Android Challenges

Cristian Domenico Dramisino (12338532)

Introduction

The challenge can be accessed using the link https://sas.hackthe.space/cms/files/72dca2ff7be89806c320805128da886f1c6a1589f62f35107c0eb646393287c2/braf.2024S.apk .

The file that is downloaded is a apk file named `braf.2024S.apk`

In this challenge we will perform a static analysis and a dynamic analysis on the application in order to extract useful information from it.

The static analysis is used to analyze the app without running it and only looking at the code, dependencies, configurations and resources.

The dynamic analysis is used to analyze the running app in order to extract useful information about the application behavior, such as its resource utilization, error handling mechanisms, runtime interactions with external systems.

For these challenges we will use some interesting tools:

- Android Studio + SDK (adb and emulator)
- the jadx decompiler
- apktool
- the androguard static analysis framework
- the software reverse engineering (SRE) suite of tools
- the frida dynamic analysis framework

Challenge part 1

Introduction

From the description of the first flag, we can understand that maybe in this case there is something that must not happen:

```
This little bug is not a bug at all, it's just a happy little accident giving us a way to surprise the developers.
```

- For example it's possible that the flag is hardcoded somewhere in the application soource code or in the resources of the apk.

Static analysis

So what we will do first is to analyze the content of the apk file using the tool JADX.

Basically we use JADX-GUI to decompile the apk and having an overview of the content of the application, such as the source code and all the resources it uses:

AndroidManifest.xml analysis

We know that one of the most important files in an android application is the AndroidManifest.xml

This file contains many important information about the application such as **information about the package, and also components of the application such as activities, services, broadcast receivers, content providers etc**.

You can find a better overview here

In this specific application, the `AndroidManifest.xml` contains:

```xml
<?xml version="1.0" encoding="utf-8"?>
<manifest xmlns:android="http://schemas.android.com/apk/res/android" android:versionCode="2"
android:versionName="2024.SS.0" android:compileSdkVersion="33" android:compileSdkVersionCodename="13"
package="wien.secpriv.challenges.braf" platformBuildVersionCode="33" platformBuildVersionName="13">
    <uses-sdk android:minSdkVersion="30" android:targetSdkVersion="33"/>
    <uses-permission android:name="android.permission.INTERNET"/>
    <application android:theme="@style/AppTheme" android:label="@string/app_name"
android:icon="@mipmap/braf_launcher" android:name="wien.secpriv.challenges.braf.BobRossApp4Fans"
android:allowBackup="true" android:supportsRtl="true" android:extractNativeLibs="false"
android:networkSecurityConfig="@xml/network_security_config" android:roundIcon="@mipmap/braf_launcher"
android:appComponentFactory="androidx.core.app.CoreComponentFactory">
        <activity android:name="wien.secpriv.challenges.braf.ImageDialog"/>
        <activity android:name="wien.secpriv.challenges.braf.GuessingActivity"/>
        <activity android:name="wien.secpriv.challenges.braf.GalleryActivity"/>
        <activity android:name="wien.secpriv.challenges.braf.QuoteActivity"/>
        <activity android:name="wien.secpriv.challenges.braf.RiddleActivity"/>
        <activity android:name="wien.secpriv.challenges.braf.AboutActivity"/>
        <activity android:name="wien.secpriv.challenges.braf.MainActivity" android:exported="true">
            <intent-filter>
                <action android:name="android.intent.action.MAIN"/>
                <category android:name="android.intent.category.LAUNCHER"/>
            </intent-filter>
        </activity>
        <meta-data android:name="com.google.android.gms.version"
android:value="@integer/google_play_services_version"/>
        <provider android:name="androidx.startup.InitializationProvider" android:exported="false"
android:authorities="wien.secpriv.challenges.braf.androidx-startup">
            <meta-data android:name="androidx.emoji2.text.EmojiCompatInitializer"
android:value="androidx.startup"/>
            <meta-data android:name="androidx.lifecycle.ProcessLifecycleInitializer"
android:value="androidx.startup"/>
        </provider>
        <activity android:theme="@style/Theme.PlayCore.Transparent"
android:name="com.google.android.play.core.common.PlayCoreDialogWrapperActivity" android:exported="false"
android:stateNotNeeded="true"/>
    </application>
</manifest>
```

As we can see there are no interesting information here, but we can notice that `android:minSdkVersion="30"`. We have to take this in mind if we want to use an emulator, because we need an emulator that has atleast the `API 30`.

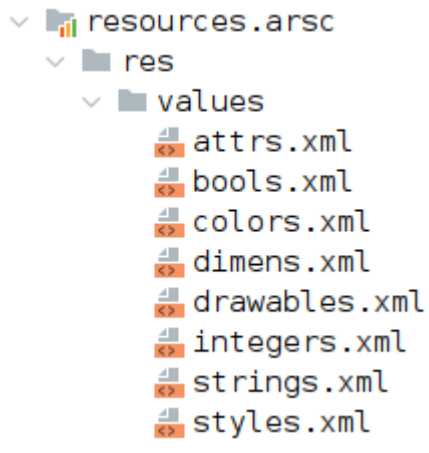
However, there is an activity that has the property `exported = "true"`

```xml
<activity android:name="wien.secpriv.challenges.braf.MainActivity" android:exported="true">
```

- This means that it could be accessed from other application or in general from the extern.
- **We could take note about it**, maybe it could contain some useful information.

Resources Analysis

Since the AndroidManifest.xml has no other interesting information we could take a look at the resources, starting from the generic folder `values` :



==In many cases android applications leak sensitive information because developers hardcode this information in the source code or generally into the file == `strings.xml`

Strings.xml file analysis

This file is generally used for providing a centralized file for managing textual content, facilitating localization, promoting code maintainability, and supporting accessibility features.

These strings can be easily referred also in the source java code of the application.

So it could make sense to study it and try to figure out if there are some useful information in it.

We know that the flag has the format `FLG_PT1{xxx}`, so we can try to find it reading one by one the strings contained in the file or also searching for `FLG_PT1`, always using JADX.

In fact, we could see that we cna find:

```
<string name="easyone">FLG_PT1{here_is_already_the_first_one}</string>
```

So, the flag for the part 1 is: `FLG_PT1{here_is_already_the_first_one}`

Challenge part 2

Professor Bleier Jakob gave me some advice when I was stuck.

Introduction

We use also here JADX-GUI to decompile the apk and to have access to the source code.

We know by the description that the target for the second flag is for sure the GalleryActivity and all the activities related to it:

```
What's our favourite picture in the Gallery? Find the correct name and get a flag! Though it's a bit more complicated since we ~~accidentally~~ hid part of the flag. Hint: Not everything is the same.

The Flag format is `FLG_PT2{xxx_4567890abcde}`. That's a 12-character, lower-case, hexadecimal suffix.
```

So the first thing that we can do is to study the code of:

- GalleryActivity
- GuessingActivity

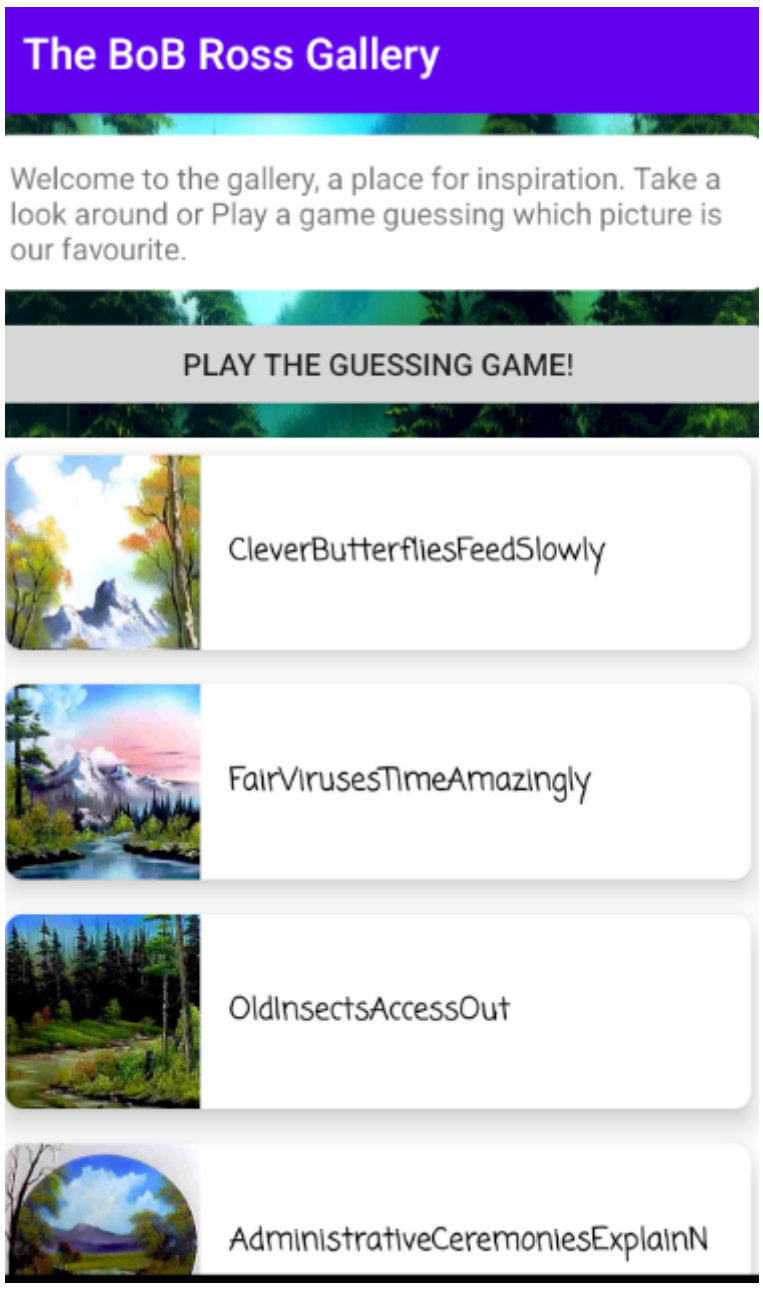
Static analysis

GalleryActivity.java analizing

The code contained in this class is used to charge all the paintings in the Activity that is shown to the user:

```
public void onCreate(Bundle bundle) {
        super.onCreate(bundle);
        setContentView(R.layout.activity_gallery);
        setTitle("The BoB Ross Gallery");
        this.galleryClasses = getGalleryClasses();
        this.recyclerView = (RecyclerView) findViewById(R.id.recyclerView);
        this.galleryViewAdapter = new GalleryViewAdapter(this.galleryClasses);
        this.recyclerView.e0(new LinearLayoutManager());
        this.galleryViewAdapter.setOnItemClickListener(new a(0, this));
        this.recyclerView.d0(this.galleryViewAdapter);
    }
```

- as we can see it uses a `RecyclerView` to show all the painting as a "button"
- each one "button" opens, when the user clicks on it, another activity that shows the painting and the details of it is opened



==**Basically nothing relevant happens in this activity.** ==

So we can pass to the GuessingActivity.

GuessingActivity.java

The code contained in this class is very interesting because it is used to create an activity that allows an user to input the name of a painting:

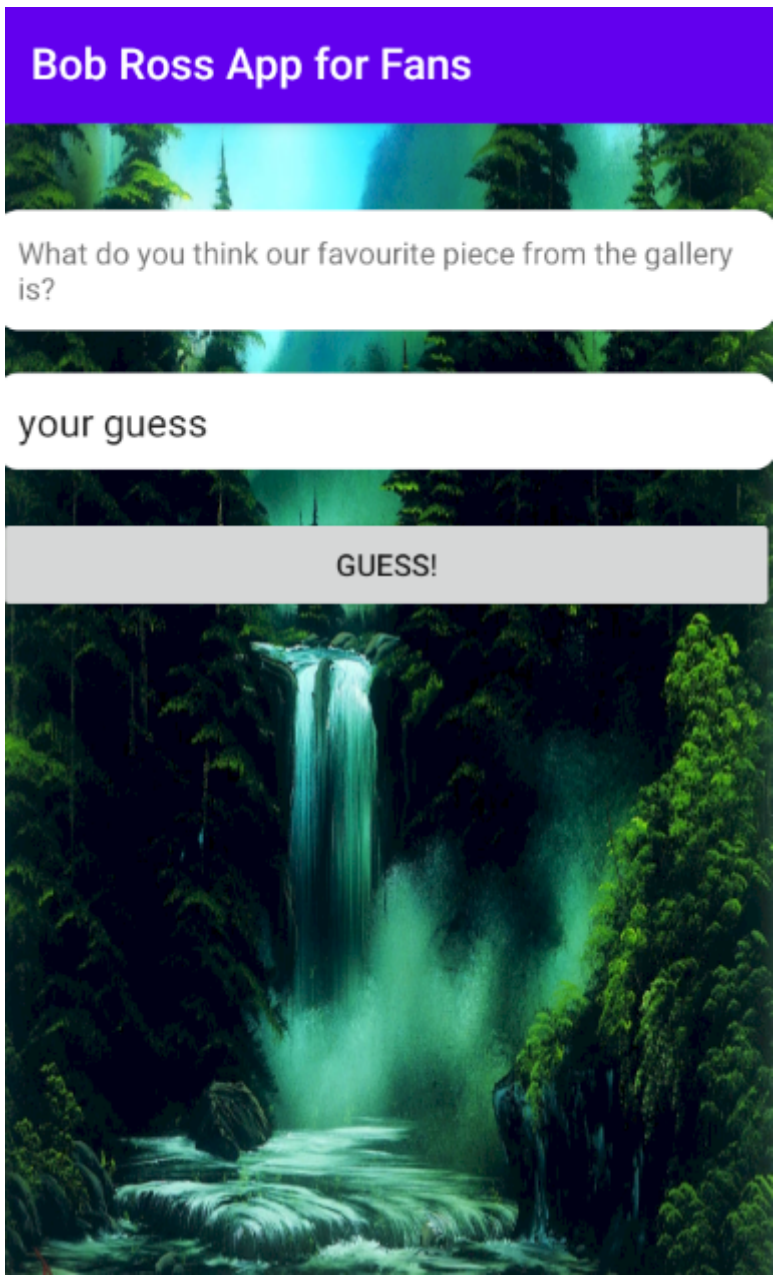
```
public class GuessingActivity extends AbstractActivityC0027n {
    private boolean verify(String str) {
        String str2;
        Log.d(MainActivity.TAG, "got guess: " + str);
        Class galleryContentByName = GalleryActivity.getGalleryContentByName(str);
        if (galleryContentByName == null) {
            return false;
        }
        try {
            str2 = (String) galleryContentByName.getMethod("getID", new Class[0]).invoke(null, new
Object[0]);
        } catch (IllegalAccessException | NoSuchMethodException | InvocationTargetException unused) {
            Log.e(MainActivity.TAG, "Yeah this should not happen");
            str2 = "";
        }
        return str2.equals("6E3E25FC3EBEE6CDF0B383683B261045D3DD52D5D3C106BD5F11FBCAD01C8285");
    }

    public void checkGuess(View view) {
        Toast.makeText(this, verify(((EditText) findViewById(R.id.input_guess)).getText().toString()) ?
"Correct!" : "That's not it!", 1).show();
    }

    /* JADX INFO: Access modifiers changed from: protected */
    @Override // androidx.fragment.app.j, androidx.activity.g, androidx.core.app.e, android.app.Activity
    public void onCreate(Bundle bundle) {
        super.onCreate(bundle);
        setContentView(R.layout.activity_guessing);
    }
}
```

- If the name of this painting is the `correct one` then a `Toast` shows to the user the label `Correct!`

> NOTE: A toast is a lightweight user interface element used to display brief, non-intrusive messages to the user.



From the description of the flag we know that we need the name of the correct painting to find the flag:

- `"What's our favourite picture in the Gallery? Find the correct name and get a flag!"`

For this reason the idea is to try to figure out which is the painting from which the flag is generated (the "fauvorite one") in order to generate this flag by ourself.

So looking at the code of the file GuessingActivity.java we can notice:

```
 public void checkGuess(View view) {
        Toast.makeText(this, verify(((EditText) findViewById(R.id.input_guess)).getText().toString()) ?
"Correct!" : "That's not it!", 1).show();
    }
```

- this snippet of code is responsible to check if the string insert by the user in the textfield is name of the "favourite painting"
- **to verify the condition, this method calls another method, ==**`verify`**, ==passing to it the string insert by the user**

Let's get a look at the verify method:

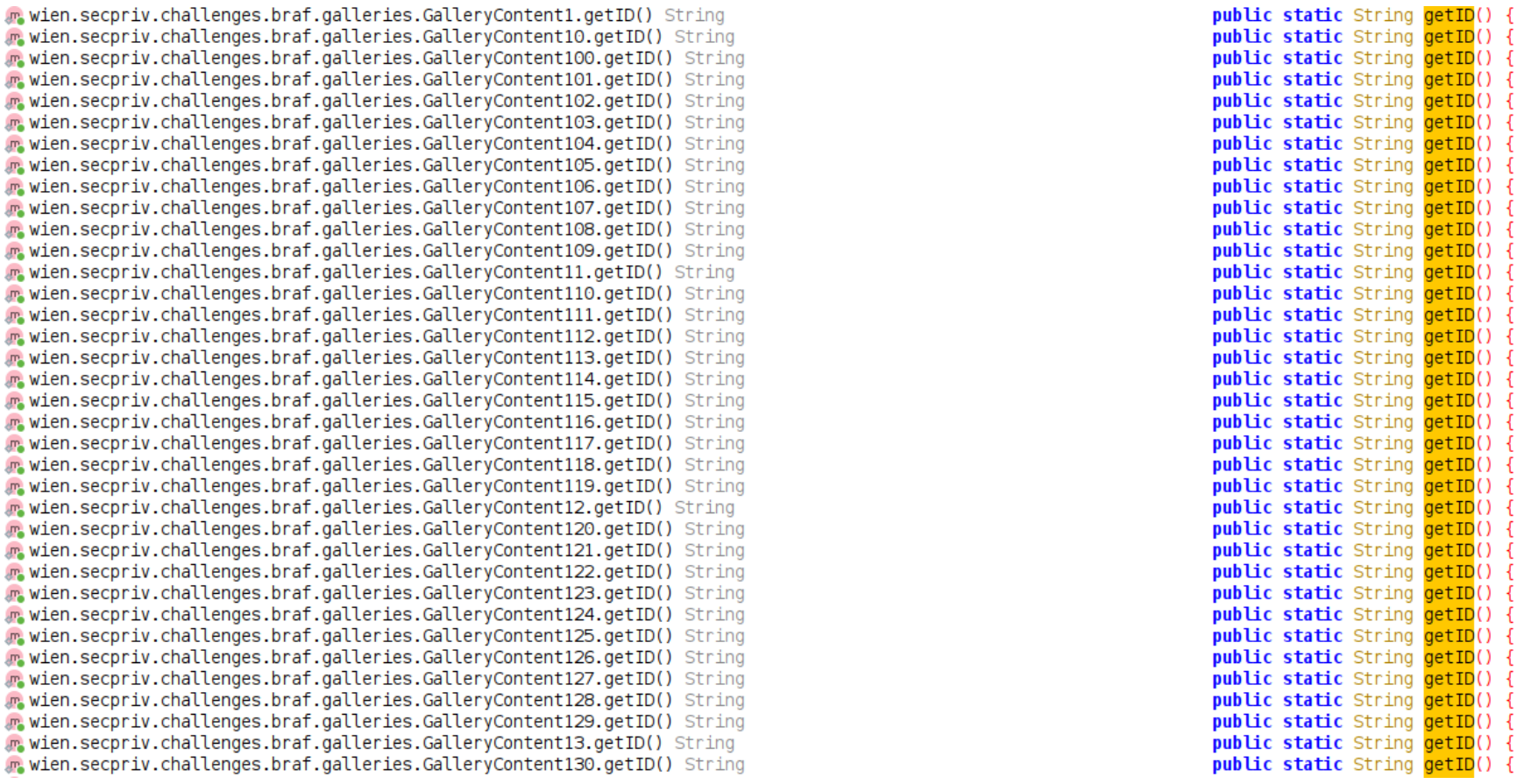
```
private boolean verify(String str) {
        String str2;
        Log.d(MainActivity.TAG, "got guess: " + str);
        Class galleryContentByName = GalleryActivity.getGalleryContentByName(str);
        if (galleryContentByName == null) {
            return false;
        }
        try {
            str2 = (String) galleryContentByName.getMethod("getID", new Class[0]).invoke(null, new
Object[0]);
        } catch (IllegalAccessException | NoSuchMethodException | InvocationTargetException unused) {
            Log.e(MainActivity.TAG, "Yeah this should not happen");
            str2 = "";
        }
        return str2.equals("6E3E25FC3EBEE6CDF0B383683B261045D3DD52D5D3C106BD5F11FBCAD01C8285");
    }
```

Analyzing it we can say that this method:

1. **Takes as input a string** `str` (the one used by the user in the textview)
2. It uses the string `str` **to obtain the Class object related to this string via a method of the GalleryActivity class**
    1. If the obtained object is null, it returns false
3. It uses reflection to **invoke the method** `getID` **on the obtained object and stores the result in a string** `str2`
    1. If an exception occurs during reflection, it logs an error message and sets str2 to an empty string
4. **It returns true if the value of str2 is equal to a "6E3E25FC3EBEE6CDF0B383683B261045D3DD52D5D3C106BD5F11FBCAD01C8285"**

So basically to understand which is the right painting we need to insert the name of a painting such that the return of the method getID() of the java class related to this painting is equal to "6E3E25FC3EBEE6CDF0B383683B261045D3DD52D5D3C106BD5F11FBCAD01C8285"

Now we can search in JADX "getID" and we can notice that each one painting java class has a specific java class in the source code and each one implements this method:



If we look at them we can see that each one implements this method calling the static method of the class GalleryContent, but in two different ways:

```
public static String getID() {
        return GalleryContent.computeHash(secret, name);
    }
```

```
public static String getID() {
        return GalleryContent.computeHash(name, secret);
    }
```

So, the method is the same but is called with a different order of the parameters by the different classes.

- we have to take this in mind for later

At this point we could take a look to the method GalleryContent.computeHash(String str, String str2):

```
    public static String computeHash(String str, String str2) {
        try {
            MessageDigest messageDigest = MessageDigest.getInstance("SHA-256");
            messageDigest.update(str.getBytes("UTF-8"));
            messageDigest.update(str2.getBytes("UTF-8"));
            return bytesToHex(messageDigest.digest());
        } catch (UnsupportedEncodingException | NoSuchAlgorithmException unused)         {
            Log.e(MainActivity.TAG, "Yeah this should not happen");
            return "";
        }
    }
```

- it generates a SHA-256 digest
- then `bytesToHex` method is used to convert bytes in hex representation

Painting name discovering

So now we have all the information to try to find out which is the painting that generates an ID equal to 6E3E25FC3EBEE6CDF0B383683B261045D3DD52D5D3C106BD5F11FBCAD01C8285 .

To do this we can try to offline brute force this digest using the same process used by the application source code.

For this purpose we can use a java class that automates the process using a txt file that contains for each class (representing a painting) :

- **for each line the name / secret pair or the secret / name pair depending on how they are used in the computeHash function**

Txt file content:

```
CleverButterfliesFeedSlowly QuickBadgersGatherExpectantly  # in this case (name,secret)
RapidShowersInterviewSupposedly BrownShopsSaveInstead  # in this case (secret,name)
PointedCommercialsExtractUneasily IdeologicalPhotographersHangUnfortunately
AloneDebtsDriftLovingly CharacteristicKnowledgesUpdateLast
AdvanceDamsInterviewDouble AvailableJamsRealizeThough
SecureGoodsMoveFurthermore WidePridesContrastBarely
IntensePropagandasAidCompletely ComparableBatteriesSimulateSubstantially
DestructiveRemediesPauseAbroad InsideChaosConsiderOff
SufficientAdviceImpressFrankly RepeatedTransmissionsComplainLow
PerfectEmergenciesStabilizeAstonishingly UltimateFarmingsGeneratePermanently
ExcessiveBooksDiscoverObviously IntermediateIntensitiesSupposeAround
SecondaryMusicalsCountAppropriately PreciseMeatsInterfereFreely
...
```

> Note: I upload in TUWEL the java file and the txt file.
> The txt file contains only 100 lines because i tested the java "script" each 25 lines and I found the result at line 81, so it had no sense to fill the file with the other values.

Java "script" used to find the correct result :

```java
import java.io.UnsupportedEncodingException;
import java.security.MessageDigest;
import java.security.NoSuchAlgorithmException;
import java.io.BufferedReader;
import java.io.FileReader;
import java.io.IOException;


/* loaded from: classes.dex */
public class DigestCreator  {


 public static String bytesToHex(byte[] bArr) {
    char[] cArr = new char[bArr.length * 2];
    char[] HEX_ARRAY = "0123456789ABCDEF".toCharArray();
    for (int i2 = 0; i2 < bArr.length; i2++) {
        int i3 = bArr[i2] & 255;
        int i4 = i2 * 2;
        char[] cArr2 = HEX_ARRAY;
        cArr[i4] = cArr2[i3 >>> 4];
        cArr[i4 + 1] = cArr2[i3 & 15];
    }
    return new String(cArr);
}

public static String computeHash(String str, String str2) {
    try {
        MessageDigest messageDigest = MessageDigest.getInstance("SHA-256");
        messageDigest.update(str.getBytes("UTF-8"));
        messageDigest.update(str2.getBytes("UTF-8"));
        return bytesToHex(messageDigest.digest());
    } catch (UnsupportedEncodingException | NoSuchAlgorithmException unused) {
        return "";
    }
```

```
    }


    public static void main(String[] args) {

          String fileName = "paintings.txt";

        try (BufferedReader br = new BufferedReader(new FileReader(fileName))) {
            String line;
            int i =0;
            while ((line = br.readLine()) != null) {

                String[] parts = line.split("\\s+");

                if (parts.length >= 2) {
                    String part1 = parts[0];
                    String part2 = parts[1];
                    i++;
                    if(computeHash(part1,
part2).equals("6E3E25FC3EBEE6CDF0B383683B261045D3DD52D5D3C106BD5F11FBCAD01C8285")){
                        System.out.println(part1+" " + part2);
                    }

                }
                //System.out.println(i);

            }


        } catch (IOException e) {
            e.printStackTrace();
        }
        //System.out.println(getFlag2());
    }


}
```

So what we do here is to:

1. read the file `paintings.txt` line by line
2. for each line we split on the space. So we get 2 strings from each line `part1` and `part2`
3. we check if the hash value computed with `part1` and `part2` is equal to "6E3E25FC3EBEE6CDF0B383683B261045D3DD52D5D3C106BD5F11FBCAD01C8285"
    1. if they are equal then we found the name of the correct painting

The function `computeHash(String str, String str2)` is the one contained in the application source code and performs the calculation of the SHA-256 digest.

The function `bytesToHex(byte[] bArr)` is the one contained in the source code of the application and converts an array of bytes into a string that is the hexadecimal representation of the array.

So we can compile and run this java file

1. `javac DigestCreator.java`
2. `java DigestCreator`

And we obtain:

- `CrowdedFilmMakersDisappointWhatever SustainableLeftsMaintainOriginally`

We search for both of them in JADX and we notice that the name of the painting is `SustainableLeftsMaintainOriginally`

```
public class GalleryContent171 extends GalleryContent {
    public static final String image = "painting171";
    public static final String name = "SustainableLeftsMaintainOriginally";
    protected static String secret = E.a(R.string.ConstantAnxietiesComputeMagnificently);
```

getting the flag

Now we have all the information to have access to the flag.

We know that the flag is generated by the painting `SustainableLeftsMaintainOriginally` so if we take a look at the source code of his java class we can see that there is the method getFlag2():

```
    public static String getFlag2() {
        String str;
        try {
            MessageDigest messageDigest = MessageDigest.getInstance("SHA-256");
            messageDigest.update("yesthisone".getBytes("UTF-8"));
            messageDigest.update(name.getBytes("UTF-8"));
            messageDigest.update(secret.getBytes("UTF-8"));
            str = GalleryContent.bytesToHex(messageDigest.digest());
        } catch (UnsupportedEncodingException | NoSuchAlgorithmException unused) {
            Log.e(MainActivity.TAG, "Yeah this should not happen");
            str = "";
        }
        return E.b(str, 0, 12, new StringBuilder("FLG_PT2{random_flag_is_"), "}");
    }
```

- also here we have a SHA-256 digest generation from `str` that is the name of the painting

If we take a look at the method `E.b` we can see:

```
 public static String b(String str, int i2, int i3, StringBuilder sb, String str2) {
        sb.append(str.substring(i2, i3));
        sb.append(str2);
        return sb.toString();
    }
```

So:

```
return E.b(str, 0, 12, new StringBuilder("FLG_PT2{random_flag_is_"), "}");
```

- is basically used to append `str` after `FLG_PT2{random_flag_is_` and before `}`

So what we can do here is to extract the method getFlag2() and compute by ourself the flag:

```
public static String getFlag2() {

        // this is written after getting the correct painting
        String found_name = "SustainableLeftsMaintainOriginally";
        String found_secret = "CrowdedFilmMakersDisappointWhatever";
        String str;
        try {
            MessageDigest messageDigest = MessageDigest.getInstance("SHA-256");
            messageDigest.update("yesthisone".getBytes("UTF-8"));
            messageDigest.update(found_name.getBytes("UTF-8"));
            messageDigest.update(found_secret.getBytes("UTF-8"));
            str = bytesToHex(messageDigest.digest());



        } catch (UnsupportedEncodingException | NoSuchAlgorithmException unused) {
            str = "";
        }


        // I construct it in this way because the flag must be of the form: FLG_PT2{random_flag_is_**12
chars returned by the digest in this method**}
        return "FLG_PT2{random_flag_is_" +str.toLowerCase().substring(0,12)+ "}";
    }
```

The flag is: `FLG_PT2{random_flag_is_0f5943ffc0fb}`

Discussion of the vulnerabilities

As we can see in this part of the challenge we have a vulnerability involving Exposure of Sensitive Information.

In fact we can see that in the source code the hardcoding of the string "6E3E25FC3EBEE6CDF0B3683B261045D3DD52D5D3C106BD5F11FBCAD01C8285" which represents the ID of the framework

used to access the flag is done.

A possible solution to this problem might be to store the sensitive data not in the source code but in configuration files.

Another possible solution might be to initiate an interaction with an external server to perform the verification and thus prevent it from being exposed in the source code.

Also exposed in plain text is the method used to calculate a framework ID, which uses SHA-256.

The solution in this case could be to use code obfuscation.

Challenge part 3

Professor Hahnenkamp Klaus gave me some advice when I was stuck.

Introduction

We use also here JADX-GUI to decompile the apk and to have access to the source code.

We know by the description that the target for the third flag is for sure the RiddleActivity:

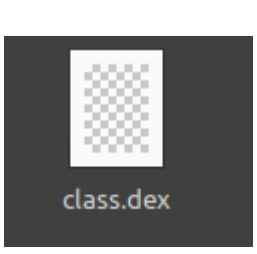
```
Riddle me this: What's the flag?
```

Static analysis

So we start on analyzing the source code of RiddleActivity.java and we notice the `onCreate` method:

```
public void onCreate(Bundle bundle) {
        super.onCreate(bundle);
        setContentView(R.layout.activity_riddle);
        System.out.println(H.a.d(getApplicationContext()));
        setValue("");
        flagObserver();
        File filesDir = getApplicationContext().getFilesDir();
        ClassLoader classLoader = getApplicationContext().getClassLoader();
        Context applicationContext = getApplicationContext();
        B b2 = new B(new A());
        D d = new D();
        d.g("https://class.sas.hackthe.space/class");
        new j(b2, d.a(), false).f(new e(this, filesDir, applicationContext, classLoader));
    }
```

One interesting thing here is the presence of an URL, and if we copy and paste it in the browser we are able to download a file called Class.dex:

- 

So we can take note of that.

At this point we continue to analyze the source code and we can notice that line below is only used to check if the url is in the correct format or if it needs to be normalized, so not interesting at all:

```
d.g("https://class.sas.hackthe.space/class");
```

In, fact the function `d.g(String str)`:

```
public final void g(String str) {
        String substring;
        String str2;
        S.b.f(str, "url");
        if (!W.g.C(str, "ws:", true)) {
            if (W.g.C(str, "wss:", true)) {
                substring = str.substring(4);
                S.b.e(substring, "this as java.lang.String).substring(startIndex)");
                str2 = "https:";
```

```
            }
            S.b.f(str, "<this>");
            w wVar = new w();
            wVar.f(null, str);
            this.f152a = wVar.a();
        }
        substring = str.substring(3);
        S.b.e(substring, "this as java.lang.String).substring(startIndex)");
        str2 = "http:";
        str = str2.concat(substring);
        S.b.f(str, "<this>");
        w wVar2 = new w();
        wVar2.f(null, str);
        this.f152a = wVar2.a();
    }
```

- Basically parses a URL string, changes the protocol (from "ws:" to "https:" or from "http:" to "https:") if necessary, and creates a new object using the new URL

So we can focus on the line:

```
new j(b2, d.a(), false).f(new e(this, filesDir, applicationContext, classLoader)
```

> Note: we will refer to this line in the following text.

In this case the obfuscation doesn't help us in the analyzing process but reading the source code we can extract useful information.

j java class analyzing

In fact, if we open the class `j` we can read some strings that help us to understand that this class is used to manage a connection via web sockets.

For example in the constructor we can read `"client"`, `"originalRequest"`:

```
public j(B b2, E e2, boolean z2) {
        S.b.f(b2, "client");
        S.b.f(e2, "originalRequest");
        this.f2179a = b2;
        this.f2180b = e2;
        this.f2181c = z2;
        this.d = b2.g().d();
        X.r rVar = b2.l().f308a;
        S.b.f(rVar, "$this_asFactory");
        this.f2182e = rVar;
        i iVar = new i(this);
        iVar.g(b2.d(), TimeUnit.MILLISECONDS);
        this.f2183f = iVar;
        this.f2184g = new AtomicBoolean();
        this.f2192o = true;
    }
```

But also in another method called `b` we can read `"web socket" : "call"`:

```
public static final String b(j jVar) {
        StringBuilder sb = new StringBuilder();
        sb.append(jVar.f2193p ? "canceled " : "");
        sb.append(jVar.f2181c ? "web socket" : "call");
        sb.append(" to ");
        sb.append(jVar.t());
        return sb.toString();
    }
```

B class analyzing

We can also see (always from the line seen before) that an object `b2` is passed as parameter for the `j` class contructor.

If we take a look on the B class then we can be sure that a web socket connection is established:

```
List list = this.f125c;
S.b.d(list, "null cannot be cast to non-null type kotlin.collections.List<okhttp3.Interceptor?>");
```

- this lines **contained in the constructor allows us to understand that okhttp3 is used to perform http requests**

> Note: I reported only this lines because the method has an high number of lines.
> It was interesting only to see that `okhttp3` is used.

So we can be pretty sure that the application downloads the file Class.dex from the link to do something. It is a dex file so maybe it is used to load code dynamically.

Now we can continue to analyze the line:

```
new j(b2, d.a(), false).f(new e(this, filesDir, applicationContext, classLoader)
```

e class analyzing

We have to understand what this line does:

```
new e(this, filesDir, applicationContext, classLoader)
```

So we take a look on the `e` java class, focusing on the method `b`:

```
public final void b(H h2) {
        String str;
        boolean G2 = h2.G();
        RiddleActivity riddleActivity = this.f2988a;
        if (G2) {
            File file = this.f2989b;
            File file2 = new File(file, "data.jar");
            FileOutputStream fileOutputStream = new FileOutputStream(file2);
            if (h2.y() != null) {
                String d = H.a.d(this.f2990c);
                J y2 = h2.y();
                long y3 = y2.y();
                if (y3 > 2147483647L) {
                    throw new IOException("Cannot buffer entire body for content length: " + y3);
                }
                h A2 = y2.A();
                try {
                    byte[] g2 = A2.g();
                    H.a.k(A2, null);
                    int length = g2.length;
                    if (y3 != -1 && y3 != length) {
                        throw new IOException("Content-Length (" + y3 + ") and stream length (" + length
+ ") disagree");
                    }
                    try {
                        SecretKeySpec secretKeySpec = new SecretKeySpec(d.getBytes(), "AES");
                        Cipher cipher = Cipher.getInstance("AES/CBC/PKCS5Padding");
                        cipher.init(2, secretKeySpec, new IvParameterSpec(new byte[16]));
                        fileOutputStream.write(cipher.doFinal(g2));
                    } catch (MediaCodec.CryptoException | IOException |
InvalidAlgorithmParameterException | InvalidKeyException | NoSuchAlgorithmException | BadPaddingException
| IllegalBlockSizeException | NoSuchPaddingException e2) {
                        e2.printStackTrace();
                        riddleActivity.postToastMessage("Decryption error");
                    }
                    try {
                        Class<?> loadClass = new DexClassLoader(file2.getAbsolutePath(),
file.getAbsolutePath(), null, this.d).loadClass("SetFlag");
                        file2.delete();
                        loadClass.getMethod("setFlag", Class.class).invoke(null, RiddleActivity.class);
                        return;
                    } catch (Exception e3) {
                        e3.printStackTrace();
                        return;
                    }
                } catch (Throwable th) {
                    try {
```

```
                    throw th;
                } catch (Throwable th2) {
                    H.a.k(A2, th);
                    throw th2;
                }
            }
        }
        str = "Error empty response";
    } else {
        str = "Error loading data";
    }
    riddleActivity.postToastMessage(str);
}
```

In summary this method handles the loading and decoding of data from the URL and so the file `Class.dex` and loads from this file a class in a dynamical way.

Here we have some interesting sections:

```
File file = this.f2989b;
File file2 = new File(file, "data.jar");
FileOutputStream fileOutputStream = new FileOutputStream(file2);
```

- if we follow the code, "file" is simply initialized as getApplicationContext().getFilesDir(); so basically it contains the path used by the application to store on the device the files and the data (generally `/data/data/package_name`)
- `fileOutputStream` is simply used to have the possibility to write on the file `data.jar` created with the previous line

```
byte[] g2 = A2.g();
```

- g2 is basically the array of bytes that represents the body of the https request executed. So in this case it represents the file `Class.dex`

```
SecretKeySpec secretKeySpec = new SecretKeySpec(d.getBytes(), "AES");
Cipher cipher = Cipher.getInstance("AES/CBC/PKCS5Padding");
cipher.init(2, secretKeySpec, new IvParameterSpec(new byte[16]));
fileOutputStream.write(cipher.doFinal(g2));
```

- this snippet of code is used to decrypt the `Class.dex` file and write the decrypted content into `data.jar` (`fileOutputStream.write(cipher.doFinal(g2));`)
- it is decrypted using `AES` with the using of `Cipher Block Chaining` and `PKCS5` padding
- it uses an initialization vector of 16 bytes but empty
- it uses as decryption key `d.getBytes()`

```
Class<?> loadClass = new DexClassLoader(file2.getAbsolutePath(), file.getAbsolutePath(), null,
this.d).loadClass("SetFlag");
file2.delete();
loadClass.getMethod("setFlag", Class.class).invoke(null, RiddleActivity.class);
return;
```

- it loads in a dynamic way the decrypted file
- it loads the class `SetFlag` contained in the decrypted file
- deletes the decrypted file
- invokes the `setFlag` method, to show a message in the RiddleActivity

Decryption key creation

Now we can see how the decryption key is generated:

```
public static String d(Context context) {
    try {
        StringBuilder sb = new StringBuilder();
        sb.append((String) Context.class.getMethod("getPackageName", new Class[0]).invoke(context, new
Object[0]));

        Class cls = Integer.TYPE;

        sb.append((String) String.class.getMethod("substring", cls, cls).invoke((String)
```

```
Context.class.getMethod("getPackageName", new Class[0]).invoke(context, new Object[0]), 0, 4));

        return sb.toString();
        } catch (IllegalAccessException | NoSuchMethodException | InvocationTargetException unused) {
            return "";
        }
    }
```

1. it creates a StringBuilder `sb`
2. adds to `sb` basically the package name of the application
    1. we can see in the `AndroidManifest.xml` that it is `wien.secpriv.challenges.braf`
3. adds to `sb` only the first 4 chars of the package name
4. returns the constructed string

So basically the decryption key is: `wien.secpriv.challenges.brafwien`

Class.dex decryption via java "script"

With all the information obtained before we can try to download and decrypt by ourself the `Class.dex` file in order to analyze it and to see if it contains useful information.

In order to do that we can write a java "script" that:

1. **performs an HTTP request to the url contained in the source code**
2. **inserts the body of the response into an array of bytes**
3. **uses the same approach used by the application source code to decrypt it and store it into a file called** `data.jar`

> NOTE: the key is constructed directly from the package name which can be found into `AndroidManifest.xml`

Java file content:

```
import java.io.*;
import java.net.URI;
import java.net.http.HttpClient;
import java.net.http.HttpRequest;
import java.net.http.HttpResponse;
import java.nio.charset.StandardCharsets;

import javax.crypto.*;
import javax.crypto.spec.*;

public class Decrypt {

    public static String get_decrypt_key() {

        /*
         * We know which is the package name -> we can see it from the
         * AndroidManifest.xml
         * So we can construct directly the encryption key used by the code to decode
         * the dex file
         *
         * however the key will be  wien.secpriv.challenges.brafwien
         */
        try {
            StringBuilder sb = new StringBuilder();
            sb.append("wien.secpriv.challenges.braf");
            Class cls = Integer.TYPE;
            sb.append((String) String.class.getMethod("substring", cls,
cls).invoke("wien.secpriv.challenges.braf", 0,
                    4));
            return sb.toString();
        } catch (Exception e) {
            return "";
        }
    }

    public static void main(String[] args) {
        String decryption_string = get_decrypt_key();
        String dex_file_do_decrypt = "class.dex";

        /* I can try to download directly the file and decrypt it  */
```

```
        HttpClient httpClient = HttpClient.newBuilder().build();
        HttpRequest httpRequest = HttpRequest.newBuilder()
                .uri(URI.create("https://class.sas.hackthe.space/class"))
                .GET()
                .build();

        try {
            HttpResponse<byte[]> response = httpClient.send(httpRequest,
HttpResponse.BodyHandlers.ofByteArray());
            byte[] responseBody = response.body();

            try (FileOutputStream fileOutputStream = new FileOutputStream("data.jar")) {

            SecretKeySpec secretKeySpec = new SecretKeySpec(decryption_string.getBytes(), "AES");

            Cipher cipher = Cipher.getInstance("AES/CBC/PKCS5Padding");

            cipher.init(Cipher.DECRYPT_MODE, secretKeySpec, new IvParameterSpec(new byte[16]));

            fileOutputStream.write(cipher.doFinal(responseBody));

            } catch (Exception e) {
                e.printStackTrace();
            }
        } catch (Exception e) {
            System.err.println(e.getMessage());
        }
    }
}
```

What we do here is:

1. we construct the decryption key from the package name with `String decryption_string = get_decrypt_key();`
2. we perform an HTTP request to `https://class.sas.hackthe.space/class`
3. we insert the body of the response, that contains the file `Class.dex` into an array of bytes `byte[] responseBody = response.body();`
4. we use the same procedure used by the application code to decrypt the file and we write the decrypted file into the file `data.jar`

> NOTE: I upload it in TUWEL

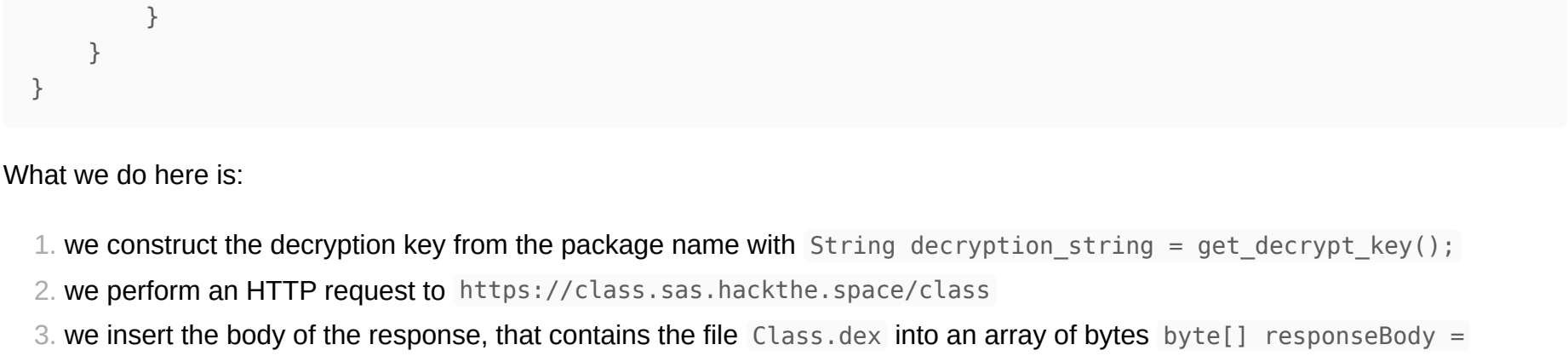
Compile and run it:

- `javac Decrypt.java`
- `java Decrypt`

Get the flag

At this point we have the decrypted file and we could take a look to see if it contains useful information.

We open it with a text editor and we can read:



So the flag for this part of the challenge is `FLG_PT3{I_WAS_HERE_ALL_THE_TIME}`

Discussion of the vulnerabilities

In this case there are 2 vulnerabilities.

The first is based on public exposure of the `Class.dex` file that can be downloaded by making an unauthenticated request to the URL `https://class.sas.hackthe.space/class`.
So a first line of defense could be to use a server that requires authentication to access the resource.

The second vulnerability relies on exposing the key used to decrypt the `Class.dex` file, which turns out to be `wien.secpriv.challenges.brafwien`.
It can be easily obtained by studying the code used to generate it. A line of defense here could be applied by storing the key in Android's Trusty TEE and then not exposing it.

Challenge Part 4

Professor Hahnenkamp Klaus gave me some advice when I wa stuck.

Introduction

We know by the description that the last flag is hidden somewhere in the Quotes section of the application:

```
Quotes, so many quotes! But where is the flag?
```

So we can start to analyze the code of the activities:

- QuoteActivity
- QuoteService

Static analysis

QuoteActivity analysis

Looking at the method onCreate of this activity, we can notice that it is based on the using of a WebView.

A WebView basically allows the usage of web content directly in the application and so allows the processing of web content such as HTML files, javascript code etc...

```
public void onCreate(Bundle bundle) {
        super.onCreate(bundle);
        setContentView(R.layout.activity_quote);
        Button button = (Button) findViewById(R.id.getQuoteButton);
        this.getQuoteButton = button;
        button.setEnabled(false);
        jsonObserver();
        QuoteObserver.getInstance().addObserver(this);
        getNewQuote();
        WebView webView = (WebView) findViewById(R.id.contentWebView);
        this.webView = webView;
        webView.getSettings().setJavaScriptEnabled(true);
        this.webView.getSettings().setAllowFileAccess(true);
        this.webView.getSettings().setAllowFileAccessFromFileURLs(true);
        this.webView.getSettings().setCacheMode(1);
        this.webView.setWebViewClient(new WebViewClient());
        this.webView.setWebChromeClient(new WebChromeClient());
        this.webView.addJavascriptInterface(this, "userValidation");
        this.webView.loadUrl("file:///android_asset/index.html");
    }
```

From this code we can notice something interesting:

- `webView.getSettings().setJavaScriptEnabled(true);` **enables the usage of javascript in the webView**
- `this.webView.loadUrl("file:///android_asset/index.html");` **is used to load the file `index.html` contained in the `asset` folder of the application**
- `this.webView.addJavascriptInterface(this, "userValidation");` **adds a javascript interface into the WebView**
  - a javascript interface allows the possibility to the javascript code processed by the WebView to call public methods of this class (QuoteActivity) that have the annotation `@JavascriptInterface`

Another thing really interesting here is the method:

```
    @JavascriptInterface
    public String validateUsername(String str) {
        return
            nativeUsernameValidation(Base64.getDecoder().decode(str.getBytes()));
    }
```

- it can be accessed by the javascript processed by the WebView

- it calls the method `nativeUsernameValidation`

So we can get a look to the method `nativeUsernameValidation`:

```
public native String nativeUsernameValidation(byte[] bArr);
```

- **it is a native method so basically is loaded from an external library written in C/C++**

In fact we can also notice that the class QuoteActivity loads a native library from the application libraries contained into the folder `/lib:

```
static {
    System.loadLibrary("safetynetchallengepart");
}
```

We take note of that, because it will be useful later.

index.html analysis

At this point we can continue our investigation and we can get a look to the file `index.html` to see what it contains:

```
<html>
<body>
<div id="username">
</div>

<div id="quote">
</div>
<script>
    onmessage = function(event) {
        const jsonData = JSON.parse(event.data);
        if ('error' in jsonData) {
            document.getElementById('username').innerHTML = `ERROR`;
            document.getElementById('quote').innerHTML = `${jsonData.quote}`;
        } else if ('quote' in jsonData) {
            //Add validateUsername(jsonData.username) to the next version. Unfortunately, there was no
time to implement it...
                document.getElementById('username').innerHTML = `${jsonData.username}, your personal
quote is:`;
            document.getElementById('quote').innerHTML = `${jsonData.quote}`;
        } else {
            document.getElementById('username').innerHTML = "";
            document.getElementById('quote').innerHTML = "";
        }

    }
</script>
</body>
</html>
```

The comment `//Add validateUsername(jsonData.username) to the next version. Unfortunately, there was no time to implement it...` gives us an useful information:

- **The username field is not validated at all.**

The line `document.getElementById('username').innerHTML = '${jsonData.username}, your personal quote is:';` basically changes the innerHTML of the element with the id = username.

So, **the fact that the username is not validated opens the door to an** `XSS attack (Cross Side Scripting)`.

- This because we can inject javascript code, by the fact that the content of username is not validated and so processed and executed by the WebView.

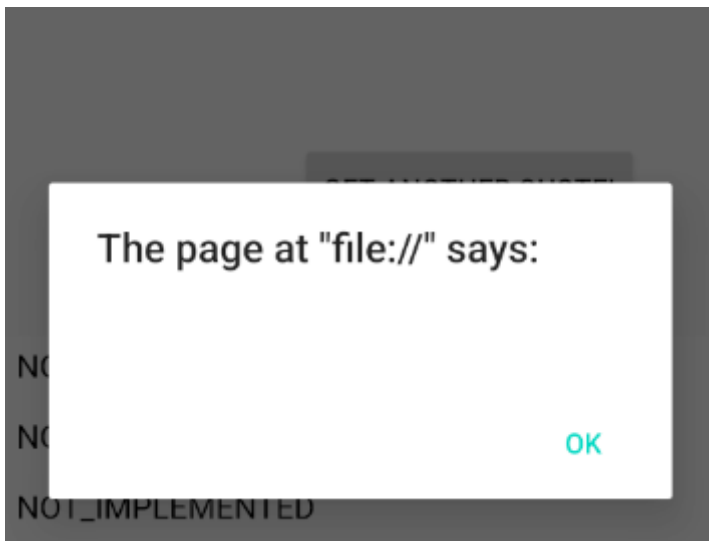
We know that `.innerHTML` doesn't allow the injection of tags like `<script>` so we need to use the tag `<img>` and try to trigger an error with the event `onerror`.

XSS attack contruction

We can easily start trying:

```
<img src=1 onerror="alert()" >
```

And it works:



We have seen that the QuoteActivity contains a method `validateUsername` that can be invoked by the javascript processed by the WebView, so we could try to execute through the `XSS` attack.

Another part of the puzzle. We take note about that.

safetynetchallengepart.so library analysis

Now we can focus on this library to see what the method `nativeUsernameValidation` does.

We will use in this purpose the tool Ghidra.

Ghidra is basically a suite software for reverse engineering (SRE) developed by the NSA Research Directorate.

nativeUsernameValidation function analysis

We load in it the library and we start to analyze the code of the function `nativeUsernameValidation` :

```
/* nativeUsernameValidation(unsigned char const*) */
char * nativeUsernameValidation(uchar *param_1)
{
  void *__s2;
  int iVar1;
  char *pcVar2;

  __s2 = (void *)o_54832dacb65f7c6e96c8b10f0795d249();
  iVar1 = memcmp(param_1,__s2,0xe);
  if (iVar1 == 0) {
    pcVar2 = (char *)getFlag();
  }
  else {
    pcVar2 = "NOT_IMPLEMENTED";
  }
  return pcVar2;
}
```

In this code:

- **`iVar2 = memcmp(param_1,__s2,0xe);` basically ckecks if the first 14 (0xe) chars (or blocks of memory) of `param_1` and `pcVar1` are equal**
- **if they are equal and so `iVar1==0` then the code will return the content of the function `getFlag()`**

Maybe we can try to obtain what the function `o_54832dacb65f7c6e96c8b10f0795d249()` returns in order to reuse and force the application to print the flag.

So we can start to create an XSS that triggers the function `validateUsername(String str)` of QuoteActivity that invokes `nativeUsernameValidation(Base64.getDecoder().decode(str.getBytes()))`

XSS to trigger validateUsername(String str)

We can construct an < img > tag to for the execution of this method, showing the return value of the method at all in this way:

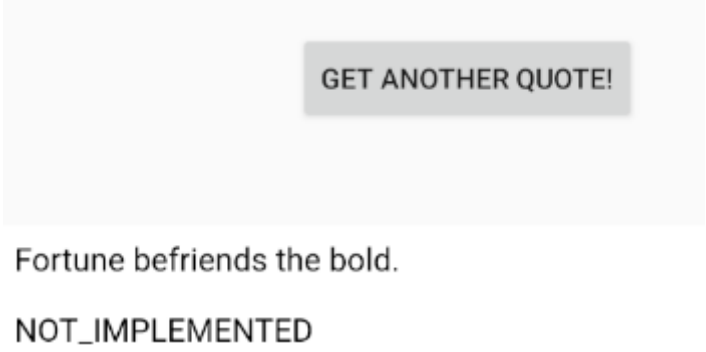
```
<img src=1 onerror="var par = document.createElement('p'); var username = 'HELLO'; var base64 =
btoa(username); par.textContent = window.userValidation.validateUsername(base64);
```

```
document.body.appendChild(par);" >
```

In this code:

- `var par = document.createElement('p');` **-> creates a < p > html element**
- `var username = 'HELLO';` **-> is the username we want to inject**
- `var base64 = btoa(username);` **-> is used to convert the username string in base64. This beacuse the method** `validateUsername(String str)` **passes to the method** `nativeUsernameValidation(byte[] arr)` **this** `Base64.getDecoder().decode(str.getBytes())`
- `par.textContent = window.userValidation.validateUsername(base64);` **-> is used to invoke the method** `validateUsername(String str)` **using the object** `userValidation` **that refers to the Javascript Interface created for the WebView**
- `document.body.appendChild(par);` **-> appends to the HTML the result of the called function, so shows the return value of the native method**

In fact if we inject it we can see:



- NOT_IMPLEMENTED is the result of nativeUsernameValidation, this because we didn't use the right username

So now we want to trace the methods and try to see if we are able to intercept what the method `o_54832dacb65f7c6e96c8b10f0795d249()` of the native library returns.

Basically this method will return the correct "username" to use to force the execution of getFlag().

Dynamic Analysis

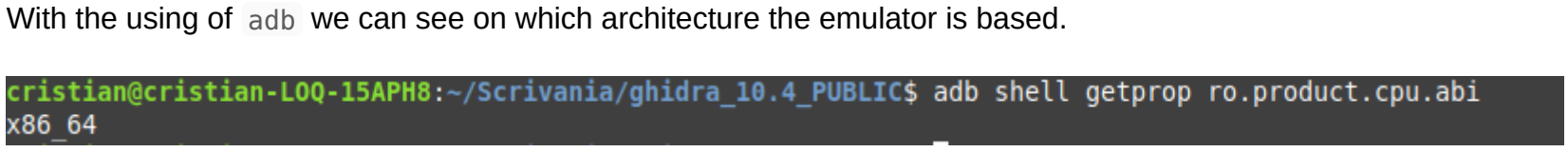
o_54832dacb65f7c6e96c8b10f0795d249() tracing with Frida

Frida is a toolkit used to inject your own scripts into black box processes, hook any function, spy on crypto APIs or trace private application code.

- so we can use it to trace the execution of the function o_54832dacb65f7c6e96c8b10f0795d249()

Frida setup

First of all we need to run on the device, in this case the emulator, a frida server.

1. With the using of `adb` we can see on which architecture the emulator is based.
   - 

So we download the frida server for x86_64.

2. Now we have to make sure that adb is run as root on the device, so we can easily run the command `adb root`
3. From this point we can easily follow the steps:
   1. Copy the frida server file into the Android phone’s tmp directory using
      1. `adb push frida_server_name /data/local/tmp/`
   2. Change the permission of the frida-server file.
      1. `adb shell "chmod 755 /data/local/tmp/frida_server_name"`
   3. Run frida server on the device
      1. `adb shell "/data/local/tmp/frida_server_name`

Frida javascript to hook the function o_54832dacb65f7c6e96c8b10f0795d249()

The idea of frida is to connect to the process of the application executed on the emulator and perform some actions using a javascript code.

So first of all we need a javascript code to hook the function we want to trace:

```
Interceptor.attach(Module.getExportByName("libsafetynetchallengepart.so",
"_Z34o_54832dacb65f7c6e96c8b10f0795d249v"), {
    onEnter: function (args) {
        console.log("nativeUsernameValidation");

        var param_1 = args[0];
        var username = Memory.readCString(param_1);
        console.log(username);
    },
    onLeave: function (retval) {

        var resultBytes = Memory.readByteArray(retval,50);
        console.log( resultBytes);
    }
});
```

Here:

- `Interceptor.attach(Module.getExportByName("libsafetynetchallengepart.so", "_Z34o_54832dacb65f7c6e96c8b10f0795d249v"),` is used to find the address of the function `o_54832dacb65f7c6e96c8b10f0795d249` in the library `libsafetynetchallengepart.so`
- `onEnter` is used to execute code when the application enters in the function we have hooked
- `onLeave` is used to execute code when we are leaving the function.

> NOTE we use `_Z34o_54832dacb65f7c6e96c8b10f0795d249v` instead of `o_54832dacb65f7c6e96c8b10f0795d249` because frida needs the mangled name of the function, so basically the name assigned by the compiler.
>
> We can find it using nm that is used to enumerate the symbols contained in the library `nm -D libsafetynetchallengepart.so | grep o_54`
>
> ```
> cristian@cristian-LOQ-15APH8:~/Scrivania/challenges/braf.2024S/lib/x86_64$ nm -D libsafetynetchallengepart.so | grep o_54
> 0001a780 T _Z34o_54832dacb65f7c6e96c8b10f0795d249v
> ```

The block:

```
onEnter: function (args) {
        console.log("nativeUsernameValidation");

        var param_1 = args[0];
        var username = Memory.readCString(param_1);
        console.log(username);
    },
```

- is just used to print the username we pass to the function when we enter in the execution of that function

But, the most important block of code here is:

```
onLeave: function (retval) {

        var resultBytes = Memory.readByteArray(retval,50);
        console.log( resultBytes);
    }
```

It takes the return value of the function we have hooked and prints it in the console.

The second thing to do is to find the name of the application we want to trace on the device.
So now what we need to do is to find the name of the application that is running on the device:

- **`frida-ps -U`**

```
 PID  Name
----  ------------------------------------------------------------------------------------
5321  Bob Ross App for Fans
4975  Calendar
5130  Messaging
4294  SIM Toolkit
```

Frida execution

So now we have everything to trace the application and in fact we can run:

`frida -U -l frida.js Bob\ Ross\ App\ for\ Fan`

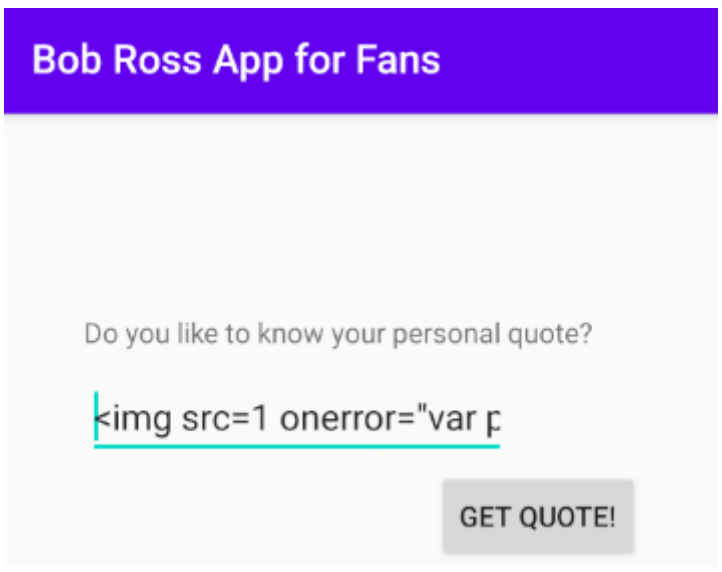
- **frida.js is the javascript seen before**

At this time frida is waiting for us, because we need to force the execution of the function we want to trace.

So we can use the `XSS`:

```
<img src=1 onerror="var par = document.createElement('p'); var username = 'HELLO'; var base64 =
btoa(username); par.textContent = window.userValidation.validateUsername(base64);
document.body.appendChild(par);" >
```

- directly in the user input of the application running on the emulator:



The execution of the `xss` triggers the execution of the method `nativeUsernameValidation` that triggers internally the execution of the method `o_54832dacb65f7c6e96c8b10f0795d249`

This execution is intercepted by our frida javascript code:

```
   . . . .
   . . . .   Connected to Android Emulator 5554 (id=emulator-5554)

[Android Emulator 5554::Bob Ross App for Fans ]-> nativeUsernameValidation
HELLOq
           0  1  2  3  4  5  6  7  8  9  A  B  C  D  E  F  0123456789ABCDEF
00000000  5a 5f 55 77 29 71 62 2a 2b 67 58 67 74 39 00 00  Z_Uw)qb*+gXgt9..
00000010  01 b0 00 00 00 00 05 dc 00 00 00 00 00 00 00 00  ................
00000020  00 00 00 00 00 00 00 00 00 00 00 00 be 00 00 00  ................
00000030  01 a0                                            ..
```

We cna see that the return of the function `o_54832dacb65f7c6e96c8b10f0795d249` is `Z_Uw)qb*+gXgt9`

So the correct "Username" is then: `Z_Uw)qb*+gXgt9`

Getting the flag

The game is done because we have the correct username to bypass the check contained in `nativeUsernameValidation` function.

So we change our `xss` into:

```
<img src=1 onerror="var par = document.createElement('p'); var username = 'Z_Uw)qb*+gXgt9'; var base64 =
btoa(username); par.textContent = window.userValidation.validateUsername(base64);
document.body.appendChild(par);" >
```

- here we have `ar username = 'Z_Uw)qb*+gXgt9';`

If we use it we obtain on the application:

FLG_PT4{FINALLY_YOU_FOUND_ME}

The flag for this part is `FLG_PT4{FINALLY_YOU_FOUND_ME}`

Approach 2

I tried also another approach to take the correct username, that is really interesting.

In this one we analyze the native code and we try to reproduce the function `o_54832dacb65f7c6e96c8b10f0795d249` using a C "script".

We know from Ghidra that the code of `o_54832dacb65f7c6e96c8b10f0795d249` is:

```
void o_54832dacb65f7c6e96c8b10f0795d249(void)

{
  undefined8 uVar1;
  byte bVar2;
  byte bVar3;
  byte bVar4;
  int iVar5;
  undefined8 *puVar6;
  undefined8 *puVar7;
  byte *pbVar8;

  iVar5 = getSignature();
  puVar6 = (undefined8 *)operator.new[](0xe);
  uVar1 = *(undefined8 *)(iVar5 + 0x236);
  *puVar6 = *(undefined8 *)(iVar5 + 0x230);
  *(undefined8 *)((int)puVar6 + 6) = uVar1;
  puVar7 = (undefined8 *)operator.new[](0xe);
  uVar1 = *(undefined8 *)(iVar5 + 0x389);
  *puVar7 = *(undefined8 *)(iVar5 + 899);
  *(undefined8 *)((int)puVar7 + 6) = uVar1;
  pbVar8 = (byte *)malloc(0xe);
  bVar2 = *(byte *)((int)puVar6 + 1);
  *pbVar8 = *(byte *)puVar6 ^ *(byte *)puVar7 ^ 0xd;
  bVar3 = *(byte *)((int)puVar6 + 2);
  pbVar8[1] = bVar2 ^ *(byte *)((int)puVar7 + 1) ^ 0x57;
  bVar2 = *(byte *)((int)puVar6 + 3);
  pbVar8[2] = bVar3 ^ *(byte *)((int)puVar7 + 2) ^ 7;
  bVar3 = *(byte *)((int)puVar6 + 4);
  pbVar8[3] = bVar2 ^ *(byte *)((int)puVar7 + 3) ^ 0x23;
  bVar2 = *(byte *)((int)puVar6 + 5);
  pbVar8[4] = bVar3 ^ *(byte *)((int)puVar7 + 4) ^ 0x29;
  bVar3 = *(byte *)((int)puVar6 + 6);
  pbVar8[5] = bVar2 ^ *(byte *)((int)puVar7 + 5) ^ 0x21;
  bVar2 = *(byte *)((int)puVar6 + 7);
  pbVar8[6] = bVar3 ^ *(byte *)((int)puVar7 + 6) ^ 0x36;
  bVar3 = *(byte *)(puVar6 + 1);
  bVar4 = *(byte *)(puVar7 + 1);
  pbVar8[7] = bVar2 ^ *(byte *)((int)puVar7 + 7) ^ 0x28;
  pbVar8[8] = bVar3 ^ bVar4 ^ 0x7b;
  pbVar8[9] = *(byte *)((int)puVar6 + 9) ^ *(byte *)((int)puVar7 + 9) ^ 0x6b;
  pbVar8[10] = *(byte *)((int)puVar6 + 10) ^ *(byte *)((int)puVar7 + 10) ^ 0x5b;
  pbVar8[0xb] = *(byte *)((int)puVar6 + 0xb) ^ *(byte *)((int)puVar7 + 0xb) ^ 0x3a;
  bVar2 = *(byte *)((int)puVar6 + 0xd);
  bVar3 = *(byte *)((int)puVar7 + 0xd);
  pbVar8[0xc] = *(byte *)((int)puVar6 + 0xc) ^ *(byte *)((int)puVar7 + 0xc) ^ 0x22;
  pbVar8[0xd] = bVar2 ^ bVar3 ^ 0x62;
  return;
}
```

Here we can see that the return type of the function is wrong, because we know that there is a comparison between two arrays of char.

So first of all we need to change the function return type using Ghidra into `char*`.

> Right click on the function -> Edit function signature -> change "void" to "char*"

After the change it becames:

```
char * o_54832dacb65f7c6e96c8b10f0795d249(void)

{
  undefined8 uVar1;
  byte bVar2;
  byte bVar3;
```

```
  byte bVar4;
  int iVar5;
  undefined8 *puVar6;
  undefined8 *puVar7;
  byte *pbVar8;

  iVar5 = getSignature();
  puVar6 = (undefined8 *)operator.new[](0xe);
  uVar1 = *(undefined8 *)(iVar5 + 0x236);
  *puVar6 = *(undefined8 *)(iVar5 + 0x230);
  *(undefined8 *)((int)puVar6 + 6) = uVar1;
  puVar7 = (undefined8 *)operator.new[](0xe);
  uVar1 = *(undefined8 *)(iVar5 + 0x389);
  *puVar7 = *(undefined8 *)(iVar5 + 899);
  *(undefined8 *)((int)puVar7 + 6) = uVar1;
  pbVar8 = (byte *)malloc(0xe);
  bVar2 = *(byte *)((int)puVar6 + 1);
  *pbVar8 = *(byte *)puVar6 ^ *(byte *)puVar7 ^ 0xd;
  bVar3 = *(byte *)((int)puVar6 + 2);
  pbVar8[1] = bVar2 ^ *(byte *)((int)puVar7 + 1) ^ 0x57;
  bVar2 = *(byte *)((int)puVar6 + 3);
  pbVar8[2] = bVar3 ^ *(byte *)((int)puVar7 + 2) ^ 7;
  bVar3 = *(byte *)((int)puVar6 + 4);
  pbVar8[3] = bVar2 ^ *(byte *)((int)puVar7 + 3) ^ 0x23;
  bVar2 = *(byte *)((int)puVar6 + 5);
  pbVar8[4] = bVar3 ^ *(byte *)((int)puVar7 + 4) ^ 0x29;
  bVar3 = *(byte *)((int)puVar6 + 6);
  pbVar8[5] = bVar2 ^ *(byte *)((int)puVar7 + 5) ^ 0x21;
  bVar2 = *(byte *)((int)puVar6 + 7);
  pbVar8[6] = bVar3 ^ *(byte *)((int)puVar7 + 6) ^ 0x36;
  bVar3 = *(byte *)(puVar6 + 1);
  bVar4 = *(byte *)(puVar7 + 1);
  pbVar8[7] = bVar2 ^ *(byte *)((int)puVar7 + 7) ^ 0x28;
  pbVar8[8] = bVar3 ^ bVar4 ^ 0x7b;
  pbVar8[9] = *(byte *)((int)puVar6 + 9) ^ *(byte *)((int)puVar7 + 9) ^ 0x6b;
  pbVar8[10] = *(byte *)((int)puVar6 + 10) ^ *(byte *)((int)puVar7 + 10) ^ 0x5b;
  pbVar8[0xb] = *(byte *)((int)puVar6 + 0xb) ^ *(byte *)((int)puVar7 + 0xb) ^ 0x3a;
  bVar2 = *(byte *)((int)puVar6 + 0xd);
  bVar3 = *(byte *)((int)puVar7 + 0xd);
  pbVar8[0xc] = *(byte *)((int)puVar6 + 0xc) ^ *(byte *)((int)puVar7 + 0xc) ^ 0x22;
  pbVar8[0xd] = bVar2 ^ bVar3 ^ 0x62;
  return (char *)pbVar8;
}
```

So we can start to write our C "script".

First of all the type of the variables are wrong. Looking at them we can see that

- `undefined8` **undefined8 could mean that the type is of 8 bytes, so it could be in C a** `uint_64_t`
- `byte` **is for sure** `char` **because** `pbVar8` **is returned and we know that the return must be a pointer to** `char`

With these information we start to follow line by line the function and try to figure out the correct code in C:

```
char* o_54832dacb65f7c6e96c8b10f0795d249(void)
{
    uint64_t uVar1;
    char bVar2;
    char bVar3;
    char bVar4;
    char *pcVar5;
    uint64_t *puVar6;
    uint64_t *puVar7;
    char *pbVar8;

    pcVar5 = getSignature();

    //puVar6 = (undefined8 *)operator.new[](0xe);
    puVar6 = (uint64_t *)malloc(0xe);

    //uVar1 = *(undefined8 *)(pcVar5 + 0x236);
    uVar1 = *(uint64_t *)(pcVar5 + 0x236);
```

```
    //*puVar6 = *(undefined8 *)(pcVar5 + 0x230);
    *puVar6 = *(uint64_t *)(pcVar5 + 0x230);

    //*(undefined8 *)((int)puVar6 + 6) = uVar1;
    *(uint64_t *)((char *)puVar6 + 6) = uVar1;

    //puVar7 = (undefined8 *)operator.new[](0xe);
    puVar7 = (uint64_t *)malloc(0xe);

    //uVar1 = *(undefined8 *)(pcVar5 + 0x389);
    uVar1 = *(uint64_t *)(pcVar5 + 0x389);

    //*puVar7 = *(undefined8 *)(pcVar5 + 899);
    *puVar7 = *(uint64_t *)(pcVar5 + 899);

    //*(undefined8 *)((int)puVar7 + 6) = uVar1;
    *(uint64_t *)((char *)puVar7 + 6) = uVar1;

    //pbVar8 = (byte *)malloc(0xe);
    pbVar8 = (char *)malloc(0xe);

    //bVar2 = *(byte *)((int)puVar6 + 1);
    bVar2 = *((char *)puVar6 + 1);

    //*pbVar8 = *(byte *)puVar6 ^ *(byte *)puVar7 ^ 0xd;
    *pbVar8 = *((char *)puVar6) ^ *((char *)puVar7) ^ 0xd;

    //bVar3 = *(byte *)((int)puVar6 + 2);
    bVar3 = *((char *)puVar6 + 2);

    //pbVar8[1] = bVar2 ^ *(byte *)((int)puVar7 + 1) ^ 0x57;
    pbVar8[1] = bVar2 ^ *((char *)puVar7 + 1) ^ 0x57;

    //bVar2 = *(byte *)((int)puVar6 + 3);
    bVar2 = *((char *)puVar6 + 3);

    //pbVar8[2] = bVar3 ^ *(byte *)((int)puVar7 + 2) ^ 7;
    pbVar8[2] = bVar3 ^ *((char *)puVar7 + 2) ^ 7;

    //bVar3 = *(byte *)((int)puVar6 + 4);
    bVar3 = *((char *)puVar6 + 4);

    //pbVar8[3] = bVar2 ^ *(byte *)((int)puVar7 + 3) ^ 0x23;
    pbVar8[3] = bVar2 ^ *((char *)puVar7 + 3) ^ 0x23;

    //bVar2 = *(byte *)((int)puVar6 + 5);
    bVar2 = *((char *)puVar6 + 5);

    //pbVar8[4] = bVar3 ^ *(byte *)((int)puVar7 + 4) ^ 0x29;
    pbVar8[4] = bVar3 ^ *((char *)puVar7 + 4) ^ 0x29;

    //bVar3 = *(byte *)((int)puVar6 + 6);
    bVar3 = *((char *)puVar6 + 6);

    //pbVar8[5] = bVar2 ^ *(byte *)((int)puVar7 + 5) ^ 0x21;
    pbVar8[5] = bVar2 ^ *((char *)puVar7 + 5) ^ 0x21;

    //bVar2 = *(byte *)((int)puVar6 + 7);
    bVar2 = *((char *)puVar6 + 7);

    //pbVar8[6] = bVar3 ^ *(byte *)((int)puVar7 + 6) ^ 0x36;
    pbVar8[6] = bVar3 ^ *((char *)puVar7 + 6) ^ 0x36;


    //bVar3 = *(byte *)(puVar6 + 1)
    bVar3 = *((char *)puVar6 + 1);

    //bVar4 = *(byte *)(puVar7 + 1);
    bVar4 = *((char *)puVar7 + 1);

    //pbVar8[7] = bVar2 ^ *(byte *)((int)puVar7 + 7) ^ 0x28;
    pbVar8[7] = bVar2 ^ *((char *)puVar7 + 7) ^ 0x28;

    //pbVar8[8] = bVar3 ^ bVar4 ^ 0x7b;
```

```
    pbVar8[8] = bVar3 ^ bVar4 ^ 0x7b;

    //pbVar8[9] = *(byte *)((int)puVar6 + 9) ^ *(byte *)((int)puVar7 + 9) ^ 0x6b;
    pbVar8[9] = *((char *)puVar6 + 9) ^ *((char *)puVar7 + 9) ^ 0x6b;

    //pbVar8[10] = *(byte *)((int)puVar6 + 10) ^ *(byte *)((int)puVar7 + 10) ^ 0x5b;
    pbVar8[10] = *((char *)puVar6 + 10) ^ *((char *)puVar7 + 10) ^ 0x5b;

    //pbVar8[0xb] = *(byte *)((int)puVar6 + 0xb) ^ *(byte *)((int)puVar7 + 0xb) ^ 0x3a;
    pbVar8[0xb] = *((char *)puVar6 + 0xb) ^ *((char *)puVar7 + 0xb) ^ 0x3a;

    //bVar2 = *(byte *)((int)puVar6 + 0xd);
    bVar2 = *((char *)puVar6 + 0xd);

    //bVar3 = *(byte *)((int)puVar7 + 0xd);
    bVar3 = *((char *)puVar7 + 0xd);

    //pbVar8[0xc] = *(byte *)((int)puVar6 + 0xc) ^ *(byte *)((int)puVar7 + 0xc) ^ 0x22;
    pbVar8[0xc] = *((char *)puVar6 + 0xc) ^ *((char *)puVar7 + 0xc) ^ 0x22;

    //pbVar8[0xd] = bVar2 ^ bVar3 ^ 0x62;
    pbVar8[0xd] = bVar2 ^ bVar3 ^ 0x62;

    return pbVar8;
}
```

So now we just need to reproduce `getSignature()` to assign a value to `pcVar5`

From Ghidra we can see:

```
undefined4 getSignature(void)

{
  undefined4 uVar1;
  _jmethodID *p_Var2;
  _jobject *p_Var3;
  undefined4 uVar4;
  _jmethodID *p_Var5;

  uVar1 = (**(code **)(*currentEnv + 0x7c))(currentEnv,currentMain);
  p_Var2 = (_jmethodID *)
           (**(code **)(*currentEnv + 0x84))
                     (currentEnv,uVar1,"getPackageManager","()Landroid/content/pm/PackageManager;");
  p_Var3 = (_jobject *)_JNIEnv::CallObjectMethod((_JNIEnv *)currentEnv,currentMain,p_Var2);
  uVar4 = (**(code **)(*currentEnv + 0x7c))(currentEnv,p_Var3);
  p_Var2 = (_jmethodID *)
           (**(code **)(*currentEnv + 0x84))
                     (currentEnv,uVar4,"getPackageInfo",
                      "(Ljava/lang/String;I)Landroid/content/pm/PackageInfo;");
  p_Var5 = (_jmethodID *)
           (**(code **)(*currentEnv + 0x84))
                     (currentEnv,uVar1,"getPackageName","()Ljava/lang/String;");
  uVar1 = _JNIEnv::CallObjectMethod((_JNIEnv *)currentEnv,currentMain,p_Var5);
  uVar1 = _JNIEnv::CallObjectMethod((_JNIEnv *)currentEnv,p_Var3,p_Var2,uVar1,0x40);
  uVar4 = (**(code **)(*currentEnv + 0x7c))(currentEnv,uVar1);
  uVar4 = (**(code **)(*currentEnv + 0x178))
                    (currentEnv,uVar4,"signatures","[Landroid/content/pm/Signature;");
  uVar1 = (**(code **)(*currentEnv + 0x17c))(currentEnv,uVar1,uVar4);
  p_Var3 = (_jobject *)(**(code **)(*currentEnv + 0x2b4))(currentEnv,uVar1,0);
  uVar1 = (**(code **)(*currentEnv + 0x7c))(currentEnv,p_Var3);
  p_Var2 = (_jmethodID *)
           (**(code **)(*currentEnv + 0x84))
                     (currentEnv,uVar1,"toCharsString","()Ljava/lang/String;");
  uVar1 = _JNIEnv::CallObjectMethod((_JNIEnv *)currentEnv,p_Var3,p_Var2);
  uVar4 = (**(code **)(*currentEnv + 0x2a4))(currentEnv,uVar1,0);
  (**(code **)(*currentEnv + 0x2a8))(currentEnv,uVar1,uVar4);
  return uVar4;
}
```

- It basically calls with a Java Native Interface the java line `getPackageManager().getPackageInfo().getPackageName().sigantures.toCharsString()`
- **So it simply returns the signature of the the certificate of the application**

We can write a pyhton script that return the certificate of the application in hexadecimal format using Androguard.

> Note: be sure that the version of Androguard is 3.4.0a1
> The new versions doesn't have a good documentation and this script won't work with them.
> You can install the version 3.4.0a1 with `pip install androguard==3.4.0a1`

This is the python script we need to take the certificate in hexadecimal format:

```
from androguard.core.bytecodes.apk import APK

apk = APK('braf.2024S.apk')

for cert in apk.get_certificates():
    print(cert.dump().hex())
```

- it basically takes all certifcates and prints them

Run it:

- `python3 get_certificate.py`

In this case the result is:

```
308202bb308201a3a003020102020401f4f45e300d06092a864886f70d01010b0500300e310c300a060355040b1303534153301e1
70d323131313131313134323230355a170d343631313035313134323230355a300e310c300a060355040b13035341533082012300d06
092a864886f70d0101010500038201 0f003082010a02820101008cfb5d5560445eb635cd56f69c1da17bee0a3b864312eba07b8dc
37bc2f174e94efc4b3b1f78cce4a2bbaf9636a3c840ce28b35a59bf866623186be6bb2d4f02cbc66901f3a35e152521352c0bff7b
8cbf539b03bceebf0765d26ebb4f3855c90d26611d9e555438e379e6d4cdb864572a417c82b2347b89d6b82c799b9c9f67960fe48
31c1e677a6c19fc7bf66bdebfca8a9244ac59dba371140d4c4920a24b1cc7fc55102fad2516f5c06dbaee868b029d55b7ace4c9fc
c94dfaf889cfc422f97048e418a74e3c11da18623f964f56e750c82fe02baa97d9f9827bc44987953766b868a2e2136292b16c02f
45b7ec4babc25af8d6ce70a0ef8d5e7f9fbc4a50203010001a321301f301d0603551d0e04160414411465dbf011aff9d742cec358
a94e3da8934799300d06092a864886f70d01010b050003820101000b456b166acc3a839b881463ba8903a8cc6401a4f39f5ee0ef7
64f78244b351a03df00d0f5938a900c108bb64e82a8ef1828384afa27ac7a733740ce60ee6f582066c44d9573b67d66792ed30bd5
17f466c50f418658076f980ffa49ab178620738693d95e9e1e9cae0e2e4ba0641f770c5e18e0c4a2e8af79537165569d12d72809a
c82cff54f76fc1ea08ebe92984631d54a0e7b0bc5e0db669b59cd4e4fb8c4a527f896e3b48b3d659f5a9b72caae7542d51ba2688d
865ffe58615b157ebeeffe498a66ded82fff1a95ff583b287dbfeb5b3a230e9efa687ca44a93693fdfda2e18ccf21f8f042f2c83f
05f77cab60a11513166a93a6a4945219440283593
```

So now we can write the final C script:

```
#include <stdlib.h>
#include <stdint.h>

char* getSignature()
{
    return
"308202bb308201a3a003020102020401f4f45e300d06092a864886f70d01010b0500300e310c300a060355040b1303534153301e
170d323131313131313134323230355a170d343631313035313134323230355a300e310c300a060355040b13035341533082012300d0
6092a864886f70d0101010500038201 0f003082010a02820101008cfb5d5560445eb635cd56f69c1da17bee0a3b864312eba07b8d
c37bc2f174e94efc4b3b1f78cce4a2bbaf9636a3c840ce28b35a59bf866623186be6bb2d4f02cbc66901f3a35e152521352c0bff7
b8cbf539b03bceebf0765d26ebb4f3855c90d26611d9e555438e379e6d4cdb864572a417c82b2347b89d6b82c799b9c9f67960fe4
831c1e677a6c19fc7bf66bdebfca8a9244ac59dba371140d4c4920a24b1cc7fc55102fad2516f5c06dbaee868b029d55b7ace4c9f
cc94dfaf889cfc422f97048e418a74e3c11da18623f964f56e750c82fe02baa97d9f9827bc44987953766b868a2e2136292b16c02
f45b7ec4babc25af8d6ce70a0ef8d5e7f9fbc4a50203010001a321301f301d0603551d0e04160414411465dbf011aff9d742cec35
8a94e3da8934799300d06092a864886f70d01010b050003820101000b456b166acc3a839b881463ba8903a8cc6401a4f39f5ee0ef
764f78244b351a03df00d0f5938a900c108bb64e82a8ef1828384afa27ac7a733740ce60ee6f582066c44d9573b67d66792ed30bd
517f466c50f418658076f980ffa49ab178620738693d95e9e1e9cae0e2e4ba0641f770c5e18e0c4a2e8af79537165569d12d72809
ac82cff54f76fc1ea08ebe92984631d54a0e7b0bc5e0db669b59cd4e4fb8c4a527f896e3b48b3d659f5a9b72caae7542d51ba2688
d865ffe58615b157ebeeffe498a66ded82fff1a95ff583b287dbfeb5b3a230e9efa687ca44a93693fdfda2e18ccf21f8f042f2c83
f05f77cab60a11513166a93a6a4945219440283593";
}

char* o_54832dacb65f7c6e96c8b10f0795d249(void)
{
    uint64_t uVar1;
    char bVar2;
    char bVar3;
    char bVar4;
    char *pcVar5;
    uint64_t *puVar6;
    uint64_t *puVar7;
    char *pbVar8;

    pcVar5 = getSignature();

    //puVar6 = (undefined8 *)operator.new[](0xe);
```

```
    puVar6 = (uint64_t *)malloc(0xe);

    //uVar1 = *(undefined8 *)(pcVar5 + 0x236);
    uVar1 = *(uint64_t *)(pcVar5 + 0x236);

    //*puVar6 = *(undefined8 *)(pcVar5 + 0x230);
    *puVar6 = *(uint64_t *)(pcVar5 + 0x230);

    //*(undefined8 *)((int)puVar6 + 6) = uVar1;
    *(uint64_t *)((char *)puVar6 + 6) = uVar1;

    //puVar7 = (undefined8 *)operator.new[](0xe);
    puVar7 = (uint64_t *)malloc(0xe);

    //uVar1 = *(undefined8 *)(pcVar5 + 0x389);
    uVar1 = *(uint64_t *)(pcVar5 + 0x389);

    //*puVar7 = *(undefined8 *)(pcVar5 + 899);
    *puVar7 = *(uint64_t *)(pcVar5 + 899);

    //*(undefined8 *)((int)puVar7 + 6) = uVar1;
    *(uint64_t *)((char *)puVar7 + 6) = uVar1;

    //pbVar8 = (byte *)malloc(0xe);
    pbVar8 = (char *)malloc(0xe);

    //bVar2 = *(byte *)((int)puVar6 + 1);
    bVar2 = *((char *)puVar6 + 1);

    //*pbVar8 = *(byte *)puVar6 ^ *(byte *)puVar7 ^ 0xd;
    *pbVar8 = *((char *)puVar6) ^ *((char *)puVar7) ^ 0xd;

    //bVar3 = *(byte *)((int)puVar6 + 2);
    bVar3 = *((char *)puVar6 + 2);

    //pbVar8[1] = bVar2 ^ *(byte *)((int)puVar7 + 1) ^ 0x57;
    pbVar8[1] = bVar2 ^ *((char *)puVar7 + 1) ^ 0x57;

    //bVar2 = *(byte *)((int)puVar6 + 3);
    bVar2 = *((char *)puVar6 + 3);

    //pbVar8[2] = bVar3 ^ *(byte *)((int)puVar7 + 2) ^ 7;
    pbVar8[2] = bVar3 ^ *((char *)puVar7 + 2) ^ 7;

    //bVar3 = *(byte *)((int)puVar6 + 4);
    bVar3 = *((char *)puVar6 + 4);

    //pbVar8[3] = bVar2 ^ *(byte *)((int)puVar7 + 3) ^ 0x23;
    pbVar8[3] = bVar2 ^ *((char *)puVar7 + 3) ^ 0x23;

    //bVar2 = *(byte *)((int)puVar6 + 5);
    bVar2 = *((char *)puVar6 + 5);

    //pbVar8[4] = bVar3 ^ *(byte *)((int)puVar7 + 4) ^ 0x29;
    pbVar8[4] = bVar3 ^ *((char *)puVar7 + 4) ^ 0x29;

    //bVar3 = *(byte *)((int)puVar6 + 6);
    bVar3 = *((char *)puVar6 + 6);

    //pbVar8[5] = bVar2 ^ *(byte *)((int)puVar7 + 5) ^ 0x21;
    pbVar8[5] = bVar2 ^ *((char *)puVar7 + 5) ^ 0x21;

    //bVar2 = *(byte *)((int)puVar6 + 7);
    bVar2 = *((char *)puVar6 + 7);

    //pbVar8[6] = bVar3 ^ *(byte *)((int)puVar7 + 6) ^ 0x36;
    pbVar8[6] = bVar3 ^ *((char *)puVar7 + 6) ^ 0x36;


    //bVar3 = *(byte *)(puVar6 + 1)
    bVar3 = *((char *)puVar6 + 1);

    //bVar4 = *(byte *)(puVar7 + 1);
    bVar4 = *((char *)puVar7 + 1);
```

```
    //pbVar8[7] = bVar2 ^ *(byte *)((int)puVar7 + 7) ^ 0x28;
    pbVar8[7] = bVar2 ^ *((char *)puVar7 + 7) ^ 0x28;

    //pbVar8[8] = bVar3 ^ bVar4 ^ 0x7b;
    pbVar8[8] = bVar3 ^ bVar4 ^ 0x7b;

    //pbVar8[9] = *(byte *)((int)puVar6 + 9) ^ *(byte *)((int)puVar7 + 9) ^ 0x6b;
    pbVar8[9] = *((char *)puVar6 + 9) ^ *((char *)puVar7 + 9) ^ 0x6b;

    //pbVar8[10] = *(byte *)((int)puVar6 + 10) ^ *(byte *)((int)puVar7 + 10) ^ 0x5b;
    pbVar8[10] = *((char *)puVar6 + 10) ^ *((char *)puVar7 + 10) ^ 0x5b;

    //pbVar8[0xb] = *(byte *)((int)puVar6 + 0xb) ^ *(byte *)((int)puVar7 + 0xb) ^ 0x3a;
    pbVar8[0xb] = *((char *)puVar6 + 0xb) ^ *((char *)puVar7 + 0xb) ^ 0x3a;

    //bVar2 = *(byte *)((int)puVar6 + 0xd);
    bVar2 = *((char *)puVar6 + 0xd);

    //bVar3 = *(byte *)((int)puVar7 + 0xd);
    bVar3 = *((char *)puVar7 + 0xd);

    //pbVar8[0xc] = *(byte *)((int)puVar6 + 0xc) ^ *(byte *)((int)puVar7 + 0xc) ^ 0x22;
    pbVar8[0xc] = *((char *)puVar6 + 0xc) ^ *((char *)puVar7 + 0xc) ^ 0x22;

    //pbVar8[0xd] = bVar2 ^ bVar3 ^ 0x62;
    pbVar8[0xd] = bVar2 ^ bVar3 ^ 0x62;

    return pbVar8;
}

int main()
{
    char* content = o_54832dacb65f7c6e96c8b10f0795d249();
    printf("%s", content);
    free(content);

    return 0;
}
```

Username getting

Unfortunately something is wrong with the code and it prints a result that is quite similar to the correct one: `Z_Uw)qb*sgXgt9`

To test it:

1. `gcc almost_correct.c`
2. `./a.out`

But after the *must be + and not s. The correct one is* `*Z_Uw)qb*+gXgt9`

Discussion of the vulnerabilities

The vulnerability here is related to improper validation of user input that through the use of a WebView is inserted into the html of the rendered page and then in our case interpreted as javascript and executed.

The solution to this problem concerns the proper validation of user input.

Another solution can be the using of mechanism that are able to detect when Frida is executed on the device or when the device is rooted.