

Android Penetration Testing: APK Reversing (Part 2)

February 18, 2021 By Raj Chandel

Introduction

Android reverse engineering refers to the process of decompiling the APK for the purpose of investigating the source code that is running in the background of an application. In **part 1** (refer [here](#)) we saw how an attacker would be able to decompile, change the smali files and recompile and sign the APK to perform some kind of function, for example, changing root detection logic of the application. In this extension of reversing articles, we'll see different scenarios in which an attacker is able to extract juicy information by decompiling an APK while understanding various default files present in android. It is wrong to say this article covers all of the aspects of reversing and extracting info, however, it will lay out a fundamental process as to how one can proceed to do so. Let's begin

Table of Content

- Understanding default files
- Understanding improper WebView implementation
- Hardcoded secrets
- Secrets in strings.xml file
- Insecure classes implementation
- Insecure decryption method for hardcoded values
- Insecure cryptography in SQLite database
- Hard coded AWS credentials

Understanding default files

AndroidManifest.xml – Manifest literally means to show. This is the most literal file in the whole of the application as it does precisely what it's name suggests. It begins with a **<manifest>** tag. It should also contain **“xmlns: attribute”** which declares several system attributes used within the file. It declares:

- **<permissions>** – Permissions that APK requires to run
- **<activity>** – Various activities in the APK
- **<intent-filter>** – Intent filters
- **<data android:scheme=”string” />** – Data Schemes
- **<action android:name=”string” />** – Action that an intent performs
- **<uses-configuration>** tag – specifies input mechanisms
- **<uses-sdk>** tag – specifies android API to be used

Hence, Manifest file is the guide which is needed by the package to make particular components behave the way the developer intended them to.

Strings.xml – This XML file contains strings that can be referenced from the application or from other resource files (such as an XML layout). In other words, I can use a reference to strings' object in my actual Java class so that I don't have to type in the complete value then and there and also if there is a need to change that value in future, I won't need to go to my java class every single time to change it, I can just change the value in the strings file. One example is supposed in Java file the code flows like this:

```
String string = getString(R.string.hello);
```

Here, R.string.<string object> sets the value of variable string. I can make its reference in the layout file like this:

<TextView

android:layout_width="fill_parent"

android:layout_height="wrap_content"

android:text="@string/hello" />

This sets the type of view that the hello object would have while displaying in app. Here, **@string/hello** refers to strings.xml file's hello object. So, finally in the strings.xml file code would be something like:

<?xml version="1.0" encoding="utf-8"?>

<resources>

<string name="hello">Hello!</string>

</resources>

So, the variable string would indirectly get assigned a value "Hello!" Note that <resources> tag allocates a string value.

R.java file – It is an auto-generated file by aapt (Android Asset Packaging Tool) that contains resource IDs for all the resources of res/ directory. If you create any component in the **activity_main.xml** file, id for the corresponding component is automatically created in this file. This id can be used in the activity source file to perform any action on the component. If you delete this, it'll get auto-generated again.

There are various other default files as well, we'll talk about them later when we discover each.

Understanding improper WebView implementation

If you have no idea what a WebView is please first read [this](#) article. This will lay up your basics. Now, this article specifically talks about what code to look for in order to understand if webview implementation is

insecure or not. Now, for the purpose of this article, I'll be using InjuredAndroid.apk developed by Kyle Benac. You can find it [here](#). It would look something like this



8:22

InjuredAndroid



XSSTEXT

FLAG ONE - LOGIN

FLAG TWO - EXPORTED ACTIVITY

FLAG THREE - RESOURCES

www.hackingarticles.in

FLAG FOUR - LOGIN 2

FLAG FIVE - EXPORTED BROADCAST RECEIVER

FLAG SIX - LOGIN 3


FLAG SEVEN - SQLITE

FLAG EIGHT - AWS

FLAG NINE - FIREBASE

In the very first challenge of the application, we'll see a poor webview implementation. First activity's source code is:

master ▾ InjuredAndroid / InjuredAndroid / app / src / main / java / b3nac / injuredandroid / XSSTextActivity.java

 **B3nac** Assembly flag exercise progress

1 contributor

26 lines (21 sloc) | 813 Bytes

```
1 package b3nac.injuredandroid;
2
3 import android.content.Intent;
4 import android.os.Bundle;
5 import android.view.View;
6 import androidx.appcompat.app.AppCompatActivity;
7 import android.widget.EditText;
8
9 public class XSSTextActivity extends AppCompatActivity {
10     public static final String POST_STRING = "com.b3nac.injuredandroid.DisplayPostXSS";
11
12     @Override
13     protected void onCreate(Bundle savedInstanceState) {
14         super.onCreate(savedInstanceState);
15         setContentView(R.layout.activity_xsstext);
16     }
17
18     public void submitText(View view) {
19         Intent intent = new Intent(this, DisplayPostXSS.class);
20         EditText editText = findViewById(R.id.editText);
21         String post = editText.getText().toString();
22         intent.putExtra(POST_STRING, post);
23         startActivity(intent);
24     }
25 }
```

Here, we can see that a string type variable post is receiving a value from an input field and supplying it to DisplayPostXSS.class using intent. We'll understand what happens when it reaches the said class from the source code:

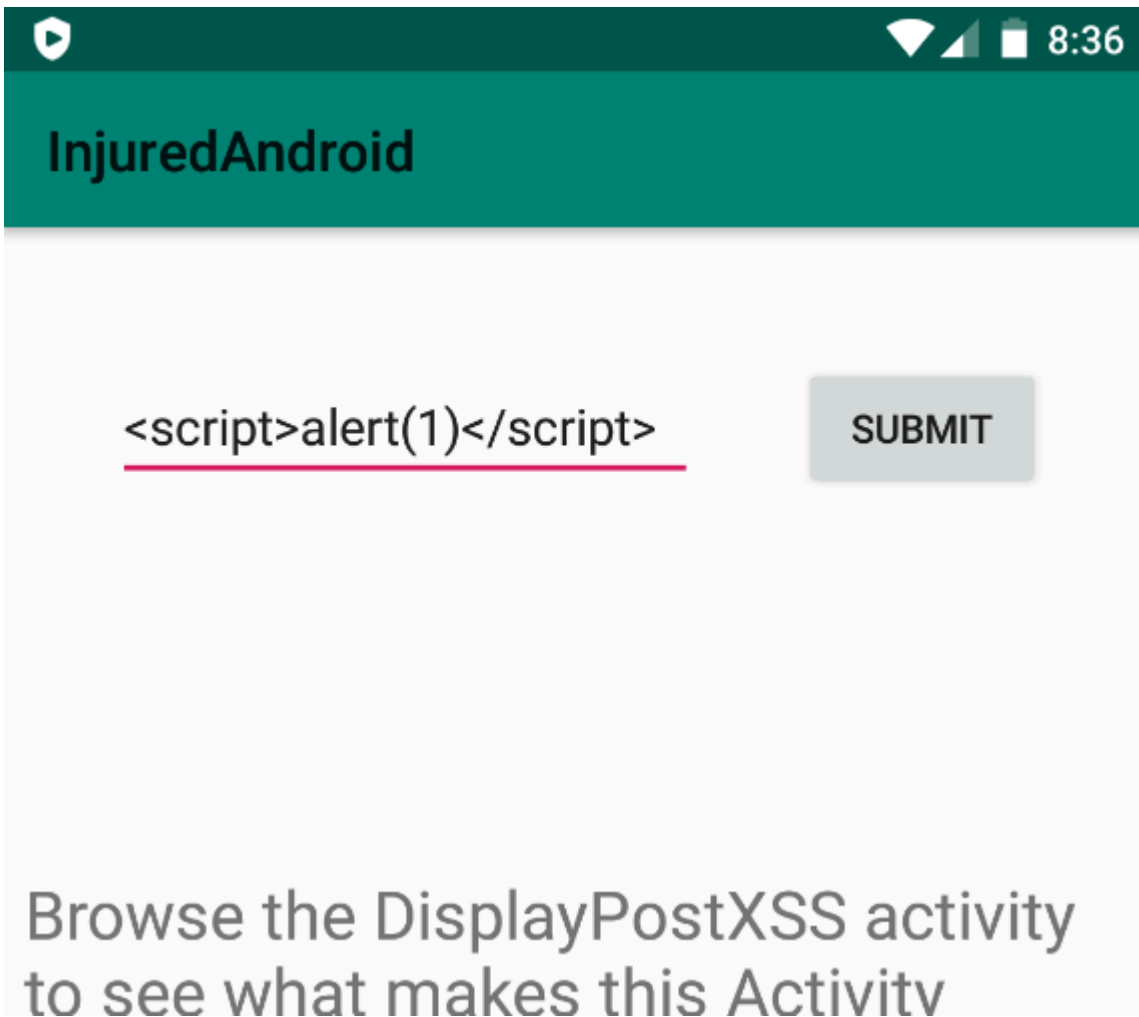
**B3nac** Fixed SecurePreferences fix. Some dank Context.

1 contributor

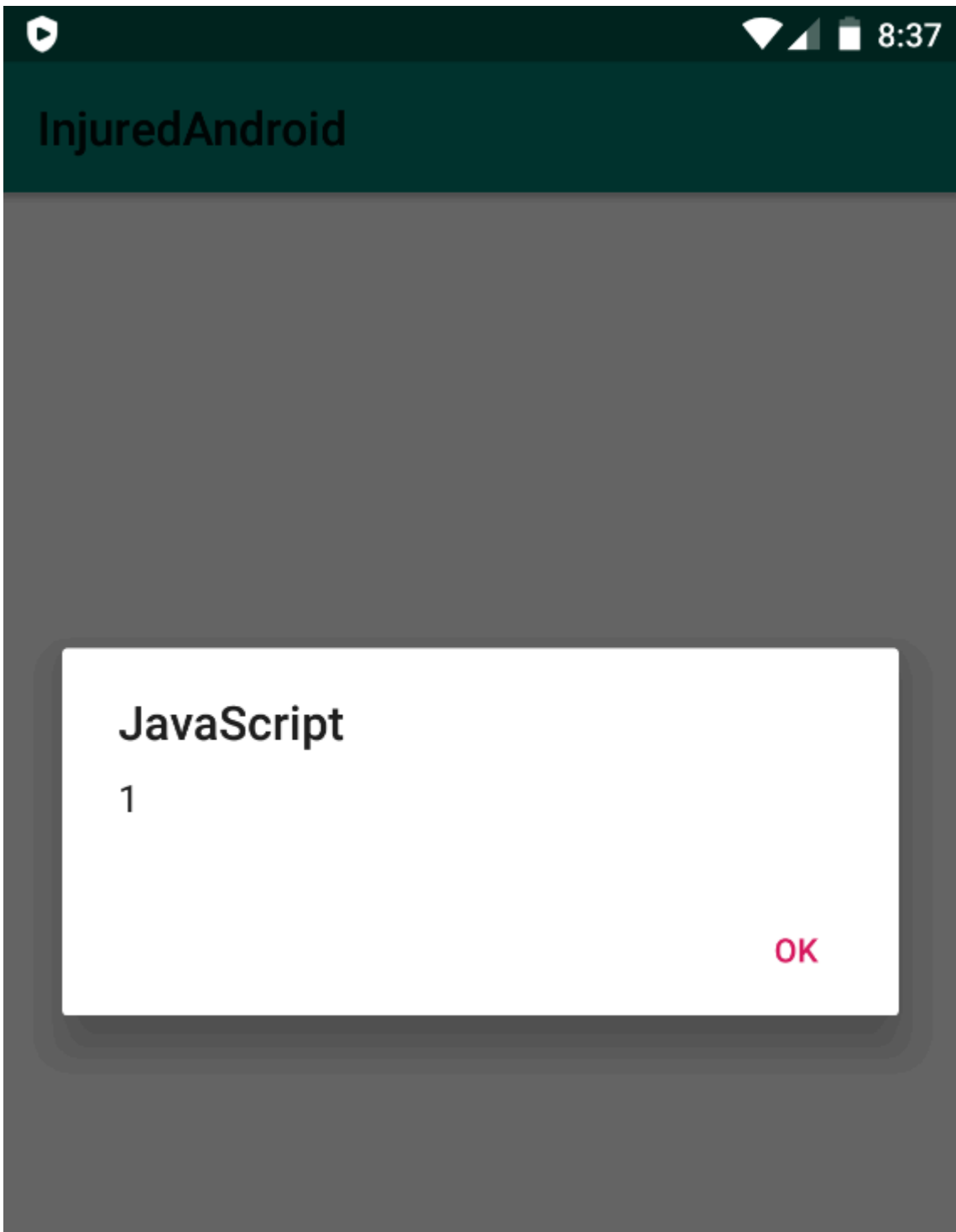
19 lines (17 sloc) | 673 Bytes

```
1 package b3nac.injuredandroid
2
3 import android.os.Bundle
4 import android.webkit.WebChromeClient
5 import android.webkit.WebView
6 import androidx.appcompat.app.AppCompatActivity
7
8 class DisplayPostXSS : AppCompatActivity() {
9     override fun onCreate(savedInstanceState: Bundle?) {
10         super.onCreate(savedInstanceState)
11         val vulnWebView = WebView(this)
12         setContentView(vulnWebView)
13         val intent = intent
14         val postText = intent.getStringExtra(XSSTextActivity.POST_STRING)
15         vulnWebView.settings.javaScriptEnabled = true
16         vulnWebView.webChromeClient = WebChromeClient()
17         vulnWebView.loadData(postText, "text/html", "UTF-8")
18     }
19 }
```

Here, we can easily see that the said input is getting parsed as an HTML and being displayed within a webview. Also, JavaScript is enabled. This means only one thing for us researchers. XSS!



After sending in the payload we can see that WebView is able to display the supplied input in an HTML format like this



Hence, this taught us how we can look for XSS bugs by decompiling the application

HardCoded Secrets

Oftentimes, for ease of access developers would hard code various secret keys, decryption functions and other juicy things within the application and so by reversing the APK we can extract them. For example, if we see **FlagOneLoginActivity** developer has laid out a little hint at the bottom.

FlagOneLoginActivity

Enter flag

SUBMIT

Input the flag and push submit.



The flag is also under the GUI.

Now, let's reverse the application using JADX and see what's hidden under GUI. It can be easily inferred even after obfuscation that submitFlag() method is comparing a user supplied input with a given flag "Flag_0n3". Hence, the entire login mechanism is bypassed just by looking for the right class after reversing.



```
21     a(FlagOneLoginActivity flagOneLoginActivity) {
22         this.f1887c = flagOneLoginActivity;
23     }
24
25     public final void onClick(View view) {
26         if (this.f1887c.F() == 0) {
27             Snackbar X = Snackbar.X(view, "The flag is right under your nose.", 0);
28             X.Y("Action", null);
29             X.N();
30             FlagOneLoginActivity flagOneLoginActivity = this.f1887c;
31             flagOneLoginActivity.G(flagOneLoginActivity.F() + 1);
32         } else if (this.f1887c.F() == 1) {
33             Snackbar X2 = Snackbar.X(view, "The flag is also under the GUI.", 0);
34             X2.Y("Action", null);
35             X2.N();
36             this.f1887c.G(0);
37         }
38     }
39 }
40
41 public final int F() {
42     return this.t;
43 }
44
45 public final void G(int i) {
46     this.t = i;
47 }
48
49 /* access modifiers changed from: protected */
50 @Override // androidx.fragment.app.d, androidx.activity.ComponentActivity, androidx.core.
51 public void onCreate(Bundle bundle) {
52     super.onCreate(bundle);
53     setContentView(R.layout.activity_flag_one_login);
54     j.g.a(this);
55     C((Toolbar) findViewById(R.id.toolbar));
56     ((FloatingActionButton) findViewById(R.id.fab)).setOnClickListener(new a(this));
57 }
58
59 public final void submitFlag(View view) {
60     EditText editText = (EditText) findViewById(R.id.editText2);
61     d.b(editText, "editText2");
62     if (d.a(editText.getText().toString(), "Flag_0n3")) {
63         Intent intent = new Intent(this, FlagOneSuccess.class);
64         new FlagsOverview().J(true);
65         new j().b(this, "flagOneButtonColor", true);
66         startActivity(intent);
67     }
68 }
69 }
```

After we input the flag, we see that it was indeed correct.



www.hackingarticles.in
Congrats you found the flag! :D

Secrets in strings.xml file

Oftentimes, developers create a reference string and use that object in Java class whose actual value is stored in strings.xml file. For example, in **FlagThreeActivity** we can see the developer has laid out a little hint at the bottom.

FlagThreeActivity

Enter flag

SUBMIT

Input the flag and push submit.



R stands for resources.

Now, let's look at this activity's code in jadx first. We'll observe that an object with a gibberish name is being used.

```
b3nac.injuredandroid.FlagThreeActivity ✕
21     a(FlagThreeActivity flagThreeActivity) {
22         this.fl1898c = flagThreeActivity;
23     }
24
25     public final void onClick(View view) {
26         if (this.fl1898c.F() == 0) {
27             Snackbar X = Snackbar.X(view, "R stands for resources.", 0);
28             X.Y("Action", null);
29             X.N();
30             FlagThreeActivity flagThreeActivity = this.fl1898c;
31             flagThreeActivity.G(flagThreeActivity.F() + 1);
32         } else if (this.fl1898c.F() == 1) {
33             Snackbar X2 = Snackbar.X(view, "Check .xml files.", 0);
34             X2.Y("Action", null);
35             X2.N();
36             this.fl1898c.G(0);
37         }
38     }
39 }
40
41 public final int F() {
42     return this.t;
43 }
44
45 public final void G(int i) {
46     this.t = i;
47 }
48
49 /* access modifiers changed from: protected */
50 @Override // androidx.fragment.app.d, androidx.activity.ComponentActivity, androidx.core.
51 public void onCreate(Bundle bundle) {
52     super.onCreate(bundle);
53     setContentView(R.layout.activity_flag_three);
54     j.g.a(this);
55     C((Toolbar) findViewById(R.id.toolbar));
56     ((FloatingActionButton) findViewById(R.id.fab)).setOnClickListener(new a(this));
57 }
58
59 public final void submitFlag(View view) {
60     EditText editText = (EditText) findViewById(R.id.editText2);
61     d.b(editText, "editText2");
62     if (d.a(editText.getText().toString(), getString(R.string.cmVzb3VyY2VzX3lv))) {
63         Intent intent = new Intent(this, FlagOneSuccess.class);
64         new FlagsOverview().L(true);
65         new j().b(this, "flagThreeButtonColor", true);
66         startActivity(intent);
67     }
68 }
69 }
```

It is quite easily understood that user-supplied input is being compared with a resource that is stored in **strings.xml** file. Hence, what we'll do is decompile the package using **apktool** and then traverse to the directory: **/res/values/strings.xml**

```
apktool d InjuredAndroid-1.0.10-release.apk
```

```
cd InjuredAndroid-1.0.10-release/res/values
cat strings.xml | grep cmVz
```

```
root@hex:/home/hex/Downloads/InjuredAndroid-1.0.10-release# cd res/values/
root@hex:/home/hex/Downloads/InjuredAndroid-1.0.10-release/res/values# cat strings.xml | g
rep cmVzb3
    <string name="cmVzb3VyY2VzX3lv">Flag_thr33</string>
root@hex:/home/hex/Downloads/InjuredAndroid-1.0.10-release/res/values#
```

Hence, we have our third flag.

Insecure classes implementation

Just like an object can be referred from strings.xml file, similarly, other classes can also be implemented to perform a specific function with an object, and its return value be used. For example, in the **FlagFourActivity** we see something similar

Enter flag

SUBMIT

Input the flag and push submit.



Classes and imports.

Now, after observing the code underneath the class we see that return value from a certain **g** class' **a()** method. Now, due to obfuscation the original classes and methods names have been changed.

```
b3nac.injuredandroid.FlagFourActivity X b3nac.injuredandroid.g X
22     }
23
24     public final void onClick(View view) {
25         if (this.f1883c.F() == 0) {
26             Snackbar X = Snackbar.X(view, "Where is bob.", 0);
27             X.Y("Action", null);
28             X.N();
29             FlagFourActivity flagFourActivity = this.f1883c;
30             flagFourActivity.G(flagFourActivity.F() + 1);
31         } else if (this.f1883c.F() == 1) {
32             Snackbar X2 = Snackbar.X(view, "Classes and imports.", 0);
33             X2.Y("Action", null);
34             X2.N();
35             this.f1883c.G(0);
36         }
37     }
38 }
39
40 public final int F() {
41     return this.t;
42 }
43
44 public final void G(int i) {
45     this.t = i;
46 }
47
48 /* access modifiers changed from: protected */
49 @Override // androidx.fragment.app.d, androidx.activity.ComponentActivity, androidx.core.app
50 public void onCreate(Bundle bundle) {
51     super.onCreate(bundle);
52     setContentView(R.layout.activity_flag_four);
53     j.g.a(this);
54     ((FloatingActionButton) findViewById(R.id.fab)).setOnClickListener(new a(this));
55 }
56
57 public final void submitFlag(View view) {
58     EditText editText = (EditText) findViewById(R.id.editText2);
59     d.b(editText, "editText2");
60     String obj = editText.getText().toString();
61     byte[] a2 = new g().a();
62     d.b(a2, "decoder.getData()");
63     if (d.a(obj, new String(a2, d.p.c.f3141a))) {
64         Intent intent = new Intent(this, FlagOneSuccess.class);
65         new FlagsOverview().I(true);
66         new j().b(this, "flagFourButtonColor", true);
67         startActivity(intent);
68     }
69 }
70 }
```

Pondering over **g.a()** we'll see that a base64 encoded value is being decoded and supplied as an input and typecasted in a byte array.


```

b3nac.injuredandroid.FlagFourActivity ✕  b3nac.injuredandroid.g ✕
1 package b3nac.injuredandroid;
2
3 import android.util.Base64;
4
5 public class g {
6
7     /* renamed from: a  reason: collision with root package name */
8     private byte[] f1911a = Base64.decode("NF9vdmVyZG9uZV9vbWVsZXRz", 0);
9
10    public byte[] a() {
11        return this.f1911a;
12    }
13 }

```

After encoding this, we'll get this flag.

```
echo "NF9vdmVyZG9uZV9vbWVsZXRz" > to_decode
cat to_decode | base64 --decode
```

```
root@hex:/home/hex/Downloads# echo "NF9vdmVyZG9uZV9vbWVsZXRz" > to_decode
root@hex:/home/hex/Downloads# cat to_decode | base64 --decode
4_overdone_omeletsroot@hex:/home/hex/Downloads#
```

Insecure decryption method for hard coded values

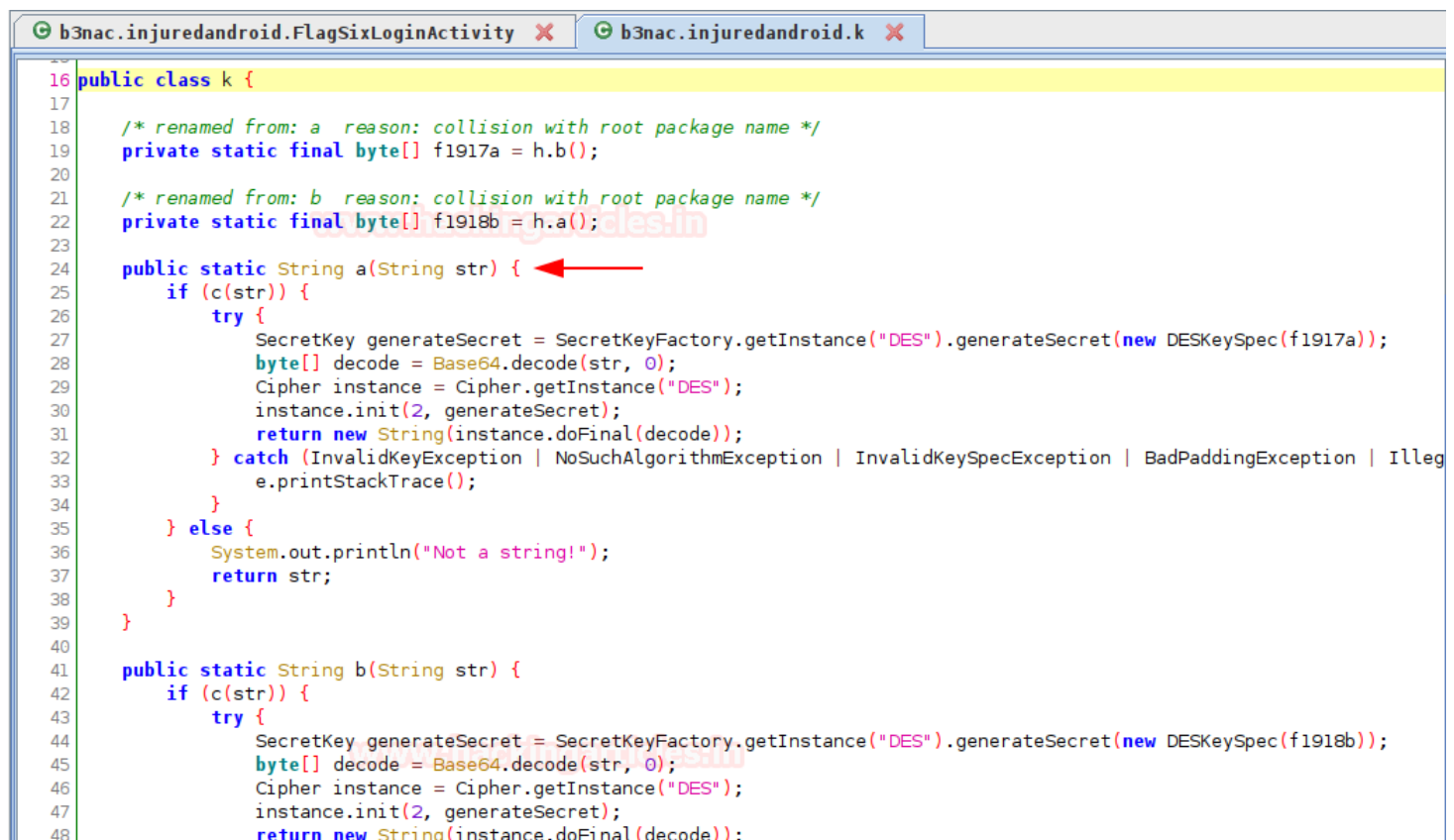
Oftentimes, developers implement methods to generate keys and encrypt some information which is either received from a user or to encrypt a secret such as login credentials within the application. For example in **FlagSixLoginActivity** we can see that a certain `k.a()` return value is being compared with a user input.

```

b3nac.injuredandroid.FlagSixLoginActivity ✕
55
56 public final void G(int i) {
57     this.t = i;
58 }
59
60 /* access modifiers changed from: protected */
61 @Override // androidx.fragment.app.d, androidx.activity.ComponentActivity, androidx.core.app.e, androidx.appcompat.app.d
62 public void onCreate(Bundle bundle) {
63     super.onCreate(bundle);
64     setContentView(R.layout.activity_flag_six_login);
65     j.g.a(this);
66     C((Toolbar) findViewById(R.id.toolbar));
67     ((FloatingActionButton) findViewById(R.id.fab)).setOnClickListener(new a(this));
68 }
69
70 public final void submitFlag(View view) {
71     EditText editText = (EditText) findViewById(R.id.editText3);
72     d.b(editText, "editText3");
73     if (d.a(editText.getText().toString(), k.a("k3FELEG9lnowb0ateGhj5pX6QsXRNJKh///8Jxi8KXW7iDpk2xRxhQ=="))) {
74         Intent intent = new Intent(this, FlagOneSuccess.class);
75         FlagsOverview.D = true;
76         new j().b(this, "flagSixButtonColor", true);
77         startActivity(intent);
78     }
79 }
80 }

```

This gibberish value is a base64 encoded byte array. Now, we are going to inspect k.a() method.



```
16 public class k {
17
18     /* renamed from: a reason: collision with root package name */
19     private static final byte[] f1917a = h.b();
20
21     /* renamed from: b reason: collision with root package name */
22     private static final byte[] f1918b = h.a();
23
24     public static String a(String str) { ←
25         if (c(str)) {
26             try {
27                 SecretKey generateSecret = SecretKeyFactory.getInstance("DES").generateSecret(new DESKeySpec(f1917a));
28                 byte[] decode = Base64.decode(str, 0);
29                 Cipher instance = Cipher.getInstance("DES");
30                 instance.init(2, generateSecret);
31                 return new String(instance.doFinal(decode));
32             } catch (InvalidKeyException | NoSuchAlgorithmException | InvalidKeySpecException | BadPaddingException | Illegal
33                     e.printStackTrace();
34             }
35         } else {
36             System.out.println("Not a string!");
37             return str;
38         }
39     }
40
41     public static String b(String str) {
42         if (c(str)) {
43             try {
44                 SecretKey generateSecret = SecretKeyFactory.getInstance("DES").generateSecret(new DESKeySpec(f1918b));
45                 byte[] decode = Base64.decode(str, 0);
46                 Cipher instance = Cipher.getInstance("DES");
47                 instance.init(2, generateSecret);
48                 return new String(instance.doFinal(decode));
```

This certainly implies that a is receiving string str, which is getting base64 decoded first and then encrypted using a DES key generated. Reversing this function manually is very difficult, but it is certainly possible if we intercept the () method's return value when it calculates and supplies the value to FlagSixLoginActivity. Kyle Benac has given a code in his repository to do this with javascript using Frida. Here is the code:

```
console.log("Script loaded successfully ");
Java.perform(function x() {
    console.log("Inside java perform function");
    var my_class = Java.use("b3nac.injuredandroid.VGV4dEVuY3J5cHRpb25Ud28");
    var string_class = Java.use("java.lang.String");
    my_class.decrypt.overload("java.lang.String").implementation = function (x) { //hooking the new function
        console.log("*****")
        var my_string = string_class.$new("k3FE1EG9lnoWbOateGhj5pX6QsXRNJKh//8Jxi8KXW7iDpk2xRqhQ==");
        console.log("Original arg: " + x);
        var ret = this.decrypt(my_string);
        console.log("Return value: " + ret);
        console.log("*****")
        return ret;
    };
    Java.choose("b3nac.injuredandroid", {
        onMatch: function (instance) {
            console.log("Found instance: " + instance);
            console.log("Result of secret func: " + instance.decrypt());
        },
        onComplete: function () { }
    });
});
```

```
root@hex: /home/hex/Desktop/an... x root@hex: /home/hex/android-too... x root@hex: /home/hex/android-too... x root@hex: /home/hex x
GNU nano 4.8 flag6.js Modified
console.log("Script loaded successfully ");
Java.perform(function x() {
  console.log("Inside java perform function");
  var my_class = Java.use("b3nac.injuredandroid.VGV4dEVuY3J5cHRpb25Ud28");
  var string_class = Java.use("java.lang.String");

  my_class.decrypt.overload("java.lang.String").implementation = function (x) { //hooking the new function
    console.log("*****")
    var my_string = string_class.$new("k3FE1EG9lnoWb0ateGhj5pX6QsXRNJKh//8Jxi8KXW7iDpk2xRxhQ==");
    console.log("Original arg: " + x);
    var ret = this.decrypt(my_string);
    console.log("Return value: " + ret);
    console.log("*****")
    return ret;
  };
  Java.choose("b3nac.injuredandroid", {
    onSuccess: function (instance) {
      console.log("Found instance: " + instance);
      console.log("Result of secret func: " + instance.decrypt());
    },
    onComplete: function () { }
  });
});
});
```

Pretty easy to code this right? No? Then you must follow my previous article on Frida [here](#) to understand how to create hooks in a better way.

Insecure cryptography in SQLite database

Oftentimes, juicy information is obtained from an application's SQLite database. This happens because the developer has stored some values like credentials, keys etc. within the sqlite itself. Let's look at one such implementation in **FlagSevenSqliteActivity**

FlagSevenSqliteActivity

Enter flag www.hackingarticles.in

Enter password

SUBMIT



Stay on this activity.

Now, each database has its own SQLite database associated with itself. We simply need to traverse to this database and dump values. Note that having root access is sometimes critical to do this, other times if your user

has the permissions you'd be able to have a look at the database.

```
adb shell
cd /data/data/b3nac.injuredandroid/databases
ls
sqlite3 Thisisatest.db
select * from Thisisatest;
```

```
vbox86p:/data/data/b3nac.injuredandroid/databases # pwd
/data/data/b3nac.injuredandroid/databases
vbox86p:/data/data/b3nac.injuredandroid/databases # sqlite3 Thisisatest.db
SQLite version 3.9.2 2015-11-02 18:31:45
Enter ".help" for usage hints.
sqlite> select * from Thisisatest;
1|The flag hash!|2ab96390c7dbe3439de74d0c9b0b1767
2|The flag is also a password!|9EEADi^^:?:FC652?5C@:5]7:C632D6:@]4@>^DB=:E6];D@?
sqlite>
```

And just like that we have the flag's hash which is in MD5 and password which is encrypted in some format as well (ROT47). One can decrypt it and is good to go. It is essential for developers not to use old encryption and hashing techniques.

Hardcoded AWS key and access ID

“World is moving to cloud” ~ Google

“But stupidity exists everywhere” ~ Pentesters

Oftentimes developers hard code or create a reference value of AWS bucket credentials. One such example could be seen in **FlagEightLoginActivity**. Here, the developer has given a little hint at the bottom too.

FlagEightLoginActivity

Enter flag

SUBMIT



AWS profiles and credentials.

On exploring the activity's code we see that this client authenticates with aws.

```

class FlagEightLoginActivity : AppCompatActivity() {
    var click = 0
    var database = FirebaseDatabase.getInstance().reference
    var childRef = database.child("/aws") ←

    override fun onCreate(savedInstanceState: Bundle?) {
        super.onCreate(savedInstanceState)
        setContentView(R.layout.activity_flag_eight_login)
        SecureSharedPrefs.setContext(this)

        val toolbar = findViewById<Toolbar>(R.id.toolbar)
        setSupportActionBar(toolbar)
        val mAuth: FirebaseAuth
        mAuth = FirebaseAuth.getInstance() ←
        mAuth.signInAnonymously()
            .addOnCompleteListener(this) { task: Task<AuthResult?> ->
                if (task.isSuccessful) {
                    // Sign in success, update UI with the signed-in user's information
                    Log.d(TAG, "signInAnonymously:success") ←
                } else {
                    // If sign in fails, display a message to the user.
                    Log.w(TAG, "signInAnonymously:failure", task.exception)
                    Toast.makeText(this@FlagEightLoginActivity, "Authentication failed.",
                        Toast.LENGTH_SHORT).show()
                }
            }
    }
}

```

One of the most common places to look for AWS creds is in the strings.xml file. So we check it:

```

cd /InjuredAndroid-1.0.10-release/res/values
cat strings.xml | grep AWS

```

And it worked like a charm.

Now we'll create a profile of aws and can easily use AWS cli to connect to it. To make profile:

```

cd ~ && mkdir .aws
cd .aws && nano credentials

```

Copy the obtained ID and access key in it.

```
root@hex:/home/hex/Downloads/InjuredAndroid-1.0.10-release/res/values# cat strings.xml |  
grep AWS  
    <string name="AWS_ID">AKIAZ36DGKTUI0LD0BN6</string>  
    <string name="AWS_SECRET">KKT4xQAQ5cKzJ0soSImlNFFTRxjYkoc7lvuRP48S</string>  
root@hex:/home/hex/Downloads/InjuredAndroid-1.0.10-release/res/values# cd ../../..  
root@hex:/home/hex/Downloads# cd ..  
root@hex:/home/hex# nano ~/.aws/credentials  
root@hex:/home/hex#  
root@hex:/home/hex# ls  
Android  android-toolkit  Desktop  Documents  Downloads  GNUstep  Music  Pictures  Public  
snap  Templates  Videos  
root@hex:/home/hex# cd ~  
root@hex:~# mkdir .aws  
root@hex:~# cd .aws  
root@hex:~/.aws# nano credentials  
root@hex:~/.aws# cat credentials  
[injuredandroid]  
aws_access_key_id = lookinstrings.xmlnotputtingitheresoawsdoesn'talert  
aws_secret_access_key = lookinstrings.xmlnotputtingitheresoawsdoesn'talert  
root@hex:~/.aws#
```

Conclusion

Even after string obfuscation, various classes, objects and strings can be reversed and used to harm an organisation. It is very essential that developers follow strict practices for security given by OWASP MSTG.

Thanks for reading.