University of Padova

Department of Mathematics "Tullio Levi-Civita"

Master Thesis in Computer Science

Security comparison between Xiaomi system applications and Xiaomi applications on the Google Play Store

Supervisor
Prof. Eleonora Losiouk
University of Padova

Master Candidate
Michele Agnello

Student ID
1238581

Academic Year

2021-2022

“Programming isn’t about what you know; it’s about what you can figure out.”
— Chris Pine

Abstract

Whenever a new smartphone that relies on Android as its OS comes out, vendors such as Xiaomi, Samsung, Motorola, Huawei can customize the phone with preinstalled applications with vendor-specific functionalities. System applications require root permission to be uninstalled, and they also need to be signed with the same key as the smartphone ROM that has been signed. Such applications can also be found on the Google Play store, which can guarantee the security of the app. In this thesis we are going to compare these two types of the same app to find out if one behaves differently than the other or if an application has some hidden functionalities or vulnerabilities with respect to the other type. Xiaomi smartphones will be the test subject for this research, since recently there have been a lot of blog posts and news on how the company is tracking and recording user's private information.

Contents

Abstract v

List of figures ix

Listing of acronyms xi

1 Introduction 1

2 Background 3
2.1 Android Architecture . . . 3
2.1.1 Linux Kernel . . . 5
2.1.2 Hardware Abstraction Layer . . . 5
2.1.3 Android Runtime . . . 5
2.1.4 Native C/C++ libraries . . . 6
2.1.5 Java API Framework . . . 6
2.1.6 System Applications . . . 6
2.2 Android Applications . . . 7
2.2.1 Applications components . . . 7
2.3 Android Manifest . . . 12
2.4 Android Permissions and Security . . . 12
2.4.1 Install-Time permissions . . . 13
2.4.2 Run-Time permissions . . . 13
2.5 Permission Usage . . . 13
2.6 Protection Level . . . 14

3 Related Works 15
3.1 Analysis of Pre-Installed Android Software . . . 15
3.2 FirmScope . . . 16
3.3 User's data collection articles . . . 17
3.3.1 Xiaomi Devices Tracking . . . 17
3.3.2 Xiaomi Bug Bounty Program . . . 17
3.3.3 Xiaomi Privacy Policy . . . 18
3.3.4 Logging Contact-Tracing Data . . . 18

4 Dataset Collection 19

4.1 Xiaomi Phones . . . 19
4.2 Applications Collection . . . 19
4.3 Related Obstacles . . . 20

5 Tools employed 23
5.1 Androwarn . . . 23
5.2 Maldrolyzer . . . 24
5.3 RiskinDroid . . . 24
5.4 SUPER . . . 25
5.5 StaCoAn . . . 25
5.6 Quark Engine . . . 26
5.7 Apkleaks . . . 26
5.8 Frida . . . 26

6 Design 29
6.1 Preliminary analysis . . . 29
6.1.1 Static Analysis . . . 30
6.2 Privileged Permissions Discovery . . . 31
6.2.1 Static Analysis . . . 32
6.2.2 Dynamic Analysis . . . 34

7 Implementation 35
7.1 Preliminary analysis . . . 35
7.1.1 Static Analysis . . . 35
7.2 Privileged permissions Analysis . . . 36
7.2.1 Static Analysis . . . 36
7.2.2 Dynamic Analysis . . . 37

8 Results 39

9 Conclusion 41

References 43

Acknowledgments 45

Listing of figures

2.1 Shows the android architecture stack 4
2.2 Shows the activity lifecycle 9
2.3 Shows how content providers manage access to storage 11
2.4 shows the workflow to follow for using app permissions 12

6.1 List of the partitions found on the phone, in particular product, system and vendor 31
6.2 Shows how privileged permissions get declared on the XML file 32

Listing of acronyms

AOSP .... Android Open Source Project

IPC .... Inter Process Communication

HAL .... Hardware Abstraction Layer

AOT .... Ahead Of Time (Compilation)

DEX .... Dalvik Executable

API .... Application Programming Interface

NDK .... Native Development Kit

SDK .... Software Development Kit

ADB .... Android Debug Bridge

XML .... eXtensible Markup Language

ROM .... Read Only Memory

1 Introduction

The Open Source Android system is one of the projects developed by Google. Android is an operating system that is used mostly to power mobile devices, and its code is entirely open source. This means that everyone can read or customize this operating system to meet their needs. For example, big tech companies such as Xiaomi developed a customized ROM based on android called MIUI, and this ROM now powers most, if not all, of their mobile devices. One of the advantages of customized ROMs is that they can have pre-installed applications which can enhance user experience or make things easier for the end user. On the downside, these apps can also have inside ads, or can easily be a trial version of the final app that you have to buy on google play, in this case such applications are called bloatware, because instead of making things easier for the user, they can take up space, reduce battery life, and cripple the smartphone performance. What is interesting though is that these same pre-installed application can be found in the google play store, and can be downloaded and even installed on a smartphone developed by a different vendor. So what we asked ourselves was what are the differences between third-party apps found on the Play Store and pre-installed apps found on the smartphone? Do they behave differently? Do they have different source code? But the most important question of all was do they have different sets of permissions that can make the application run in a different state? In this thesis i am going to walk you through the in-depth analysis of these applications, how they behave and if they run the same way or expose different privileged states.

The thesis is organized as follows:

- Chapter 2 contains a brief overview of the android operating system and a detailed description of android application's permissions
- Chapter 3 explores some of the reviewed research works concerning static and dynamic analysis of the android application
- Chapter 4 explains the major differences between system and third party apps and also describes how the apps to analyze were retrieved
- Chapter 5 describes the procedure that was followed in the analysis of these applications, in particular which app features we needed to focus on for the comparison
- Chapter 6 describes the tools used for static and dynamic analysis of the applications
- Chapter 7 exposes the results found by this analysis and describes some critical issues brought up by the analysis procedure
- Chapter 8 concludes the thesis, also exposing some of the problems encountered during the research

2

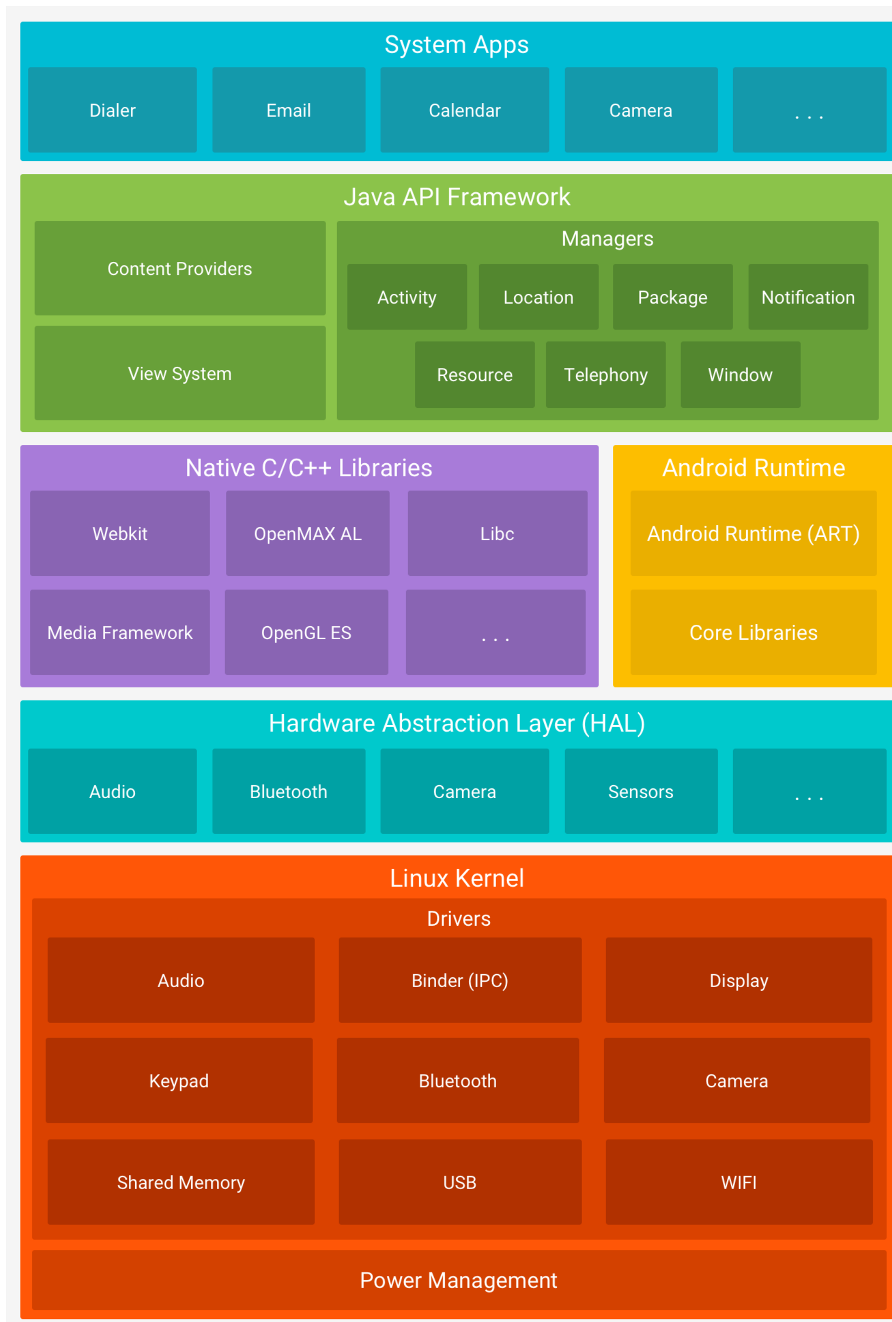
Background

In this chapter, the Android platform [1] is presented. In section 1.1 we describe briefly how the android architecture is composed; In section 1.2 we introduce the Android OS and its core components; Section 1.3 gives a description of the Android manifest file;In section 1.4 we introduce the concept of permissions in Android, how they are used and how they work, plus an overview of the Android security mechanisms. The objective is to expose this thesis' research context to the reader.

2.1 Android Architecture

The Android platform can be described as a stack of different components, we have an operating system, a middleware, and system applications. The middleware, which is the main part of the stack, runs on top of the Linux Kernel and hardware drivers, allowing it to take advantage of security features, libraries, runtime environments and application frameworks also giving device manifacturers the ability to develop specific hardware drivers.

Figure 2.1: Shows the android architecture stack



2.1.1 Linux Kernel

The Android platform provides the security of the Linux kernel, as well as secure interprocess communication (IPC) to enable secure communication between applications running in different processes. The Linux kernel is used in millions of security-sensitive environments, and has it provides Android with several key security features like:

- A user-based permission model
- Process isolation
- Extensible mechanism for secure IPC
- The ability to remove unnecessary and potentially insecure parts of the kernel
- Filesystem permissions
- Verified boot

The Linux kernel is also a multi-user OS and is able to deny access to user resources by a different user; for example, it guarantees that user A does not exhaust user B's memory, CPU resources or Devices.

2.1.2 Hardware Abstraction Layer

The hardware abstraction layer (HAL) is an interface that lets the android system services communicate with the linux kernel and vice versa. To make this communication possible, this layer defines a standard interface that hardware vendors can implement and personalize without changing the higher-level system. Vendors need to implement the specified HAL and driver for the corresponding hardware component, these HAL implementations are then built into shared library modules that are then loaded whenever an API call tries to access the specified hardware.

2.1.3 Android Runtime

The Android Runtime is the virtual environment that is used by android applications and services on Android. Its predecessor, which was used before Android 5.0, is called Dalvik and is also a runtime that was created specifically for Android. These tools were created to run Dalvik executable written in the DEX format, which is a bytecode optimized for low memory

footprint. Android applications written in Java are compiled in DEX bytecode that will run on the Android platform through the ART. Some of the main features of ART include the following: Ahead-of-time (AOT) compilation that can improve the performance of applications; improved garbage collection; development and debugging improvements. Android can also provide most of the functionality available in the Java programming language, for example, Java 8 features like lambda expressions, method references, type annotations, etc., through the Android Gradle Plugin.

2.1.4 Native C/C++ libraries

The Android platform provides a set of tools that allow developers to insert C and C++ code into android applications. This tool is called the Native Development Kit (NDK) and can also provide platform libraries to access physical device components and native activities. Through NDK developers can achieve low-latency or computationally intensive applications or use external C, C++ libraries. Most of the components and services of the Android system, such as ART and HAL, are built from native code that requires native libraries written in C and C++.

2.1.5 Java API Framework

All the features available in the Android platform are exposed to the developer through APIs written in the Java programming language. The usage of these APIs simplifies the creation of Android apps through the reuse of core, modular system components, and services. For example, such APIs offer a View System to build the UI of an application, a content provider where applications can access data from other apps, an activity manager to manage the lifecycle of the application, and many more. Moreover, Android developers can also take advantage of framework APIs used in Android System apps.

2.1.6 System Applications

The Android OS also has a set of pre-installed applications for various functions such as email, SMS messaging, calendars, browsing the Internet, contacts, etc. These applications are called system apps, but they do not have a special status among other applications that the user chooses to install. In particular, an application downloaded from third parties can become the user's default email manager, web browser, or even SMS messenger. Of course, there exist some exceptions, such as the system's settings app. System applications also provide functionalities

that developers can access from their own application. For example, if your app wants to visit a web page, you don't need to build a new web browser app, you can instead invoke the system browser app to visit a web page.

2.2 Android Applications

To write Android applications, you can choose between Java, C++ or Kotlin, the new language developed by JetBrains. The Android Software Development Kit (SDK) will then compile all the code together with any external data or resource files to create either an APK or an Android App Bundle. An APK is a simple archive file with the extension.apk, contains all the files that the application needs at runtime and is also the file needed by android devices to install the application. Every Android app runs in its own virtual environment or sandbox, and it is isolated from other apps. Furthermore, the system identifies every app as a different user and assigns each and every app a unique ID, so that all files in an application can be accessed by the user ID assigned to that application. By default, each android application can only access the components that it needs to work and nothing more. This principle is called the least privileged and is implemented by the Android system. This principle guarantees a secure environment where an application can access only system parts for which it has been given permission; however, there are ways for applications to share data or access system services. If two apps share the same user ID they can access each other's files and can also run on the same sandbox or Linux process. To use this feature, the apps need to be signed with the same certificate. Also, if the user explicitly grants the specific permission, the application can access device hardware such as device's location, camera, and Bluetooth.

2.2.1 Applications components

Android applications are built from components, which are entry points for the system or the user to access the application. There are four different types of app components: activities, services, broadcast receivers, content providers. These components can be activated by messages called intents.

Intents

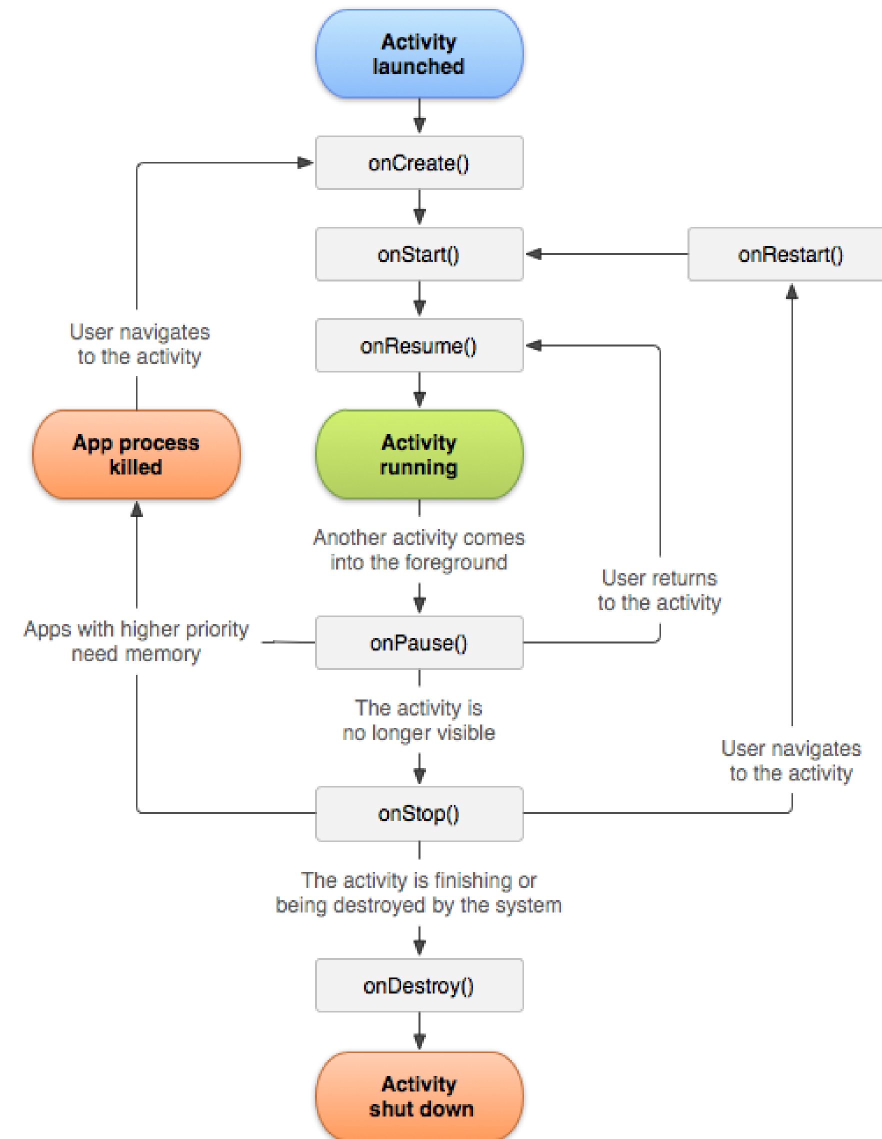
Intents are asynchronous messages that can activate app components. At runtime, intents can connect different app components and request action from an app component. A developer

can create an intent from the Intent class, and decide if the intent is implicit or explicit. Implicit intents do not activate a specific component; instead, they declare a specific action to perform. Explicit intents target the specific application or a fully-qualified component class name and are typically used to activate a new activity inside an application. Depending on the component type, an intent can behave in different ways. For activities and services, an intent defines the actions to perform, like read, send, update, and also specifies the data to act on with an URI. For broadcast receivers, the intent can specify the notification being broadcast, for example, the low battery notification. Content providers are instead not activated by intents. To receive an implicit intent, the developer needs to insert an intent filter in the manifest file specifying the accepted intent type.

Activities

Activities are the building blocks for developing an Android application; in fact, the activity class is the entry point for the interaction between user and application. Whenever the user opens an application, an activity gets invoked rather than the application as an atomic whole. Also when the app itself wants to interact with another app it invokes one of the other app's activities, this is because in the mobile-app world a user does not always interact with an application in the same way, as opposed to desktop application. For example, if a user can open the email app to view all his emails, but it can also use another application that launches an activity pointing to the same email app. A developer can implement an activity by creating a class inheriting from Activity, and the application can have multiple activities interacting with each other. Activities provide a window where the application's UI gets drawn; usually an application has at least one activity called main activity which is the first screen that appears when the user launches the application. When the user of an application starts to move between activities, such activities transition through different states of their lifecycle. An activity lifecycle is composed of different states that can be traversed by implementing the system's core callbacks: onCreate(), onStart(), onResume(), onPause(), onStop(), and onDestroy().

Figure 2.2: Shows the activity lifecycle



Services

A service is a component of an Android application that can run in the background and perform time-consuming operations. It doesn't have a UI and once started it continues to run even if the user opens another application. An application component can easily interact with the service through IPC; in this case the component is said to be bound to such service. In Android, there exist three types of service:

- Foreground, this service performs operations that are visible to the user, in fact, it is forced to display a notification visible to the user to remind him that the service is running. When this service is stopped, the notification can be canceled.

- Background, on the other end this service performs operations that are not directly visible to the user.

- Bound, this type of service is generated when an application calls the bindService() method and binds to this service. The application can then begin a client-server interaction with the service, so it can exchange data by sending requests and receiving responses, and it can do so also within multiple processes. The service life span terminates when it has no other components that bind to it.

The advantage of using a service is that the developer can create a service that is a mix between the aforementioned types. For example, it is possible to create a service that runs indefinitely in the background and that can be bound to. It depends on the implemented callback methods: onStartCommand(), onBind(), onCreate(), onDestroy().

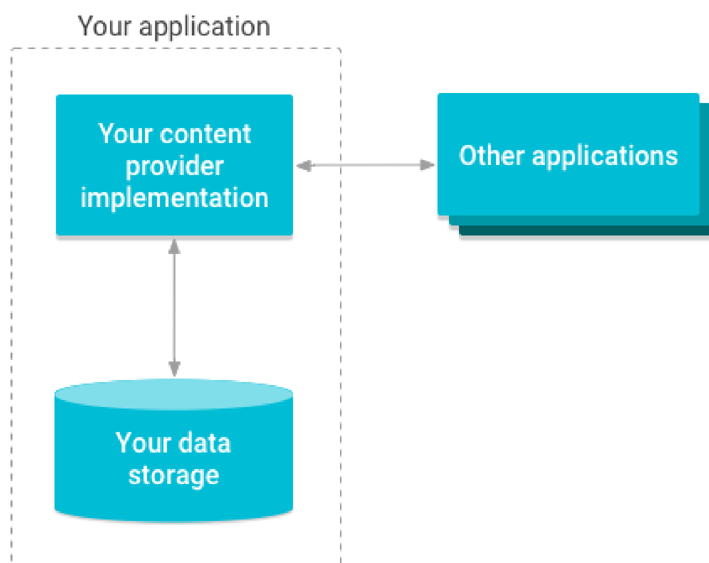
Broadcast Receivers

Applications on android are able to send and receive broadcast messages to and from other applications. These messages can also be customized by an application in order to notify a target app. In order to receive the message, apps need to register a specific broadcast receiver, so that when the broadcast message is sent the system routes the messages to those applications who subscribed to that particular receiver. Android applications can receive broadcast messages by declaring a receiver in their manifest or by registering with context-registered receivers. In the first method, the developer declares a receiver in the applications manifest, and implements onReceive(...) callback method of the class BroadcastReceiver. Then the system registers the broadcast receiver at install time and that receiver becomes a separate component the system can run to receive the message, even if the application is not currently running. In the latter method the developer does not need to declare a receiver in the manifest, but he needs to create an instance of BroadcastReceiver, create an IntentFilter, and then register the receiver implementing the registerReceiver(...) method. Context-registered receivers have a lifespan depending on the context in which they are registered, if they are registered in an Activity they will keep receiving messages as long as the activity is not destroyed, if they are registered in the application context they will receive messages until the app is closed.

Content Providers

Content providers are components that can share data between applications. The shared data can be stored in the phone's file system, in an SQLlite database, on the web, or any other persistent storage that applications can access to. A content provider is able to encapsulate data and provide a standard interface where applications can securely connect, read, or modify such data. Tipically, developers work with content providers in two ways: they need to access the content provider of another app or they need to create a content provider to share data between applications. In the first method, they need to create an object of the class ContentResolver to communicate with the provider much like a client-server communication. The resolver methods that the developer needs to implement offer the basic CRUD functions to operate on persistent data. Once implemented, the resolver will communicate with the contentprovider that will perform the required action and responds with some results. In the second method developers need to create a class that is a subclass to ContentProvider, they will need to decide if the stored data is data that normally goes into files or structured data that, for example, can go into a database. Thay will also need to define a content URI that identifies the data in the provider. Content URIs contain a symbolic name of the entire provider (its authority) and a name that points to a table (the so-called path). The content resolver will then parse the URI to resolve the access to the data. The provider must also be defined in the application's manifest file.

Figure 2.3: Shows how content providers manage access to storage



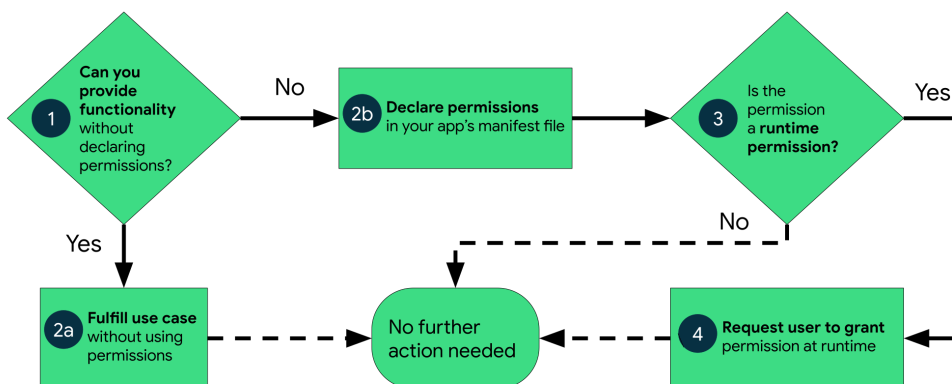
2.3 Android Manifest

The Android system needs to know that an app component exists before starting it, this is done through the AndroidManifest.xml file. In this file all the app's components are declared, all the app's permissions are specified, the minimum Android API level required by the app is declared, the app's hardware and software features are inserted, and all API libraries the app needs are specified. The components are declared by specific tags (<activity>, <service>, <receiver>, <provider>), if they are not present in the manifest, they will never be visible to the system. In the manifest we can define even the various component' capabilities by including intent filters in the component's tags. The app features are instead declared by the <uses-feature> tag.

2.4 Android Permissions and Security

When an Android application requires access to restricted data or actions that are considered restricted, this application needs to have the appropriate permissions. There are three main types of android permissions, Install-time permissions, granted automatically when the app is installed; Run-time permissions, the app prompts the user to grant the permission at runtime; Special permissions, these are associated to particularly powerful type of actions, and they can only be defined by the platform and the OEM. Each special permissions has its own implementation details.

Figure 2.4: shows the workflow to follow for using app permissions



2.4.1 Install-Time permissions

When install type permissions are declared in the application, the system automatically grants these permissions when the user installs the app. When the user wants to install an application from an app store, the store notifies to the user about the install-time permission the app needs. The permission included in this type can also be subdivided into two categories such as normal permissions and signature permissions. The first sub-type includes permissions that allow access to data and actions outside of the application range, or sandbox. Signature permissions instead are defined by another application; if the application defining the permission and the application using the permission are signed with the same key, the system grants the permission at install time to the app using the permission.

2.4.2 Run-Time permissions

These permissions are also called dangerous permissions because they give applications access to sensitive data and grant applications the ability to perform actions that can affect the system and other applications. Before accessing restricted data, the application notifies the user, and he can grant or deny this run-time permission. For example, accessing the microphone or camera requires run-time permission because the application can access sensitive information.

2.5 Permission Usage

Each permission in the android architecture is identified by a unique label. This label needs to be included in the manifest file of the application.

Listing 2.1: Syntax used to declare permission in manifest file

```
<manifest ... >
   <uses-permission android:name="android.permission.SEND_SMS"/>
   ...
</manifest>
```

Starting from Android 6.0 the user is able to approve or reject some run-time permissions, if the permission is granted the application is able to use the restricted features, if the permission is rejected the app will fail to use those features. An application can also define its own permissions, but in this case it will declare the permission with the ”<permission>” tag.

2.6 Protection Level

Every permission in the android system has a so-called protection level assigned by the OS. This protection level is a tag written with ”android:protectionLevel” that signals the potential risk attributed to the permission and exposes the procedure the system should follow when choosing between granting or denying the permission to the requesting application.
Each protection level is a combination of one base type and zero or more flags. For example, every non-custom permission has a base type of ”normal” and no other flags, while the protection level of ”signature|privileged” is a combination of the signature base type with the privileged flag. The base type for protection level are:

- ”normal”, this is the default value, it gives applications access to isolated application-level features with minimum risk to other apps. Usually this level is automatically attributed to install-time permission, without asking for the user's approval;
- ”dangerous”, this is attributed to higher risk permissions that can give the application access to sensitive user data or control over the device that can impact negatively the user. In this case the permission will not be automatically granted at the requesting application; instead it may ask for the user's approval to accept the usage of the dangerous feature;
- ”signature”, this means that the application requesting the permission has to be signed with the same certificate of the application that declared the permission. If the certificates match, the permission is granted without notifying the user.
- ”signatureOrSystem” (”signature|privileged”), in this case, the system grants the permission to applications that are either in a dedicated folder of the android system image or signed with the certificate of the app who is declaring the permission. This level of permission is used for certain special situations where multiple vendors have applications built into a system image and need to share specific features explicitly because they are being built together.

3
Related Works

This chapter reviews some of the research work which helped to gather data for the confrontation of applications. In particular the first two papers analyze in detail pre-installed applications both in Android Software and Android Firmware. In the final section, the chapter exposes a series of blogs that support the reason for this research project. More than the tools presented and the results found in those papers, we were interested in how the researchers gathered data and applications and how to consider if an application is a pre-installed Android app or not. This part was crucial because we needed to be sure that the pre-installed apps and the same applications gathered from the Google Play store had to be comparable, both in code and in behavior. In section 2.1 we describe the paper "An Analysis of Pre-installed Android Software"; In section 2.2 instead we describe the tool FirmScope. In section 2.3 we present a series of blogs that focus on the collection of user data by Xiaomi applications.

3.1 Analysis of Pre-Installed Android Software

In J. Gamba et al. [2] the objective of the authors is to explore the environment of pre-installed android software in terms of privacy and security. The authors present a large-scale study of software from more than 200 vendors collected through crowd-sourcing methods. Through this study, we can analyze the involvement of different actors in what the authors call the supply chain, which is the collaboration of different actors, for example, device manufacturers, mobile network operators, and third party organizations in the making of the final product.

The questions that the authors aim to answer are: what is the ecosystem of pre-installed apps?; what is the relationship between vendors and stakeholders?; Pre-installed apps private and personally identifiable information? And if so with whom they share it?; Among these applications, are there any harmful or potentially dangerous apps?. The study consists of the following four main steps: data collection, ecosystem overview, permission analysis, and behavioral analysis. In the first step, applications and traffic information are collected from real-world devices. This dataset includes apps gathered from different users, device models, and more than 200 vendors. Moreover, the authors included traffic flows from different applications using the Lumen app. Subsequently, all the ecosystem is investigated by analyzing the applications' manifest files, their certificates, and the third-party libraries used. In the next step, a set of custom permissions is extracted and analyzed. These permissions could be used to escalate the android permission model and access privileged system resources. The final step consists of static and dynamic analysis of these applications through different tools to check for vulnerabilities or unwanted behavior. As mentioned before, we consulted this study mainly to understand what pre-installed applications are and how they can be recognized, how to collect data, and what tools are best suited for our analysis.

3.2 FirmScope

Firmscope [3] is a tool capable of detecting privilege escalation vulnerabilities in pre-installed Android applications in automatic. This tool can scan an Android firmware for vulnerabilities mainly in two steps:

- Pre-processing, in this phase the tool gathers the Android firmware to scan and start to unpack the individual file-system images contained within it. Then it extracts all the system apps contained in an image file, checks each app's manifest and metadata, exported components and finally starts to disassemble its DEX files;
- Static Taint Analysis, in this phase FirmScope starts to build an interprocedural control flow graph (ICFG) for every app, this graph represents control flow that transfers from target method from the same class or even different apps. After building the graphs, the tool reconstructs the heirarchy of the classes and resolves all calls, it then infers def-use chains and builds the interprocedural data flow graph (IDFG), and finally it performs the taint analysis to identify execution paths that are vulnerable.

As in J. Gamba et al. [2] presented in Section 2.1, we decided to use Firmscope [3] to help us understand how to recognize and gather system apps and what are privileged permissions and

privileged apps.

3.3 User's data collection articles

3.3.1 Xiaomi Devices Tracking

The article C. Singh [4] exposes the findings of a cybersecurity researcher named Gabriel Cirlig, who accused the Xiaomi browser, Mi Browser Pro, of collecting all search queries and items viewed by users, the collection happens even in incognito mode.The researcher was also able to prove this behavior even for other Xiaomi phones, including Mi 10, Redmi K20, and Mi MIX 3. Xiaomi responded to this claim and justified their action saying "collection of anonymous browsing data is one of the most common solutions adopted by internet companies.". However, Cirlig claimed that if the information collected from the browser is coupled with phone's "metadata" collected by Xiaomi, the company is able to identify the user. Moreover, Cirlig was also able to notice the monitoring of his touches on his screen and even the collection of his listening abits by the System Music App. The researcher discovered that the collection of the data was connected with SensorDataAPI, an API that enables third-party access to the application's data. This third party used by Xiaomi is Sensor Analytics, a startup that deliberately tracks users. Xiaomi even responded to this claim, saying that the data collected by Sensor Analytics is anonymous and safely stored in their servers.
Despite the responses to these claims, Xiaomi has released updates for its Mi browser and Mint browser that include a toggle to disable the collection of aggregated usage data in incognito mode [5].

3.3.2 Xiaomi Bug Bounty Program

In the Xiaomi Bug Bounty Policy [6] Xiaomi presents its bug bounty program, specifying the methods for disclosure, how rewards are calculated, and other modalities of execution. What is interesting of this policy is that there is a section in Privacy Vulnerabilities concerning the Xiaomi mobile applications pre-installed on Xiaomi phones. In the thesis, we analyze some of the applications present on the Xiaomi package list. In the scopes section, which is the final section of the policy, a list of apks that have a critical level of vulnerability is shown, and in particular the Mi Browser apk is present with a high level vulnerability. According to the policy, the apk presents a vulnerability that is either undisclosed or new depending on new technologies, and

it will have a great impact on the Xiaomi business.

3.3.3 Xiaomi Privacy Policy

In the Xiaomi Privacy Policy [7], in particular Section 3, is written how the company shares, transfers, and publicly discloses personal information. This section states for example that the company may share personal data with Xiaomi affiliates, a group of independent companies forming the Mi Ecosystem, third party service providers and business partners, advertisers all to provide the user with all the functions of their products or services. In the first two sections of this policy is also stated what information the company collects (including personal data and log features present, for example, in applications) and the cookies and other technologies they use, for example, log files and mobile analytics. This policy is interesting relevant to this research because it is stating beneath the lines that the company is free to do anything they want with the user's personal information.

3.3.4 Logging Contact-Tracing Data

In this article [8] the author explains how a Google-implemented service, the Google-Apple Exposure Notification, used to do digitally assisted contact tracing, can log directly to the system log crucial information that can be read by third-party applications and used for privacy attacks. The article then exposes how these logs can not be seen by third-party applications downloaded from the Play Store, but they can potentially be seen by pre-installed apps. Google allows phone hardware manufacturers to ship their phones with these pre-installed apps that have access to privileged permissions, for example READ_LOGS, a permission that lets the application read system logs. The author presents an example of a Xiaomi Redmi Note 9 that has 77 pre-installed apps, 54 of which have the READ_LOGS permission, furthermore the collection of log data is explicitly mentioned in the privacy policy.

4
Dataset Collection

One of the most fundamental processes in this research is data collection. In this section, we will explain how we collected the data and the reason we proceeded in the following way. In detail: section 3.1 exposes the reason why xiaomi phones are the subject of our research; section 3.2 explains how the applications to analyze were chosen and lists which are those applications; section 3.3 presents the problems encountered during the process.

4.1 Xiaomi Phones

As explained in previous sections, the objective of this research project is to check if there are differences between system apps of phones from different vendor and the same app found on the Google Play Store. Every vendor has their system apps, we thought that Huawei and Xiaomi were the more accessible vendors, and eventually we chose Xiaomi because we already had the phones to execute our research. From there on, we activated the USB debugging and started to gather application using the Android Debug Bridge (ADB) pull command.

4.2 Applications Collection

After choosing the vendor for our research, we started to find Xiaomi system apps. We used a personal Xiaomi Redmi Note 7 and tried different approaches to find system apps:

- Firstly, we tried to see which applications could be uninstalled from the phone by the uninstall feature. Since the phone wasn't rooted system apps couldn't be uninstalled by a normal user;
- Next, from the phone's settings, in particular the App section, we could set System App Settings and from there we could see all the apps the phone consider as system;
- Finally from online sources we discovered that system apps could be recognized because their certificate is signed using the rom image key. Of course, this method was not practical because we were lacking the Xiaomi rom key.

There was also another method to recognize system applications, and that was to search these applications under the /system/app partitions. To use this method we needed to root a phone and since the Redmi Note 7 was personal, we used a new phone, the Xiaomi 9A. After rooting the phone, we pulled the applications from the system folder.
Afterwards, we researched those applications in the Google Play store and tried to download them. The third party applications were gathered from external sites like ApkMirror [9], ApkPure [10], but we could not confirm if these applications were really taken from the Play Store. To solve this problem we chose to download the applications from the store using another vendor's phone, a Huawei 7 and then pull the applications using adb. Of course, not all the system applications were available on the play store, but in the end we found the following applications ready for a confrontation: Mi Browser, Weather, Music, Mi Video, Share Mi, Mi Remote, Calendar, Xiaomi Community, Xiaomi Store, Files.

4.3 Related Obstacles

In this section, we will expose the problems encountered during data collection and we will explain the solution we took along with the reason for it. During the first collection of third party applications, we decided to use the android emulator and download directly the applications in x86 format. The problem was that those applications do not exist in x86 format, not from the play store or other online sources. So we could not use any emulator for our analysis, and in the end we decided to use real devices for the download of third party applications. The next major problem we encountered was with the versions of the system apps and the third party apps. Some of the app's versions were not always the same, so the confrontation between the two would not have been accurate or precise. To solve this problem we tried to find the correct version of the third party app through the sites mentioned before but for some applications these versions weren't available. In the end we decided to gather third-party applications

with the closest versions to the system apps, if the same wasn't available. The next difficulty we encountered was during the identification of the system application. Using the method of analysis of applications under /system/app, we noted that depending on the Android version, some system applications had a path that was pointing to /data/app. For example, for Android 9 all the system applications were in the /system/app partition but for Android 10 and 11 some of them had the apk under /data/app. This, of course, created confusion in the identification of system apps. Furthermore, we noted that the system apps had two versions of themselves, one under /system/app that was very old, and the other version under /data/app which was recent and updated. What we concluded from online sources and the AOSP documentation was that the system applications under /data/app were the official ones and that the counterpart under /system/app was a backup version used to restore the application in case it was uninstalled as root.

5
Tools employed

This chapter presents the tools used during the analysis. In detail, we will describe what their outputs are and the reason behind their usage. In section 5.1 we will present Androwarn, following with section 5.2 Maldrolyzer, section 5.3 RiskinDroid, section 5.4 SUPER, 5.5 StaCoAn, 5.6 Quark Engine and section 5.7 ApkLeaks. The tools mentioned above are mainly focused on static analysis; we decided to use many static analysis tools in hopes of finding vulnerabilities in applications through the different output formats and analysis methods of the tools. In the final section 5.8 we instead present Frida, the tool used for dynamic instrumentation.

5.1 Androwarn

Androwarn [11] is a static code analyzer for malicious android applications. The tool analyzes the Dalvik bytecode of the applications through the androguard library. The user of the tool can choose the technical level of the report generated by the analysis. Androwarn can detect malicious behavior of an application and categorize it in the following classes:

- Telephony identifier exfiltration
- Device settings exfiltration
- Geolocation information leakage

- Connection interfaces information exfiltration
- Telephony services abuse
- Audio/video flow interception
- Remote connection establishment
- PIM data leakage
- External memory operations
- PIM data modification
- Arbitrary code execution
- Denial of Service

This tool was chosen because it was relatively easy to use and has an output that spans multiple app behaviors that can be easily compared.

5.2 Maldrolyzer

Maldrolyzer [12] is a tool that is able to extract sensitive data that is considered actionable from an android malware. These data can be a simple command and control server where the malware sends or receives information, or it can even be a phone number, IMEI and more. This tool was chosen because of its simplicity, we wanted to detect if the system application or the third-party applications by xiaomi could be considered malware or if they had a common C&C server to which they send personal information.

5.3 RiskInDroid

RiskInDroid [13] is a tool written in both Java and Python, it analyzes the permissions of the applications and based on them attributes a risk from 0 to 100 to the app. The Java language is used to look at the app's permission while the Python language is used for the classification techniques to calculate the risk factor, in fact python has a lbrary called scikit-learn specialized in machine learning. Moreover, the tool does not only look through the application manifest, but through reverse engenering and static analysis it can infer which permission are really used and which not by dividing them in for sets of permission cathegories:

- Declared permissions, are permissions extracted from the manifest
- Exploited permissions, permissions declared in the manifest and used in the bytecode
- Ghost Permissions, are permissions used in the bytecode but not declared on the manifest
- Useless permissions, are permissions declared in the manifest but never used in the bytecode

From the built permissions set and from the official android permission list, the application creates an input for the classifier that will output the risk factor. The tool was chosen for this analysis because it focuses on the permission analysis, which was crucial in our process, and since the tool can create a list of permission actually used in the bytecode, we could easily focus our research in finding the API call in the source code.

5.4 SUPER

SUPER [14] is a Rust-developed CLI tool that analyzes apks to search for vulnerabilities. What differentiates this tool by the other stat of the art static analysis tools is that it is rules-based and extensible. The user of this tool can easily create a personal rule to search for specific vulnerabilities. The downside of this tool is that its output is in HTML format and is very verbose, so in order to find differences between the system app output and the third party output we had to rely on the git diff command. Using Git diff, it was still difficult to spot the difference since the command gives a difference even if a line of code is inverted. Another difficulty of this tool is that of the rule format; it is not easy to understand how to modify the rules.json file correctly. Despite the difficulties, we insisted on using this tool in order to find varying categories of vulnerabilities in the applications.

5.5 StaCoAn

StaCoAn [15] is a cross-platform static analysis tool for mobile applications. The tool focuses on the code of the application looking for lines that can contain hardcoded credentials, API keys, URL's of API's, decryption keys, and major coding mistakes. It was built to aid the user through the tool's usability and graphical interface, in fact it is very easy to use, the user can just start the tool drag and drop the apk in the interface and obtain the generated report. This is

a static analysis tool that we chose because of its different output compared to the other tools mentioned before. The only downside of StaCoAn is that it does not work well with obfuscated code.

5.6 Quark Engine

Quark engine [16] is a malware reverse engineering tool. The tool offers a static analysis feature as well as a dynamic analysis one, and we overlooked the dynamic analysis feature because we needed a tool for dynamic instrumentation. It is another rule-based tool, but what is different from SUPER is that its rule set gets continuously updated, and it has a rule generator that lets the user create a personalized rule in the correct format. Furthermore, the authors developed a personal theory of Android malware, creating a five-stage process to recognize if the application is engaging in malicious behavior. To recognize malware behavior, first the tool searches for the requested permissions, then it looks through native API calls, analyzes certain combinations of native API, looks at their calling sequence, and finally looks for API that handles the same register. The tool defines weight and thresholds through the stages of the process, in order to calculate the threat level of the malware, and it neglects some cases of code obfuscation. Quark engine was chosen because it is a well-documented project, it is still rule based and easy to personalize, and it is still maintained with respect to other tools mentioned before. The report it generates is available in different formats and detailed.

5.7 Apkleaks

Apkleaks [17] is yet another static analysis tool, but it focuses on scanning an apk for URIs, endpoints, and secrets. We chose this tool to give us another overview on the code of analyzed apps, every static analysis tool can scan for URIs and endpoints (we mentioned StaCoAn), we wanted to have different static analysis tools to have a complete scan of an application through different points of view. So in the end we could have a better comparison between system apps and third party apps.

5.8 Frida

Frida [18] is a dynamic code instrumentation toolkit, it can inject javascript code into the chosen application. This tool was chosen because it was easy to integrate with our rooted phones,

since we rooted using the Magisk application. The tool requires a server to be installed in the phone in order to run the custom javascript, after a search on the web we found out that magisk offers a plug-in that can install and start the frida server automatically on the phone. Through ADB we can connect the phone to a computer and discover which process runs on the device, and after choosing the target process, we can inject code while the process is running. This way we have a running instance of the application targeted that we can explore, for example, we can log information on the computer's terminal or call APIs directly from the application.

6 Design

In this section, we will present our method of research and comparison and our reasons behind this method. In detail, section 4.1 will introduce the preliminary analysis performed in order to identify which features of the application we should have considered for the comparison. Then in section 4.2 we will describe our analysis on privileged permissions and why we considered them the core of our comparison, we will also describe our process in order to trigger these permissions and why we chose some specific tools.

6.1 Preliminary analysis

Since this research focuses on the differences between the same app obtained from different sources, we decided to execute an analysis on both code and behavior, which is equivalent to static and dynamic analysis. The idea was to start with the static analysis of the application's code and manifest, check for meaningful differences, and then, through dynamic instrumentation, observe in which ways the applications behave differently. After having discovered the vulnerable or dangerous behaviors, we could create a threat model and eventually find out an attack that can be attributed to the vulnerability. We decided to follow this approach because the analysis can be performed through a reverse engineering process because we had all the necessary resources. Both Applications APK, Manifest, Decompiled Source Code, and phones to test the applications.

6.1.1 Static Analysis

After gathering all the applications, we proceeded with static analysis. Through a list of Android static analysis tools we gathered on the internet, we begin working on the analysis of these applications. However, some of these tools were outdated, discontinued, or deprecated; here are the functioning tools used in the process: Androwarn, Maldrolyzer, RiskInDroid, SUPER, StaCoAn, Quark-Engine, ApkLeaks, ApkAnalyzer, Jadx. As mentioned beforehand some applications were not in the same version, but we decided to proceed anyway with the analysis in those cases because there are possible scenarios that go along with the purpose of our research. For example if the application found on the Play Store is a downgraded version with respect to the system app and in this version there is a security issue which is not present in the recent version of the system app, this scenario represents a vulnerability that can be distributed between devices that download this application.

Manifest and Source Code

During the static analysis we thought that in android applications the Manifest and the Source Code would be the most crucial part where we could find results. In the source code we hoped to find more suspicious code in the system app than third party apps; of course this difference is more clear if both the system app and the third party app are on the same version, otherwise the result may be unreliable. On the third party app there should not be secret vulnerable code since the application is distributed through the Google Play store and is supervised.
What we searched for in the manifest was the over- or under-permissioning of one application over the other. We also hoped to find some dangerous or privileged-level permissions on the system application because of the same reason explained for the source code.

Permission APIs

Directly connected to the permission are the APIs, every permission has specific APIs that can be called in the source code. When we analyzed the source code, we also searched for difference in API calls between apps. For example, the system app can execute some privileged API calls which cannot be executed by the third party application. This led to the discovery of privileged permissions [19].

6.2 Privileged Permissions Discovery

From the first results obtained by the static analysis (which will be discussed in detail in Section 5) we did not notice significant differences between system apps and third-party applications. This type of result led us to believe that maybe the system applications run on a different status in contrast with the third-party applications downloadable from the play store. Since system apps are pre-installed, which means they are signed with the same key used to sign the vendor's ROM, they can use a set of permissions specific to system applications that make them execute on a privileged status. With this idea in mind, we researched the Android documentation and found out all about privileged permissions and privileged applications.
Privileged applications are apks located under the /priv-app folder in one of the system partitions of the image. Up to Android 8.1 this system partition was called /system, from Android 9 onward there exist three system partitions called /system /product/ /vendor. In these partitions we can find a folder called /etc/permissions/priv-app, and in this last folder we can find files that specify which signature|privileged permission to grant to a privileged application.

Figure 6.1: List of the partitions found on the phone, in particular product, system and vendor

```
product
product_services -> /system/product_services
sbin
sdcard -> /storage/self/primary
storage
sys
system
ueventd.rc
vendor
verity_key
```

These xml files, in the case of Xiaomi phones, have a list of applications and in every application the signature|privileged permission granted is specified. Moreover, these XMl files can grant or deny the permissions for applications who are on the same partition. For example, if we find a file named privapp-permissions.xml under the /vendor partition, the application under the same partition can request these privileged permissions and the request will be resolved based on privapp-permissions.xml under the /vendor partition.

6.2.1 Static Analysis

With this new information in mind we decided to chek for the request and consequent usage of these privileged permissions. From the xml files found in /system/etc/permissions/priv-app we obtained a list of all the privileged permission that can be granted to system app, and decided to repeat a Static Analysis checking traces of the usage of such permissions.

Manifest

Since these are permissions, the first place we decided to check was the manifest of the applications, and surprisingly we noticed the presence of privileged applications according to the XML file. Not all the permissions specified by the file were present in the applications, but there were also applications without this privileged permissions; this allowed us to restrict our research and focus on a smaller set of applications, namely: Music, Weather, Mi Browser, Share Me, and Calendar. Eventually from the Play store on the Samsung phone we only found Calendar, Share Me and Mi Browser.

Figure 6.2: Shows how privileged permissions get declared on the XML file

```
<privapp-permissions package="com.xiaomi.midrop">
    <permission name="android.permission.INTERACT_ACROSS_USERS"/>
    <permission name="android.permission.LOCATION_HARDWARE"/>
    <permission name="android.permission.MANAGE_USERS"/>
    <permission name="android.permission.MODIFY_PHONE_STATE"/>
    <permission name="android.permission.MOUNT_UNMOUNT_FILESYSTEMS"/>
    <permission name="android.permission.OVERRIDE_WIFI_CONFIG"/>
    <permission name="android.permission.STATUS_BAR"/>
    <permission name="android.permission.WRITE_APN_SETTINGS"/>
    <permission name="android.permission.WRITE_MEDIA_STORAGE"/>
    <permission name="android.permission.WRITE_SECURE_SETTINGS"/>
    <permission name="android.permission.LOCAL_MAC_ADDRESS"/>
</privapp-permissions>
```

Permission APIs

Afterwards we tried to connect the list of privileged permissions to their associated API through the Android documentation and tools such as PScout. Then we repeated the static analysis on the source code of the application, in order to find usage of these APIs which would have justified the presence of privileged permissions in the manifest file.

Permission per applications	Presence in Xiaomi	Presence in Samsung
Xiaomi Calendr	12.5.6 Xiaomi	12.5.6 Samsung
INTERACT_ACROSS_USERS	NO	NO
MOUNT_UNMOUNT_FILESYSTEMS	NO	NO
READ_PRIVILEGED_PHONE_STATE	NO	NO
STATUS_BAR	NO	NO
WRITE_SECURE_SETTINGS	YES	YES
GET_ACCOUNTS_PRIVILEGED	NO	NO
Xiaomi Mi Browser	13.10.0-gn Xiaomi	13.10.0-gn Samsung
GET_ACCOUNTS_PRIVILEGED	YES	YES
INTERACT_ACROSS_USERS	YES	YES
READ_PRIVILEGED_PHONE_STATE	YES	YES
STOP_APP_SWITCHES	YES	YES
WRITE_SECURE_SETTINGS	YES	YES
Xiaomi Share Me	3.22.06 Xiaomi	3.22.06 Samsung
INTERACT_ACROSS_USERS	YES	YES
LOCATION_HARDWARE	YES	YES
MANAGE_USERS	YES	YES
MODIFY_PHONE_STATE	NO	NO
MOUNT_UNMOUNT_FILESYSTEMS	NO	NO
OVERRIDE_WIFI_CONFIG	YES	YES
STATUS_BAR	YES	YES
WRITE_APN_SETTINGS	NO	NO
WRITE_MEDIA_STORAGE	YES	YES
WRITE_SECURE_SETTINGS	YES	YES
LOCAL_MAC_ADDRESS	YES	YES

Table 6.1: Privileged permissions found on the applications' manifest

6.2.2 Dynamic Analysis

The next step in the analysis was to test if the applications present different behavior during execution. However, not having the third party applications for x86 system implied that we couldn't use the android studio emulator for our tests. Our solution then was to root two phones, one Xiaomi phone so that we could test the execution of Xiaomi system applications, and one phone from another vendor so that we could freely download the third party applications from the Play store. For the phone from a vendor different from Xiaomi we decided to go with a Samsung phone, since it was already in our hands and had been previously used in another research project.

Dynamic instrumentation through Frida

After having rooted the two phones, we needed to choose a framework for dynamic instrumentation, we decided to go with Frida since it was one of the most popular on the Web and it was relatively easy to integrate in the two rooted phones. In detail, since we rooted the phones through Magisk, we could install the Frida server on them using a Magisk plugin. After this set up, we could test if the API of the permission were triggered through a Javascript script that can be executed during the runtime of the application. This process seemed reasonable since this way we could see if third party applications could trigger privileged API.

7
Implementation

In this section, we will present the actual implementation of the research. The process is the same as the one presented in the ”Design” chapter. In section 4.1 we will present our preliminary analysis, what are the applications gathered and the tools used for the comparison. Then in section 4.2 we will present our study on privileged permission, a new static analysis considering those permissions, and to conclude, our dynamic instrumentation of applications.

7.1 Preliminary analysis

As mentioned in the previous chapter, in our preliminary analysis we started gathering the applications to analyze. First, we started collecting the system applications from a personal phone, a Xiaomi Redmi Note 7 with Android 10, and then we collected the third party applications from online sources as ApkMirror, ApkCombo, ApkPure. In the end, our data set consisted of 10 applications: Music, Weather, Mi Browser, Share Me and Calendar, Mi Video, Mi Store, Mi Community, Mi Remote, and Files. On these applications, we started to execute our static analysis.

7.1.1 Static Analysis

For every application we analyzed the third party app and the correspondent system app through a set of open source tools specific for static analysis of android applications. The set of tools

consisted of: Androwarn, Maldrolyzer, RiskInDroid, SUPER, StaCoAn, Quark-Engine, ApkLeaks, ApkAnalyzer, Jadx.

Manifest and Source Code

Androwarn, SUPER, StaCoAn, Quark Engine and ApkLeaks are tools that focus on the analysis of the source code of the applications while ApkAnalyzer and RiskInDroid are tools that analyze the manifest of an android application. In particular, RiskInDroid attributes a risk factor to the application.
With applications that have matching version we haven't found differences, if not on the signature of the apk. Instead, with applications that have a different version, the tools found discrepancies in both manifest and source code.

7.2 Privileged permissions Analysis

After this first preliminary analysis with no results, we moved on to the analysis of privileged permissions. The process for the discovery of these permissions has been explained in Chapter 4. For the privileged permission our dataset shortens, because on the file xml we found only five applications out of the 10 in our initial dataset, the apps are: Music, Weather, Mi Browser, Share Me, and Calendar. The system apps were taken from a rooted Xiaomi 9A while the third party applications were downloaded from a Samsung phone, not all the applications where available only ShareMe, MiBrowser and Calendar. We proceeded with a full analysis on these applications.

7.2.1 Static Analysis

For the static analysis we first decided to focus on the manifest analysis of the apps, checking if the privileged permissions were present in both the system app and the third-party app.

Manifest

For the manifest analysis, we used Apkanalyzer and RiskInDroid. The three apps that we analyzed had all the correspondent versions, so the results could be trusted.

Permission APIs

After having found the privileged permissions in both the third-party app and the system app, we needed to connect these permissions to their APIs. To this end we consulted the AOSP and third-party tools like PScout, eventually as a cross-check for the correctness of the mapping we used a temporary application programmed in Android Studio. In the end, we used Jadx to check for calls of these Apis in the applications.

7.2.2 Dynamic Analysis

After the static analysis, we proceeded with the dynamic analysis of the applications. We manually tested the applications on the rooted phones for differences in behavior. The test consisted of trying the system applications in all of their features and checking for the same behavior in the third-party app.

Dynamic instrumentation through Frida

Using Frida, we created a script that can test whether the application having privileged permission can actually call an API during its execution. If the application does not have the right privileges, meaning the XML file and the install in the correct partition, it will crash. The scripts functions in the following way, it first hooks the android log class and the interested class that has the API call we want to execute. In our case we need to generate a new TelephonyManager object; since it is not used in the application, we cannot hook it. The permission READ_PRIVILEGED_PHONE_STATE allows for the call to TelephonyManager.getImei(), so we're rewriting the code forcing the system app to call that method. If the application has the right privileges and permission, the phone Imei gets logged on the computer terminal, otherwise the application running crashes.

Listing 7.1: JavaScript code injected into the applications

```
Java.perform(function () {
   var Log = Java.use("android.util.Log");
   var TelephonyManager = Java.use("android.telephony.TelephonyManager");

   Log.w.overload('java.lang.String', 'java.lang.String').implementation =
   function(tag, msg) {
     console.warn("\nInside Log.w");
     console.log("Log.w tag: " + tag);
     console.log("Log.w msg: " + msg);
     var TelephonyManagerObject = TelephonyManager.$new();
     var Imei = TelephonyManagerObject.getImei();

     console.log("TelephonyManagerObject.getImei retval " + Imei);
     var retval = this.w(tag, msg);
     console.log("Log.w retval: " + retval);
     return retval;
   }

 });
```

8

Results

As mentioned in the Desing and Implementation chapter, the results found in this analysis were not very significative. In the first preliminary analysis the applications that were on the same version had no difference in the code. The applications with different versions had little differences but it is impossible to establish if this is due to the provenience of the application or the application's version. In the second analysis where we restricted the dataset but

Table 8.1: The table shows differences found in the Weather application, the versions do not match

Tool		System	Play Store
		G-12.3.6.8 Xiaomi Version	G.12.3.6.3 Samsung Version
Androwarn	Telephony Identifiers leakage	More	Less
	Device Settings Harvesting	Various differences	Various differences
	Fingerprint	Various differences	Various differences
	Receivers	Rest is equal	com.google.firebase.iid.FirebaseInstanceIdReceiver
	Provider	com.miui.bugreport.logprovider.DumpLogProvider	Rest is equal
	Permissions Asked	android.permission.ACCESS_WIFI_STATE	Not asked
		Not asked	com.google.android.c2dm.permission.RECEIVE
		com.miui.bugreport.permission.DUMP_CACHED_LOG	Not asked
	Permission Implied	android.permission.READ_EXTERNAL_STORAGE	android.permission.READ_EXTERNAL_STORAGE
	Libraries	com.miui.system	Rest is equal
	Intents Sent	Rest is equal	Ljava/lang/String
		Rest is equal	Ljava/lang/StringBuilder;->toString()Ljava/lang/String
Maldrolyzer		No malaware	No malaware
Riskindroid		Higher risk	Lower risk (Difference is only of 1 unit)
SUPER		2x Rooted device detection CommonUtils.java (613, 616)	2x Rooted device detection CommonUtils.java (877, 880)
		3x Weak Algorithm Util.java (272, 289, 302)	1x Weak Algorithm zzt.java (70)
		Overall more low criticality vulnerabulities and warnings	
Stacoan		525 instances with keys written in plain sight	525 instances with keys written in plain sight
Quark-Engine		Various differences	Various differences
Apkleaks		Similar	Similar

added the dynamic instrumentation, we have not found differences in the code or manifest. Both the system apps and the apps obtained from Google Play had matching versions and

both had the same manifest with the privileged permissions and no difference in the source code. What we found interesting was that even if the application is downloaded from the Play store it had privileged permissions. To check if the privileged permission can be effectively used by a third-party app, we proceeded with the dynamic instrumentation. Through Frida, we tested if the applications, both system and third party, can invoke an API connected to a privileged permission that has been found on the manifest. The permission in question is READ_PRIVILEGED_PHONE_STATE and the API associated with it is getImei() from the TelephonyManager class. When the script is executed through Frida in the system application, nothing happens, which means that the application has the privileges to execute this API. When instead the script is executed on a third party app, the applications crashes and an error is thrown, meaning the app does not have the privileges to execute the API call.
What we presume is that since the system app is installed in the correct partition and has an xml file with the specified privileged permission, it is able to execute the interested APIs. On the other hand, the third party application is installed under the data partition and has no xml file with permission specified, so it is not able to execute the privileged APIs. We can deduce that the third-party applications are basically copies of the system applications and they also have permissions on the manifest that are not used.

9 Conclusion

In conclusion, what we can say about the analysis is that the third-party applications are not different from the system applications. They have the same privileged permissions and do not use privileged APIs to execute dangerous actions that can harm the user. Moreover, privilege escalation attacks or confused deputy attacks should not be possible because the third-party application gets installed under the /data partition. If the third party application gets installed under the /system/priv-app partition and it has a set of permissions granted by the xml file under /system/etc/permissions it could be possible for the application to behave like a system application.

References

[1] "Android open source project." [Online]. Available: https://source.android.com/

[2] J. Gamba, M. Rashed, A. Razaghpanah, J. Tapiador, and N. Vallina-Rodriguez, "An analysis of pre-installed android software."

[3] M. Elsabagh, R. Johnson, A. Stavrou, C. Zuo, Q. Zhao, and Z. Lin, "Firmscope: Automatic uncovering of privilege-escalation vulnerabilities in pre-installed apps in android firmware."

[4] "Xiaomi devices found tracking and recording browsing data of millions." [Online]. Available: https://malwaretips.com/threads/xiaomi-devices-found-tracking-and-recording-browsing-data-of-millions.100533/

[5] "[update: Toggle to opt-out] xiaomi devices found tracking and recording browsing data of millions." [Online]. Available: https://fossbytes.com/xiaomi-devices-found-tracking-and-recording-browsing-data-of-millions/

[6] "Xiaomi bug bounty policy." [Online]. Available: https://hackerone.com/xiaomi?type=team

[7] "Xiaomi privacy policy." [Online]. Available: https://privacy.mi.com/all/en_IN/

[8] "Why google should stop logging contact-tracing data." [Online]. Available: https://blog.appcensus.io/2021/04/27/why-google-should-stop-logging-contact-tracing-data/

[9] "Apkmirror, free and safe android apk downloads." [Online]. Available: https://www.apkmirror.com/

[10] "Apkpure." [Online]. Available: https://m.apkpure.com/it/

[11] "Androwarn, yet another static code analyzer for malicious android applications." [Online]. Available: https://github.com/maaaaz/androwarn

[12] "Maldrolyzer, simple framework to extract "actionable" data from android malware (c&cs, phone numbers etc.)." [Online]. Available: https://github.com/maldroid/maldrolyzer

[13] "Riskindroid, a tool for quantitative risk analysis of android applications based on machine learning techniques." [Online]. Available: https://github.com/ClaudiuGeorgiu/RiskInDroid

[14] "Secure, unified, powerful and extensible rust android analyzer." [Online]. Available: https://github.com/SUPERAndroidAnalyzer/super

[15] "Stacoan, a crossplatform tool which aids developers, bugbounty hunters and ethical hackers performing static code analysis on mobile applications." [Online]. Available: https://github.com/SUPERAndroidAnalyzer/super

[16] "Quark-engine, an obfuscation-neglect android malware scoring system." [Online]. Available: https://github.com/quark-engine/quark-engine

[17] "Apkleaks, scanning apk file for uris, endpoints & secrets." [Online]. Available: https://github.com/dwisiswant0/apkleaks

[18] "Frida, a world-class dynamic instrumentation framework." [Online]. Available: https://frida.re/docs/android/

[19] "Privileged permission allowlisting." [Online]. Available: https://source.android.com/docs/core/config/perms-allowlist?hl=en

Acknowledgments

I would like to thank my family and friends who supported me throughout my life.