In order to modify the binary, the developer and security researcher must use software such as Hopper among others, in order to change the execution flow of the application that will be installed in the jailbroken device. To top it all off, the developer must have an account that enables him/her as a developer of iOS apps.

In the latest versions of Xcode [114] (6.0 onwards), this limit has been removed, allowing any user with the Xcode IDE, and a free Apple user account, to install applications, if only intended to run on the device associated with that account, or for debugging purposes.

Despite all these limitations, if all the requirements mentioned in the two previous paragraphs are met, the **codesign** [115] command can be used to sign the modified app, created by developing apps for iOS. The syntax of this command is:

```
codesign -fs "Developer Name Account" folder_name
```

Once the iOS application is signed, and in order to install it on the system, the directory must be compressed in a zip file using any compression tool on the market that allows you to work with zip files. Once this has been done, the resulting file extension must be changed from zip to ipa, which is the extension used by applications running on Apple's mobile platform.

4.8.3. Analysis of Permission System

One of the main tasks that every mobile security analyst has when analyzing an Android and iOS application is to observe its permissions system in order to evaluate if there is any risk in the privacy and security of the system user when using the app.

For this reason, and in order to be able to observe an unnatural behavior of the app to be analyzed, a researcher must use a series of software tools that will allow him to extract and evaluate the set of permissions that an application requests during its installation and execution.

Next, we will describe the procedure followed when facing this milestone in the main Mobile Operating Systems under study, iOS and Android.

a) Android Permission System

The permissions system of the Google Operating System, has always been a controversial issue since its inception, because although through a modular development made it possible to classify the set of actions that could be carried out on a mobile device, the ability to be defined only and exclusively during the development stage of the application, turned it into a very restrictive and abusive system that was exploited for many years by developers, and malicious users, to extend the use of the system. The main reason for this problem is that when users of the Android Operating System downloaded and installed an app from the official store: Google Play, they were asked via popup that will accept a series of permissions in order of installing the app in their systems.

If the mobile user wanted to install the application in question, he had to accept all the privileges that were requested, since the rejection of them meant the impossibility of installing it, and also there was no system capable of allowing the user to disable those permissions desired, or whose use was not strictly related to the normal use of the application. All this, together with the ignorance of most users of a mobile platform, about the consequences of installing applications that are too permissive in terms of

permissions, meant that for years one of the main ways of malware entry into Android was through the installation of applications from official stores, or alternatives. Some examples of these incidents are those applications that subscribe to premium messaging services through SMS, or that have access to different system resources, such as the contact list or photo library, and share them through the app's access to network connection.

As described in previous sections, during the development of an Android application, the developer specifies what permissions an app needs to request during installation. These permissions are stored in the manifest file that is part of any Android app existing in Android.

Because of the above, it is advisable for any researcher to have access to the list of these permissions used by an app, in order to know after a thorough analysis if their use is justified, or could pose a risk to the security of the mobile platform user.

Therefore, permissions are a mechanism used by developers to provide or deny access to each of the main assets of the mobile device, but if they are not used with responsibility and knowledge, they can be the gateway for attacks, so a correct and concise specification of them is essential, so as not to cause unwanted security flaws.

One of these characteristics that every researcher must always check if an Android app is being researched is whether the Android attribute: `debuggable` [116] has the 'true' value associated with it, since many times by carelessly, sometimes deliberately, some developers leave this important parameter in the manifest file of their application, allowing an attacker to have access to some features that should generally be restricted, such as: the ability to backup information, plus in the case of a security analyst make your life easier, as it will give you the ability to perform a dynamic analysis of that environment.

In addition, there are certain system APIs that during the development of an application are associated with some of the classes or interfaces that make up the same, so in order not to allow these capabilities and behavior are accessible from other applications, taking advantage of the communication between processes, all developers must ensure that in the manifested file, `export=true` parameter is enabled.

Although permissions are a very important feature in applications on any mobile system, and one of the fundamental pillars of Google's security infrastructure, Android does not provide a way to query permissions or see what they imply. For this reason, there are some applications that allow to overcome this stumbling block, making a dump of the same ones, and allowing its later analysis. One of these tools will be: `manifest_interpreter`, a program developed in Go that will be presented in later sections, and that has been developed as one of the jobs to present in this Master Thesis, but in addition to it there are others that have been pioneers in security analysis in the Android platform, among which stand out: `Androguard` [117]. Specifically, `Androguard` has a script developed in Python called: `androlyze` [118] that makes it easier to analyze security on this mobile platform.

To download `Androguard`, we can go to its repository in github: <https://github.com/androguard/androguard> and from there clone it in our system.

In order to invoke it, it is necessary to launch the command: `androlyze -s`. Once it has been executed, a text console type interface will be available, from which it will be possible to load a specific app downloaded on the researcher's computer, as well as to access information of different types.

An example of how this tool works is as follows:

`app = apk.APK("android_app_path")` -> Declare a variable in Python with the content of the Android application loaded in memory.

`app.get_permissions()` -> If this method is called, the set of permissions declared in the application to be analyzed will be returned.

`dalvik = DalvikVMFormat(app.get_dex())` -> It will allow us to get the system API calls with which this application was developed.

`dex_analysis = VMAnalysis(d)` -> In order to analyze the dex file (the signed Android application), we will invoke the following action. In this way, we will see an association between the required permission and the interface or class that invokes it.

b) iOS Permission System

In the case of permission analysis, and some unrestricted access resources from Apple's Operating System apps, the tools available to a researcher are much more limited. In general, all available solutions need to have a jailbroken mobile device, and besides that, access to dispositives that have the latest versions of the iOS system installed is limited or not supported. That is why, in this case, if the analyst in mobile security has a device with jailbreak, and a version lower than iOS 10, you will be able to check the permissions, and even the calls that the system makes to certain APIs of restricted access such as: file access, application of cryptographic methods, Internet access among others, through a tool called Instropy [119], whose github repository is as follows:

<https://github.com/iSECPartners/Introspsy-Analyzer>

These calls to sensitive APIs by the system, are called within Apple's system nomenclature as: hooks [120], and correspond to a kind of software link between the app, and an external resource to access media out of reach of the sandbox established in each application.

In the case that the researcher does not have jailbreak (which if it is the latest versions of iOS is the most common), their path to discover these important assets in the privacy and security of users is severely complicated, and should analyze the apps through a disassembler such as Hopper, and try to search within their disassembled code for strings that refer to some sensitive resource accessed by the app.

Some of the most important strings that should be searched whenever an analysis is performed using "Reverse Engineering" techniques are: CommonCryptoHooks (look at the encryption process), CommonDigestHooks (creation of the digest associated with the cryptographic method), CommonHMACHooks (creation of the Hash Message Authentication Code (HMAC) [121] which is used in some cryptographic operations), CommonKeyDerivationHooks, DelegateProxies, KeychainHooks (associates to Keychain, the password store for iOS) and SecurityHooks.

With a detailed search of these system asset names, the security researcher will be able to discern whether the analysis app asks for access to certain parameters that are sensitive, or that their use depending on the purpose of the application, can endanger the security and privacy of the user of that mobile application.

4.8.4. Evaluating Network Connectivity

Due to the possibilities offered to users by the new smartphones and tablets, and their ability to act as small pocket PCs always connected to the Internet, it is necessary that the next step when performing a security analysis on mobile platforms, is to observe this type of network connectivity which is undoubtedly one of the most important points that is carried out in this type of research, since most of today's threats, such as the following, are the most common threats come the moment a computer or device connects to the Internet, and either starts downloading different malicious software aimed at infringing the recipient's computer, or suffers attacks by malicious users who use the ease and anonymity provided by this network to carry out attacks with impunity.

An analysis of network connections can be obtained by running a static analysis, but due to the characteristics of the same, and that the application is therefore not running, should not draw hasty conclusions about its results, since the variability, and all the jumps that can occur in a connection to the network, makes them extremely variable, and unpredictable under this scenario. For this reason, the performance of a study supervising the execution of the mobile application under study (dynamic analysis) is essential if a certain analyst in mobile security, wants to obtain results, as faithful as possible to reality.

The main objectives to be pursued when reaching this point of research are as follows.

- Assets what information is transmitted over the Internet by that mobile app, and with what implications. Is it necessary to send this information in order to carry out the activities for which the application is designed?
- How this data is transmitted. Are the security mechanisms provided by the Operative System used, such as encryption and obfuscation of sensitive user information, or is this information sent in plain format and accessible to everyone?
- This type of Internet communications, are they the result of some action of the user, and are they controlled by him, or is it a strange behavior and hidden from the user of this mobile application, which aims to filter data to the outside?
- Are incoming and outgoing communications made with the privacy and security guarantees that are required for this scenario?
- Does the app connect to the cloud or any associated cloud service? If so, with what purpose? Send user data without their consent?
- Although we are generally familiar with the fact that Internet connections are made through the HTTP/HTTPS [122] protocol, are there other types of protocols present in the mobile application's communication with the outside world? If so, what impact do they have on the security and privacy of the system user?

As you can see, network connections are one of the most important points in a security analysis, since they are not only the fundamental source of the greatest number of threats, but they are also the main mechanisms by which various malicious applications

manage to circumvent the security measures of the system, in order to compromise the security of the user of the main platforms of the mobile panorama.

a) Android Procedure

In order to analyze the network connections in the Android ecosystem, it can be done in several ways as described throughout this section, although the most common ways are through a sniffer [123] that takes over network communications made by a particular application when it connects to the outside. Even so, this procedure should be done with a dynamic analysis, and in order to follow the logical order, a researcher will always have to consume all the options available, first from a static point of view, and then from a dynamic one with the execution of the mobile application itself.

From the standpoint of a static analysis, there are several options but in one of the best known and most effective, it is through the tool: Androguard, which was presented in previous chapters of this project.

Below is a normal analysis performed through this powerful Android application analysis software [124].

-app, d, d, dex = AnalyzeAPK("app_name. apk") -> this command analyzes the Android mobile application, loading its content, and extracting the dex file for later analysis.

- strings = dex.tainted_variables. get_strings () -> next and because it is a static analysis on the source code of the application, we proceed to extract its strings, in order to check if there is any text string, which refers to network connections with the outside.

-for value in strings:

 if "http" in value[0]. get_info():

 print value[0]. get_info()

 print value[0]. show_paths(d) -> This command searches the whole

set of strings found inside the mobile app, and shows those that have some kind of reference to the use of the HTTP protocol, to make network connections.

Another more advanced way to make this query that returns much more information, is once you have done in step 1 in which you get the analysis of the Android application. In order to run this command, the researcher can proceed as follows:

-showPaths (d, dex. tainted_packages. search_methods (".","getInputStream",".")

In this way, we will get a list of the following values (for incoming connections) with this format:

[Class that executes the call to a network connection, App method that executes the call, and Name of the particular method]

If you want to obtain a similar result, for outgoing connections, you should execute the showPaths method with the getOutputStream parameter in the following way:

-showPaths (d, dex. tainted_packages. search_methods (".","getOutputStream".")

With the results obtained by this analysis, every researcher will be able to get an initial idea of how a determined mobile application communicates with the outside world, and thus have a first approach when consulting the methods involved in this process, of what is the purpose of this type of Internet connections.

b) iOS Procedure

In the case of iOS, the mobile security analyst will also have to exhaust all the possibilities offered by the static analysis of a given app, but in this case with the full set of limitations and complications involved in working on the Apple platform when doing this type of analysis, since previously it will be necessary to extract and decrypt the application, in order to disassemble it, and therefore to be able to investigate those chains and methods that make reference in previous sections.

It should be noted that in the case of the iOS ecosystem, the names of the interfaces that refer to this type of connections are three: `NSURL` [125], `NSURLConnection`, and `NSURLRequest`, which serve to declare a certain URL (Uniform Resource Locator), establish the connection, and then make a request, respectively.

In addition to the above mentioned, it is also necessary to highlight that there are other types of names, referring to calls to network connections that need to be searched when analyzing the source code of a given application, and that are:

- `NSHTTPCookieHooks`
- `NSURLConnectionHooks`
- `NSURLConnectionDelegateProx`
- `NSURLCredentialHooks`

Which correspond to how the connection is made via HTTP, and what security parameters are set for that connection. In addition, due to the development structure in iOS, it would also be necessary to consult certain methods, which are very likely to be used whenever the app makes use of network resources, and which ones are:

- `sendSynchronousRequest: returningResponse:error:`
- `initWithRequest:delegate:`
- `initWithRequest:delegate:startImmediately:`
- `continueWithoutCredentialForAuthenticationChallenge:`

All this set of text strings, should be searched by a researcher once they load the application into the disassembler Hopper, thanks to its integrated strings and labels finder.

Currently, and due to the proliferation of HTTP secure (HTTPS) protocol, we should also consult those strings that refer to this type of connections, or those elements that have some kind of relationship with them, such as SSL certificates, TLS protocol, SSL Pinning between certificates and CAs (Certificate Authorities) among others.

4.8.5. Additional Elements

Thanks to the evaluation of the strings [126] carried out in a static analysis of mobile applications, you can obtain really interesting information when it comes to

categorizing the security and privacy of a mobile app, regardless of the platform to which it belongs, so using everything described in the previous section, this section will relate some of the possible text strings that can be searched to access other resources included in the system, which contain interesting information.

a) Android Procedure

Due to the ability of some apps to store their information either in a personal database enabled for that application, or in the SharedPreferences [127] area of shared preferences, some of the strings that should be searched using the tool quoted in previous sections of this project are:

- username, user_id, login_name, login, among others, in order to know the name with which a certain user is identified through this application. Also, don't forget to determine the origin
- password, pass,token, auth_token, and others, which are strings with the password of certain user account stored within an application, besides every analyst should search information according the origin of the app, because the developer may name the option, using names in his/her native language to categorize system elements.

Moreover, all hints applied with the username should be extendible to this and other features.

- Calls made to the shared storage space (getSharedPreferences [128]).
- Credentials, cred, or other types of strings, to try to search as its name indicates the user credentials, grouped somewhere in the application.
- db, database, as well as its extensions db, sqlite, sqlite3, etc... In order to control the accesses to databases created by the application, to store its important information.
- putExtra method, which has to do with information that is passed to the Intent class when making any call.
- Comprehensive WebViews search, which are web containers that are generated within a mobile app when you want to display a certain web resource hosted on the Internet.

It is also highly recommended, in order to make a competent mobile security analysis, to take into account that this set of data may, due to malpractice, have been hardcoded in the source code of the mobile application, often due to a neglect of the developer when deleting the help comments with which he/she has populated the code, or due to an insufficient encryption mechanism that has made this information accessible to any user.

b) iOS Procedure

The method of searching for text strings in Apple's Mobile Operating System is very similar to that specified in Android, since both systems, after all, follow a similar philosophy when storing their data within the system, and furthermore, the assets that can affect security and privacy of the user of this platform are the same as those that would affect an Android user, so only the particularities or new resources will be cited.

Coupled with the search for credentials, which is one of the first and most crucial steps in this research to extract additional information to enrich the security analysis performed, will be added:

-Search for calls to NSUserDefaults [129], which is a store of iOS options, equivalent to Android's SharedPreferences.

-Localization of calls to NSLog, which is in charge of saving logs of the application, which are often used by the developer to debug the application during its development process. The problem is when the programmer, (often inadvertently) save in these logs information regarding user credentials, which due to the nature of the system logs that are stored in plain text, making this sensitive information accessible to any type of user who extracts such files.

Because iOS, the use of the SQLite3 [130] database is very widespread, together with the query of the following style strings: db, sqlite3 typical strings that should be done to find this type of information, you cannot let pass the search for SQL sentences that have been hardcoded in the system, not only to know the structure of the database itself, but also to search for some kind of security problem when it comes to making the database, or also during making a request using insecure method which allow a malicious user to tamper the sentence, because the developer did not use precompiled request by the time he/she programmed the mobile application.

4.8.6. Extracting data on Mobile Platforms

Next, there will be a step-by-step explanation of how the information stored on a mobile device would be extracted, in order to analyze the data obtained for checking the security and privacy of the system, as well as to illustrate some of the possible methods that could be used by malicious users, who decide to violate the system in order to invade the privacy of the victim. This chapter of the project will be divided according to the procedure is oriented to the Google Operating System, or Apple's, as the process in each differs, and the prerequisites for developing such acquisition of information are more limited in one than the other.

a) Android Procedure

First of all, it is necessary to point out that a rooted mobile device is required in order to be able to carry out this data extraction mechanism, in addition to installing the SDK associated with the Android version installed on the smartphone or tablet. If these conditions are met then the security analyst must connect the device via USB to the computer in charge of carrying out this investigation, which after a short period of time will make it be identified by the system, and can proceed properly at the beginning of the procedure described in this section [131].

First, the security researcher will need to open a command terminal, and type:

-adb shell (which will start as its own name indicates a shell on the mobile device, which will allow us to manipulate it as if we were inside the system).

The next step is to check the file system block size, which is one of the crucial steps when executing the command-line utility `dd`. To do this, the security researcher will invoke the following command:

`-df /data` (which will show us information on the different partitions of the system, with a series of parameters associated with them, including the block size applied to each particular partition).

Once we have completed these steps, we will need to have a sufficient amount of storage to be able to host the information extracted from the system. In the case of Android devices, it is easy to solve this problem, since you can add an external memory such as: SD card to expand its capacity, so we will be ready to locate, in which partition is mounted the SD card, to specify as dump path obtained at this stage.

To be able to discern everything that is mounted in the system, you can do this command:

`-mount | grep sdcard` (in the first place we list all the mounted devices, and then with `grep` we filter only the partition associated to the SD card).

After the discovery of this important route, we will proceed to extract the information stored in the system, as described in previous chapters of this thesis:

`-dd if=/dev/block/mmcblk0 of=/storage/sdcard1/dump.img bs=4096` (as important parameters, we can observe that in the case of `of=` we specify the path for a name image: `dump.img` in the SD card installed in the system, and also that the block size is: 4096, which is a typical value for the Google Operating System).

After a few minutes of waiting, until the backup process is successfully completed, finally, this created image can be exported to the analyst's computer, by means of the command:

`-adb pull /storage/sdcard1/dump.img`

In addition to the procedure, to call it in some way: official for backing up the system, in Google Play, the official store for Android, there are many applications such as: BusyBox [132], which allow this procedure to be performed automatically, and more comfortable for the common user of this platform. These apps also have the peculiarity that, in addition to allowing the creation of a copy, they allow you to select what information will be exported in the copy, and which will be discarded, which could also be an alternative for the advanced user, who does not want to use the command terminal to work.

Another noteworthy point, although it corresponds to an advanced procedure due to its difficulty at the time of its execution, but due to the multiple scenarios that can be found in a mobile platform it is necessary to take into account, is that the researcher does not have a device with an SD card installed, or that it does not have enough space to first store the backup obtained in the mobile device to be analyzed. In this case, Android provides a solution that by means of sockets allows to send all the information to the

computer of the security analyst, without needing to save all the procedure in the smartphone or tablet investigated. To carry out this complex method, we must:

Creating a socket between the mobile device and the researcher's computer using:

```
-adb forward tcp:6666 tcp:6666 (This means that all traffic passing through port 6666 on the device will be sent through port 6666 to the analyst's computer in security).
```

Then, we run the image generation from the internal storage:

```
-dd if=/dev/block/android_storage | nc -l -p 6666 (using the netcat tool we create a socket that listens to port 6666 specified above)
```

As described when creating the bridge using adb, this information will end up in port 6666 of the analyst's computer in security, so only one terminal must be opened on the computer and type:

```
-nc localhost 6666 > external_image.img (this will create a socket that will collect the information communicated in the previous step by the socket generated on the Android mobile device, and will dump the content of that communication to an image called: external_image.img)
```

Because Google Android is an open-source Operating System, the options that exist to work with it are much greater than in iOS, and this is demonstrated by the multitude of alternatives to perform forensic analysis on this platform. One of the best known is: AFLogical [133], which can be shameless from its github repository: <https://github.com/nowsecure/android-forensics>

Once compiled the app with the typical apk format, due to the possibilities offered by adb, and as long as we have activated the "unknown origins", we can install by means of this sentence the AFLogical:

```
-adb install AFLogical.apk
```

After its installation, we will be able to run it from the Android mobile device as any conventional app installed via Google Play, and from an austere graphical interface, it will allow us to create a customized backup, selecting (as it could be done with: BusyBox), what information: phone contacts, SMS/MMS, and others we want to export from the device, and which ones we do not.

As the only difference, it is necessary to emphasize that with AFLogical the information will not be obtained in image format, but in separate csv files with the classified information according to its nature, which is a great advantage for its later analysis by the security researcher.

b) iOS Procedure

The process to be carried out in the Apple Operating System, is quite similar to its counterpart in Android, but is much more limited due to the features imposed on the

platform itself: it is much more closed than Google's system, and is also encrypted by default.

The main requirement that must be fulfilled in order to be able to backup the information stored in the iOS device, is that jailbreak has been performed on it, and that tools such as OpenSSH and dd have been installed on the system, which will allow this procedure to be performed correctly.

In addition, and because Apple mobile devices do not have the ability to add external storage such as Android's, there should be enough space on the internal storage to contain the backup that will be created at the end of this process.

If all of the above is true, simply connect via SSH to your smartphone or tablet, as described in previous chapters of this project, and then type this command:

```
-dd if=/dev/rdisk0 of=ios_image.img bs=1M (whose syntax is identical to that of Android, and only differs in block size, which in the case of iOS is usually 1MB).
```

Although this procedure is feasible, the latest versions of the system due to its high encryption of the information stored in mobile devices that mount iOS, makes it advisable to use other types of tools, such as: iTunes which will be able to perform this process in an advanced way.

4.8.7. Extraction of information stored in RAM memory

As described between the fundamentals and rules to be followed in any forensic analysis, one of the most important points to be taken into account is RAM memory, as it loads all the applications running on the system, and can therefore contain information of enormous usefulness for a privacy and security analysis, as is the case of this project.

a) Android Procedure

In the case of dumping the information contained in the RAM memory of an Android mobile device, you can use the tool called: LIME [134] (Linux Memory Extractor). To proceed to download it, we must once again go to its github repository and clone it: <https://github.com/504ensicsLabs/LiME>

After compiling the kernel module to be added to the kernel of the mobile system, the security researcher should upload using the tool adb:

```
-adb push lime.ko /storage/sdcard1/lime.ko (which will store it on the SD card of the Android device).
```

Once this process is completed, a socket must be created to redirect all information sent to port 4444 of the mobile device, so that it is directed to port 3333 of the security analyst:

```
-adb forward tcp:4444 tcp:3333
```

After doing this, a shell will open on the Android device using: `adb shell`, and then do the following:

-su (we will identify ourselves as root. From this step, it is understood that in order to be able to perform the dump of RAM memory of the mobile system, we need a rooted mobile phone)

-insmod /storage/sdcard1/lime.ko "path=/storage/sdcard1/ram.lime format=lime" (the previous command indicates where to dump the memory dump information into the SD card, in addition to its format).

Then, and as it has already been done in previous sections of this project, the file obtained on the SD card will be downloaded to the security researcher computer. To do this:

-adb pull /storage/sdcard1/ram.lime

Finally, and although performing a complex forensic analysis of the Android system's volatile memory is not one of the main objectives of this thesis, it is worth mentioning that this RAM dump image can be analyzed by other advanced forensic tools such as: volatility [135], which is installed by default in: "Security Analysis Workshop".

b) iOS Procedure

The analysis of the RAM memory of an app in iOS is much more limited than the possibilities offered by the Android Operating System, since the main recommendation in this regard, is a tool that is included in the "Developer Tools" of macOS, which is called: "Instruments". Nevertheless, the tool is really powerful and allows to monitor: Overall memory used by a certain app, as well as leaked memory produced by memory space not dellocated, and even abandoned memory or zombies (code referenced, although it is no longer necessary). Removing these problems corresponds to a developer's own work, since although iOS has an ARC (Automatic Reference Counting), automating the allocation and dellocation process, sometimes it is necessary to make this procedure programmatically.

4.8.8. Analysis over Sensitive Information

Once a complete image is extracted from the information stored on a mobile device, it is important for the security analyst to have the necessary knowledge about what type of information is usually stored in these devices, as well as where to find it.

Due to the nature of the applications available in both: iOS and Android, we can highlight this set of main sources of information:

- Shared storage space (SharedPreferences on Android, NSUserDefaults on iOS).
- SQLite3 databases that store the information managed by each of the mobile applications installed in the system.
- All types of plain text files that may not be taped in the system, and therefore being visible if they include relevant information would be a risk to the security and privacy of the user).
- Binary files.

- Passwords saved in the system.
- Credit cards numbers, or any type of information identifying the mobile system user.
- Web browsing history.
- Other possible information.

Due to the large number of aspects to be taken into account in this analysis, and the diversity of the information stored, the following chapters will specify the main routes where to look for the information cited, in order to help the research work carried out on these mobile platforms.

a) Android Procedure

As described in previous chapters, Android's file system is distributed in a set of partitions, among which the following can be highlighted:

- bootloader: It is a read-only partition that includes the code that is executed for the first time, when the Android device is booted. It is in charge of loading the operating system kernel.
- boot: where the Android kernel is included, and other files needed for booting.
- userdata: includes all the information stored by the user, including: applications, photos, and other multimedia content.
- system: includes the main system libraries, and basically the Android Operating System itself.
- cache: contains temporary information stored by the applications installed on the mobile device, in addition to storing temporary information generated by the Dalvik virtual machine that interprets the code of the installed apps.

In addition to these directories that define the structure of Android at a global level, it is also necessary to mention:

- system/app: which is where the applications that come installed by default in the system (generally Google apps), perform their installation. For security reasons, as with the bootloader partition, this is read-only, which prevents users from uninstalling this set of apps that come by default installed on the system, unless they format and install an alternative image or ROM such as LineageOS [136], which is not one of the purposes of this project.
- data/data: The data that is created by the applications, and includes all types of information entered by the user, or created during the process of installing the applications themselves. In this folder, you can also find the sandboxes of the different apps, which as specified in previous chapters of this thesis, is a mechanism that allows you to isolate the degree of interaction between the applications installed in the system, with the resources external to them.

Within each of the sandboxes created for an app, it is necessary to highlight the following structure:

- files: in this directory, the app creates the files that are necessary for the execution of the app. It usually stores some multimedia resources such as images.
- lib: it hosts links to the software libraries that are required to run the app itself.

- shared_prefs: includes the SharedPreferences files created by the app.
- databases: as its own name indicates, it stores SQLite3 databases created during the process of installing and running a given mobile application.
- cache: temporary information stored in the app.
- /data/app: is where the apk files are stored, so to speak the compressed container that allows to install each app in Android, and which contain the source code of the app compiled in classes.

Apart from all the folders mentioned above, also and due to the multitude of information they may contain, we must mention where the photographs, videos, and other multimedia resources created by the user of the system are stored.

The chosen directory, although it usually varies depending on the manufacturer, has the following structure:

-/storage/[android/sdcard]/DCIM (whereas can be seen it can be either a system path, or an SD card path. It is necessary to emphasize that in this file you can also store the photos saved by some of the main messaging systems such as: WhatsApp, Telegram, and also some of the most used social networks: Twitter, and Facebook.

Other interesting folders are those where the unlock pattern is stored, widely used in Android devices, until the appearance of fingerprint recognition. The path that this pattern stores is:

The information contained in this file is a numeric value that identifies each point of the graphic pattern with an associated number, starting from upper-left dot which takes the value of: 0.

This combination is stored in the file by means of the hash: SHA1 function to protect the information stored in the file. Because of how Google discovered in 2017, this hash function has collisions, it is easy to identify by brute force which graphic pattern is stored in this file.

In the event that a numerical pattern is used for unlocking the Android mobile device, the information of this key will be stored in:

-/data/system/locksettings. db (the contents of this file is the result of performing an MD5 hash on the unlocking code, concatenated with the associated jump when that function has been executed).

To consult the information stored in this database, we must execute this SQL statement, from the SQL Manager included in the "Security Analysis Workshop":

```
SELECT value FROM lock settings WHERE name= "lock screen. password_salt"
```

Once this data is obtained through the script also included in the workshop and called:"android_password_cracker.py", or through another more advanced tool called: hashcat, which is also installed by default in the "Security Analysis Workshop", it will be possible by brute force to obtain the unlocking code of the mobile device with Android Operating System.

If you want to use the tool: hashcat [137], which has as advantages that can use the GPU of the computer of the security analyst, to increase the calculation capacity and therefore reduce the time it takes cracking the password, you must use this statement by command-line [138]:

```
-hashcat -a 3 -m 100 file.hash -1? d? 1?1?1?1?1?1?1?1?1 (in which -a indicates the method of cracking: forced shoot, -m 110 indicates the type of hash: SHA1 pass-salt, -1 allows to create a custom search pattern which is this case is? d i. e. an integer number, and finally a space of 8 possible spaces will be specified
```

Due to the ability of these devices to be permanently connected to the Internet, another parameter that every researcher should take into account is not only the cellular connection, but also the ability to connect to WiFi networks of the mobile device.

The information associated with WiFi networks saved and identified on the smartphone or tablet is stored in the directory: /data/misc/wifi. Within it, we find a very important file called: wpa_supplicant.conf, which if you view its content, you can see information such as: the SSID associated with the latest WiFi networks accessed, as well as your password, and even the encryption associated with it: WEP (obsolete), or WPA/WPA2-PSK (currently recommended).

Another of the interesting data [139] to be taken into account by a security analyst is to observe the information stored in calendars, since many users keep a schedule of their inventions in them, and are a real source of great importance in order to know the habits of the owner of the device.

Although this information may differ from the manufacturing, it is generally stored in: /data/data/com.android.providers.calendar/databases/calendar.db

The next step would be to check the text messages (SMS/MMS) sent or received, and which have been stored on the Android smartphone. To view this information, the route may change again depending on the manufacturer, but usually:

```
-/data/data/android.providers.telephony/databases/mmssms.db.
```

With the data obtained on SMS and calendar events, the next procedure to be carried out in a study of security and privacy on a mobile platform, is to know where the user's contacts are stored, in addition to the calls you have made from your smartphone.

The location of these resources is usually located at:

```
-/data/data/android.providers.contacts/databases/contacts2.db (within this database in SQLite3 there is a table called call that includes, as its name indicates, the calls made by the user of the mobile device).
```

Due to the importance of email today, and that in addition to the fact that every time we start a device with Android installed, the user must be identified with a gmail account, the next step will be to identify where the information regarding this important account is located, which is also the main one associated with the backup process through a cloud service system.

The path that contains the system user's account is usually: /data/data/com.google.android.gm/databases/mailstore.[account_name].db

Every database obtained in procedures described in previous steps of this section, it can be checked via: SQLite Manager provided in the “Security Analysis Workshop”.

Once this information has been evaluated, the next step is to take a look at the user's web browsing history, which usually includes interesting information about the mobile device user, but if obtained by a malicious user it could lead to a violation of the privacy of the holder of such information.

By default, the web browser installed in Android is: Google Chrome, and because it is one of the most used in our days, and more in the Android ecosystem, if you access the route where the web browser is installed (/data/data/com.android.chrome), you can access information as diverse as: cookies, login data, bookmarks, browsing history, among other interesting information.

Finally, but not less important, other types of information that could seriously jeopardize the security and privacy of the Android mobile device owner, is location information stored by certain system services such as Google Maps that make intensive use of it.

If GPS (Global Positioning System) is enabled, a detailed itinerary of the smartphone user is saved in the "My Places" section of Google Maps, as well as an almost perfect identification of where your home is located. Therefore, one of the main recommendations for a lost or stolen device is to disable this option in order not to give more information than the attacker.

In addition, and in view of the information that a security researcher can consult, regarding the information extracted from the mobile device described in the previous sections, if we access the path where the Google Maps application is stored (/data/data/com.google.android.apps.maps), we can see a set of images stored in the cache directory of that app, which correspond to portions of the map that represent the last sites consulted by the user of that app.

More information about “Android security” in: [\[157\]](#).

b) iOS Procedure

Within the iOS ecosystem, information extraction is often complicated by the fact that it is a closed operating system, which also imposes an encryption mechanism on the data stored on mobile devices, which provides an extra degree of security and privacy to the users of this platform, since from iPhone 3GS onwards, Apple included a co-processor in its devices whose task, among others, is to perform crude operations. However, if the mobile device to be analyzed has jailbreak, the possibilities of carrying out a safety analysis increase exponentially, since there are fewer barriers to overcome than those of a safety analyst.

The following are the main routes within iOS for data extraction, which is one of the first steps that any security researcher must take, and due to the above paragraph, the

device on which the study is being conducted will be considered to have jailbreak performed, and it is unlocked, since if this were not the case, the information dump would really be a very difficult task to perform.

In general, in order to observe the internal infrastructure of the applications installed in the system, the first route [140] to be inspected is:

`-/private/var/mobile/Library` (which contains all the data of the applications installed by default on the mobile device. These apps are generally the apps that come with an initial iOS installation, and are created by Apple. They are common among all devices, and the user, as with Android's default applications, which have been developed by Google, does not have the possibility to uninstall them.

In case you want to consult the data of the third party's apps installed in the system, the path where all the available information of the same ones is stored:

`-/private/var/mobile/Applications` (it should be noted that due to the characteristics of iOS that have been cited in previous chapters of this project, each app is installed in a directory with a name that is randomly generated from the UDID of the mobile device (32 characters) during its manufacturing process, so that at first glance it will not be possible to discern the association between the name of the mobile application and the folder name).

After this short introduction, how iOS apps themselves and third-party apps are distributed, it should be borne in mind that the basic structure of any directory associated with an application is as follows:

- Documents: contains the set of resources created by the application, and which are necessary for its operation.
- Library: as its name indicates it maintains configuration files, and other software libraries that are necessary for the execution of the analyzed app.
- tmp: is a directory that contains temporary files created during application execution, and therefore has a limited durability, as they are usually deleted regularly.

In addition, and as with the Google Operating System, every researcher must take into account one of the most reliable possible sources of information available on mobile devices, and that leakage can seriously harm the users of this platform, and is photos, videos, and other multimedia content generated by the owner of the mobile device, which usually include information that can uniquely identify you, through the analysis of metadata EXIF associated with these.

This important multimedia directory in iOS is on the path:

`-/private/var/mobile/mobile/media/DCIM/100APPLE` (photos taken in JPG format (before iOS 11) or in HEIF (from iOS 11 onwards) are shown here. In addition, this directory will contain the screenshots taken by the user, or the set of recorded videos.

Another of the fundamental points to consider in an iOS investigation is the passwords stored in KeyChain, which is the password vault of the iOS ecosystem. This

information can be viewed locally from the mobile device itself, or from the cloud: iCloud service, which, if enabled, saves a backup associated with your user account.

Next, another of the main sources of information in this mobile platform, you can use some of the default apps installed, such as: Notes. Notes is a simple app that as its name suggests allows the user to take concise notes or reminders in order to serve as a kind of digital notebook. The information stored in it, can be of very diverse nature, from text, passwords, even photos, and location information so it is one of the fundamental points from which the security researcher could draw more information.

The path where the Notes app data is stored is:

`~/private/private/var/mobile/Library/Notes/notes.sqlite` (It is therefore a SQLite3 database that can be conveniently accessed from SQLite Manager. Even so, it is necessary to point out that since iOS 9.3 version, this database can be encrypted if the user decides to do so, so in that case if the password associated with it is not known, it would not be possible to consult it, and this procedure would be greatly complicated).

Later, another key source of information will be SMS/MMS sent or received by the user. This data, as with Android, will be stored in an SQLite3 database, which this time will be stored in this path:

`~/private/var/mobile/Library/SMS/sms.db` (In this database, the fundamental tables to consult are: "Messages" as it includes the SMS sent and received, "msn_pieces" that includes the attachments included in these messages). It is necessary to emphasize that all information stored by the Apple messaging system called: iMessage is also stored in the "Messages" table.

If you want to consult the information created during web browsing made by the user of a given mobile device, the most important routes that any researcher should consult are:

`~/private/var/mobile/Library/Safari` (corresponds to the directory where the default iOS web browser is stored. Since the platform allows installing other alternatives such as Firefox, or Chrome, in order to extract all the information from this point, the security analyst may have to search within the directories created by these other browsers, if they exist.

Within this directory, there is usually a vast amount of information, but in order to observe the habits of a user, you could consult: the database: `Bookmarks.db`, which includes the favorite pages, which have been visited by the user of the mobile device).

Continuing with the compendium of information stored today, in our mobile devices, the next stage of a conventional security analysis on this platform would be to extract information from the user's contacts, in addition to calls made if that device is a smartphone. In order to do this, the following directories should be considered:

`~/private/var/mobile/Library/AddressBook` (which contains the contact list, and contains two interesting tables: `ABPerson`: saves the name of the contact, and personal information

associated with it, and ABMultiValue: which stores phone numbers, email addresses, and other information).

If you want to consult the call history, it is stored in:

`~/private/var/Library/CallHistory/call_history.db` (In this database, not only the call list is included, but also its duration, the direction of the call (incoming or outgoing), and the telephone number and contact number of the list associated with it).

Another feature, which might be interesting to consult, is the ability to create voice memos (voicemails) within the iOS device. These voice memos may contain interesting information that could compromise the iPhone user. The path to consult these stored voice files is:

`~/private/var/mobile/mobile/Library/Vociemail/voicemail.db` (Inside this database is the location within the device of the recorded audio files).

Because of the inclusion of GPS devices on smartphones and tablets that have iOS installed, they can be a huge source of information when enabled, and their malicious use can seriously violate the privacy of the owner of such data.

This kind of information is stored inside Apple's mobile operating system in the path:

`~/private/var/root/Library/Caches/locationd/consolidated.db` (This SQLite3 database will contain geographic information about where the device user has been, as well as information about the WiFi APs (Access Points) to which he has connected).

As we have commented in previous sections of this thesis, one of the fundamental parts of any iOS application is its configuration files, or also called: plist file that can contain information of interest to be analyzed.

Some of these plist files, which should be inspected by the security analyst in charge of the investigation, are:

`-com.apple.accountsettings.plist`: contains information about the e-mail accounts configured on the mobile device.

`-com.apple.AppStore.plist`: Last user searches in the official iOS store (AppStore).

`-com.apple.facetime.plist`: information associated with the iOS FaceTime app, which allows VoIP calls and video calls between users of this mobile platform.

Finally, it is also necessary to highlight the possible information that could be obtained from the analysis of backups created in iOS, through the official tool: iTunes. Prior to iOS 10.3, the encryption algorithm used was weak, so brute force could be used to bypass the information stored in them, provided that physical access to it was available. As of iOS 10.3, the encryption algorithm changed, and currently if you have an associated password, there is no known method of discovering the data from these backups that is feasible to use.

Apart from the above, as long as there is physical access to the computer from which the copy of the mobile device has been synchronized to analysis, another series of plist files associated with iTunes could be consulted, and which could include interesting information to enrich the investigation under way.

These files would be: Status.plist (which contains information about when a certain backup has been generated in the system), Manifest.plist (which includes information from official applications, as well as third parties installed in iOS), and finally: Info.plist (which contains private information associated with the mobile device itself, such as: IMEI, serial SIM, among others).

In short, if the security analyst manages to overcome all the obstacles imposed by the platform, the information that can be obtained from the analysis of a given iOS mobile device, be it smartphone or tablet, although more limited, will be comparable to that obtained in the Google platform, thus allowing an analysis to be developed in this ecosystem, of quality, and with sufficient guarantees that the security and privacy associated with its architecture is being evaluated. For a more detailed information about Android and iOS collection of information, see: [155], and [156] respectively.

4.9. Security Mobile Analysis tools

Throughout this chapter of the project, a set of tools will be listed that will allow a security analyst on mobile platforms to perform static and dynamic analysis on mobile apps on any of the platforms under study: iOS and Android. Accompanied by a description, in addition to naming the tool, an attempt will be made to explain how to use it, listing the commands if it is necessary, which should be typed in order to perform the main actions allowed by the analysis software. This section of this thesis for the Master in Cybersecurity will also serve as a guide to the tools that will be available in the "Security Analysis Workshop" which is the distribution with tools for mobile analysis. In order to see more detailed information about Android & iOS analysis, see: [152].

4.9.1. Utilities for Android

In this part, the set of tools used for analyzing Android applications are going to be introduced one by one, in order to create a list of the different options a security analyst has by the time he/she decides to make a security analysis.

a) Quark [83] is a tool developed by LinkedIn, which automates the analyses carried out on Android applications, in APK (Android Package Kit) format. During its analysis, the tool collects the entire set of vulnerabilities, errors, or problems of any kind that could be used by malicious users or crackers, when violating the security of the system, and compromise the security of the user, in addition to being able to help for the creation of tools that exploit those specific vulnerabilities, and therefore it could serve to create an exploit in the system.

b) apktool [84] is a software which is able to decode, decypher and build a normal apk file which is the normal format for Android applications, when we are dealing with a security analysis in this platform.

c) **Androguard** [85] is a set of tools which are meant to analyze the inner structure of an Android application. To do this, Androguard provides the researcher with a great number of different scripts which help the user to perform certain research of the different elements which form an apk file. Every script in this group of security applications is developed in Python, and therefore it needs at least a Python2.7 or over for working. Python 3.5 and over is also compatible with these tools.

d) **dex2jar** [86] It is an application designed to interpret files in dex (Dalvik Executable) format, which correspond to the content of the compiled java classes and which can be later interpreted by the Dalvik virtual machine when executing any application on the system by the user.

e) **JD-GUI** [87] It is a program with a graphical interface, and that has support for some of the most famous development IDEs in the Java environment such as: Eclipse, IntelliJ, or Android Studio, and which allows by means of a GUI, to load the jar container once it has been obtained through tools such as: dex2jar to decompile the classes contained, and show the source code in .java perfectly.

The way to load these .jar containers are by using the toolbar by selecting File and then opening the .jar file. After this, the JD-GUI sidebar will display a list of the set of classes stored in the container, and if you select one by one you will see a display of your source code in the side-view of it. This highly relevant information in a static analysis can be exported, saved and modified whenever you want to change the app code in order to test, add some functionality by creating an alternative version of that mobile application, or in the case of investigating some kind of vulnerability whose use could damage the user's security.

4.9.2. Utilities for iOS

Difficulties in analyzing applications on the Apple platform make the set of tools exposed a fairly small number, most of them limited to use on iOS devices with jailbreak applied. Because most of them must be installed on the jailbroken device, or on a computer with Apple (Mac) accepted architecture, instructions for installation will be specified, although they will not be included in the "Security Analysis Workshop" included in this project, due to its impossibility of installation on this desktop architecture.

a) **Clutch** [88] as mentioned throughout this project in reference to the iOS mobile platform, once the user of the device downloads a given app to his or her mobile device and installs it, it is encrypted. Because of that, the job of security analysts, or hackers when trying to work with it to look for vulnerabilities or other types of problems, is really weighed down.

That is the main reason why in the iOS ecosystem there are tools such as: Clutch that is no more than a solution to decrypt these mobile applications, without altering the content of the device, since these dumped apps end up in a specific dump directory of the utility.

b) **cycrypt** [89] it is a tool available in the unofficial store of iOS: Cydia when the iOS device is jailbroken. In short, is a command-line tool which allows with a mix of a

Javascript-like and an Objective-C-like syntax, to inject code in the system, in order to modify the behavior of application loaded in foreground in the Operating System.

c) Hopper [90] it is a powerful disassembler only available on macOS platform which allows the researcher not only to disassemble the code, but also decompile it and debug it. Thus, it is a very useful tool not only during the development process, because its ability to decompile or disassemble make it one of the chosen utilities by the time a security researcher, in order to discover the source code of an application for performing a static analysis, he/she uses Reverse Engineering techniques to recover the code written by the developer of a certain iOS app. Despite it is one of the best reverse engineering utilities for macOS, it has one limitation: this tool is shareware. Therefore, its free version only allows a short number of analysis at the same time, and it only allows 30 minutes of research, plus it is not possible to export the result obtained, but to analyze an application is perfectly capable, so when dealing with disassembling, if the security researcher owns an Apple device, this will be the chosen selection.

d) iNalyzer [91] is a tool developed by AppSec Labs, and to install it, it is necessary a jailbroken iOS device, and also adding this repository: <http://appsec-labs.com/cydia>, to Cydia unofficial store, in order to install this tool for making a static and dynamic analysis on iOS ecosystem.

e) idevicebackup2 [92] is a multi-platform utility that allows to a security researcher to create and restore backups in iOS platform. This software is written in C programming language and it is available from the official repositories of the most famous Linux distribution. Therefore, its installation process is very simple, and it does not require to clone any repository to install it (only if you want to get the latest version of the tool).

f) rvictl [93] by the time, a security analyst need to carry out a research about network connections, it is necessary to have a tool which allow him/her to collect the network packets (generally TCP/UDP) generated when a mobile device communicate outside. In order to perform this sort of analysis, the researcher have to dealt with the creation of a virtual network interface which is very simple, using rvictl (Remote Virtual Interface controller). This tool allows the generation of this interface, passing the UDID of his/her iOS device when is invoked, and after that a fully-functional network interface is created, redirecting every packet forged in the mobile dispositive through it. Once this kind of bridge is established, the only thing is necessary it is: to deploy a sniffer which is a capturer of network packets in order to analyze the traffic produced in the device to analyze. The election for a sniffer is more varied, because there are a lot of alternatives, through command line or graphical user interface, but in general the two most used for computer research are: tcpdump [94] (command-line tool), or a more sophisticated solution called: Wireshark [95] (with GUI integrated, although it has a command-line version: tshark).

4.9.3. Tools for Network connection in both platforms

Tools described in this section can be used for both mobile platforms (iOS and Android), due to they are just utilities for collecting and capturing network packets, in order to be analyzed lately by the security analyst.

a) tcpdump: It is one of the most famous and common network command-line tools, and generally it is installed, practically in every distribution of Linux, or UNIX-like system. When a security analyst needs to deal with the analysis of network traffic, tcpdump offers a professional way of capturing packets in any network communication, and the ability to dump the results in a file for a future research. In order to use this software in the analysis of privacy and security in both: iOS and Android devices, it is only necessary to specify the interface to watch and let tcpdump running for several minutes (depending of the detail of the research), in order to collect the information needed. For a complete guide about how to use this command, visit: [96].

b) tshark: is the command-line version of Wireshark, and it allows the analyst to check and filter certain things, tcpdump do not allow so easily. In this reference, we can see a complete guide about it usage [97].

c) wireshark: It is basically the graphical interface (GUI) for tshark that delivers a comfortable and visual way of analyzing network traffic from information collected thanks to tshark. Although its operation is easy to use, it is very complicated to dominate every characteristic this tool offers. For a conventional security analysis, the researcher only needs to know how to select the interface (click on the toolbar over Capture, Interfaces and then selecting the network interface wanted). After doing this process a table with a list of packets will appear in Figure 4.11, and the researcher will have a top textbox for filtering every piece of information collected by this network tool).

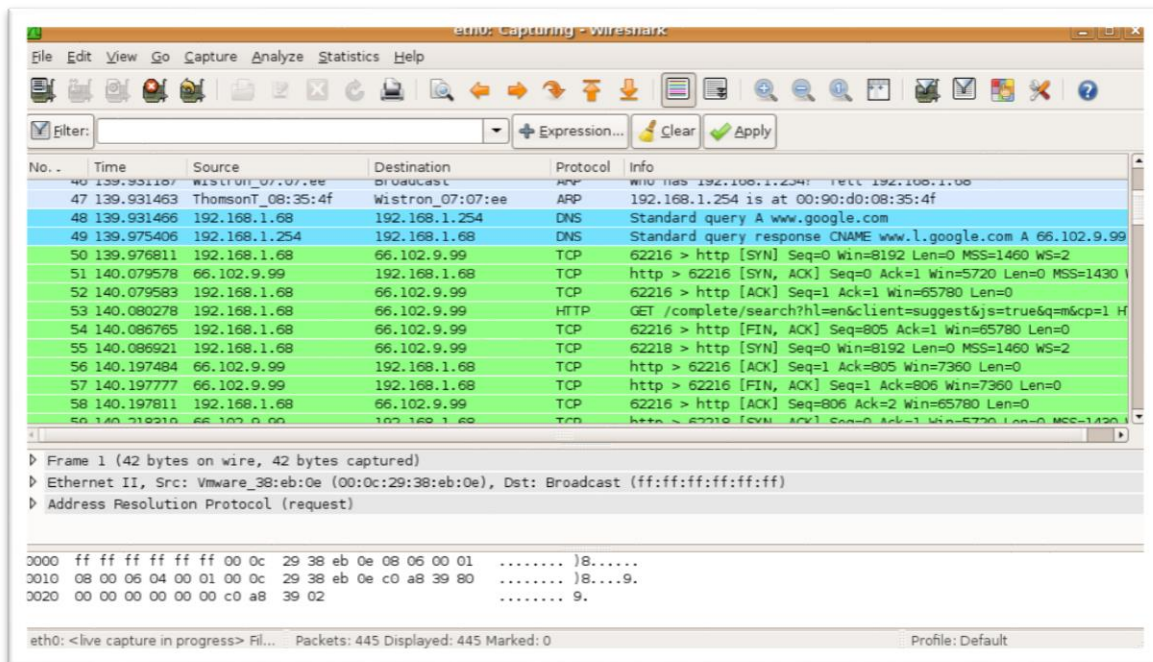


Figure 4.11. Wireshark (Source: Linux Secrets)

As in the case of sniffers, to provide a proxy for the security analyst, we will select a single option, which will work for any platform, since a proxy (see more information about what a proxy is: [98]) is neither more nor less than a kind of intermediary between a device and its access to the network, allowing that before both incoming and outgoing traffic arrives or leaves a certain device, it runs through the proxy allowing the researcher to perform an analysis. The utility that will act as the chosen proxy for its power and its enormous possibilities is:

d) Burp Suite [99] is a software for HTTP/HTTPS communications which provides to a security analyst, the ability to stop the traffic before going inside or outside a computer which acts like a proxy, in order to analyze the content of the requests and responses generated during the process of establishing a connection through the web orientated protocols previously mentioned. In order to configure it, it is necessary to create a proxy SOCKS in the localhost computer of the security researcher in the "Connection tab", and next set up this option like the outgoing method for Internet connection on iOS system. To do that in the iOS device to analyze, it is needed to go to Settings, WiFi, and finally filling up the HTTP Proxy section with the data provided in the previous step. From now on, all the Internet traffic generated in the mobile device will go through the proxy, so the security analyst will be able to check and tamper this information in order to test the mobile application.

Once, all these settings are configured, the next step is setting up the intercept options which the researcher can select what protocol to scan, and even the kind of information of matching like: certain image format, get or post requests, and so on. Finally, when iOS device starts its network traffic, the security analyst will be able to watch the incoming and outgoing traffic, modifying it would be necessary, or just checking if the information sent during any transmission, fulfills the security and privacy measures needed to protect the user experience independently of the Operating System used in all this process. In Figure 4.12, we can see the interface of the application.

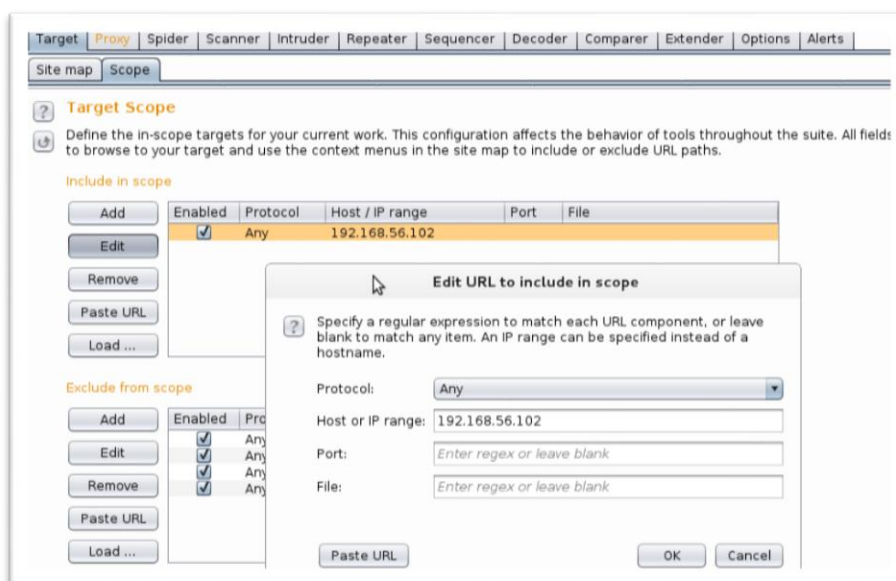


Figure 4.12. Burp Suite

4.9.4. Database Managers

These software utilities are meant to be the main tools for managing and modifying the structure of the information stored by a mobile application. Due to in mobile environment, the main database used is: SQLite 3 thanks to its features which made it the proper solution for iOS and Android. In order to manipulate this kind of databases, the tool suggested for security researchers is:

SQLite Manager [100]: SQLite3 is a fast and very light-weight (Unlike other DB, this one does not have an independent server and client part), and portable database used in practically all mobile Operating System (in both, iOS and Android is the default option) which allows the user of the mobile application and the app itself to save different kind of information, in order to provide a store to recover data when it is needed, and develop properly the activity for which the app was developed.

4.9.5. Tools for a Mobile Forensic Analysis

In order to perform a proper forensic analysis, it is needed to know a set of tools which help the security researcher to carry out this methodology successfully. Many of these software tools are properly installed in any system (above all if it is a UNIX-like Operating System), but some of them are not part of this group, so it is necessary to install them, to run the different tests to do during the forensics stage of any security and privacy analysis. Next, we are going to describe a collection of tools which will be used during the analysis of iOS and Android mobile devices, in order to interpret the information gathered in the extraction process.

a) dd [101] is probably one of the simplest and most common commands available in practically any Linux, or UNIX-like Operating System, but at the same time one of the more powerful, and less known by the users. This command-line utility allows the user to carry out some interesting actions [102] over the file system of a certain OS like:

- Data transfer (from one device to another). This is used for moving information from hard drive or storage to another external resource like: SD cards, CD, floppy disks, and so on, and vice versa.
- Recovering and restoring the Master Boot Record (MBR): which is the first sector of many storage solutions which contains information about active partitions in the system.
- Data modification: that carries out any update of information stores in a certain file in the system.
- Data wipe: which performs an erasing over certain storage resources like: hard drives, or in terms of mobile ecosystem: internal storage and SD Cards. This wipe operation can be carried out in several ways, but the most common are: zero-based and random-based erasing.
- Data recovery: from a disk image previously created, any user can restore certain storage resource with the content of the previous image disk mentioned.
- Benchmark storing system: making read and write operations in order to check the overall performance of the internal storage in this mobile system.

b) exiftool [103] Nowadays, not many people know the amount of information present in the data stored in their mobile devices. In general, when talking about personal information, people think: user credentials, credit cards number, password of e-mails accounts, and so on, but there is a general ignorance about the metadata associated to any file saved within the mobile system. The metadata EXIF (Exchangeable Image File

format) is just pieces of data which identifies in many occasions univocally not only the resource where they have been generated, but also the user.

Therefore, it is something to be particularly careful about, because this metadata tags may contain information whose leakage could put in risk the user privacy of the mobile system. From pictures, videos to documents, all of them have metadata associated which could identify not only the device itself where these files have been created, but also the user himself/herself, therefore these assets are one of the main targets by malicious users, and also in terms of performing a security analysis of certain system, because provides a great amount of useful information, in order to evaluate a system.

5. Results: Security Analysis Workshop (SAW)

In this section of this document, we are going to develop, one of the main objectives of this master thesis: the creation of a platform with every tool cited until now, besides the addition of new utilities, created especially for this work

5.1. Proposal

In order to analyze mobile apps on the iOS and Android platforms, a “Security Analysis Workshop” has been created, which includes a set of tools that facilitate the development of tests and investigations by a security and privacy researcher in mobile environments. The decision to build this platform is because the tools that are available so far, may not be as clear and simple as possible to use, and even the way in which they present the information to the user, is not clear and simple as it must be expected. In addition, there is no distribution that brings together all the necessary tools in a single package, to avoid that professionals in the sector have to look for and install these software solutions when preparing a security and privacy analysis. All this, along with the creation of applications to speed up the aforementioned procedures, it has been the main reasons why it has been decided to undertake the creation of this practical learning and professional analysis environment.

5.2. Minimum Requirements

The Security Analysis Workshop has decided to mount on a Linux distribution, in its LTS (Long Term Support) version, to ensure not only stability, but also that the whole set of packages and dependencies used, remains updated. The system is packaged in an image intended to be used in a virtual machine such as: VirtualBox [141], or VMWare, which gives comfort when installing this application. The following Table 5.1, describes in detail not only the characteristics of the mounted system, but also the hardware required to mount it, and the steps to be followed if the system is to be deployed, in order to perform the research tasks.

Security Analysis Workshop	
Operating System [142]	Xubuntu 17.10 (Artful Aardvark) x86_64
Recommended RAM	4GB
Recommended GPU Memory	128MB or over
Recommended Disk Space	50GB or more
Tool's Repository	https://www.github.com/Hallec/SAWTools

Table 5.1 "Security Analysis Workwhop" Minimum Requirements

Download link for SAW

<https://drive.google.com/file/d/1u0Yg4MfxS6pcCQzg6YbyOx87gXfZaywh/view>

In addition to previously mentioned, it is necessary to install:

- Python 3.6.x (in order to run some scripts developed for this work).
- Java JDK 1.8 update 151 (to execute Android Studio, and other sort of software created for Android ecosystem).
- Golang 1.9.2 for compiling source code of some program developed for this Master Thesis.

5.3. How to install “Security Analysis Workshop”?

The installation process explained in this section is only for Virtualbox (the method for importing and installing this image, is practically equal in other VM solutions). It is important to follow these steps, in order to install, configure, and run the workshop properly.

Next, the necessary steps are described:

- 1) Download the Workshop image from Internet: link not available yet.
- 2) Open VirtualBox in your system.
- 3) Go to the tool bar menu and select File, and then import virtualized services...
- 4) When next window appears, you need to choose the OVF (Open Virtualization Format) file, in order to import it.
- 5) Wait until the process of importation is done. It could take more than five minutes.

Once the “Security Analysis Workshop” is imported, it is very important to recheck some properties which are paramount for the best performance of the virtual environment. For doing this process, the steps to follow are these ones:

- 1) Right click over the OS imported on the left column of VirtualBox.
- 2) Click over the first Option: “Configuration”.
- 3) Check in “System Section” if its base memory is not less than: 4096MB.
- 4) Make sure in the “Screen Section”, the video memory is not less than 128MB.

After doing these verifications, the security analyst will be ready to launch the workshop, and start working with it, in order of analyzing mobile apps. A picture about the “Security Analysis Workshop” is shown in Figure 5.1.



Figure 5.1. "Security Analysis Workwhop" first glance

Due to the size of the Linux distribution developed for this project, around 10GB, it is not possible to distribute this solution for security and privacy analysis on mobile platforms, through a removable media such as: CD/DVD, so we will choose to include it in a USB memory, or host it on a web page for download by users, who want to use it.

5.3.1. How to login and update the tools?

The user credentials are: saw and password: saw123. It is recommended to change the password (passwd a type new password twice), the first time the user logs in. To update, the user must launch the command: update_tools in a terminal.

5.4 Tools installed in “Software Analysis Workshop”

Based on the Related work, in order to create a platform for the analysis of security on mobile platforms, efficient and competent, we believe that the following tools should be present:

- quark (Exist): It is a script to perform auto-analysis over Android apps.
- apktool (Exist): an app for decompiling Android apps.
- androguard (Exist): its aim is for analyzing the structure of an Android app.
- dex2jar (Exist): for converting dalvik executable to jar.

- JD-GUI (Exist): graphical tool for displaying the decompiled source code.
- idevicebackup2 (Exist): tool for backing up iOS app information.
- tcpdump (Exist): tool for network analysis.
- tshark (Exist): advanced command-line tool for network analysis.
- wireshark (Exist): graphical interface for tshark.
- Burp suite (Exist): powerful proxy for security analysis.
- SQLiteMan (Exist): database manager for SQLite.
- DVIA (Exist): Damn Vulnerable iOS app.
- dd (Exist): powerful command-line for copying/converting information
- exiftool (Exist): software for extracting metadata for mobile resources.

The main limitations of these tools is in general they are software for professionals, and in most of the cases there is no way to control its workflow. Plus, in general these utilities display its information in a technical way, without taking into account users without any knowledge about mobile platforms.

That is why, we think in order to improve the visualization and automation of some tasks by the user, in addition to making certain tools more affordable to the novice user, it should exist the following:

- androguard_downloader (Does not exist): script for downloading automatically androguard tools.
- apktool_checker (Does not exist): script for installing and setting up apktool.
- apk_downloader (Does not exist): program for download automatically apps to be analyzed, in Android platform.
- manifest_interpreter (Does not exist): script for interpreting and extracting permission information.
- metada_extractor (Does not exist): script for extracting sensitive information for mobile resources, which allows to interact with the resource itself.

Next, we attach a Table 5.2, with all the tools integrated and developed in this platform (created or not developed for this project), which will be able to use to perform security and privacy analysis over the mobile ecosystem.

TASK	TOOL	Nature	Integrated	SCOPE
Auto-analysis over Android	quark	E	I	Professional
Decompiling Android apps	apktool	E	I	Professional
Analyze structure of an app	androguard	E	I	Professional
Convert dex file to jar	dex2jar	E	I	Professional/User
Display source code of classes	JD-GUI	E	I	Professional
Decypher iOS apps	Clutch	E	C	Professional
Tweak iOS apps	Cycript	E	C	Professional
Disassembler for mac/iOS	Hopper	E	C	Professional
Perform backups over iOS	idevicebackup2	E	I	Professional/User
Create virtual iOS network	rvictl	E	C	Professional/User
Network analysis	tcpdump	E	I	Professional
Networking over terminal	tshark	E	I	Professional
GUI version of tshark	Wireshark	E	I	Professional
Network proxy	Burp suite	E	I	Professional/User
Database Manager	SQLite Man	E	I	Professional/User
Learning iOS app	DVIA	E	I	Professional
Tool for copying/converting	dd	E	I	Professional/User
Metadata collector	exiftool	E	I	Professional/User
Update androguard suite	androguard_downloader	D	I	Professional/User
Update apktool	apktool_checker	D	I	Professional/User
Downloader of Android apps	apk_downloader	D	I	Professional/User
Display manifest info	manifest_interpreter	D	I	Professional/User
Extractor of metadata	metadata_extractor	D	I	Professional/User
IDE for iOS apps	Xcode	E	C	Professional
IDE for Android apps	Android Studio	E	I	Professional

Table 5.2. Tools integrated in "Security Analysis Workshop"

Legend		
- Nature: E (Existent). D (Developed).	- Integrated: I (Integrated) C (Complementary)	-Scope: Professional Professional/User

5.4.1. Integrated tools

Some of the main utilities installed, and ready to use in the “Software Analysis Workshop” are listed next (See section 4.9. Security Analysis Tools for further information):

- quark
- apktool
- androguard
- dex2jar
- JD-GUI
- idevicebackup2
- tcpdump
- tshark
- wireshark
- burp suite
- SQLite Manager
- DVIA
- dd
- exiftool
- Android Studio

5.4.2. Tools developed for this Master Thesis

In this section, it is a detailed description about a set of programs and scripts that have been developed, during the development of this document, to help to automate the security and privacy analysis process in the main mobile platforms of the market: iOS & Android.

The following scripts created for that purpose are:

- **androguard downloader**: a script that downloads the latest version of the Android app analysis utility called: Androguard.
- **apktool checker**: allows you to keep the latest version of the tool apktool updated, which among other tasks, allows: decompile and package a certain Android app.
- **apk downloader**: this a script that allows downloading an Android app from an unofficial store.
- **manifest interpreter**: is a program that allows to automate the decompilation process, and analysis of the manifest file in Android applications. This manifest file has among other tasks to host the set of permissions that are requested by this app, to be able to perform the functions for which it has been developed, during runtime.
- **metadata extractor**: is a script which dump metadata associated to a certain file, doing special attention in that information of private character, as it can be: name of the owner, geolocation data, among others, and in that case of being found, then creates a json file

for a more comfortable importation/exportation by other existing tools, as well as create a map with the location of the user when it has generated that resource.

These utilities are available in the github repository: www.github.com, besides that they are consequently deployed in the Linux distribution that accompanies the development of this final Master thesis, in the Scripts folder of the home directory of the default user (see previous sections where the Security Analysis Structure's structure is detailed).

After this brief preamble, a detailed description of each of the developed utilities will be carried out, explaining how they work, as well as how they have been developed through an organizational chart that allows to visualize the internal structure of the program or script, in order to be able to visualize what its mission is.

Inner structure of the directory that hosts the scripts

The content of the directory Script, is distributed in the following folders:

-call_command: Class developed in Go [143] that allows invoking any command provided by the shell of a system, and returns the response of that command parsed for the use by other programs. The program **manifest_interpreter** makes use of it.

-configparser: this is another class created in Go, and used by the program: **manifest_interpreter**, whose task is to load and interpret the included configuration files in the conf directory.

-conf: is a folder that contains the main configuration files of the scripts presented in this project. These configuration files are: **api_versions.cfg** (descriptive file of the Android environment APIs), and **permissions.cfg** (where it is included a classification, according to its degree of danger, of the set of permissions that can be declared in an application for the Operating System: Android).

-input: saves the applications in apk format, which are downloaded by the script: **apk_downloader**, for later analysis by the security researcher.

-output: this is the directory where the contents of the application are dumped from its apk format, once it has been decompiled. At the same time, it is also here where the program: **manifest_interpreter**, exports the information processed during its execution in json format into a folder with the same name.

With this group of directories, together with the scripts and programs mentioned above, it is possible to automate a great number of the procedures that must be carried out by a security researcher during its analysis of applications in mobile ecosystems.

1. androguard

It is nothing more than a bash script, which downloads the tool repository using the git command (control version system). If this script is executed, the most up-to-date version of Androguard will be downloaded, in the user's home directory, in the androguard folder, so that it can be executed by the security researcher when he/she wants

to perform an analysis. Due to the simplicity of the program, in this case it will not show an associated organization chart, nor give more details of it, since its content is only the following sentence:

```
git clone https://github.com/androguard/androguard.git (which downloads the content from the master repository to the local system of the user invoking it).
```

2. check apktool

It is a script developed in Python, which allows you to download the latest version (currently 2.3.0) of the Android apps decompilation tool: apktool. In order to perform this procedure, the script saves a log into the conf folder, with the apktool version installed on the system in versions.cfg file. In this way, whenever the command line script is launched (./apktool_checker), it is first checked that the developer's website does not have any more updated version, so as not to repeat the installation procedure unnecessarily.

In the event that there is a more updated version, the script will download the new version of the tool, in the associated temporary directory tmp, and then it will proceed to perform the relevant tasks in order to update it correctly in the system. It is necessary, in order to execute it, that the user is identified as root (super-user), since certain directories where the tool is installed, are not accessible to the normal user of the system (saw).

In addition, part of the commented source code is also attached in Table 5.3, so that it can be explained what steps the script actually follows when installing the apktool tool in the system.

```
#!/usr/bin/env python3

import os
import re
import requests
import datetime
import subprocess
from collections import OrderedDict
from bs4 import BeautifulSoup as bs
import time

VERSIONS = OrderedDict()

TERM_COLORS = {"red": "\033[91m",
               "green": "\033[92m",
               "end": "\033[0m",}

#Function for getting the current time
def get_time():
    return datetime.datetime.now().strftime("%d/%m/%Y - %H:%M:%S")

#Function for moving the file to the correct bin folder
def mv(current_path, future_path):
    os.mv(current_path, future_path)

#Function for removing files in temp folder
def rm(files):
    files_path = map(lambda f: "tmp/{}".format(f), files)
    for f in files_path:
        subprocess.Popen(["rm", "-rf", f])

#Function for setting correct permissions for files a list of files
def chmod(permission, files):
    files_path = list(map(lambda f: "tmp/{}".format(f), files))
    for cfile in files_path: os.chmod(cfile, permission)
```

```

#Function for updating configuration files
def update_cfg_files():
    with open("cfg/versions.cfg","w") as f:
        for key in VERSIONS:
            f.write("{}={}\n".format(key,VERSIONS[key]))

#Checking apktool version installed in the system
def check_current_version():
    with open("cfg/versions.cfg","r") as f:
        lines = f.read().splitlines()

    for line in lines:
        software,version = line.split("=")
        VERSIONS[software.strip()] = version.strip()

    apktool_web = requests.get("https://bitbucket.org/iBotPeaches/apktool/downloads/").text
    html = bs(apktool_web,"html.parser")
    uploaded_files = html.find_all("table",{ "id": "uploaded-files"})[0].find_all("td",{ "class":"name" })
    latest_version = re.search(r"^(.*)\.jar$",uploaded_files[1].find("a")["href"]).group(1)
    if latest_version > VERSIONS["APKTOOL_VERSION"]:
        print("{} Downloading wrapper script...".format(get_time()))
        wrapper = requests.get("https://raw.githubusercontent.com/iBotPeaches/Apktool/master/scripts/linux/apktool",
        stream=True)

        with open("tmp/apktool","wb+") as f:
            for chunk in wrapper.iter_content(chunk_size=1024):
                if chunk: f.write(chunk)
                print("{} Downloading latest version of apktool {}...".format(get_time(),latest_version))

    apkjar = requests.get("https://bitbucket.org/iBotPeaches/apktool/downloads/apktool_{}.jar".format(latest_version),
    stream=True)

    with open("tmp/apktool.jar","wb+") as f:
        for chunk in apkjar.iter_content(chunk_size=1024):
            if chunk: f.write(chunk)
            print("{} Download process complete...".format(get_time()))
            print("{} Fixing permissions for files...".format(get_time()))
            chmod(0o755,["apktool","apktool.jar"])
            print("{} Moving apktool to bin directory...".format(get_time()))

    try:
        for cfile in ["apktool","apktool.jar"]: mv("tmp/{}".format(cfile),"/usr/local/bin/{}".format(cfile))
        pass
    except:
        print(TERM_COLORS["red"],"* [ERROR] You need root permission to move files.",TERM_COLORS["end"])
        print("{} Cleaning temp directory...".format(get_time()))
        rm(["apktool","apktool.jar"])
        exit(-1)
    print("{} Updating configuration files...".format(get_time()))
    print(TERM_COLORS["green"],"{} apktool is ready to use.".format(get_time()),TERM_COLORS["end"])
    VERSIONS["APKTOOL_VERSION"] = latest_version
    update_cfg_files()
    else:
        print(TERM_COLORS["green"],"* {} You have installed the current version of apktool. No updates
        available.".format(get_time()),TERM_COLORS["end"])

if __name__ == "__main__": check_current_version()

```

Table 5.3. Source code snippet from check_apktool

The Figure 5.2 shows a simple organizational chart of the functions involved in this script:

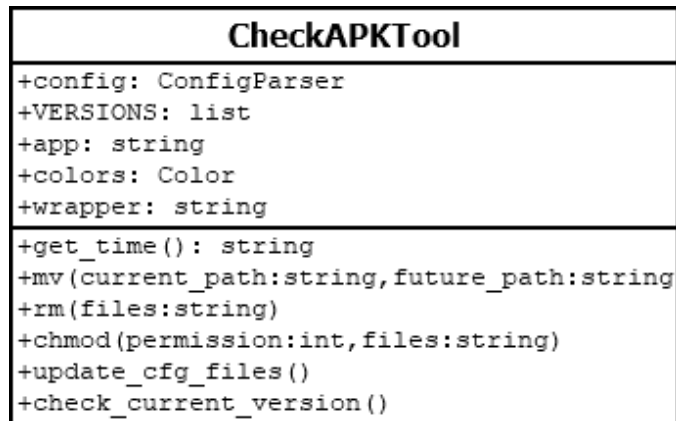
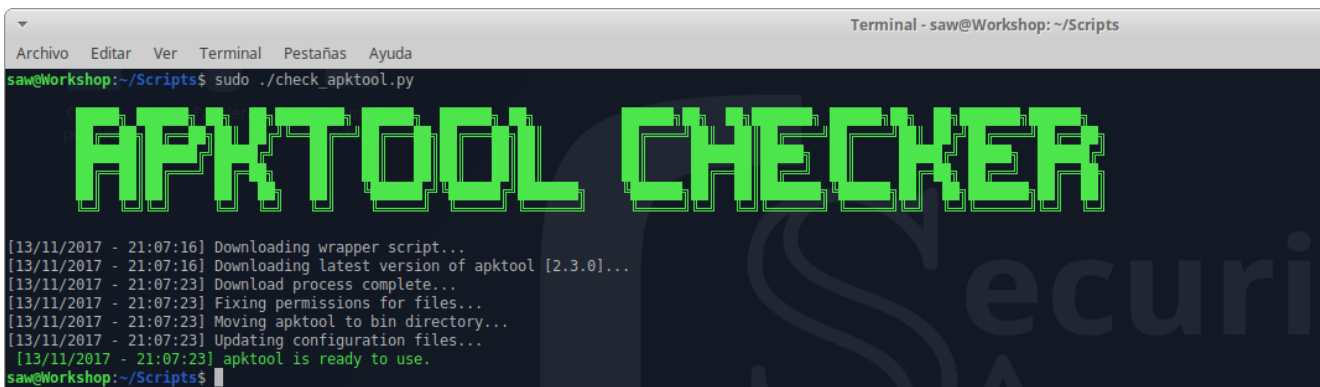


Figure 5.2. Organizational Chart for CheckAPKTool

Finally, a Figure 5.3. is shown that corresponds to the information dump after the execution of check_apktool from a command-line terminal:



```
Terminal - saw@Workshop: ~/Scripts
saw@Workshop:~/Scripts$ sudo ./check_apktool.py
APKTOOL CHECKER
[13/11/2017 - 21:07:16] Downloading wrapper script...
[13/11/2017 - 21:07:16] Downloading latest version of apktool [2.3.0]...
[13/11/2017 - 21:07:23] Download process complete...
[13/11/2017 - 21:07:23] Fixing permissions for files...
[13/11/2017 - 21:07:23] Moving apktool to bin directory...
[13/11/2017 - 21:07:23] Updating configuration files...
[13/11/2017 - 21:07:23] apktool is ready to use.
saw@Workshop:~/Scripts$
```

Figure 5.3. check_apktool execution

3. apk downloader

As in the case of apktool_checker, this script is newly developed thanks to the use of the programming language: Python, which allows a security researcher to automate in an agile and fast way, procedures that can become repetitive. On this occasion, the purpose of this script is to download the applications in Android apk format, in order to analyze them from the computer of the security analyst, to discover the risks involved in their installation.

The downloaded mobile applications are not hosted in the official store (Google Play), but are stored on the web: <https://www.apkmirror.com>, which hosts a large number of the most common applications on the Google platform. One of the advantages of this unofficial store is that it allows you to download more obsolete versions of a certain app, allowing the analyst to evaluate its evolution over time, besides studying the fixes made by its developers to solve a certain issue, which led to the creation of a new build. It can also be used for testing purposes to determine if the app to study has any mechanism that could jeopardize the privacy of users who install it.

Due to the reasons listed in the previous paragraph, once the script is executed by command line (./apk_downloader), the user is asked to enter the name of the Android mobile application that he/she wants to download. In the case of not being in the web, the user will be indicated by warning message on the screen, but in the case of existing versions, a list of available versions will be returned which can be downloaded to the analyst's local directory.

The security researcher must then select the version he/she wants to download, and then the script will search for the set of hardware architectures available in which the apk can be downloaded (usually ARM [144] should be selected, although there may be occasions to choose other options in order to enrich the research). After selecting the architecture, the script will prompt the user to enter the name under which the apk is to be saved (default will use the search term used in previous steps).

If everything has been done correctly, the selected application will be downloaded to the input folder of the Scripts directory, so that it can be analyzed in later steps by another one of the programs developed for this Final Master Thesis: manifest_interpreter.

A sample of a part of the source code of this script, commented in order to detail its operation, can be seen in the Table 5.4 below:

```
apk = input("Introduce the app name you want to download: ")

#Checking up the different options
response = requests.get("https://www.apkmirror.com/?post_type=app_release&searchtype=apk&s={}".format(
apk.lower()))
html_response = bs(response.text,"html.parser")
res = html_response.find_all("div",{"class":"addpadding"})

#Function for listing every download option
def list_options(title,opt_list):
    valid = False
    while not valid:
        print(title)
        for index in opt_list:
            print("{} {}".format(index,opt_list[index]["name"]))
        try:
            opt = int(input("Introduce the app version, you want to download: "))
            if opt < 1 or opt > len(opt_list):
                if len(opt_list) == 1: print("[ERROR] The option must be: 1")
                print("[ERROR] The option must be between: 1 and {}".format(len(opt_list)))
            else:
                return opt
        except:
            print("[ERROR] The option chosen must be numeric".format(len(opt_list)))

#The app is found in the website
if not res:
    list_widget = html_response.find_all("div",{"class":"listWidget"})
    rows = list_widget[0].find_all("div",{"class":"appRow"})
    print("App found. {} {}".format(len(rows),"results" if len(rows) > 0 else "result"))
    apks_list = {}

    for index,row in enumerate(rows):
        a_link = row.find("a")
        apk_name = a_link.text
        apk_link = a_link["href"]
        apks_list[index+1] = {"name": apk_name, "link": apk_link}
```

```

#Listing every option found
opt = list_options(title="--- APPS LIST ---",opt_list=apks_list)

for index in apks_list:
    print("{}[{}]. {}".format(colors["green"] if index == opt else "", "*" if index == opt else " ",
    apks_list[index]["name"], colors["end"] if index == opt else ""))
    print("\n"*2)

print("Searching available architectures...")
res_download = requests.get("https://www.apkmirror.com/{}".format(apks_list[opt]["link"]))
res_download = bs(res_download.text, "html.parser")
list_widget = res_download.find_all("div", {"class": "listWidget"})[0]
rows = list_widget.find_all("div", {"class": "table-row headerFont"})
archs_list = {}

for index, row in enumerate(rows[1:]):
    a_link = row.find("a")
    a_name = a_link.text.strip()
    apk_link = a_link["href"]
    arch = row.find_all("div", {"class": "table-cell rowheight addseparator expand pad dowrap"})[1].text
    archs_list[index+1] = {"name": a_name, "link": apk_link, "arch": arch}
#Listing hardware architectures available
opt = list_options(title="--- ARCHITECTURES LIST ---",opt_list=archs_list)

for index in archs_list:
    print("{}[{}]. {}".format(colors["green"] if index == opt else "", "*" if index == opt else " ",
    archs_list[index]["name"], colors["end"] if index == opt else ""))
    print("\n"*2)
res_apk = requests.get("https://www.apkmirror.com/{}".format(archs_list[opt]["link"]))
res_apk = bs(res_apk.text, "html.parser")

download_link = "https://www.apkmirror.com/{}".format(res_apk.find("a", {"class": "btn btn-flat
downloadButton"})["href"])

apk_name = input("Choose a name to save the file. (By omission: {}): ".format(apk))

if apk_name:
    apk = apk_name
#Downloading the Android app to input directory, in order to analyzing its manifest file
app_apk = requests.get(download_link, stream=True)

try:
    with open("input/{}.apk".format(apk), 'wb') as f:
        for chunk in app_apk.iter_content(chunk_size=1024):
            if chunk:
                f.write(chunk)
                print("* apk: {}.apk downloaded successfully.".format(apk))
except:
    print("[ERROR] While downloading app. Retry again.")

#App not found in the website
else:
    print("APK not found. Please, try another Android app..")

```

Table 5.4. Source Code snippet from apk_downloader

In the following Figure 5.4 is shown an organizational chart, which summarizes its internal structure:

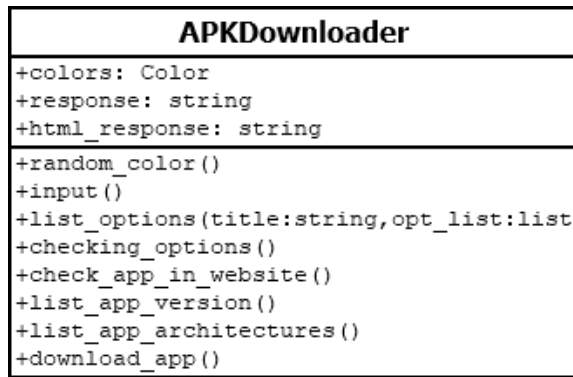


Figure 5.4. Organization chart for APKDownloader

Finally, in this Figure 5.5, it can be seen a simulation of a normal execution of this script, in order to download the Instagram app for a later analysis.

```

Terminal - saw@Workshop: ~/Scripts
saw@Workshop:~$ cd Scripts/
saw@Workshop:~/Scripts$ ./apk_downloader.py

APK DOWNLOADER

Introduce the app name you want to download: Instagram
App Found: 10 results.

--- APPS LIST ---
1. Instagram 23.0.0.12.135 (79757)
2. Instagram 23.0.0.6.135 (79369)
3. Instagram 22.0.0.17.68 (78982)
4. Instagram 23.0.0.2.135 (78983)
5. Instagram 22.0.0.15.68 (78610)
6. Instagram 22.0.0.8.68 (78246)
7. Instagram 22.0.0.3.68 (77973)
8. Instagram 21.0.0.11.62 (77790)
9. Instagram 21.0.0.8.62 (77472)
10. Instagram 21.0.0.3.62 (77131)
Introduce the app version, you want to download: 1
[*] Instagram 23.0.0.12.135 (79757)
[ ] Instagram 23.0.0.6.135 (79369)
[ ] Instagram 22.0.0.17.68 (78982)
[ ] Instagram 23.0.0.2.135 (78983)
[ ] Instagram 22.0.0.15.68 (78610)
[ ] Instagram 22.0.0.8.68 (78246)
[ ] Instagram 22.0.0.3.68 (77973)
[ ] Instagram 21.0.0.11.62 (77790)
[ ] Instagram 21.0.0.8.62 (77472)
[ ] Instagram 21.0.0.3.62 (77131)

Searching available architectures...
--- ARCHITECTURES LIST ---
1. 7975769
Introduce the app version, you want to download: 1
[*] 7975769

Choose a name to save the file. (By omission: instagram):
* apk: instagram.apk downloaded successfully.
saw@Workshop:~/Scripts$

```

Figure 5.5. apk_downloader execution

4.manifest interpreter

To finish this section, the program presented below is the most complex one developed to form part of the final thesis of this master, and has been programmed in: Go, a compiled programming language that has among its main assets: efficiency and concurrency in recurring tasks.

The purpose of this program, which can be executed via command line by means of: ./manifest_interpreter, is first of all to proceed to the automatic decompiling of a

certain mobile app indicated by the user. To do this, initially presents the security analyst with a list of the ppks downloaded to the system, which are stored in the 'input' directory (see `apk_downloader`), in order to request which one the security analyst wants to decompile, to proceed with a further analysis. Once the researcher selects one of the available apps, it will proceed to decompile it, which will dump all the files (classes, smali code, manifest file among others), to the output folder, from which it will later be able to analyze the source code of the application under study, with the use of other security analysis tools, which have already been presented during previous sections of this document.

Once the decompilation process is finished, the tool searches for the manifest file in the output folder, which is where the `manifest_interpreter` program dumps the data obtained after the decompilation of the mobile application selected by the user, and once it finds the manifest file, it loads it into the system, and proceeds to its analysis.

In this procedure, the information contained in this file will be shown in a structured way, including: activities, services, receivers, providers, and other components typical of the Android ecosystem, as well as the version of the API used during the development of the app, which determines the version of the Operating System of the mobile platform on which it can be installed (This association is made through the file `api_versions.cfg`).

As a final part of all this information, the user is provided with a list of the permissions used by the application, which shows a description, in addition to being assigned a category: DANGER, NEUTRAL, DEPRECATED, or GOOD (See Android Permission Classification in previous chapters) according to the impact on privacy and security that could have the use of that permission by the mobile application. This classification has been established in the file: `permissions.cfg`, which is located inside `conf`, in the Scripts directory.

It should be noted that this assessment is a generalised classification of the impact that such a permission could have, and that it should not be taken into account in a strict manner, since it is up to the user (in this case the security analyst) to determine whether or not such permission is necessary. As has been explained on numerous occasions during this document, one of its main objectives is to raise awareness, and because of the dangerousness involved in the use of permissions in mobile applications, the program: `manifest_interpreter` has been developed that allows the security researcher to check at a glance what permissions are used for, leaving the user of the application, the responsibility for determining whether the application being processed makes use of a correct, or abusive way of the permissions declared within it.

After the procedure of analysis the manifest, the last stage that the program performs before its completion, is the creation of a json file with a summary of everything explained above. By default, the manifest file is a XML format, but after the execution of `manifest_interpreter`, a JavaScript Object Notation file will be generated in the `output/json` folder, which is more readable and allows it to be imported/exported more easily by other types of web and desktop analysis tools.

A small snippet in Table 5.5 from the source code with comments, for illustrating how part of the described tasks are carried out during the execution of the program:

```

//Manifest Overview
type ManifestLevel struct {
    Color Colors
    Manifest xml.Name `xml:"manifest"`
    Package_Name string `xml:"package,attr"`
    Schema string `xml:"android,attr"`
    Android_Version_Name string `xml:"versionName,attr"`
    Version_Code int8 `xml:"versionCode,attr"`
    Install_Location string `xml:"installLocation,attr"`
    App Application `xml:"application"`
    MinSDKVersion UseSDKTag `xml:"uses-sdk"`
    ScreenSupport ScreenSupportTag `xml:"supports-screens"`
    Permissions []PermissionTag `xml:"uses-permission"`
    PermissionsSdk23 []PermissionTag `xml:"uses-permission-sdk-23"`
    APIConfig map[string] string
    PermissionConfig map[string] string
}

//Function for casting the manifest information to string
func (m ManifestLevel) String() string {
    m.Color.New()
    var buffer bytes.Buffer
    s := fmt.Sprintf("%s%s%s\n", colors.Yellow, "----- Manifest Overview -----", colors.End)
    buffer.WriteString(s)
    s = fmt.Sprintf("* Package Name: %s\n", m.Package_Name)
    buffer.WriteString(s)
    s = fmt.Sprintf("* Schema: %s\n", m.Schema)
    buffer.WriteString(s)
    s = fmt.Sprintf("* Android Version Name: %s\n", m.Android_Version_Name)
    buffer.WriteString(s)
    if m.Version_Code != 0 {
        s = fmt.Sprintf("* Version Code: %d\n", m.Version_Code)
        buffer.WriteString(s)
    }

    s = fmt.Sprintf("* Install Location: %s\n", m.Install_Location)
    buffer.WriteString(s)
    buffer.WriteString(m.App.String())
    buffer.WriteString(m.MinSDKVersion.String(m.APIConfig))
    buffer.WriteString(m.ScreenSupport.String())
    buffer.WriteString(fmt.Sprintf("\n\t%s%s%s\n", colors.Yellow, "--- Permissions ---", colors.End))
    if len(m.PermissionsSdk23) != 0 {
        m.Permissions = append(m.Permissions, m.PermissionsSdk23...)
    }

    for _, value := range m.Permissions {
        buffer.WriteString(value.String())
        desc := strings.Split(m.GetPermissionDesc(value.Name), "|")
        if len(desc) != 2 {
            desc = []string{"UNKNOWN", "Customized permission. Description not available."}
        }
        grade := desc[0]
        description := desc[1]
        color_permission := colors.GetGradeColor(grade)
        buffer.WriteString(fmt.Sprintf("%s\t\tF [%s] %s%s\n", color_permission, grade, description, colors.End))
        buffer.WriteString(NEW_LINE)
    }
    return buffer.String()
}

//Function for dumping Manifest Information to a json file
func (m ManifestLevel) ToJSON()string {
    var res []string
    res = append(res, fmt.Sprintf("\"package_name\": \"%s\"", m.Package_Name))
    res = append(res, fmt.Sprintf("\"schema\": \"%s\"", m.Schema))
}

```

```

res = append(res,fmt.Sprintf("\"android_version\": \"%s\"",m.Android_Version_Name))

if m.Version_Code != 0 {
    res = append(res,fmt.Sprintf("\"version_code\": \"%s\"",m.Version_Code))
}else {
    res = append(res,fmt.Sprintf("\"version_code\": \"not specified\""))
}
res = append(res,fmt.Sprintf("\"install_location\": \"%s\"",m.Install_Location))

var permission_value []string
if len(m.PermissionsSdk23) != 0 {
    m.Permissions = append(m.Permissions,m.PermissionsSdk23...)
}

for _,value := range m.Permissions {
    desc := strings.Split(m.GetPermissionDesc(value.Name),",")
    if len(desc) != 2 {
        desc = []string{"UNKNOWN","Customized permission. Description not available."}
    }
    grade := desc[0]
    description := desc[1]
    permission_value = append(permission_value,fmt.Sprintf("<%s>: %s -> %s",value.Name,description,grade))
}
permission_res := fmt.Sprintf("\"permissions\": [%s]",strings.Join(permission_value,","))

res = append(res,permission_res)
res = append(res,m.App.ToJSON())
return fmt.Sprintf("{\"manifest\":{\"%s}}",strings.Join(res,","))
}

//Function for writing the json file in a certain folder
func (m ManifestLevel) Write(name string,ext string) {
    if ext == "json"{
        f_ := os.Create(fmt.Sprintf("output/json/%s_manifest.json",strings.Split(name, ".apk")[0]))

        defer f.Close()
        w := bufio.NewWriter(f)
        w.WriteString(m.ToJSON())
        w.Flush()
    }
}

//Function for setting the initial configuration to the program
func (m *ManifestLevel) SetConfig() {
    var api_config configparser.ConfigParser
    api_config.Load(API_VERSION_FILENAME)
    var permission_config configparser.ConfigParser
    permission_config.Load(PERMISSION_FILENAME)
    m.APIConfig = api_config.GetConfigOpts()
    m.PermissionConfig = permission_config.GetConfigOpts()
}

//Function for getting the description information for a certain permission
func (m ManifestLevel) GetPermissionDesc(permission string) string {
    index_dot := strings.LastIndex(permission, ".") + 1
    permission = permission[index_dot:]
    return m.PermissionConfig[permission]
}

```

Table 5.5. Source code snippet from manifest_interpreter

In addition, an explanatory organizational chart is attached in Figure 5.6., in which it can be glimpsed all the functions that make up the program, as well as its functionality:

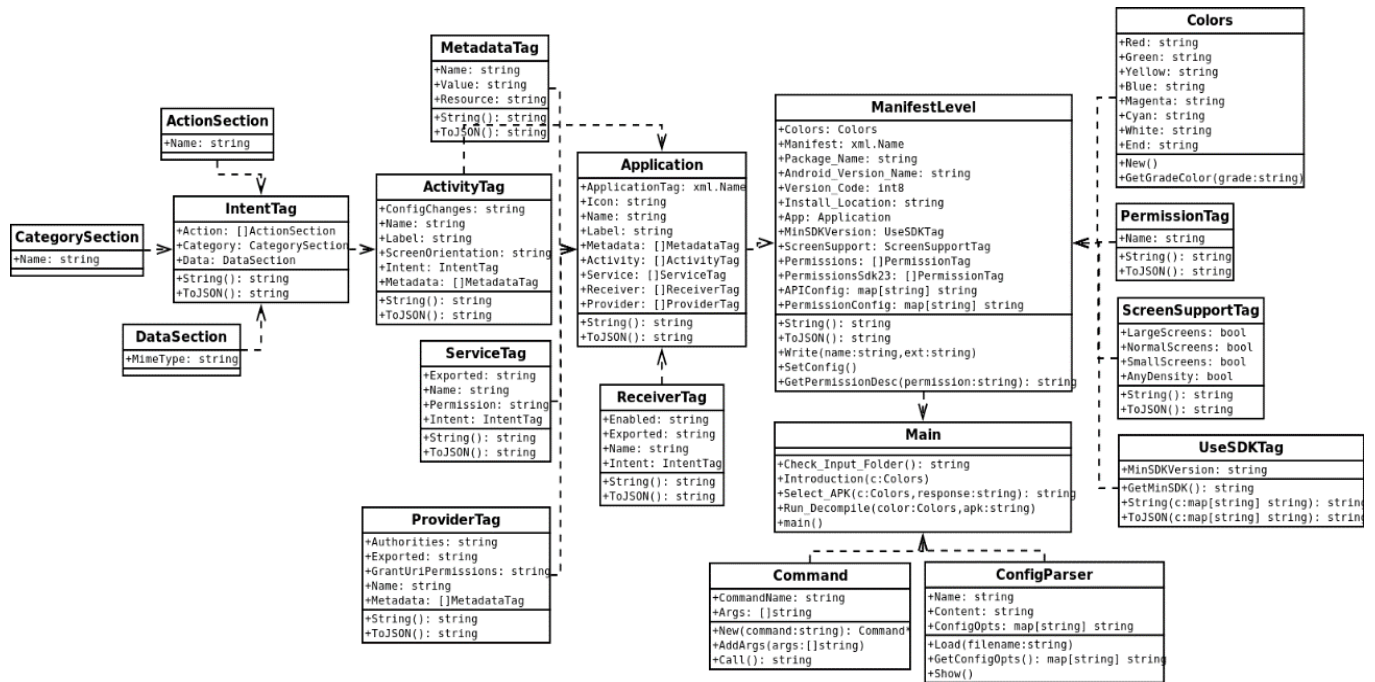


Figure 5.6. Organizational Chart for Manifest Interpreter

Finally, in the Figure 5.7 shown below, it can be seen a normal execution of the manifest_interpreter program, which returns the analysis of the Instagram application manifest file, previously downloaded. A list of Android Permissions can be seen in Table B1, in Annexes.

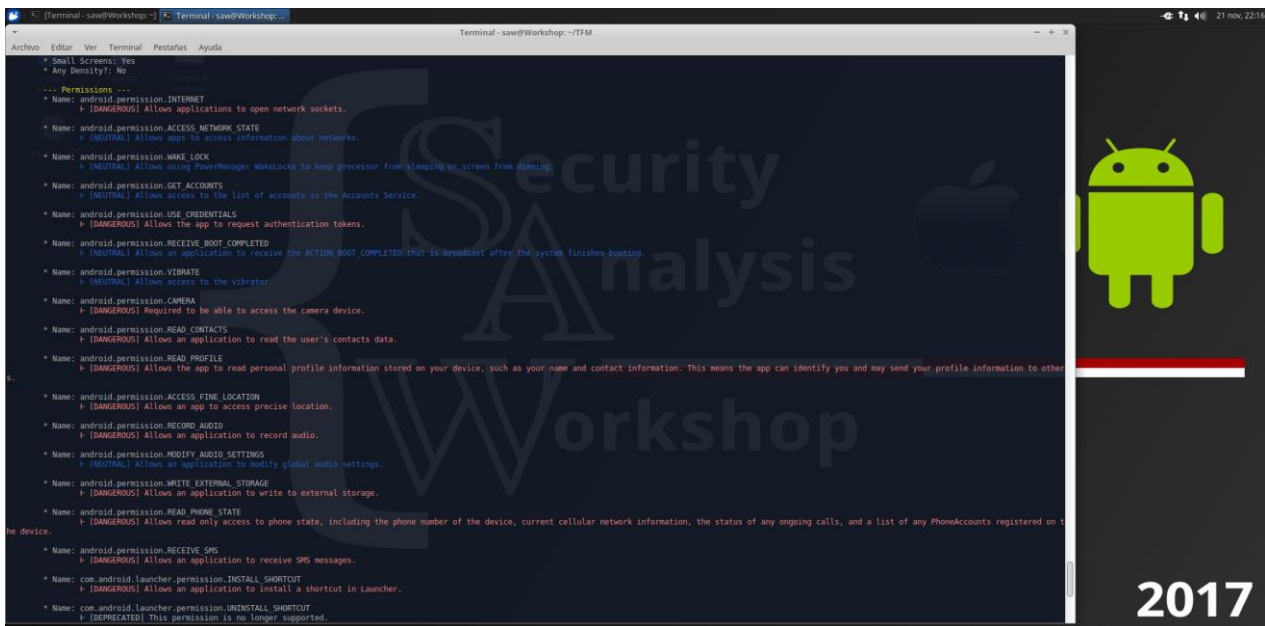


Figure 5.7. Manifest Interpreter running

5. metadata extractor

This script has been created to automate the process of extracting metadata from files, and which is included as another utility in "Security Analysis Workwhop". to speed up the process of security scanning on mobile platforms.

To launch the tool, firstly the analyst needs to configure a Google Maps API Key (a detailed explanation will be found in the "Annexes" of this project, in a section called: "How to get a Google Maps API Key"). Once this key is set, simply open a command terminal and invoke the following: `metadata_extractor`. The process programmed for this script will then start, asking the user to select the file, or image within the input folder of the Scripts directory, that you want to be analyzed. Once this has been done, the analysis process carried out by the tool will begin, in which all the information associated with the file that can be extracted will be collected firstly, and then the information that can be considered of a personal nature will be classified.

Then, the user will be asked to extract any geolocation information associated with the file. If so, the coordinates of: altitude, latitude and longitude will be obtained in order to discover where the resource was generated. Subsequently, a screenshot of the location on a map, which has been obtained in this previous step, will be saved in the directory: `output/maps` with the same name as the file to be analyzed, but with `png` extension.

To finish this procedure, all the data considered as: sensitive that has been extracted during the execution of the script, it will be grouped in a dictionary, and later it will be dumped to a JSON (Javascript Object Notation) file, which allows its import and interpretation by other solutions present in the market, in quick and easy way.

Finally, there is also the option of the user to be able to anonymize a particular file, in order to remove all those metadata that can be interpreted to give information about himself/herself. To do this, simply run the command: `metadata_extractor --anonymize`, and select the desired file.

Part of the Python language code with which the script was developed is attached in the Table 5.6 below.

```
#!/usr/bin/env python3

import os
import re
import time
import json
import secrets
import requests
import datetime
import argparse
import googlemaps
import subprocess
import configparser
from PIL import Image
from selenium import webdriver
from selenium.webdriver.chrome.options import Options

class MetadataExtractor:
    #Terminal colors
```

```

colors = {
    "blue" : '\033[94m',
    "green" : '\033[92m',
    "yellow" : '\033[93m',
    "red": '\033[91m',
    "end": '\033[0m',
}

#Constructor
def __init__(self):
    self.metadata = {}
    self.gps_tags = {}
    self.config = configparser.ConfigParser()
    self.config.read("conf/api_keys.cfg")
    self.config.sections()

    #Enable key/value, gps and latitude/longitude detection
    self.__create_key_value_regex()
    self.__detect_location_tags()
    self.__extract_lat_long_values()

#Function for selecting a random color for terminal messages
def __random_color(self):
    choice = secrets.choice(range(len(self.colors)-2))

    for i,color in enumerate(self.colors):
        if i == choice: return self.colors[color]

#Function for getting current date time
def __get__time(self):
    return datetime.datetime.now().strftime("%A - %d-%m-%Y at %H:%M:%S").capitalize()

#Function for checking Google Maps API key
def __checking_api_key(self):
    try:
        self.api_key = self.config["KEYS"]["GoogleMaps"].strip()
        if self.api_key == "":
            print("[ERROR] The Google Maps API Key is empty. Please, introduce
your key in order to use this service. (conf/api_keys.cfg)")
    except:
        exit()

#Function for checking the input folder in order to find files
def check_input_dir(self):
    self.files = os.listdir("input/metadata")
    if not self.files:
        print("[ERROR] There is no files in the input/metadata folder. Please drop some files
in order to be analyzed.")
        exit()
    else:
        self.list_input_dir()

#Function for listing the content of the input folder
def list_input_dir(self):
    valid = False
    files_size = len(self.files)
    while not valid:
        print("\n"*3)
        print("What file do you want to analyze?")
        print("-----")
        for index,file in enumerate(self.files):
            print("{}). {}".format(index+1,file))
        try:
            self.option = int(input("Select an option: "))

```

```

        except:
            self.option = 0
        if self.option >= 1 and self.option <= files_size:
            valid = True
        else:
            if files_size == 1:
                print("[ERROR] The option must be a numeric argument of 1")
            else:
                print("[ERROR] The option must be a numeric argument from: 1
to {}".format(files_size))
        self.file_selected = [self.files[self.option-1]]

#Function for mapping the file selected to a folder format
def __map_file_folder(self):
    self.file_selected = list(map(lambda x: "input/metadata/{}".format(x),self.file_selected))

#Function for calling the exiftool command
def call(self):
    command = ["exiftool"]
    self.__map_file_folder()
    command.extend(self.file_selected)
    self.output =
subprocess.Popen(command,stdout=subprocess.PIPE,stderr=subprocess.DEVNULL).communicate()
    self.__parse()
    self.prettify(self.file_selected[0])

#Function for calling the exiftool command with arguments
def call_with_args(self,args,output=True,overwrite=False):
    command = ["exiftool"]
    #self.__map_file_folder()
    command.extend([args])

    #To overwrite original file
    if overwrite: command.extend(["-overwrite_original"])
    command.extend(self.file_selected)

    lines =
subprocess.Popen(command,stdout=subprocess.PIPE,stderr=subprocess.DEVNULL).communicate()[0].decode("u
tf-8").splitlines()
    if not output: lines = []
    response_command = {}
    for line in lines:
        #Cleaning useless characters from output
        response = self.key_value_regex.search(line).groupdict()
        key,value = response["key"].strip(),response["value"].strip()
        response_command[key] = value
    return response_command

```

Table 5.6. Source code snippet from metadata_extractor

The following Figure 5.8 shows an organizational chart with the main structure of the script


```

MetadataExtractor
+colors: dictionary
+metadata: dictionary
+config: ConfigParser
+api_key: string
+files: list
+option: int
+file_selected: list
+key_value_regex: regex
+location_regex: regex
+lat_long_regex: regex
+private_information: dictionary
+gps_tags: dictionary
+parser: ArgParse

+__init__()
-__random_color()
-__get_time()
-__checking_api_key()
+introduction()
+check_input_dir()
+list_input_dir()
-__map_file_folder()
+call()
+call_with_args(args:list)
-__parse()
-__create_key_value_regex()
-__detect_location_tags()
-__extract_lat_long_values()
+prettify(file_name:string)
+lat_long_to_decimal()
-__parse_place()
+collect_private_information()
-__display_place(place_information:dictionary)
-__screen_size()
-__save_screen()
-__dump_json()
+anonymize()
+fake()
    
```

Figure 5.8. Organizational chart for Metadata Extractor

To finish this section, a Figure 5.9. is shown, in which it can be seen the tool in question, running in the security analyst's system.

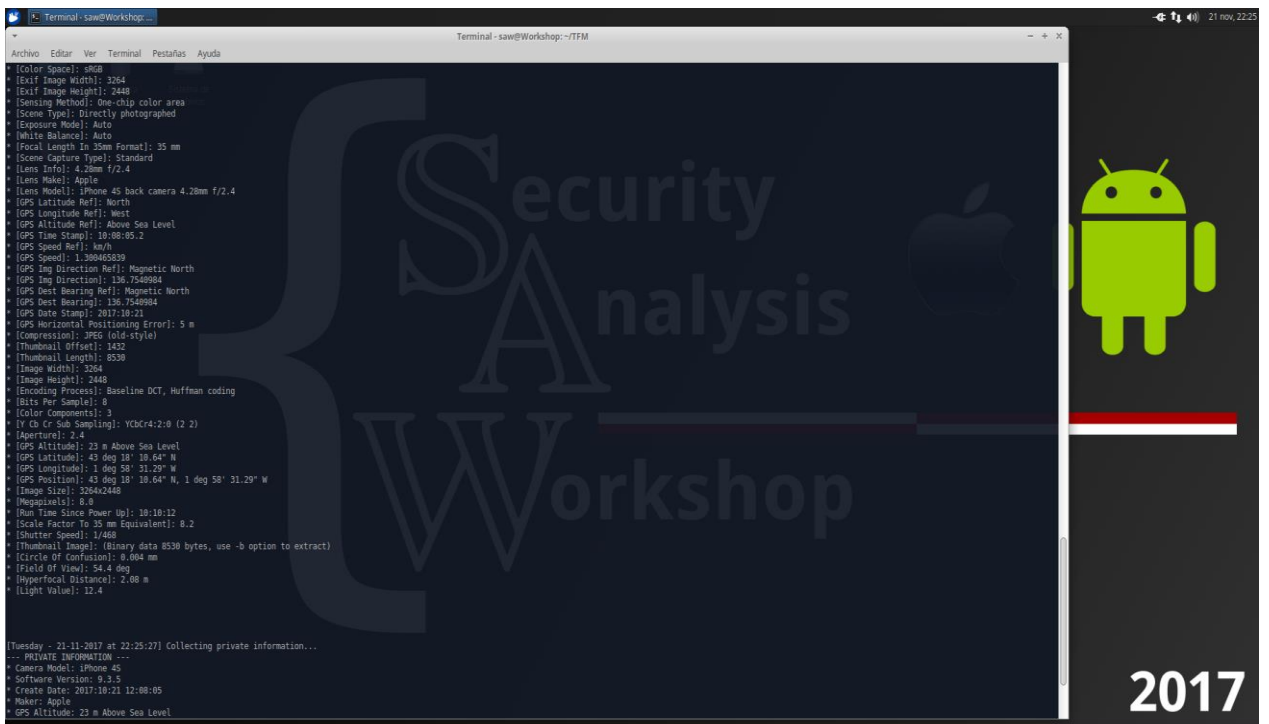


Figure 5.9. Metadata extractor collecting private information

5.5. Metrics

In order to evaluate the effectiveness of this platform, based on the objectives described in previous sections, we have created a series of metrics or indicators that will allow us to evaluate whether these have been correctly fulfilled.

Depending on the scope to which these metrics are directed, we can divide them into:

- Metrics for the User
- Metrics for Professionals
- Metrics for Learning

a) Metrics for the user

- Is the user able to understand the concepts specified throughout the work?
- In addition to understanding them, is he/she able to assimilate them?
- Is the supplied platform useful to apply the concepts learned by the user?
- What kind of technical mastery does the tool require? Is it affordable/accessible to users with little technical knowledge? Does it give them any kind of facilities?

b) Metrics for Professionals

- Does it allow them to automate tedious, costly procedures in an easy and comfortable way?
- Does it provide a clear and concise view of the information provided to the user?
- Is it multi-platform?
- Are open-source tools? Can a user with technical knowledge modify, or alter the operation of the platform tools according to their use?

c) Metrics for learning

- Are the contents of security and forensic analysis applicable from the platform: "Security Analysis Workshop"?
- Can this tool be adapted to the different needs of users? Is it interactive?
- Can it be applied to real cases?
- Is it dynamic?

5.5.1. Limitations found

Due to lack of time and other external factors, this project could not be tested with the general public in order to test its effectiveness. However, and due to the metrics that have been established during the execution of the project, and to the fact that these have been met at the time of its implementation, it is considered that the main objectives have been met, and therefore the work has been carried out successfully.

5.6. Real Cases of Security Analysis

Throughout this section, a series of real cases of security analysis on mobile platforms will be cited, indicating the tools that have been used to develop them, as well as the procedures carried out and the results or conclusions that have been drawn during their study. Firstly, an analysis of the iOS application (DVIA - Damn Vulnerable iOS App) will be detailed, which is an app for Apple's Mobile Operating System that has been purposely designed, with a series of bugs and vulnerabilities that serve as a learning tool for all those researchers who want to start in the world of analysis on mobile platforms. Then, an issue will be evaluated in one of the most used messaging apps in the world (WhatsApp), and as a result of bad decisions or negligence during its development, the door was left open for a long time to an exfiltration of personal data by users who used it. Finally, the problem of the danger of metadata associated with the majority of digital resources that populate the Internet will be explored in depth, and how their collection and subsequent analysis can enable personal information to be obtained from the user who created them, including unambiguously identifying them.

5.6.1. Security Analysis over iOS app: DVIA

In this section, we are going to perform a security analysis over: DVIA [146] (Damn Vulnerable iOS Application), in order to show how a security analyst should carry out any research about possible vulnerabilities in this mobile platform. DVIA is a special application because it has been created to provide a learning platform for researchers, students, and everybody who wants to know how the security and privacy architecture is built in iOS. This application has been developed intentionally with errors and other common issues that it works like example of how not to develop an iOS application, dividing into sections the app to teach the user about: how to detect whether or not the jailbreak is enabled, if communications through Internet are secure enough, and even how to store user credentials, ensuring that these are properly encrypted.

The github repository where the iOS application is stored is: <https://github.com/prateek147/DVIA> . The first thing a security analyst needs to do is: cloning the repository in order to have the source code of the iOS app available to inspect it, in addition to its corresponding ipa file which is the common format for application in iOS ecosystem. Once the information in this repository has been downloaded correctly, the first thing, if the analyst is using a Linux distribution (for instance the distro provided in this master thesis: "Security Analysis Workshop"), then he/she must go to the main folder of the github repository which is: DVIA. To do that, It is only necessary to type on the terminal:

```
- cd DVIA (change directory from the current folder to DVIA)
```

After going into this folder, we will see an ipa file called: `DamnVulnerableiOSApp.ipa` which is just a compressed zip file with another extension. Therefore, in order to uncompress its content, firstly it will be needed to change the extension of the file. Then, the security analyst will invoke this command-line:


```
- mv DamnVulnerableiOSApp.ipa DamnVulnerableiOSApp.zip (This sentence will change the original name for a new one with extension .zip).
```

As we have a zip file, to see how many files are compressed within it, and start to inspect them, we will execute this command:

```
- unzip DamnVulnerableiOSApp.zip (which uncompress the content of the zip file in the same working directory where the security analyst is).
```

Inside the zip there are a list of different files, as can be seen in the picture below, but the most important is one named: `DamnVulnerableiOSApp` without extension. This is the binary file, the code compiled for the developer of the application, and it is where the security researcher must search in order to comprehend the behavior of the mobile application to analyze. One way to achieve this, it is researching over the source code of the application, but this part of the application is not available, so in order to perform a static analysis through the code of the app, the analyst must use a disassembler for disassembling the binary cited previously. The utility chosen for this stage of the analysis will be: Hopper which is a disassembler, debugger and decoder introduced in previous chapters, that is only available on macOS platforms, but it is the best choice to work with binary files compiled in the ecosystem created by Apple. This step in the project could be an obstacle for certain researchers, because if they do not own a Mac computer, then they will not be able to carry out this important part of the analysis, or maybe they will have more problems, because other software utilities like: volatility are not ready enough to work on Apple platform.

However, if the security analyst owns a Mac computer and he/she downloads Hopper, he/she only needs to run this utility and when is loaded a window with a side bar will show up on the screen. In order to start disassembling the: "Damn Vulnerable iOS App", it is only needed to click on option File in toolbar menu, and then over "Read binary executable from...". After doing that, the researcher must select the path where the app to analyze is stored, in this case: DVIA, and after clicking on the binary file: `DamnVulnerableiOSApp`, Hopper will ask the user to select between two different architectures: 32bits or 64 bits. We will choose 64 bits, and then ARMv64 because this application is ready for supporting the architecture of the latest iOS mobile devices. Finally, it will start a procedure in which the mobile application will be disassembled, displaying its assembly code on the center part of the Hopper.

Once, we reach this point, it is necessary to highlight two different sections and features in Hopper, that we will use frequently as long as we deal with this security analysis. This two characteristics are: in first place the section in the left sidebar which allows the security analyst to find strings, and labels in the source code of the application, and then this button  in the upper-right side of the window called: pseudo-code, which creates as its name suggests a pseudo-code version of the source code written by the developer. Therefore, the code will not be exactly as it was coded but it will be a great approximation of the original source code, providing to the security researcher a

clearer view of the application target than the assembly version, which in many cases when the mobile application is bigger than in this case, it will be really complicated to understand deeply the purpose of the software decoded by Hopper [147].

To teach how to reproduce a lookup in the assembly code displayed by the tool, the researcher can search this common delegate in iOS application: `willFinishLaunchingWithOptions`, which show up the results to the user. With this simple step, we will accomplish two interesting things. Firstly, we learned how to search a method, delegate, or whatever string inside the mobile application disassembled, and in addition we found out the "Damn Vulnerable iOS App" is programmed in: Objective-C, instead of the current trend of coding in the new programming language introduced by Apple a few years ago: Swift.

The content of the compress file has two different folders called: Payload (it includes the .app file which contains among others the binary file to be disassembled), and Symbols (includes symbolic information created during the execution of the mobile application for creating reports in terms of logging). In order to see the files inside the .app file, it is necessary to click on right button and then: "Show package contents". After doing this, we will see a binary file which will be disassembled in the next steps of this analysis.

In addition to the application's own resources, such as images, videos, and so on, it should be noted that the content of any app package usually contains:

- Info.plist** - configuration information (such as the manifest file in Android).
- Pkginfo** - indicates the packet type, plus a set of bytes to identify the application.
- CodeSignature**: the signature itself of the mobile application.

Next, and in order to more conveniently analyze the source code of the application and run it in a terminal software simulator, if we do not have an iOS mobile device to do so, we will import the `DamnVulnerableiOSApp` project into Apple's official IDE: Xcode. To do this, double-click on the file called: `DamnVulnerableiOSApp.xcodeproj`, which will proceed to the import process of all source code, and application resources into Xcode. The typical interface of the official iOS IDE and applications for Mac, consists of a sidebar on the right side from which we can select the project to work on, as well as the set of resources of the same, and a central visualization area, where the source code of each class, methods, and delegates created during the process of developing a mobile app for iOS.

From Xcode, if the researcher selects in the upper toolbar the device on which to perform the emulation: (iPhone or iPad), and later gives to the "Play" button, the emulation will take place in a virtual environment of the DVIA app, imitating its behavior as if it were installed in a conventional physical device.

If you want to investigate how this mobile application is stored in the device emulator, you should simply access the macOS Finder and then go to the following path: `/Users/user/Library/Developer/CoreSimulator`.

As can be seen in the figure below, each app is assigned a random identifier number, but not the app's proper name, and a sandbox environment is generated that limits the scope of each one of them, as previously explained in this project. Once located where the application has been compiled, it could be loaded as it was previously done in Hopper, in order to proceed with its disassembling, and be able to evaluate certain characteristics associated with it.

As with Android, its info.plist configuration file is one of the most important because it has configuration information from the app itself, and can greatly help the research process, as it lists the most important components that are used by it.

The info.plist file saved in the root folder of the iOS application. Among many of the components that are identified, priority should always be given to those that have to do with communications via the Internet, which is why we are attracted by the use of the URL tag.

Therefore, we decided to check how the URLs are handled by the application, and for that we go to Hopper and write: URL. We found that there is an AppDelegate method, which serves to manage URLs in the app, and that is called: openURL. If we look at some of the strings contained, we notice the use of "phone" or "Calling without validation", which is not a very characteristic behavior when going to consult web pages on the Internet. It is therefore, that we keep this in quarantine, to consult at the end, but everything seems to indicate that it is a strange behavior, which may have been created to violate the security and privacy of the user of the application.

As mobile applications become larger and more complex, it becomes increasingly difficult to consult and perform a conclusive security investigation, since it is very difficult to determine what elements each app contains and the relationships between them. To solve this problem, the disassembler Hopper from its left sidebar allows you to search for "Labels".

To give an example, if the researcher wanted all the drivers of the mobile application, which are precisely Controllers inherited from the parent class: ViewController, he would only have to put ViewController in the text box, and then Hopper would return a drop-down list of the set of drivers defined by the developer of the app to be analyzed.

After this analysis of DVIA's structure, the next step will be to evaluate the information stored in it, since many times the main privacy problems in mobile applications can be caused by a bad configuration of the security mechanisms provided by the Operating System to store information, or by an exfiltration of data taking advantage of a vulnerability present in the mobile system.

In the case of iOS, the main storage media will be: UserDefaults (the equivalent of Android's SharedPreferences), plist files, CoreData, and SQLite3 databases.

To analyze UserDefaults, we must look for the method: standardUserDefaults, which is the one that makes calls to access the content of this important section of Apple's system. Once that string is searched, we will be returned a set of elements in the app that make use of the previously mentioned method.

If we access the pseudo-code of the method: `InsecureDataStorageVulnVC`, we can verify that a value called: "DemoValue" is stored in `NSUserDefaults`. It seems in this case, something harmless, that could have been left as a legacy of the early stages of mobile app development.

In the case of plist files, they are usually created by invoking the `writeToFile` method, `NSDictionary` or `NSMutableDictionary` objects, so we will go to the Strings search bar on the left sidebar and type: `writeToFile`. This will return a list of methods that meet that pattern. If you right click on: `writeToFile:automatically`, and then select `References to highlighted...`, a list of those elements will be returned, where this method is executed.

Within the returned list, it is necessary to look for suspicious names, or that can call the attention of the investigator in security due to the implementation of the same one. In this case, it is curious to find a method called: `InsecureDataStorageVulnVC`, that if we worry about looking at the source code written in Objective-C, we see that it saves the username, and the user's password in plain text, in the file `userInfo.plist`.

If we analyze the following discordant element later: `FlurryHTTPResponse`, we can see that the headers, the body, and the status code of calls made to a web page that we cannot identify are stored.

Next, if we analyze: `downloadAssetForTransaction`, we can see that it is stored in the mobile device, a certain file whose origin is unknown, but we only know that it has been downloaded via the Internet.

If another possible method of storing information is used: `CoreData`, the `NSManagedObjectContext` method should be searched in strings. If we look again at: `InsecureDataStorageVulnVC`, we can see that various data such as: name, telephone, or user's e-mail, will be stored in this mechanism of system storage, through calls to the methods: `nameTextField`, `emailTextField`, or `phoneTextField`.

Because of the ability to be permanently connected to the network of today's smartphones and tablets, another potential entry point for security threats on mobile devices is through the network connections established between them. In order to analyze them, the security analyst should focus on searching for the following strings, which have to do with common elements in all Internet communication: `SSL`, `SSLPinning`, `NSURLRequest`, `NSURLConnection`, and more specifically the methods used to initialize these classes such as `initWithRequest`, or `URLWithString`.

If we focus on `URLWithString`, searching for it will return a list of methods that invoke it in such a way:

Since at the present time how information is transmitted via the web medium (whether through non-encrypted protocol: `HTTP`, or encryption: `HTTPS`) is one of the points that determine to a greater extent the degree of security of the platform, we will focus on analyzing the method: `TransportLayerProtectionVC`, which will return us if we dump its pseudo-code, three interesting methods: `sendOverHTTP`, `tapped`, `send`. Because non-encrypted means may be used for the transmission of information, this will be another possible point to highlight as conflicting, as it could jeopardize user privacy.

Finally, another of the most common storage media, which usually store information of various kinds in the system, and which should be analyzed in any security analysis of mobile platforms, are the following: system logs.

In the case of the iOS platform, the NSLog method is usually used to create logs of the various actions to be carried out. If we search for the NSLog string in the disassembler Hopper, it will return the set of methods that refer to log calls within the system. Because it is a very common function, the investigator's own intuition can help you determine which methods to inspect more closely because they are suspicious.

As in many of the sections of this section devoted to the analysis of an iOS vulnerable application, the method to be investigated is: InsecureDataStorageVulnVC. If we look at the body of the same one, an exception can be seen that could be generated when saving user information through the system logs.

In short, the main vulnerabilities, or risks that have been seen during the analysis of this iOS application are: the ability to make calls in a covert manner without the app user noticing, web communications through non-encrypted media using the HTTP protocol, exfiltration of sensitive information associated with the user stored in various storage mechanisms of the platform such as: NSUserDefaults, CoreData or NSLog for all these facts, it is demonstrated once again the ignorance that the user may have about the actions carried out by the applications of your system, and as the conjunction of the actions carried out by them, can put at grave risk the security and privacy of users who decide to install, and use that vulnerable app on your system.

5.6.2. Metadata Analysis over Multimedia Assets

As has been described in previous sections of this Master thesis on mobile platforms: iOS and Android, the information that can be extracted from multimedia media that can be captured thanks to the capabilities offered by the new smartphones, and tablets, make it possible that if you don't take special care, the data that an attacker can obtain about the user can identify it, often uniquely, causing not only a violation of their privacy, but also a violation of their privacy.

To demonstrate this fact, the following assumption will be made:

"In a group of friends, it is decided to share the photos that have been taken during the holidays of each one, in a shared folder of Dropbox. Given a specific set of photographs, could it be determined where these photographs have been taken, in addition to who has taken them?"

The answer, as will be seen throughout the development of this chapter, is yes. Not only will we be able to determine where the photographs have been taken, but we will also determine the device that has taken them, and therefore we will be able to relate it to the holder of this mobile device.

In order to perform this procedure, we will first access the Linux distribution based on Ubuntu, and presented for this thesis called: "Security Analysis Workshop", and once booted we will open a terminal of commands, clicking on the menu at the top left, and clicking on: Terminal of Commands. Once we have a shell, we will move to the directory

where we have downloaded the photos for analysis (in this case, in Images/EXIF Analysis).

cd ~/Images/EXIFAnalysis

Inside that directory, if we list its contents, by means of the command `ls`, we can see that there are two photographs of a cloudy landscape, called `DCIM_1.jpg`, and `DCIM_2.jpg`. Thanks to one of the tools installed within this Linux distribution, and which has been presented in previous chapters of this document (`exiftool`), we will be able to list the set of metadata associated with both photographs, in order to discover if the information provided by them, allows us to obtain some kind of indication about what and who has made them.

In order to obtain a list of these metadata, simply run the following command:

- **exiftool DCIM_1.jpg** (equivalent to `DCIM_2.jpg` and the rest of pictures), which shows us all the data associated with the last medium as a parameter, which is sometimes not the most appropriate, since sometimes the information is so vast, that it becomes quite complex, to see at a glance what information shown is useful, and which is not.

Next, there is a Figure 5.10 below with the result of this command, which displays all the metadata associated to a certain picture.



```
1. bash
Last login: Mon Oct 30 21:16:01 on ttys001
MacBook-Pro-de-Alex:~ alejandra$ cd Desktop/EXIFAnalysis/
MacBook-Pro-de-Alex:EXIFAnalysis alejandra$ exiftool DCIM_1.JPG
ExifTool Version Number      : 10.64
File Name                    : DCIM_1.JPG
Directory                   : .
File Size                    : 2.4 MB
File Modification Date/Time  : 2017:10:21 12:07:56+02:00
File Access Date/Time       : 2017:10:30 20:24:22+01:00
File Inode Change Date/Time  : 2017:10:30 19:53:35+01:00
File Permissions            : rw-----
File Type                   : JPEG
File Type Extension         : jpg
MIME Type                   : image/jpeg
Exif Byte Order             : Big-endian (Motorola, MM)
Make                       : Apple
Camera Model Name          : iPhone4S
Orientation                 : Rotate 90 CW
X Resolution                : 72
Y Resolution                : 72
Resolution Unit             : inches
Software                   : 9.3.5
```

Figure 5.10. Exiftool extracting metadata from a certain picture

As we can see, there are a lot of information to research. Fortunately, the `exiftool` tool contains a series of filters, which will allow us to measure what information to show to the security analyst, and allow him to divide in detail the investigation that is being processed.

From now on, we are going to introduce a set of different commands, in order to show the main set of parameters to call, for displaying the more useful information which can help the security analyst to perform an analysis with enough guarantees to prove how multimedia resources may compromise the security and privacy of the users.

In Figure 5.11, we can see a list of command sentences followed by its output which is going to be explained step by step.

```
MacBook-Pro-de-Alex:EXIFAnalysis alejandro$ exiftool -Model DCIM_1.JPG
Camera Model Name      : iPhone 4S
MacBook-Pro-de-Alex:EXIFAnalysis alejandro$ exiftool -Software DCIM_1.JPG
Software               : 9.3.5
MacBook-Pro-de-Alex:EXIFAnalysis alejandro$ exiftool -CreateDate DCIM_1.JPG
Create Date           : 2017:10:21 12:07:56
MacBook-Pro-de-Alex:EXIFAnalysis alejandro$ exiftool -Make DCIM_1.JPG
Make                  : Apple
MacBook-Pro-de-Alex:EXIFAnalysis alejandro$ exiftool -gpsposition DCIM_1.JPG
GPS Position          : 43 deg 18' 10.91" N, 1 deg 58' 31.65" W
MacBook-Pro-de-Alex:EXIFAnalysis alejandro$ exiftool -gpsposition DCIM_1.JPG | sed "s/deg/°/g"
GPS Position          : 43 ° 18' 10.91" N, 1 ° 58' 31.65" W
MacBook-Pro-de-Alex:EXIFAnalysis alejandro$ exiftool -gpsaltitude DCIM_1.JPG
GPS Altitude          : 28 m Below Sea Level
MacBook-Pro-de-Alex:EXIFAnalysis alejandro$ exiftool -Compression DCIM_1.JPG
Compression           : JPEG (old-style)
MacBook-Pro-de-Alex:EXIFAnalysis alejandro$
```

Figure 5.11. List of exiftool parameters

One of the first steps, which any security researcher could take to determine the source of the resource, is to check the camera model of the mobile device. This could help as a first step to determine which mobile device is smartphone or tablet, has taken the photograph, and thus be able to narrow down a little more the search among potential people who may have been the owners of this multimedia resource.

In order to display information about the camera that took the picture, we should run the following command from the terminal:

-exiftool -Model DCIM_1. Jpg

As we can see, it is the camera from an iPhone 4S. This command not only show the camera type but also allow us to determine the model of the smartphone. Now, it is easier to know the source of this picture, because from now on, we can discard Android mobile devices.

The next command: `exiftool -Software DCIM_1.JPG` is for determining the version of the Operating System installed in the mobile device. In this case, we can see is the version: 9.3.5, which is an obsolete iOS version, which is normal because an iPhone 4S is considered a vintage generation by Apple. In addition, this important data may help the analyst to exploit some vulnerabilities of this old version which have not been patched by Apple, due in general the latest patches are meant to be part of the newest versions of this mobile system.

To extract when the photograph was taken, it is only necessary to invoke: `exiftool -CreateDate DCIM_1.jpg`, which displays the date when this picture was created is: October 21st, 2017 at 12:07:56p.m. Therefore, it was taken at morning, as we can see in the picture attached previously.

There is no need to search this data, because the model of the smartphone was discovered in the first step of this analysis, but if the security researcher launches: `exiftool -Make DCIM_1.jpg`, Apple value is showed up, which is completely normal because we are inspecting a picture taken by the official smartphone of Apple: iPhone.

When the releasing of the new version of iOS (iOS 11), Apple launched a new type of compression system called: HEIF. We discovered in the software stage that this picture was taken from iOS 9.3.5, therefore this extension will not be available, but in order to show all the different possibilities this powerful tool provides, if the security analyst invokes: `exiftool -Compression DCIM_1.JPG`, it will be displayed: the common in this

version: JPEG format which is one of the most popular picture format with loss, available today.

Likely, the most important and maybe the part which pay attention, because its ability to violate the privacy of the user, it is in which is shown the GPS coordinates associated to the photograph taken by some user. It is necessary to highlight that in order to have this important characteristic; the location feature must be enabled in the smartphone or tablet. This feature, due to export this ability in cars, the “Find my iPhone” app, or because the use of smartphones in sport activities for monitoring the heart rate and different aspects of the physical condition of the user, it is really common of keeping enabled it. It is on, then the latitude and longitude associated to the place where the picture was taken will be embedded within this multimedia resource.

These important parameters, for exiftool can be executed in its single version:

```
-exiftool -gpslongitude DCIM_1.JPG
```

```
-exiftool -gpslatitude DCIM_1.JPG
```

But the most useful way of invoking this sentence is through a mix version of the previous command-line sentences.

```
-exiftool -gpsposition DCIM_1.JPG
```

When the security analyst run this command, the latitude and longitude associated to this asset is displayed on the screen. In this case, the result obtained is:

```
43 deg 18' 10.91" N, 1 deg 58' 31.65"W
```

Besides, there is another command which provides the researcher with additional information to add to the previous one, and that is: `exiftool -gpsaltitude DCIM_1.JPG` that display: 28 m Below Sea Level. Maybe a seaside city?

With all these coordinates, the security analyst can check through the different online map solutions available, where is this place, but first of all, it is needed to format the output received by the previous command sentences. To do that, the analyst can invoke this instruction as follows:

```
-exiftool -gpsposition DCIM_1.JPG | sed 's/deg/°/g'
```

After executing it, the output now is:

```
43° 18' 10.91" N, 1° 58' 31.65"W
```

Now, it is time to browse to an online map platform, for instance: <https://www.google.com/maps>, and type in its search textbox the coordinates obtained:



Figure 5.12. Searching a place by its GPS coordinates

As we can see, this picture was taken in: San Sebastián, Guipúzcoa (Spain) in Figure 24, nearby the Anoeta Stadium. This metadata allows the security analyst to know where was the city that appears in the picture, and as we suspect is a seaside city in North Spain. Despite this prove the power of this interesting tools, it also shows how vulnerable users are, because if some attacker takes this multimedia resource and he/she has proficiency to perform this procedure, he/she will be able to know certain personal information about the owner which means a severe privacy risk.

Other useful fields that must be taken into account are those which delivers information about the person who takes the photograph. Not every system shows this data, but it does, invoking one of these commands, the security researcher will be able to unveil the name of the person who owns the mobile device.

```
-exiftool -Author pic_name
```

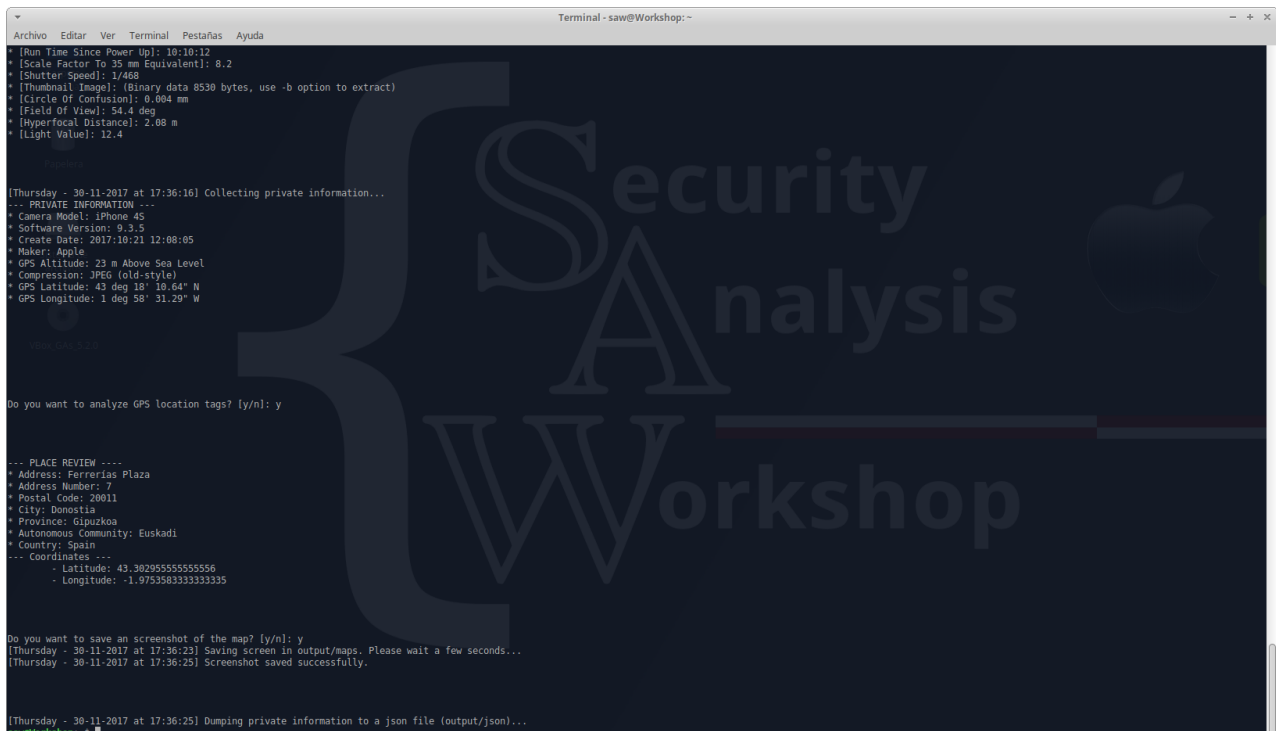
```
-exiftool -Creator pic_name
```

Due to its important privacy hole, which allows malicious users or other kinds to discover certain private information about the user of the mobile dispositive, it recommended to execute this command: `exiftool -all= pic_name`, once a picture is taken, in order to remove the metadata fields which, identify the owner of the device, and thus preserving his/her privacy.

Finally, it is necessary to stand out that this procedure can be simplified, using the tool developed for this purpose during this work, and called: `metadata_extractor`. The steps to run this tool, are as follows:

- Go to the command prompt, and invoke the `metadata_extractor` script.
- The user will then be asked to choose the set of images, or other multimedia resources to evaluate, and for them to be detected by the tool, they must be located in the input folder of the home directory of the system (`home/saw/Scripts/input`).
- After selecting one of the resources, the analysis will begin, listing all the information obtained by screen, and indicating if any associated sensitive information has been found.
- If found, it will be displayed on the screen, and in case they are GPS coordinates, the location will be geolocated, showing a map to the user of the location found, besides saving this capture in the output directory inside Scripts folder.
- Finally, the user will also save a summary of all the information obtained in previous steps in a JSON format, for later analysis or import to other tools.

A complete tool execution is shown in Figure 5.13, and 5.14.



```
Terminal - saw@Workshop:~
[Run Time Since Power Up]: 10:10:12
* [Scale Factor To 35 mm Equivalent]: 0.2
* [Shutter Speed]: 1/400
* [Thumbnail Image]: (Binary data 8530 bytes, use -b option to extract)
* [Circle Of Confusion]: 0.004 mm
* [Field Of View]: 54.4 deg
* [Hyperfocal Distance]: 2.08 m
* [Light Value]: 12.4

[Thursday - 30-11-2017 at 17:36:16] Collecting private information...
--- PRIVATE INFORMATION ---
* Camera Model: iPhone 4s
* Software Version: 9.3.5
* Create Date: 2017:10:21 12:08:05
* Maker: Apple
* GPS Altitudes: 23 m Above Sea Level
* Compression: JPEG (old-style)
* GPS Latitude: 43 deg 18' 10.64" N
* GPS Longitude: 1 deg 58' 31.29" W

Do you want to analyze GPS location tags? [y/n]: y

--- PLACE REVIEW ---
* Address: Ferrerías Plaza
* Address Number: 7
* Postal Code: 20011
* City: Donostia
* Province: Gipuzkoa
* Autonomous Community: Euskadi
* Country: Spain
--- Coordinates ---
* Latitude: 43.30295555555556
* Longitude: -1.9753583333333335

Do you want to save an screenshot of the map? [y/n]: y
[Thursday - 30-11-2017 at 17:36:23] Saving screen in output/maps. Please wait a few seconds...
[Thursday - 30-11-2017 at 17:36:25] Screenshot saved successfully.

[Thursday - 30-11-2017 at 17:36:25] Dumping private information to a json file (output/json)...
saw@Workshop:~$
```

Figure 5.13 Execution of `metadata_extractor`

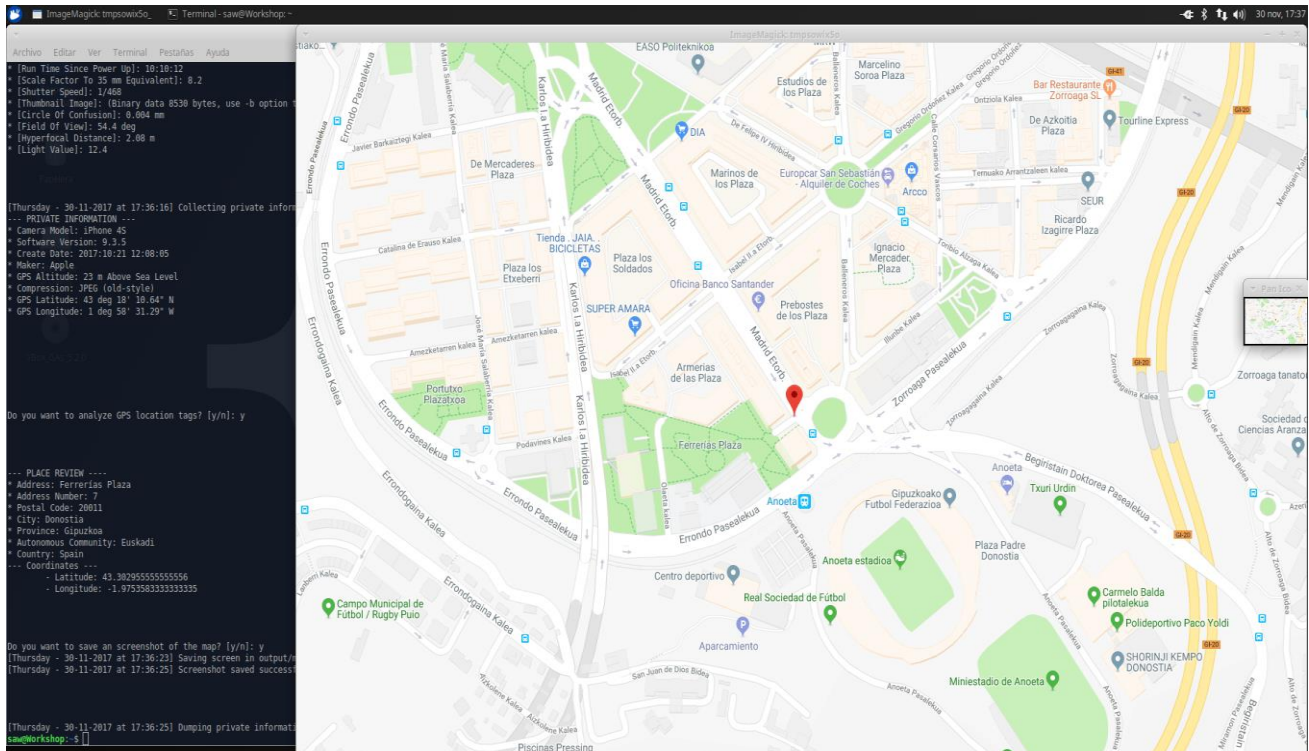


Figure 5.14 Results of metadata_extractor

In addition to the above, metadata_extractor has two additional options:

- Anonymizer: invoke metadata_extractor -a, in order to eliminate all the sensitive information of a determined multimedia resource.
- Faker: by calling metadata_extractor -f, random values associated with this resource can be generated to falsify it, so that no real data can be extracted that could compromise the user. Options can be configured in the file: conf/fake.cfg.

5.6.3. Downloading automatically applications

In order to obtain the applications that can be analyzed by other tools in “Software Analysis Workshop” such as for instance: manifest_interpreter, applications should be downloaded and manually incorporated one by one into the computer of the security analyst. To automate this process, the script called: apk_downloader has been created. This tool allows you to download applications from the Android platform in an automated way, from the apkmirror repository, which as advantages compared to the official Google store, offers a history of versions of the application a user want to download, apart from an added value, because the user could get obsolete versions, or with some security issue in order to learn the main errors and risks that are found on a mobile platform.

To run this utility included in "Software Analysis Workshop", the following will be performed:

- From a terminal, run the command: apk_downloader.

- The user will be asked to enter the name of the application he/she wants to download.
- If the app is available in the repository, it will proceed to display the set of available versions of it.
- After selecting one of the available versions, the user will be asked to enter the architecture or build version associated with that mobile application.
- When selected, it will prompt the user to enter the name with which he/she wants the app to be saved in the system (If none is entered, a default name will be used).
- Finally, the apk file is downloaded to the input folder of the Scripts directory, for use by other tools.

In Figure 5.15, a complete execution of a download of an Android application, through this script, is shown.

```
saw@workshop:~$ apk_downloader
APK DOWNLOADER
Introduce the app name you want to download: instagram
App found. 10 results.

--- APPS LIST ---
1. Instagram 25.0.0.11.136 (82293)
2. Instagram 23.0.0.14.135 (80125)
3. Instagram 25.0.0.1.136 (81947)
4. Instagram 24.0.0.12.201 (81140)
5. Instagram 24.0.0.11.201 (80851)
6. Instagram 24.0.0.8.201 (80489)
7. Instagram 24.0.0.2.201 (80132)
8. Instagram 23.0.0.12.135 (79757)
9. Instagram 23.0.0.6.135 (79369)
10. Instagram 22.0.0.17.68 (78982)
Introduce the app version, you want to download: 1
[*]. Instagram 25.0.0.11.136 (82293)
[ ]. Instagram 23.0.0.14.135 (80125)
[ ]. Instagram 25.0.0.1.136 (81947)
[ ]. Instagram 24.0.0.12.201 (81140)
[ ]. Instagram 24.0.0.11.201 (80851)
[ ]. Instagram 24.0.0.8.201 (80489)
[ ]. Instagram 24.0.0.2.201 (80132)
[ ]. Instagram 23.0.0.12.135 (79757)
[ ]. Instagram 23.0.0.6.135 (79369)
[ ]. Instagram 22.0.0.17.68 (78982)

Searching available architectures...
--- ARCHITECTURES LIST ---
1. 25.0.0.11.136
Introduce the app version, you want to download: 1
[*]. 25.0.0.11.136

Choose a name to save the file. (By omission: instagram):
* apk: instagram.apk downloaded successfully.
```

Figure 5.15. Downloading an app via apk_downloader

5.6.4. Permission classification from a Manifest file

In the analysis of an Android application, one of the first steps to be carried out is the extraction of the permissions requested by the app, in order to see its impact when is used by the user.

As it can be seen in previous sections of this document ("Analysis of Permission System"), in order to know the list of permissions in an Android application, the user should look at the AndroidManifest.xml file, associated with each application. One of the main problems with this file is that although it is a structured XML file, many times when it is very large, or when it is evaluated by people with little knowledge of the platform, it is difficult to discern the parts within it.

This is why the program called: `manifest_interpreter`, developed especially for this work, evaluates, classifies and displays the information of this type of files in an organized and orderly way to the user.

To proceed with its implementation, the following steps will be taken:

- From a terminal, it is necessary to execute: `manifest_interpreter`.
- The program requests a selection of the available apps in the system. Android apps with an apk extension must be downloaded and saved in the input file of the Scripts folder, in order to be detected by the tool.
- After selecting a given application, the program will automatically decompile it, collect all the available information and display it in a structured way by screen, highlighting the main parts of the manifest file, and classifying according to its dangerousness the set of permissions associated with the mobile application under study.
- Finally, and automatically, it will save all the information collected in JSON format, in the output folder of the Scripts directory, so that this file can be exported by other types of tools that support it.

Next, a series of screenshots show the complete tool execution process. In the Figure 28, it can be seen the initial running process of `manifest_interpreter`, followed by the selection procedure of the application to be analyzed.


```
Terminal - saw@Workshop:~  
Archivo Editar Ver Terminal Pestañas Ayuda  
saw@Workshop:~$ manifest_interpreter  
MANIFEST INTERPRETER  
Lista de apks, encontradas en el directorio 'input':  
-----  
1) facebook.apk  
2) instagram.apk  
3) WhatsApp Messenger v2.17.395.apkpure.com.apk  
Selecciona la app deseada, indicando su número:  
2  
APK Seleccionada:  
[ ] facebook.apk  
[x] instagram.apk  
[ ] WhatsApp Messenger v2.17.395.apkpure.com.apk  
Comenzando el proceso de decompilación...  
Proceso de decompilación realizado con éxito...  
----- Manifest Overview -----  
* Package Name: com.instagram.android  
* Schema: http://schemas.android.com/apk/res/android  
* Android Version Name:  
* Install Location: auto  
  
--- Application ---  
* Icon: @mipmap/icon  
* Name: com.instagram.app.InstagramAppShell  
* Label: @string/app_name  
  
--- Metadata ---  
* Name: com.instagram.android.channel  
* Value: playstore  
  
* Name: com.facebook.rscmp  
* Value: true  
  
* Name: com.facebook.build_rule  
* Value: instagram_multi_bytecode_beta_dextr_comp_armv7_release_fbsign  
  
* Name: com.facebook.package_type  
* Value: release  
  
* Name: com.facebook.build_time  
* Value: 1510348618000L  
  
* Name: com.facebook.versioncontrol.branch  
* Value: master  
  
* Name: com.facebook.versioncontrol.revision  
* Value: MASTER  
  
* Name: com.facebook.rti.fbns.FB_SHARED_VERSION  
* Value: 4.0
```

Figure 5.16. Initial execution of manifest_interpreter

In Figure 5.17, you can see how the tool collects and displays the information collected from the application under study, in this case: Instagram.

```
Terminal - saw@Workshop:~  
Archivo Editar Ver Terminal Pestañas Ayuda  
+ [ DANGEROUS ] Allows an app to access precise location.  
  
* Name: android.permission.RECORD_AUDIO  
+ [ DANGEROUS ] Allows an application to record audio.  
  
* Name: android.permission.MODIFY_AUDIO_SETTINGS  
+ [ NEUTRAL ] Allows an application to modify global audio settings.  
  
* Name: android.permission.WRITE_EXTERNAL_STORAGE  
+ [ DANGEROUS ] Allows an application to write to external storage.  
  
* Name: android.permission.READ_PHONE_STATE  
+ [ DANGEROUS ] Allows read only access to phone state, including the phone number of the device, current cellular network information, the status of any ongoing calls, and a list of any PhoneAccounts registered on the device.  
  
* Name: android.permission.RECEIVE_SMS  
+ [ DANGEROUS ] Allows an application to receive SMS messages.  
  
* Name: com.android.launcher.permission.INSTALL_SHORTCUT  
+ [ DANGEROUS ] Allows an application to install a shortcut in Launcher.  
  
* Name: com.android.launcher.permission.UNINSTALL_SHORTCUT  
+ [ DEPRECATED ] This permission is no longer supported.  
  
* Name: .permission.C2D_MESSAGE  
+ [ NEUTRAL ] Allows a server to send small amounts of data to the device.  
  
* Name: .permission.RECEIVE_ADM_MESSAGE  
+ [ NEUTRAL ] Allows an application to protect its ADM (Amazon Device Messaging), against external interception.  
  
* Name: com.instagram.android.permission.LOGGED_IN_USERS_CONTENT  
+ [ UNKNOWN ] Customized permission. Description not available.  
  
* Name: com.instagram.direct.permission.LOGGED_IN_USERS_CONTENT  
+ [ UNKNOWN ] Customized permission. Description not available.  
  
* Name: com.google.android.c2dm.permission.RECEIVE  
+ [ UNKNOWN ] Customized permission. Description not available.  
  
* Name: com.amazon.device.messaging.permission.RECEIVE  
+ [ UNKNOWN ] Customized permission. Description not available.  
  
* Name: com.htc.launcher.permission.READ_SETTINGS  
+ [ UNKNOWN ] Customized permission. Description not available.  
  
* Name: com.htc.launcher.permission.UPDATE_SHORTCUT  
+ [ UNKNOWN ] Customized permission. Description not available.  
  
* Name: com.huawei.android.launcher.permission.CHANGE_BADGE  
+ [ UNKNOWN ] Customized permission. Description not available.  
  
* Name: com.sonyericsson.home.permission.BROADCAST_BADGE  
+ [ UNKNOWN ] Customized permission. Description not available.  
  
* Name: android.permission.UPDATE_APP_BADGE  
+ [ NEUTRAL ] Allows an app to update the app badge - in order to count new notifications
```

Figure 5.17. Results of manifest_interpreter

5.6.5. Updating apk tool easily

The tool called: apk_tool is one of the most powerful tools in the analysis sector on Android applications. For this reason, it has been installed and integrated into "Software Analysis Workshop", in order to be used for decompilation and package creation, which it is provided by this utility. Nevertheless, one of the main problems associated with apk_tool, is its installation is tedious, and may even be complicated for some users who do not have enough experience. Therefore, in order to speed up and facilitate this process, a script called: check_apktool has been included in: "Software Analysis Workwhop", which basically checks the version of the tool that is installed in the system, and if it is an obsolete version it automatically updates it, avoiding that the user has to download it, associate it with a certain wrapper, and finally add it to the PATH of the system environment variables, which is the standard procedure that must be performed.

It is necessary to point out that the execution of this service must be performed with administrator privileges (sudo check_apktool), because certain actions require root privileges in order to be performed.

In order to execute this tool, the following will be done:

- This command must be invoked from the terminal: sudo check_apktool
- If there is a newer version than the one installed, it will be updated.
- If not, a message will be shown to the user, indicating that the latest version is already installed.

The following screenshots show the three possible cases that can occur. First, it can be seen in Figure 5.18, as if it is not specified that it runs with root privileges, the app gives an error at runtime.



```
saw@workshop:~$ check_apktool
APKTOOL CHECKER
[30/11/2017 - 17:10:10] Downloading wrapper script...
[30/11/2017 - 17:10:11] Downloading latest version of apktool [2.3.0]...
[30/11/2017 - 17:10:18] Download process complete...
[30/11/2017 - 17:10:18] Fixing permissions for files...
[30/11/2017 - 17:10:18] Moving apktool to bin directory...
[Errno 2] No such file or directory: 'tmp/apktool' -> '/usr/local/bin/apktool'
* [ERROR] You need root permission to move files.
[30/11/2017 - 17:10:18] Cleaning temp directory...
```

Figure 5.18. check_apktool executed with normal user privileges

Next, in Figure 5.19, the normal execution is shown when a new version is available, which requires updating.

```
saw@Workshop:~/Scripts$ sudo check_apktool  
APKTOOL CHECKER  
[30/11/2017 - 17:11:38] Downloading wrapper script...  
[30/11/2017 - 17:11:38] Downloading latest version of apktool [2.3.0]...  
[30/11/2017 - 17:11:45] Download process complete...  
[30/11/2017 - 17:11:45] Fixing permissions for files...  
[30/11/2017 - 17:11:45] Moving apktool to bin directory...  
[30/11/2017 - 17:11:45] Updating configuration files...  
[30/11/2017 - 17:11:45] apktool is ready to use.
```

Figure 5.19. Check_apktool executed successfully

Finally, in Figure 5.20, it can be seen what the tool shows when the most current version of apktool is installed in the system.

```
saw@Workshop:~/Scripts$ check_apktool  
APKTOOL CHECKER  
* [30/11/2017 - 17:11:59] You have installed the current version of apktool. No updates available.
```

Figure 5.20. Latest apk tool version already installed

5.7. Limitations

Most of the tools that are installed in "Security Analysis Workshop", belong to those that allow security and privacy analysis on the Android platform. Tools for use with iOS have also been included, but the number of tools is lower than its competitor's, as iOS is more closed, and many utilities are available only for the Desktop Operating System: macOS, or require jailbreak on the mobile terminal.

We have not been able to prove it in a professional or learning environment, but the possible applications of each of the tools have been included, as well as metrics or indicators to help assess whether the objectives established when developing this Master's thesis have been achieved.

Furthermore, it has not been possible to test the platform with users in order to test its functionality in real life, so one of the next steps which should be done is to verify that the quality and compliance metrics described in previous sections are feasible.

5.8. Budget

There is no budget for this work, because its purpose to serve as Master Thesis.

6. Conclusions

Nowadays, mobile technologies are beginning to take on such prominence and importance that by the end of 2017, a growing trend is beginning to be noticed in which it can be seen that mobile devices are beginning to surpass their desktop counterparts.

Due to the portability and mobility provided by these devices, more and more developers choose these solutions to create their new applications, making mobile devices the main private data host for: credentials, instant messaging conversations, social networks, contact information, or even number of credit cards. This has caused that in recent years, the interests of malicious users and main security professionals, have been attracted to this emerging ecosystem, which day by day offers new possibilities and services to users of these platforms,

Because of this, new vulnerabilities, and various security issues are discovered every day, which help to improve system stability, but also serve if it is exploited by hackers, or other malicious users, to create procedures, and tools aimed at violating the security and privacy of the owners of affected devices.

This is why this project has placed special emphasis on showing a current state of the security landscape in today's two main mobile operating systems: iOS and Android, to give users an insight into how the situation is really going, and what hazards or risks they are exposed to. Since most of the time, these dangers are caused by ignorance, an attempt has been made to create a solid knowledge base on the security architecture of these mobile ecosystems, as we have considered that there is no better way to avoid a problem than to prevent it.

In addition, it has been jointly developed, a solution that serves not only as a practical workshop of what was presented during this document, but also as a distribution that combines the main existing mobile security analysis tools, as well as others specially created for this development, and whose main objective is to provide an agile and comfortable experience not only for professionals in the security sector, but also for users who want to discover what this community offers.

Therefore, with the achievement of this Final Master's Thesis, it is hoped to have been able to provide users with a complete and robust software platform, that allows them to perform security and privacy analysis in the mobile ecosystem, as well as provide global knowledge to users of these technologies, so that they are able to avoid and fight against the main risks that populate these platforms.

References

Bibliography

- [20] “iOS Security Guide”
Author/s: various authors.
Apple © 2017 (68 pages)
- [151] “iOS Application Security: The Definitive Guide for Hackers and Developers”
Author/s: David Thiel
No Starch Press © 2016 (297 pages)
- [152] “The Mobile Application - Hacker’s Playbook”
Author/s: Dominic Chell, Tyrone Erasmus, Shaun Colley & Ollie Whitehouse
John Wiley & Sons © 2015 (535 pages)

Papers

- [153] “Emerging Security Threats for Mobile Platforms”
Author/s: G. Delac, M. Silic and J. Krolo
Faculty of Electrical Engineering and Computing, University of Zagreb, Croatia, 2011
- [154] “iOS Security and Privacy: Authentication Methods, Permissions, and Potential Pitfalls with Touch ID”
Author/s: Stephen J. Tipton, Daniel J. White II, Christopher Sershon, and Young B. Choi
Virginia Beach, USA, 2014
- [155] “Towards a General Collection Methodology for Android Devices”
Authors: Timothy Vidas, Chengye Zhang and Nicolas Christin
USA, 2011
- [156] “Identifying back doors, attack points, and surveillance mechanisms in iOS devices”
Authors: Jonathan Zdziarski
2014
- [157] “A Study of Android Application Security”
Authors: William Enck, Damien Ocateau, Patrick McDaniel, and Swarat Chaudhuri
Department of Computer Science and Engineering, The Pennsylvania State University, USA,

Webgraphy

The following references are sorted in order of appearance:

[1] IDC, Smartphone Vendor Market Share (2001-2017), USA

Retrieved from:

<https://www.idc.com/prodserv/smartphone-market-share.jsp>

Accessed: 29/11/2017

[2] Apple, iPhone 4S – Especificaciones técnicas (1987-2017), USA

Retrieved from:

https://support.apple.com/kb/SP643?locale=es_ES&viewlocale=es_ES

Accessed: 29/11/2017

[3] Apple, iPad Air 2 – Especificaciones técnicas (1987-2017), USA

Retrieved from:

https://support.apple.com/kb/SP708?locale=es_ES&viewlocale=es_ES

Accessed: 29/11/2017

[4] Smart-GSM, Moto G4 – Características... (2005-2017), USA

Retrieved from:

<http://www.smart-gsm.com/moviles/motorola-moto-g4>

Accessed: 29/11/2017

[5] Smart-GSM, Xperia Tipo – Características... (2005-2017), USA

Retrieved from:

<http://www.smart-gsm.com/moviles/sony-xperia-tipo>

Accessed: 29/11/2017

[6] Stanford, Mobile device and Platform Security (2004-2017), USA

Retrieved from:

<https://crypto.stanford.edu/cs155/lectures/17-mobile-platforms.pdf>

Accessed: 29/11/2017

[7] Wikipedia, Sandbox (2001-2017), USA

Retrieved from:

[https://en.wikipedia.org/wiki/Sandbox_\(computer_security\)](https://en.wikipedia.org/wiki/Sandbox_(computer_security))

Accessed: 29/11/2017

[8] Apple, App Sandboxing in iOS (1987-2017), USA

Retrieved from:

<https://developer.apple.com/app-sandboxing/>

Accessed: 29/11/2017

[9] Android, SELinux (1997-2017), USA

Retrieved from:

<https://source.android.com/security/selinux/>

Accessed: 29/11/2017

[10] Meinit, Linux Permission System (2007-2017), Netherland

Retrieved from:

<http://meinit.nl/linux-permission-system-explained>

Accessed: 29/11/2017

[11] Fonepaw, Unlocking mechanisms on Android (2014-2017), USA

Retrieved from:

<https://www.fonepaw.com/android-lock/lock-screen-options.html>

Accessed: 29/11/2017

[12] Android, Signing apps on Android (1997-2017), USA

Retrieved from:

<https://developer.android.com/studio/publish/app-signing.html>

Accessed: 29/11/2017

[13] Apple, Signing apps on iOS (1987-2017), USA

Retrieved from:

<https://developer.apple.com/support/code-signing/>

Accessed: 29/11/2017

[14] Washington University, Secure boot chain on iOS (1987-2017), USA

Retrieved from:

https://www.cse.wustl.edu/~jain/cse571-14/ftp/ios_security/index.html

Accessed: 29/11/2017

[15] Android, Android Keystore (1997-2017), USA

Retrieved from:

<https://developer.android.com/training/articles/keystore.html?hl=es-419>

Accessed: 29/11/2017

[16] TechTarget, How iOS encryption and data protection (1999-2017), USA

Retrieved from:

<http://searchmobilecomputing.techtarget.com/tip/How-iOS-encryption-and-data-protection-work>

Accessed: 29/11/2017

[17] PCMag, Remote, wipe and block (1995-2017), USA

Retrieved from:

<https://www.pcmag.com/article2/0,2817,2352755,00.asp>

Accessed: 29/11/2017

[18] Apple, iOS: Find my iPhone (1987-2017), USA

Retrieved from:

<https://support.apple.com/es-es/explore/find-my-iphone-ipad-mac-watch>

Accessed: 29/11/2017

[19] Google, Find my device (1997-2017), USA

Retrieved from:

<https://support.apple.com/es-es/explore/find-my-iphone-ipad-mac-watch>

Accessed: 29/11/2017

[20] Apple, iOS Security Guide (1987-2017), USA

Retrieved from:

https://www.apple.com/business/docs/iOS_Security_Guide.pdf

Accessed: 29/11/2017

[21] Wikipedia, AES (Advanced Encryption Standard) (2001-2017), USA

Retrieved from:

https://es.wikipedia.org/wiki/Advanced_Encryption_Standard

Accessed: 29/11/2017

[22] Apple, TouchID (1987-2017), USA

Retrieved from:

<https://support.apple.com/en-us/HT204587>

Accessed: 29/11/2017

[23] Apple, FaceID Security Guide (1987-2017), USA

Retrieved from:

https://images.apple.com/business/docs/FaceID_Security_Guide.pdf

Accessed: 29/11/2017

[24] Apple, AppStore GuideLines (1987-2017), USA

Retrieved from:

<https://developer.apple.com/app-store/review/guidelines/>

Accessed: 29/11/2017

[25] Statista, AppStore 06/08 – 06/16 (2005-2017), USA

Retrieved from:

<https://www.statista.com/statistics/263795/number-of-available-apps-in-the-apple-app-store/>

Accessed: 29/11/2017

- [26] Apple, Apple File Management (1987-2017), USA
Retrieved from:
<https://developer.apple.com/library/content/documentation/FileManagement/Conceptual/>
Accessed: 29/11/2017
- [27] iMore, APFS (Apple File System) (1999-2017), USA
Retrieved from:
<https://www.imore.com/apfs>
Accessed: 29/11/2017
- [28] Apple, UIKit (1987-2017), USA
Retrieved from:
<https://developer.apple.com/documentation/uikit/uiview>
Accessed: 29/11/2017
- [29] Apple, Programming with Objective-C (1987-2017), USA
Retrieved from:
<https://developer.apple.com/library/content/documentation/Cocoa/Conceptual/ProgrammingWithObjectiveC>
Accessed: 29/11/2017
- [30] Apple, Information Property List (1987-2017), USA
Retrieved from:
<https://developer.apple.com/library/content/documentation/General/Reference/InfoPlistKeyReference>
Accessed: 29/11/2017
- [31] Google, Android Framework (1997-2017), USA
Retrieved from:
<https://developer.android.com/guide/platform/index.html>
Accessed: 29/11/2017
- [32] Google, Google Play (1997-2017), USA
Retrieved from:
<https://play.google.com/store?hl=en>
Accessed: 29/11/2017
- [33] Google, What's new in Android 8.0 Oreo (1997-2017), USA
Retrieved from:
<https://developer.android.com/about/versions/oreo/android-8.0-changes.html>
Accessed: 29/11/2017

- [34] All-Things, Android File System Hierarchy (2012-2017), USA
Retrieved from:
<https://developer.android.com/about/versions/oreo/android-8.0-changes.html>
Accessed: 29/11/2017
- [35] Google, Android Architecture Components (1997-2017), USA
Retrieved from:
<https://developer.android.com/guide/components/fundamentals.html?hl=en>
Accessed: 29/11/2017
- [36] Kaspersky UK, Malware on Mobile Platforms (2000-2017), USA – Retrieved from: 29
<https://www.kaspersky.co.uk/resource-center/threats/mobile>
Accessed: 29/11/2017
- [37] OWASP, TOP 10 Mobile Risks 2016 (2001-2017), USA
Retrieved from:
https://www.owasp.org/index.php/Mobile_Top_10_2016-Top_10
Accessed: 29/11/2017
- [38] OWASP, TOP 10 Mobile Risks 2014 (2001-2017), USA
Retrieved from:
https://www.owasp.org/index.php/Mobile_Security_Project_Archive#tab=Top_10_Mobile_Risks
Accessed: 29/11/2017
- [39] Wikipedia, Jailbreak (2001-2017), USA
Retrieved from:
https://en.wikipedia.org/wiki/IOS_jailbreaking
Accessed: 29/11/2017
- [40] Wikipedia, Root (Android) (2001-2017), USA
Retrieved from:
[https://en.wikipedia.org/wiki/Rooting_\(Android\)](https://en.wikipedia.org/wiki/Rooting_(Android))
Accessed: 29/11/2017
- [41] University Erlangen, Android Cool Boot Attack (2012-2017), Germany
Retrieved from:
<https://www1.informatik.uni-erlangen.de/filepool/projects/frost/frost.pdf>
Accessed: 29/11/2017
- [42] iPhone-Tricks, ScreenLock: Bypassing iOS Locking (2013-2017), USA
Retrieved from:
<http://iphone-tricks.com/tutorial/212-how-to-bypass-iphone-passcode-and-lock-screen>
Accessed: 29/11/2017

- [43] IOActive, Bank XSS Attacks (1998-2017), USA
Retrieved from:
<http://blog.ioactive.com/2015/12/by-ariel-sanchez-two-years-ago-idecided.html>
Accessed: 29/11/2017
- [44] Wikipedia, Celebgate Scandal (2001-2017), USA
Retrieved from:
https://en.wikipedia.org/wiki/ICloud_leaks_of_celebrity_photos
Accessed: 29/11/2017
- [45] FireEye, Masque Attack (1995-2017), USA
Retrieved from:
https://www.fireeye.com/blog/threat-research/2015/02/ios_masque_attackre.html
Accessed: 29/11/2017
- [46] LookOut, Xsser mRat (2001-2017), USA
Retrieved from:
<https://blog.lookout.com/xsser-mrat-ios>
Accessed: 29/11/2017
- [47] Palo Alto Networks, YiSpecter: Malware for iOS (2005-2017), USA
Retrieved from:
<http://researchcenter.paloaltonetworks.com/2015/10/yispecter-first-ios-malware-attacks-non-jailbroken-ios-devices-by-abusing-private-apis/>
Accessed: 29/11/2017
- [48] Palo Alto Networks, XcodeGhost (2005-2017), USA
Retrieved from:
<http://researchcenter.paloaltonetworks.com/2015/09/novel-malware-xcodeghost-modifies-xcode-infests-apple-ios-apps-and-hits-app-store/>
Accessed: 29/11/2017
- [49] Blasting, No more Jailbreaking on iOS (2012-2017), USA
Retrieved from:
<http://us.blastingnews.com/tech/2017/08/jailbreak-ios-1032-analysis-no-more-jailbreaking-apple-ios-001912417.html>
Accessed: 29/11/2017
- [50] Palo Alto Networks, AceDeceiver: Exploiting DRM (2005-2017), USA
Retrieved from:
<https://researchcenter.paloaltonetworks.com/2016/03/acedeceiver-first-ios-trojan-exploiting-apple-drm-design-flaws-to-infect-any-ios-device/>
Accessed: 29/11/2017
- [51] Business Insider, Pegasus (1998-2017), USA
Retrieved from:
<http://www.businessinsider.com/pegasus-nso-group-iphone-2016-8>
Accessed: 29/11/2017

[52] iMore, iTunes Backup Vulnerability (1999-2017), USA – Retrieved from: **37**

<http://www.imore.com/itunes-backup-vulnerability-what-you-need-know>

Accessed: 29/11/2017

[53] Apple, iOS 10.1 (iTunes Backup fix) (1987-2017), USA

Retrieved from:

<https://support.apple.com/en-us/HT207271>

Accessed: 29/11/2017

[54] CyberScoop, Cellebrite (2003-2017), USA

Retrieved from:

<https://www.cyberscoop.com/cellebrite-iphone-6-ufed-samsung-galaxy-facebook-messenger-snapchat/>

Accessed: 29/11/2017

[55] SpamFighter, Fake iTunes Gift Cards (1999-2017), USA

Retrieved from:

<http://www.spamfighter.com/News-17109-Fake-Gift-Certificate-from-iTunes-has-Malware.htm>

Accessed: 29/11/2017

[56] Wikileaks, Vault7 (2006-2017), USA

Retrieved from:

<https://wikileaks.org/ciav7p1/>

Accessed: 29/11/2017

[57] Threatpost, Broadcom chip vulnerability on iOS (2008-2017), USA

Retrieved from:

<https://threatpost.com/remote-wi-fi-attack-backdoors-iphone-7/128163/>

Accessed: 29/11/2017

[58] Techworld, Moonpig (1997-2017), USA

Retrieved from:

<https://www.techworld.com/news/security/moonpig-android-app-flaw-puts-three-million-accounts-at-risk-3592812/>

Accessed: 29/11/2017

[59] Android Central, Stagefright (2007-2017), USA

Retrieved from:

<https://www.techworld.com/news/security/moonpig-android-app-flaw-puts-three-million-accounts-at-risk-3592812/>

Accessed: 29/11/2017

[60] TrendMicro, Dresscode (1995-2017), USA

Retrieved from:

<https://www.techworld.com/news/security/moonpig-android-app-flaw-puts-three-million-accounts-at-risk-3592812/>

Accessed: 29/11/2017

[61] Forbes, Gooligan (1993-2017), USA

Retrieved from:

<https://www.forbes.com/sites/thomasbrewster/2016/11/30/gooligan-android-malware-1m-google-account-breaches-check-point-finds/#5b94c9ec1ad8>

Accessed: 29/11/2017

[62] Graham Cluley, Android Tordow (2001-2017), USA

Retrieved from:

<https://www.grahamcluley.com/tordow-2-0-android-banking-trojan-gains-root-access-mimics-ransomware/>

Accessed: 29/11/2017

[63] Bleeping Computer, Skyfin (2004-2017), USA

Retrieved from:

<https://www.bleepingcomputer.com/news/security/android-trojan-hijacks-google-play-store-covertly-downloads-or-purchases-apps/>

Accessed: 29/11/2017

[64] Bleeping Computer, Energy Rescue (2004-2017), USA

Retrieved from:

<https://www.bleepingcomputer.com/news/security/charger-android-ransomware-reaches-google-play-store/>

Accessed: 29/11/2017

[65] The Inquirer, HummingWhale (2001-2017), USA

Retrieved from:

<https://www.theinquirer.net/inquirer/news/3003152/hummingwhale-android-malware-blows-through-millions-of-devices>

Accessed: 29/11/2017

[66] ZDNet, Android Skinner (1995-2017), USA

Retrieved from:

<http://www.zdnet.com/article/sneaky-adware-exploits-android-users-with-precision-targeting/>

Accessed: 29/11/2017

[67] Security Affairs, Chryasor (2011-2017), USA

Retrieved from:

<http://securityaffairs.co/wordpress/57702/malware/android-chrysaor-spyware.html>

Accessed: 29/11/2017

[68] Softzone, Malware Xavier (2007-2017), USA

Retrieved from:

<http://securityaffairs.co/wordpress/57702/malware/android-chrysaor-spyware.html>

Accessed: 29/11/2017

[69] Marketecheasier, Malware Xavier (2007-2017), USA

Retrieved from:

<https://www.maketecheasier.com/android-virus-ghostctrl/>

Accessed: 29/11/2017

[70] Welivesecurity, Bankbot (2012-2017), USA

Retrieved from:

<https://www.welivesecurity.com/la-es/2017/09/25/troyano-bankbot-google-play/>

Accessed: 29/11/2017

[71] CVE Details (2010-2017), USA

Retrieved from:

<http://www.cvedetails.com/>

Accessed: 29/11/2017

[72] CVE Details: iOS (2010-2017), USA

Retrieved from:

http://www.cvedetails.com/product/15556/Apple-Iphone-Os.html?vendor_id=49

Accessed: 29/11/2017

[73] CVE Details: iOS vulnerabilities by version (2010-2017), USA

Retrieved from:

<http://www.cvedetails.com/version-list/49/15556/1/Apple-Iphone-Os.html>

Accessed: 29/11/2017

[74] CVE Details: Android (2010-2017), USA

Retrieved from:

http://www.cvedetails.com/product/19997/Google-Android.html?vendor_id=1224

Accessed: 29/11/2017

[75] CVE Details: Android vulnerabilities by version (2010-2017), USA

Retrieved from:

<http://www.cvedetails.com/version-list/1224/19997/1/Google-Android.html>

Accessed: 29/11/2017

- [76] Software Fundamentals, White box vs. Black box (2009-2017), USA
Retrieved from:
<http://softwaretestingfundamentals.com/differences-between-black-box-testing-and-white-box-testing/>
Accessed: 29/11/2017
- [77] Wikipedia, Penetration Test (2001-2017), USA
Retrieved from:
https://en.wikipedia.org/wiki/Penetration_test
Accessed: 29/11/2017
- [78] Testing Excellence, Static Analysis (2007-2017), USA
Retrieved from:
<https://www.testingexcellence.com/static-analysis-vs-dynamic-analysis-software-testing/>
Accessed: 29/11/2017
- [79] Testing Excellence, Dynamic Analysis (2007-2017), USA
Retrieved from:
<https://www.testingexcellence.com/static-analysis-vs-dynamic-analysis-software-testing/>
Accessed: 29/11/2017
- [80] Cypress Data, Static Analysis in multiple platforms (2013-2017), USA
Retrieved from:
<https://www.cypressdatadefense.com/security-assessments/application-security-testing/mobile-application/static-analysis/>
Accessed: 29/11/2017
- [81] Forensic Control, Forensics Analysis (2008-2017), USA
Retrieved from:
<https://forensiccontrol.com/resources/beginners-guide-computer-forensics/>
Accessed: 29/11/2017
- [82] Forensic Mag, Forensics Analysis - Fundamentals (2003-2017), USA
Retrieved from:
<https://www.forensicmag.com/article/2011/03/validation-forensic-tools-and-software-quick-guide-digital-forensic-examiner>
Accessed: 29/11/2017
- [83] Github, Qark (2007-2017), USA
Retrieved from:
<https://github.com/linkedin/qark>
Accessed: 29/11/2017

[84] Github, apktool (2007-2017), USA

Retrieved from:

<https://ibotpeaches.github.io/Apktool/>

Accessed: 29/11/2017

[85] Github, androguard (2007-2017), USA

Retrieved from:

<https://github.com/androguard/androguard>

Accessed: 29/11/2017

[86] Bitbucket, dex2jar (1997-2017), USA

Retrieved from:

<https://github.com/androguard/androguard>

Accessed: 29/11/2017

[87] Benow, JD-GUI (2001-2017), USA

Retrieved from:

<http://jd.benow.ca/>

Accessed: 29/11/2017

[88] Github, Clutch (2007-2017), USA

Retrieved from:

<https://github.com/KJCracks/Clutch>

Accessed: 29/11/2017

[89] Cycrypt, cycrypt (2009-2017), USA

Retrieved from:

<http://www.cycrypt.org/>

Accessed: 29/11/2017

[90] Hopper, Hopper Disassembler (2011-2017), USA

Retrieved from:

<https://www.hopperapp.com>

Accessed: 29/11/2017

[91] AppSec Labs, iNalyzer (2011-2017), USA

Retrieved from:

<http://appsec-labs.com/cydia>

Accessed: 29/11/2017

[92] Github, idevicebackup2 (2007-2017), USA

Retrieved from:

<https://github.com/libimobiledevice/libimobiledevice>

Accessed: 29/11/2017

- [93] UseYourLoaf, idevicebackup2 (2009-2017), USA
Retrieved from:
<https://useyourloaf.com/blog/remote-packet-capture-for-ios-devices/>
Accessed: 29/11/2017
- [94] rm-rf, tcp-dump (2008-2017), USA
Retrieved from:
<http://rm-rf.es/tcpdump-ejemplos/>
Accessed: 29/11/2017
- [95] Wireshark, wireshark (2006-2017), USA
Retrieved from:
<https://www.wireshark.org/>
Accessed: 29/11/2017
- [96] Daniel Miessler, tcpdump examples (2009-2017), USA
Retrieved from:
<https://danielmiessler.com/study/tcpdump/>
Accessed: 29/11/2017
- [97] Wireshark, tshark manual (2006-2017), USA
Retrieved from:
<https://www.wireshark.org/docs/man-pages/tshark.html>
Accessed: 29/11/2017
- [98] Wikipedia, Proxy (2001-2017), USA
Retrieved from:
https://en.wikipedia.org/wiki/Proxy_server
Accessed: 29/11/2017
- [99] Port Swigger, Burp Suite (2009-2017), USA
Retrieved from:
<https://portswigger.net/burp>
Accessed: 29/11/2017
- [100] SourceForge, SQLite Manager (1999-2017), USA
Retrieved from:
<https://sourceforge.net/p/sqlitemanager/wiki/Home/>
Accessed: 29/11/2017
- [101] Linuxnix, dd usage (2009-2017), USA
Retrieved from:
<https://www.linuxnix.com/what-you-should-know-about-linux-dd-command/>
Accessed: 29/11/2017

- [102] Wikipedia, dd uses (2001-2017), USA
Retrieved from:
[https://en.wikipedia.org/wiki/Dd_\(Unix\)](https://en.wikipedia.org/wiki/Dd_(Unix))
Accessed: 29/11/2017
- [103] Queen's University, exiftool (2000-2017), USA
Retrieved from:
[https://en.wikipedia.org/wiki/Dd_\(Unix\)](https://en.wikipedia.org/wiki/Dd_(Unix))
Accessed: 29/11/2017
- [104] Google, Dex Files (1997-2017), USA
Retrieved from:
<https://developer.android.com/reference/dalvik/system/DexFile.html>
Accessed: 29/11/2017
- [105] Apple, Swift Programming Language (1987-2017), USA
Retrieved from:
<https://developer.apple.com/swift/>
Accessed: 29/11/2017
- [106] Saurik, Cydia (1997-2017), USA
Retrieved from:
<https://cydia.saurik.com/>
Accessed: 29/11/2017
- [107] OpenSSH, OpenSSH (1999-2017), USA
Retrieved from:
<https://www.openssh.com/>
Accessed: 29/11/2017
- [108] Digital Forensics, Clutch usage (2012-2017), USA
Retrieved from:
<http://digitalforensicstips.com/2015/05/a-quick-guide-to-using-clutch-2-0-to-decrypt-ios-apps/>
Accessed: 29/11/2017
- [109] Wikipedia, Reverse Engineering (2001-2017), USA
Retrieved from:
https://en.wikipedia.org/wiki/Reverse_engineering
Accessed: 29/11/2017
- [110] Hopper, Hopper tutorial (2011-2017), USA
Retrieved from:
<https://www.hopperapp.com/tutorial.html>
Accessed: 29/11/2017

[111] GitHub, apktool usage (2007-2017), USA

Retrieved from:

<https://ibotpeaches.github.io/Apktool/documentation/>

Accessed: 29/11/2017

[112] Google Blog, SHA1 first collision (2002-2017), USA

Retrieved from:

<https://security.googleblog.com/2017/02/announcing-first-sha1-collision.html>

Accessed: 29/11/2017

[113] Google, zipalign (1997-2017), USA

Retrieved from:

<https://developer.android.com/studio/command-line/zipalign.html>

Accessed: 29/11/2017

[114] Apple, Xcode (1987-2017), USA

Retrieved from:

<https://developer.apple.com/xcode/>

Accessed: 29/11/2017

[115] Apple, codesign (1987-2017), USA

Retrieved from:

<https://developer.apple.com/legacy/library/documentation/Darwin/Reference/ManPages/man1/codesign.1.html>

Accessed: 29/11/2017

[116] Stack Overflow, What does debuggable do? (2003-2017), USA

Retrieved from:

<https://stackoverflow.com/questions/37143960/androidstudio-what-does-debuggable-do>

Accessed: 29/11/2017

[117] Ztrela, Androguard manual (2014-2017), USA

Retrieved from:

<http://cyborg.ztrela.com/androguard.php/>

Accessed: 29/11/2017

[118] Hackers Online, androlyze, main tool of Androguard (2011-2017), USA

Retrieved from:

<https://blog.hackersonlineclub.com/2016/12/androguard-reverse-engineering-and.html>

Accessed: 29/11/2017

[119] GitHub, Instropy Analyzer (2007-2017), USA

Retrieved from:

<https://github.com/iSECPartners/Introspy-Analyzer>

Accessed: 29/11/2017

[120] Wikipedia, iOS & macOS Hooking (2001-2017), USA

Retrieved from:

<https://en.wikipedia.org/wiki/Hooking>

Accessed: 29/11/2017

[121] Wikipedia, HMAC (2001-2017), USA

Retrieved from:

https://en.wikipedia.org/wiki/Hash-based_message_authentication_code

Accessed: 29/11/2017

[122] Wikipedia, HTTPS (2001-2017), USA

Retrieved from:

<https://en.wikipedia.org/wiki/HTTPS>

Accessed: 29/11/2017

[123] Symantec, What a sniffer is? (1992-2017), USA

Retrieved from:

<https://www.symantec.com/connect/articles/sniffers-what-they-are-and-how-protect-yourself>

Accessed: 29/11/2017

[124] Free, Android Malware Reverse Engineering (1999-2017), France

Retrieved from:

<http://wikisec.free.fr/papers/androidre-insomnihack2017.pdf>

Accessed: 29/11/2017

[125] NSHipster, NSURL (2012-2017), USA

Retrieved from:

<http://nshipster.com/nsurl/>

Accessed: 29/11/2017

[126] The Geek Stuff, strings (2008-2017), USA

Retrieved from:

<http://www.thegeekstuff.com/2010/11/strings-command-examples/>

Accessed: 29/11/2017

[127] Google, Android SharedPreferences (1997-2017), USA

Retrieved from:

<https://developer.android.com/training/data-storage/shared-preferences.html?hl=en>

Accessed: 29/11/2017

[128] Stack Overflow, How to get the SharedPreferences (2003-2017), USA

Retrieved from:

<https://stackoverflow.com/questions/5950043/how-to-use-getsharedpreferences-in-android>

Accessed: 29/11/2017

- [129] Apple, UserDefaults (1987-2017), USA
Retrieved from:
<https://developer.apple.com/documentation/foundation/userdefaults>
Accessed: 29/11/2017
- [130] Wikipedia, SQLite3 (2001-2017), USA
Retrieved from:
<https://en.wikipedia.org/wiki/SQLite>
Accessed: 29/11/2017
- [131] Packt Pub, Extracting Data on Android (2003-2017), USA
Retrieved from:
<https://www.packtpub.com/books/content/extracting-data-physically-dd>
Accessed: 29/11/2017
- [132] Google, BusyBox (1997-2017), USA
Retrieved from:
<https://play.google.com/store/apps/details?id=stericson.busybox&hl=es>
Accessed: 29/11/2017
- [133] GitHub, AFLogical (2007-2017), USA
Retrieved from:
<https://github.com/nowsecure/android-forensics/downloads>
Accessed: 29/11/2017
- [134] GitHub, LIME (Linux Memory Extractor) (2007-2017), USA
Retrieved from:
<https://github.com/504ensicsLabs/LiME>
Accessed: 29/11/2017
- [135] GitHub, volatility (2007-2017), USA
Retrieved from:
<https://github.com/volatilityfoundation/volatility>
Accessed: 29/11/2017
- [136] LineageOS, LineageOS (2016-2017), USA
Retrieved from:
<https://lineageos.org/>
Accessed: 29/11/2017
- [137] Hashcat, hashcat (2009-2017), USA
Retrieved from:
<https://hashcat.net/hashcat/>
Accessed: 29/11/2017

[138] Hashcat, wiki (2009-2017), USA

Retrieved from:

<https://hashcat.net/wiki/doku.php?id=hashcat>

Accessed: 29/11/2017

[139] SANS, Forensics on Android (1995-2017), USA

Retrieved from:

<https://hashcat.net/wiki/doku.php?id=hashcat>

Accessed: 29/11/2017

[140] SANS, Forensics on iOS (1995-2017), USA

Retrieved from:

<https://www.sans.org/reading-room/whitepapers/forensics/forensic-analysis-ios-devices-34092>

Accessed: 29/11/2017

[141] VirtualBox, VirtualBox (2006-2017), USA

Retrieved from:

<https://www.virtualbox.org/>

Accessed: 29/11/2017

[142] Ubuntu, Xubuntu (XFCE Edition) (2005-2017), Isle of Man

Retrieved from:

<https://xubuntu.org/>

Accessed: 29/11/2017

[143] Google, golang (2009-2017), USA

Retrieved from:

<https://golang.org/>

Accessed: 29/11/2017

[144] Wikipedia, ARM Architecture (2001-2017), USA

Retrieved from:

https://en.wikipedia.org/wiki/ARM_architecture

Accessed: 29/11/2017

[145] CeoLevel, Metrics for a Project Manager (2013-2017), Spain

Retrieved from:

<http://www.ceolevel.com/7-metricas-que-todo-project-manager-deberia-medir>

Accessed: 30/11/2017

[146] DVIA, Damn Vulnerable iOS Application (2013-2017), USA

Retrieved from:

<http://damnvulnerableiosapp.com/>

Accessed: 29/11/2017

[147] Enharmonic HQ, Hopper Disassembler Class Dump (2012-2017), USA

Retrieved from:

<http://www.enharmonicq.com/tutorial-ios-reverse-engineering-class-dump-hopper-dissassembler/>

Accessed: 29/11/2017

[148] Google, Android Dashboard (1997-2017), USA

Retrieved from:

<https://developer.android.com/about/dashboards/index.html?hl=es-419>

Accessed: 29/11/2017

[149] David Smith, iOS Version Stats (2001-2017), USA

Retrieved from:

<https://david-smith.org/iosversionstats/>

Accessed: 29/11/2017

[150] Google, Android Permissions (1997-2017), USA

Retrieved from:

<https://developer.android.com/reference/android/Manifest.permission.html>

Accessed: 29/11/2017

Annexes

Finally, some annexes are included at the end of this thesis to illustrate not only some materials that have been used during the development of the project, either mobile devices, statistics and other resources created specifically for this work, but also ongoing research, or other information related to the topic on which this document is based, which is considered of interest in order to see the future trends that may arise in this area.

A, Usage on both platforms

Accordinging of Android Dashboards in Figure A.1, the distribution of Android versions in November 2017 [148] is as follows:

Version	Codename	API	Distribution
2.3.3 - 2.3.7	Gingerbread	10	0.6%
4.0.3 - 4.0.4	Ice Cream Sandwich	15	0.6%
4.1.x	Jelly Bean	16	2.4%
4.2.x		17	3.5%
4.3		18	1.0%
4.4	KitKat	19	15.1%
5.0	Lollipop	21	7.1%
5.1		22	21.7%
6.0	Marshmallow	23	32.2%
7.0	Nougat	24	14.2%
7.1		25	1.6%

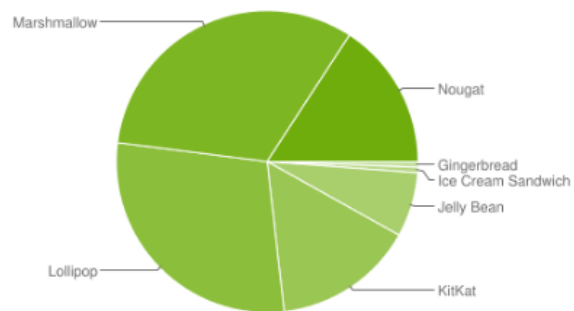


Figure A.1 Android Distribution on September 2017 (Source: Android Dashboard)

Just an almost 16% use the latest version of Android. In the case of the iOS operating system, fragmentation is not as palpable as in Android, as we see in Table A.1:

iOS Distribution (December 2017) [149]	
Version	Percentage
11.x	68.0%
10.x	18.6%
9.x	10.3%
8.x	0.5%
7.x	0.8%
6.x	1.2%
5.x	0.5%
4.x	0.1%

Table A.1. iOS Distribution (December 2017)

As can be seen, almost 87% of users use the most up-to-date versions of the system, which helps them to be protected from the latest vulnerabilities in terms of security.

B. Android Permissions Table

It could be seen a table B1 [150] with the main permissions, and their use taken from the Android Developer page, classified according to their degree of danger, so that they can serve as a guide for any analyst in mobile security, at the moment that it is ready to make an evaluation of the permissions demanded by each application of the system being analyzed. At the same time, this classification, is taken into account, by the time one of the scripts developed for this thesis (manifest_interpreter), is running.

Android Permission System		
Permission	Description	Scope
ACCESS_CHECKIN_PROPERTIES	Allows r/w access to the properties table in the checkin database.	DANGER
ACCESS_COARSE_LOCATION	Allows an app to access approximate location.	DANGER
ACCESS_FINE_LOCATION	Allows an app to access precise location.	DANGER
ACCESS_LOCATION_EXTRA_COMMANDS	Allows and app to access extra location provider commands.	DANGER
ACCESS_NETWORK_STATE	Allows apps to access information about networks.	NEUTRAL
ACCESS_NOTIFICATION_POLICY	Allows apps to access to notification policy.	NEUTRAL
ACCESS_WIFI_STATE	Allows apps to access information about WiFi networks.	NEUTRAL
ACCOUNT_MANAGER	Allows apps to call into AccountAuthenticators.	NEUTRAL
ADD_VOICEMAIL	Allows apps to add voicemails into the system.	NEUTRAL
ANSWER_PHONE_CALLS	Allows an app to answer an incoming phone call.	DANGER
BATTERY_STATS	Allows an app to collect battery statistics.	NEUTRAL
BIND_ACESSIBILITY_SERVICE	Must be required by an AccessibilityService to ensure that only the system can bind to it.	NEUTRAL
BIND_APPWIDGET	Allows an app to tell the AppWidget service which app can access to its data.	NEUTRAL
BIND_AUTOFILL_SERVICE	Must be required by an AutofillService to ensure that only the system can bind to it.	NEUTRAL
BIND_CARRIER_MESSAGING_SERVICE	Constant deprecated from API 23 onwards.	DEPRECATED
BIND_CARRIER_SERVICES	The system process that is allowed to bind to services in carrier apps will have this permission.	NEUTRAL
BIND_CHOOSER_TARGET_SERVICE	Must be required by a ChooserTargetService to ensure that only the system can bind to it.	NEUTRAL

BIND_CONDITION_PROVIDER_SERVICE	Must be required by a ConditionProviderService to ensure that only the system can bind to it.	NEUTRAL
BIND_DEVICE_ADMIN	Must be required by device administration receiver to ensure that only the system can interact with it.	NEUTRAL
BIND_DREAM_SERVICE	Must be required by DreamService to ensure that only the system can bind to it	NEUTRAL
BIND_INCALL_SERVICE	Must be required by an InCallService to ensure that only the system can bind to it.	NEUTRAL
BIND_INPUT_METHOD	Must be required by an InputMethodService to ensure that only the system can bind to it.	NEUTRAL
BIND_MIDI_DEVICE_SERVICE	Must be requires by a MidiDeviceService to ensure that only the system can bind to it.	NEUTRAL
BIND_NFC_SERVICE	Must be required by a HostApuService or OffHostApuService to ensure that only the system can bind to it.	NEUTRAL
BIND_NOTIFICATION_LISTENER_SERVICE	Must be required by an NotificationListenerService to ensure that only the system can bind to it.	NEUTRAL
BIND_PRINT_SERVICE	Must be required by a PrintService to ensure that only the system can bind to it.	NEUTRAL
BIND_QUICK_SETTINGS_TILE	Allows an app to bind to third party quick setting tiles.	NEUTRAL
BIND_REMOTEVIEWS	Must be required by a RemoteViewsService to ensure that only the system can bind to it.	NEUTRAL
BIND_SCREENING_SERVICE	Must be required by a CallScreeningService to ensure that only the system can bind to it.	NEUTRAL
BIND_TELECOM_CONNECTION_SERVICE	Must be required by a ConnectionService to ensure that only the system can bind to it.	NEUTRAL
BIND_TEST_SERVICE	Must be required by a TextService.	NEUTRAL
BIND_TV_INPUT	Must be required by a TvInputService to ensure that only the system can bind to it.	NEUTRAL
BIND_VISUAL_VOICEMAIL_SERVICE	Must be required by a link VisualVoicemailService to ensure that only the system can bind to it.	NEUTRAL
BIND_VOICE_INTERACTION	Must be required by a VoiceInteractionService to ensure that only the system can bind to it.	NEUTRAL
BIND_VPN_SERVICE	Must be required by a VpnService to ensure that only the system can bind to it.	NEUTRAL
BIND_VR_LISTENER_SERVICE	Must be required by a VrListenerService to ensure that only the system can bind to it.	NEUTRAL
BIND_WALLPAPER	Must be required by a WallpaperService to ensure that only the system can bind to it.	NEUTRAL

BLUETOOTH	Allows apps to connect to paired bluetooth devices.	DANGER
BLUETOOTH_ADMIN	Allows apps to discover and pair bluetooth devices.	DANGER
BLUETOOTH_PRIVILEGED	Allows apps to pair bluetooth devices without user interaction and to allow/disallow phonebook/message access.	DANGER
BODY_SENSORS	Allows an app to access data from sensors like for instance: heart rate.	NEUTRAL
BROADCAST_PACKAGE_REMOVED	Allows an app to broadcast a notification that an app package has been removed.	DANGER
BROADCAST_SMS	Allows an app to broadcast an SMS receipt notification.	DANGER
BROADCAST_STICKY	Allows an app to broadcast sticky intents.	NEUTRAL
BROADCAST_WAP_PUSH	Allows an app to broadcast a WAP PUSH receipt notification.	NEUTRAL
CALL_PHONE	Allows an application to initiate a phone call without going through the Dialer user interface for the user to confirm the call.	DANGER
CALL_PRIVILEGED	Allows an application to call any phone number, including emergency numbers, without going through the Dialer user interface for the user to confirm the call being placed.	DANGER
CAMERA	Required to be able to access the camera device.	DANGER
CAPTURE_AUDIO_OUTPUT	Allows an application to capture audio output.	DANGER
CAPTURE_SECURE_VIDEO_OUTPUT	Allows an application to capture secure video output.	DANGER
CAPTURE_VIDEO_OUTPUT	Allows an application to capture video output.	DANGER
CHANGE_COMPONENT_ENABLED_STATE	Allows an application to change whether an application component (other than its own) is enabled or not.	NEUTRAL
CHANGE_CONFIGURATION	Allows an application to modify the current configuration, such as locale.	NEUTRAL
CHANGE_NETWORK_STATE	Allows applications to change network connectivity state.	NEUTRAL
CHANGE_WIFI_MULTICAST_STATE	Allows applications to enter Wi-Fi Multicast mode.	NEUTRAL
CHANGE_WIFI_STATE	Allows applications to change Wi-Fi connectivity state.	NEUTRAL
CLEAR_APP_CACHE	Allows an application to clear the caches of all installed applications on the device.	NEUTRAL
CONTROL_LOCATION_UPDATES	Allows enabling/disabling location update notifications from the radio.	NEUTRAL
DELETE_CACHE_FILES	Allows an application to delete cache files.	NEUTRAL
DELETE_PACKAGES	Allows an application to delete packages.	NEUTRAL

DIAGNOSTIC	Allows applications to RW to diagnostic resources.	NEUTRAL
DISABLE_KEYGUARD	Allows applications to disable the keyguard if it is not secure.	NEUTRAL
DUMP	Allows an application to retrieve state dump information from system services.	NEUTRAL
EXPAND_STATUS_BAR	Allows an application to expand or collapse the status bar.	NEUTRAL
FACTORY_TEST	Run as a manufacturer test application, running as the root user.	NEUTRAL
GET_ACCOUNTS	Allows access to the list of accounts in the Accounts Service.	NEUTRAL
GET_ACCOUNTS_PRIVILEGED	Allows access to the list of accounts in the Accounts Service.	DANGER
GET_PACKAGE_SIZE	Allows an application to find out the space used by any package.	NEUTRAL
GET_TASKS	This constant was deprecated in API level 21. No longer enforced.	DEPRECATED
GLOBAL_SEARCH	This permission can be used on content providers to allow the global search system to access their data.	NEUTRAL
INSTALL_LOCATION_PROVIDER	Allows an application to install a location provider into the Location Manager.	NEUTRAL
INSTALL_PACKAGES	Allows an application to install packages.	DANGER
INSTALL_SHORTCUT	Allows an application to install a shortcut in Launcher.	DANGER
INSTANT_APP_FOREGROUND_SERVICE	Allows an instant app to create foreground services.	NEUTRAL
INTERNET	Allows applications to open network sockets.	DANGER
KILL_BACKGROUND_PROCESSES	Allows an application to call killBackgroundProcesses(String).	DANGER
LOCATION_HARDWARE	Allows an application to use location features in hardware, such as the geofencing api.	DANGER
MANAGE_DOCUMENTS	Allows an application to manage access to documents, usually as part of a document picker.	DANGER
MANAGE_OWN_CALLS	Allows a calling application which manages its own calls through the self-managed ConnectionService APIs.	DANGER
MASTER_CLEAR	Not for use by third-party applications.	NEUTRAL
MEDIA_CONTENT_CONTROL	Allows an application to know what content is playing and control its playback.	NEUTRAL
MODIFY_AUDIO_SETTINGS	Allows an application to modify global audio settings.	NEUTRAL
MODIFY_PHONE_STATE	Allows modification of the telephony state - power on, mmi, etc.	DANGER
MOUNT_FORMAT_FILESYSTEMS	Allows formatting file systems for removable storage.	DANGER
MOUNT_UNMOUNT_FILESYSTEMS	Allows mounting and unmounting file systems for removable storage.	DANGER
NFC	Allows applications to perform I/O operations over NFC.	DANGER

PACKAGE_USAGE_STATS	Allows an application to collect component usage statistics	NEUTRAL
PERSISTENT_ACTIVITY	This constant was deprecated in API level 9. This functionality will be removed in the future; please do not use. Allow an application to make its activities persistent.	DEPRECATED
PROCESS_OUTGOING_CALLS	Allows an application to see the number being dialed during an outgoing call with the option to redirect the call to a different number or abort the call altogether.	DANGER
READ_CALENDAR	Allows an application to read the user's calendar data.	DANGER
READ_CALL_LOG	Allows an application to read the user's call log.	DANGER
READ_CONTACTS	Allows an application to read the user's contacts data.	DANGER
READ_EXTERNAL_STORAGE	Allows an application to read from external storage.	DANGER
READ_FRAME_BUFFER	Allows an application to take screen shots and more generally get access to the frame buffer data.	NEUTRAL
READ_INPUT_STATE	This constant was deprecated in API level 16. The API that used this permission has been removed.	DEPRECATED
READ_LOGS	Allows an application to read the low-level system log files.	DANGER
READ_PHONE_NUMBERS	Allows read access to the device's phone number(s).	DANGER
READ_PHONE_STATE	Allows read only access to phone state, including the phone number of the device, current cellular network information, the status of any ongoing calls, and a list of any PhoneAccounts registered on the device.	DANGER
READ_SMS	Allows an application to read SMS messages.	DANGER
READ_SYNC_SETTINGS	Allows applications to read the sync settings.	NEUTRAL
READ_SYNC_STATS	Allows applications to read the sync stats.	NEUTRAL
READ_VOICEMAIL	Allows an application to read voicemails in the system.	NEUTRAL
REBOOT	Required to be able to reboot the device.	DANGER
RECEIVE_BOOT_COMPLETED	Allows an application to receive the ACTION_BOOT_COMPLETED that is broadcast after the system finishes booting.	NEUTRAL
RECEIVE_MMS	Allows an application to monitor incoming MMS messages.	DANGER
RECEIVE_SMS	Allows an application to receive SMS messages.	DANGER
RECEIVE_WAP_PUSH	Allows an application to receive WAP push messages.	DANGER
RECORD_AUDIO	Allows an application to record audio.	DANGER
REORDER_TASKS	Allows an application to change the Z-order of tasks.	DANGER

REQUEST_COMPANION_RUN_IN_BACKGROUND	Allows a companion app to run in the background.	NEUTRAL
REQUEST_COMPANION_USE_DATA_IN_BACKGROUND	Allows a companion app to use data in the background.	NEUTRAL
REQUEST_DELETE_PACKAGES	Allows an application to request deleting packages.	NEUTRAL
REQUEST_IGNORE_BATTERY_OPTIMIZATIONS	Permission an application must hold in order to use a certain permission	NEUTRAL
REQUEST_INSTALL_PACKAGES	Allows an application to request installing packages.	NEUTRAL
RESTART_PACKAGES	This constant was deprecated in API level 8. The restartPackage(String) API is no longer supported.	DEPRECATED
SEND_RESPOND_VIA_MESSAGE	Allows an application (Phone) to send a request to other applications to handle the respond-via-message action during incoming calls.	NEUTRAL
SEND_SMS	Allows an application to send SMS messages.	DANGER
SET_ALARM	Allows an application to broadcast an Intent to set an alarm for the user.	NEUTRAL
SET_ALWAYS_FINISH	Allows an application to control whether activities are immediately finished when put in the background.	NEUTRAL
SET_ANIMATION_SCALE	Modify the global animation scaling factor.	NEUTRAL
SET_DEBUG_APP	Configure an application for debugging.	NEUTRAL
SET_PREFERRED_APPLICATIONS	This constant was deprecated in API level 7. No longer useful, see addPackageToPreferred(String) for details.	DEPRECATED
SET_PROCESS_LIMIT	Allows an application to set the maximum number of (not needed) application processes that can be running.	NEUTRAL
SET_TIME	Allows applications to set the system time.	DANGER
SET_TIME_ZONE	Allows applications to set the system time zone.	DANGER
SET_WALLPAPER	Allows applications to set the wallpaper.	NEUTRAL
SET_WALLPAPER_HINTS	Allows applications to set the wallpaper hints.	NEUTRAL
SIGNAL_PERSISTENT_PROCESSES	Allow an application to request that a signal be sent to all persistent processes.	NEUTRAL
STATUS_BAR	Allows an application to open, close, or disable the status bar and its icons.	NEUTRAL
SYSTEM_ALERT_WINDOW	Allows an app to create windows using the type TYPE_APPLICATION_OVERLAY, shown on top of all other apps.	NEUTRAL
TRANSMIT_IR	Allows using the device's IR transmitter, if available.	NEUTRAL
UNINSTALL_SHORTCUT	This permission is no longer supported.	DEPRECATED

UPDATE_DEVICE_STATS	Allows an application to update device statistics.	NEUTRAL
USE_FINGERPRINT	Allows an app to use fingerprint hardware.	DANGER
USE_SIP	Allows an application to use SIP service.	DANGER
VIBRATE	Allows access to the vibrator.	NEUTRAL
WAKE_LOCK	Allows using PowerManager WakeLocks to keep processor from sleeping or screen from dimming.	NEUTRAL
WRITE_APN_SETTINGS	Allows applications to write the APN settings.	DANGER
WRITE_CALENDAR	Allows an application to write the user's calendar data.	DANGER
WRITE_CALL_LOG	Allows an application to write (but not read) the user's call log data.	DANGER
WRITE_CONTACTS	Allows an application to write the user's contacts data.	DANGER
WRITE_EXTERNAL_STORAGE	Allows an application to write to external storage.	DANGER
WRITE_GSERVICES	Allows an application to modify the Google service map.	NEUTRAL
WRITE_SECURE_SETTINGS	Allows an application to read or write the secure system settings.	NEUTRAL
WRITE_SETTINGS	Allows an application to read or write the system settings.	DANGER
WRITE_SYNC_SETTINGS	Allows applications to write the sync settings.	NEUTRAL
WRITE_VOICEMAIL	Allows an application to modify and remove existing voicemails in the system.	NEUTRAL

Table B.1. Android Permissions Classification

Acronyms

• AES (Advanced Encryption Standard)	pp.14,15
• APFS (Apple File System)	pp.17
• APK (Android Package Kit)	pp.45,57,59
• AP (Access Point)	pp.73
• API (Application Programming Interface)	pp.34,57,86,88,90,91,116,119,120
• AR (Augmented Reality)	pp.4
• ARC (Automatic Reference Counting)	pp.66
• ART (Android Runtime)	pp.20
• ARM (Advanced RISC Machine)	pp.20,83
• BYOD (Bring Your Own Device)	pp.29
• C&C (Command & Control)	pp.25
• CA (Certificate Authority)	pp.61
• CBC (Cypher Block Chaining)	pp.14
• CD (Compact Disc)	pp.50,77
• CIA (Central Intelligence Agency)	pp.33
• CVE (Common Vulnerabilities and Exposures)	pp.32,37,38
• DB (DataBase)	pp.49
• DCIM (Digital Camera Images)	pp.68,72,102,103,104
• DDR (Double Data Rate)	pp.9
• DEX (Dalvik Executable)	pp.45
• DRM (Digital Rights Management)	pp.32,54
• DoS (Denial of Service)	pp.24
• DVD (Digital Video Disc)	pp.77
• DVIA (Damn Vulnerable iOS App)	pp.77,78,96,97,99
• EXIF (Exchangeable Image File Format)	pp.50,72,102
• GB (GigaByte)	pp.9
• GHz (GigaHertz)	pp.9
• GPS (Global Positioning System)	pp.70,73,104,105,106
• GUI (Graphical User Interface)	pp.4,46,47,52,77,78
• HFS+ (Hierarchical File System plus)	pp.17
• HMAC (Hash Message Authentication Code)	pp.58
• HTTP (Hypertext Transfer Protocol)	pp.49,61,101,124
• HTTPS (Hypertext Transfer Protocol Secure)	pp.49,61,124
• IDE (Integrated Development Environment)	pp.18,19,31,46,51,54,55,77
• IMEI (International Mobile Equipment Identity)	pp.74
• I/O (Input/Output)	pp.119
• IP (Internet Protocol)	pp.31
• IT (Information Technology)	pp.3
• JFFS2 (Journal Flash File System 2)	pp.22
• JSON (JavaScript Object Notation)	pp.90,96,106,109
• LG (Lucky-Goldstar)	pp.21

- **LIME** (Linux Memory Extractor) pp.66
- **LLB** (Low-Level Bootlander) pp.14
- **LTS** (Long Term Support) pp.75,95
- **MB** (MegaByte) pp.65
- **MD5** (Message-Digest 5) pp.69
- **MMS** (Multimedia Messaging Service) pp.65,70,72,120
- **NFC** (Near Field Communication) pp.40,117,119
- **OS** (Operating System) pp.3,12,14,15,17,21,30,32,36,38,50,76
- **OTA** (On-The-Air) pp.21
- **OVF** (Open Virtualization Format) pp.76
- **OWASP** (Open Web Application Security Project) pp.25,26,27
- **PBKDF2** (Password-Based Key Derivation Function 2) pp.32
- **PIN** (Personal Identificator Number) pp.33,35
- **PSK** (Pre-Shared Key) pp.69
- **RAM** (Random Access Memory) pp.9,10,29,42,54,65,66,75
- **RISC** (Reduced Instruction Set Computer) pp.20
- **RSA** (Rivest-Shamir-Addleman) pp.54
- **SAW** (Security Analysis Workshop) pp.75
- **SD** (Secure Digital) pp.14,22,50,63,66,68
- **SDK** (Software Development Kit) pp.17,18,54,63
- **SELinux** (Security-Enhanced Linux) pp.12
- **SHA** (Secure Hash Algorithm) pp.32
- **SIM** (subscriberIdentity Module) pp.29,74
- **SMS** (Short Messages Service) pp.24,29,31,35,36,65,70,72,118,120
- **SQL** (Structured Query Language) pp.49,62,69
- **SSH** (Secure Shell) pp.52,65
- **SSL** (Secure Sockets Layer) pp.20,25,31,61,100
- **TCP** (Transmission Control Protocol) pp.47
- **TLS** (Transport Layer Security) pp.22,32,61
- **UDID** (Unique Device Identifier) pp.12,18,47,71
- **UDP** (User Datagram Protocol) pp.47
- **UI** (User Interface) pp.18,21
- **UPnP** (Universal Plug and Play) pp.25
- **URI** (Uniform Resource Identifier) pp.19
- **URL** (Uniform Resource Locator) pp.60,99
- **USB** (Universal Serial Bus) pp.29,30,63,77
- **UTF-8** (Unicode Transformation Format 8) pp.19
- **WEP** (Wired Equivalent Privacy) pp.69
- **WPA** (Wi-Fi Protected Access) pp.69
- **Wi-Fi** (Wireless Fidelity) pp.28,118
- **XML** (eXtensible Markup Language) pp.19,86,109
- **XSS** (Cross-site Scripting) pp.31

