Cracking Damn Insecure and Vulnerable App (DIVA) – part 1:

What is DIVA?

DIVA is a vulnerable Android application. According to their official website, *"DIVA (Damn insecure and vulnerable App) is an App intentionally designed to be insecure. The aim of the App is to teach developers/QA/security professionals, flaws that are generally present in the Apps due poor or insecure coding practices."*

Getting ready with the lab:

We are going to solve all these challenges on an emulator. So, below are the steps to follow.

1. Download the application and extract the APK from the compressed tar file. You can download the application from here.

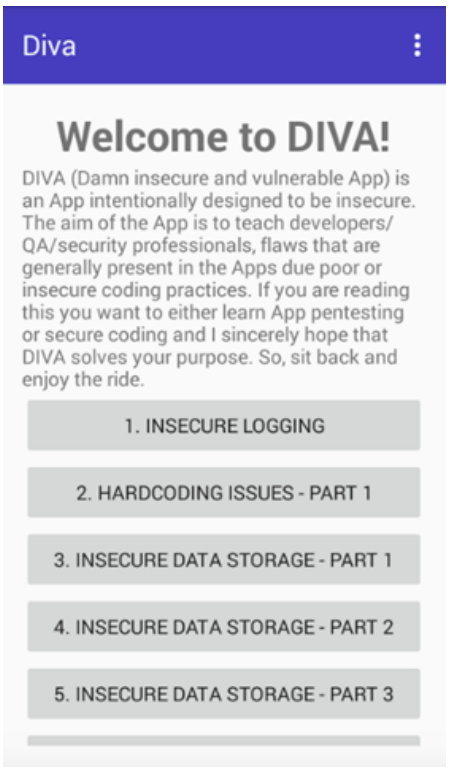
```
$ ls

diva-beta.apk    diva-beta.tar

$
```

2. Launch an emulator and install the apk file using as shown below.

```
$ adb install diva-beta.apk

1863 KB/s (1502294 bytes in 0.787s)
```

```
        pkg: /data/local/tmp/diva-beta.apk

Success

$
```

3. If you have done everything successfully, launch your application. You should see the following screen.



We are ready with the target application now. Let's also some important tools to crack these challenges.

We will use both manual and automated techniques to crack these challenges. So, set up the following tools preferably on a Unix based machine. I am using Mac OSX.

4. Download and install (if required) the following tools.

- Dex2jar & JD-GUI
- APKTOOL
- Drozer
- Android utilities – adb, sqlite3, aapt etc.

With this, you are ready with the setup. Now, let's explore how to solve the challenges provided in this application.

Reversing the target application:

One of the first steps to find vulnerabilities is static analysis by reversing the app. So, lets reverse engineer our target application to get ready to crack the challenges.

Note: This step is must, as some of the challenges require you to look at the source code.

Getting .java files Using Dex2Jar & JD-GUI:

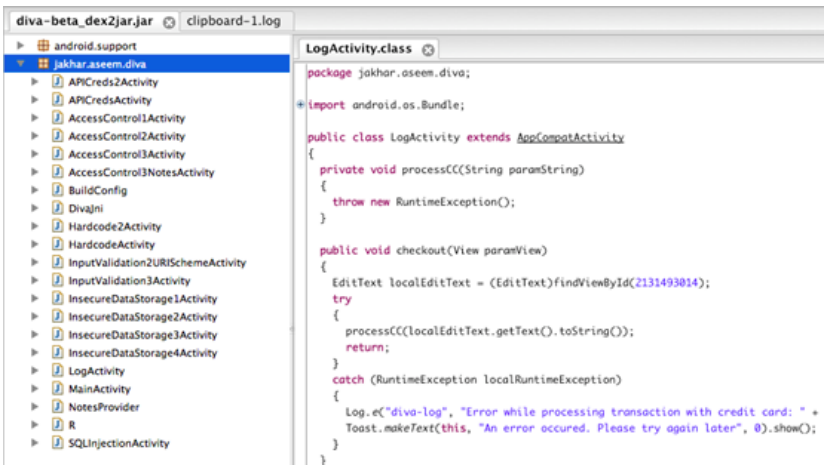
Getting the readable java files is always helpful during an assessment. So, let's get .java files using dex2jar and JD-GUI tools we set up earlier. As mentioned earlier, I have set up my lab on a machine running Mac OSX. So, run the following command to convert the dex file into a jar file.

```
$ sh dex2jar.sh diva-beta.apk

this cmd is deprecated, use the d2j-dex2jar if possible

dex2jar version: translator-0.0.9.15

dex2jar diva-beta.apk -> diva-beta_dex2jar.jar
```

Done.

Once done, you should see `diva-beta_dex2jar.jar` file in the same directory where you have dex2jar.

Open up this newly created `diva-beta_dex2jar.jar` file using JD-GUI as shown below.



Nice! As we can see above, we now have .java files.

Getting AndroidManifest.xml and smali code using apktool:

Another important thing in static analysis of Android apps is having access to Androidmanifest.xml file. This gives us great deal of information about the app and its internal structure. Additionally, apktool provides us the smali code. So, let's use apktool to get the AndroidManifest.xml file and smali code as shown below.

```
$ java -jar apktool_2.0.3.jar d diva-beta.apk -o output
```

```
srini's MacBook:output srini0x00$ cat AndroidManifest.xml
<?xml version="1.0" encoding="utf-8" standalone="no"?>
<manifest xmlns:android="http://schemas.android.com/apk/res/android" package="ja
khar.aseem.diva" platformBuildVersionCode="23" platformBuildVersionName="6.0-216
6767">
    <uses-permission android:name="android.permission.WRITE_EXTERNAL_STORAGE"/>
    <uses-permission android:name="android.permission.READ_EXTERNAL_STORAGE"/>
    <uses-permission android:name="android.permission.INTERNET"/>
    <application android:allowBackup="true" android:debuggable="true" android:ic
on="@mipmap/ic_launcher" android:label="@string/app_name" android:supportsRtl="t
rue" android:theme="@style/AppTheme">
```

Figure 1 Contents of AndroidManifest.xml(truncated)

```
APICreds2Activity.smali                    NotesProvider$DBHelper.smali
APICredsActivity.smali                     NotesProvider.smali
AccessControl1Activity.smali               R$anim.smali
AccessControl2Activity.smali               R$attr.smali
AccessControl3Activity.smali               R$bool.smali
AccessControl3NotesActivity.smali          R$color.smali
BuildConfig.smali                          R$dimen.smali
DivaJni.smali                              R$drawable.smali
Hardcode2Activity.smali                    R$id.smali
HardcodeActivity.smali                     R$integer.smali
InputValidation2URISchemeActivity.smali    R$layout.smali
InputValidation3Activity.smali             R$menu.smali
InsecureDataStorage1Activity.smali         R$mipmap.smali
InsecureDataStorage2Activity.smali         R$string.smali
InsecureDataStorage3Activity.smali         R$style.smali
InsecureDataStorage4Activity.smali         R$styleable.smali
LogActivity.smali                          R.smali
MainActivity.smali                         SQLInjectionActivity.smali
```

Figure 2: List of extracted smali files

If you want more details on how to setup the above mentioned tools, please refer to my article on Reverse Engineering here.

All set to crack the challenges. Let's crack some of the challenges now.

Challenge 1: "1. INSECURE LOGGING"

Steps to solve:

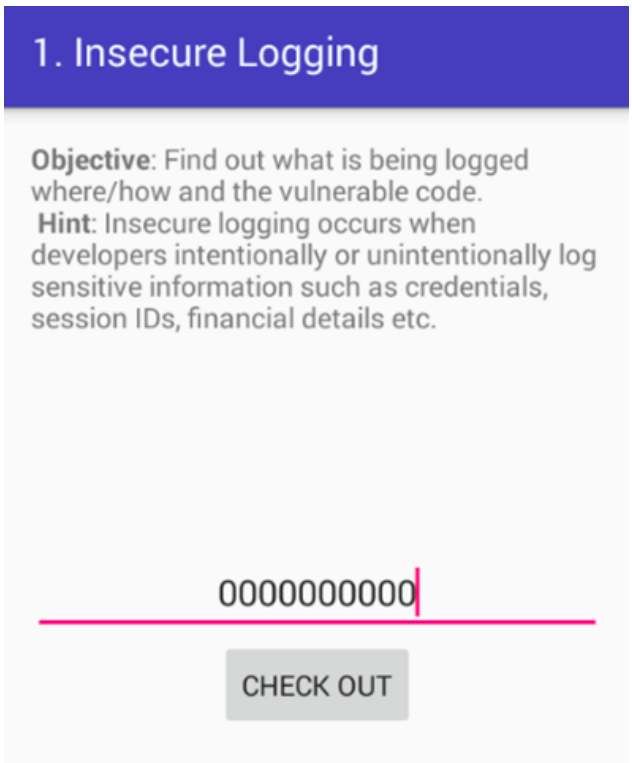
Click on "1. INSECURE LOGGING" in your application. The goal is to find out where the user-entered information is being logged and also the code making this vulnerable.

It is common that Android apps log sensitive information into logcat. So, let's see if this application is logging the data into logcat.

Run the following command in your terminal.

```
$adb logcat
```

Now, enter some data into the application's edit text field.



Check your logs after clicking "CHECK OUT".

Output in logcat:

```
D/MobileDataStateTracker( 469): default:
setPolicyDataEnable(enabled=true)

D/LightsService( 469): Excessive delay setting light: 86ms

D/dalvikvm( 1695): GC_CONCURRENT freed 136K, 6% free
3845K/4060K, paused 7ms+4ms, total 93ms

E/diva-log( 1695): Error while processing transaction with
credit card: 0000000000

E/SoundPool( 469): error loading
/system/media/audio/ui/Effect_Tick.ogg
```

As you can see in the above logs, the data entered by the user is being logged.

Vulnerable code:

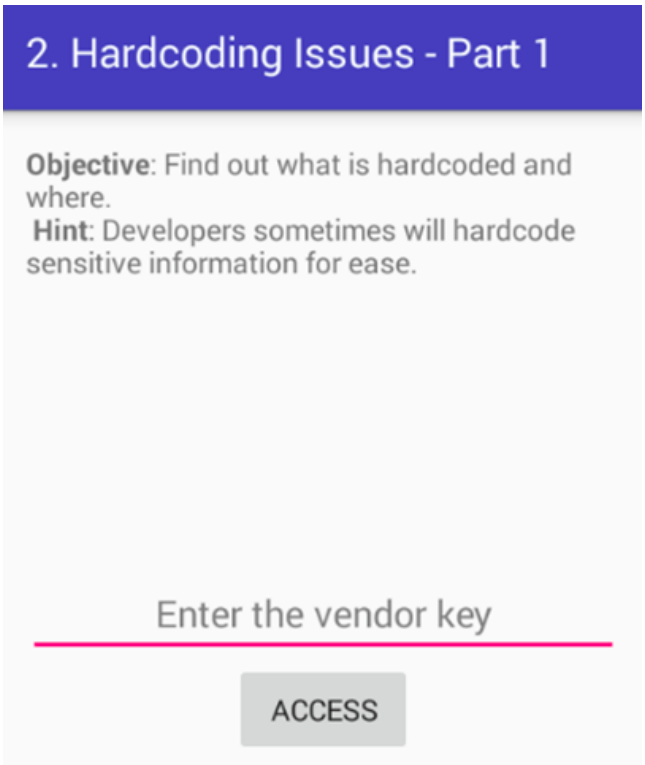
Open up LogActivity.class file using JD-GUI and check the following piece of code.

```
public void checkout(View paramView)
{
  EditText localEditText = (EditText)findViewById(2131493014);
  try
  {
    processCC(localEditText.getText().toString());
    return;
  }
  catch (RuntimeException localRuntimeException)
  {
    Log.e("diva-log", "Error while processing transaction with credit card: " + localEditText.getText().toString());
    Toast.makeText(this, "An error occured. Please try again later", 0).show();
  }
}
```

As you can see in the above figure, the following line is used log the data entered by the user into logcat.

```
Log.e("diva-log", "Error while processing transaction with credit card: " + localEditText.getText().toString());
```

Challenge 2: "2. HARDCODING ISSUES – PART 1"



Steps to solve:

Click on "2. HARDCODING ISSUES – PART 1" in your application. The goal is to find out the vendor key and enter it into the application to gain access.

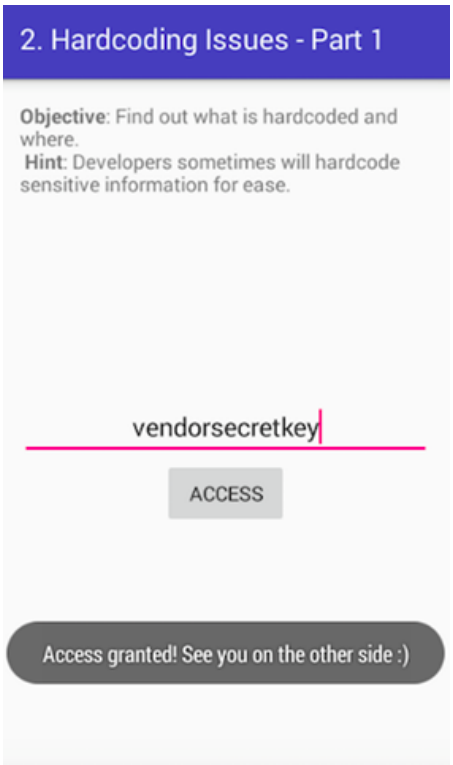
It is common that developers hardcode sensitive information in the application's source code. So, open up "HardcodeActivity.class" file using JD-GUI once again and observe the following piece of code.

```
public class HardcodeActivity extends AppCompatActivity
{
  public void access(View paramView)
  {
    if (((EditText)findViewById(2131492987)).getText().toString().equals("vendorsecretkey"))
    {
      Toast.makeText(this, "Access granted! See you on the other side :)", 0).show();
      return;
    }
    Toast.makeText(this, "Access denied! See you in hell :D", 0).show();
  }
```

The secret key has been hardcoded to match it against the user input as shown in the line below.

```
if (((EditText)findViewById(2131492987)).getText().toString().equals("vendorsecretkey"))
```

Just enter this secret key found in the source code and you are done ☺



Access granted!

We will discuss "INSECURE DATA STORAGE" solutions in the next article.

More Links:

DIVA Official website: http://www.payatu.com/damn-insecure-and-vulnerable-app/

NEXT: CRACKING DAMN I...

AUTHOR
Srinivas

Srinivas is an Information Security professional with 4 years of industry experience in Web, Mobile and Infrastructure Penetration Testing. He is currently a security researcher at Infosec Institute Inc. He holds Offensive Security Certified Professional(OSCP) Certification. He blogs atwww.androidpentesting.com. Email: srini0x00@gmail.com

About InfoSec

InfoSec Institute is the best source for high quality information security training. We have been training Information Security and IT Professionals since 1998 with a diverse lineup of relevant training courses. In the past 16 years, over 50,000 individuals have trusted InfoSec Institute for their professional development needs!

Connect with us

Stay up to date with InfoSec Institute and Intense School - at info@infosecinstitute.com

Join our newsletter

Get the latest news, updates & offers straight to your inbox.

ENTER YOUR EMAIL SUBSCRIBE