Universe Weird C132 : Android Application Based CTF Challenge Walkthrough

Saurabh Jain
Sep 28 · 5 min read

Universe Weird C132 is a beginner level Android application CTF challenge. It makes you realise that every application is a treasure hunt, more of a treasure and less of a hunt.

The aim of this CTF challenge is to concentrate on the basic flaws which are found while performing security assessment of a mobile application. We will be observing the basic mis-configurations which can lead to huge data loss.

Let's take a minute to thank Moksh for creating this challenge. If someone wants to try and solve the challenge, the link for the CTF can be found [here] and the application can be downloaded from [here].

Tools Used :

> ***adb*** *: command line tool that lets you communicate with device*

> ***apktool*** *: command line tool for reverse engineering android applications*
>
> ***jadx-gui*** *: tool for producing Java source code from Android Dex and APK files*
>
> ***Android Studio* :** *official Integrated Development Environment (IDE) for Android app development*
>
> ***Device*** *: Android Device/Android Studio Emulator/Genymotion Emulator*

Connecting the device with a USB cable and entering command for checking proper connectivity.

> *adb devices*

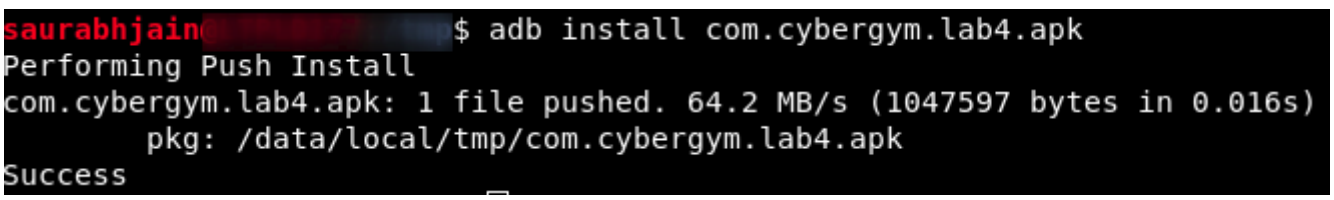
The above command will list down all the connected devices/emulators.

The above exhibit shows the list of devices connected to the system

> **Note** *:* Make sure to connect the android phone with debugging mode enabled for initiating the application installation process.

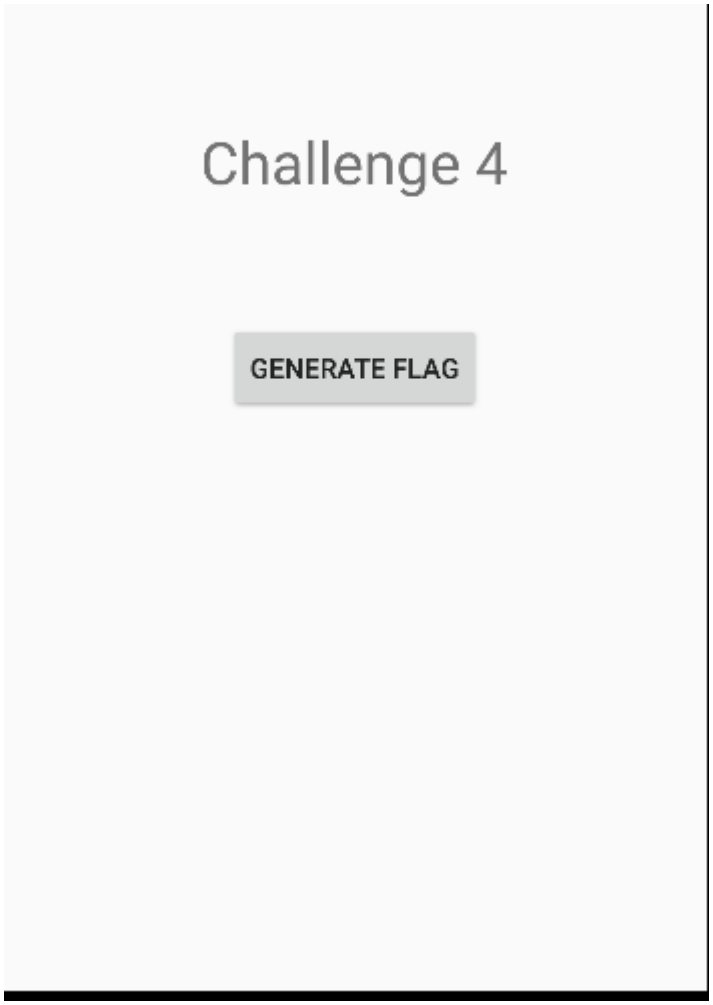
After downloading the application from the above given link, the application can be installed in device/emulator by a very simple command.

> *adb install <apk-name>*



The above exhibit shows that the application has been successfully installed in the device

We can run the application in the device/emulator



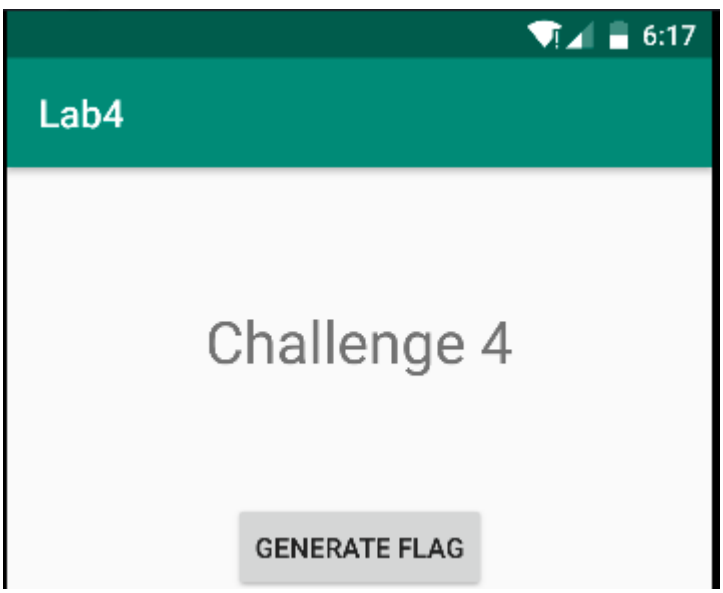
The above exhibit shows that first page while running the application

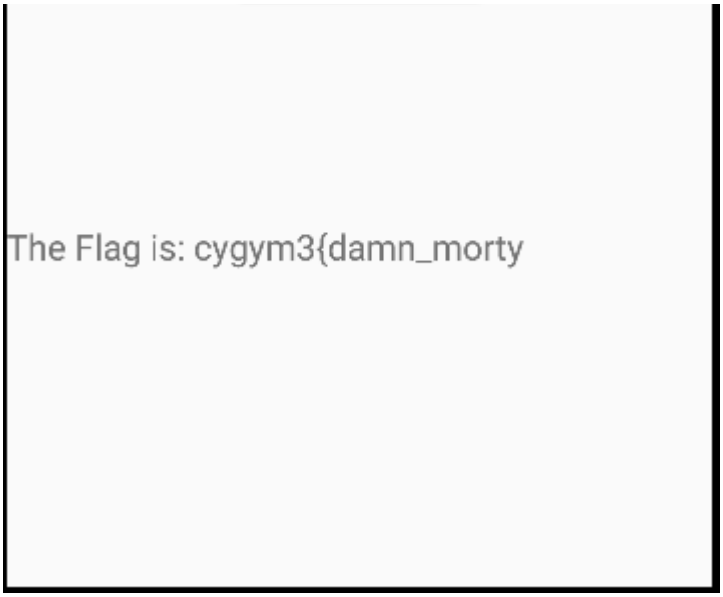
Here, button "**GENERATE FLAG**" is observed.

On a single click we got nothing.. Let's tap up on it again and again..

After clicking on the button ***three*** times we got a message…

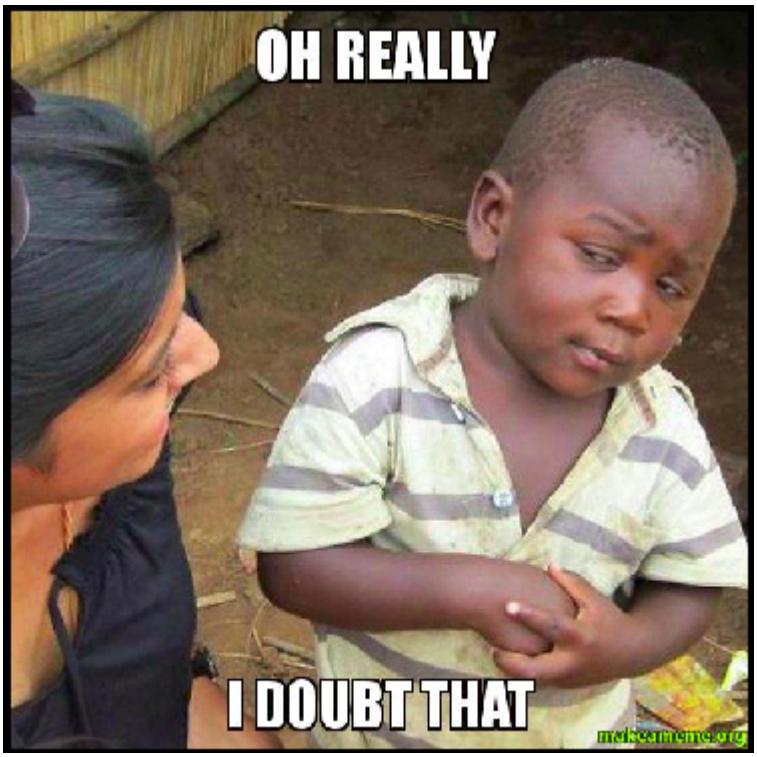
> "The Flag is : cygym3{damn_morty"





The above exhibit shows the message after tapping three times of the **GENERATE FLAG** button.

> *Is this it ? Are we done..*
>
> *Is it that easy ? Just tapping three times.*



Let's see the application source code using **jadx-gui**

> *jadx-gui <apk-name>*

We can start analysing the application source code from *MainActivity.java*

```
10 public class MainActivity extends AppCompatActivity {
11   Button a;
12
13   /* renamed from: a  reason: collision with other field name */
14   TextView f1003a;
15
16   /* renamed from: a  reason: collision with other field name */
17   String f1004a;
18   int b = 0;
19
20   /* renamed from: b  reason: collision with other field name */
21   String f1005b;
22
23   public void onCreate(Bundle bundle) {
24     super.onCreate(bundle);
25     setContentView((int) R.layout.activity_main);
26     this.f1003a = (TextView) findViewById(R.id.flagValue);
27     this.a = (Button) findViewById(R.id.button);
28     int a2 = jb.a(this);
29     this.f1005b = "The Flag is: ";
30     if (a2 <= 0) {
31       Toast.makeText(this, "Application Tampered", 1).show();
32       finishAffinity();
33     }
34     this.f1004a = "cygym3{damn_morty";
35     this.a.setOnClickListener(new View.OnClickListener() {
36       public final void onClick(View view) {
37         MainActivity.this.b++;
38         if (MainActivity.this.b == 10) {
39           Toast.makeText(MainActivity.this, "Oh, the flag looks incomplete. Fishy!", 0).show();
40           MainActivity.this.b = 0;
41         } else if (MainActivity.this.b == 3) {
42           MainActivity.this.f1003a.setText(MainActivity.this.f1005b + MainActivity.this.f1004a);
```

The above exhibit shows that the string is hard coded in application source code

So the string *"The Flag is: cygym3{damn_morty"* is hard coded.

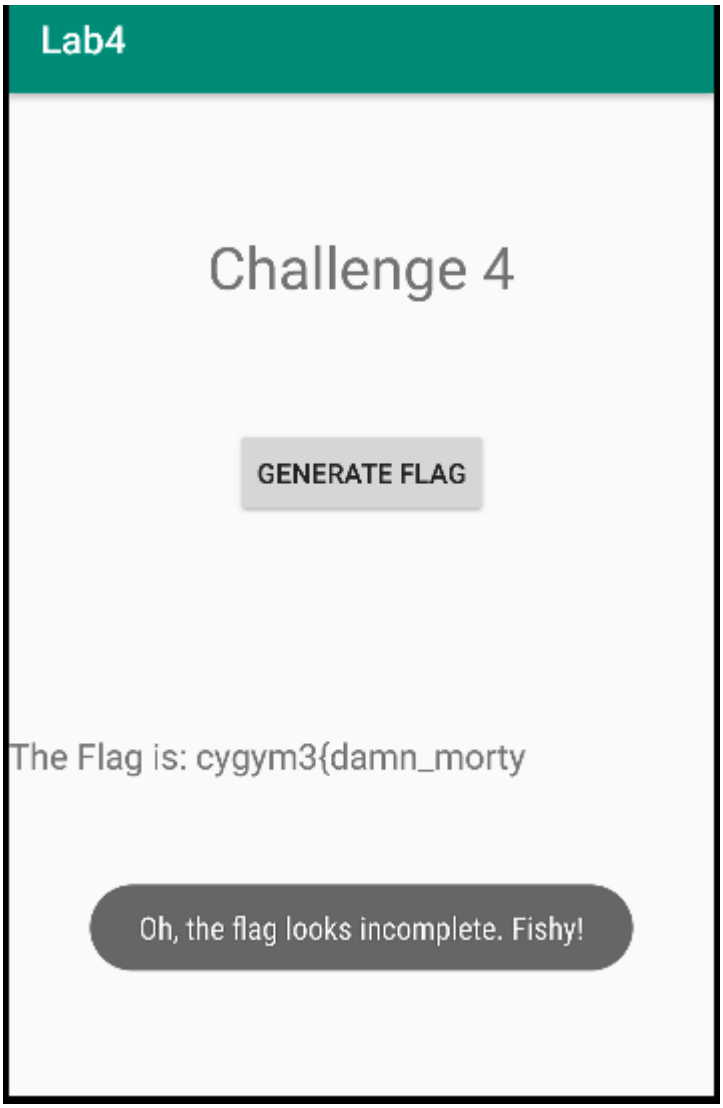
So here, there are two possibilities..

- This is a false flag just to divert us from the path
- This is an incomplete flag

> *"}" is missing in* "cygym3{damn_morty"
>
> "cygym3{damn_morty}" *is this a complete flag ?* **NOOOOOOO!**

Let's move ahead and see the code..

So, the source code shows that after tapping 10 times on the button it will give us another hint…



The above exhibit shows the message after tapping 10 times on **GENERATE FLAG** button

> *The message says* "Oh, the flag looks incomplete. Fishy!"

The hint indicated that the flag is incomplete and we need to find the second part of the flag.

Let's read more source code..

```
public void onCreate(Bundle bundle) {
  super.onCreate(bundle);
  setContentView((int) R.layout.activity_main);
  this.f1003a = (TextView) findViewById(R.id.flagValue);
  this.a = (Button) findViewById(R.id.button);
  int a2 = jb.a(this);
  this.f1005b = "The Flag is: ";
  if (a2 <= 0) {
    Toast.makeText(this, "Application Tampered", 1).show();
    finishAffinity();
  }
  this.f1004a = "cygym3{damn_morty";
  this.a.setOnClickListener(new View.OnClickListener() {
    public final void onClick(View view) {
      MainActivity.this.b++;
```

```
        if (MainActivity.this.b == 10) {
            Toast.makeText(MainActivity.this, "Oh, the flag looks incomplete. Fishy!", 0).show();
            MainActivity.this.b = 0;
        } else if (MainActivity.this.b == 3) {
            MainActivity.this.f1003a.setText(MainActivity.this.f1005b + MainActivity.this.f1004a);
        }
    }
});
ja jaVar = new ja();
jaVar.a("mRte5rK801pHFRv");
jaVar.a("LYdn7wH2lAMhvAN");
jaVar.a("xT2Zuo3nCbiqvyV");
}
```

three random hardcoded strings and called by a function ja.jaVar(String)

The above exhibit shows that function call in reverse engineered source code

We can see there are three random hard coded strings being called by a function ***ja.jaVar(<String>)***

Let's read the function declaration…

```
public final class ja {
    private int a = 0;

    /* renamed from: a  reason: collision with other field name */
    private String f1646a;

    public final void a(String str) {
        this.a++;
        if (this.a < 2) {
            this.f1646a = str;
        }
    }
}
```

the return type of the function is void ?
Am I going correct ?

The above exhibit shows the definition of the function in reverse engineered source code

> So here, we are in a chaotic situation.. Strings are getting called but the function returns void. ***Strange***!

It seems like a dead end.

Let's start analysing the APK again…

Reverse engineering the application again using ***apktool*** with a very basic command.

```
apktool d <apk-name>
```

OR

```
apktool d <apk-name> -o <new-folder-name>
```

Note : Here **"d"** stands for decompiling the application.

```
saurabhjain@[redacted]:~$ apktool d com.cybergym.lab4.apk -o UniverseWeird
Picked up _JAVA_OPTIONS: -Dawt.useSystemAAFontSettings=on -Dswing.aatext=true
I: Using Apktool 2.4.1 on com.cybergym.lab4.apk
I: Loading resource table...
I: Decoding AndroidManifest.xml with resources...
I: Loading resource table from file: /home/saurabhjain/.local/share/apktool/framework/1.
apk
I: Regular manifest package...
I: Decoding file-resources...
I: Decoding values */* XMLs...
I: Baksmaling classes.dex...
I: Copying assets and libs...
I: Copying unknown files...
I: Copying original files...
```

The above exhibit shows that the application is being reverse engineered using apktool

> The basic difference between **apktool** and **jadx-gui** is that, we get smali code while analysing APK using **apktool** and on the other side we get java source code using **jadx-gui**.

Straight away after reverse engineering the source code, we can start analysing the application XML files in a text editor.

```
<string name="enter_fname">Enter First Name</string>
<string name="enter_lname">Enter Last Name</string>
<string name="enter_mobile">Mobile Number</string>
<string name="enter_mobile_number">Enter mobile number</string>
<string name="firebase_database_url">https://moksh-test.firebaseio.com</string>
<string name="google_storage_bucket">moksh-test</string>
```

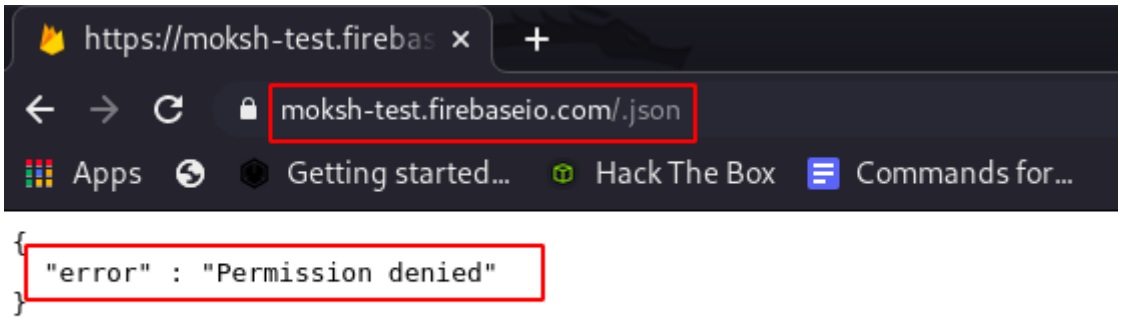
```
<string name="know_more">Know more</string>
<string name="label_cancel">Cancel</string>
<string name="label_continue">Continue</string>
```

The above exhibit shows the firebase instance identified in application source code

See what we got.. a Firebase instance.. ***JUICY!***

Lets see the instance, if it is properly configured or not ? by checking it as

> https://<firebase-instance>/**.json**



The above exhibit shows how to check the configuration of an firebase instance so that it does not leak any information

The firebase instance looks like it is properly configured and does not leak any information.

Are we at a dead end again ???

What else we founded in “res/values/strings.xml”

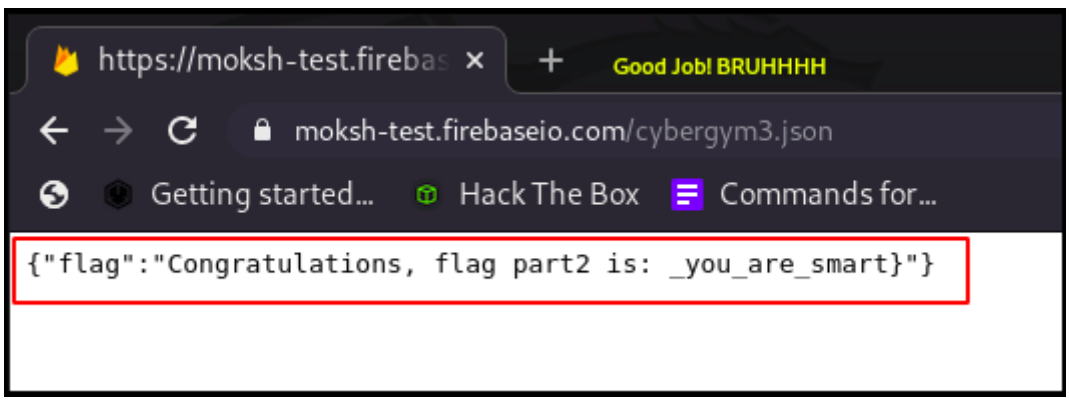
```
<string name="lable_please_wait">please wait</string>
<string name="lbl_brightness">BRIGHTNESS</string>
<string name="lbl_contrast">CONTRAST</string>
<string name="lbl_fashion">Fashion</string>
<string name="lbl_url_part2">/cybergym3.json</string>
<string name="mobile">Mobile Number</string>
<string name="mobile_code">+91</string>
<string name="mobile_number">Mobile Number</string>
<string name="msg_button">Generate Flag</string>
<string name="msg_challenge">Challenge 4</string>
```

The above exhibit shows the other half of the URL being stored in application source code

> The other part of the URL : /cybergym3.json

Now let’s check the firebase configuration again.

> *https://<firebase-instance>/ + <second-part-of-url>*



The above exhibit shows that we finally found the flag

Hey ! You are smart !

So let’s complete the flag

> **cygym3{damn_morty_you_are_smart}**

Takeaways :

> *Learned reading application source code*
>
> *Tracing down application source code*
>
> *Checking firebase instance configuration*
>
> *Hard coding data is the first step to danger zone*

Reverse Engineering Android Ctf Writeup Pentesting Security