PROJECT REPORT

VULNERABILITY ASSESSMENT & PENETRATION TESTING

ANDROID PENTESTING WITH INJUREDANDROID

By

Name of the student:
Anant Kaul

Enrolment/Registration No.
1800201C202

Prepared in the partial fulfillment of the
Vulnerability Assessment & Penetration Testing (VAPT)
Internal Component (End-Term)

BML MUNJAL UNIVERSITY

Date of Submission: 28th November, 2021

Submitted To: Dr. Rajesh Yadav
[Asst. Prof.]

Introduction

This project is presented as a complete walkthrough/guide for the **InjuredAndroid** Application (Official Latest PlayStore version as of November, 2021). It is prepared in order to showcase and perform the **Android Pentesting** by using a **Catch The Flag (CTF)** approach and scenario of the intentionally vulnerable android application (apk), named as described. By performing this, many vulnerabilities came across and all these can be classified with the help of OWASP Mobile Security Top 10 for categorizing and substituting a type to each vulnerability respectively. For a clear and better understanding, visit my respective GitHub Repository.

Requirements

- ✓ Java 9+
- ✓ Python3
- ✓ Android Studio

Tools Used

- ✓ SDK Platform Tools (ADB)
- ✓ Apktool
- ✓ Jadx or Jadx-Gui (exe)
- ✓ Objection
- ✓ MobSF
- ✓ Ghidra
- ✓ Cloud Enum
- ✓ Firebase Enum
- ✓ DB Browser for SQLite
- ✓ BurpSuite

Main Work

Starting with a CTF based scenario means starting with zero knowledge about the application and then digging deep into it. It can be directly considered as a black box testing as the pentester is only provided with the application and he/she has to catch the vulnerabilities by performing various techniques, then let it be static or dynamic, the testing should be performed equally with equal concentration. Technically, in the static analysis, we perform the source code review and try to catch the vulnerabilities related to hard-coded strings or verifying the user with very week complexity or authentication. The only way to catch this kind of vulnerability is to go with the static analysis and devote proper time and concentration to it. On the other hand, in dynamic analysis, we run the application and try to capture the vulnerabilities based on the actions performed, activities executed, permissions granted and so on. So, analyzing and reviewing the application in the real-time becomes one of the most important part of the testing as a whole.

Now, before beginning the testing, we're clear with the types of testing methodologies. As performing the real pentesting, we start with downloading the apk either from the official PlayStore or any third party. But for making it ethically right, we download it from the official PlayStore so that we can report the bug as well for that particular version of the application. After having the apk installed in our *Android Debugger (ADB)*, we try to get its source code for starting with the static analysis. In order to get it, we use the **apktool**, which has the ability to reverse engineer and decode/decompile the source code from the given packed apk. It also provides the *smali code*. In short, smali is where the application's source code is stored. After having all the source code and the apk unpacked, we can start with our static analysis.

To be able to go with the flow, we need another tool, which can perform the same but, providing a much easier way of doing it. For that we have the **Jadx tool**. It is the tool that has the ability to produce Java source code from Android Dex and Apk files. Having that in out android pentest toolbox, gives a way easier approach of doing the static analysis as it provides an editor type interface without having the trouble of decompiling and then again compiling the apk. Then, we start performing our static analysis and start with solving the flags as showcased in detail with respective steps in the GitHub repository.

After completing some flags, we come to the next step, where we are required to go with the dynamic analysis and perform some activities in order to have the flags solved. So, for that, we have the basic adb shell, where we can communicate with the android debugger and can access it at the root level in its filesystem. Additionally, we need to know that android works on the SE-Linux which is again a linux environment, so the commands and the basic environment remain the same. Moreover, many applications are able to scan that we're accessing the apk at the root level and hence, the application doesn't work. So, for overcoming the situation, we have the well-known tool **Objection**, which has the ability to bypass it and perform the SSL Pinning. After having it done, we're able to bypass the SSL Pinning and have the root access of the working application and therefore continue with our testing (dynamic analysis).

In many flags, we do get the hints, while performing the source code review, that the application uses the firebase database as well as the AWS S3 storage for its back-end data. With that information, we proceed by capturing the vulnerable parameters and enumerate the same with the help of two great tools, named **FirebaseEnum** and **Cloud-Enum** respectively, as already described and given above. Furthermore, we come across various scenarios, where we see that the application uses the SQLite database to store its local data and for enumerating the same, we have another well-known tool **DB Browser for SQLite**. These tools have a great ability to enumerate different databases according to their respective use cases.

Also, sometimes, we come across a scenario, where we need to dig deep into the apk and have a static code review of its assembly code and see if any of the parameter is vulnerable there or not. With this in mind, we start solving the 15th flag, where we needed the help of **Ghidra** tool to have the assembly code decoded on our screen and have a really great debugger interface as a whole and hence, were able to capture the flag successfully.

Besides the above process, we can also go with another well-known automated tool **MobSF**, which performs the static as well as the dynamic analysis of the application and showcases the result of the findings at ease. But we know that automated scans can never replace the real pentesting as in the automated scan we come across many false positives which can make the pentester go in the wrong direction. On the contrary, MobSF provides a very nice interface for the dynamic analysis as it provides its own interface for the adb, that we use in our host machine. It has the ability to use the same debugger and provides a great playground in order to perform the pentest at ease with everything in the same workflow, in a very well designed and well maintained environment. Here, we can also directly use the Frida payload script and have the logs on the go. This helped us in cracking the 6th flag in the InjuredAndroid application. Also, sometimes we can use the RCE (remote code exploits) from the same to get our flags solved, as we did in flag 13.

At last, when we are done with all the static and dynamic analysis, we finally come to the last flag, where we have to intercept the traffic and capture the flag. For that, first we need to bypass the SSL Pinning and then finally we intercept it using the widely used industry standard tool, commonly known as **BurpSuite**. Now, when we are done finding all the flags, we can browse to the last of the application main activity and can see that we have cracked all the flags, which turn out to be green from red. Hence, we are complete with our Android Pentesting and are on the stage of generating the report, then let it be using the MobSF automated static and dynamic report or the manually created efficient pentesting report. Therefore finishing the whole pentetration testing of the mobile (android) application, we report all the findings to the authorized organization.