Logo

Description automatically generated

Vulnerability Testing Evoleon App

Chameleon Security

Adrian Thaus – Student ID: 222275741

S222275741@deakin.edu.au

Table of contents

[Introduction 2](#_Toc166521559)

[Environment Setups 3](#_Toc166521560)

[Simulated Environment 3](#_Toc166521561)

[Real Environment 4](#_Toc166521562)

[Decompiling App 5](#_Toc166521563)

[Rationale 5](#_Toc166521564)

[Process 5](#_Toc166521565)

[Vulnerability Tests 6](#_Toc166521566)

[Upskilling 6](#_Toc166521567)

[Examining Manifest and Code 6](#_Toc166521568)

[Root Check 7](#_Toc166521569)

# A logo for an animal adoption company Description automatically generatedIntroduction

This report aims to investigate and test common vulnerabilities in the Chameleon Eveleon EV app.

Due to this unexplored app environment in terms of the security team, it prominently outlines the processes and methodologies used to get to different stages of the analysis. In doing so, I intend this report to pose as a starting point for those further investigating the app. Following this approach, I have included my brainstorming of test ideas throughout the report which were beyond my initial investigation or later proved to be too time consuming for my remaining time but could pose as individual tests for others in the future.

# Environment Setups

## Simulated Environment

Following the Evoleon team’s recommendation, I followed the steps in the ‘Getting Started’ document of the Evoleon Github ([here](https://github.com/Chameleon-company/Evoleon/blob/a88ac60bce23eb7ea00951d8885dd72f0df1dfa9/_Documentation/Getting%20Started.md)).

Put simply, this involves cloning the Evoleon app Github repository and installing Expo’s commandline tool, Expo CLI. Now on the mobile device I needed to install the Expo Go app, which was quite flexible in terms of OS, SDK versions, and emulator options. Using all of these together I could start a server on desktop, creating a QR code for the mobile device to scan and view the app. This also requires an Expo account to connect the mobile device to the server.

I experienced some difficulties in logging in during this process through the commandline when prompted, since it would timeout quite quickly. Logging in on with the mobile device prior to trying to join the server automatically confirmed the device and fixed the problems I was experiencing.

|  |
| --- |
| A screenshot of a computer  Description automatically generated A logo for a pet adoption company  Description automatically generated A screenshot of a phone  Description automatically generated |

Running the through the simulated environment in Expo Go presented some pros and cons. I found that without the actual app installed, my investigation was quite limited. Although comparing it to the next example, this environment was more stable and so more features functioned correctly.

## Real Environment

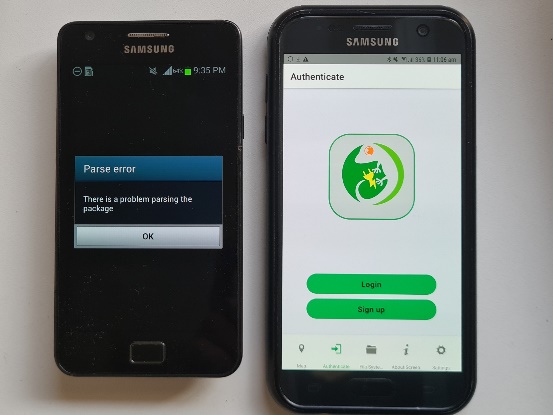
Compiling the Github repository into an apk file for android was something I was most interested in. Doing some research I found this could be done with the tools already installed from setting up the simulated environment. I mostly followed [this guide](https://dev.to/chinmaymhatre/how-to-generate-apk-using-react-native-expo-kae).

Using the ‘build’ command in Expo CLI, a build of the project could be downloaded from the dashboard of my Expo account. Initially my attempts to build the apk kept failing in the Expo project. With some research my error in ‘build.gradle’ was solved by the ‘eject’ command in Expo CLI.

A screenshot of a computer

Description automatically generated

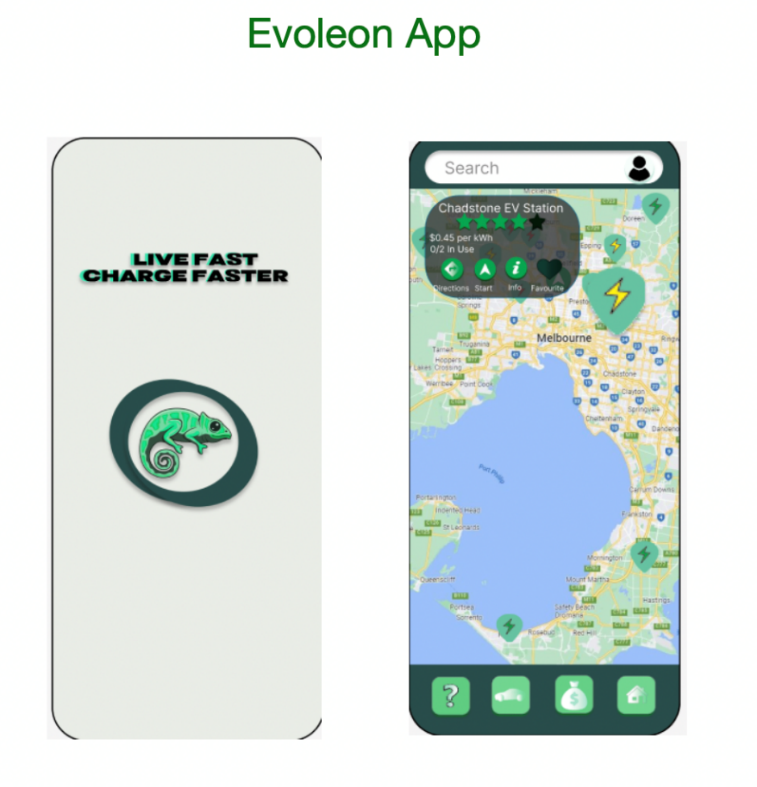
Once finshed, I could download the apk and use it to install the actual app on a variety of devices.



Using this apk or other methods, the use of emulators could be another environment type to further explore and test the app.

# Decompiling App

## Rationale

After conducting this process I had realised it could be entirely unnecessary since the clone of the Github repository is fundamentally the decompiled app. Although comparing the two folders presented some advantages of undertaking this step.

|  |  |
| --- | --- |
| **Github repository folder** | **Decompiled folder** |

Decompiling the app had stripped the folder of files I was uninterested in with my investigation, namely notes, IOS files, and unused assets. Additionally, what I found the most valuable was that the android files were now arranged in a more familiar way.

## Process

A screenshot of a computer program

Description automatically generatedIn order to decompile the apk file I addressed [this guide](https://book.hacktricks.xyz/mobile-pentesting/android-app-pentesting/smali-changes). Following the ‘Fast Way’, I installed the APKLab extension for Visual Studio Code and followed steps from there. Although I believe there are more benefits to using [APKTool](https://apktool.org/) for both decompiling the file and further analysis.

APKTool provides opportunities for reverse engineering through access to edit the file, promeniently smali code, and allows for proper recompilation of the apk with signing and a generated key. This reverse engineering ability could be used for my unexplored test ideas in the following topic.

# Vulnerability Tests

## Upskilling

In my research to find elements of the app to test, I had read articles such as [this one](https://www.hackthebox.com/blog/intro-to-mobile-pentesting) and put together a list of simple security measures which could be checked and, if necessary, easily resolved.

Unfortunately due to the Evoleon app being an untested area of the security team at this time, the majority of my time was already invested at this point of the task. So the bold ideas in this list are those I tested and will elaborate further in the following sections, whilst the others can pose as individual tests for those wanting to investigate further in this area.

* Screenshot protections
* Authentication and cryptography of login page
* **Root detection**
* SSL/TLS protocols
* Code obfuscation to prevent reverse engineering
* **Manifest restrictions**
* **Hard coded information**
* Runtime code verification to prevent code tampering

## Examining Manifest and Code

A screen shot of a computer code

Description automatically generatedDuring my upskilling I particularly noted common vulnerabilities in the code, such as those in [this article](https://book.hacktricks.xyz/mobile-pentesting/android-app-pentesting). Focusing on ‘AndroidManifest.xml’ and variable-based files such as ‘strings.xml’, I explored to check simple weaknesses and mistakes in security.

Elements I had checked include:

* **‘debuggable =\= true’**

Disabling possible vulnerabilities made present by debug functions

* **‘allowBackup=false’**

Preventing copies being made for reverse engineering or data stealing

* **No custom network security configurations**

These custom configurations may contain exceptions placed for development or debugging, specifically not intended for the commercial versions

* **No SDK restriction**

Potential in reverse engineering to roll back compatibility issues and introduce vulnerabilities present in older SDK platforms

* **Hard coded information**

API keys, custom schemas, or other developer notes may be left in commercial versions and in plain text

## Root Check

Transfering and using the apk to install the app on a variety of devices, I could experiement with whether or not the app checked if the android device had root access and would place restrictions accordingly. In this test I used a standard Samsung Galaxy S20 as a control, and a rooted S2 and S7. . Although a rooted device is not necessarily dangerous, it may present security risks for both genuine and malicious users of the app.

Stock S20: A two cell phones on a table

Description automatically generated Rooted S2 & S7: A black cell phone with a screen on the screen

Description automatically generated with medium confidence

Although the rooted S2 failed to install the app showing a parse error, most likely due to the incompatibility on an old device, a potential investigation in the future could be to alter the code in an attempt to lower the SDK restriction to uncover vulnerabilities as I found the SDK compatibility was not explicitly given in the manifest. 

Whereas the stock S20 and rooted S7 functioned identically, ignoring cosmetic elements from differing themes and screen ratios. Implementing root check to the app can be done through several ways, file path analysis, system command execution, or superuser app checks.

File path analysis aims to check the status of the device by attempting to access or modify system files. This is most simply done by returning files in the system directory, such as "/system/app” and/or "/system/bin/su", and checking whether or not the app is capable of accessing and reading the file.

System command execution is similar to file path analysis, where instead of checking for file access privileges, the app checks for system command privileges. Using a try-catch exception, a system command, such as “su”, can be run during runtime to check the privileges of the device by the outcome of the command.

Superuser app checks detect root access by searching for the presence of apps requiring superuser privileges, and therefore very likely to be a rooted device. Most effectively this method searches for the package name of common apps on rooted devices, such as “Magisk”, “SuperSU”, and/or “BusyBox”.

In a similar fashion, it may be of interest to further root detection with emulation detection. By restricting the use of the app in an emulator, only intended environments will allow the app to function to avoid experimentation and reverse engineering which should only be capable during testing, development, and debugging. This can be implemented by verifying the build configuration of the device, such as “Build.FINGERPRINT”, “Build.PRODUCT”, and “Build.BRAND”, for standard or emulator specific values, such as “generic”, “sdk”, or “emulator”.