Department of Media and Information

Hochschule Offenburg

Badstraße 24

D–77652 Offenburg

Masterarbeit

Give Me Everything: Analyzing leakage of information in the Android platform with respect to wireless communications and personally identifiable information

Çağrı Erdem

Course of Study: Enterprise and IT Security

Supervisors: Prof. Dr. Dirk Westhoff

Prof. Dr. Daniel Hammer

Commenced: April, 2021

Completed: August, 2021

Abstract

Among the billions of smartphone users in the world, Android still holds more than 80% of the market share. The applications which the users install have specific set of features that need access to some device functionalities and sensors that may hold sensitive information about the user. Therefore, Android releases have set permission standards to let the user know what information is being disclosed to the application. Along with other security and privacy improvements, significant changes to the permission scheme are introduced with the Android 6.0 version (API level 23). In this master thesis, the Android permission scheme is tested on two devices from different eras. The evolution of Android over the years is examined in terms of confidentiality. For each device, two applications are built; one focused on extracting every piece of information within the confidentiality scope with every permission declared and/or requested, and the other app focused on getting this type of information without user notification. The resulting analysis illustrates whether how and in what way the Android permission scheme declined or improved over time.

Declaration

I hereby declare that the work presented in this thesis is entirely my own and that I did not use any other sources and references than the listed ones. I have marked all direct or indirect statements from other sources contained therein as quotations. Neither this work nor significant parts of it were part of another examination procedure. I have not published this work in whole or in part before. The electronic copy is consistent with all submitted copies.

Erklärung

Ich versichere, diese Arbeit selbstständig verfasst zu haben. Ich habe keine anderen als die angegebenen Quellen benutzt und alle wörtlich oder sinngemäß aus anderen Werken übernommene Aussagen als solche gekennzeichnet. Weder diese Arbeit noch wesentliche Teile daraus waren bisher Gegenstand eines anderen Prüfungsverfahrens. Ich habe diese Arbeit bisher weder teilweise noch vollständig veröffentlicht. Das elektronische Exemplar stimmt mit allen eingereichten Exemplaren überein.

Ort, Datum, Unterschrift

Contents

List of Figures

List of Tables

1 Introduction

The last decade has witnessed the adoption of smartphones reaching total ubiquity all over the world, in addition to a critical increase in the number of applications a smartphone user has installed. Accordingly, the amount of vulnerabilities that come with these apps have seen a considerable growth [12]. When an application is downloaded on the Android platform, the user is prompted with an alert box that asks for specific permissions that the app will use to perform its functions. These permissions may include the user's physical activities (e.g., sleeping, commuting, exercising), location history, contacts, SMSs, device specifications, camera access, internet connection, etc.

In the pre-2015 versions of Android, if the user were to deny these permission requests, the app would either refuse to work, or be removed from the smartphone entirely [18]. Google introduced dynamic permissions after the Android Marshmallow (6.0) update in 2015, allowing users to grant or revoke permissions post-installation.

The permission scheme of Android platform provides interesting avenues for research. It is known that overwhelming majority of users usually ignore the contents of end-user license agreements (EULAs) and permission prompts, or fail to comprehend them entirely [5]. Android apps are prone to being over-privileged, as in they often ask for permissions beyond their functions require [22]. Moreover, as Taylor et al. also discuss, there exists no way to know if the application merely uses this sensitive information for purposes limited to app parameters, or malicious intents that extend beyond the EULA. An exploratory study points out that even with post-Marshmallow devices, circumventing select permissions are possible, and some dangerous permissions are still

automatically granted [2]. Kywe et al. found that by exploiting some unprotected APIs, they could access to sensitive information such as device ID, Wi-Fi and network details, and user settings information without asking any permissions from the user [13].

As a result, this master thesis aims to answer following questions: Analyzing comparatively between the *all permissions granted* and *no permissions granted* states of an application that will be built;

1. What kind of information can be gathered in terms of

    a) personally identifiable information (PII) (e.g., contacts, names, SMSes, device specific information, MAC addresses),

    b) and wireless devices that the user smartphone has been in contact with (e.g., other smartphones, Bluetooth devices, APs)?

2. If one were to install the same application and conducted the exact same comparative analysis on the same Android smartphone with elevated root privileges, how would the obtainable information differ from the unmodified device?

Following this introduction, Chapter 2 delves into the basics of the Android permission scheme. Two points in time are taken as reference to examine the permission scheme. In addition, the Android boot sequence is described to deliver a wider comprehension of the application sandboxing embedded into the OS.

Chapter 3 deals with the existing research. It primarily focuses on the issues affiliated with the permission scheme. Scholars emphasize on matters such as application overprivilege, network usage of the apps, vulnerability analysis, and changes brought by the Android 6.0.

Subsequently, Chapter 4 describes research methods employed in this thesis. Moreover, it demonstrates the device specification and application details for the investigation. Next, Chapter 5 provides in-depth analysis of the permission scheme. In Chapter 6, a confidentiality conscious examination of the investigation findings is conducted. Consequently, Chapter 7 gives the final verdict based on this review.

2 An Overview of the Android Permission Scheme

In this section, the Android permission scheme will be examined from the perspective of pre-2015 and post-2015 era. The reason for this distinction comes down to Google's complete overhaul in the handling of runtime permissions after Android 6.0 update.

2.1 Pre-2015 Era

The application security of Android is founded on isolating the applications from each other, and allowing specific set of permissions the app requires [14]. Only some system processes run with root privileges.

After the Android Bootloader loads kernel to the memory, the root process `init.rc` mounts system directories, and starts native daemons such as Service Manager and Media Server. The init scripts calls `AndroidRuntime.start()`, where Dalvik VM is activated. Subsequently, `ZygoteInit.main()` is called. It first starts a special process for the system server through `startSystemServer()`, then it preloads classes and listens for incoming commands. Zygote is an OS process that enables apps to be isolated in a sandbox. Every time a Java application is launched, `ZygoteConnection.runOnce()` is called, and Zygote gets forked into a new process through `Zygote.forkAndSpecialize()`. The Linux kernel's implementation of the copy-on-write resource management approach is responsible for this forking functionality. Forking involves establishing a new process that is a mirror of the original.

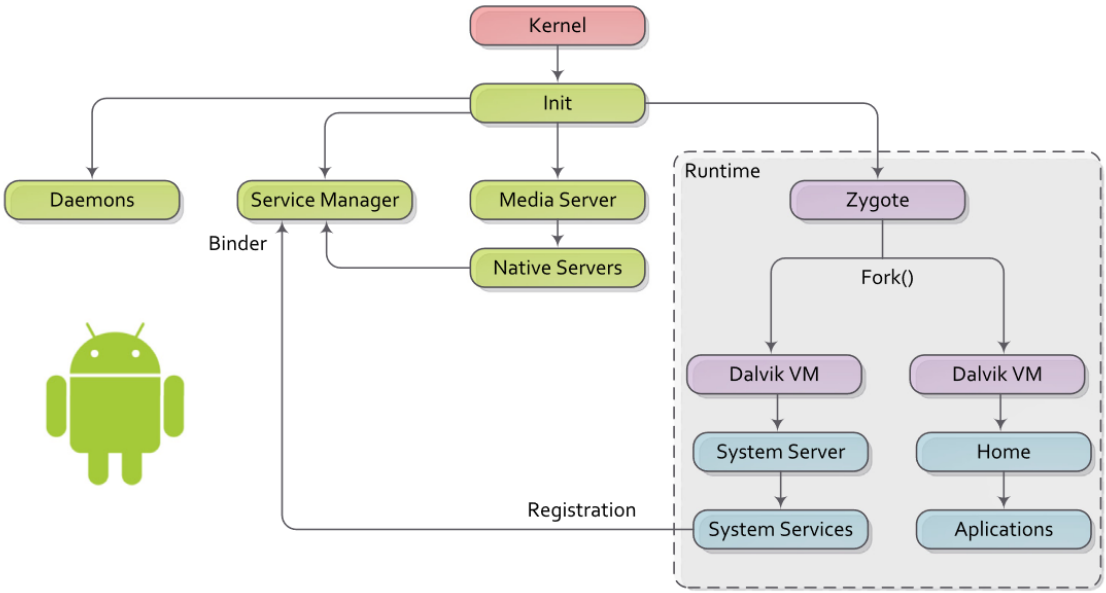

Figure 2.1: Android boot sequence [20].

Rather than actually copying anything, it maps the new process pages to the parent process, and only makes copies when the new process writes to a page. Each application runs in its own process inside a Dalvik VM instance with its own user and group ID, and with no means to communicate with other apps.

This sandbox application design inherently restricts potentially dangerous functionality if not declared by the app [3]. The required permissions of an application is declared in the AndroidManifest.xml file, such as to get location services, access contacts, or images. At this point in time, Android had over 110 permissions. The permissions listed in the file are presented at the install-time to the user, who can either accept and continue the installation, or deny, which halts the installation. Permissions are divided into normal and dangerous, where normal permissions are hidden from the user, until they wish to expand the total list of permissions the app require.

2.2 Post-2015 Era

As of Android 6.0 (API level 23), Google introduced runtime permissions and special permissions in addition to the install-time permissions [11].

2.2.1 Install-time permissions

Install-time permissions enable applications to access to restricted data, and allow it to perform restricted actions that may have marginal effect on other system or third party apps. When the developer declares install-time permissions, they are displayed on the app store, and automatically granted when the user installs the app. Sub-types of install-time permissions include normal permissions and signature permissions.

Normal permissions

These include data and actions that extend beyond app's sandbox. They are classified as normal protection level because they present little to no risk to user's privacy and the operation of other apps.

Signature permissions

A permission that the system grants only if the requesting application is signed with the same certificate as the application that declared the permission. Therefore, if an app declares a signature permission that another app has defined, and both applications are signed by the same certificate, the system grants the permission to the first app after installation without the users explicit approval. Otherwise, the permission will not be provided to the first app. While the definition of install-time permissions state that the overall impact is marginal to the system and the other apps, signature permissions somewhat defy this interpretation. However, since the access given to the app by this type of permission is bound on a certificate, a developer can only give access to a secondary application that is built by them. The meaning is that, the type of access enabled is not marginal, however the scope is reduced so much that the effect on the user is naught. An app with signature permissions has only so much access that the user already consented to give to the developer. Therefore, signature permissions enable the developer to provide a convenience for the user at no cost to their privacy.

2.2.2 Runtime permissions

Known as the dangerous permissions before Android 6, runtime permissions give applications additional access to restricted data that can have a significant effect on system and other third party apps. As opposed to install-time permissions, declaring these permissions result in a permission prompt during runtime. They can access personally identifiable information of the user. Hence, these permissions are classified as dangerous protection level.

2.2.3 Special permissions

These permissions are defined by the platform and OEMs to carry out specific app functionalities. In addition, the platform and OEMs can define special permissions in order to restrict or protect access to dangerous actions. Special permissions are classified as the *appop* protection level. These are basically signature permissions for manufacturers and platform developers, hence the designation `signature|appop`.

2.2.4 Alternatives to permissions

As seen in Figure 2.2, modern Android documentation urges application developers to declare permissions only in the situations where alternatives are not available. Asking the user for a postal code or an address instead of declaring `ACCESS_COARSE_LOCATION` permission is suggested. Another example is the app using the system camera app, where it is encouraged to invoke `ACTION_IMAGE_CAPTURE`, or `ACTION_VIDEO_CAPTURE` instead of declaring the `CAMERA` permission. In the situations where the app needs to access content that another app has created, it is stated that instead of declaring `READ_EXTERNAL_STORAGE` permission, it is better to use the smartphone media store to open media files, and Storage Access Framework to open other files and miscellaneous documents. Media store is a framework that provides an index to media collections, allowing for retrieving and updating media files. Storage Access Framework has a similar functionality for the non-media content. Documentation indicates that as of Android 10 (API level 29), accessing the

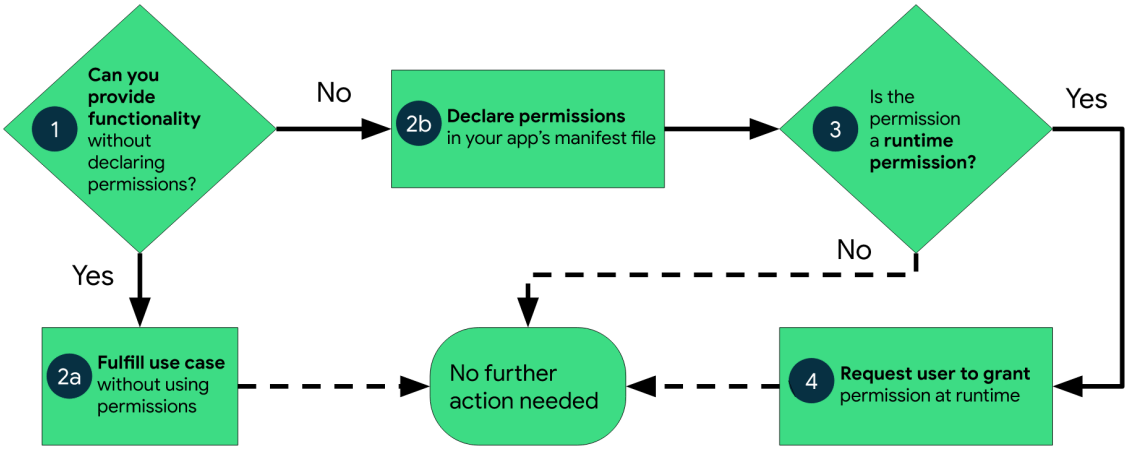

Figure 2.2: High-level workflow for using permissions on Android [11].

device IMEI is not possible. If the application requires identification of the device for whatever purpose, it needs to get a unique device identifier for specific app's instance using the Instance ID library. In the case of pairing a device over Bluetooth, using companion device pairing instead of declaring `BLUETOOTH_ADMIN`, and `ACCESS_FINE_LOCATION` permissions is recommended. Companion device pairing is a service for devices running on Android 8.0 (API level 26) and higher, that allows for Bluetooth or Wi-Fi scan of nearby devices. Barrera et al. had noted that in the previous Android versions, it was difficult to find a balance between finer and coarser grained permissions. This issue will further be discussed in the literature review section.

2.2.5 Custom App Permissions

In an effort to further protect Inter-Process Communication (ICP), Android also introduced custom permissions [10]. Apps can make their functionality available to other apps by defining new permissions that can be requested, in addition to defining automatically granted applications that share the same signature.

3 Literature Review

In this chapter, existing research is compiled. The prevailing range of issues facing the Android permission scheme that is popular among scholars include application overprivilege, analysis of app network usage, vulnerability analysis, attacks conducted without any permissions, OS version distribution, and most importantly, changes that came with the Android 6.0.

3.1 Overprivilege

According to Wang et al. in an 2013 paper, application overprivilege can be a serious problem in the Android permission scheme [21]. They claim that quite a lot of developers and users are hardly mindful of whether the requested permissions hold any functional purpose. In their analysis, they randomly select 50 apps from the Google Play Store and another third-party store, and find that overprivilege is a common problem. Although the results show that the issue is much more severe in the third-party store, the authors found that around 44% of the apps they tested requested unnecessary permissions.

A study on application overprivilege of 71 educational apps show worrying results [1]. Authors claim that depending on the users alone for permissions is was a poor solution for previous issues. In the experiment, they analyze every declared and used permission. Analysis showed that 25% of the total declared permissions were custom permissions. It was also found that approximately 80% of the apps declared more permissions than what they actually used. Alenezi et al. suggest an

integrated solution that detects discrepancies between declared and used permissions. They believe that even under development, an IDE should display whether the declared permissions are being used or not.

3.2 Application analysis for pre-2015 Android

In 2014, on their analysis of 35 Android applications, Taylor et al. found significant problems regarding network usage of apps [18]. All but one of the applications sent some amount of data encrypted using SSL, however 80% of the apps also sent information in plaintext. Well over half of these apps were responsible for privacy leaks. There is some reasoning behind why encrypted and unencrypted connections were used in combination. It may be due to the scalability issues that came with the SSL at the time, or simply the price of SSL certificates, both reasons resulting in a higher upkeep cost. Attackers can find out make and model of the device, including the OS version and installed apps. All of this information can lead an attacker to pinpoint specific exploits for particular devices. In addition, attacks on encrypted connections are also possible. Stöber et al. note that although the content is hidden from eavesdroppers, an attacker can gather side-channel information such as timing and data volume through fingerprinting periodic traffic patterns with a high success rate [17].

3.3 Network usage, privacy and security

Users do not explicitly see the network usage and security configuration of an application. This can lead to data theft, location tracking, or device damage [6]. Android Play Store's automated screening process to detect malware is unreliable and can be dodged. Approximately 70% of the network traffic generated by a device is unseen by the user. Android users can check the volume of the traffic, but the exact time and destination data is not easily available. It is difficult for a user to discern a malicious app that may be generating undesirable or potentially damaging traffic. Furthermore, if an app functions merely in the background, the user has little idea on the purpose of

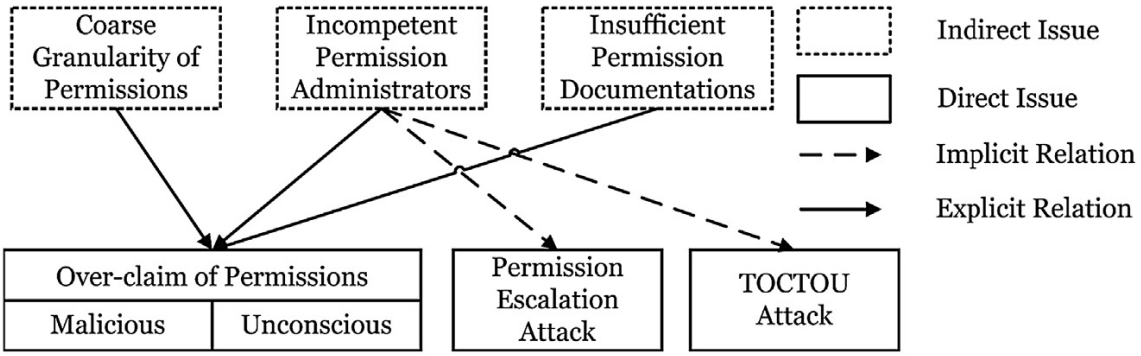

Figure 3.1: Relationship between issues in Android permission scheme [4].

the internet access beyond what the app claims. Ferreira et al. demonstrate that around 56% of connections set up by third party apps were on insecure ports. In addition, as Taylor et al. also mention, even the secure SSL connections are used widely incorrectly. Incorrect implementation of SSL can lead to passive eavesdropping attacks.

A 2014 paper from Fang et al. highlights the issues that come with the permission based security of Android, such as coarse-granularity of permissions, difficulty of permission administration, lack of adequate documentation, overprivilege, etc [4]. As shown in the Figure 3.1, they issue categorization is twofold; direct and indirect issues. Direct issues can cause leakage of private information, or monetary damage, while indirect issues may be utilized as a vehicle in an attack scenario. Direct issues include overprivilege, permission escalation attack, and TOCTOU (Time of Check to Time of Use) attack. Indirect issues include coarse granularity, incompetent permission administration, and lack of documentation. It is known that coarse grained permissions lead to difficulty for users to detect overprivilege. TOCTOU attack refers to some signature permissions granted for another application. Even if the app is uninstalled, the permissions are never revoked, so a new app with the same certificate would have access to same permissions without approval. Authors cite the `INTERNET` permission as an example for coarse granularity. A malicious developer might declare this permission to display advertisements in their app, at the same time, they can utilize this permission to access tolled websites secretly.

As it can be seen in the Figure 3.1, indirect issues can be implicitly or explicitly related to direct issues. Many permissions of this era give arbitrary access to device resources. It gives way for malicious apps to put on a mask of legitimacy. Manifest files are written by developers for requesting permissions. When this file is being written, the developer may or may not know in detail what kind of risks they entail. Some developers may take their time to educate themselves on the capabilities of these permissions, while others may simply claim them to make sure their app works.

3.4 Vulnerability analysis

Huang et al. conducts a general trend analysis to demonstrate the situation in the Android mobile vulnerability market [12]. An evaluation of a known vulnerability is conducted by taking note of the disclosure date of each CVE vulnerability, and calculating the disclosed vulnerabilities year by year since 2008. In addition, all vulnerabilities disclosed in NVD are also retrieved to estimate the percentage of vulnerabilities per year. It is observed from Figure 3.2 that vulnerabilities related to Android increased from 2008 to 2012. Following a small decline in 2013, there was an outbreak of vulnerabilities in 2014. The sharp escalation in 2014 is due to the fact that several Android software products could not validate SSL certificates given by HTTPS connections. Overall, authors conclude that security of the Android ecosystem have worsened over time. They divide the evolution of vulnerability market in three periods: period A (2008-2010) where majority of vulnerabilities are functional, period B (2009-2013) where market is still dominated by functional vulnerabilities, however management vulnerabilities arise (permission, privileges, and access control), and lastly period C (2014-2015) where management vulnerabilities such as CWE264 and CWE310 prevail.

Huang et al. analyzes vulnerability severity pattern by comparing all vulnerabilities in the software industry to relevant Android vulnerabilities. They find that average risk score for the Android market is greater than the industry for the most part. The average CVSS risk score for Android market is over 7, which can be considered as high risk, while the score for the indstry is between 6 and 7, which makes it medium risk. The software market vulnerabilities are decreasing over time.

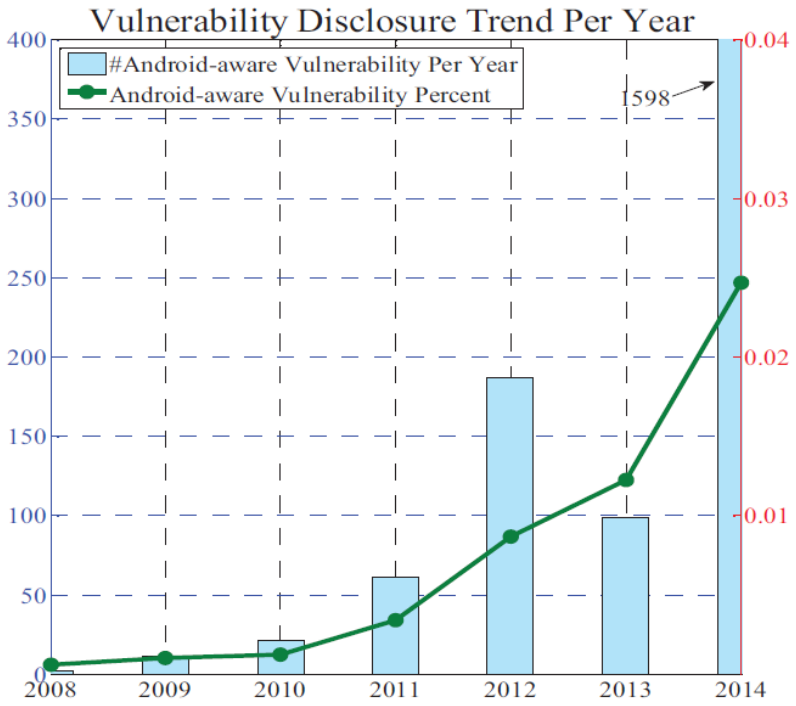

Figure 3.2: Vulnerability Disclosure Trend Per Year [12].

Furthermore, the percentage of high risk vulnerabilities to others are considerably higher to that of whole market. Therefore, one can state that Android market's security is worse than the whole software industry market.

CVSS includes an exploitability sub-score which indicates a vulnerability's likelihood to be exploited in terms of access vector, access complexity and authentication. It is noted that remotely exploitable vulnerabilites for the industry is approximately 90% with a slight decrease over the years, while the Android market's remotely exploitable vulnerabilities have an upwards trend and more than 90%. Access complexity is seen to be increasing, meaning that more sophisticated methods are required for exploitation, while the opposite is true for the Android market. Across the entire industry, the percentage of exploitable vulnerabilities without authentication is dropping. Overwhelming majority of the Android market vulnerabilities can be exploited without requiring authentication. As a result, the vulnerabilities in the Android market are more exploitable and easier to exploit over time than the vulnerabilities in the entire market. Strengthening the authentication process is the most effective approach for securing the Android market.

CVSS impact subscore assesses the adverse effects to a system if the vulnerability is successfully exploited. The confidentiality impact refers to the scope that attackers can access, integrity impact refers to what the attackers can alter, and availability impact refers to how attackers can affect the accessibility of system resources. Most vulnerabilities in the Android market have a complete confidentiality, integrity and avaiability impact. Hence, impact score for the Android market is higher than the entire market.

Authors conclude that number of vulnerabilities in the Android market is higher, and they are more exploitable. While the entire market is seeing an improvement, the security situation in Android market has been declining until 2015 in which the year this vulnerability analysis study was written.

3.5 Attacks without permissions

Kywe et al. demonstrate that even without any permissions, an attacker can gain access to device ID, phone service state, SIM card state, Wi-Fi and network information, and user configuration information [13]. First, they analyze unprotected APIs, which permit third-party apps to interact with device resources without permissions. Following this, they exhibit various attacks using the unprotected APIs. A rigorous process is required for retrieving unprotected APIs. Three types of static source-code analysis is performed to achieve this. A call graph analysis on system services to discover permissions without Linux ID checking and permission checking mechanisms on Android Interface Definition Language (AIDL); a component analysis on system applications for identification of unprotected broadcast receivers, services, and activities; a data-flow analysis for locating dynamically registered broadcasts. The analysis is implemented on Android Open Source Project (AOSP) version 5.1.0_rl and 4.4.0_rl. They find 735 unprotected APIs in system services for version 5.1.0_rl. The more recent version contains more unprotected APIs than the 4.4.0_rl due to additional functionalities added.

Following the discovery of unprotected APIs, Kywe et al. develop a third-party application that conducts Java reflection attacks, broadcast injection attacks, broadcast hijacking attacks, malicious activity launch attacks, activity hijacking attacks, malicious service launch attacks, and service hijacking attacks without requesting any permissions. The findings show that on the version 4.4.0_r1, the malicious app can block email synchronization, calendar events, and alter device settings, browser settings, Google documents. On the version 5.1.0_r1, the malicious app can see private user information such as country, Wi-Fi details, Bluetooth, cell signal strength, NFC, power state, SIM card state, and device ID strength. An attacker can control the volume of Android phones and play the ringtones, alarms, and notification tones that users have configured for calls and texts. Even if the target devices are secured, an attacker can disable Bluetooth discovery services and access camera, mail, music, and other device applications.

3.6 OS version distribution

In the iOS market, there exists no manufacturer variables. iOS devices are supported for seven years after the last time Apple sold that specific model. Over 80% of the devices with iOS in the market have the latest OS version installed [19]. From a security perspective, it is a good practice to use the latest OS version in a device, however, this is not a reality for the Android market. Distribution of Android OS versions in the market is affected by variables such as the large number of different types of devices and numerous manufacturers who sell these devices. On average, an Android smartphone is supported two to three years. This results in a considerable number of devices with outdated and unsupported versions being actively used in the market [15]. Figure 3.3 demonstrates that approximately 60% of the devices are running on an Android OS that is over two years old. Despite being six years old, Android 6.0 was the second most widely used version in 2020. Over 15% of the devices are being run with OS versions between 4.0 and 5.1. In a market with well over two billion smartphones, these percentages are worrying to say the least.

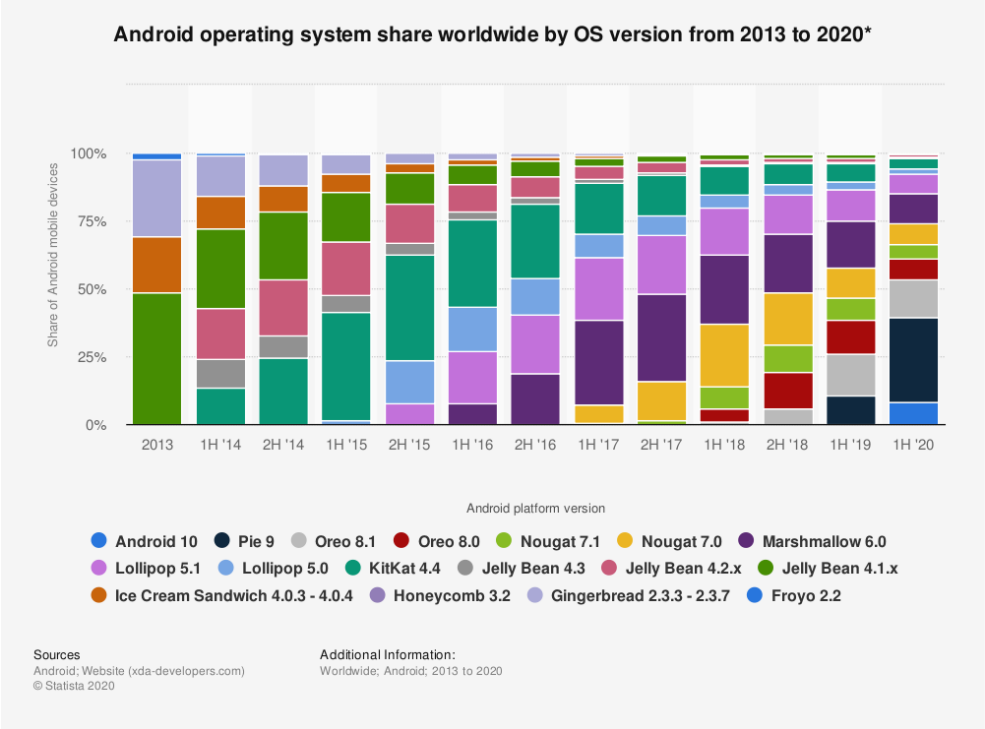

Figure 3.3: Android OS version distribution in the worldwide market [15].

3.7 Changes with Android 6.0

As mentioned before, after Android 6.0, the permissions were divided into install-time and runtime permissions. Normal and signature permissions were granted during install, while dangerous permissions granted at runtime, with option given to the user to revoke them at any time. These runtime permissions were divided into groups such as the one shown in the Figure 3.4. Special protected API calls were introduced to `PackageManager` to provide runtime permissions, allowing users to grant and remove access dynamically. New APIs were also added, allowing app developers to check if permissions are granted at runtime and, if necessary, request them. If one permission from a group was granted to an application, the other permissions in that group were granted to that app as well. Zhauniarovich et al. claim that Android 6.0 has undergone a change for the worse, where a more coarse-grained structure is implemented rather than the opposite [23]. In the previous versions, apps with the same digital signature as the package declaring the permission were granted signature permissions at install time. However, Android has introduced various new kinds

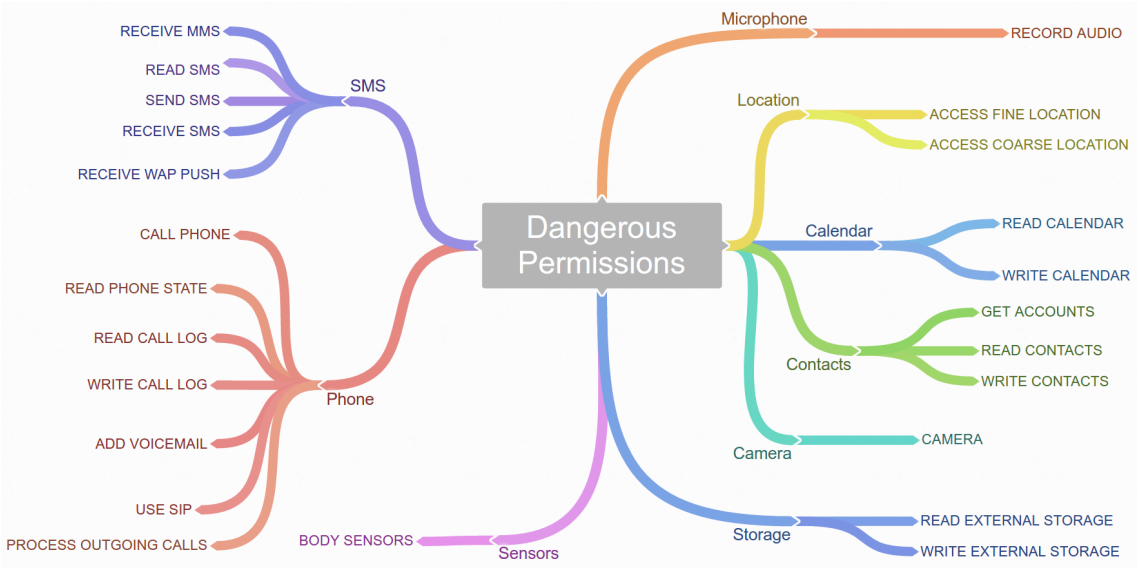

Figure 3.4: Groups for dangerous permissions [2].

of signature permissions that can be requested by third-party apps that do not meet this requirement. Moreover, 22 permissions that were previously regarded as dangerous are now granted at install time, with no way to revoke them by the user. Permissions such as `BLUETOOTH`, `NFC` and `INTERNET` need not be approved by the user anymore.

Since many devices in use were running previous versions of Android, new apps had to be forward compatible with older versions. Google's new special compatibility library included the API calls for checking granted permissions. Authors discovered that if the app is ran on an older version of Android that hasn't yet declared the permission, API call from special compatibility library returns *denied* for some permissions that are not even required in the first place. Since older versions of Android are hardly supported, it is highly difficult to apply a fix to the problem. Authors suggest that it should at least be noted in the documentation, so the developers can look out for it.

The AppOps system allows users to give and revoke permissions at runtime through a dedicated user interface within the Settings, allowing old apps to work with the new version of Android for backwards compatibility. AppOps, on the other hand, solely manages platform permissions and so is unable to enforce developer-defined custom dangerous permissions. When a legacy app is installed on a device, the user is indeed prompted with the pre-6.0 type of permission alert box. If

the user does not consent to the prompted permissions, the app will not be installed. The intention was to make legacy and new apps behave in the same manner, but that did not happen according to Zhauniarovich et al. Backwards compatibility also opened a way for malicious app developers to take advantage of post-Marshmallow OS versions, specifically creating apps that do not traverse beyond API Level 23, so that their apps do not hold any runtime permissions, and all their dangerous permissions accepted at install-time [2]. Since the user is accustomed to the new permission system, initially accepting what the app asks might not seem as dangerous.

Zhauniarovich et al. take issue with runtime permissions being granted per permission groups. They claim that while this reduces the amount of interruptions a user receives during runtime, however, users who want to have a finer-grained permission control on their devices are not being considered. Alepis et al. also give an example to the problems come with dangerous permission groups. They consider a use case where an app that requires access to phone functionalities, and declares the permission `READ_PHONE_STATE`. The user will be prompted by an alert box stating Ällow application to make and manage phone calls?". This vague statement will allow the app to read and change the call history, and make phone calls without user notification.

Alepis et al. also argue that after Android 6.0, transparency of permissions have decreased, and a fine-grained control is not being offered, and the new system altogether is not an improvement to the pre-6.0 versions [2]. Authors also note users are not completely informed what the app actually holds for the user. Play Store goes as far as stating that the app to be installed does not require special access while in reality, for some cases the opposite is true. The specific timing for asking runtime permissions are problematic as well, usually asking just after the first launch of an app, instead of asking at a time when the app needs access. Another transparency issue comes with apps that only ask for normal (install-time) permissions. In a case where an app asks for both dangerous and normal permissions, users are able to see both types on permissions in the Settings app. However, if an app only declares normal permissions in its Manifest file, users cannot see any permissions in the Settings app, this capability is not enabled. This might lead to some

serious security problems, such as an app declaring absolutely no permissions at install-time, and then adding an arbitrary number of normal permissions with updates, leaving the user without any control for the matter.

4 Methodology

The research methodology includes an extensive literature review located in Chapter 3. The experimentation is carried out with two devices, of which the specification is explained in Section 4.1. For the first part of the investigation, two applications are built using Android Studio IDE using Java programming language. Section 4.2 gives an overview of the applications. Second part of the investigation consists of rooting the devices, and noting the differences between the retrievable information, in addition to attempting to provide the apps with superuser privileges. The experimentation is conducted with a focus on confidentiality principle of information security. The findings are analyzed in Chapter 6. Comparisons are drawn based on the differences across devices and applications.

4.1 Test Device Specification

In order to conduct the experiment, two smartphones of which one employing older and one newer OS versions were acquired. The older device is an HTC One SV, which is equipped with an Android version 4.2.2 (API level 17). The newer device is a BQ Aquaris X Pro, which has the 8.1.0 version (API level 27) of Android. Device information for both smartphones can be seen in the Figure 4.1. These two devices cover the version discrepancy of the Android devices in the market, enabling us to get a broad idea on both sides of the coin.

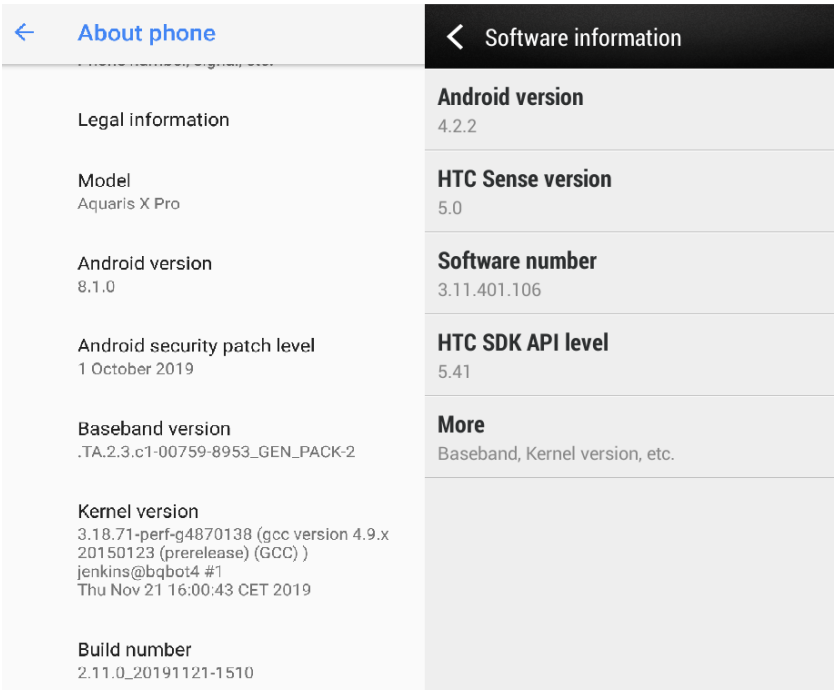

Figure 4.1: Device information pages for the two smartphones (left: BQ Aquaris X Pro, right: HTC One SV).

4.2 Application Overview

Since the app does not have any functionality for the user, it is built upon an empty project template from the Android Studio IDE. The app differs with respect to the permission declaration, where one would have to request permissions in the BQ device during runtime, as opposed to the app for the HTC device, where all the permissions are requested during install-time.

4.2.1 The application for the BQ device

First app to be built was for the BQ device. It was previously mentioned that the app would have two states, one state with all permissions requested from the user, and another state with zero declared permissions.

For the first state with all permissions declared, the initial action was to list all permissions in the alphabetical order using the completion assist in Android Studio. This resulted in a grand total of 169 permissions declared in the `AndroidManifest.xml` file. However, Android Studio immediately flagged a considerable number of these permissions for various reasons such as being deprecated or being reserved for the system applications. Of all the permissions listed, 89 were shown as reserved for the Android system; or they were signature permissions, thus commented out of the manifest file. Nine of the permissions listed were seen as deprecated. Seven of these permissions were commented out except for `PROCESS_OUTGOING_CALLS` and `USE_FINGERPRINT` because these two were deprecated later, and were valid for API level 27. The remaining 73 permissions were categorized as install-time or runtime because this distinction makes it easy to see which permissions need to be actually requested from the user. Since the app was configured to work with devices up to API level 27, the runtime permissions that were added after this level were also commented out. These include one API level 28 permission, namely `ACCEPT_HANDOVER`, and three API level 29 permissions, which are `ACCESS_BACKGROUND_LOCATION`, `ACCESS_MEDIA_LOCATION`, and `ACTIVITY_RECOGNITION`. The final number of runtime permissions was 27, while 42 permissions were listed under the install-time category. Lastly, runtime permissions were divided to their nine specified permission groups. These permissions can be seen in the Table 4.1.

Following the declaration of permissions in the `AndroidManifest.xml` file, a permission request button was created [7]. Next, the code was expanded to request all declared permissions with that one button. In this version of Android, even if one declares runtime permissions in the manifest file, the app will behave as if no permissions are required at install, unless they are specifically requested.

The second state of the app is much simpler, which only includes creating a basic application out of an empty template. Alepis et al.'s extensive 2017 study mentioned if the app only declares normal permissions in its manifest file, the user would not be notified and would not be able to see what normal permissions were declared in Settings. Therefore, the app contains no runtime permissions, but all install-time permissions. This notion is further explained in Section 5.2.

Runtime Permissions	
Permission Groups	Permissions
CONTACTS	GET_ACCOUNTS READ_CONTACTS WRITE_CONTACTS
CALENDAR	READ_CALENDAR WRITE_CALENDAR
SMS	SEND_SMS RECEIVE_SMS READ_SMS RECEIVE_WAP_PUSH RECEIVE_MMS
STORAGE	READ_EXTERNAL_STORAGE WRITE_EXTERNAL_STORAGE
LOCATION	ACCESS_FINE_LOCATION ACCESS_COARSE_LOCATION
PHONE	READ_PHONE_STATE READ_PHONE_NUMBERS CALL_PHONE READ_CALL_LOG WRITE_CALL_LOG ADD_VOICEMAIL USE_SIP PROCESS_OUTGOING_CALLS ANSWER_PHONE_CALLS
MICROPHONE	RECORD_AUDIO
CAMERA	CAMERA
SENSORS	BODY_SENSORS USE_FINGERPRINT

Install-time Permissions
ACCESS_NOTIFICATION_POLICY
ACCESS_WIFI_STATE
BLUETOOTH
BLUETOOTH_ADMIN
BROADCAST_STICKY
CALL_COMPANION_APP
CHANGE_NETWORK_STATE
CHANGE_WIFI_MULTICAST_STATE
CHANGE_WIFI_STATE
DISABLE_KEYGUARD
EXPAND_STATUS_BAR
FOREGROUND_SERVICE
ACCESS_LOCATION_EXTRA_COMMANDS
ACCESS_NETWORK_STATE
GET_PACKAGE_SIZE
INSTALL_SHORTCUT
INTERNET
KILL_BACKGROUND_PROCESSES
MANAGE_OWN_CALLS
MODIFY_AUDIO_SETTINGS
NFC
NFC_PREFERRED_PAYMENT_INFO
NFC_TRANSACTION_EVENT
READ_SYNC_SETTINGS
READ_SYNC_STATS
RECEIVE_BOOT_COMPLETED
REORDER_TASKS
REQUEST_COMPANION_RUN_IN_BACKGROUND
REQUEST_COMPANION_USE_DATA _IN_BACKGROUND
REQUEST_DELETE_PACKAGES
REQUEST_IGNORE_BATTERY_OPTIMIZATIONS
REQUEST_PASSWORD_COMPLEXITY
SET_ALARM
SET_WALLPAPER
SET_WALLPAPER_HINTS
TRANSMIT_IR
UNINSTALL_SHORTCUT
USE_BIOMETRIC
USE_FULL_SCREEN_INTENT
VIBRATE
WAKE_LOCK
WRITE_SYNC_SETTINGS

Table 4.1: Declared permissions in AppBQ_full application

4.2.2 The application for the HTC device

The creation of the HTC app for the full permissions state was fairly similar to that of the BQ app. Using the Android documentation [16], a categorized list of all permissions were written into the `AndroidManifest.xml` file. In total, there were approximately 200 permissions listed in the file. 70 permissions under 25 different categories were declared, while the remaining ones were flagged as signature, or as reserved for system applications. Of all the declared permissions, 46 were classified as dangerous protection level, while 24 of them as normal protection level. The full list of permissions can be seen in the Table 4.2.

Declared Permissions		
Permission Groups	Permissions	Protection Level
MESSAGES	SEND_SMS	dangerous
	RECEIVE_SMS	dangerous
	RECEIVE_MMS	dangerous
	READ_SMS	dangerous
	WRITE_SMS	dangerous
	RECEIVE_WAP_PUSH	dangerous
SOCIAL_INFO	READ_CONTACTS	dangerous
	WRITE_CONTACTS	dangerous
	READ_CALL_LOG	dangerous
	WRITE_CALL_LOG	dangerous
	READ_SOCIAL_STREAM	dangerous
	WRITE_SOCIAL_STREAM	dangerous
PERSONAL_INFO	READ_PROFILE	dangerous
	WRITE_PROFILE	dangerous
CALENDAR	READ_CALENDAR	dangerous
	WRITE_CALENDAR	dangerous

USER_DICTIONARY	READ_USER_DICTIONARY	dangerous
	WRITE_USER_DICTIONARY	normal
BOOKMARKS	READ_HISTORY_BOOKMARKS	dangerous
	WRITE_HISTORY_BOOKMARKS	dangerous
DEVICE_ALARMS	SET_ALARM	normal
VOICEMAIL	ADD_VOICEMAIL	dangerous
LOCATION	ACCESS_FINE_LOCATION	dangerous
	ACCESS_COARSE_LOCATION	dangerous
	ACCESS_LOCATION_EXTRA_COMMANDS	normal
NETWORK	INTERNET	dangerous
	ACCESS_NETWORK_STATE	normal
	ACCESS_WIFI_STATE	normal
	CHANGE_WIFI_STATE	dangerous
BLUETOOTH_NETWORK	BLUETOOTH	dangerous
	BLUETOOTH_ADMIN	dangerous
	NFC	dangerous
ACCOUNTS	GET_ACCOUNTS	normal
	AUTHENTICATE_ACCOUNTS	dangerous
	USE_CREDENTIALS	dangerous
	MANAGE_ACCOUNTS	dangerous
AFFECTS_BATTERY	CHANGE_WIFI_MULTICAST_STATE	dangerous
	VIBRATE	normal
	FLASHLIGHT	normal
	WAKE_LOCK	normal
AUDIO_SETTINGS	MODIFY_AUDIO_SETTINGS	normal
MICROPHONE	RECORD_AUDIO	dangerous
CAMERA	CAMERA	dangerous

PHONE_CALLS	PROCESS_OUTGOING_CALLS	dangerous
	READ_PHONE_STATE	dangerous
	CALL_PHONE	dangerous
	USE_SIP	dangerous
STORAGE	READ_EXTERNAL_STORAGE	dangerous
	WRITE_EXTERNAL_STORAGE	dangerous
SCREENLOCK	DISABLE_KEYGUARD	dangerous
APP_INFO	GET_TASKS	dangerous
	REORDER_TASKS	normal
	RESTART_PACKAGES	normal
	KILL_BACKGROUND_PROCESSES	normal
DISPLAY	SYSTEM_ALERT_WINDOW	dangerous
WALLPAPER	SET_WALLPAPER	normal
	SET_WALLPAPER_HINTS	normal
STATUS_BAR	EXPAND_STATUS_BAR	normal
SYNC_SETTINGS	READ_SYNC_SETTINGS	normal
	WRITE_SYNC_SETTINGS	normal
	READ_SYNC_STATS	normal
SYSTEM_TOOLS	PERSISTENT_ACTIVITY	normal
	GET_PACKAGE_SIZE	dangerous
	RECEIVE_BOOT_COMPLETED	normal
	BROADCAST_STICKY	normal
	SUBSCRIBED_FEEDS_READ	normal
	SUBSCRIBED_FEEDS_WRITE	dangerous
	CHANGE_NETWORK_STATE	dangerous

Table 4.2: Declared permissions in AppHTC_full application

For the second state of the app, no permissions were declared. Unlike Android 8.1.0, version 4.2.2 notifies the user at install-time about the declared permissions even if no dangerous permissions were declared. Thus, this state of the application was just an empty project template set at minimum SDK 17.

5 Experiments

It is important to remember the main objective of this thesis. The question to be answered was the leakage of information, such as PII, which is strictly a confidentiality issue. The permissions to be evaluated would be the ones that had a probability of enabling access to a resource that contains information, rather than giving access to alter or remove the data, which means that in other words, security principles aside from confidentiality such as integrity, availability, or non-repudiation is ignored.

5.1 AppBQ_full

This is the full permission state of the application for the BQ Aquaris X Pro which is equipped with Android 8.1.0 (API level 27) Oreo OS. In this section, the functionalities of requested runtime permissions will be explored. It will also be tested whether requesting one permission from a permission group enables all permissions in a group. The declared permissions are divided into nine permission groups. These permission groups are contacts, calendar, SMS, storage, location, phone, microphone, and camera.

When the user taps `AppBQ_full.apk` to install the app, Android greets them with a message saying "Do you want to install this application? It does not require any special access.". The app has one centred button, stating "REQUEST ALL PERMISSIONS". If the user taps this button, the app then requests all declared permission through nine different prompts for all permission groups These screens can be seen in Figure 5.1.

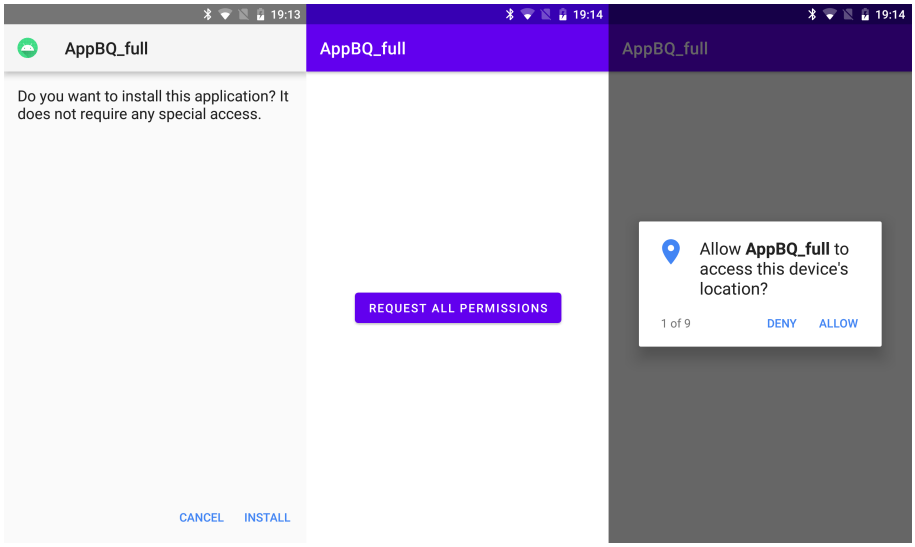

Figure 5.1: Screens upon installation of the app with full permissions on BQ Aquaris X Pro.

5.1.1 Permission groups

In order to store the permission data, a small `logToFile` method is created, where every time it is called, it appends the data into a text file in the device storage. This method can be seen below.

```
private void logToFile(Context context, String sBody) {
  File dir = new File(context.getFilesDir(), "permLogs");
  if (!dir.exists()){
    dir.mkdir();
  }
  try {
    File gpxfile = new File(dir, "plogs.txt");
    FileWriter writer = new FileWriter(gpxfile, true);
    writer.append(sBody).append("\n\n");
    writer.flush();
    writer.close();
    Toast.makeText(this, "Data logged to " + context.getFilesDir(), Toast.LENGTH_LONG).show();
  } catch (Exception e) {
    e.printStackTrace();
  }
```

```
}
```

Using `context.getFilesDir()`, the `logToFile()` method creates a "permLogs"folder inside the application path. The method checks if the directory already exists, and if it does, it moves onto creating the text file called "plogs.txt". When the method is called, the `sBody` string argument is passed to the `writer.append()` method, so the text in the data stream is appended into the text file. After flushing and closing this stream, a small toast appears in the screen, indicating where the log file is stored.

Contacts permission group

The `android.permission-group.CONTACTS` is used for runtime permissions related to contacts and profiles on a device. Out of the three permissions declared belonging to this group, two are of interest: `GET_ACCOUNTS` and `READ_CONTACTS`. The `GET_ACCOUNTS` permission allows access to the list of accounts in the Accounts Service. When the method below is called, it first creates a new `ArrayList`. Using the `AccountManager`, every e-mail account in the device is written into the array list, then to the text file using `logToFile()` method.

```
private String getAcc() {
  List<String> accountList = new ArrayList<String>();
  Pattern gmailPattern = Patterns.EMAIL_ADDRESS;
  Account[] accounts = AccountManager.get(this).getAccounts();
  for (Account account : accounts) {
    if (gmailPattern.matcher(account.name).matches()) {
      accountList.add(account.name);
    }
  }
  return accountList.toString();
}
```

This results in a folder creation in the application directory, and a .txt file with the user's e-mail written inside. The e-mail address "testsp.bq@gmail.comßeen in the Figure 5.2 was created to demonstrate this permission.

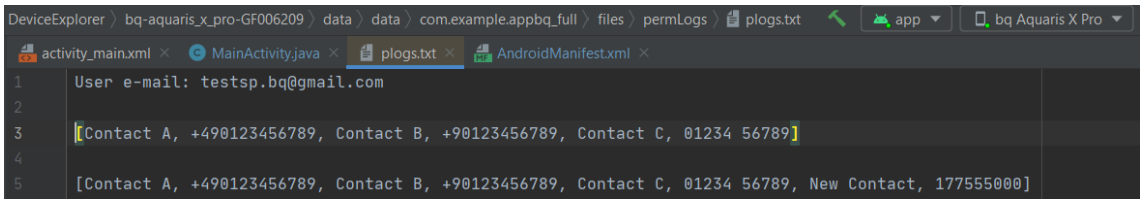

Figure 5.2: Text file for the permission log showing user e-mail.

A similar method was created to write the contacts list in the same "plogs.txt"file using `READ_ACCOUNTS` permission. The method `getContacts()` is a String type method that returns a list of contacts and their phone numbers. `CurrentResolver` allows the navigation of contacts database, which the contact id and phone number is found. After retrieving this data, it is added to the array list `contactList`, which is also what this method returns after typecasting it to string.

```
private String getContacts() {
  ArrayList<String> contactList = new ArrayList<String>();
  ContentResolver cr = getContentResolver();
  Cursor cur = cr.query(ContactsContract.Contacts.CONTENT_URI, null, null, null, null);
  if ((cur != null ? cur.getCount() : 0) > 0) {
    while (cur != null && cur.moveToNext()) {
      String id = cur.getString(cur.getColumnIndex(ContactsContract
              .Contacts._ID));
      String name = cur.getString(cur.getColumnIndex(ContactsContract
              .Contacts.DISPLAY_NAME));
      contactList.add(name);
      if (cur.getInt(cur.getColumnIndex(ContactsContract
              .Contacts.HAS_PHONE_NUMBER)) > 0) {
        Cursor pCur = cr.query(ContactsContract.CommonDataKinds
                .Phone.CONTENT_URI, null,
        ContactsContract.CommonDataKinds.Phone.CONTACT_ID + " = ?", new String[]{id}, null);
        while (pCur.moveToNext()) {
```

```
          String phoneNo = pCur.getString(pCur.getColumnIndex(ContactsContract
                  .CommonDataKinds.Phone.NUMBER));
          contactList.add(phoneNo);
        }
        pCur.close();
      }
    }
  }
  if (cur != null) {
    cur.close();
  }
  return contactList.toString();
}
```

Only contact names and phone numbers are listed, however any available contact information can be retrieved such as profile pictures and e-mail addresses. The returned array list can be passed as an argument to the logger method seen before.

Since the special profile permissions were removed by Android, the `ContactsContract.Profile` class was also moved to the contacts permission. Thus, I can use the exact same `getProfile()` method located in AppHTC_full app, to retrieve device owner's name and profile ID. The permission entry log relevant to this method can be seen below:

```
[Name: Cagri Erdem - ID: 9223372034707292161]
```

Calendar permission group

The `android.permission-group.CALENDAR` is used for runtime permissions related to user's calendar. `READ_CALENDAR`, and `WRITE_CALENDAR` permissions were declared from this group. Former permission on reading the calendar information concerns confidentiality, and the method used for this permission is the `getCalendar()` method. It simply gathers information using the content resolver, and a cursor to navigate around the `CalendarContract.Events` class.

```
private String getCalendar() {
  ContentResolver cr = getContentResolver();
  Cursor c = cr.query(CalendarContract.Events.CONTENT_URI, null, null, null, null);
  List<String> calendarInfo = new ArrayList<>();
  while (c.moveToNext()) {
    Date startDate = new Date(c.getLong(c.getColumnIndex(CalendarContract.Events.DTSTART)));
    Date endDate = new Date(c.getLong(c.getColumnIndex(CalendarContract.Events.DTEND)));
    calendarInfo.add("\naccount name: " + c.getString(c.getColumnIndex(CalendarContract.Events
.ACCOUNT_NAME)) +
    "\ncalendar display name:" + c.getString(c.getColumnIndex(CalendarContract.Events.
CALENDAR_DISPLAY_NAME)) +
    "\ntitle: " + c.getString(c.getColumnIndex(CalendarContract.Events.TITLE)) +
    "\nlocation: " + c.getString(c.getColumnIndex(CalendarContract.Events.EVENT_LOCATION)) +
    "\nevent start: " + startDate +
    "\nevent end: " + endDate);
  }
  c.close();
  return calendarInfo.toString();
}
```

Since the pre-installed calendar is app belongs to Google, the user e-mail is automatically registered to it, and as a result, the user e-mail was visible just with this permission in the entries. It enters the local holidays based on locality, so the permission log entry was filled with calendar entries such as the following.

```
title: New Year's Eve
location:
event start: Fri Dec 31 03:00:00 GMT+03:00 2021
event end: Sat Jan 01 03:00:00 GMT+03:00 2022,
account name: testsp.bq@gmail.com
calendar display name: Holidays in Turkey
title: New Year's Day
location:
```

```
event start: Sat Jan 01 03:00:00 GMT+03:00 2022
event end: Sun Jan 02 03:00:00 GMT+03:00 2022,
account name: testsp.bq@gmail.com
calendar display name: Holidays in Turkey
```

If the user had made any special events such as flights or meetings in the calendar, it would also show up here.

SMS permission group

The `android.permission-group.SMS` is used for runtime permissions related to user's SMS messages. Five permissions were declared in this group. However, only `RECEIVE_SMS` and `READ_SMS` was used. Since the code is exactly the same as the methods `readSms()` and `receiveSms()`, it is not demonstrated here for a second time. One can consult AppHTC_full section, or the appendix for the code.

Storage permission group

The `android.permission-group.STORAGE` is used for the runtime permissions related to shared external storage of a device. Two permissions were declared from this group; `READ_EXTERNAL_STORAGE`, and `WRITE_EXTERNAL_STORAGE`. These permissions enable the previously seen `logToFile()` method to write in internal shared storage of the device.

Location permission group

The `android.permission-group.LOCATION` is used for permissions that allow access the device location. Android version 8.1.0 has two location permissions for different sensitivities. `ACCESS_COARSE_LOCATION` allows applications to use cellular and Wi-Fi signals to get a rough location of the device, using less resources than `ACCESS_FINE_LOCATION`. The latter permission enables the

use of GPS signals in addition to cellular and Wi-Fi signals, getting a highly accurate position of the device. For instance, following method can get the last known location of the device using only the coarse location permission.

```
private String getLoc() {
  LocationManager lm = (LocationManager) getSystemService(Context.LOCATION_SERVICE);
  List<String> providers = lm.getProviders(true);
  Location loc = null;

  if (getApplicationContext().checkCallingOrSelfPermission
      (Manifest.permission.ACCESS_COARSE_LOCATION) == PackageManager.PERMISSION_GRANTED) {
    for (int i = providers.size() - 1; i >= 0; i--) {
      loc = lm.getLastKnownLocation(providers.get(i));
      if (loc != null) break;
    }
  }
  double[] pos = new double[2];
  if (loc != null) {
    pos[0] = loc.getLatitude();
    pos[1] = loc.getLongitude();
  }
  String s = pos[0] + ", " + pos[1];
  return s;
}
```

This returns the following location entry in the permission log file:

```
40.9826775, 27.550750699999995
```

Phone permission group

The `android.permission-group.PHONE` is used for permissions that are associated telephony features. Following permissions were decided to be related to the scope of this thesis. `READ_PHONE_STATE` allows read-only access to phone state, including the phone number of the device, current cellular network information, the status of any ongoing calls, and a list of any `android.telecom.PhoneAccount`'s registered on the device. `READ_PHONE_NUMBERS` is a subset of the capabilities granted by previous permission, but is exposed to instant applications. `READ_CALL_LOG` allows reading of the user's call log. `PROCESS_OUTGOING_CALLS` allows an application to see the number being dialed during an outgoing call with the option to redirect the call to a different number or abort the call altogether. The `getTelephonyInfo()` and `getBuildInfo()` methods are being copied from AppHTC_full, and pasted here to get relevant information. Few differences can be observed instantly, such as the methods to retrieve IMEI and IMSI are being locked behind the privileged phone state permission, and the method to get owner phone number being tied to read phone numbers permission. The IMEI and IMSI retrieval actions show the error that can be observed in Figure 5.3.

Figure 5.3: The IDE error upon IMEI and IMSI retrieval actions.

However, the privileged phone state permission is a system permission, and it is impossible to declare and request that permission. Interestingly, these errors also do not block the app from running, and after accepting the requested permission, IMEI and IMSI show up in the permission log along the other information as follows:

```
--- TELEPHONY ---
[Phone Type: GSM
, IMEI: 358627081924019
, IMSI: 286039521026739
, Network Operator: Pttcell
, SIM Operator: Pttcell - 28603
```

```
, Phone number: +905061901740]

--- BUILD ---
[Manufacturer: bq, Model: Aquaris X Pro, Serial number: GF006209, Bootloader: BOOT.BF.3.3,
Display: 2.11.0_20191121-1510]

--- CALL LOG ---
[Phone Number: 5331928099 Call Type: 2 Date: Mon Aug 09 18:08:50 GMT+03:00 2021 Duration: 0,
Phone Number: 5331928099 Call Type: 2 Date: Mon Aug 09 18:09:13 GMT+03:00 2021 Duration: 0,
Phone Number: 05079776076 Call Type: 1 Date: Mon Aug 09 18:19:20 GMT+03:00 2021 Duration: 49]
```

After some research, I found out that the new privileged phone state permişsion was introduced with Android 10, and since this is version is 8.1.0, the error did not make the app crash, just as it did not for the HTC app as well. This of course, means that any app with a minimum SDK less then 29 will have access to this data using the phone state permission.

Microphone, camera, and sensor permission groups

The `android.permission-group.MICROPHONE` is used for permissions that are associated with accessing microphone. It is noted that phone calls also capture audio, but that functionality is in a separate and more visible permission group. Only one permission is located here, which is `RECORD_AUDIO`.

The `android.permission-group.CAMERA` is used for permissions that are associated with accessing camera or capturing images and video from the device. The only permission here is `CAMERA`.

The `android.permission-group.SENSORS` is used for permissions that are related to sensors of the device that capture and measure information about user's body such as heart rate, daily steps (through `BODY_SENSORS`) or fingerprint (through `USE_FINGERPRINT`).

The permissions in these groups are mostly related to utilizing phone's hardware functionality, thus, they were not demonstrated here.

Normal permissions

It was discovered that some information that normal permissions gave access to does not function anymore. Particularly, I found out that many network related permissions were listed as normal, however the information gathered was locked behind the location permissions. Therefore, this section contains the analysis of the results gathered from the AppBQ_none app's code pasted into the full permission state version. The reader may benefit from reading the AppBQ_none section before this one, since the comparisons are drawn based on that.

For the `wifiInfo()` method, the results are somewhat different. The device Wi-Fi MAC address is blocked from developers completely, so only a meaningless constant value is seen, just as the other version. Connected Wi-Fi SSID and BSSID can clearly be examined this time, whereas the former would return blank, and the latter would return the same constant 02:00:00:00:00:00 in the app with no runtime permissions. Rest of the values have similar results.

```
Device Wi-Fi Mac address: 02:00:00:00:00:00 - Connected Wi-Fi: SSID: Ulas, BSSID: 00:1c:7b:f9:
c5:b4, MAC: 02:00:00:00:00:00, Supplicant state: COMPLETED, RSSI: -46, Link speed: 72Mbps,
Frequency: 2437MHz, Net ID: 2, Metered hint: false, score: 60
```

For comparison, the reader can also see "WIFI-INFOëntry for the AppBQ_none application.

Previously connected Wi-Fi networks returned the same results, with SSID's and security configurations filled but without MAC addresses. It turns out that this information is not saved, or it is not available after disconnecting.

The Wi-Fi scan was entirely different, because without runtime permissions, the scan was not conducted at all. In this version, the scan is conducted, and the scanned networks are written into the log file as such:

```
[SSID: NetMASTER Uydunet-9D7D, BSSID: 3a:6b:1c:03:e9:41, capabilities: [WPA-PSK-CCMP][WPA2-PSK
-CCMP][ESS], level: -43, frequency: 2422, timestamp: 1114235528308, distance: ?(cm),
distanceSd: ?(cm), passpoint: no, ChannelBandwidth: 1, centerFreq0: 2432, centerFreq1: 0,
80211mcResponder: is not supported, Carrier AP: no, Carrier AP EAP Type: -1, Carrier name:
null,
```

SSID: Ulas, BSSID: 00:1c:7b:f9:c5:b4, capabilities: [WPA2-PSK-CCMP][ESS], level: -48, frequency: 2437, timestamp: 1114235528390, distance: ?(cm), distanceSd: ?(cm), passpoint: no, ChannelBandwidth: 0, centerFreq0: 0, centerFreq1: 0, 80211mcResponder: is not supported, Carrier AP: no, Carrier AP EAP Type: -1, Carrier name: null,

SSID: SUPERONLINE_WiFi_7889, BSSID: 94:0b:19:5e:97:0f, capabilities: [WPA-PSK-TKIP+CCMP][WPA2-PSK-TKIP+CCMP][ESS][WPS], level: -80, frequency: 2462, timestamp: 1114235528478, distance: ?(cm), distanceSd: ?(cm), passpoint: no, ChannelBandwidth: 0, centerFreq0: 0, centerFreq1: 0, 80211mcResponder: is not supported, Carrier AP: no, Carrier AP EAP Type: -1, Carrier name: null,

SSID: TURKSAT-KABLONET-0133-2.4G, BSSID: 18:48:59:1b:49:35, capabilities: [WPA2-PSK-CCMP][ESS], level: -85, frequency: 2472, timestamp: 1114235528561, distance: ?(cm), distanceSd: ?(cm), passpoint: no, ChannelBandwidth: 0, centerFreq0: 0, centerFreq1: 0, 80211mcResponder: is not supported, Carrier AP: no, Carrier AP EAP Type: -1, Carrier name: null]

For Bluetooth findings, the MAC address and the previously paired devices list returned the same contents with the app's no runtime permissions state. This time, the Bluetooth discovery was not blank, and returned the Bluetooth devices that are ready to pair in the environment.

[--- Name: OPPO A9 2020 Address: A4:C9:39:AA:4E:2C Contents: 0 Class: 5a020c Type: 1 UUIDs: null---,
--- Name: Jaybird Tarah Address: C0:28:8D:A1:92:49 Contents: 0 Class: 240404 Type: 1 UUIDs: null---,
--- Name: [TV] UE65JU7500 Address: FC:8F:90:29:41:3F Contents: 0 Class: 8043c Type: 3 UUIDs: null---]

5.1.2 Permission grants per group

It is observed that in Android 8.1.0, when a permission is requested, the user is only asked to confirm the group that permission belongs to, as shown in the Figure 5.1. However, since 6.0, some changes have been made, and requesting one permission from a group does not grant all permissions in that group anymore. Google added the requirement that developer calls `requestPermissions()` for specific permissions, and not just one of the group. User still grants by the group, but if the developer fails to request the specific permission, the access is not given to that resource.

5.2 AppBQ_none

For this state of application for the BQ device with Android 8.1.0, wording for the objective is somewhat changed. This means that no permission state indicates that there are no runtine permissions requested, but the install-time permissions that can be seen in Table 4.1 remain declared.

The following reasoning is given why the install-time permissions are declared. When the app installation is executed from the .apk file, Android again notifies the user that the app requires no special access. In the AppBQ_full app, the user could see the install-time permissions along with the runtime permissions in Settings, however this time user cannot see any declared permissions, even though there are a grand total of 42 permissions declared. This can be seen in the Figure 5.4. The validity of this, was also tested in Android R version 11 (API level 30), which still do not show any declared permissions in Settings for the app. Although it is worth noting that this observation was made on a device created on AVD (Android Virtual Device) Manager, and not with an actual smartphone.

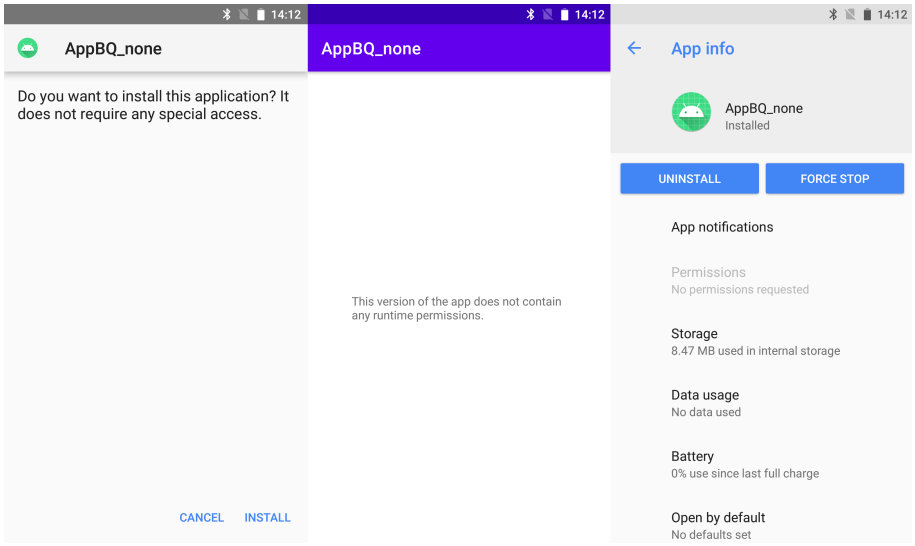

Figure 5.4: BQ application installation screens with no permissions requested.

This time the data gathered from the user cannot simply be written into the device storage, since writing and reading external storage permissions are labeled as dangerous and are runtime permissions. There exists some ways to save data on the device across multiple sessions such as Shared preferences API or one of the multiple database APIs that save the data in a local database cache.

5.2.1 Shared preferences

Shared preferences enable storage of small amounts of primitive data as key/value pairs in a file on the device. To get a handle to a preference file, and to read, write, and manage preference data, use of `SharedPreferences` class is appropriate. The Android framework manages the shared preferences file itself. The file is accessible to all the components of the app, but it is not accessible to other apps. Interface for accessing and modifying preference data returned by `Context.getSharedPreferences(String, int)`. For any particular set of preferences, there is a single instance of this class that all clients share. Modifications to the preferences must go through an Editor object to ensure the preference values remain in a consistent state and control when they are committed to storage. Objects that are returned from the various get methods must be treated as immutable by the application.

In order to use shared preferences, first a reference to the shared preference object is created under `MainActivity` as such:

```
private SharedPreferences mPreferences;
```

Then, in the `onCreate()` method, shared preferences is initialized using this code:

```
mPreferences = this.getSharedPreferences(this.getFilesDir().getName(), MODE_PRIVATE);
```

The argument `this.getFilesDir().getName()` returns the folder name of the application's installation path. The other argument `MODE_PRIVATE` prevents the shared preferences file to be read by other apps. As a result, an xml file is created inside the app folder, where information can be stored across different sessions (meaning that the app can be closed and opened again, and the information would persist).

5.2.2 Wi-Fi related findings

As an example to how shared preferences saves data, the `ACCESS_WIFI_STATE` permission is used. First, the Wi-Fi state is discovered through the following method:

```
private boolean checkWifiConnection() {
  WifiManager wifiMgr = (WifiManager) getSystemService(WIFI_SERVICE);

  if (wifiMgr.isWifiEnabled()) {
    return true;
  } else {
    return false;
  }
}
```

The boolean `checkWifiConnection()` is a straightforward method that returns a true or false value depending on device Wi-Fi connection. First, an instance of WifiManager is created. Then, `getConnectionInfo()` method is called from the Wi-Fi manager class, which returns a

representational value showing whether the Wi-Fi adapter is on or off. `getNetworkId()` method from Wi-Fi info class is called as well, in order to see if the active adapter is connected to an access point. If the adapter is on and connected to an AP, the return value is true, else, it is false.

Since the same classes are being used, a highly simple string method is also created to display all the information stored under `getConnectionInfo()` method.

```
private String wifiInfo() {
  WifiManager wifiManager = (WifiManager)
    this.getApplicationContext().getSystemService(Context.WIFI_SERVICE);
  String wifiMac = "Device Wi-Fi Mac address: " + wifiManager.getConnectionInfo().
getMacAddress();
  String connectedAp = " - Connected Wi-Fi: " + wifiManager.getConnectionInfo().toString();
  return wifiMac + connectedAp;
}
```

Together, the information gathered from two methods is written in the `onPause()` method with shared preferences as seen below.

```
protected void onPause() {
  super.onPause();

  SharedPreferences.Editor preferencesEditor = mPreferences.edit();
  if (getApplicationContext().checkCallingOrSelfPermission
      (Manifest.permission.ACCESS_WIFI_STATE)
      == PackageManager.PERMISSION_GRANTED) {
    boolean checkWifi = checkWifiConnection();
    preferencesEditor.putString("CONNECTIVITY", "Connection:" + checkWifi);
    preferencesEditor.putString("WIFI-INFO", wifiInfo());
    preferencesEditor.apply();
  }
}
```

As a result, the data is stored under /data/data/com.example.appbq_none/shared_prefs/files.xml", and without any runtime permissions. The "files.xml"file includes the following information inside:

```
<?xml version='1.0' encoding='utf-8' standalone='yes' ?>
<map>
<string name="CONNECTIVITY">Connection:true</string>
<string name="WIFI-INFO">Device Wi-Fi Mac address: 02:00:00:00:00:00 - Connected Wi-Fi: SSID:
, BSSID: 02:00:00:00:00:00, MAC: 02:00:00:00:00:00, Supplicant state: COMPLETED, RSSI: -46,
Link speed: 72Mbps, Frequency: 2437MHz, Net ID: 2, Metered hint: false, score: 60</string>

</map>
```

It is observed that information such as MAC address and SSID is not visible at all, either returning blank or a static meaningless address. This is due to a change that came with Android 6.0; programmatic access to the device's local hardware identifiers using Wi-Fi and Bluetooth APIs is removed [8]. Documentation indicates that this change was brought to provide the users an increased data protection, so access to things such as SSID are locked behind location runtime permissions, namely `ACEESS_FINE_LOCATION`, or `ACCESS_COARSE_LOCATION`; and access to local hardware identifiers were completely removed.

Continuing on the Wi-Fi theme, I noticed an interesting issue with the Android Studio. If one tries to get previously saved network with `getConfiguredNetworks()`, the IDE will flag it as an error, stating that "Missing permissions required by Wifimanager.getConfiguredNetworks", and names the missing permission as the location access. The method where the error is seen can be viewed below:

```
private String prevConnNetworks() {
  WifiManager wifiManager = (WifiManager)
  this.getApplicationContext().getSystemService(Context.WIFI_SERVICE);
  List<WifiConfiguration> configuredList = wifiManager.getConfiguredNetworks();
  return configuredList.toString();
}
```

```
<string name="CONFIGURED-NETWORKS">
[ID: 0 SSID: &quot;Inca&quot; PROVIDER-NAME: null BSSID: null FQDN: null PRIO: 0 HIDDEN: false&#10; NetworkSelectionStatus
NETWORK_SELECTION_ENABLED&#10; hasEverConnected: true&#10; numAssociation 10&#10; validatedInternetAccess&#10; KeyMgmt: NONE
Protocols: WPA RSN&#10; AuthAlgorithms: OPEN&#10; PairwiseCiphers: TKIP CCMP&#10; GroupCiphers: WEP40 WEP104 TKIP CCMP&#10;
PSK: &#10; sim_num &#10;Enterprise config:&#10;IP config:&#10;IP assignment: DHCP&#10;Proxy settings: NONE&#10; cuid=1000
cname=android.uid.system:1000 luid=1000 lname=android.uid.system:1000 lcuid=1000 userApproved=USER_UNSPECIFIED
noInternetAccessExpected=false roamingFailureBlackListTimeMilli: 1000&#10;recentFailure: Association Rejection code:
0&#10;ShareThisAp: false&#10;,

ID: 1 SSID: &quot;gofret&quot; PROVIDER-NAME: null BSSID: null FQDN: null PRIO: 0 HIDDEN: false&#10; NetworkSelectionStatus
NETWORK_SELECTION_ENABLED&#10; connect choice: &quot;Ulas&quot;WPA_PSK connect choice set time: 1627648599170 hasEverConnected:
true&#10; numAssociation 4&#10; validatedInternetAccess&#10; KeyMgmt: WPA_PSK Protocols: WPA RSN&#10; AuthAlgorithms: OPEN&#10;
PairwiseCiphers: TKIP CCMP&#10; GroupCiphers: WEP40 WEP104 TKIP CCMP&#10; PSK: *&#10; sim_num &#10;Enterprise config:&#10;IP
config:&#10;IP assignment: DHCP&#10;Proxy settings: NONE&#10; cuid=1000 cname=android.uid.system:1000 luid=1000
lname=android.uid.system:1000 lcuid=1000 userApproved=USER_UNSPECIFIED noInternetAccessExpected=false
roamingFailureBlackListTimeMilli: 1000&#10;recentFailure: Association Rejection code: 0&#10;ShareThisAp: false&#10;, *

ID: 2 SSID: &quot;Ulas&quot; PROVIDER-NAME: null BSSID: null FQDN: null PRIO: 0 HIDDEN: false&#10; NetworkSelectionStatus
NETWORK_SELECTION_ENABLED&#10; hasEverConnected: true&#10; numAssociation 10&#10; validatedInternetAccess&#10; KeyMgmt: WPA_PSK
Protocols: WPA RSN&#10; AuthAlgorithms: OPEN&#10; PairwiseCiphers: TKIP CCMP&#10; GroupCiphers: WEP40 WEP104 TKIP CCMP&#10;
PSK: *&#10; sim_num &#10;Enterprise config:&#10;IP config:&#10;IP assignment: DHCP&#10;Proxy settings: NONE&#10; cuid=1000
cname=android.uid.system:1000 luid=1000 lname=android.uid.system:1000 lcuid=1000 userApproved=USER_UNSPECIFIED
noInternetAccessExpected=false roamingFailureBlackListTimeMilli: 1000&#10;recentFailure: Association Rejection code:
0&#10;ShareThisAp: false&#10;,

ID: 3 SSID: &quot;TestNetwork&quot; PROVIDER-NAME: null BSSID: null FQDN: null PRIO: 0 HIDDEN: false&#10;
NetworkSelectionStatus NETWORK_SELECTION_ENABLED&#10; hasEverConnected: true&#10; numAssociation 2&#10;
validatedInternetAccess&#10; KeyMgmt: WPA_PSK Protocols: WPA RSN&#10; AuthAlgorithms: OPEN&#10; PairwiseCiphers: TKIP CCMP&#10;
GroupCiphers: WEP40 WEP104 TKIP CCMP&#10; PSK: *&#10; sim_num &#10;Enterprise config:&#10;IP config:&#10;IP assignment:
DHCP&#10;Proxy settings: NONE&#10; cuid=1000 cname=android.uid.system:1000 luid=1000 lname=android.uid.system:1000 lcuid=1000
userApproved=USER_UNSPECIFIED noInternetAccessExpected=false roamingFailureBlackListTimeMilli: 1000&#10;recentFailure:
Association Rejection code: 0&#10;ShareThisAp: false&#10;]
</string>
```


Figure 5.5: The configured networks entry in the shared preferences file.

Normally, an error marked with a red exclamation mark would stop the *Run selected configuration* action. However, when I ignored this error and ran the program, it indeed continued to launch the application without any problems. Writing the return value of this method with to the shared preferences file also did not result in any in-app crashes. To confirm that this is not a general error with my configuration, I conducted some tests. First, I tried different methods that required runtime permissions, such as reading contacts list, or getting user accounts. The read contacts method did not contain any errors, but it would crash the app on pause, just as the contacts list were being written into the file. The get accounts method also did not contain any errors, but the entry on the shared preferences file would be blank, exactly the same result when I tried to get connected SSID from the user. With `prevConnNetworks()`, I wrote the contents of `getConfiguredNetworks` into the shared preferences file, which include all saved Wi-Fi SSIDs, key management information, Wi-Fi protocols, pairwise ciphers, and group ciphers pertaining to previously connected networks. Unfortunately, this did not indicate which of the listed networks the device was actually connected to at the moment. The Figure 5.5 demonstrates the configured networks entry in the shared preferences file.

In terms of Wi-Fi connections, there was one remaining thing to check, which was to get a list of available Wi-Fi connections. This is done with `getScanResults()`, however, it turns out that the method returns no values unless one the location permissions are enabled. This method can be found in the program code section. It only works when a location permission is requested.

5.2.3 Bluetooth related findings

In order to get information on Bluetooth connections, several approaches can be thought out. These approaches will follow a similar pattern to the Wi-Fi information extraction attempts. I tried to get Wi-Fi information based on the details of the active connection if there was any, previously connected networks, and networks that were available for connection based on an instantaneous scan. In turn, the approaches for Bluetooth would be as follows. First, attempt to find any active connections, following that, try to find the details of previously paired Bluetooth devices, and finally, conduct a scan for any available Bluetooth devices in the area.

A `getConnBtDevice()` method was created to get details of the connected Bluetooth device. Within this method, an instance of the Bluetooth manager was created, which enabled me to use the available `getConnectedDevices()` method, which is supposed to return the set of devices that is in the *connected* state as such:

```
List<BluetoothDevice> btDevice = bluetoothManager.getConnectedDevices(GATT_SERVER);
```

The `btDevice` was used as the return value of the method. This did not result in any visible errors, however it caused the app to crash. This happened even when the location permission was requested temporarily to test this, and the result was the same. Therefore, extracting the connected device name was a failure.

Next goal was to get information on the paired Bluetooth devices. The `btDiscoverDevices()` method was created to achieve this. The method can be seen down below:

```
private String btDiscoverDevices() {
  BluetoothAdapter bluetoothAdapter = ((BluetoothManager)
```

```
  this.getApplicationContext().getSystemService(Context.BLUETOOTH_SERVICE)).getAdapter();

  bluetoothAdapter.startDiscovery();

  IntentFilter filter = new IntentFilter(BluetoothDevice.ACTION_FOUND);
  registerReceiver(btBroadcastReceiver, filter);

  return btDeviceList.toString();
}
```

The method first creates an instance of the Bluetooth adapter, then starts the remote device discovery process. The result is sent to the broadcast receiver, which is a base class for code that handles broadcast intents. There, the list string `btDeviceList` is filled with information pertaining to the found devices, and used as the return value of the `btDiscoverDevices()` method. Finally, contents of the return value is saved to the shared preferences file. Unfortunately, this returned a blank string. It was observed that when one the location permissions was requested, the list would be filled up by available Bluetooth devices, thus, this action also was locked behind the location permissions. Similar to Wi-Fi findings, these first two operations were a failure.

Finally, I tried to get a list of previously paired Bluetooth devices. It was done through a simple method called `getPairedBt()`, which can be observed below:

```
private String getPairedBt() {
  BluetoothAdapter bluetoothAdapter = ((BluetoothManager)
  this.getApplicationContext().getSystemService(Context.BLUETOOTH_SERVICE)).getAdapter();

  Set<BluetoothDevice> btDeviceList = bluetoothAdapter.getBondedDevices();

  List<String> list = new ArrayList<>();
  for(BluetoothDevice bluetoothDevice : btDeviceList) {
    list.add("--- Name: " + bluetoothDevice.getName());
    list.add("Address: " + bluetoothDevice.getAddress());
    list.add("Contents: " + bluetoothDevice.describeContents());
```

```
    list.add("Class: " + bluetoothDevice.getBluetoothClass());
    list.add("Type: " + bluetoothDevice.getType());
    list.add("UUIDs: " + bluetoothDevice.getUuids() + "---");
  }
  return list.toString();
}
```

The method simply creates an instance of the bluetooth adapter, then assigns the return value of Bluetooth adapter member method `getBondedDevices()` to a Bluetooth device set variable. This information is written into a list with a loop and finally, the list is used as the string return value. This value is then written into the shared preferences file. The entry in the file looks as such:

```
    <string name="PAIRED-BT">
    [--- Name: Cagri (main), Address: 20:EE:28:E3:44:0D, Contents: 0, Class: 7a020c, Type: 2,
UUIDs: [Landroid.os.ParcelUuid;@38a581b---,
    --- Name: Jaybird Tarah, Address: C0:28:8D:A1:92:49, Contents: 0, Class: 240404, Type: 1,
UUIDs: [Landroid.os.ParcelUuid;@8a1edb8---]</string>
```

The two devices were paired as an example to the BQ device, one of which is an iPhone 7, and the other is a Bluetooth Jaybird Tarah earphone. Interestingly, this method neither created any errors in the IDE nor during runtime. Names of the previously paired devices, and their Bluetooth MAC addresses are clearly visible.

Consequently, the Bluetooth findings coincided with the Wi-Fi findings, where only the previously configured and/or paired devices were extracted without any (runtime) permissions.

5.2.4 NFC related findings

Even though the `NFC` permission was declared, I was in possession of no NFC tags, and could not put it to test.

5.2.5 Unique identifiers

Without using any permissions, one can also get a unique identifier using Secure Android ID. While this identifier is a constant, it is not hard-coded to the device. A factory reset or a OS upgrade can reset this value, leading the device to have a different identifier. However, it is observed that this value stays the same regardless of app restart, reinstall, or device restart. Following function is used to get this ID, using the Settings provider.

```
private String getSecureId() {
  String androidId = Settings.Secure.getString(getContentResolver(), Settings.Secure.
ANDROID_ID);
  return androidId;
}
```

Return value `androidId` is written to the shared preferences file as such:

```
<string name="SECURE-ANDROID-ID">9aad5b2905182c59</string>
```

5.3 AppHTC_full

This is the full permission state application for the HTC One SV device which carries Jelly Bean 4.2.2 (API level 17) version of Android. Documentation indicates that permissions also were categorized, although somewhat differently from the API level 27. In addition, these permission groups aren't depicted to the user in the UI side of the coin, and may be done so out of a concern for developer convenience, for both Android application developers, and for the ones who contribute to the Android platform itself.

When the user taps on the .apk file for installation, the device simply states "Do you want to install this application? It will get access to:", and lists every declared permission regardless of protection level or permission group. This can be seen in Figure 5.6.

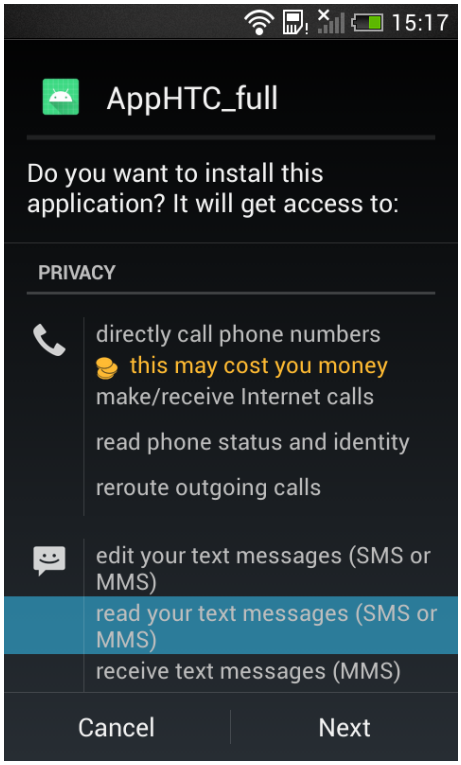

Figure 5.6: HTC device permission prompt on app install.

The full permission list for this app can be seen in the Table 4.2. In this section, the permissions related to the scope of this thesis will be examined by groups. These are the permissions that let us read or receive information related to the user.

First in the list is the MESSAGES group. Within it, `RECEIVE_SMS` and `READ_SMS` permissions are of some interest. The MMS and WAP push permissions are ignored, since almost everyone uses instant messaging applications instead. The SMSs in the device are located in the internal Android SMS table. As such, the information will be gathered from the table by rows and tables. Here is what the function `readSms()` looks like this:

```
private String readSms() {
  Cursor c = getContentResolver().query(Uri.parse("content://sms/inbox"),
  null, null, null, null);
  String message = "";
  if (c.moveToFirst()) {
    do {
```

```
      for (int i = 0; i < c.getColumnCount(); i++) {
        if (c.getColumnName(i).equals("date")) {
          Date date = new Date(c.getLong(i));
          message += " Date: " + date +" / ";
        }
        else if (c.getColumnName(i).equals("address")) {
          message += " Address: " + c.getString(i) + " / ";
        }
        else if (c.getColumnName(i).equals("body")) {
          message += " Message body: " + c.getString(i) + "\n";
        }
      }
    } while (c.moveToNext());
  } else {
    return "No messages to log!";
  }
  c.close();
  return message;
}
```

The SMS table includes two columns for the entry name and value, and various rows for the types of data the SMS message holds. For demonstration, I select the column names that equal to "date", ädress", and "body", and add it to the string variable message, which will be the return value of the function. If the table is empty, the return value indicates that there are no messages to log, if there are any, it is written into the permission log file, and the portion that is written to text file is as following:

```
   Address: +905459430635 /  Date: Wed Jul 28 16:32:55 EEST 2021 /  Message body: Another test
 message
  Address: +905459430635 /  Date: Wed Jul 28 16:32:40 EEST 2021 /  Message body: Test message
```

Two messages was sent from another device consecutively, and it can be seen that they are written into the file with the sender number, date sent, and what the message contains.

A broadcast receiver was used to pick up the incoming SMS message that is given access by the RECEIVE_SMS permission. The receiver is registered with the onCreate() method, and listens for the SMS received action. In the method, the incoming message body and sender number is extracted, and the date is noted at the time of arrival. Then, the contents are written into the permission logs file, and the broadcast receiver is unregistered with the onDestroy() method. The contents of the receiver can be seen below:

```
private static final String SMS_RECEIVED = "android.provider.Telephony.SMS_RECEIVED";
List<String> receivedMessage = new ArrayList<>();
private final BroadcastReceiver smsBr = new BroadcastReceiver() {
  @Override
  public void onReceive(Context context, Intent intent) {
    Bundle bundle = intent.getExtras();
    SmsMessage[] messages;
    String messageAddr = "";
    String body = "";
    if (bundle != null) {
      try {
        Object[] pdus = (Object[]) bundle.get("pdus");
        messages = new SmsMessage[pdus.length];
        for (int i = 0; i < messages.length; i++) {
          messages[i] = SmsMessage.createFromPdu((byte[])pdus[i]);
          messageAddr += messages[i].getDisplayOriginatingAddress();
          body += messages[i].getMessageBody();
        }
        receivedMessage.add("Message Address: " + messageAddr +
        ", Message body: " + body +
        ", Time received: " + Calendar.getInstance().getTime());
        logToFile(getApplicationContext(), receivedMessage.toString());
      } catch (Exception e) {
        Log.d("Exception caught ", e.getMessage());
      }
    }
```

```
    else {
      logToFile(getApplicationContext(), "\nBundle is null.");
    }
  }
};0
```

To test if it works, a SIM was inserted into the device and a message was sent to the same phone number. The resulting SMS was picked up and written to the file by the receiver. The resulting file entry can be seen below:

```
[Message Address: +905331928099, Message body: Message to myself: test receive sms , Time
received: Tue Aug 03 19:01:26 EEST 2021]
```

SOCIAL_INFO group includes the permissions that provide access to the user's social data, such as contacts, the call log, and the social stream. Normally, three permissions would be tested in this group, namely, READ_CONTACTS, READ_CALL_LOG, and READ_SOCIAL_STREAM. However the latter permission which deals with the social stream is not really functional anymore. It would give access to the social media application's stream of data related to friends, family, and followed people. However, since than, these social media apps changed how they work on their end, and do not necessarily share this data with the device OS. Thus, I argue that this relic of a permission does not need further effort.

So for the reading contacts permission, a similar action to the SMS method is required. That is to say, the information regarding the contacs is stored under a table, and extraction includes using cursor to find the desired columns, and storing the related information to a variable as seen below:

```
private String getContacts() {
  List<String> contactList = new ArrayList<String>();
  ContentResolver cr = getContentResolver();
  Cursor c = cr.query(ContactsContract.Contacts.CONTENT_URI, null, null, null, null);
  if ((c != null ? c.getCount() : 0) > 0) {
    while (c.moveToNext()) {
      String id = c.getString(c.getColumnIndex(ContactsContract.Contacts._ID));
```

```
      String name = c.getString(c.getColumnIndex(ContactsContract.Contacts.DISPLAY_NAME));
      contactList.add(name);
      if (c.getInt(c.getColumnIndex(ContactsContract.Contacts.HAS_PHONE_NUMBER)) > 0) {
        Cursor pCur = cr.query(ContactsContract.CommonDataKinds.Phone.CONTENT_URI, null,
        ContactsContract.CommonDataKinds.Phone.CONTACT_ID + " = ?", new String[]{id}, null);
        while (pCur.moveToNext()) {
          String phoneNo = pCur.getString(pCur.getColumnIndex(ContactsContract.CommonDataKinds
.Phone.NUMBER));
          contactList.add(phoneNo);
        }
        pCur.close();
      }
    }
  }
  if (c != null) {
    c.close();
  }
  return contactList.toString();
}
```

I added three different contacts with random telephone numbers, and as a result, the method finds this information and stores it into the permission log file.

```
[Contact A, 05329990099, Contact B, 05331234567, Contact C, 05340008521]
```

As such, the call log is also stored in a local table, and the `getCallLog()` method's contents are quite similar to the contacts and the SMS methods.

```
private String getCallLog() {
  ContentResolver cr = getContentResolver();
  Cursor c = cr.query(CallLog.Calls.CONTENT_URI, null, null, null, null);
  List<String> callList = new ArrayList<>();
  while (c.moveToNext()) {
    Date date = new Date(c.getLong(c.getColumnIndex(CallLog.Calls.DATE)));
```

```
    callList.add("\nPhone Number: " + c.getString(c.getColumnIndex(CallLog.Calls.NUMBER)) +
    " Call Type: " + c.getString(c.getColumnIndex(CallLog.Calls.TYPE)) +
    " Date: " + date +
    " Duration: " + c.getString(c.getColumnIndex(CallLog.Calls.DURATION)));
  }

  c.close();
  return callList.toString();
}
```

Some calls were made to various random numbers, and since the phone did not include a SIM card within, the calls were immediately halted. The duration is always zero as it can be seen below. Other than that, the permission log entry includes the phone number, type of call (outgoing, incoming etc.), date, and duration.

```
[Phone Number: 05329990099 Call Type: 2 Date: Tue Aug 03 17:09:41 EEST 2021 Duration: 0,
Phone Number: 05340008521 Call Type: 2 Date: Tue Aug 03 17:09:52 EEST 2021 Duration: 0,
Phone Number: 05340008521 Call Type: 2 Date: Tue Aug 03 17:10:00 EEST 2021 Duration: 0,
Phone Number: 05331234567 Call Type: 2 Date: Tue Aug 03 17:10:05 EEST 2021 Duration: 0]
```

Next group is named PERSONAL_INFO, and has two permissions, which is related to reading and writing of user profile. Just like before, a content resolver is used to query the related entry, which is `ContactsContract.Profile` this time, and a cursor is defined to navigate to the display name and id. The method `getProfile()` is as follows:

```
private String getProfile() {
  ContentResolver cr = getContentResolver();
  Cursor c = cr.query(ContactsContract.Profile.CONTENT_URI, null, null, null, null);
  List<String> profile = new ArrayList<>();
  while (c.moveToNext()) {
    profile.add("Name: " + c.getString(c.getColumnIndex(ContactsContract.Profile.DISPLAY_NAME)
) +
    " - ID: " + c.getString(c.getColumnIndex(ContactsContract.Profile._ID)));
```

```
  }
  c.close();
  return profile.toString();
}
```

The permission log entry includes the user e-mail, and an ID value, which may represent the user account.

```
[Name: Cagri HTC Erdem - ID: 9223372034707292161]
```

CALENDAR permission group includes the permissions that allow an application to read and write calendar data. The method `getCalendar()` was used in AppBQ_full app, and it will also be used here. One can view the method in Calendar permission group section, or the appendix with the full codes. It demonstrates how and what kinds of data are being read.

As it can be seen below, calendar details such as future event titles, places and times they take place is saved in the permission log file.

```
[account name: PC Sync
calendar display name:HTC Sync Manager
title: Important event 1
location: Offenburg, Offenburg, Germany
event start: Mon Aug 09 03:00:00 EEST 2021
event end: Tue Aug 10 03:00:00 EEST 2021,
account name: PC Sync
calendar display name:HTC Sync Manager
title: Unimportant event 99
location: SÃ¼leymanpasa/Tekirdag, Turkey
event start: Wed Aug 11 03:00:00 EEST 2021
event end: Thu Aug 12 03:00:00 EEST 2021]
```

Since the code is getting quite repetitive, I skipped the user dictionary group which had the `READ_USER_DICTIONARY` permission. Instead, it can be examined in the appendix, or the GitHub page which is also located in the appendix section. Following that, it is observed that bookmarks group

has also lost its functionality due to the changes on the browser end. The history bookmarks would normally be reported by the browser to the OS in the old days, however since then, Android stopped this and effectively deprecated the permission, making this change valid for all versions. Another similar example will come up with the telephony group later.

The method for the location permission group works exactly like the one for the BQ full app, so the code and the results are identical. Once can refer to the BQ full findings, or the appendix to view it.

The methods related to Wi-Fi and Blueooth permissions stayed practically same with ones for the BQ device, however, the results have differences. In this version, the device Wi-Fi and Bluetooth MAC addresses aren't locked, and return the actual value instead of a meaningless constant.

```
Device Wi-Fi Mac address: 1c:b0:94:a4:47:39 - Connected Wi-Fi: SSID: Ulas, BSSID: 00:1c:7b:f9:
c5:b4, Supplicant state: COMPLETED, RSSI: -70, Link speed: 39, Frequency: 2412, Net ID: 1,
Metered hint: false


Bluetooth MAC Address: BC:CF:CC:C1:FC:7A
```

Other than that, the information pertaining to the connected AP, discovered Wi-Fi networks and Bluetooth devices, and previously connected networks and paired devices resulted in the same outcome with the AppBQ_full.

The ACCOUNTS group included the saved accounts, and just as the same before, I could get the user Gmail with the same code in the `getAcc()` method. The difference is that the user e-mail can also be retrieved from the calendar and profile related methods in this app.

PHONE CALLS group had one interesting candidate, namely the `READ_PHONE_STATE` permission. Two methods were created to retrieve the related information. First method is `getTelephonyInfo()`. The contents are as follows:

```
private String getTelephonyInfo() {
  TelephonyManager tm = (TelephonyManager) getSystemService(Context.TELEPHONY_SERVICE);
  List<String> telInfo = new ArrayList<>();
```

```
  String phoneType = null;
  if (tm.getPhoneType() == TelephonyManager.PHONE_TYPE_GSM)
  phoneType = "Phone Type: GSM\n";
  else if (tm.getPhoneType() == TelephonyManager.PHONE_TYPE_CDMA)
  phoneType = "Phone Type: CDMA\n";
  telInfo.add(phoneType);
  telInfo.add("IMEI: " + tm.getDeviceId() + "\n");
  telInfo.add("IMSI: " + tm.getSubscriberId() + "\n");
  telInfo.add("Network Operator: " + tm.getNetworkOperatorName() + "\n");
  telInfo.add("SIM Operator: " + tm.getSimOperatorName() + " - " + tm.getSimOperator() + "\n")
;
  telInfo.add("Phone number: " + tm.getLine1Number());
  return telInfo.toString();
}
```

In this method, an instance of the telephony manager object is created to get relevant information. First, the phone type is determined and the result saved to the string list. Following that, the IMEI, IMSI, Network and SIM operator names, and phone number are written into the list which is returned by the method. The `telInfo` string list is then written into the permission log file. The results clearly show the retrieved data.

```
[Phone Type: GSM
, IMEI: 355026055113951
, IMSI: 286016296911274
, Network Operator: Turkcell
, SIM Operator: Turkcell - 28601
, Phone number: +905331928099]
```

The second method `getBuildInfo()` includes information affiliated with the current device build extracted from the system properties.

```
private String getBuildInfo() {
  List<String> buildInfo = new ArrayList<>();
```

```
  buildInfo.add("Manufacturer: " + Build.MANUFACTURER);
  buildInfo.add("Model: " + Build.MODEL);
  buildInfo.add("Serial number: " + Build.SERIAL);
  buildInfo.add("Bootloader: " + Build.BOOTLOADER);
  buildInfo.add("Display: " + Build.DISPLAY);

  return buildInfo.toString();
}
```

This method gets the manufacturer name, device model, device serial number, bootloader version number, and the firmware build ID.

```
[Manufacturer: HTC, Model: HTC One SV, Serial number: SH2CVTP01483, Bootloader: 2.21.0000,
Display: JDQ39]
```

Lastly, using the phone state permission, I tried to get the relevant cell tower details, however this was a failure. In the old versions, the documentation included the method `getNeighboringCellInfo()` to get this information, however it turns out that Android has deprecated this method after 2015, and completely removed it in Android Q. A method called `getAllCellInfo()` was added, however sub-methods such as `getCellIdentity()` was only available for applications with the minimum SDK level 30. In addition, Android added a new system-reserved permission called `READ_PRIVILEGED_PHONE_STATE`, which outright blocks every application from retrieving this information regardless of the version, except the system apps.

5.4 AppHTC_none

For this version, every permission declared shows up upon installation and in the Settings. Staying in objective scope for this app means that no permissions can be declared in any way. First order of business, is to try the AppBQ_none code in this app. After some small adjustments to make it work in the older SDK level app, and some lint suppressions to disregard the errors, it was launched. However, this immediately resulted in a crash.

Next, the lint suppressors were removed, and the erroneous code was commented out to see the remaining information. This in turn, downright eliminated some methods such as `prevConnNetworks()` (previously connected Wi-Fi networks), `getBtInfo()` (Bluetooth MAC), `btDiscoverDevices()`(Bluetooth discovery method), and finally, `getPairedBt()` (previously bonded Bluetooth devices). The remaining entries in the shared preferences file was Wi-Fi info, Wi-Fi scan, and secure Android ID. Interestingly, this also resulted in a direct crash. It turned out that even the remaining methods without any errors made the app crash, and in the end, only piece of information gathered from the device was the secure Android ID, which as I mentioned before, is not a completely sound way of retrieving an identifier, since it can be changed to another value after a factory reset, or version update.

5.5 Findings on the rooted devices

The root access was given to the BQ device through Magisk Manager, and to the HTC device through the SuperSU app. Both applications are similar in what they do, which gives superuser access to the user, and enables applications to run *su* commands in their code, provided that the user agrees to give them this access. The root access was done on the stock ROM with the same Android versions, so virtually nothing was changed on the devices except the access to superuser privileges for device user.

Rooting the device does not change anything for what the applications return in the log files, since this privilege is given to the user and not the app. The user still needs to grant the superuser request by prompted the app. Following this, I set up two more apps to test these shell commands through Java. The point here is to get information relevant to the scope of this thesis without any permissions.

For the BQ device, a `runCommand()` method was implemented. The method takes a shell command as argument, and writes it into standard input of a su process.

```
public void runCommand(String...commands) {
  try {
    Process process = Runtime.getRuntime().exec("su");
    DataOutputStream dataOutputStream = new DataOutputStream(process.getOutputStream());
    for(String s : commands) {
      dataOutputStream.writeBytes(s + "\n");
      dataOutputStream.flush();
    }
    dataOutputStream.writeBytes("exit\n");
    dataOutputStream.flush();
    try {
      process.waitFor();
    } catch(Exception e) {
      logToFile(this, e.getMessage());
    }
    logToFile(this, "recording saved.");
    dataOutputStream.close();
  } catch (Exception e) {
    logToFile(this, e.getMessage());
  }
}
```

When this method is called with a shell command, the application receives a prompt from Magisk app, givin the option to grant or deny the superuser request as seen in Figure 5.7. A Google

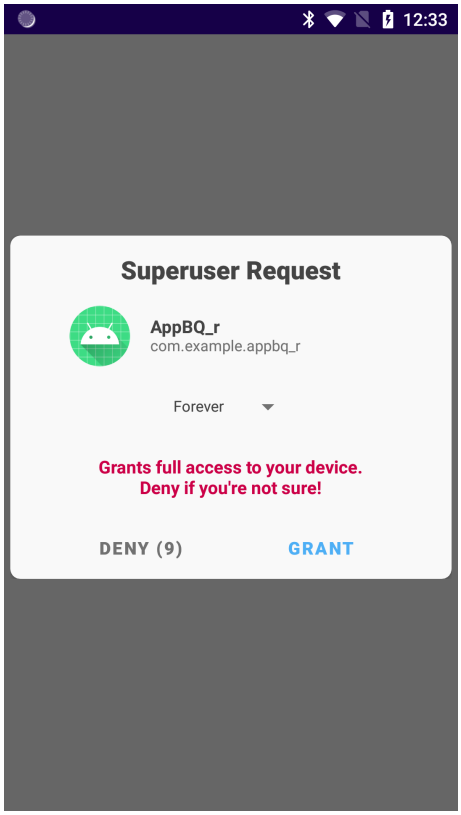

Figure 5.7: Magisk prompt to regarding superuser access on the BQ device.

Developers page that lists the Android Debug Bridge (adb) shell commands is available [9]. After granting the superuser request, the app functions as normal. I wrote some commands from the adb shell command list in order to try the `runCommands()` method as seen below:

```
runCommand("screenrecord --time-limit 5 "
+ getApplicationContext().getFilesDir() + "/sRec.mp4\n");
runCommand("screencap "
+ getApplicationContext().getFilesDir() + "/screen.png");
```

First command is supposed to record the screen for 5 seconds, and the second to take a screenshot. After these actions are completed, a process completion note to the log file is written for each process. The screen.png and sRec.mp4 is supposedly saved to the files folder in the application directory. Device File Explorer in Android Studio confirms this fact. From the file sizes, one can

tell that the recording and screenshotting actions are actually being completed. If I change the time limit for the recorder, the sRec.mp4 file increases in size. However, when I try to save the files, copy them, or simply open them, Device File Explorer gives the following error:

```
Error opening contents of device file "screen.png": cp: /data/local/tmp/temp7eac41bc-4f0a-4d9d
-a6e3-686823b7b656: Permission denied
```

Interestingly, on Windows File Explorer, these files were invisible, as in they were not set as invisible, they were simply not there. Android's file explorer showed screen.png to be 39.4 KB, and the sRec.mp4 to be 745.4 KB, however when I looked at the same location from the Windows' explorer with the address `AppData\Local\Google\AndroidStudio2020.3\device-explorer\bq-aquaris_x_pro-GF006209\data\data\com.example.appbq_r\files`, the total size of the folder was 56 bytes, the exact same value as the plogs.txt file located in the same place. The `data/local/tmp` folder was also nowhere to be seen in Windows file explorer, and showed up as empty in Android device file explorer. To mitigate this problem, I wrote `chmod 777` superuser command to change the permissions in the folder.

```
try {
  Runtime.getRuntime().exec("chmod -R 777 " + getApplicationContext().getFilesDir());
  Runtime.getRuntime().exec("chmod -R 777 " + "/data/local/tmp");
} catch (Exception e) {
  logToFile(this, e.getMessage());
}
```

This actions results in a change to permissions somewhat, which can be observed in Figure 5.8. It results in a permission change for all the folders I specified, but screen.png and sRec.mp4 is unchanged. This persisted even when I made chmod specific to those files. As a last resort, I tried to request `READ_EXTERNAL_STORAGE` permission, but alas, it did not make any difference. In the end, I can say that it is possible to get information from the user with a rooted device, however accessing to those files are another matter.

Unfortunately, the results were the same for the HTC device. Following the same actions did not give a different outcome, and Android device file explorer kept giving the same error for the file.

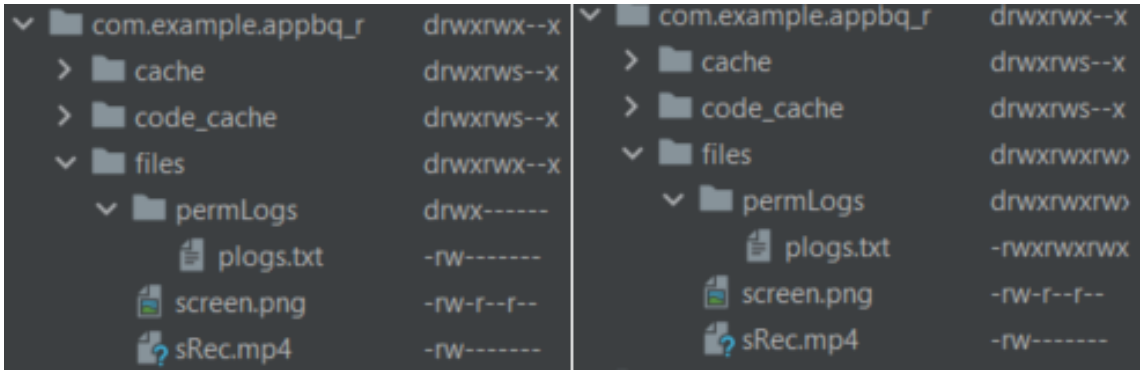

Figure 5.8: The effect of chmod 777 to folder permissions in Android Device File Explorer.

6 Discussion

In this chapter, findings from the experimentation are discussed. First section will delve into the overall evolution of the permissions. Next section will draw a comparison on the results.

6.1 Evolution of permissions

First, a broad observation of the changes is made, then the specific instances of some differences are explored. One main distinction between the version 4.2.2 and 8.1.0 is the permission groups themselves. To begin with, mentioning groups is unheard of in the papers related to permissions in Android before 2015. This, as pointed out before, is caused by the fact that they don't matter at all. I speculate that the reason permission groups are there is for categorical reasons, providing developers a convenience. Overall, the groups were fairly numerous, since permissions such as calendar, user dictionary, bookmarks, alarm, voicemail got their own groups. In total, permissions declared in the manifest file for 4.2.2 had 25 permission groups, and for 8.1.0, this number was only nine. The normal, or install-time permissions were also flagged under these groups for 4.4.2. The groups for 8.1.0 did not include any install-time permissions, since the permission group feature was strictly adapted for requesting runtime permissions.

While the number of permissions increased greatly with new Android releases, it is observed that this change was done on permissions with protection levels system and signature. The total number of permissions was around 220 for version 4.4.2, and 400 for 8.1.0. In this thesis, the examined protection levels for permissions were normal and dangerous, and no discernable amount could

be observed between the versions tested. Naturally some permissions were added and some were deprecated, or even removed, but the number stayed relatively same, with 70 permissions declared for 4.4.2 (24 normal and 46 dangerous) , and 69 permissions declared for 8.1.0 (42 normal, and 27 dangerous). One realizes that the main issue here arises from the change on the balance of power between normal and dangerous permissions. The change is almost mirrored across versions where, as time passed, Android reversed the protection levels of some permissions.

The change includes some permissions being removed. Namely, such as profile, social stream, user dictionary, history bookmarks, subscribed feeds etc. Android notes that these permissions are removed, but still kept around for backwards compatibility. Thus, their protection levels are switched to normal, regardless of their functionality, and flagged as removed. Depending on what they do, their functionality is either non-functional in the newer versions, or combined with other permissions. The outcome of this was tested with the `READ_PROFILE` permission in the AppBQ_full app. It turned out that `ContactsContract.Profile` was tied to the `GET_ACCOUNTS`. When the `getProfile()` method from the HTC app was used in the BQ app, it worked as expected. Upon removing the `GET_ACCOUNTS` permission, the app crashed. The app also stopped responding when `READ_PROFILE` permission was declared. This suggests that the removed permission had no functionality for an app built for version 8.1.0. In this case, the `AccountManager` and `ContactsContract.Profile` classes were tied to the same permission, whereas before they were associated with their respective permissions. The implication here is clear, as Zhauniarovic et al. argued, Android permissions moved into a more coarse-grained arrangement over time, when instead, scholars such as Fang et al. were asking for a more fine-grained structure even before version 6.0.

A certain number of permissions were demoted, as in, their protection levels were changed from dangerous to normal, making them install-time permissions. Notable instances include `INTERNET`, `BLUETOOTH`, `BLUETOOTH_ADMIN`, and `NFC`. The implications of this will be discussed in the next section.

6.2 Comparison of results

7 Conclusion

This is my conclusion.

Bibliography

[1] M. Alenezi, I. Almomani. “Abusing Android permissions: A security perspective”. In: *2017 IEEE Jordan Conference on Applied Electrical Engineering and Computing Technologies (AEECT)*. IEEE. 2017, pp. 1–6 (cit. on p. 15).

[2] E. Alepis, C. Patsakis. “Hey doc, is this normal?: exploring android permissions in the post marshmallow era”. In: *International Conference on Security, Privacy, and Applied Cryptography Engineering*. Springer. 2017, pp. 53–73 (cit. on pp. 9, 23, 24).

[3] D. Barrera, H. G. Kayacik, P. C. Van Oorschot, A. Somayaji. “A methodology for empirical analysis of permission-based security models and its application to android”. In: *Proceedings of the 17th ACM conference on Computer and communications security*. 2010, pp. 73–84 (cit. on p. 11).

[4] Z. Fang, W. Han, Y. Li. “Permission based Android security: Issues and countermeasures”. In: *computers & security* 43 (2014), pp. 205–218 (cit. on p. 17).

[5] A. P. Felt, E. Ha, S. Egelman, A. Haney, E. Chin, D. Wagner. “Android permissions: User attention, comprehension, and behavior”. In: *Proceedings of the eighth symposium on usable privacy and security*. 2012, pp. 1–14 (cit. on p. 8).

[6] D. Ferreira, V. Kostakos, A. R. Beresford, J. Lindqvist, A. K. Dey. “Securacy: an empirical investigation of Android applications’ network usage, privacy and security”. In: *Proceedings of the 8th ACM Conference on Security & Privacy in Wireless and Mobile Networks*. 2015, pp. 1–11 (cit. on p. 16).

[7] C. in Flow. *Runtime permission request*. https://codinginflow.com/tutorials/android/runtime-permission-request. Accessed: 2021-07-01 (cit. on p. 28).

[8] Google. *Android 6.0 Changes*. https://developer.android.com/about/versions/marshmallow/android-6.0-changes.html. Accessed: 2021-07-20 (cit. on p. 50).

[9] Google. *Android Debug Bridge*. https://developer.android.com/studio/command-line/adb. Accessed: 2021-08-15 (cit. on p. 68).

[10] Google. *Define a Custom App Permission*. https://developer.android.com/guide/topics/permissions/defining. Accessed: 2021-06-01. 2020 (cit. on p. 14).

[11] Google. *Permissions on Android*. https://developer.android.com/guide/topics/permissions/overview. Accessed: 2021-05-30. 2021 (cit. on pp. 11, 14).

[12] K. Huang, J. Zhang, W. Tan, Z. Feng. "An empirical analysis of contemporary android mobile vulnerability market". In: *2015 IEEE International Conference on Mobile Services*. IEEE. 2015, pp. 182–189 (cit. on pp. 8, 18, 19).

[13] S. M. Kywe, Y. Li, K. Petal, M. Grace. "Attacking android smartphone systems without permissions". In: *2016 14th Annual Conference on Privacy, Security and Trust (PST)*. IEEE. 2016, pp. 147–156 (cit. on pp. 9, 20).

[14] S. Liebergeld, M. Lange. "Android security, pitfalls and lessons learned". In: *Information Sciences and Systems 2013*. Springer, 2013, pp. 409–417 (cit. on p. 10).

[15] S. Liu. *Android operating system share worldwide by OS version from 2013 to 2020**. https://www.statista.com/statistics/271774/share-of-android-platforms-on-mobile-devices-with-android-os/. Accessed: 2021-06-05. 2020 (cit. on pp. 21, 22).

[16] A. O. S. Project. *Android Code Search, AppManifest.xml for android4.2.2_r1.2*. https://cs.android.com/android/platform/superproject/+/android-4.2.2_r1.2:frameworks/base/core/res/AndroidManifest.xml;bpv=0. Accessed: 2021-07-05 (cit. on p. 30).

[17] T. Stöber, M. Frank, J. Schmitt, I. Martinovic. "Who do you sync you are? smartphone fingerprinting via application behaviour". In: *Proceedings of the sixth ACM conference on Security and privacy in wireless and mobile networks*. 2013, pp. 7–12 (cit. on p. 16).

[18] V. Taylor, J. R. Nurse, D. Hodges. “Android apps and privacy risks: what attackers can learn by sniffing mobile device traffic”. In: (2014) (cit. on pp. 8, 16).

[19] L. S. Vailshery. *Share of Apple iPhones by iOS version worldwide from 2016 to 2020*. https://www.statista.com/statistics/565270/apple-devices-ios-version-share-worldwide/. Accessed: 2021-06-05. 2021 (cit. on p. 21).

[20] J. M. Vidal, M. A. S. Monge, L. J. G. Villalba. “A novel pattern recognition system for detecting Android malware by analyzing suspicious boot sequences”. In: *Knowledge-Based Systems* 150 (2018), pp. 198–217 (cit. on p. 11).

[21] Y. Wang, J. Zheng, C. Sun, S. Mukkamala. “Quantitative security risk assessment of android permissions and applications”. In: *IFIP Annual Conference on Data and Applications Security and Privacy*. Springer. 2013, pp. 226–241 (cit. on p. 15).

[22] J. Xiao, S. Chen, Q. He, Z. Feng, X. Xue. “An Android application risk evaluation framework based on minimum permission set identification”. In: *Journal of Systems and Software* 163 (2020), p. 110533 (cit. on p. 8).

[23] Y. Zhauniarovich, O. Gadyatskaya. “Small changes, big changes: an updated view on the android permission system”. In: *International Symposium on Research in Attacks, Intrusions, and Defenses*. Springer. 2016, pp. 346–367 (cit. on p. 22).

A Appendix

Full program codes, manifest and output files can also be found in the following GitHub page.