

Security Lab – Finding and Exploiting Vulnerabilities in Android Apps

VMware

In this lab, you'll work with two images:

- The **Hacking-Lab LiveCD VM image** and an **Android VM image**

1 Introduction

In this lab, you will search and exploit vulnerabilities in Android apps. For this, you are going to use two apps that contain various vulnerabilities and that have been developed for training purposes. The goal of this lab is that you get familiar with finding and exploiting some typical vulnerabilities that are often found in mobile apps, that you get more sensitive with respect to such vulnerabilities in mobile apps, and that you get familiar with some of the basic tools to analyze mobile apps. The lab consists of two parts:

- In the first part, you are working with the first of the two apps to get familiar with tools and vulnerabilities. There's a lot of step-by-step guidance in this part.
- In the second part, you are working with the second app. There, you must find and exploit the vulnerabilities on your own.

2 Getting ready

Hacking-Lab VM:

Prepare your VM as follows:

- Start the Hacking-Lab VM in **NAT** networking mode and run the following commands in a console to make sure that the required tools are installed:

```
apt-get update  
apt-get install dex2jar adb jd-gui apktool
```
- Get the file Lab_FindAndroidAppVuln.zip from Moodle and unzip it to a directory of your choice. We reference this directory with <working-dir>.
- Switch the VM to **Host-Only** networking mode and reboot it

Android VM

- Start the Android VM in **Host-Only** networking mode (Hacking-Lab VM and Android VM should be in the same subnet)
- When booted, press the SPACE key to continue beyond the lock-screen
- Enter the pin: *1111* and click the checkmark.
- Press the left mouse button and swipe upward to see the installed apps. Open the Terminal Emulator app and enter `ifconfig`.
- Note down the IP address of the Android VM (interface wlan0).

Now switch to the Hacking-Lab VM and use a terminal to connect to the Android VM with:

```
adb connect <Android IP Address>
```

adb stands for Android Debug Bridge, which is a very versatile command line tool that is part of the Android SDK and that allows to install apps and to interact with Android from a remote system.

Important: If your Android VM should crash / not react to user input anymore or lose its IP address (check with `ifconfig`), you must restart it and reconnect to it using the statement above.

Kommentiert [BT1]: If you use the Hacking-Lab VM provided by the tutor, you don't need to do anything. You will find the files mentioned in this lab under `/home/hacker/Lab/android`. This is the <working-dir>.

If you use the bare-bone Hacking-Lab VM downloaded from the Hacking-Lab web page, you can get it ready as follows:

3 Part 1: Finding and Exploiting Vulnerabilities in DIVA

In the first part, you are working with DIVA, which stands for Damn Insecure and Vulnerable App¹. The Android Application Package (apk) file (diva-beta.apk) can be found in `<working-dir>/DIVA`.

3.1 Decompiling the apk File

Android apps are written in Java, compiled to Java bytecode and then compiled to Dalvik (DEX) bytecode, which is included in the apk file. Unless specific obfuscation measures have been taken by the developer, this can be reversed to the original Java source code.

To do this, perform the following steps on the Hacking-Lab VM. Convert the Dalvik bytecode to Java bytecode using dex2jar:

```
d2j-dex2jar diva-beta.apk
```

This creates a file `diva-beta-dex2jar.jar`. Decompile and view this jar file using JD-GUI. To do this, open JD-GUI with:

```
jd-gui
```

Open the jar file in JD-GUI.

As a result, you get access to the Java source code. The interesting classes are in package `jakhar.aseem.diva` and you'll analyze them in detail later.

3.2 Reverse Engineering the apk file

apk files use a binary format. To analyze its content, it must be decoded. For this, apktool is used. Decode the apk file as follows:

```
apktool d diva-beta.apk
```

This creates a directory `diva-beta`, which contains the decoded package content. For us, especially the file `AndroidManifest.xml` will be interesting, which is the central configuration file for Android apps.

3.3 Starting the App

The DIVA app is already installed on the Android VM. Start it by clicking on the DIVA app. As you can see, DIVA contains different lessons, and we will look at several of them in the following sections.

3.4 Lesson: Insecure Logging

Android uses a system-wide log facility *logcat*. All apps and the system can write this log. Written log entries remain in the log until they are eventually overwritten (the log is implemented as a ring buffer). The log can be accessed from the Hacking-Lab VM with adb as follows:

```
adb logcat
```

Start the lesson in the app, enter a credit card number and check the log. As you can see, the credit card number is written to the log. This unnecessarily exposes this number and increases the risk that an attacker can access the credit card number, or any other sensitive data you write to the log. For instance, if a colleague borrows your unlocked phone for 5 minutes «to make a phone call», they can easily access the log by connecting the phone to the USB port and accessing it with adb as described above.

Next, use JD-GUI to identify the Activity class that corresponds to this lesson (in an Android app, every screen basically corresponds to an Activity class and they are all located in package `jakhar.aseem.diva`). In general, this can easily be done for all lessons by comparing the name of a lesson with the names of the Activity classes. In the case of the current lesson, the class is `LogActivity`. Then, analyze the source code to identify the offending code. Which line is responsible for logging?

¹ <http://www.payatu.com/damn-insecure-and-vulnerable-app/>

As you can see, the app logs the credit card number when an error occurs. While this is reasonable during development and testing, make sure that the productive app does only do the necessary logging (or none) and especially make sure no sensitive data is logged.

3.5 Lesson: Hardcoding Issues – Part 1

With web applications as they were developed years ago, it was clear to put most logic on the server. With apps (and modern web applications), this has changed, and more code is on the client. As a result, the risk that security-critical activities are done on the client increases. Often, this can be exploited or circumvented because the attacker has complete control over the client device and therefore also over the app.

A typical case of this are secrets that are hardcoded into the app. In this lesson, you should have no problem to find the hardcoded vendor key within the app code (again make sure to use the right Activity class). What's its value and where is it «hidden»? Enter it in the app to verify you have found the correct secret.

Do you have a better idea how such a secret could be hidden in the code? Please write down what you would do to make it harder or impossible to discover the secret in the code. If you want to read an interesting blogpost about this topic, the following might be a good option:

<https://rammic.github.io/2015/07/28/hiding-secrets-in-android-apps/>

3.6 Lesson: Insecure Data Storage – Part 1

Enter a username and a password and save them. This data is stored somewhere in the file system, but where? In this case, it's stored in the shared preferences. That's a standard location for Android apps.

You can access this directory via adb. The following command gives you shell access on the device:

```
adb shell
```

Change to the shared preferences directory of the DIVA app with

```
cd /data/data/jakhar.aseem.diva/shared_prefs
```

Note that this app (just like any other app) stores its local data below `/data/data/jakhar.aseem.diva`, where the last directory corresponds to the fully qualified name of the app. Try to list the content of this directory with

```
ls -l
```

Why does the command fail? To answer this, you should first identify the shell-user (with the command `whoami`). Then, change the current user to `root` (with the command `su`) and check the access permissions of the directory `/data/data/jakhar.aseem.diva/shared_prefs`. Finally, compare these access permissions with the shell-user you identified. This should tell you why the command `ls -l` failed. Also, having a look at the lecture slides where the sandboxing model of Android is described should be helpful.

Note that changing to user `root` (as you did above with `su`) is possible as we are using an Android VM. On a real device, this is not possible unless the device is rooted.

Now that you have changed the user to `root`, listing the content of directory `/data/data/jakhar.aseem.diva/shared_prefs` should of course work. Verify this by executing `ls -l` again. Just one file should be listed: `jakhar.aseem.diva_preferences.xml`. View the content of the file with

```
more jakhar.aseem.diva_preferences.xml
```

Obviously, username and password are stored in this file, so they are exposed in the file system. However, other apps cannot access this data due to the sandboxing of Android apps, which means that storing data in this way – even usernames and passwords – is acceptable to a certain degree, but it's important that you are aware of the potential risks. For instance, if an attacker gets access to your rooted Android device or if they manage to find an exploitable vulnerability that gives them root access to the device, they can of course access this information as well. In addition, the shared preferences are included in backups, which exposes this data further.

Also, check the source code with JD-GUI (by using the right Activity class) to identify the code section where the data is stored in the shared preferences and log the relevant parts in the text box below.

Kommentiert [BT2]: If the app has not yet been started and the credentials have not yet been stored with the insecure storage lesson, this directory is not there yet.

3.7 Lesson: Insecure Data Storage – Part 2

This is very similar to the lesson above, but this time, another storage location is used. First, store again username and password. Then, analyze the source code in the corresponding Activity class to find out what happens in the background and document your findings.

Apparently, the data is stored in a database. Databases of the app are stored below `/data/data/jakhar.aseem.diva/databases/`, with the same access restrictions as the shared preferences – which means they cannot be accessed by other apps. These databases are SQLite databases and the command line tool to access them, `sqlite3`, is available on Android devices. To interact with the database that is used here (to do this, use the root shell access you have established via adb before), change to the `databases` directory and access the database with

```
sqlite3 ids2
```

Then, use a `SELECT` statement to check whether the entered username and password are indeed present. Write down the SQL statement you entered to check for the presence of this data.

Now delete the content from the table using an SQL `DELETE` command:

```
DELETE FROM myuser WHERE user = "value"
```

Check with a `SELECT` command whether the table indeed is empty. Then, leave `sqlite3` with

```
.exit
```

Next, read the documentation regarding secure deletion with SQLite:

https://www.sqlite.org/pragma.html#pragma_secure_delete

Then inspect the raw content of the database using `hexdump` (that's a standard Linux tool which is also present on Android):

```
hexdump -C ids2
```

Are there any traces of the username and password left in the file? What about the filesize before and after the deletion? Explain what this tells you about SQLite as used/configured in the Android VM and about SQLite data files in general.

This lesson teaches us two things: First, these databases are accessible in exactly the same way as the shared preferences, which means they cannot be accessed by other apps but by an attacker with root rights. In addition, they are also included in backups. Therefore, storing credentials in such a database provides the same security as when storing them in the shared preferences. Furthermore, you should be careful regarding SQLite and its behavior with DELETES. Depending on SQLite's configuration, you shouldn't use such a database for temporary storage of sensitive data because deleting the data from the database does not necessarily remove it from the database file right away. It's might only be «unlinked» and remains readable until it is overwritten with new database content (this applies to other database-technologies too). Until this happens, this data is also included in backups.

3.8 Lesson: Insecure Data Storage – Part 4

Another local storage issue. Analyze the source code of the corresponding Activity class to learn where the data is stored this time. Use online resources to understand what this storage location is and where it can be found in the file system. Verify your findings by inspecting the generated file in the file system.

Summarize your findings in the box below. Your summary must include the code section that is used to store the data and also the location where the data can be found on the device. What can you say about the API used to get the storage location? In addition, provide an answer why this storage location is «much worse» than the locations that were used before to store sensitive data.

3.9 Summary – Insecure Data Storage

It was mentioned above that storing sensitive data in the shared preferences is not such a bad idea as it cannot be accessed by other apps on the device. But it's still exposed in backups and can be accessed with root rights. Try to identify a way how this could be improved, so that even a user with root rights cannot (directly) access the data and that the data is not exposed in backups. Can you also identify limitations with your approach(es)? Your solution(s) should work without Internet access.

3.10 Lesson: Input Validation Issues – Part 1

You should be well familiar with SQL Injection issues on the server side. With databases used locally by the app, SQL injection also becomes an issue on the client. Find and exploit the vulnerability to achieve the goal described in the lesson. Looking at the source code of the corresponding Activity class (*SQLInjectionActivity*) may help you to achieve this. Note in the following text box why SQL injection is possible in this case and what string you injected to exploit the vulnerability.

One can argue that that's no big deal as the user could also access all data in the database directly via the file system. Therefore, how do you assess the criticality of local SQL injection in an app compared to server-side SQL injection? Can you imagine situations where local SQL injection is critical?

3.11 Lesson: Input Validation Issues – Part 2

In this scenario, the intention of the developer was to allow the user to enter a Web-URL to get the web page and display it in the WebView below the VIEW button. Unfortunately, the developer assumed that the user would enter an http-URL and therefore did not include any URL-scheme filtering, which can easily be seen by inspecting the source code in the Activity class. This means the (malicious) user can enter whatever they like.

Try to exploit the vulnerability to access the underlying file system. Accessing the file system is possible with the *file:///* URL scheme (yes, that's three slashes) followed by the fully qualified file path. Of course, the user is restricted to the files of the app or to files that are readable system wide. But you should certainly be able to information that was stored in the shared preferences and below */sdcard* during previous lessons. Verify that this can be done and write down the URLs that you used.

So, if you lend your device to a colleague, he or she can easily exploit this to access your credentials, without requiring adb or other low-level access.

3.12 Lesson: Access Control Issues – Part 1

Activities are a core concept of an Android app. Basically, every screen of an app is an Activity. When an Activity opens another screen, the corresponding Activity is “pushed” onto the current Activity and so on. Clicking the back button always removes the Activity on the top and returns to the one underneath it.

Based on this knowledge, it’s time to have look at *AndroidManifest.xml*. This file is in the folder *diva-beta* that was created when you decoded the apk file in section 3.2. It’s illustrated below:

```
<?xml version="1.0" encoding="utf-8" standalone="no"?>
<manifest xmlns:android="http://schemas.android.com/apk/res/android" package="jakhar.aseem.diva"
    platformBuildVersionCode="23" platformBuildVersionName="6.0-2166767">
    <uses-permission android:name="android.permission.WRITE_EXTERNAL_STORAGE"/>
    <uses-permission android:name="android.permission.READ_EXTERNAL_STORAGE"/>
    <uses-permission android:name="android.permission.INTERNET"/>
    <application android:allowBackup="true" android:debuggable="true" android:icon="@mipmap/ic_launcher" android:
        label="@string/app_name" android:supportRtl="true" android:theme="@style/AppTheme">
        <activity android:label="@string/app_name" android:name="jakhar.aseem.diva.MainActivity" android:theme=
            "@style/AppTheme.NoActionBar">
            <intent-filter>
                <action android:name="android.intent.action.MAIN"/>
                <category android:name="android.intent.category.LAUNCHER"/>
            </intent-filter>
        </activity>
        <activity android:label="@string/d1" android:name="jakhar.aseem.diva.LogActivity"/>
        <activity android:label="@string/d2" android:name="jakhar.aseem.diva.HardcodeActivity"/>
        <activity android:label="@string/d3" android:name="jakhar.aseem.diva.InsecureDataStorage1Activity"/>
        <activity android:label="@string/d4" android:name="jakhar.aseem.diva.InsecureDataStorage2Activity"/>
        <activity android:label="@string/d5" android:name="jakhar.aseem.diva.InsecureDataStorage3Activity"/>
        <activity android:label="@string/d6" android:name="jakhar.aseem.diva.InsecureDataStorage4Activity"/>
        <activity android:label="@string/d7" android:name="jakhar.aseem.diva.SQLInjectionActivity"/>
        <activity android:label="@string/d8" android:name="jakhar.aseem.diva.InputValidation2URISchemeActivity"/>
        <activity android:label="@string/d9" android:name="jakhar.aseem.diva.AccessControl1Activity"/>
        <activity android:label="@string/apic_label" android:name="jakhar.aseem.diva.APICredsActivity">
            <intent-filter>
                <action android:name="jakhar.aseem.diva.action.VIEW_CREDS"/>
                <category android:name="android.intent.category.DEFAULT"/>
            </intent-filter>
        </activity>
        <activity android:label="@string/d10" android:name="jakhar.aseem.diva.AccessControl2Activity"/>
        <activity android:label="@string/apic2_label" android:name="jakhar.aseem.diva.APICreds2Activity">
            <intent-filter>
                <action android:name="jakhar.aseem.diva.action.VIEW_CREDS2"/>
                <category android:name="android.intent.category.DEFAULT"/>
            </intent-filter>
        </activity>
        <provider android:authorities="jakhar.aseem.diva.provider.notesprovider" android:enabled="true" android:
            exported="true" android:name="jakhar.aseem.diva.NotesProvider"/>
        <activity android:label="@string/d11" android:name="jakhar.aseem.diva.AccessControl3Activity"/>
        <activity android:label="@string/d12" android:name="jakhar.aseem.diva.Hardcode2Activity"/>
        <activity android:label="@string/pnotes" android:name="jakhar.aseem.diva.AccessControl3NotesActivity"/>
        <activity android:label="@string/d13" android:name="jakhar.aseem.diva.InputValidation3Activity"/>
    </application>
</manifest>
```

As a side note, before looking at the Activities: The name of the app is also included in this file, right at the top: *package="jakhar.aseem.diva"*.

Every Activity that is used by the app must be listed in this file. In addition, the main Activity (which is called when the app is started) must contain an intent-filter with action and category as described above. Otherwise, the app cannot be started.

Other Activities that are marked with *android:exported=true* or that have an *intent-filter* can also be called «from the outside». This is typically used when an app provides some functionality to other apps (via inter-app communication). If such inter-app communication should not be allowed, then Activities should usually not be made available «to the outside» (except the main Activity).

With these basics, we can look at the lesson. Clicking the *VIEW API CREDENTIALS* button shows the credentials. Looking at the source code of the Activity class of the lesson (*AccessControl1Activity*), we can see the following:


```
public void viewAPICredentials(View paramView)
{
    Intent localIntent = new Intent();
    localIntent.setAction("jakhar.aseem.diva.action.VIEW_CREDS");
    if (localIntent.resolveActivity(getPackageManager()) != null)
    {
        startActivity(localIntent);
        return;
    }
}
```

This shows us that to display the API credentials, the code calls an Activity that provides the action *jakhar.aseem.diva.action.VIEW_CREDS*. Looking at *AndroidManifest.xml*, we can see that this is provided by the Activity *jakhar.aseem.diva.APICredsActivity*.

As this Activity contains an intent-filter, it can be called from anywhere (i.e. by every other app), not just within the app. Doing this is easy, again using adb (this time, root access is not needed):

```
adb shell
```

To call any Activity that can be called from the outside, use *am start* together with the desired activity, in our case

```
am start -n jakhar.aseem.diva/.APICredsActivity.
```

To check that this even works if the app is not currently running, you should first terminate the app and then execute the command.

Note that instead of the Activity, you can also specify the action:

```
am start -a jakhar.aseem.diva.action.VIEW_CREDS.
```

Note also that the «attack» in this lesson does not really provide any benefit, as the app also shows the API credentials via a click on a button. It is therefore primarily meant to make you familiar with calling Activities from outside the app.

3.13 Lesson: Access Control Issues – Part 2

This is an extended version of the previous lesson. The idea here is that the user can only access the Tweeter API credentials after having registered online and having confirmed this within the app by entering a PIN received during online registration. Note that the lesson does not enforce this, it simply simulates this as follows: If the radio button *Register Now* is checked, the app asks the user to enter the PIN. Otherwise, the API credentials are displayed. In reality, the app would of course only show the credentials if a correct PIN was entered previously.

Looking at the code of class *AccessControl2Issues*, we see the following:

```
public void viewAPICredentials(View paramView)
{
    paramView = (RadioButton)findViewById(2131492973);
    Intent localIntent = new Intent();
    boolean bool = paramView.isChecked();
    localIntent.setAction("jakhar.aseem.diva.action.VIEW_CREDS2");
    localIntent.putExtra(getString(2131099686), bool);
    if (localIntent.resolveActivity(getPackageManager()) != null)
    {
        startActivity(localIntent);
        return;
    }
}
```

Just like above, an Activity is called. This time using the action *jakhar.aseem.diva.action.VIEW_CREDS2*. Looking at *AndroidManifest.xml*, we can see that this corresponds to Activity *jakhar.aseem.diva.APICreds2Activity*, which has an intent-filter (so it can be called «from the outside»).

Before the Activity is called, an Extra (method *putExtra*) is set, which uses the value of the boolean variable *bool*. An Extra is basically a parameter that is passed from the calling to the called Activity. Here, this value corresponds to the value received by *paramView.isChecked()* and *paramView* corresponds to the radio button *Register Now* that is displayed in the lesson. This means that *paramView.isChecked()* returns whether the radio button is checked (true) or not (false).

Looking at the code of the called Activity (*APICreds2Activity*), we can see that if this value is false (method *getBooleanExtra*), then the credentials are displayed.

```
protected void onCreate(Bundle paramBundle)
{
    super.onCreate(paramBundle);
    setContentView(2130968606);
    TextView localTextView = (TextView)findViewById(2131492983);
    EditText localEditText = (EditText)findViewById(2131492984);
    Button localButton = (Button)findViewById(2131492985);
    if (!getIntent().getBooleanExtra(getString(2131099686), true))
    {
        localTextView.setText("TVEETER API Key: secrettveeterapikey\nAPI User name: diva2\nAPI Password: p@ssword2");
        return;
    }
}
```

What remains is finding out the name of the Extra. We can see in the code that the name of the Extra is referred as *getString(2131099686)*. Within the code, this is mapped to the actual name of the Extra in two steps.

First, we must get the corresponding name of this numeric identifier. To do so, we must look at the class *R* and search for the name of the numeric identifier:

```
public static final int character_counter_pattern = 2131099685;
public static final int chk_pin = 2131099686;
public static final int d1 = 2131099687;
```

Here, we see that the name of this identifier is *chk_pin*. To get the actual string, we must look into *res/values/strings.xml* in the decoded apk file (which you got when using dex2jar in section 3.2), which contains the strings that are used in the app:

```
<string name="character_counter_pattern">%1$d/%2$d</string>
<string name="chk_pin">check_pin</string>
<string name="d1">1. Insecure Logging</string>
<string name="d10">10. Access Control Issues - Part 2</string>
```

This shows us that the actual name of the extra is *check_pin*.

Now we have everything we need to call the Activity. Terminate the app and do the following in a terminal:

```
adb shell
```

To call the Activity with the Extra, use the following:

```
am start -n jakhar.aseem.diva/.APICreds2Activity --ez check_pin false
```

This should open the Activity with the credentials. Note that instead of the Activity, you can also specify the action:

```
am start -a jakhar.aseem.diva.action.VIEW_CREDS2 --ez check_pin false
```

In contrast to the previous lesson, this is a real attack, as it allows the user of the app to access the credentials without having registered online.

To conclude, this and the previous lesson should clearly have shown you that you should only make available Activities to the outside if this is really needed. Intent-filters provide no protection even if Extras are used, as an attacker can usually find out the name of the Extras and suitable values.

3.14 Lesson: Hardcoding Issues – Part 2

This lesson deals once more with a secret that is somewhere hardcoded in the code. If you look at the code of the Activity (*Hardcode2Activity*), you can see that method *access* of the class *DivaJni* is used to check the correctness of the entered vendor key.

Looking at the class *DivaJni* shows that a native library called *divajni* is loaded, which provides the method *access*. Libraries are included in the apk files and are usually located within the *lib* directory of the app. To verify this, do the following in a terminal:

```
adb shell
cd /data/app/jakhar.aseem.diva-xxxxxxx/lib/x86_64
```

Note that you must replace the *xxxx* part in the above path with the correct value.

```
ls -l
```

This should show one file, the native library *libdivajni.so*.

This is binary file that cannot easily be decompiled. Your task is to find the vendor key in this file. Write down the key and how you found it. Of course, you should verify in the app whether the vendor key works.

Hint: Check out the command line tool *strings* (man strings).

Note that when doing reverse engineering of binaries, you should always start with the *strings* command line tool to quickly extract possibly interesting information.

4 Task 2: Finding and Exploiting Vulnerabilities in the DIMBA App

Your second task is finding and exploiting vulnerabilities in an insecure Android app on your own. The app we are using is from the Damn Insecure Mobile Banking App (DIMBA) project which has been developed at ZHAW with contributions from several Bachelor and Master students. The DIMBA project is split into two parts. The first part is the DIMBA app which provides the following features:

- Several activities that are typical for a banking app, such as login screen, registering for the service, account balance view, doing payments, loading a payment slip from a file, investment area, live view of stock market, and sending and receiving messages.
- Modern look and feel thanks to Material Design.
- Plenty of vulnerabilities to discover, currently there are 33 vulnerabilities to find.

The second part is the DIMBA server with which the DIMBA app communicates. For the training app to be realistic, such a component must since it is usually the backend where most of the business logic is implemented. The DIMBA server provides the following features:

- REST API, Authentication via JSON Web Tokens (JWT)
- An integrated database for persisting transactions/activities during training sessions
- SMS message with confirmation code that must be entered by the app user to authorize a payment. Note that the SMS message is simulated and written to the server output.

4.1 DIMBA App Configuration

- Open the DIMBA app.
- Open the meta settings via the menu at the top left.
- Change the IP address of the server to match the IP address of the machine where the server runs.
- Make sure the certificate check security level is set to **1** so that there are no certificate checks performed. This makes MitM or the use of an interceptor proxy easy. Save the settings.

Note: To do app-testing with apps that use properly secured connections only, we recommend to use Frida, a dynamic code instrumentation toolkit, to intercept their traffic. See e.g., <https://infosecwriteups.com/hail-frida-the-universal-ssl-pinning-bypass-for-android-e9e1d733d29>

4.2 Intercepting Network Communication

To understand and exploit some vulnerabilities, access to the communication between app and server is required. This works best using an interceptor proxy. To do this, perform the following steps:

- On the host where the DIMBA server is running, use an interceptor proxy such as Burp Suite, OWASP ZAP or some other tool.
- Make sure the interceptor proxy listens for incoming connections on the external IP address of your host and check the port that it uses (often something like 8008 or 8080).
- Configure the Android VM to use the interceptor proxy:
 - Open the Settings app, select Network & internet, then WiFi and then VirtWiFi.
 - Next, select the pen icon at the top right and expand Advanced options.
 - Specify Proxy => Manual, enter the IP address and port used by your interceptor proxy in the Proxy host name and port field. Finally, make sure to save the settings.

4.3 Starting the DIMBA Server

To run the DIMBA server, change to *DIMBA/DIMBA_Server* and execute the following command:

```
java -jar DIMBA_server.jar
```

4.4 Resetting App and the Server

If you need to reset the app or the server (get rid of all app or server state):

Reset the app: Tap on Reset App in the Meta Settings.

Reset the server: Stop the server, delete all files in folder db. Start the server.

4.5 Test Your Setup

In the DIMBA app's login-screen, click on "**Create User**" and create the user:

Username: **h@cker**
Password: **dimba**

You should see the requests in the interception proxy and in the server log (stdout). Make sure that the interception proxy forwards the requests to the server automatically, i.e., does not run in interception mode where you have to trigger the forwarding of each request manually.

4.6 Challenges

As already stated, the DIMBA app and its communication with the server suffer from over 33 different vulnerabilities. Some of these vulnerabilities, for example the (in)security of the communication with the server, are only present at certain security levels of the app. The security level can be changed in the Meta-Settings screen of the app.

The server part of the app might also contain vulnerabilities, but these shall not be exploited.

In the following, you find a list of six challenges with different difficulty levels. Most of these challenges require multiple steps to be solved. For example, extraction and analysis of the source code, identification of an attack vector and exploitation of the attack vector.

To get the lab points for this part, solve at least *four of the six challenges*. Use the text boxes at the end of the document or a separate text file for your solutions. Your documentation must contain:

- the steps performed to find the solution [hacking-journal]
- how the vulnerability could be prevented/mitigated

Attacker Model: Assume that you have access to the APK file of the DIMBA app and that the server is a black box (no code or binary) when trying to find and exploit vulnerabilities.

1. Investments VIP Code (medium)

To do investments, a VIP code is required that can be purchased from the bank. However, due to a vulnerability, it is possible to find out the correct VIP code without having to purchase it.

Goal: Find out the valid VIP access code and get access to the Investments functionality by entering the code.

2. Little Bobby Tables (easy)

The Messages functionality allows to send and receive messages to/from the bank. The messages are cached locally on the device so that they can be viewed even if there's no network connection. As the app can be used by different users on the same device, messages of multiple users may be cached. Therefore, for confidentiality reasons, a user should only get access to his own messages. Unfortunately, this was not implemented correctly.

Goal: Find a way to access the brief exchange of messages between Alice and Bob by (mis)using the functions provided by the app itself (no adb, no root access). If you need a hint, google for the title of this challenge.

3. Copy Machine (easy)

Goal: Find a way to copy the loginPreferences.xml file in the shared_prefs folder of the app to the SD card by (mis)using the functions provided by the app itself (no adb, no root access). Verify with adb shell that the file has indeed been copied.

Hint: The file is located in `/data/data/ch.zhaw.securitylab.DIMBA/shared_prefs/loginPreferences.xml`. If absolute paths do not work, you might consider using relative paths (e.g., `../<ABSOLUTE PATH>`, if your current location is `/example/`).

4. On Screen, Ensign!

[Easy]

In Android, when displaying the currently running apps, screenshots are displayed that are taken when an app leaves the foreground. These screenshots are stored somewhere in the system, which potentially exposes them to attackers. As the DIMBA app sometimes shows sensitive information, it was implemented so that no screenshot is taken when it leaves the foreground. As a result of this, no details are shown when viewing the currently running apps. However, in some places of the DIMBA app, this was forgotten.

Goal: Find two screens with possibly sensitive information where screenshots are taken when the app leaves the foreground in the sense that the screenshots are shown when displaying the currently running apps.

5. On My Command!

[Easy]

In the Meta-Settings, there is a vulnerability that allows you to execute arbitrary commands.

Goal: Locate the vulnerability and find a way to make the content of `loginPreferences.xml` appear in the Android log (accessible using `adb logcat`).

6. On Millionaires in the Middle

[EASY/MEDIUM]

When doing a payment, one must confirm the payment using a one-time confirmation code sent by SMS. Note that the SMS content is simulated and written to the server output.

Goal: Create an additional user account `target@mail.com` with password `target1234`. This is the account of the victim. Assume the victim makes payments from time to time. Find a way to make payments on behalf of the victim `target@mail.com` without hacking into their account or requiring the victim to do anything for those payments to go through (e.g., enter confirmation codes). The icing on the cake would be to let the victim in the dark about what's happening to their money.

For MEDIUM difficulty level, assume that you must execute payments in the name of the victim with your phone and the DIMBA app only. You were able to steal the login data from the victim, but you could not change the mobile phone number to which the confirmation codes (SMS) are sent.

Tip: Have a look at the title of this task.

5 Lab Points

For **2 lab points**, you must provide reasonable answers to all questions in task 1.

For another **2 lab points**, you must provide the solution to four challenges from Task 2.

Show the filled-in sheet to the instructor. The instructor may ask you questions when inspecting your sheet.

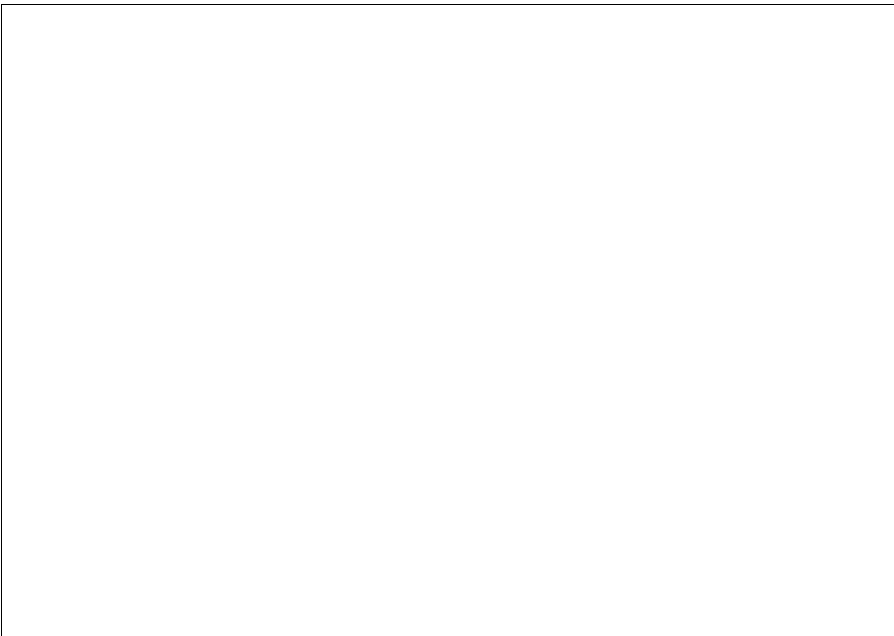
Solution for challenge _____

Solution for challenge _____

Solution for challenge _____

A large, empty rectangular box with a thin black border, intended for the user to write the solution for the challenge.

Solution for challenge _____

A large, empty rectangular box with a thin black border, intended for the user to write the solution for the challenge.