DroidSF - A framework for security analysis of mobile applications *

Extendend Abstract

João Miguel Martins Nunes[0000−0002−8413−1491]

Instituto Superior Técnico, Universidade de Lisboa, Portugal
https://tecnico.ulisboa.pt/

Abstract. Mobile devices, specially smart-phones, are an increasingly valuable target for bad actors as they often hold important personal information, that can potentially be exploited against its user.
With the growing number of mobile devices connected to the internet, it's imperative that we develop tools and document how to perform an in-depth analysis of mobile applications. We believe this knowledge will help software developers, and even users, to be more conscious about security and implement better code following the recommended practices.
This thesis will cover techniques and software one can use to analyse how an Android application was built and gain insight to what it does in background. To be able to evaluate the security of an Android mobile application it's imperative to understand how they are developed, assembled and how they operate on devices at runtime. With this in mind, we will provide details about the inner-workings of the Android platform, with special attention to its security features.
A framework was produced along side this thesis, that aggregates frequently used tools to facilitate the security analysis process. Our framework was designed to be a fully automated, easy to use and extendable. These characteristics seek to promote a good starting point for anyone that wants to analyse the behaviour of mobile applications and develop systematic tests to help assert their overall security level.
We'll focus on methodologies that allow us to inspect critical components of the application as described by the Open Web Application Security Project (OWASP) Top 10 Security Risks.
Security analysis; Android applications; Mobile security; Reverse engineering; Static analysis; Dynamic analysis; Binary instrumentation

Keywords: Security analysis · Android applications · Mobile security · Reverse engineering · Static analysis · Dynamic analysis.

* Orientation by: Prof. Pedro Adão

1 Introduction

The Internet has become an essential part of the daily life of many people. It has evolved from a basic communication network to an interconnected set of data sources with market places for the sale of products and services. Services like online banking or advertising are some of most successful areas on the Internet for commercial purposes [2].

In an effort to support all the modern functionalities and inter-connectivity that we have come to expect from recent mobile devices, software is getting increasingly more complex and often includes multiple libraries from external sources. This kind of complexity and inter-connectivity increases the risk of security vulnerabilities which, in turn, can have severe consequences to people and systems that interact with compromised software.

Just as in the physical world, there are people on the Internet with malevolent intents that relentlessly search for vulnerabilities to exploit so they can enrich themselves, while taking advantage of oblivious users.

Vulnerabilities enable a variety of attacks. The analysis of these attacks can determine the severity of damage that can be inflicted and the likelihood that the attack can be further replicated.

The task of screening and validating if an application is secure can be very time-consuming and easily overwhelms analysts that try to perform this task manually. Due to the very substantial number of sample applications submitted for security review every day, it is paramount that we use an automated approach to quickly differentiate between samples that deserve an in-depth manual analysis, and those that are a variation of already known threats [2].

1.1 Android platform

Android is a Operating System (OS) for mobile devices based on the Linux OS and it includes additional system libraries, middle-ware, and a suite of pre-installed applications. Android applications, also commonly known as 'apps', are mainly written in Java by using a rich collection of Application Programming Interfaces (APIs) provided by the Android Software Development Kit (SDK). Compiled code is packed into an archive file, alongside data and resources required by the application. This archive file is known as an Application Package Kit (APK) and once it is installed on an Android device, it runs by using the Android Runtime (ART) environment.

We chosen the Android platform because it has around 75% worldwide market share in the mobile device space [11]. Its open-source nature was also a major factor for us, as it contributes to better documentation, bigger developer communities and the availability of tools to interact with the Android OS.

1.2 Reverse Engineering

Reverse Engineer (RE) is the process of reconstructing the semantics of a compiled program's source code.

This thesis will focus on several techniques and tools that can be leveraged to analyse the security Android mobile devices, in particular RE techniques.

The motivation behind our efforts to RE an application is solely focused on assessing its overall security.

The RE process can be split in two main types of analysis:

- Static analysis allows inspection of an application without actually executing it. This process often requires the disassembly and/or decompilation of binary code to run tests and obtain human-readable code.
- Dynamic analysis refers to techniques that execute an application and allow inspection of its state at various points in execution. Using this approach one can analyse the behaviour of a binary application at runtime through the injection of instrumentation code.

1.3 Goals

We will use security analysis techniques and explain how to implement test suites for mobile applications, closely following the Open Web Application Security Project (OWASP) Top 10 Security Risks and the Mobile Security Testing Guide.

A major part of this thesis focuses on the development of a framework that enables developers and researchers to analyse Android applications programmatically without requiring too much effort to configure.

Our framework is not intended for piracy and other non-legal uses. It was built and designed solely to facilitate the security assessment of Android applications.

The framework was named Android Security Framework or DroidSF for short. It combines various existing tools created and maintained by security experts, to allow a systematic and easy way analyse mobile applications.

DroidSF Github repository: `https://github.com/neskk/droidsf`

The DroidSF framework is completely open-source, works in multiple platforms (Windows, Linux, MacOS) and was planned to be extensible and customizable, so that it can grow with the help of other developers and adapt to the ever-changing technology found in mobile devices.

We built a platform where static and dynamic analyses co-exist and complement each other. This key aspect is what sets our work apart from existing frameworks.

The DroidSF framework provides a convenient way to experiment new RE techniques, and to analyse applications in a fully automated way. Supporting all major OSes is also another important strength of our framework, since most of the existing frameworks, such as AppMon [8] and DroidStat-X [1], do not run natively on Windows targeting only Linux and MacOS environments.

Some other mobile security testing frameworks attempt to support both Android and iOS at the same time, leading to a much greater code-base which is harder to maintain and introduces unnecessary complexity. We decided to focus our framework just on the Android platform to attempt to minimize these problems.

2 State of the Art

Reverse engineering and tampering techniques have long been associated to the realm of hackers and malware analysts. For traditional security testers and researchers, RE has been more of a complementary skill, but the tides are turning. Testing mobile applications increasingly requires disassembling compiled applications, applying patches, and tampering with binary code or even live processes. The fact that many mobile applications implement defences against RE makes things harder for security analysts.

Reverse engineering a mobile application is the process of analysing the compiled software to extract information about its source code with intent of understanding how the application work [6].

Tampering is the process of modifying a mobile application, either by changing the compiled byte-code, instrumenting the running process or its environment, in order to affect the behaviour of the application being tested [6]. For example, it is common for an application to refuse to run on rooted devices, making it impossible to run certain tests or use Android's debugging functionalities. In such cases, we want to alter the application's behaviour.

Mobile security testers should have a basic understanding of RE concepts, mobile devices and operating systems. RE is an art, and describing its every facet could easily fill a whole library. The sheer volume of techniques and specializations can be overwhelming. One can spend years working on a very concrete and isolated sub-problem, such as automating malware analysis or developing improved de-obfuscation methods. Security testers have to be generalists. In order to become an effective reverse engineer, one must filter through the vast amount of relevant information [6].

In recent years, researchers have developed a variety of tools and methodologies to conduct analysis of Android applications. While all the respective papers aim at providing a thorough empirical evaluation, comparability is hindered by varying or unclear evaluation targets. These limitations make it impossible to directly compare approaches and we have accept that there are solutions more suited for some tasks than others [9]. This fact reinforces the need for security testers to diversify their knowledge and learn the generic concepts behind RE, rather than learning a specific tool or methodology.

There is no universal recipe for the RE process that always works. Acknowledging this fact, through out this chapter we will first focus on describing the current state-of-the-art of the Android platform, in particular, information regarding its security features. Secondly, we will provide details about commonly used RE methods and tools. Finally, some examples of tackling the most common anti-reverse defences will also be analysed.

3 Proposed Solution

We knew from the beginning we wanted to build a flexible framework capable of helping security researchers perform a thorough analysis of Android applications using modern reserve engineering approaches.

The framework would need to integrate both static and dynamic analyses in an way that they could complement each other. It also had to be easy to configure, while being fully automated to allow batch testing of applications.

Our work focused mostly on integrating tools already employed and well documented by the mobile security community, into a single framework, to allow a convenient way to leverage their functionalities.

We decided to start working on top of DroidStat-X [1], a static analysis framework also built on Python. This framework was especially attractive for us, because it was structured following the OWASP Mobile Top 10 categories, which we also decided follow.

3.1 Design and Requirements

In our proposal we established some broad requirements we needed to fulfil in order to create a powerful and capable mobile security analysis framework. The first requirement was to focus exclusively on the Android platform to make sure we had a well defined scope for our security analysis. The second requirement was that the whole analysis process had to allow full automation in order to be ale to cope with the significant number of applications released every day. The last requirement established that the framework would have to be able identify some of most common vulnerabilities found in Android applications. These requirements distinguish our framework from existing ones (e.g., ApkX [4], Objection [10] and AppMon [8]), and were established to help shape its design process.

We wanted to create a useful and attractive framework for developers and security researchers to build upon. To achieve this goal, we spent a considerable amount of time laying out the most important design aspects we would have to follow.

3.2 Design Choices

Below we present the key design aspects we closely followed during the implementation of our framework:

Multi-platform: It was important for us to create a framework that could work natively on all the major desktop platforms: Windows, Linux and MacOS. This removes the need to use a virtualisation software which usually adds significant performance overhead and has less resources available to perform the analysis. Frameworks with similar characteristics to ours have been developed, but mostly are built to operate on Linux and have dependencies that are not available by default on Windows. We decided to build our framework using Python because it is an interpreted, high-level, general-purpose programming language and it is available for all the major OSes. Additionally, there are many security analysis tools built on Python, so it made sense for us to follow this trend in order to take advantage of its vast and active community of developers.

Easy to install: Minimizing the time it takes to setup the framework was a priority for us. To achieve this goal we tried to reduce the number pre-requisites the user has to install manually. We also took advantage of Python's package installer 'pip', which is installed by default with Python, to automate the download and installation of required Python packages.

Batteries included: Our framework uses various third-party external tools to execute several tasks. Since all the required tools are available for free on the internet, we decided to automate the whole process of downloading and configuring them to run from within our framework. To respect the multi-platform design, we made sure to select external tools that can run either directly on Python or on Java runtime environment, as Java is also available on all major OSes. The goal was to provide a pleasant user experience and avoid time-consuming tasks where we could.

Configurable: We wanted users to be able to easily customize the analysis and allow the definition of profiles to avoid manually specifying configuration options on each run. In cases where more than one external tool can be used to perform a certain task, we decided to give users the ability to choose which one they want to use. All the configuration options can be defined through command-line parameters or using a configuration file. Users can specify a configuration file through the command-line to act as a profile for performing analysis on a batch of applications.

Extendable: In order to be able to keep up with new techniques, we allow custom static analysis checks to be performed on 'smali' code. Users can add regular expressions to be tested through a configuration parameter, and the results will be output in the analysis report. Our framework also allows running custom Frida injection scripts during the dynamic analysis process. Users just have to specify the location for the custom Frida script they wish to execute through a configuration parameter.

Instrumentation script templates: To take advantage of the information collected during static analysis we needed a convenient way to customize the instrumentation scripts. We developed a simple template system that essentially replaces predefined place-holders in the scripts with information obtained during the static analysis. Another feature we implemented in the template system enables users to build a script that includes other scripts, allowing a modular approach to scripts.

Fully automated: We knew that one of our major challenges was the automation of the dynamic analysis, since static analysis is naturally an automated process. Taking advantage of Python's ability to easily interact with the native OS, we managed to automate almost every step of the binary instrumentation we use to perform the dynamic analysis. From setting up the device, to running the instrumentation script on application, it can all be done without user interaction.

Evaluating the techniques, methodologies and tools we discussed in the present report is not a linear task. Having no simple way to directly compare

results with other frameworks with similar goals and motivations, we focused on evaluating the performance and basic ability to perform certain tasks.

3.3 Methodology

To evaluate our framework we devised a three step process.

The first step on our tests was to make sure that our framework had the ability to consistently disassemble, decompile and run static analysis tests.

Taking advantage of the findings produced by static analysis, the second step in our tests assessed the ability of customizing the instrumentation code to the application under analysis.

Afterwards, we attempt to run the application on our Android Virtual Device (AVD) environment. If we are able to execute the application, then the final test consists on injecting the instrumentation code and, after some interaction with the application, export the results.

4 Evaluating the Solution

Most of the applications were selected through an exhaustive search to maximize the amount of implemented checks we covered in our tests.

We wanted to ensure the correctness and completeness of the checks implemented in our framework. To do so, we required applications that exposed vulnerabilities and bad security practices. We searched various different sample applications and found some that were built especially for the purpose of training security testers.

Intentionally vulnerable Android applications:

- InsecureBank v.2
  `https://github.com/dineshshetty/Android-InsecureBankv2`
- PIVAA v.1
  `https://github.com/HTBridge/pivaa`
- DVHMA-FeatherWeight v.6.3.0
  `https://github.com/logicalhacking/DVHMA`
- DVHMA-OpenUI v.6.3.0
  `https://github.com/logicalhacking/DVHMA`
- Sieve v.2.3.4

These applications allowed us assess some features of our analysis were working properly.

For instance, we knew that 'DVHMA-OpenUI' was built on Apache Cordova and we wanted to check if our framework was able to correctly detect this.

4.1 OWASP Methodology

This methodology is based on OWASP's Top 10 mobile application vulnerabilities of 2016 [7], which many developers use as the standard source for information on how to test the security of mobile applications.

Important areas we need to analyse in a mobile application to evaluate its overall security level are [5]:

M1 Improper Platform Usage
M2 Insecure Data Storage
M3 Insecure Communication
M4 Insecure Authentication
M5 Insufficient Cryptography
M6 Insecure Authorization
M7 Client Code Quality
M8 Code Tampering
M9 Reverse Engineering
M10 Extraneous Functionality

We ran test applications we selected through the DroidSF framework to identify potential vulnerabilities and obtained the following results:

	M1	**M2**	**M3**	**M4**	**M5**	**M6**	**M7**	**M8**	**M9**	**M10**
InsecureBank	V	X	V	X	X	X	X	X	V	X
PIVAA	V	X	V	X	V	X	X	X	V	X
DVHMA-FeatherWeight	V	X	X	X	X	X	X	X	V	X
DVHMA-OpenUI	V	X	X	X	X	X	X	X	V	X
Sieve	V	X	V	X	V	X	X	X	V	X

Table 1. DroidSF findings - April 2019
Detected vulnerability: V, No vulnerability found: X

4.2 Limitations

Around 98% of Android mobile devices use ARM CPUs [12]. Due to this fact, a growing number of developers choose not to include native libraries compiled for x86/amd64 on the APK, which effectively prevents their applications from being executed natively on x86/amd64 CPUs. Android will even refuse to install the application if it does not match system's architecture where it is running.

Because emulating the ARM architecture on x86/amd64 CPUs introduces severe performance losses, we conducted our tests using a AVD emulator running a x86_64 Android ROM.

If developers do not possess a rooted ARM Android device, their only option is to configure an AVD emulator to use an Android ARM ROM and work through

the very slow emulation process over x86/amd64, while hoping that none of the processes crash.

Another limitation in our framework has to do with interacting with the target application when running the instrumentation checks. We can setup hooks to methods and classes, but we will not necessarily see them being called. This happens because application's logic flow might not be executing these methods

DroidBot [3] performs an exhaustive and systematic interaction with the application, but this still does not guarantee that its interaction with the application will trigger the hooked functions we wanted to inspect. Ideally one could trace APIs and system calls in an attempt to identify the call stack required to reach the desired functions, but the process is not consistent and can be very time-consuming.

5 Conclusions

The security testing of mobile applications is a relatively new area, and it has proved to be a very interesting and challenging area due to its dynamic and ever-evolving ecosystem.

We feel it is of a tremendous importance to maintain mobile devices secure given the importance of the data and functionality they hold. To this end, we understand the purpose behind the thorough scrutiny that official application stores do on every application submitted.

While working on this thesis we understood how to leverage existing tools to perform the kind of analysis an official application store does to submitted applications. Of course, we do not know for sure which checks Google, Apple and other companies employ to detect vulnerable applications, but we are confident we took a step in the right direction.

The sheer amount of information we found on topics such as Android platform, Vulnerability detection, Malware detection, Reverse Engineering, etc., was enormous. One of our biggest challenges was to filter out which information was relevant to this thesis.

It became clear to us during our research that, no matter how complex certain security features are, bad actors will always try to find new ways to circumvent them. Even though the Android documentation presents many standard security practices for developers to follow, there is always room for human error and this just reinforces the need to use an automated testing framework that can identify problems before the application is deployed.

We feel that we were successful in creating a flexible framework built around robust multi-platform software, that can be extended to perform very complex tasks. We also believe that most developers, from students to expert security testers, will find the DroidSF framework very useful to conduct multi-prone analysis on Android mobile applications. It still has many areas that require improvements but we are satisfied with what we built so far.

References

1. André, C.: droidstat-x - android applications security analyser. https://github.com/clviper/droidstatx, accessed: 27/04/2018
2. Egele, M., Scholte, T., Kirda, E., Kruegel, C.: A survey on automated dynamic malware-analysis techniques and tools. ACM computing surveys (CSUR) **44**(2), 6 (2012)
3. Li, Y.: droidbot - lightweight test input generator for android. https://github.com/honeynet/droidbot, accessed: 25/03/2019
4. Mueller, B.: Apkx - android apk decompilation for the lazy. https://github.com/b-mueller/apkx, accessed: 27/01/2019
5. OWASP: Mobile top 10 2016. https://www.owasp.org/index.php/Mobile_Top_10_2016-Top_10, accessed: 06/04/2018
6. OWASP: Tampering and reverse engineering on android. https://mobile-security.gitbook.io/mobile-security-testing-guide/android-testing-guide/0x05c-reverse-engineering-and-tampering, accessed: 25/09/2018
7. OWASP: Welcome to owasp. https://www.owasp.org/index.php/Main_Page, accessed: 06/04/2018
8. Patnaik, N.D.: Appmon - automated framework for monitoring and tampering mobile applications. https://github.com/dpnishant/appmon, accessed: 28/04/2018
9. Qiu, L., Wang, Y., Rubin, J.: Analyzing the analyzers: Flowdroid/iccta, amandroid, and droidsafe. In: Proceedings of the 27th ACM SIGSOFT International Symposium on Software Testing and Analysis. pp. 176–186. ACM (2018)
10. sensepost: objection - runtime mobile exploration. https://github.com/sensepost/objection, accessed: 28/04/2018
11. StatCounter: Os worldwide market share. http://gs.statcounter.com/os-market-share/mobile/worldwide, accessed: 09/04/2019
12. Unity3D: Android hardware stats. https://web.archive.org/web/20170808222202/http://hwstats.unity3d.com:80/mobile/cpu-android.html, accessed: 04/04/2019