MAD Spy Report

Cristian Turetta and Andrea Perazzoli

Abstract—We developed a malware for educational purposes. In particular, our goal is to provide a PoC of what is known as a *Repacking attack*, a known technique widely used by malware cybercrooks to trojanize android apps. The answer to solve this particular goal boils down in the simplicity of *APK decompiling* and *smali code injection*.

1 Introduction

In this report we will explain how we have created the **spyware** for Malware course's project. The aim of this project is to inject malicious code into a messaging app, such as Snapchat, Whatsapp, Skype, Messenger, Hangouts. In our case we have selected **Whatsapp** but our modules can be injected even in the others Android applications because it does not depend from the target application itself.

In order to infect the target device, we assume that the control of the network which is connected the target device is under our control, this allow us to use DNS hijacking which is a practice of subverting the resolution of Domain Name System (DNS) queries. Thus, with this technique we are able to send fake notifications containing Whatsapp's updates.

This work is organised as follows: in Section 2 we provide a description of the project structure and classes implementation in order to achieve the desired *spyware* behaviours and how we have build up the malware. In Section 3 we compare the original application to the repacked one which contains the *spyware*, and we try to figure out and evaluate its stealthiness analysing CPU, RAM and Network overloads. In Section 4 we talk about the conclusions figuring out the obtained results and the possible improvements.

2 Project Implementation

Before starting the implementation of the *malicious* classes we have assumed that the target device is running Android with **root privileges** enabled. As result we can grant the required privileges without user interaction in order to increase its stealthiness. This job is done by the class `StartupIntentService` which is triggered on the main activity of the target *apk* to get all required privileges.

The **spyware** is able to identify when the specific chat program is started, record the keystrokes to get the messages/password, take screenshots and send all the collected information to a database under the attacker's control. In the project implementation we have separated the **spyware** behaviour using Android **Service** for routines such as key logging, timed screenshots and data exfiltration. Android services fit perfectly thanks to their design, in fact they are not bind with the activity and they are unique, this means that we can not have two or more services of the same type. Services runs on main thread and in order to avoid exceptions like `NetworkOnMainThreadException` we have created threads triggered on services. Data exfiltration were achieved by using a Firebase database.

2.1 Spy Modules

The *malicious* classes of the *Spy malware* are designed in *divide et impera* fashion. The aims of this *malware* are the following:

- Recording the **keystrokes**, which means that we can retrieve passwords, written text, launched applications etc.
- Take **snapshot**s of the screen, in order to get some extra informations.

Retrieve **keystrokes** was possible thanks to Android **Accessibility Service**, in fact enabling this service we were able to catch every kind of interaction between user and smartphone, not only the keyboard key logging but also navigation interactions and so on. In order to do this task we have coded one class called `Keylogger`, this class extends `AccessibilityService`. This class implements the method *onAccessibilityEvent* which takes as parameter an `AccessibilityEvent`, switching by the type of this event we were able to understand and register what the user is doing.

Take **snapshot** is a routine running in background out of the main thread of the application, it does not depend on the activity that starts this routine. In order to implement this kind of feature we have created three classes with different purposes:

- `ScreenshotUtil` is a *singleton* class which contains the method *shoot* to get the snapshot.
- `ScreenshotUtilThread` is the thread which represent the routine, it trigger the screen capture once every t[1] time.
- `ScreenshotUtilService` is a class that extends Android `Service`. When the application is launched this service starts, this service is unique and when it stars it is invisible to the user. This class instantiate an object of the previous mentioned class to take screen captures.

Every I/O to files or interactions with device file system is done using `FileUtil` class[2].

1. Established in `ParameterConfig` class.
2. We used internal storage to prevent MiTD vulnerability.

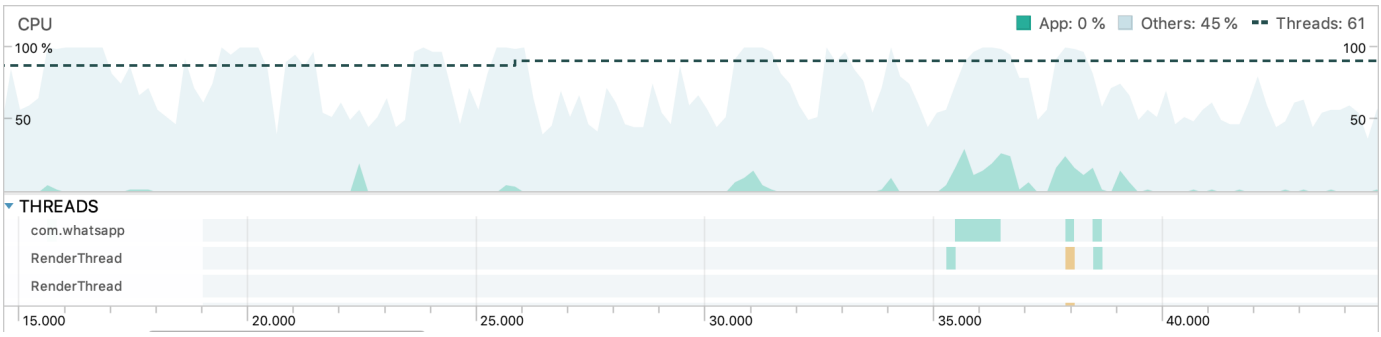

Fig. 1: Original Whatsapp CPU usage

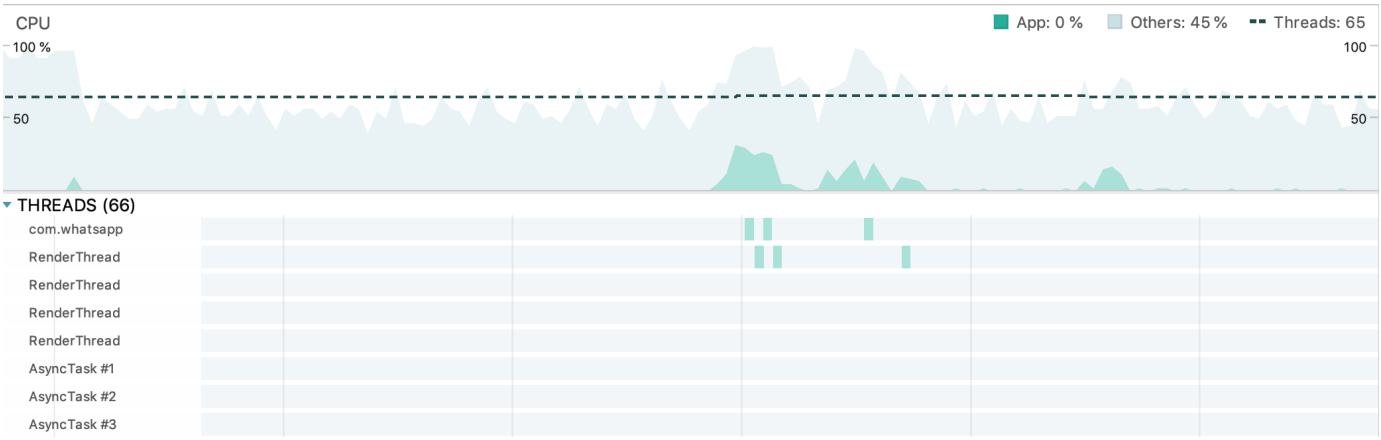

Fig. 2: Packed Whatsapp CPU usage

2.2 Data Exfiltration

Recorded data are saved locally on the device memory, every *t* time they were sent to a remote Firebase database in order to be viewed and used by the attacker, using POST web APIs.

The Firebase database is organised with the following scheme:

- **users** which is the root directory.
- **identification number** represent a user, it is unique and it is provided by Google account used on the user's smartphone.
- **timestamps** inside each **identification number** directories contain the fetched user data recorded in a period of time.

Briefly, in order to retrieve specific data the path to follow is **users/identification number/timestamp**.

This routine is performed by the usage of three different classes `Spy`, `SpyThread` and `SpyService`, the logic behind the implementation of these classes is the same of classes used to take snapshots, mentioned in 2.1.

2.3 Piggybacking

Reverse engineering in computer programming is a technique used to analyse software in order to identify and understand the parts it is composed of, usually to recreate the program, to build something similar to it, to exploit its weaknesses or strengthen its defences.

In our case we have used reverse engineering to inject malicious modules in order to steer the normal flow of the target application, in particular modifying the initialisation process.

The first step, in order to create the repacked application, is decompiling the target *apk*, obtaining its `smali` code which is byte code version of Java code written in Android apps, it is similar to Assembly language. This process is also known as **backsmaling**, the same thing is done with our malicious modules.

The second step is the injection, having the `smali` code of the target application and the smali code of our malicious modules we have to merge it, this phase is accomplished once the malicious `smali` code is inserted into *com* directory of the decompiled target and modifying its manifest by register the malicious services and by put the `smali` code that triggers these services inside the *onCreate* method of the target **Main Activity**, replacing the caller to starting service routines with *LCom/whatsapp/Main*.

The final step is recompilation and signature, once modified the target code it is time to recompile it in order to get the repacked *apk* and sign it to accomplish the piggybacking phase. At the end of these steps we obtain a working application ready to be deployed.

During the **piggybacking** phase we have used `apktool` features for decompile and compile the `smali` code and `jarSigner` to sign the repacked *apk*.

3 Evaluation

In this section we will measure the stealthiness of the packed app. In particular, we will get an insight of *CPU*, *memory* and *data usage* in comparison to the non packed one.

3.1 CPU usage

The difference between the cpu usage is minimal, if any. Background services requires a little power to run due to optimization and minimal implementation.

3.2 Memory usage

As we can see from the pictures, our malware has a very little footprint in the target phone, since the services are really lightweight: a normal user cannot recognize the malware presence using only memory usage data.

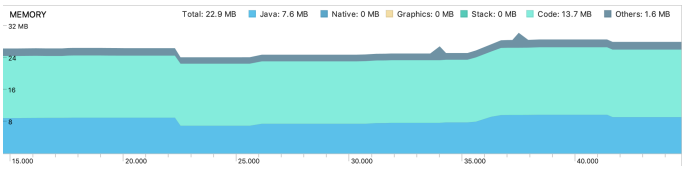

Fig. 3: Original Whatsapp memory usage

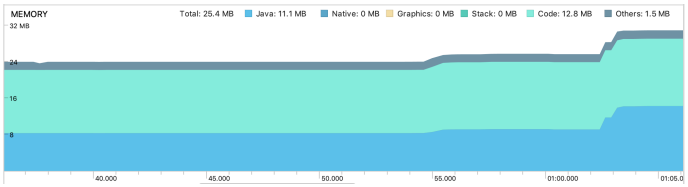

Fig. 4: Packed Whatsapp memory usage

3.3 Data Usage

We notice a little peak every minute, caused by data exfiltration. Traffic like this could be easily be interpreted as the app normal behaviour. The data are *HTTPS encrypted*, so absolutely unreadable by a sniffer, plus they can easily be confused by Whatsapp native Firebase communication.

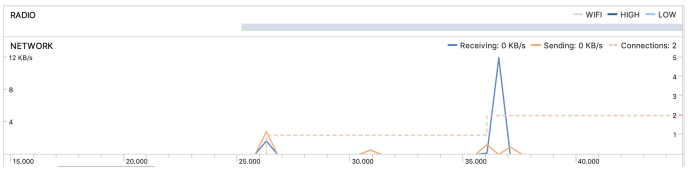

Fig. 5: Original Whatsapp data usage

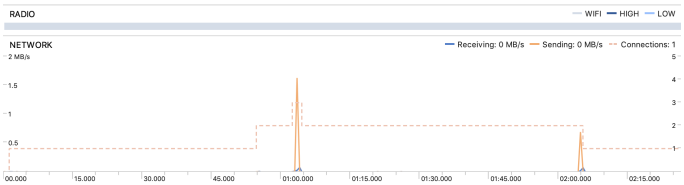

Fig. 6: Packed Whatsapp data usage

4 Conclusions and Improvements

In our research we build a modular, thread-safe and simply malware. Every parameter is extremely customizable and easy to work with. In order to work, the malware should run in a rooted environment: we assume this requirement, because an exploit development would have gone beyond educational purposes.
We should improve its stealthiness using ProGuard Obfuscation, an additional layer against APK decompiling.