

Planet X : Android Application Based CTF Challenge Walkthrough



Saurabh Jain

Sep 27 · 8 min read



Planet-X is an intermediate level Android application CTF challenge. The aim of this CTF challenge is to learn and concentrate on the basic flaws which are found while performing security assessment of a mobile application.

We will be observing the basic misconfigurations which will lead our path and help us to find the flag.

Let's take a minute to thank [Moksh](#) for creating this challenge. If someone wants to try and solve the challenge before going through the walkthrough, the link for the CTF can be found [\[here\]](#) and the application can be downloaded from [\[here\]](#).

So, before beginning the walkthrough, highlighting the fact that the challenge can be solved in two different ways. Both the ways teach us something unique and make us aware about the security flaws.

Just stay connected till the end...

First Approach is basically the intended way how the challenge was designed to be solved.

Tools Used :

adb : command line tool that lets you communicate with device

apktool : command line tool for reverse engineering android applications

jadx-gui : tool for producing Java source code from Android Dex and APK files

Android Studio : official Integrated Development Environment (IDE) for Android app development

Device : Android Device/Android Studio Emulator/Genymotion Emulator

Connecting the device with a USB cable and entering command for checking proper connectivity.

```
adb devices
```

The above command will list down all the connected devices/emulators.

```
$ adb devices
List of devices attached
52032295e8f96377    device
```

The above exhibit shows the list of devices connected to the system

Note : Make sure to connect the android phone with debugging mode enabled for initiating the application installation process.

After downloading the application from the above given link, the application can be installed in device/emulator by a very simple command.

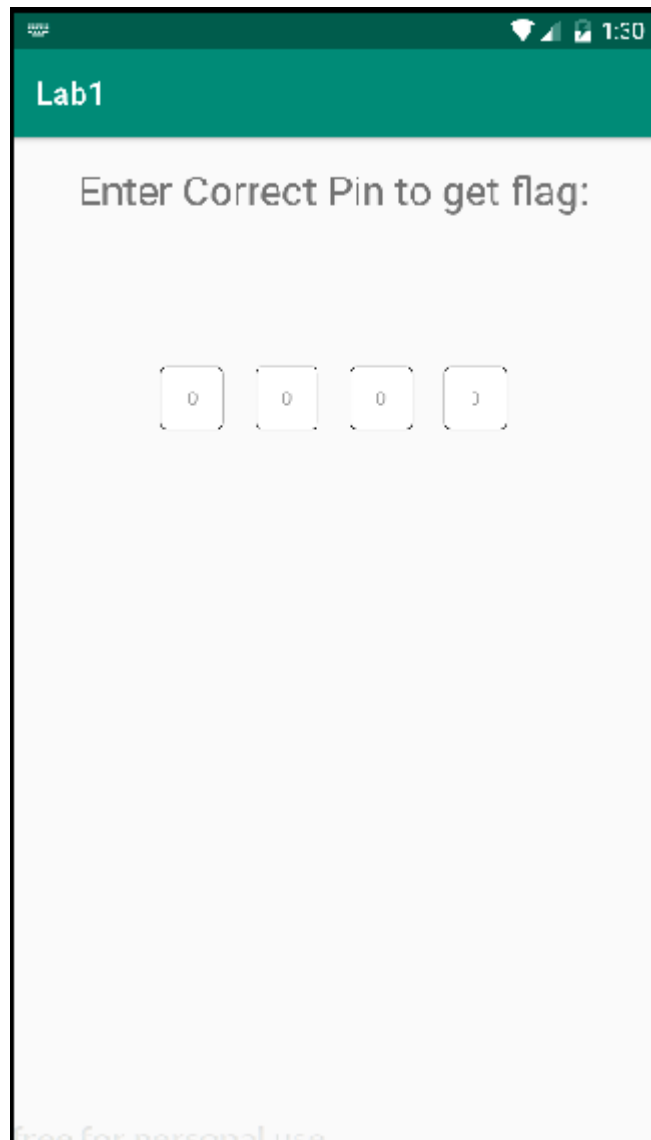
```
adb install <apk-name>
```

```
saurabhjain@kali:~$ adb install com.cybergym.lab4.apk
Performing Push Install
com.cybergym.lab4.apk: 1 file pushed. 64.2 MB/s (1047597 bytes in 0.016s)
pkg: /data/local/tmp/com.cybergym.lab4.apk
```

success

The above exhibit shows that application is successfully installed in the device

We can run the application in the device/emulator

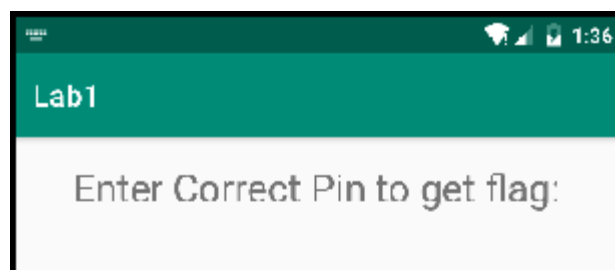


The above exhibit shows the first page after running the application

So the first page indicates that the application needs a 4 digit pin.

Hey ! Why don't we brute force the pin? It's just a 4 digit pin!

Let's begin.. Just trying "1111"

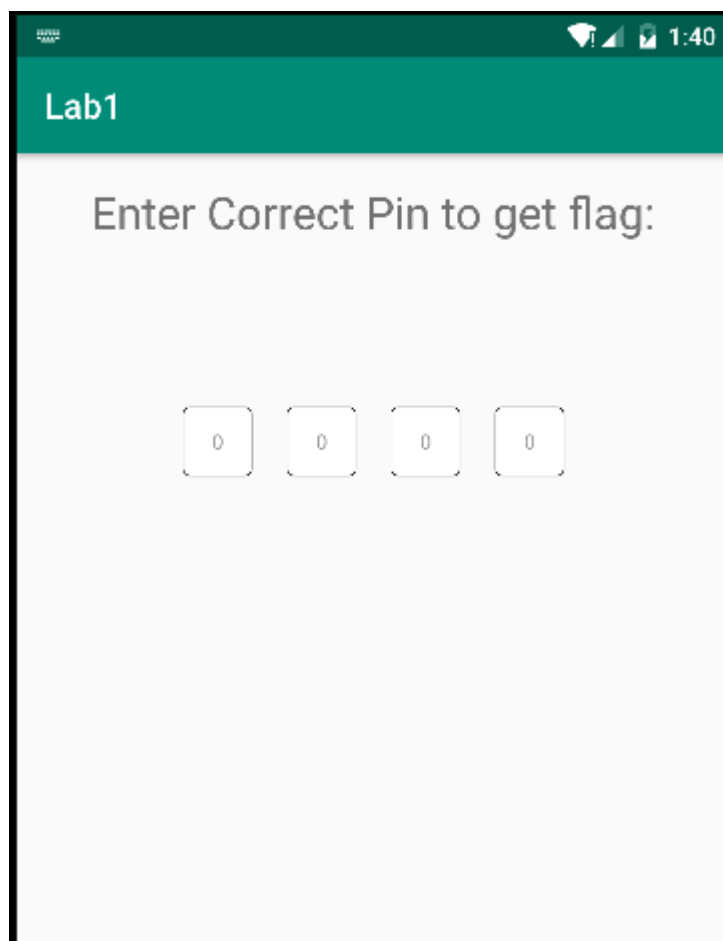


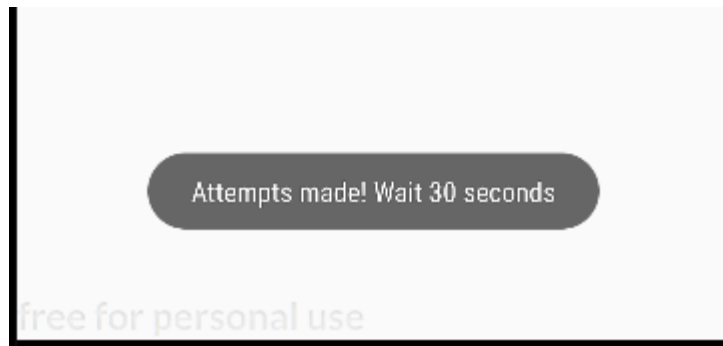


The above exhibit shows the message that only two attempts are remaining

The message says *“Incorrect Pin, 2 attempts remaining”*

That means we have only 3 attempts to break the pin. Because after that..





The above exhibits shows the message that we need to wait for 30 seconds for making next attempt

The message says “Attempts made! Wait for 30 seconds..”

So, this loop will continue i.e trying 3 wrong attempts and then waiting for 30 seconds for giving a next attempt which means we cannot bruteforce the PIN.



So, we need to find a way out of it. Let's try reverse engineering the application...

As we have **jadx-gui** in our bucket. We opened the APK using **jadx-gui** so that we can read the source code of the application.

`jadx-gui <apk-name>`

Now start reading the code from *MainActivity.java*

```
public class MainActivity extends AppCompatActivity {
    Context a;
    /* renamed from: a reason: collision with other field name */
    Cursor f1003a;
    /* renamed from: a reason: collision with other field name */
    SQLiteDatabase f1004a;
    int b;
```

Start reading code from MainActivity

The code seems a bit obfuscated

```

int b;
int c = 2;
private static String a() {
    return String.valueOf(((int) (Math.random() * 9000.0d)) + 1000);
}

public void onCreate(Bundle bundle) {
    super.onCreate(bundle);
    setContentView((int) R.layout.activity_main);
    SQLiteDatabase.loadLibs(this);
    this.a = getApplicationContext();
    Pinview pinview = (Pinview) findViewById(R.id.pinview1);
    final TextView textView = (TextView) findViewById(R.id.textView1);
}

```

The above exhibit shows that the source code is obfuscated

So, here we can see there is an instance of *SQLiteDatabase*.

Let's read some more code...

```

}
try {
    String str = new String("123456");
    getDatabasePath("q.db").getParentFile().mkdir();
    deleteDatabase("q.db");
    this.f1004a = SQLiteDatabase.openOrCreateDatabase(getDatabasePath("q.db"), str,
        (SQLiteDatabase.CursorFactory) null);
    this.f1004a.execSQL("CREATE TABLE IF NOT EXISTS a(z VARCHAR,a VARCHAR);");
    SQLiteDatabase openOrCreateDatabase = SQLiteDatabase.openOrCreateDatabase
        (getDatabasePath("kkk.db"), "12345678", (SQLiteDatabase.CursorFactory) null);
    openOrCreateDatabase.execSQL("CREATE TABLE IF NOT EXISTS name(user VARCHAR, pass
        VARCHAR);");
    openOrCreateDatabase.execSQL("INSERT INTO name VALUES('moksh','password')");
    openOrCreateDatabase.close();
} catch (Exception e) {
    Toast.makeText(this, "Error: " + e.getMessage(), 0).show();
    Log.e("PinView", e.getMessage());
    e.printStackTrace();
}
}

```

Two SQLite Databases being created
q.db and kkk.db

The above code snippet shows that the two SQLite Database files are being created

From the above highlighted code we can see that two databases are being created named as

- q.db
- kkk.db

Grab a chair for next couple of points...

```

try {
    String str = new String("123456");
    getDatabasePath("q.db").getParentFile().mkdir();
    deleteDatabase("q.db");
}

```



```

deleteDatabase("q.db");
this.f1004a = SQLiteDatabase.openOrCreateDatabase(getDatabasePath("q.db"), str,
(SQLiteDatabase.CursorFactory) null);
this.f1004a.execSQL("CREATE TABLE IF NOT EXISTS a(z VARCHAR,a VARCHAR);");
SQLiteDatabase openOrCreateDatabase = SQLiteDatabase.openOrCreateDatabase
(getDatabasePath("kkk.db"), "12345678", (SQLiteDatabase.CursorFactory) null);
openOrCreateDatabase.execSQL("CREATE TABLE IF NOT EXISTS name(user VARCHAR, pass
VARCHAR)");
openOrCreateDatabase.execSQL("INSERT INTO name VALUES('moksh','password')");
openOrCreateDatabase.close();
} catch (Exception e) {
    Toast.makeText(this, "Error: " + e.getMessage(), 0).show();
    Log.e("PinView", e.getMessage());
    e.printStackTrace();
}

```

The above code snippet highlights the details of the two created SQLite Databases

kkk.db

- one table created as 'name'
- table 'name' has two columns → (user VARCHAR, pass VARCHAR)
- values stored for (user,pass) → (VALUES('moksh','password'))

```

this.f1003a = this.f1004a.rawQuery("SELECT * from a;", (String[]) null);
this.f1003a.moveToFirst();
if (this.f1003a.getCount() <= 0) {
    String a2 = a();
    String a3 = a();
    SQLiteDatabase sQLiteDatabase = this.f1004a;
    sQLiteDatabase.execSQL("INSERT INTO a(z,a) VALUES('" + a2 + "', '" + a3 + "')");
}
this.f1003a.close();

```

← a() is function for creating random numbers

The above code snippet shows that function call used for created random numbers

Declaration of function a()

```

/* renamed from: a reason: collision with other field name */
SQLiteDatabase f1004a;
int b;
int c = 2;

private static String a() {
    return String.valueOf(((int) (Math.random() * 9000.0d)) + 1000);
}

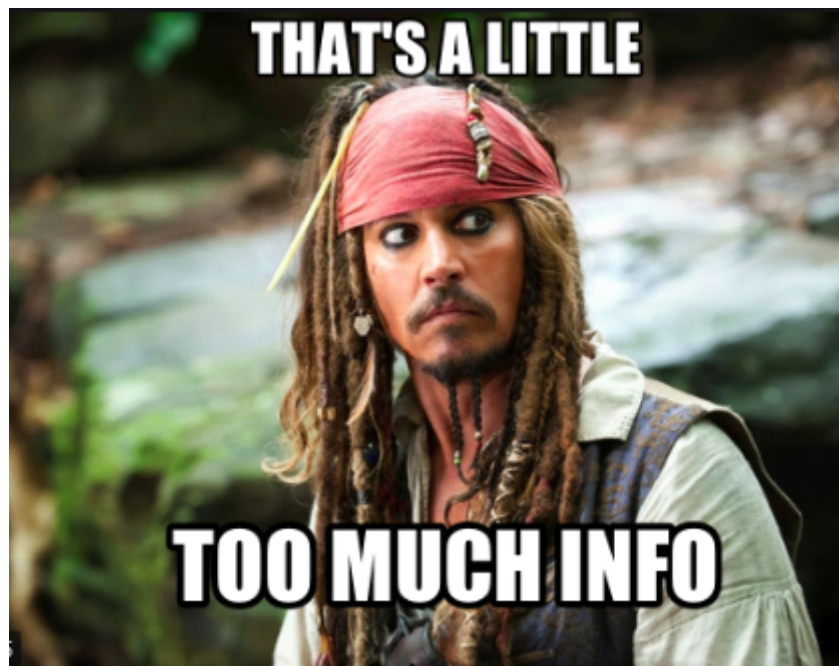
```

Function responsible for creating random numbers

The above code snippet shows the function definition for creating random numbers

q.db

- one table created as 'a'
- table name has two columns (z VARCHAR,a VARCHAR))
- values stored for (z,a) are random, which is generated by a random number generator function.



So let's grab the databases for our next hint...

```
root@vbox86p:/data/data/com.moksh.lab1/databases #  
root@vbox86p:/data/data/com.moksh.lab1/databases #  
root@vbox86p:/data/data/com.moksh.lab1/databases # ls -la  
-rw----- u0_a66 u0_a66 2048 2020-08-19 13:11 kkk.db  
-rw----- u0_a66 u0_a66 2048 2020-08-19 13:11 q.db  
root@vbox86p:/data/data/com.moksh.lab1/databases #  
root@vbox86p:/data/data/com.moksh.lab1/databases #
```

The above exhibit shows that the databases are being created and stored in application local data storage

Here we can see.. We have two databases *kkk.db* and *q.db*

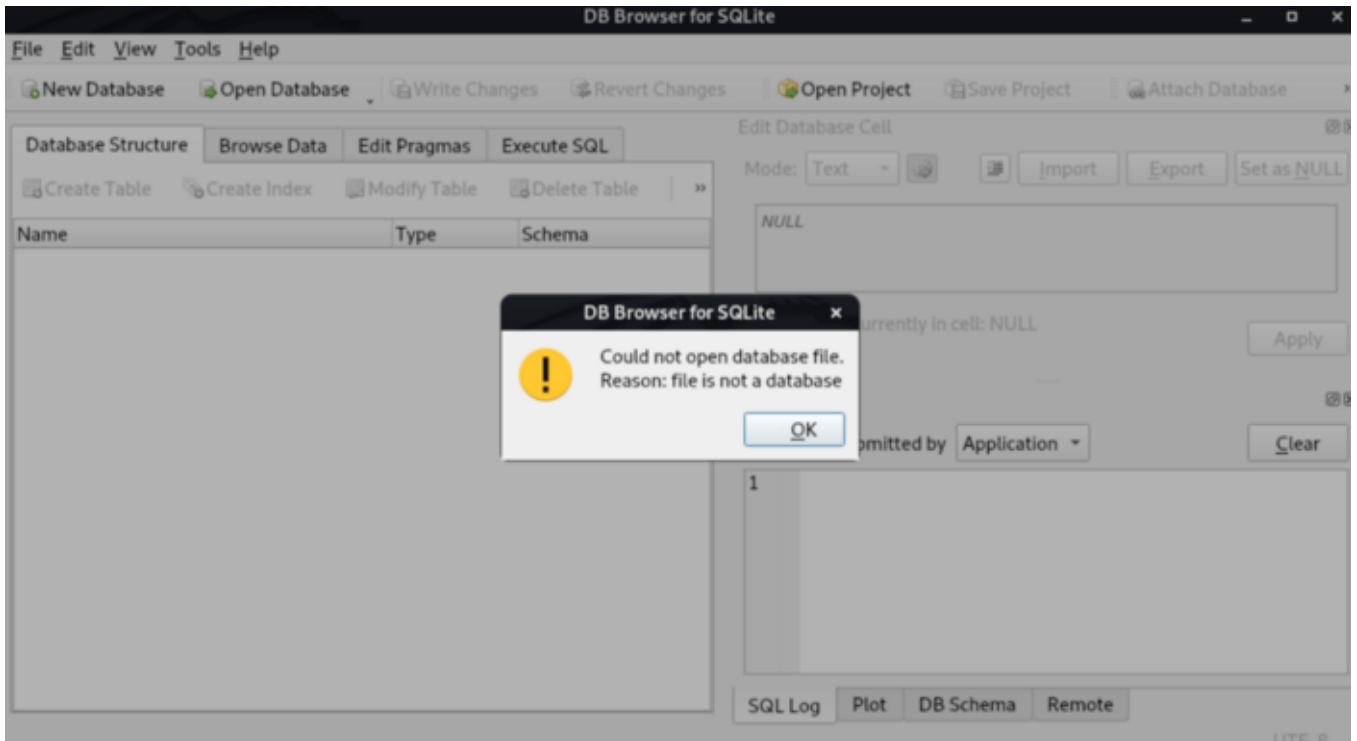
Let's pull the database from the device

```
adb pull /data/data/com.moksh.lab1/databases/
```

Now try reading them using *sqlitebrowser* using a very simple command

```
sqlitebrowser q.db
```


While trying to open them using sqlitebrowser, facing an error...



The above exhibit shows the attempt to read .db file using sqlitebrowser

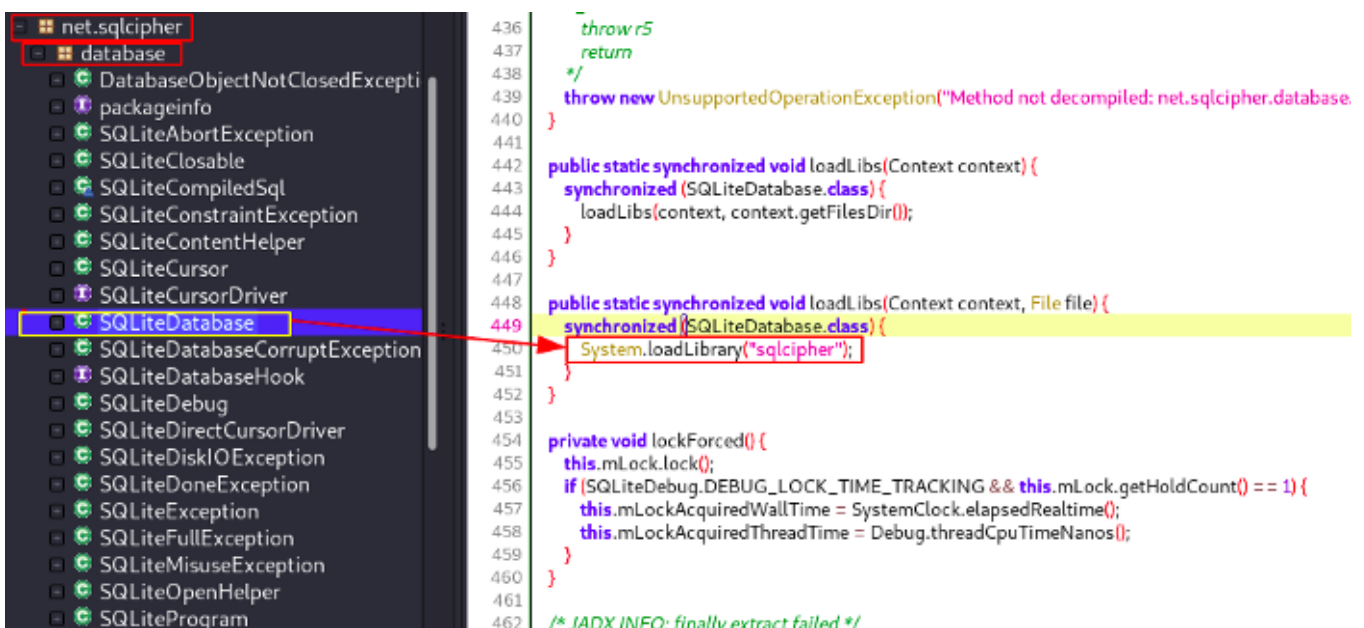
The alert message box says

“Could not open database file. Reason : file not a database”

That’s a strange behaviour, because a **.db** file is a database file.

Let’s check the source code file once again...

See ! What we got. The application loads a **SQLiteDatabase** native library.



```
463 | private void markTableSyncable(String str, String str2, String str3, String str4) {  
464 |     lock();
```

The above exhibit shows that the application uses native library in application source code

Here, we can see that the application has **SQLiteDatabase** class in **net.sqlcipher.database** package and they have used native library **System.loadLibrary("sqlcipher")** for encrypting the database.

So, it means the database is encrypted and we need to decrypt it now...



Now we can use **sqlcipher** commands for decrypting the database. But before that we need to install **sqlcipher** in our system.

A very simple basic command can be used to install **sqlcipher** in our system.

```
sudo apt-get install -y sqlcipher
```

After installing **sqlcipher**, we can go ahead and follow the below mentioned commands for decrypting the database.

```
saurabhjain@ubuntu:~$ sqlcipher q.db  
SQLCipher version 3.15.2 2016-11-28 19:13:37  
Enter ".help" for instructions  
Enter SQL statements terminated with a ";"  
sqlite> PRAGMA key = '123456';  
sqlite> ATTACH DATABASE 'plaintext.db' AS plaintext KEY '';  
sqlite> SELECT sqlcipher_export('plaintext');
```

```

sqlite> DETACH DATABASE plaintext;
sqlite> .tables
a
sqlite> .schema a
CREATE TABLE a(z VARCHAR,a VARCHAR);
sqlite> SELECT * FROM a;
4656|3531
sqlite>
sqlite>
sqlite> .exit

```

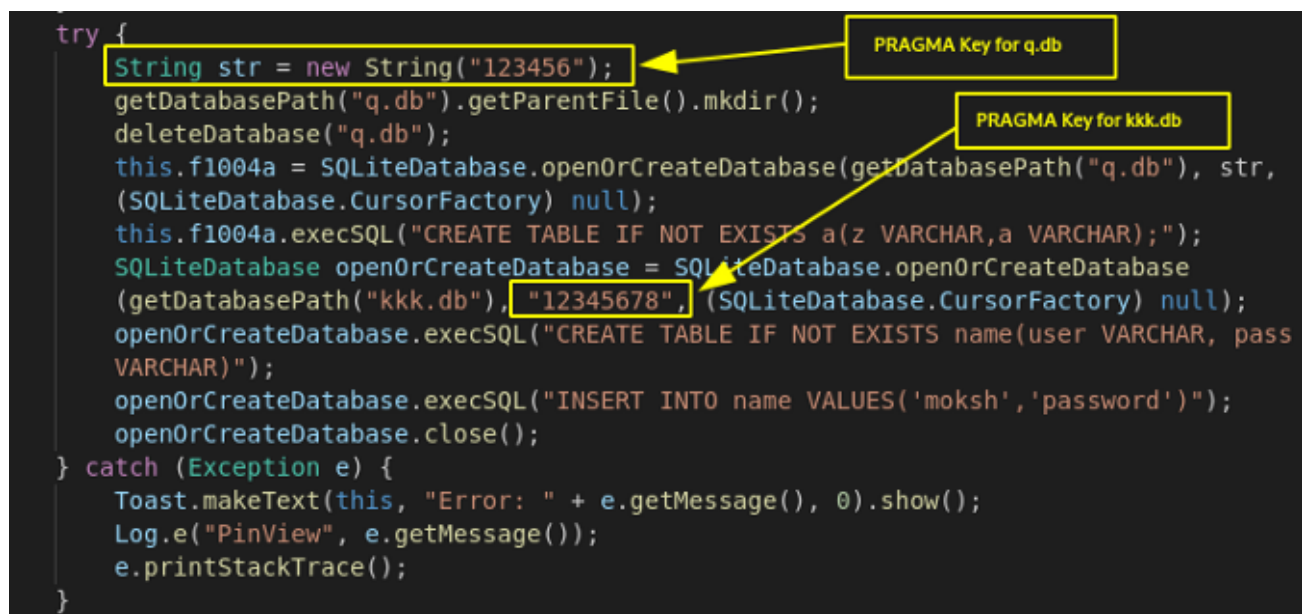
The above exhibit shows the database are decrypted using sqlcipher

Let's simplify each command one by one.

```
sqlite> PRAGMA key = '123456';
```

PRAGMA Key : Whenever a new encrypted database is created, we first need to create a key which is called “keying” the database. SQLCipher uses just-in-time key derivation at the point itNote : For pulling the database you either need a rooted device or the backup needs to be enabled and then pulling the data by adb is first needed for an operation. This means that the key must be set before the first operation on the database.

From the source code we can see that the key is “123456”



```

try {
    String str = new String("123456");
    getDatabasePath("q.db").getParentFile().mkdir();
    deleteDatabase("q.db");
    this.fl004a = SQLiteDatabase.openOrCreateDatabase(getDatabasePath("q.db"), str,
(SQLiteDatabase.CursorFactory) null);
    this.fl004a.execSQL("CREATE TABLE IF NOT EXISTS a(z VARCHAR,a VARCHAR);");
    SQLiteDatabase openOrCreateDatabase = SQLiteDatabase.openOrCreateDatabase
(getDatabasePath("kkk.db"), "12345678", (SQLiteDatabase.CursorFactory) null);
    openOrCreateDatabase.execSQL("CREATE TABLE IF NOT EXISTS name(user VARCHAR, pass
VARCHAR)");
    openOrCreateDatabase.execSQL("INSERT INTO name VALUES('moksh','password')");
    openOrCreateDatabase.close();
} catch (Exception e) {
    Toast.makeText(this, "Error: " + e.getMessage(), 0).show();
    Log.e("PinView", e.getMessage());
    e.printStackTrace();
}

```

The code snippet shows the initialization of two databases. The first database, `q.db`, is created with the key `"123456"`. The second database, `kkk.db`, is created with the key `"12345678"`. Annotations in the image point to these key strings with labels: "PRAGMA Key for q.db" and "PRAGMA Key for kkk.db".

The above exhibit shows the PRAGMA keys for the both the encrypted database

```
sqlite> ATTACH DATABASE 'plaintext.db' AS plaintext KEY '';
```

The above command indicates that a new database **plaintext.db** is being created with an empty key and attach to the present database i.e **q.db**

```
sqlite> SELECT sqlcipher_export('plaintext');
```

The above command explains that the plaintext database has been created and all the data of **q.db** is being exported to **plaintext.db**

```
sqlite> DETACH DATABASE plaintext;
```

The above command explains that the **plaintext.db** is being detached from the encrypted database **q.db**. Now **plaintext.db** is a separate entity and we can now read the database in plaintext from **plaintext.db**

Now running the most common and famous SQL command.

```
sqlite> SELECT * FROM a;
```

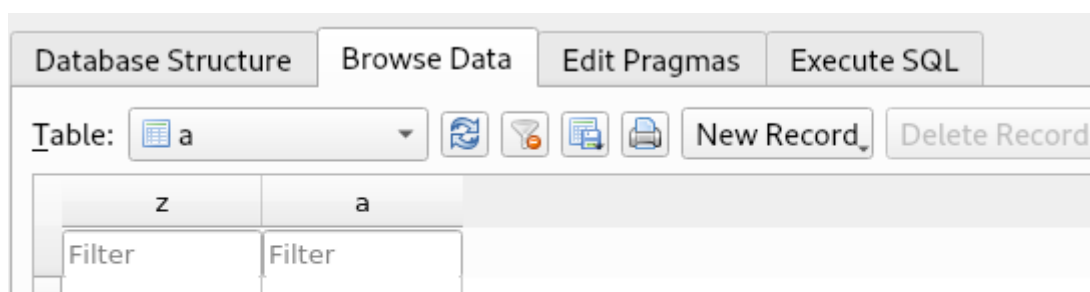
```
saaurabhjain@kali:~$ sqlcipher q.db
SQLCipher version 3.15.2 2016-11-28 19:13:37
Enter ".help" for instructions
Enter SQL statements terminated with a ";"
sqlite> PRAGMA key = '123456';
sqlite> ATTACH DATABASE 'plaintext.db' AS plaintext KEY '';
sqlite> SELECT sqlcipher_export('plaintext');

sqlite> DETACH DATABASE plaintext;
sqlite> .tables
a
sqlite> .schema a
CREATE TABLE a(z VARCHAR,a VARCHAR);
sqlite> SELECT * FROM a;
4656|3531
sqlite>
sqlite>
sqlite> .exit
```

The above exhibit shows the output of the command entered in sqlite mode

The results come up with two different numbers. We can simply enumerate the numbers and find the exact pin.

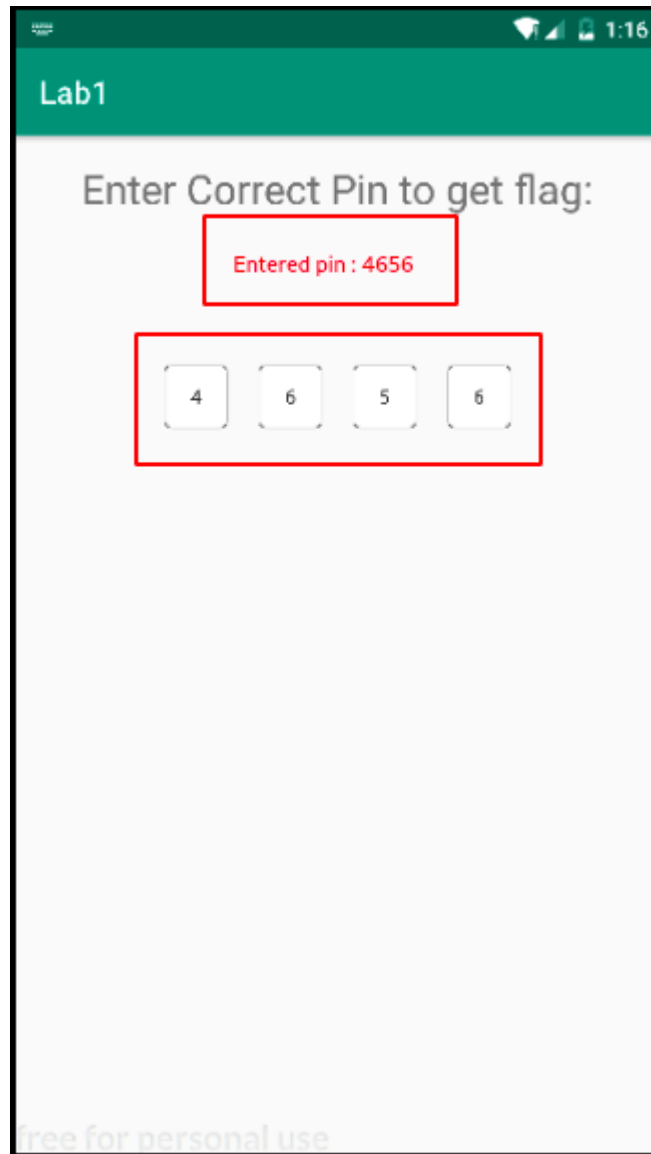
We can also see the output in **sqlitebrowser** for our ease now.



1	4656	3531
---	------	------

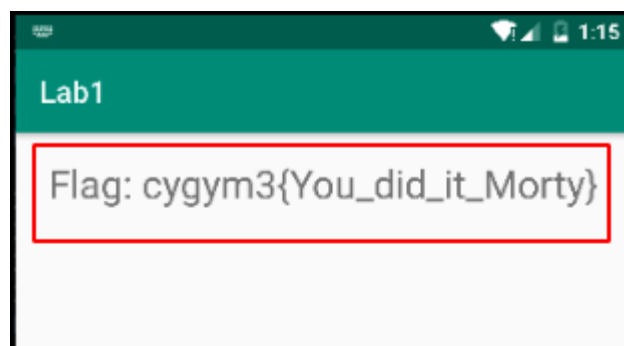
The above exhibit shows the data of encrypted database after decrypting it using sqlcipher

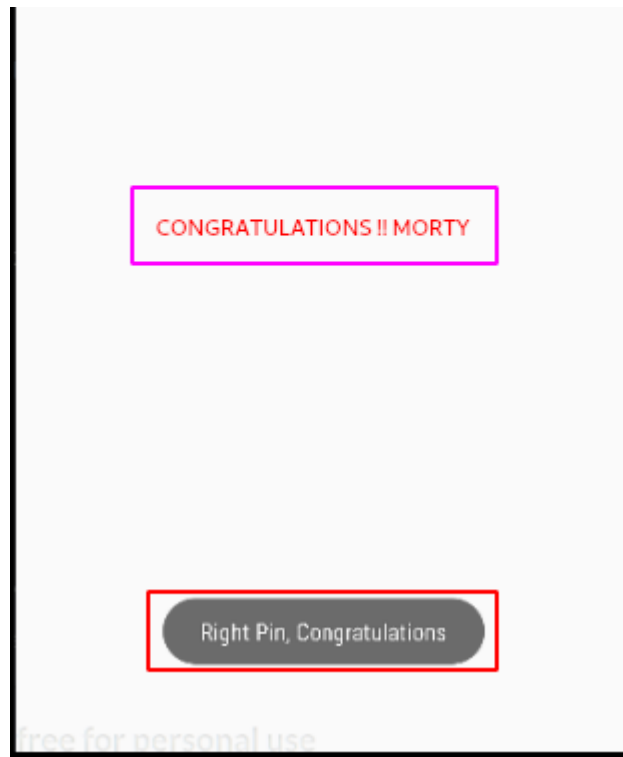
Now entering the pin “4656”



The above exhibit shows that the alternatively the PINs are being entered

and the flag is...





The above exhibit shows the flag

Yes Morty ! You finally did it !

Let's talk about the second approach to solve the challenge.

The second approach requires some knowledge of coding and reading the reverse engineered code. A bit of more manual efforts will be required for solving the challenge.

Just stay connected!

So, while traversing and reading the reverse engineered code, we encountered an eye catching statement for key.

```
MainActivity.this.b++;
edit.putInt("val", MainActivity.this.b);
edit.commit();
String value = pinview.getValue();
MainActivity mainActivity = MainActivity.this;
mainActivity.f1003a = mainActivity.f1004a.rawQuery("SELECT * FROM a;", (String[]) null);
mainActivity.this.f1003a.moveToFirst();
if (value.equalsIgnoreCase(MainActivity.this.f1003a.getString(0))) {
    Toast.makeText(MainActivity.this, "Right Pin, Congratulations", 0).show();
    pinview2.removeAllViews();
    String string = MainActivity.this.getResources().getString(R.string.google_api_key);
    new ja();
    String a2 = ja.a(ja.a(string, string.substring(4)));
    String b = ja.b(a2.substring(1), a2);
    b.substring(2);
    textView.setText("Flag: ".concat(String.valueOf(ja.c(b, b))));
```


The above code snippet gives us a hint for finding the flag in application source code

So, here we can see the statement

Right Pin, Congratulations

and below that there is a key i.e *google_api_key* being called.

```
String string = MainActivity.this.getResources().getString(R.string.google_api_key);
```

We can find the value of *google_api_key* in “/res/values/strings.xml”

```
<string name="app_name">Lab1</string>
<string name="google_api_key">R4f/mz5cIi2NHsrnAUGqYWThCTg60fHF1xYUZt73KXxS/
mHJwYl41hcJ8R3rvAzuu9MUguemAhc8yjdjifc+WiY9oVKZyN9xfscD95b9BDI=</string>
<string name="main_msg">Enter Correct Pin to get flag:</string>
<string name="search_menu_title">Search</string>
<string name="status_bar_notification_info_overflow">999+</string>
```

The above exhibit shows that the key value has been hard coded in application source code

and an another eye catching statement in *MainActivity.java*

```
MainActivity.this.f1003a.moveToFirst();
if (value.equalsIgnoreCase(MainActivity.this.f1003a.getString(0))) {
    Toast.makeText(MainActivity.this, "Right Pin, Congratulations", 0).show();
    pinview2.removeAllViews();
    String string = MainActivity.this.getResources().getString(R.string.
google_api_key);
    new ja();
    String a2 = ja.a(ja.a(string, string.substring(4)));
    String b = ja.b(a2.substring(1), a2);
    b.substring(2);
    textView.setText("Flag: ".concat(String.valueOf(ja.c(b, b))));
} else {
    Toast.makeText(MainActivity.this, "Incorrect Pin, " + ((MainActivity.this.c +
1) - MainActivity.this.b) + " attempts remaining", 0).show();
}
MainActivity.this.f1003a.close();
```

The above exhibit shows function call for finding the value of flag

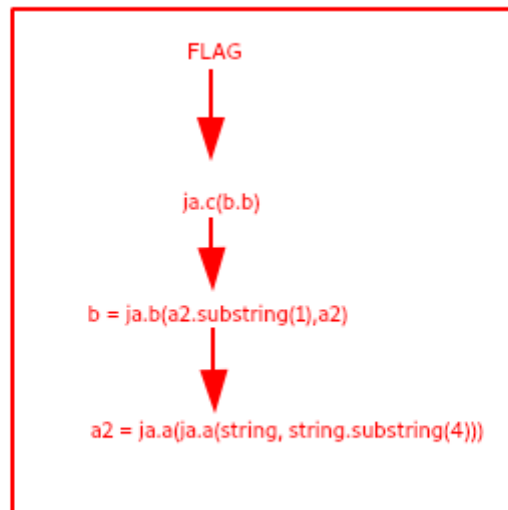
This clearly indicates that the function *ja.c(b, b)* plays the game.

Let's trace back the code...

While reverse engineering and tracing the code down, we can see the five functions are being called in sequence with the string value of *google_api_key*.

- *public static String a(String str)*
- *public static String a(String str, String str2)*
- *private static String b(String str)*
- *public static String b(String str, String str2)*
- *public static String c(String str, String str2)*

Lets understand the flow like this...



The above exhibit shows the flow of function calls for breaking the logic of finding the flag

String string = google_api_key;

String a2 = ja.a(ja.a(string, string.substring(4))); → ja.a(string, string.substring(4))

String b = ja.b(a2.substring(1), a2); → a2.substring(1), a2

String Flag = ja.c(b, b)

Replicating the complete code in Android studio..

```

public class MainActivity extends AppCompatActivity {

    String string1, string2, string3, string4, string5, string6, string7;

    @Override
    protected void onCreate(Bundle savedInstanceState) {
        super.onCreate(savedInstanceState);
        setContentView(R.layout.activity_main);
        String google_api_key = "R4f/mz5cIi2NHsrrnAUGqYwThCTg60fHF1xYUZt73KXxS/mHJwYl41hcJ8R3rvAzuu9MUguemAhc8ydjifc+Wi";
        string1 = google_api_key;
        string2 = string1.substring(4);
        string3 = a(string1, string2);
    }
}
  
```

```

string4 = a(string3);
string5 = string4.substring(1);
string6 = b(string5, string4);
string7 = c(string6, string6);

System.out.println("String 1 : " + string1);
System.out.println("String 2 : " + string2);
System.out.println("String 3 : " + string3);
System.out.println("String 4 : " + string4);
System.out.println("String 5 : " + string5);
System.out.println("String 6 : " + string6);
System.out.println("String 7 : " + string7);
System.out.println("Flag : " + string7);
}

public static String a(String str, String str2) {
    try {
        MessageDigest instance = MessageDigest.getInstance("SHA-1");
        str.getBytes("UTF-8");
    }
}

```

The above exhibit shows the code snippet of the function calls responsible for finding the flag

Let's watch out the logcat for the FLAG..

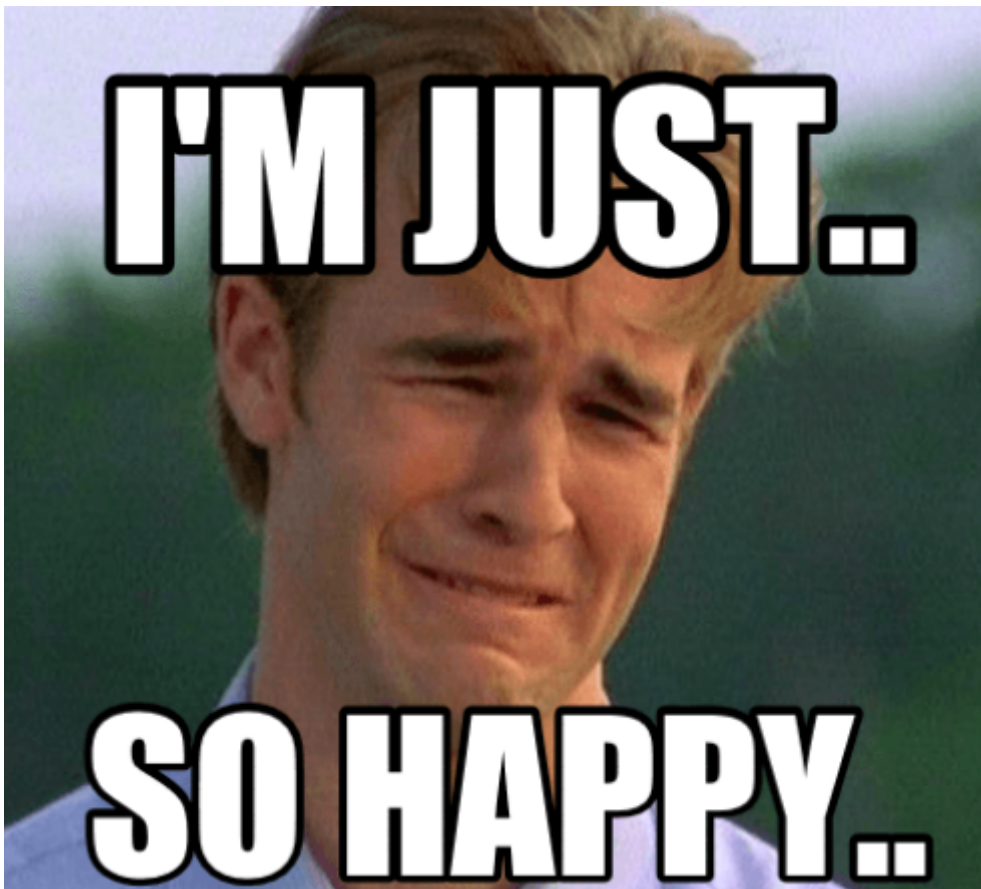
```

I/art: Rejecting re-init on previously-failed class java.lang.Class<androidx.core.view.ViewCompat$2>
I/System.out: String 1 : R4f/mz5cIi2NHsrnAUGqYwThCTg60fHF1xYUZt73KXxS/mHJwYl41hcJ8R3rvAzuu9MUguemAhc8yjdjifc+WiY9oVKZyN9xfsc0D95b9BDI=
I/System.out: String 2 : mz5cIi2NHsrnAUGqYwThCTg60fHF1xYUZt73KXxS/mHJwYl41hcJ8R3rvAzuu9MUguemAhc8yjdjifc+WiY9oVKZyN9xfsc0D95b9BDI=
I/System.out: String 3 : R4f/mz5cIi2NHsrnAUGqYwThCTg60fHF1xYUZt73KXxS/mHJwYl41hcJ8R3rvAzuu9MUguemAhc8yjdjifc+WiY9oVKZyN9xfsc0D95b9BDI=
I/System.out: String 4 : NjM3OTY3Nzk2ZDMzN2I1OTZmNzU1ZjY0Njk2NDVmNjk3NDVmNGQ2ZjcyNzQ3OTdk
I/System.out: String 5 : jM3OTY3Nzk2ZDMzN2I1OTZmNzU1ZjY0Njk2NDVmNjk3NDVmNGQ2ZjcyNzQ3OTdk
I/System.out: String 6 : 637967796d337h596f755f6469645f69745f4d6f7274797d
I/System.out: String 7 : cygym3{You_did_it_Morty}
I/System.out: Flag : cygym3{You_did_it_Morty}
D/OpenGLRenderer: Use EGL_SWAP_BEHAVIOR_PRESERVED: true
D/libEGL: loaded /system/lib/egl/libEGL_emulation.so

```

The above exhibit shows the value of the flag in android studio logcat

Yes Morty ! Finally you did it again.



So, for this way you don't need any rooted device to extract the database and find out the pin.

As the pin was random and getting created at run time but the flag was static.

Takeaways

Learned how to reverse engineer android application

Learned how to read the application source code

Decrypted a sqlcipher encrypted database

Harm caused by hard coding keys

Patience is the key

[Ctf Writeup](#)[Android](#)[Pentesting](#)[Sqlcipher](#)[Android Security](#)[About](#) [Help](#) [Legal](#)

Get the Medium app

