Morty's New Tool : Android Application Based CTF Challenge Walkthrough

Saurabh Jain
Sep 27 · 5 min read

Morty's new tool is an intermediate level Android application CTF challenge. The basic aim of this CTF challenge is to learn the dynamic transformation in the code at run time, reverse engineering of native libraries and much more.

It will give an atmosphere of real time scenarios which will teach us the working of an application, its process and data flow.

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

Tools Used :

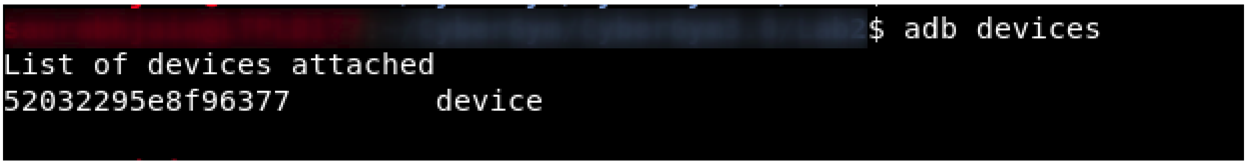
> ***adb*** *: command line tool that lets you communicate with device*

> ***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*
>
> ***Ghidra*** *: Open source reverse engineering tool*
>
> ***Frida*** *: dynamic code instrumentation toolkit for native applications*

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.



```
[redacted]$ adb devices
List of devices attached
52032295e8f96377        device
```

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.

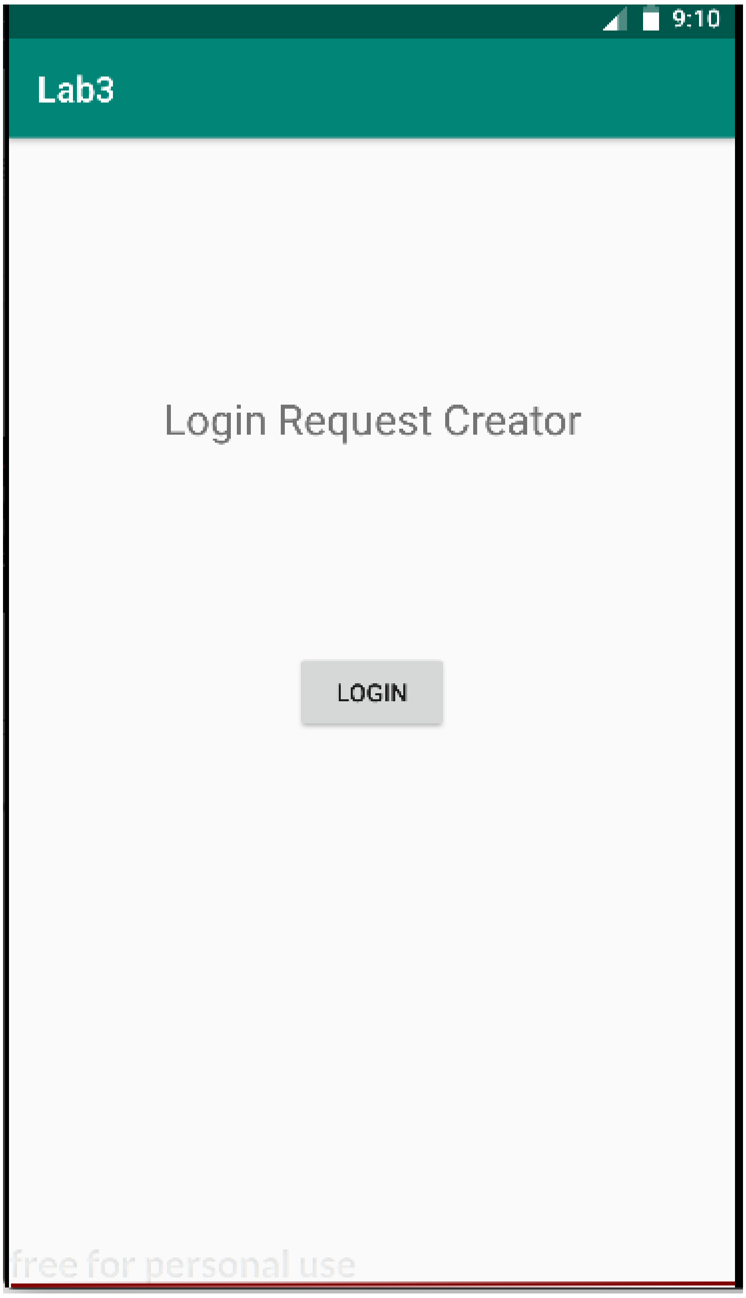
After downloading the application from the above given link, the application can be installed in device/emulator by a very simple command.

> *adb install <apk-name>*

```
saurabhjain@[redacted]$ adb install com.cybergym.lab3.apk
Performing Push Install
com.cybergym.lab3.apk: 1 file pushed. 36.5 MB/s (1065840 bytes in 0.028s)
        pkg: /data/local/tmp/com.cybergym.lab3.apk
Success
```

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

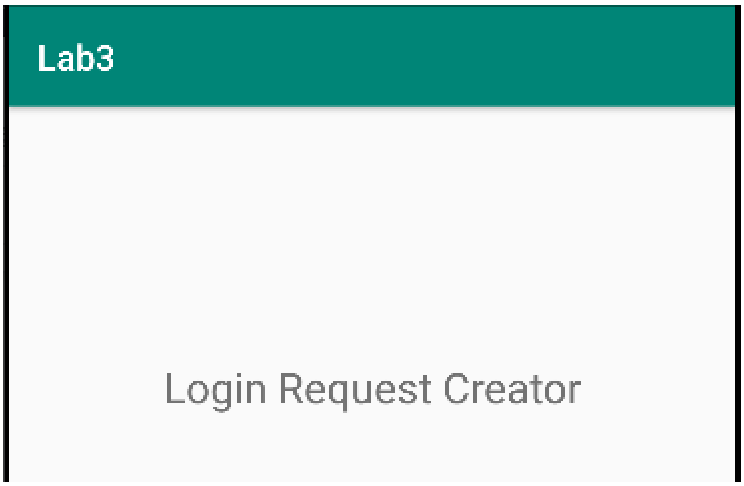
We can run the application in the device/emulator

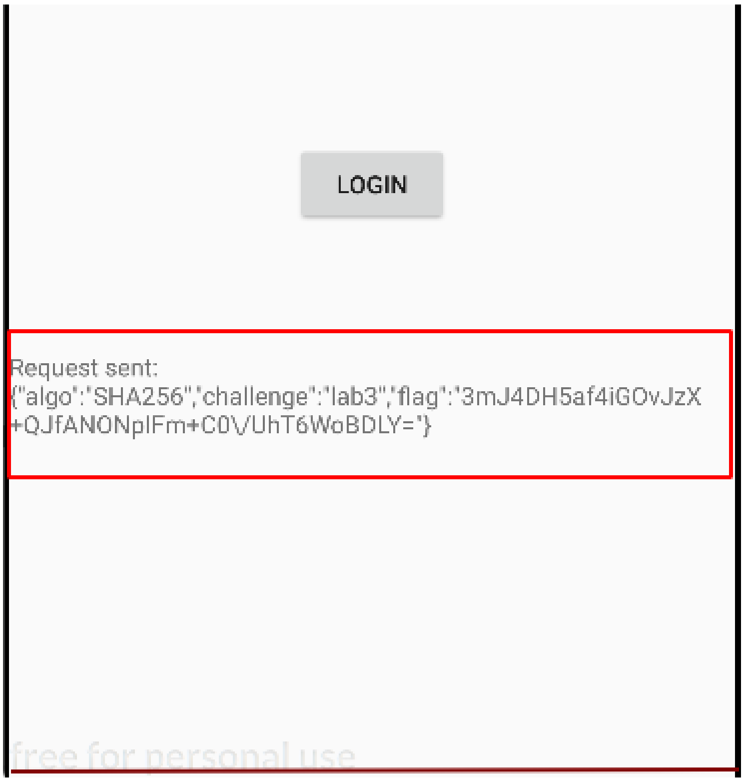


The above exhibit shows the first page after running the application in the device

Here, we can see there is a button named "**LOGIN**".

On a single click we got a message..





The above exhibit shows the action performed after clicking in LOGIN button

The message says..

> Request Sent : { "algo" : "SHA256" , "challenge" : "lab3" , "flag" : <value-of-the-flag> }

So from here, we can grab the information that the

> *Algorithm used for hashing is **SHA-256***
>
> *Value of the flag is stored in the **flag** parameter*

We just have to break the hash and we have the flag..

How much time will it take to break a SHA-256 hash ?

> *So, basically I need a supercomputer with very high computational power for brute forcing a hash function which will take approximately (2^128) operations and will consume only a million years.*

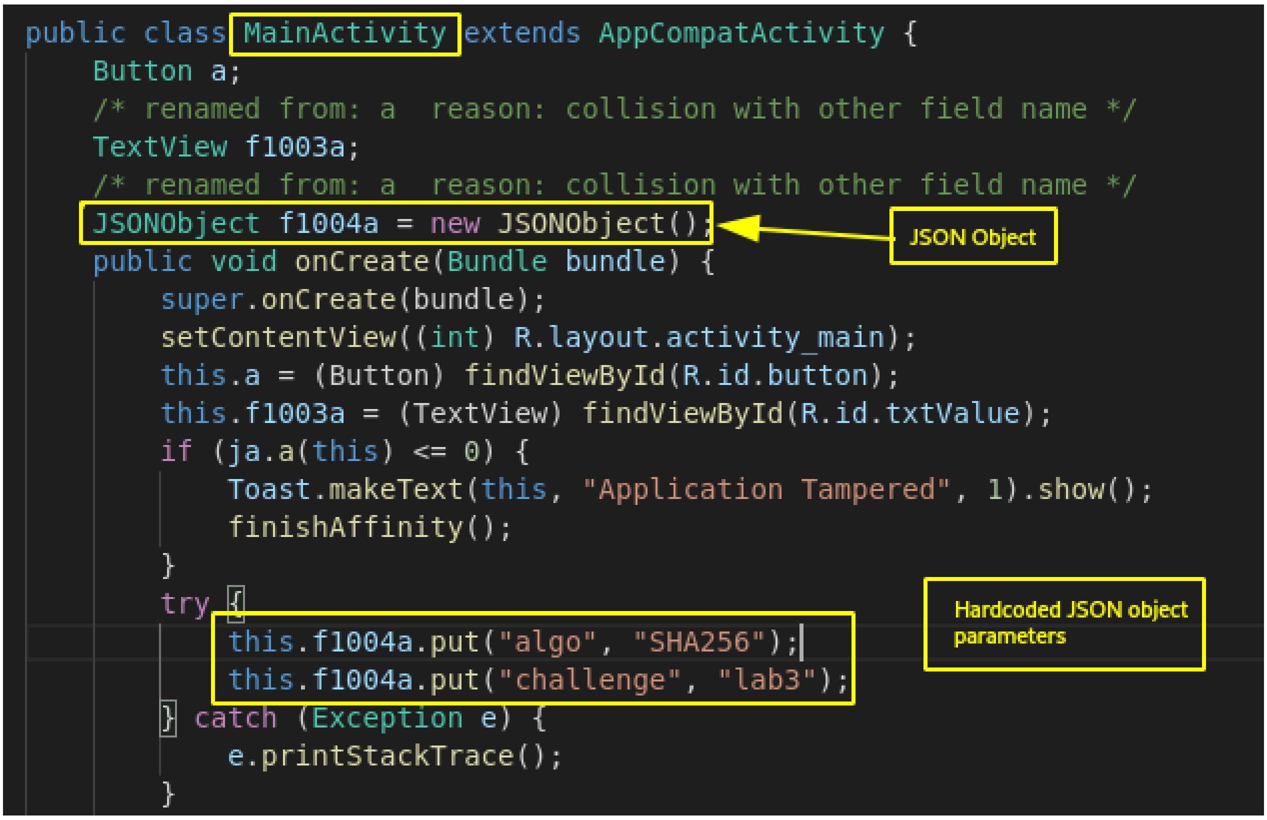
Well! We need to find an alternative for this.

> *Let's get back to our reverse engineer technique. We have the **jadx-gui**, reverse engineering tool for reading the application source code.*

So for reverse engineering entering a simple command

> *jadx-gui <apk-name>*

Reading the code from *MainActivity.java*



```
public class MainActivity extends AppCompatActivity {
    Button a;
    /* renamed from: a  reason: collision with other field name */
    TextView f1003a;
    /* renamed from: a  reason: collision with other field name */
    JSONObject f1004a = new JSONObject();
    public void onCreate(Bundle bundle) {
        super.onCreate(bundle);
        setContentView((int) R.layout.activity_main);
        this.a = (Button) findViewById(R.id.button);
        this.f1003a = (TextView) findViewById(R.id.txtValue);
        if (ja.a(this) <= 0) {
            Toast.makeText(this, "Application Tampered", 1).show();
            finishAffinity();
        }
        try {
            this.f1004a.put("algo", "SHA256");
            this.f1004a.put("challenge", "lab3");
        } catch (Exception e) {
            e.printStackTrace();
        }
```


The above code snippet shows that the source code is obfuscated and JSON object being created

> *JSON object named* **f1004a** *is being created and two parameters are being hardcoded.*

On reading further code…

```
this.a.setOnClickListener(new View.OnClickListener() {
    public final void onClick(View view) {
        try {
            MainActivity.this.f1004a.put("flag", new t().a().replaceAll("\\n", ""));
            TextView textView = MainActivity.this.f1003a;
            textView.setText("Request sent: " + MainActivity.this.f1004a.toString());
        } catch (Exception e) {
            e.printStackTrace();
        }
    }
});
```

The above code snippet shows the function call for capturing the value of the flag

> *We can grab that, there is a class named* **t()** *which has a function* **a()** *which is returning* SHA-256 *hash value of the flag*

Let's have a look at ***class t***

class t and function a()

```
public class t {
    public final String a() {
        String str = new String();
        char[] charArray = k1().toCharArray();
        String str2 = str;
        for (int i = 0; i < charArray.length; i += 2) {
            StringBuilder sb = new StringBuilder();
            sb.append(charArray[i]);
            sb.append(charArray[i + 1]);
            str2 = str2 + ((char) Integer.parseInt(sb.toString(), 16));
        }
        String str3 = new String();
        char[] charArray2 = k2().toCharArray();
        for (int i2 = 0; i2 < charArray2.length; i2 += 2) {
            StringBuilder sb2 = new StringBuilder();
            sb2.append(charArray2[i2]);
            sb2.append(charArray2[i2 + 1]);
            str3 = str3 + ((char) Integer.parseInt(sb2.toString(), 16));
        }
        String str4 = "cygym3{" + str3 + "}";
        try {
            ("cygym3{" + str2 + "}").getBytes("UTF-8");
            return a(str4.getBytes("UTF-8"));
        } catch (Exception e) {
            e.printStackTrace();
            return "123";
        }
    }
```

The above code snippet shows the application source code having **Class t** and **function a()**

> *After reading the function **a()**, we can grab a bit of information that two functions **k1()** and **k2()** are playing some role.*

Let's have a look at them.

```
public native String k1();

public native String k2();
```

The above source code shows the declaration of two functions k1() and k2() in source code

> *Here,* **k1()** *and* **k2()** *are two native functions which are being defined in a native library named* **"local"**

```
static      Native library "local"
{
    System.loadLibrary("local");
}
```

The above code snippet shows that the native library named "local" is loaded in application source code

So now for reading a native library we need a binary analysis reverse engineering tool i.e ***Ghidra***.

> function **k1()** opened in ghidra

```
void Java_com_moksh_lab3_t_k1(longlong *plParm1)

{
                    /* WARNING: Could not recover jumptable at 0x001006bc
                    /* WARNING: Treating indirect jump as call */
  (**(code **)(*plParm1 + 0x538))(plParm1,&UNK_0010095e);
  return;
}
```

The above code snippet is from ghidra for native function k1()

Here, we can see the function ***k1()*** is of no use to us as it returns nothing (void), we need to monitor ***k2()***.

> function **k2()** opened in ghidra

```
void Java_com_moksh_lab3_t_k2(longlong *plParm1)

{
  int iVar1;
  longlong in_tpidr_el0;
  int iVar2;
  char cVar3;
  longlong lVar4;
  longlong lVar5;
  ulonglong uVar6;
  char *pcVar7;
  undefined local_90 [48];
  char local_60 [24];
  longlong local_48;

  local_48 = *(longlong *)(in_tpidr_el0 + 0x28);
  lVar5 = 0;
  iVar2 = 0;
  do {
    iVar1 = 0;
    if (iVar2 != 10) {
      iVar1 = iVar2;
    }
    switch(iVar1) {
    case 0:
      cVar3 = "ghYsLZomnl z5K3XPTBlr8ansI7k"[lVar5];
```

The above code snippet is from ghidra for native function k2()

```
switchD_00100768_caseD_a:
    lVar5 = lVar5 + 1;
    iVar2 = iVar1 + 1;
    if (lVar5 == 0x17) {
      lVar5 = 0;
      lVar4 = 0x16;
      do {
        cVar3 = local_60[lVar5];
        local_60[lVar5] = local_60[lVar4];
        local_60[lVar4] = cVar3;
        lVar5 = lVar5 + 1;
        lVar4 = lVar4 + -1;
      } while (lVar5 < lVar4);
      if (local_60[0] == '\0') {
        uVar6 = 0;
      }
      else {
        uVar6 = 0;
        pcVar7 = (char *)((ulonglong)local_60 | 1);
        do {
          FUN_001008b0(local_90 + uVar6,0xffffffffffffffff);
          cVar3 = *pcVar7;
          uVar6 = uVar6 + 2;
          pcVar7 = pcVar7 + 1;
        } while (cVar3 != '\0');
        uVar6 = uVar6 & 0xfffffffe;
      }
      local_90[uVar6] = 0;
      (**(code **)(*plParm1 + 0x538))(plParm1,local_90);
      if (*(longlong *)(in_tpidr_el0 + 0x28) == local_48) {
        return;
```

The above code snippet is from ghidra for native function k2()

So, after trying to read the code we got an understanding that is very dynamic and we cannot directly capture the flag straight from the binaries by just reading the native code.

We have an alternative approach to this, which is ***runtime hooking***.

> ***Runtime hooking*** *is a concept in which function variables can be monitored and modified at runtime.*

For doing this we have a tool named **FRIDA**, a dynamic instrumentation tool.

> **Note** : You can find the installation and configuration of FRIDA in your system [here]. A rooted device/emulator is required for running **FRIDA**.

A short script in JS for hooking the return value of function ***k2()***.

```
/*frida -U -f com.moksh.lab3 -l flag.js --no-pause */

console.log("=====SCRIPT LOADED SUCCESSFULLY=====");
Java.perform(function x(){
    console.log("Inside Java function");
    var application = Java.use("com.moksh.lab3.t");
  application.a.overload().implementation = function(){
        console.log("**************************************")
        var class_name = Java.use("com.moksh.lab3.t");
        var stringClass = Java.use("java.lang.String");

        var t_class_object = class_name.$new()
        var str = stringClass.$new();
        str = t_class_object.k2();
        console.log("Original arg: " + str);
        console.log("**************************************")
    };
});
```

The above code snippet shows the frida script used to hook the function k2() at run time.

Running the above written JS script for hooking using ***FRIDA***

```
[redacted] $ frida -U -f
com.moksh.lab3 -l flag.js --no-pause

     ____
    / _  |   Frida 12.8.20 - A world-class dynamic instrumentation toolkit
   | (_| |
    > _  |   Commands:
```

```
   /_/ |_|       help      -> Displays the help system
   . . . .       object?   -> Display information about 'object'
   . . . .       exit/quit -> Exit
   . . . .
   . . . .    More info at https://www.frida.re/docs/home/
Spawning `com.moksh.lab3`...
=====SCRIPT LOADED SUCCESSFULLY=====
Spawned `com.moksh.lab3`. Resuming main thread!
[Samsung::com.moksh.lab3]-> Inside Java function
[Samsung::com.moksh.lab3]-> *************************************
Original arg: 4D6F7274795F736F6C7665645F46726964615F666C6167
*************************************
```

The above exhibit shows the return value of the function k2() captured by hooking it by FRIDA

We got a string, as a return value of function ***k2()*** in *Original arg* parameter.

Now we can replicate the code written in ***class t*** and compile it to find the flag.

```
public class HelloWorld{

     public static void main(String []args){
        String str3 = new String();
        String k2 = "4D6F7274795F736F6C7665645F46726964615F666C6167";
        char[] charArray2 = k2.toCharArray();
        for (int i2 = 0; i2 < charArray2.length; i2 += 2) {
            StringBuilder sb2 = new StringBuilder();
            sb2.append(charArray2[i2]);
            sb2.append(charArray2[i2 + 1]);
            str3 = str3 + ((char) Integer.parseInt(sb2.toString(), 16));
        }
        String str4 = "cygym3{" + str3 + "}";
        System.out.println(str4);
     }
}
```

```
$javac HelloWorld.java

$java -Xmx128M -Xms16M HelloWorld

cygym3{Morty_solved_Frida_flag}
```

The above exhibit shows the value of flag

> **CONGRATULATIONS !! Morty**

You got your new dynamic analysis tool i.e ***FRIDA***

Takeaways :

> *Learned reading application source code*
>
> *Tracing down application source code*
>
> *Using **ghidra** for reverse engineering binary files*
>
> *Dynamic method hooking using* **FRIDA**
>
> *Hard coding code is the key to failure*

Frida Android Pentesting Ctf Writeup Reverse Engineering