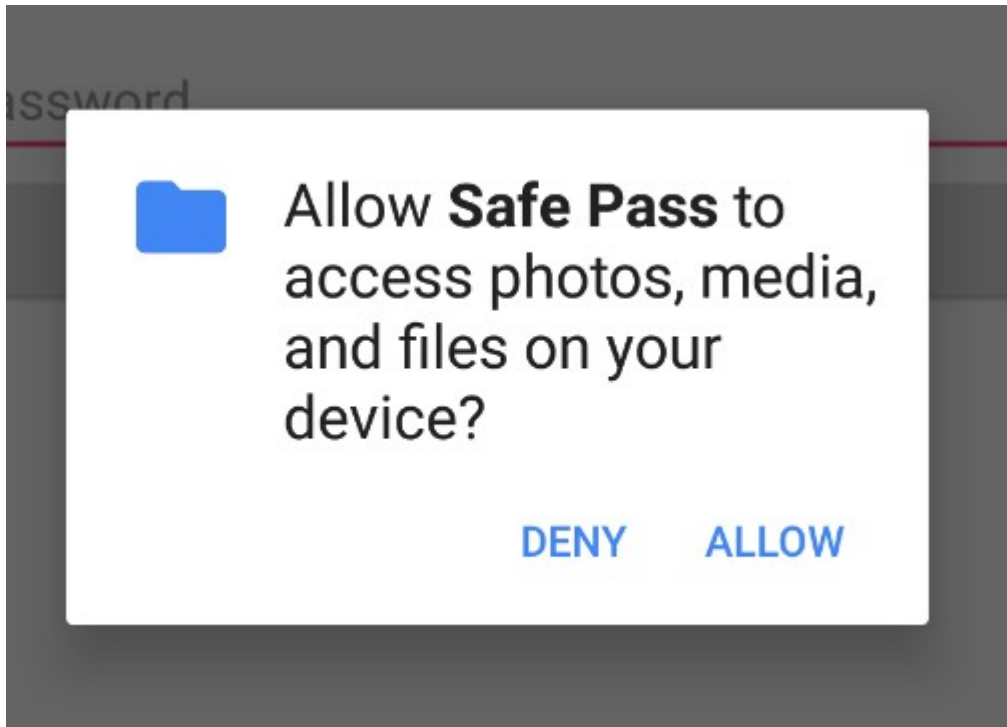
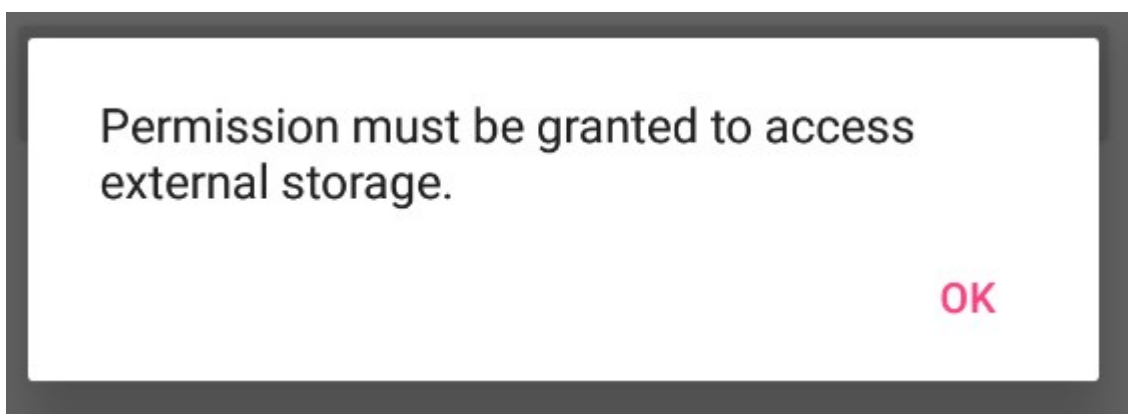


Setting Up / Prerequisites

After installing the APK, a new application named "Safe Pass" will be installed. Launching this app will firstly prompt the user to allow access to the external storage. The reason for this, is so that it can write to a globally available database; to demonstrate the purposes of working outside the application's sandbox.



Should a user deny the request, the application will exit, indicating that it needs the permission to run:



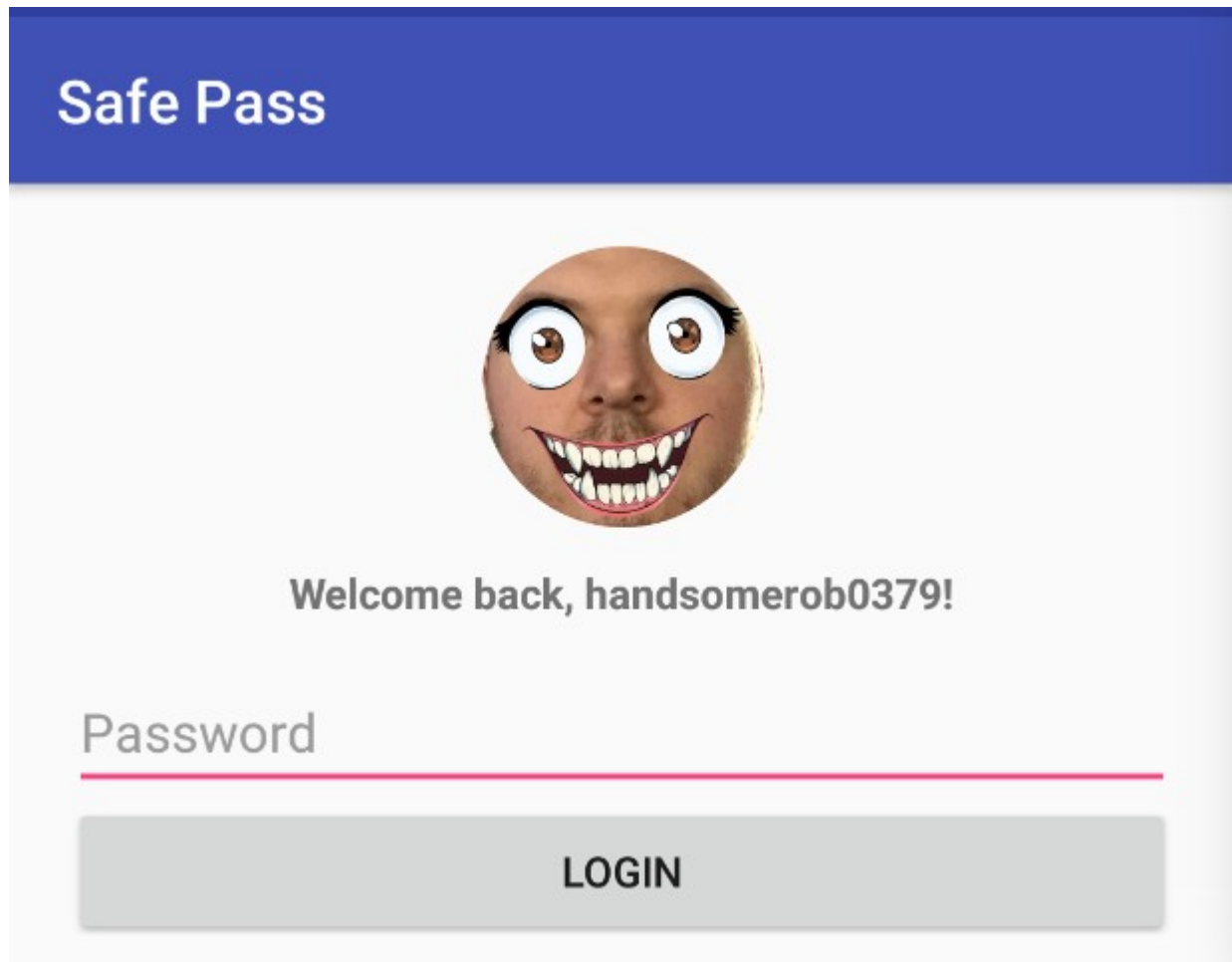
The entire challenge can be finished using:

- An Android device / emulator

- A decompiler, such as jadx - <https://github.com/skylot/jadx>
- The sqlite CLI tool or a GUI, such as DB Browser for SQLite - <http://sqlitebrowser.org/>
- A tool for performing encryption and encoding operations, such as cyberchef - <https://gchq.github.io/CyberChef/>

Step 1 - Initial Analysis

After getting the application set up, the first challenge that is presented is the login screen. A message is displayed welcoming back the user, and prompts solely for a password. At this point, one can either attempt to brute force the password (and have a small success rate), or start to delve into some OSINT, to see what can be gathered on the username **handsomerob0379**.

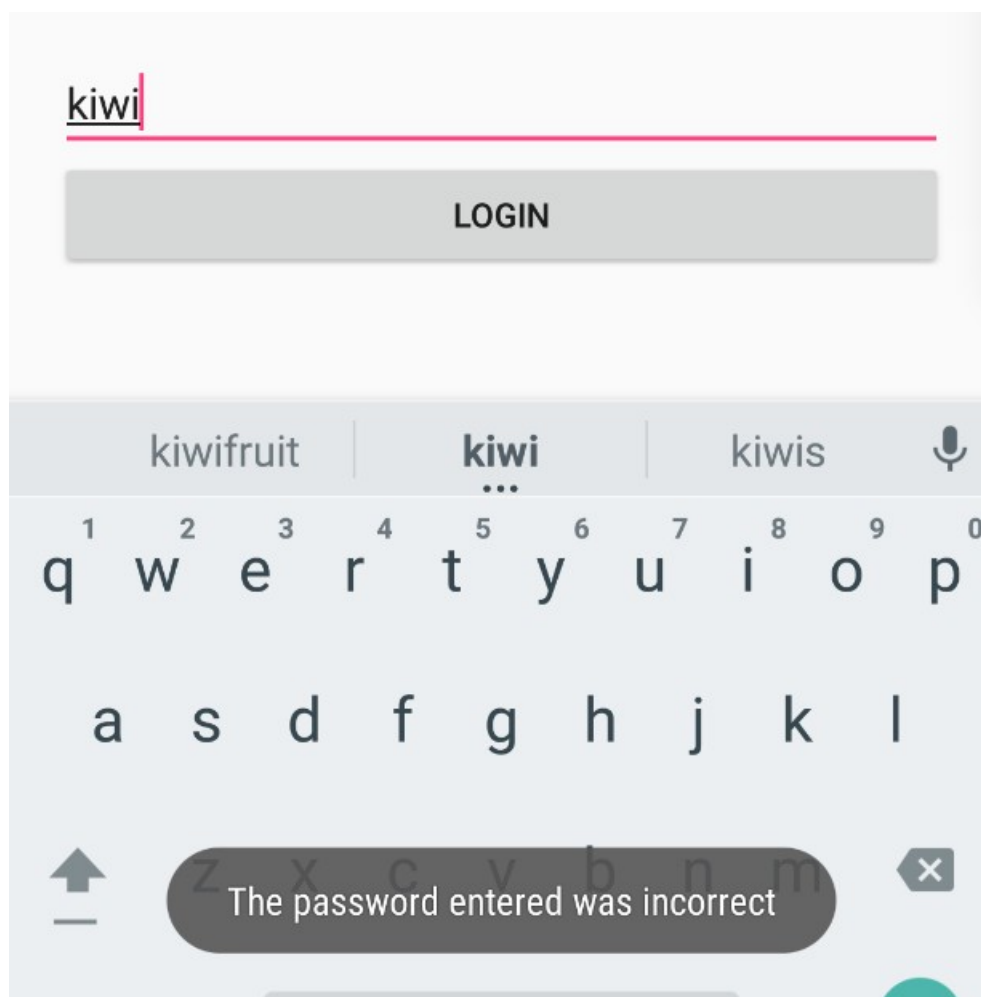
The image shows a login interface with a blue header bar containing the text "Safe Pass" in white. Below the header is a large, circular, cartoonish face with wide eyes and a wide, toothy grin. Underneath the face, the text "Welcome back, handsomerob0379!" is displayed in a bold, black font. Below this is a password input field with the label "Password" in a grey font. A pink horizontal line is positioned below the label. At the bottom of the form is a large, grey, rounded rectangular button with the text "LOGIN" in a bold, black font.

[OSINT Framework](#) will provide a significant amount of tools for carrying out reconnaissance on usernames. Running the username through one of these tools, [namechk](#), will show that the username has been reused, with a few false-positives, and provide links to the profiles it finds. One of the profiles found, will be a Twitter account (<https://twitter.com/handsomerob0379>), which seems to be owned by the same user, due to the same / scary profile picture.

Step 2 - OSINT & Password Building

With the user's Twitter profile now identified, some educated guesses can start to be made based on the posting habits of the user. Although users are forced by a lot of systems to make their passwords more complex by adding a mix of numbers, symbols etc. a common practise is for users to just stick with memorable patterns; which whilst mathematically strong, will significantly reduce the strength of the password against a targetted attack.

An initial look at the user's Twitter profile reveals a [slightly strange] passion for kiwi fruit; which means it's viable that the password will contain some kind of reference to it. Attempting to login with just the words "kiwi" or "kiwifruit" will not yield anything:




Most likely, a password will need to meet a minimum strength; this is usually done by forcing numbers or symbols to be present in the password. As most people have a numeric sequence that will be memorable to them (birth date, a pin code for building entry, phone number etc.), they tend

to lean towards reusing these sequences in passwords. Circling back to the user's username, presents a number that may be usable.

Trying to login with the password **kiwi0379** will work and then present the 2FA screen:

Safe Pass



Welcome back, **handsomerob0379!**

You enabled 2FA during initial setup. To login, enter the value displayed on your 2FA token.

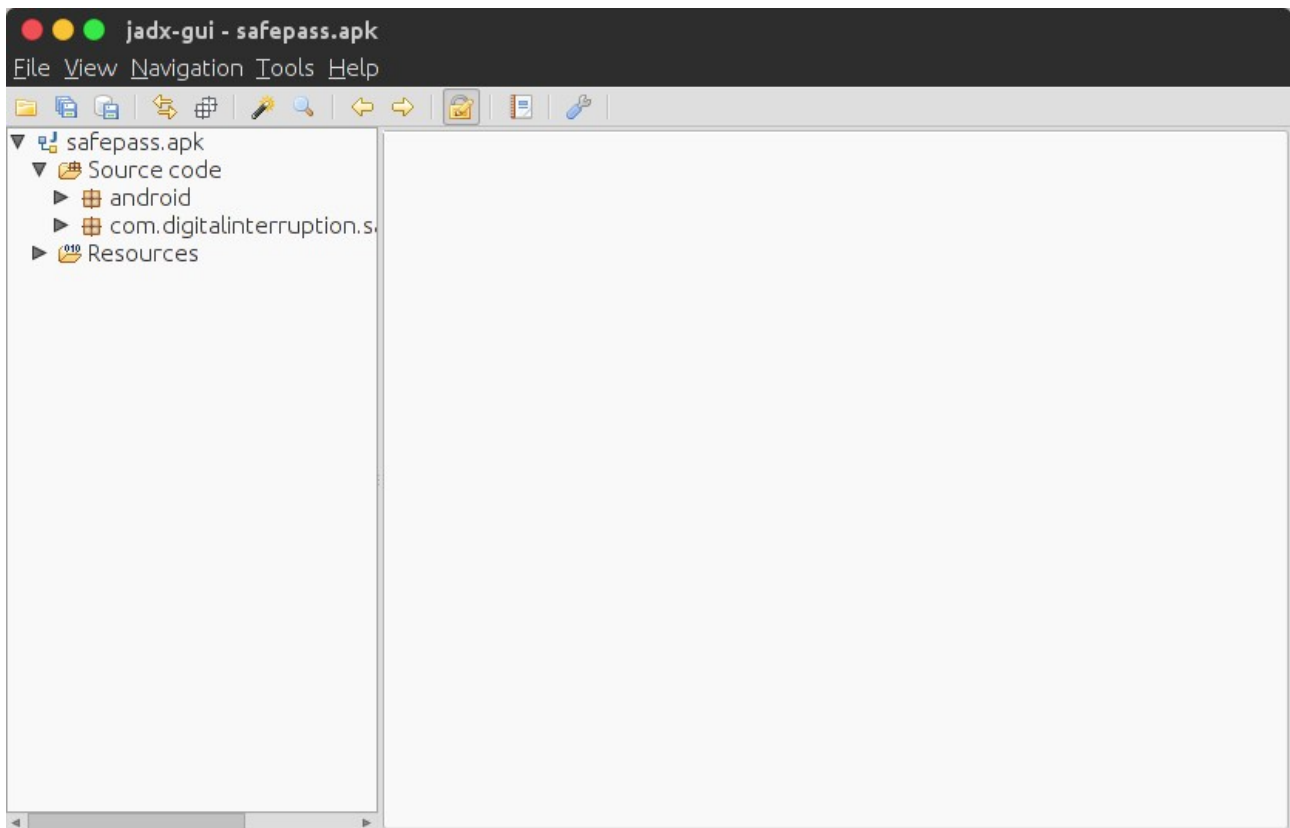
2FA Token

LOGIN

Step 3 - Source Code Analysis of Login Procedure

Upon arriving at the 2FA screen, it will become evident that logging in to the application is not going to be possible. The next step is to start looking into the source code that the application is built upon.

A number of tools exist to aid in this, but for the purpose of this document, **jadx** will be used. When launching jadx, it will instantly display a prompt to open a file, whilst here, choose the Safe Pass APK file. Shortly after opening the file, the program will load with the source code tree visible:



Expanding the **com.digitalinterruption.*** tree will reveal a number of classes; including a **LoginActivity** class. Inspecting this will reveal the code that is executed upon clicking the login button:

```

1 public void onClick(View view) {
2     if (view != this.mLoginButton) {
3         return;
4     }
5     if (this.mHasValidPassword) {
6         this.mHasValidPassword = false;
7         toggleControls(true);
8         Toast.makeText(this, "The token entered was incorrect", 1).show();
9         return;
10    }
11    login();
12 }

```

On line 5, it checks if the **mHasValidPassword** variable is set to true, if it is, it hits the 2FA blockade; otherwise, it will run the **login** method:

```

1 private void login() {
2     if (checkPassword(this.mPasswordEditText.getText().toString())) {
3         this.mHasValidPassword = true;
4         toggleControls(false);
5         return;
6     }
7     Toast.makeText(this, "The password entered was incorrect", 1).show();
8 }

```

This method passes the contents of the password input to the **checkPassword** method, and if the return value is true, sets the **mHasValidPassword** variable that was previously seen in the click handler.

```

1 public boolean checkPassword(String password) {
2     if (password.trim().equals("")) {
3         return false;
4     }
5     return new CryptoHandler(password)
6         .decrypt(getAuthenticationToken()).equals("DI{Th15_u53r_15_l0gg3d_1n}");
7 }

```

The **checkPassword** method now begins to reveal a bit more about the inner workings of the application. Assuming the user does not enter a blank password, it will use the password to construct a "CryptoHandler" object, which then decrypts the value returned by a method named

`getAuthenticationToken`, and checks the result against a hard coded string; if the value is the same, the user is now considered logged in.

```
1 private String getAuthenticationToken() {
2     Cursor cursor = this.mDatabase.query(
3         "settings", new String[]{"key", "value"}, "key = ?", new String[]{"aes_token"}, null, null, null
4     );
5     String retval = null;
6     if (cursor.moveToFirst()) {
7         retval = cursor.getString(1);
8     }
9     cursor.close();
10    return retval;
11 }
```

The `getAuthenticationToken` method reveals two important pieces of information required to further reverse the application:

1. The application is using an SQLite database (identified by checking the declaration of `mDatabase`)
2. The encryption in use appears to be AES, based on it trying to retrieve a value from the database with the key "aes_token"

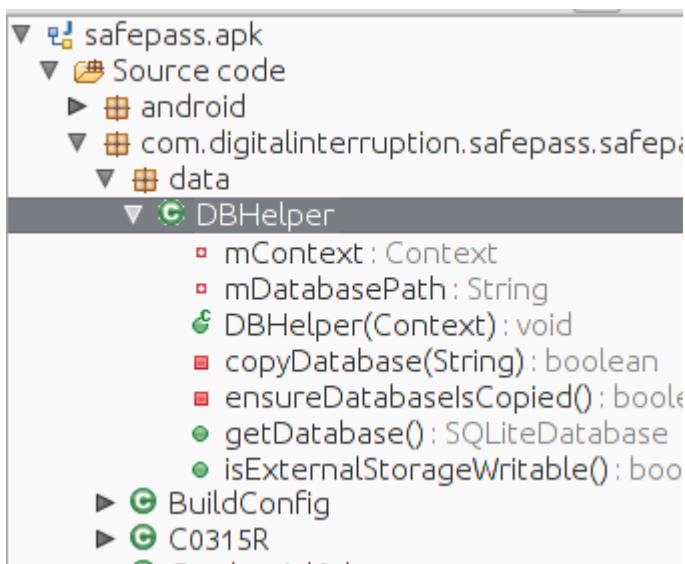
Step 4 - Finding The Database

As an SQLite database is in use, and the application previously prompted to utilise the external storage, it would be reasonable to assume there's a chance the database may be being stored in the external storage.

Examining the **setupDatabase** method of **LoginActivity** shows that the **mDatabase** variable is initialised using an instance of the **DBHelper** class:

```
1 private void setupDatabase() {
2     try {
3         this.mDatabase = new DBHelper(this).getDatabase();
4     } catch (Exception e) {
5         e.printStackTrace();
6         Builder builder = new Builder(this);
7         builder.setMessage((CharSequence) "An error occurred setting up the database.");
8         builder.setPositiveButton((CharSequence) "OK", new C03111());
9         builder.show();
10    }
11 }
```

Returning back to the source code tree, and expanding the **data** package will reveal the **DBHelper** class and allow for further analysis:



The method of the **DBHelper** class that was being used to acquire a connection was the **getDatabase** method; this method consists of only a single line, which opens the database from the path stored in the **mDatabasePath** variable:

```

1 public SQLiteDatabase getDatabase() {
2     return SQLiteDatabase.openDatabase(this.mDatabasePath, null, 0);
3 }

```

Continuing to analyse the **DBHelper** class will show that there is only a single point of initialisation for the **mDatabasePath** variable; which is found within the **ensureDatabasesCopied** method.

```

1 private boolean ensureDatabaseIsCopied() {
2     if (!isExternalStorageWritable()) {
3         return false;
4     }
5     File destFolder = new File(
6         Environment.getExternalStoragePublicDirectory(Environment.DIRECTORY_DOCUMENTS), "safepass"
7     );
8     if (destFolder.exists() || destFolder.mkdirs()) {
9         File destFile = new File(destFolder, "safepass.db");
10        this.mDatabasePath = destFile.getAbsolutePath();
11        if (destFile.exists()) {
12            return true;
13        }
14        try {
15            destFile.createNewFile();
16            return copyDatabase(this.mDatabasePath);
17        } catch (IOException e) {
18            e.printStackTrace();
19            return false;
20        }
21    }
22    Log.e("safepass", "Failed to create folder");
23    return false;
24 }

```

By taking a look at the implementation of this method, it is possible to determine where the database is stored. On line 5, the **Environment.getExternalStoragePublicDirectory** method is called with the **DIRECTORY_DOCUMENTS** enum. This method will return the "Documents" directory within the path to the storage area that is world writeable on the device; typically found at **/mnt/sdcard**.

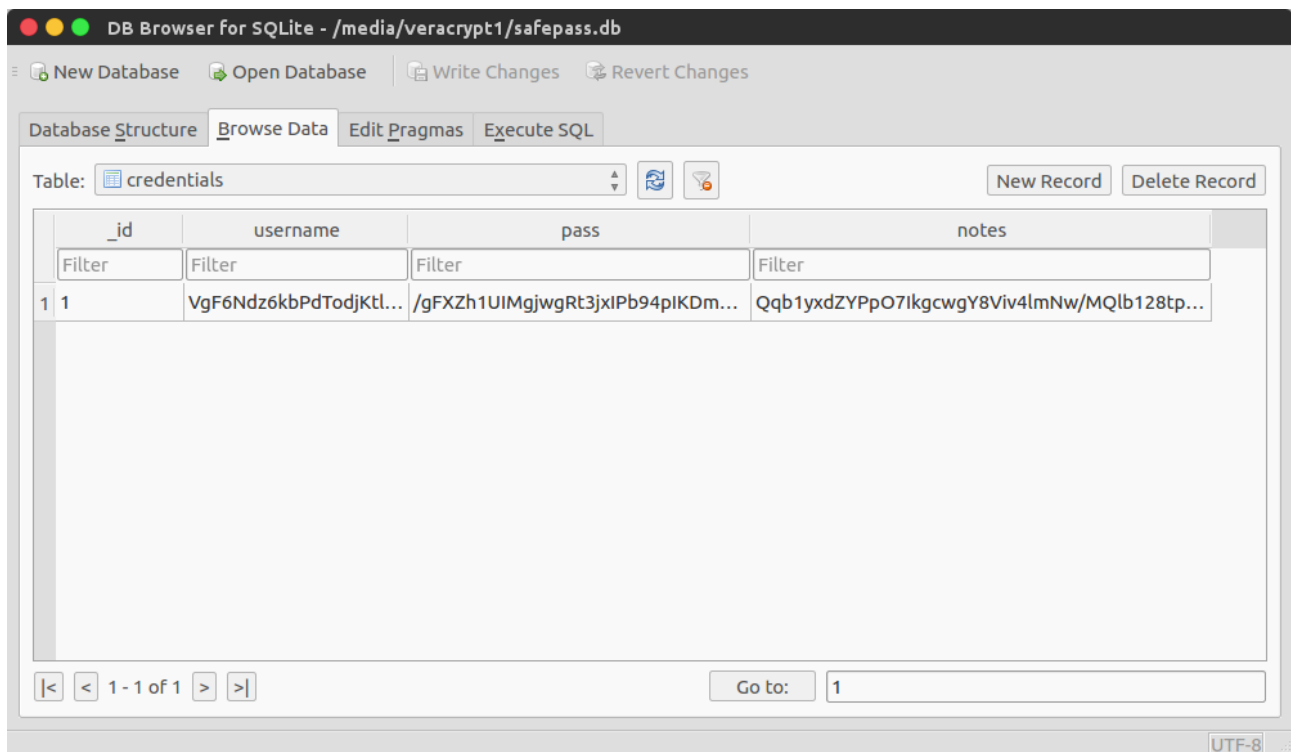
Verification of this discovery can be executed by getting a shell on to the device, using **adb**, and then listing the contents of the **/mnt/sdcard/** directory. Alternatively, if using a physical device - connecting it via USB and browsing the storage should reveal the same information.

```
$ adb shell
generic_x86:/ $ ls -lah /mnt/sdcard/Documents/safepass
total 20K
drwxrwx--x 2 root sdcard_rw 4.0K 2018-03-14 14:34 .
drwxrwx--x 3 root sdcard_rw 4.0K 2018-03-13 18:47 ..
-rw-rw---- 1 root sdcard_rw 5.0K 2018-03-14 17:00 safepass.db
-rw-rw---- 1 root sdcard_rw 3.5K 2018-03-14 17:00 safepass.db-journal
generic_x86:/ $
```

If mounting the device's storage is not possible, but adb is usable (for example, if using an emulator), the database can be pulled back using the **adb pull** command as can be seen below:

```
$ adb pull /mnt/sdcard/Documents/safepass/safepass.db
/mnt/sdcard/Documents/safepass/safepass.db: 1 file pulled. 0.1 MB/s (5120 bytes in 0.039s)
$
```

Once the database has been acquired, the [encrypted] data stored within it can be openly viewed using any application capable of opening SQLite databases:



Step 5 - Decrypting The Data

The final step of the challenge is to decrypt the data found within the recovered SQLite database. The information gathered throughout the challenge up to this point has led to the knowledge that the data seems to be being encrypted using the AES algorithm.

There are a number of ways that AES data can be encrypted / decrypted, a particularly simple method of doing this, is using CyberChef (<https://gchq.github.io/CyberChef/>). However, as one will notice, to use the AES Decrypt recipe, both a "key" and "iv" must be specified; so further analysis of the source code will be required to see how these values are generated.

Previously, the CryptoHandler class could be found being instantiated using the password that the user supplied when they logged in, which is passed to the **secretKey** variable of the constructor method found below:

```
1 public CryptoHandler(String secretKey) {
2     try {
3         this.mKey = (secretKey + "0000000000000000").substring(0, 16).getBytes("UTF8");
4         this.mIV = "itsasecret000000".getBytes("UTF8");
5     } catch (UnsupportedEncodingException e) {
6         e.printStackTrace();
7     }
8 }
```

Within this method, the **mKey** variable is set on line 3, and the **mIV** variable is set on line 4 - both of which seem to be the key and IV that are used during the AES operations.

On line 3, the user's password is padded with 16 zeros, and then the value is cut down to the first 16 characters; meaning that the AES key would be **kiwi037900000000**.

On line 4, the IV is simply hard coded as **itsasecret000000**.

Using this information, a recipe can be put together on CyberChef to decrypt the data:

[https://gchq.github.io/CyberChef/#recipe=AES_Decrypt\(%7B'option':'UTF8',string:'kiwi037900000000'%7D,%7B'option':'UTF8',string:'itsasecret000000'%7D,'CBC','Raw','Raw',%7B'option':'Hex',string:'"%7D\)&input=L2dGWFpoMVVJTWdqd2dSdDNqeElQYjk0cElLRG1jYmIX](https://gchq.github.io/CyberChef/#recipe=AES_Decrypt(%7B'option':'UTF8',string:'kiwi037900000000'%7D,%7B'option':'UTF8',string:'itsasecret000000'%7D,'CBC','Raw','Raw',%7B'option':'Hex',string:')

In this recipe, the key and IV values of the AES Decrypt operation are set to the values determined above, and their type set to UTF8, and the input is set to the value found in the **pass** column of the **credentials** table in the database.

Unfortunately, this will yield an error, indicating the data still cannot be decrypted:

Recipe	Input
AES Decrypt	length: 44 lines: 1 /gFXZh1UIMgjwgRt3jxIPb94pIKDmcbiW8AghzmWcFA=
Key UTF8 kiwi037900000000	
IV UTF8 itsasecret000000	
Mode CBC Input Raw	
Output Raw	
GCM Tag Hex	
	start: 46 time: 0ms end: 46 length: 46 length: 0 lines: 1
	Unable to decrypt input with these parameters.

One final look at the CryptoHandler class, in particular the **encrypt** method, will reveal that after the data is being encrypted, it is being Base64 encoded:

```
1 public String encrypt(String message) {
2     try {
3         byte[] srcBuff = message.getBytes("UTF8");
4         SecretKeySpec keySpec = new SecretKeySpec(this.mKey, "AES");
5         IvParameterSpec ivSpec = new IvParameterSpec(this.mIV);
6         Cipher ecipher = Cipher.getInstance("AES/CBC/PKCS7Padding");
7         ecipher.init(1, keySpec, ivSpec);
8         return Base64.encodeToString(ecipher.doFinal(srcBuff), 0);
9     } catch (Exception ex) {
10        ex.printStackTrace();
11        return "";
12    }
13 }
```

In order to deal with this, the Base64 Decode operation can be added prior to the AES Decrypt operation, which will chain the result of the first operation to the second as its input:

[https://gchq.github.io/CyberChef/#recipe=From_Base64\('A-Za-z0-9%2B/%3D',true\)AES_Decrypt\(%7B'option':'UTF8','string':'kiwi037900000000'%7D,%7B'option':'UTF8','string':'itsasecret000000'%7D,'CBC','Raw','Raw',%7B'option':'Hex','string':'"%7D\)&input=L2dGWFpo](https://gchq.github.io/CyberChef/#recipe=From_Base64('A-Za-z0-9%2B/%3D',true)AES_Decrypt(%7B'option':'UTF8','string':'kiwi037900000000'%7D,%7B'option':'UTF8','string':'itsasecret000000'%7D,'CBC','Raw','Raw',%7B'option':'Hex','string':')

Now, when running the recipe on CyberChef, the decrypted password can be seen in the output pane:

Recipe	Input
From Base64 Alphabet: A-Za-z0-9+/= <input type="checkbox"/> Remove non-alphabet chars: <input checked="" type="checkbox"/>	/gFXZh1UIMgjwgRt3jxIPb94pIKDmcbiW8AghzmWcFA=
AES Decrypt Key: UTF8 <input type="text" value="kiwi037900000000"/> IV: UTF8 <input type="text" value="itsasecret000000"/> Mode: CBC <input type="text" value="Raw"/> Output: Raw <input type="text" value="Hex"/> GCM Tag: Hex <input type="text"/>	Output start: 21 end: 21 length: 0 time: 1ms length: 21 lines: 1 DI{k3y_t0_ev3ryth!ng}

This CyberChef recipe can then be used to decrypt all remaining data found within the application's database.