iGNITE Technologies

Android Penetration Testing

APK Reverse Engineering

WWW.HACKINGARTICLES.IN

Contents

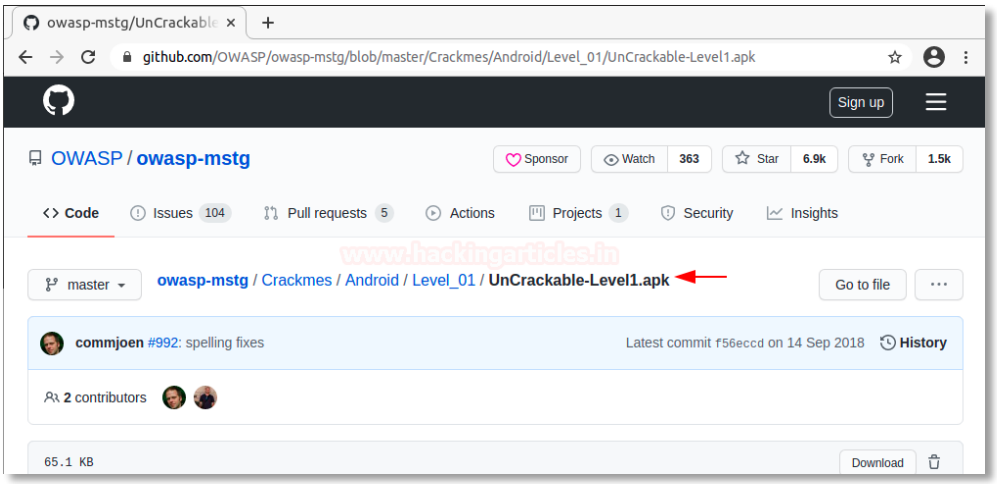
Abstract

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. An attacker would ideally be able to change the lines of bytecode to make the application behave in the way that the attacker wants. However, as easily as it is put, reversing and rebuilding an APK takes more than just a shallow statement. In this article, we'll be looking at the basics of decompilation, rebuilding, signing and changing the behavior of an application while we do this.

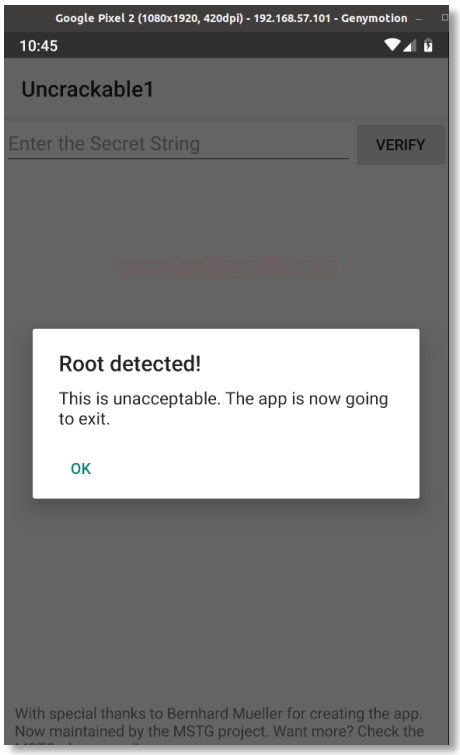
Reverse Engineering

Installation

Uncrackable is an intentionally **vulnerable APK** created by Bernhard Mueller which was later undertaken by the OWASP MSTG project. Level 1 of the 4 levelled challenge of APKs focuses on the basics of root detection bypass and hooking to find a secret encryption key. To install this application, follow **here**.



After you download the apk and install using adb in your genymotion emulator, you'd see something like this:



This means that the application has some kind of logic hardcoded that prevents it from opening in rooted devices and since genymotion's android APIs are root by default this is presenting the user with this problem. In real life environment, you'll see many applications in which developers code this root detection logic as a security measure to prevent aid to an attacker in his campaign and thus safeguard PIIs. However, this could also pose the possibility of poor coding practice and is exploitable. There are multiple ways to solve this first hurdle; hooking and removing this restriction while runtime is one option, making the application debuggable and injecting while executing is also one method but we'll follow the third method, which is reversing method. We'd decompile the application and remove the exit logic of the application to prevent exit.

Decompilation

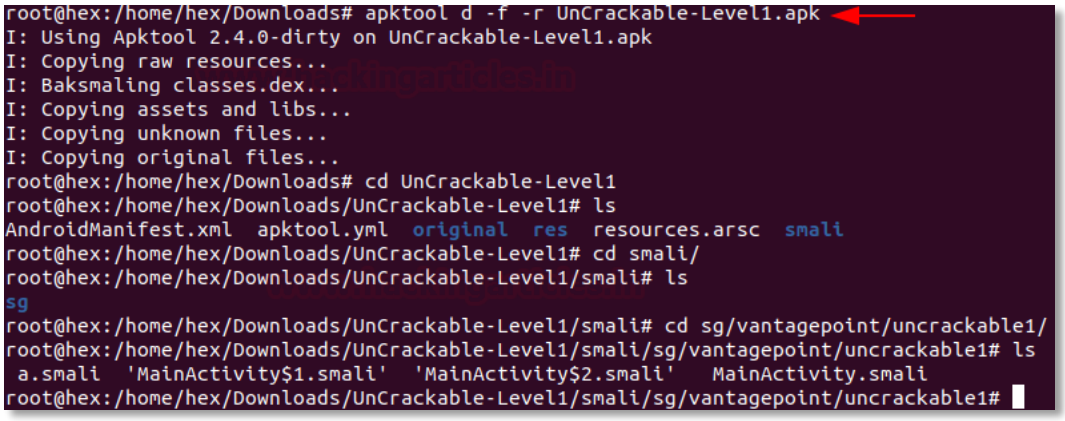
The Android decompilation process is fairly simple and resembles java decompilation in many ways. Basics of the decompilation process have already been covered in a previous article **here**. It is highly recommended you read para 3 of the article mentioned first and then resume this part.
It is to be noted that Dalvik bytecode is stored in *.dex format. This dex is the compiled version of source code which is further packed with resources, manifest, META-INF (certificate) into a zip file also known as an android app with an extension *.apk.
This *.dex file can be decompiled using **dexdump** which is provided in android SDK. In articles prior to this, we've used the **dex2jar** tool to convert dex files in readable jar format. This same was done by first unpacking the APK using apktool and then further converting **classes.dex** file into readable jar variant. So, let's unpack the APK first:

```
apktool d -f -r UnCrackable-Level1.apk
```

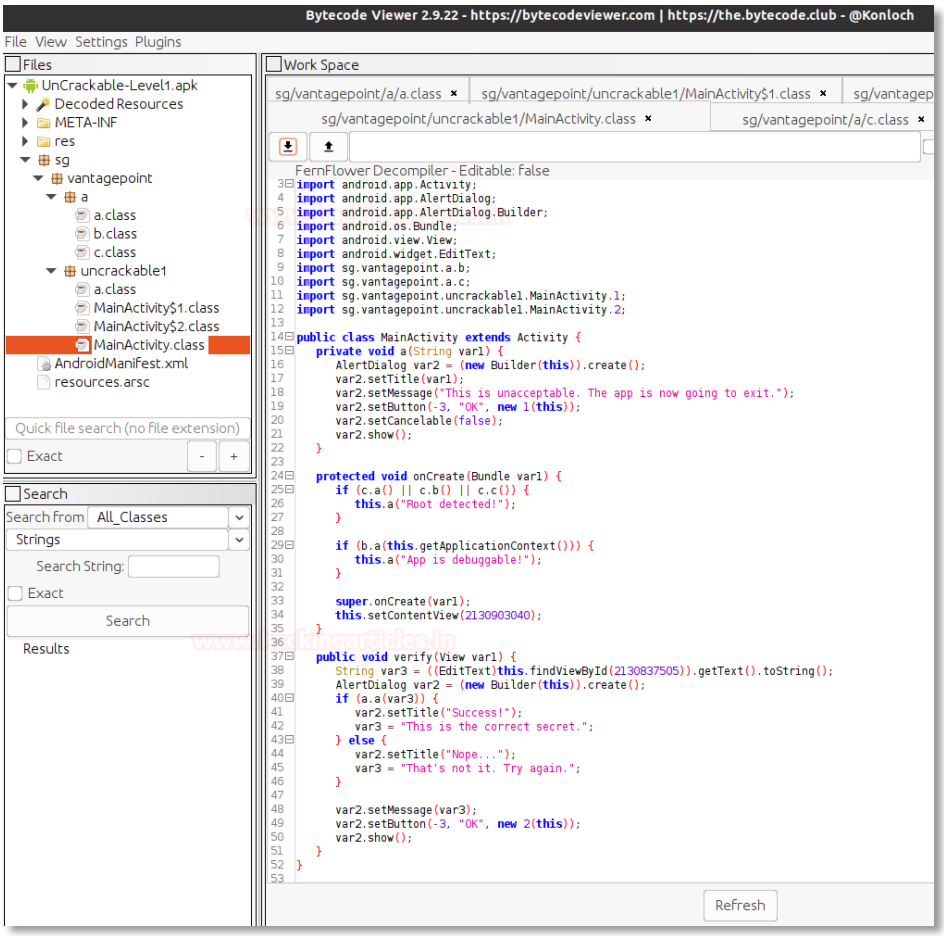
Here is something different from previous time; -r option has automatically converted classes.dex into **smali** files.

Smali files and modification

Smali in android is similar to what Assembly in Windows is. This is the human-readable version of dalvik bytecode. **Baksmali** is the tool which decompiles dex into smali files. Here, note that baksmali has converted classes.dex in smali files.



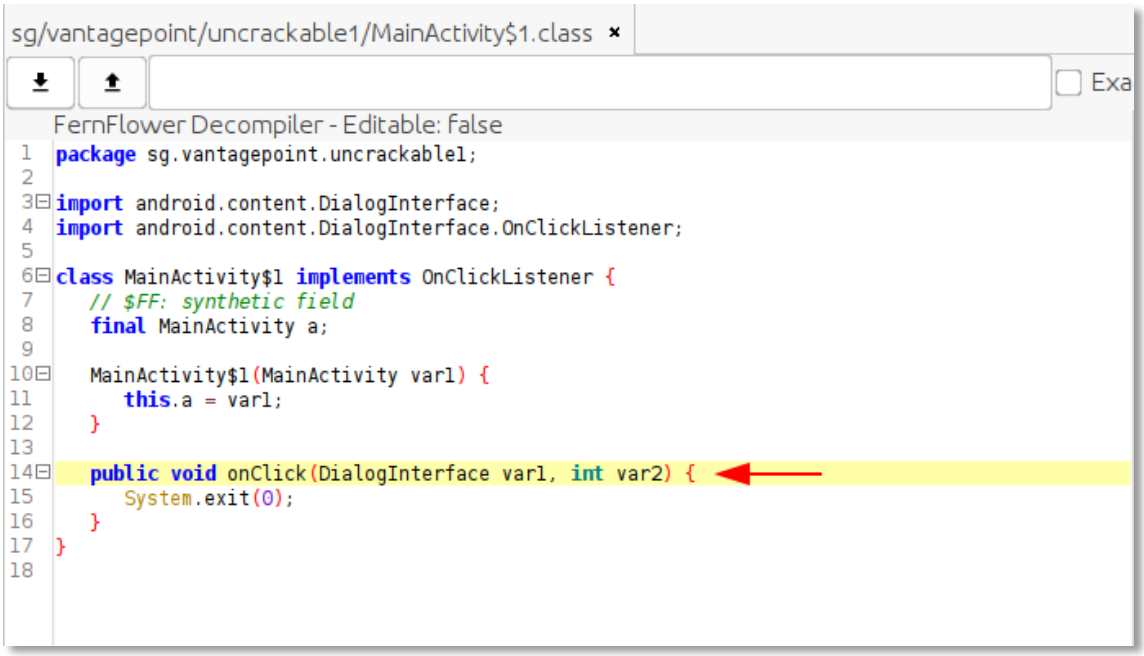
A nifty little tool known as bytecode viewer converts APK directly into readable format java code thus eliminating the need to use apktool then dex2jar and then jd-gui to view a readable java format. Here is how the application looks decompiled in bytecode viewer.



Oh, wait, while just decompiling this APK, my eye went on an interesting piece of code under MainActivity$1.class.

One thing to be noted is that since this application was obfuscated while building, it is forcing it to display ambiguous information like same class name multiple times, change of name of methods etc. This is due to Proguard obfuscation technique, which, is not properly implemented since the code is still pretty much readable. Strong obfuscation makes it a headache to reverse an application and makes it near impossible for an average attacker to patch APKs.

Now, let's have a look at MainActivity$1.class



Did you note as well that application is exiting using an onClick popup? Ahh! This popup is the popup that we saw in our installation step where the application is detecting whether the device is the root or not. So, hypothetically speaking, if I remove the logic to detect SU binaries, the system won't exist. Yes, that is one correct method, but I leave it to you readers to do and implement that. Other easier method is to remove the exit dialogue itself. This way, even if the application detects SU binaries, it will still not exit since **system.exit** won't be existing now.

To do that, I need to open the smali file of this class.

```
28
29      invoke-direct {p0}, Ljava/lang/Object;-><init>()V
30
31      return-void
32  .end method
33
34
35  # virtual methods
36  .method public onClick(Landroid/content/DialogInterface;I)V
37      .locals 0
38
39      const/4 p1, 0x0
40
41      invoke-static {p1}, Ljava/lang/System;->exit(I)V
42
43      return-void
44  .end method
```

Do you see the line where I've marked red? This is the same system.exit logic that we just saw. Now, it takes a little practice to understand smali instructions and is certainly not possible to understand this in a day or two but with a little smart work, we can make our way around to bypass root detection. Here, **invoke-static** refers to a function being invoked, that is defined in the very adjoining line of code: **Ljava/lang/System.** This is the path where a package of the system is stored. Next, **exit(I)** corresponds to exit() method of System, with I as in integer as a value which is denoted by **V**. Pretty simple right? Now let's delete this line altogether!

```
33
34
35  # virtual methods
36  .method public onClick(Landroid/content/DialogInterface;I)V
37      .locals 0
38
39      const/4 p1, 0x0
40
41
42
43      return-void
44  .end method
```

That's more than just pretty. This way we can rely on return-void instruction to return null value every time application detects a SU package and so, whole logic is rendered useless just by this alteration. Let's try to rebuild this APK now.

```
apktool b UnCrackable-Level1 -o new_uncrackable.apk
```

```
root@hex:/home/hex/Downloads# apktool b UnCrackable-Level1 -o new_uncrackable.apk
I: Using Apktool 2.4.0-dirty
I: Checking whether sources has changed...
I: Checking whether resources has changed...
I: Building apk file...
I: Copying unknown files/dir...
I: Built apk...
root@hex:/home/hex/Downloads# ls | grep new_
new_uncrackable.apk
root@hex:/home/hex/Downloads#
```

And just like that, we've built a new application. Let's try to install this new application in our genymotion device.

```
adb install new_uncrackable.apk
```

OOPS! That's peculiar. Did this work for you? Probably not. Let's understand why.

```
root@hex:/home/hex/Downloads# adb install new_uncrackable.apk
Performing Streamed Install
adb: failed to install new_uncrackable.apk: Failure [INSTALL_PARSE_FAILED_NO_CERTIFICATES: Failed
 to collect certificates from /data/app/vmdl49007080.tmp/base.apk: Attempt to get length of null
array]
root@hex:/home/hex/Downloads#
```

Signing APK and Rebuilding

The error I, and by extension, you must have received is a certificate error. Android uses something called a certificate and a Keystore. A public-key certificate, also known as a digital certificate or an identity certificate, contains the public key of a public/private key pair, as well as some other metadata identifying the owner of the key (for example, name and location). The owner of the certificate holds the corresponding private key.

When you sign an APK, the signing tool attaches the public-key certificate to the APK. The public-key certificate serves as a "fingerprint" that uniquely associates the APK to you and your corresponding private key. This helps Android ensure that any future updates to your APK are authentic and come from the original author. The key used to create this certificate is called the app signing key.

A Keystore is a binary file that contains one or more private keys.

Every app must use the same certificate throughout its lifespan in order for users to be able to install new versions as updates to the app.

When running or debugging your project from the IDE, Android Studio automatically signs your APK with a debug certificate generated by the Android SDK tools. The first time you run or debug your project in Android Studio, the IDE automatically creates the debug Keystore and certificate in $HOME/.android/debug. Keystore, and sets the Keystore and key passwords.

Because the debug certificate is created by the build tools and is insecure by design, most app stores (including the Google Play Store) will not accept an APK signed with a debug certificate for publishing.

But you must be wondering WHY IS THIS IMPORTANT?

We'd be creating our own Keystore and signing our APK using it. To do this we'll use a tool called **keytool**.

```
keytool -genkey -v -keystore harshit_key.keystore -alias harsh_key
-keyalg RSA -keysize 2048 -validity 10000
```

After that, you need to fill up your keystore password, name, org, city details and you'd have prepared yourself a keystore.

```
root@hex:/home/hex/Downloads# keytool -genkey -v -keystore harshit_key.keystore -alias harsh_k
ey -keyalg RSA -keysize 2048 -validity 10000
Enter keystore password:
Re-enter new password:
What is your first and last name?
  [Unknown]:  harshit rajpal
What is the name of your organizational unit?
  [Unknown]:  ignite
What is the name of your organization?
  [Unknown]:  ignite
What is the name of your City or Locality?
  [Unknown]:  New Delhi
What is the name of your State or Province?
  [Unknown]:  Delhi
What is the two-letter country code for this unit?
  [Unknown]:  IN
Is CN=harshit rajpal, OU=ignite, O=ignite, L=New Delhi, ST=Delhi, C=IN correct?
  [no]:  yes

Generating 2,048 bit RSA key pair and self-signed certificate (SHA256withRSA) with a validity
of 10,000 days
        for: CN=harshit rajpal, OU=ignite, O=ignite, L=New Delhi, ST=Delhi, C=IN
[Storing harshit_key.keystore]
root@hex:/home/hex/Downloads# ls | grep harshit
harshit_key.keystore
root@hex:/home/hex/Downloads#
```

Basically, your keystore now saves a self-signed certificate with 10,000 days of validity, which is an RSA 2048 bit key. Now, let's sign our patched app using this key.

```
jarsigner -verbose -sigalg SHA1withRSA -digestalg SHA1 -keystore
harshit_key.keystore new_uncrackable.apk harsh_key
```

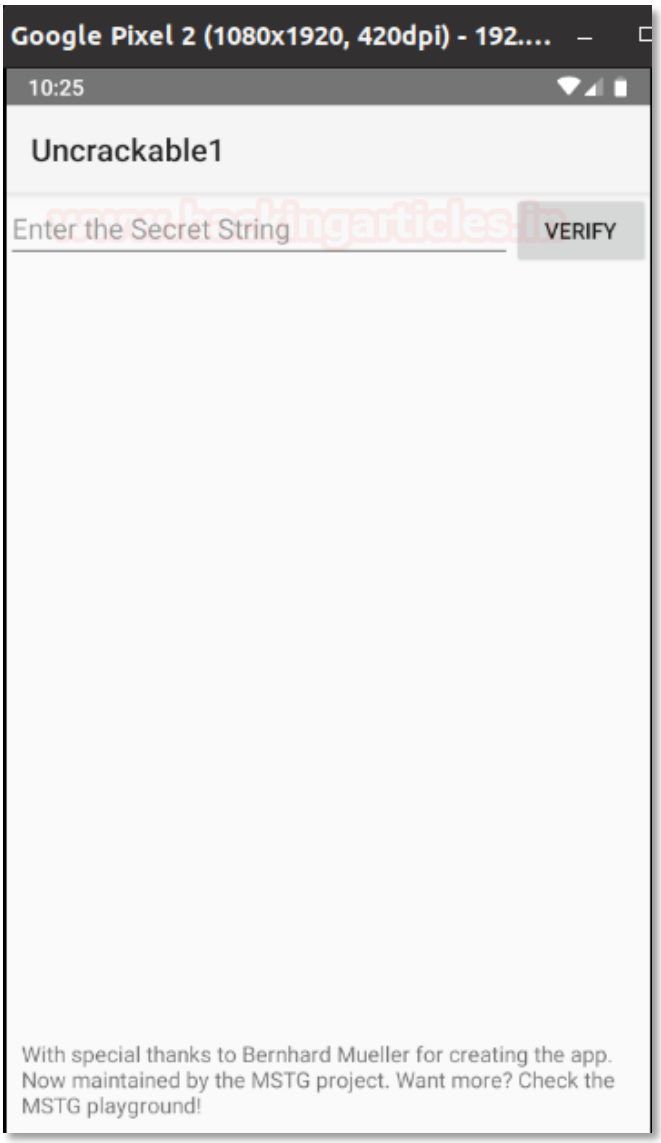
```
root@hex:/home/hex/Downloads# jarsigner -verbose -sigalg SHA1withRSA -digestalg SHA1 -keystore
 harshit_key.keystore new_uncrackable.apk harsh_key
Enter Passphrase for keystore:
   adding: META-INF/MANIFEST.MF
   adding: META-INF/HARSH_KE.SF
   adding: META-INF/HARSH_KE.RSA
  signing: classes.dex
  signing: resources.arsc
  signing: res/layout/activity_main.xml
  signing: res/mipmap-mdpi-v4/ic_launcher.png
  signing: res/mipmap-xxxhdpi-v4/ic_launcher.png
  signing: res/mipmap-xhdpi-v4/ic_launcher.png
  signing: res/mipmap-xxhdpi-v4/ic_launcher.png
  signing: res/mipmap-hdpi-v4/ic_launcher.png
  signing: res/menu/menu_main.xml
  signing: AndroidManifest.xml

>>> Signer
    X.509, CN=harshit rajpal, OU=ignite, O=ignite, L=New Delhi, ST=Delhi, C=IN
    [trusted certificate]

jar signed.

Warning:
The signer's certificate is self-signed.
The SHA1 algorithm specified for the -digestalg option is considered a security risk. This alg
orithm will be disabled in a future update.
The SHA1withRSA algorithm specified for the -sigalg option is considered a security risk. This
 algorithm will be disabled in a future update.
root@hex:/home/hex/Downloads#
```

Now, let's try once again to install our apk in genymotion device using adb and see if this time it throws an error or not.

```
adb install new_uncrackable.apk
```

```
root@hex:/home/hex/Downloads# adb install new_uncrackable.apk
Performing Streamed Install
Success
root@hex:/home/hex/Downloads#
```

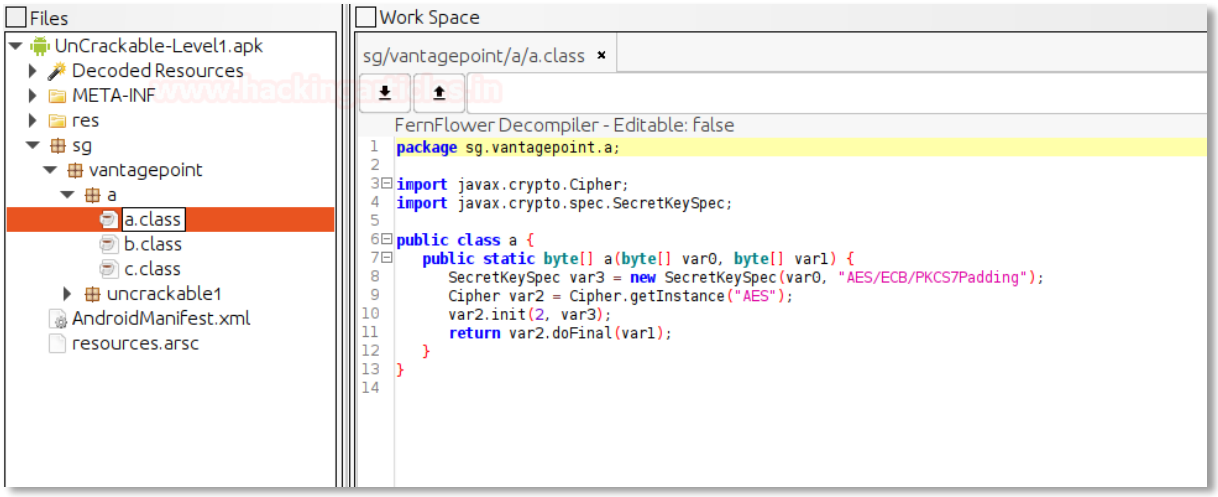
Perfect! Now that we've installed this, let's test run our application.



Voila! We've done it successfully. Let's finish the challenge now by using Frida hooking technique.

Solving Challenge

Now, the challenge is to extract the secret string and get it validated as a flag. Upon further investigating it came to our notice that method a() is returning the value of the secret string. Ha! This is poor practice but helpful for our case.



Now, all we need to do is to draft out a javascript hook for frida that will change the implementation of this a() and give the secret as an output in our very own console. Huge shoutout to 0daylabs for giving the code for this hook. Here is the code:

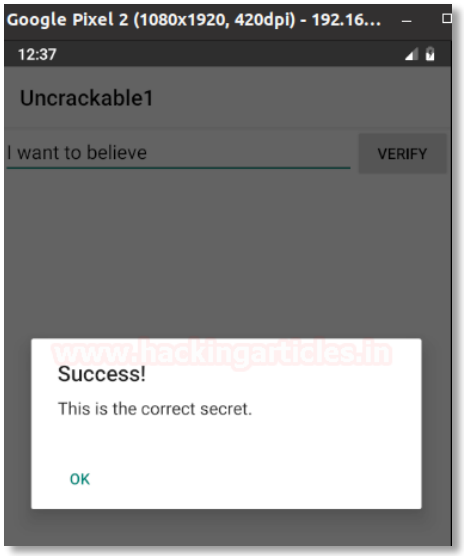
```
Java.perform(function () {

var aes = Java.use("sg.vantagepoint.a.a");

// Hook the function inside the class.

aes.a.implementation = function(var0, var1) {

// Calling the function itself to get its return value

var decrypt = this.a(var0, var1);

var flag = "";

// Converting the returned byte array to ascii and appending to a
string

for(var i = 0; i < decrypt.length; i++) {

flag += String.fromCharCode(decrypt[i]);

}

// Leaking our secret

console.log(flag);

return decrypt;
```

Now, we need to run this code using frida and check the output.

frida -U -f owasp.mstg.uncrackable1 -l expl.js --no-pause

```
root@hex:/home/hex/frida# cat expl.js
Java.perform(function () {
        var aes = Java.use("sg.vantagepoint.a.a");

        // Hook the function inside the class.
        aes.a.implementation = function(var0, var1) {

                // Calling the function itself to get its return value
                var decrypt = this.a(var0, var1);
                var flag = "";

                // Converting the returned byte array to ascii and appending to a string
                for(var i = 0; i < decrypt.length; i++) {
                                flag += String.fromCharCode(decrypt[i]);
           }

        // Leaking our secret
        console.log(flag);
        return decrypt;
        }
});
root@hex:/home/hex/frida# frida -U -f owasp.mstg.uncrackable1 -l expl.js --no-pause
     ____
    / _  |   Frida 14.2.10 - A world-class dynamic instrumentation toolkit
   | (_| |
    > _  |   Commands:
   /_/ |_|       help      -> Displays the help system
   . . . .       object?   -> Display information about 'object'
   . . . .       exit/quit -> Exit
   . . . .
   . . . .   More info at https://www.frida.re/docs/home/
Spawned `owasp.mstg.uncrackable1`. Resuming main thread!
[Google Pixel 2::owasp.mstg.uncrackable1]-> I want to believe
I want to believe
```

As you can see that the output is successfully dumped now! Let's see what the output is in the genymotion device.



And just like that, we've solved this challenge. Thanks for reading.