Mobile Security Package

- Varun S (21pc26)
- Vishnupriya R (21pc39)

The CTF box provided works like an escape room. The concept is that there are hints provided to guess the name of the vulnerability, and if you find out what vulnerability is used, you will subsequently be asked to find a flag by exploiting the vulnerability that you have found.

FLAG 1:

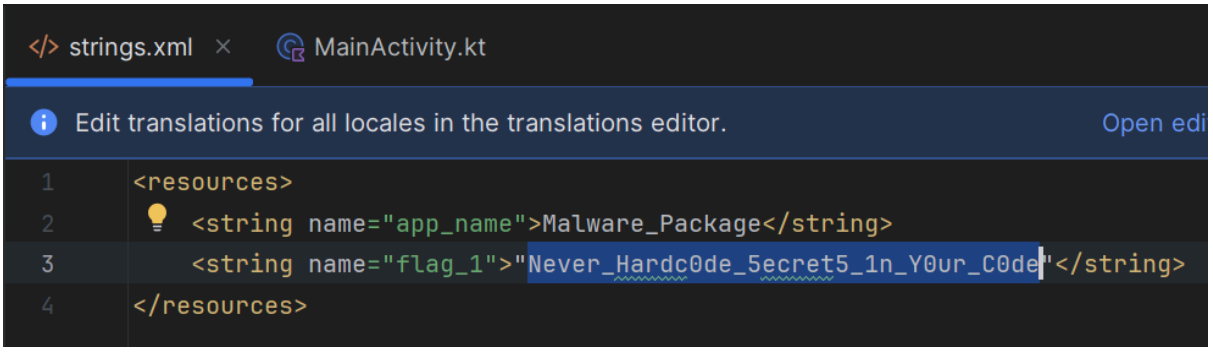
At first the user is hit with the welcome screen and asked to find out the first flag and enter the flag to continue to the first challenge.

While inspecting the flow of the app, it can be seen that the input box uses a flag checking function which reveals that the flag is stored in R.strings.flag_1.

```
val flag1Inp = flag1_tb.text.toString()
val correctFlag = getString(R.string.flag_1)

if (flag1Inp == correctFlag) {
```

Thus, looking at values in res/values/strings.xml, it can be found that the first flag is "**Never_Hardc0de_5ecret5_1n_Y0ur_C0de**"

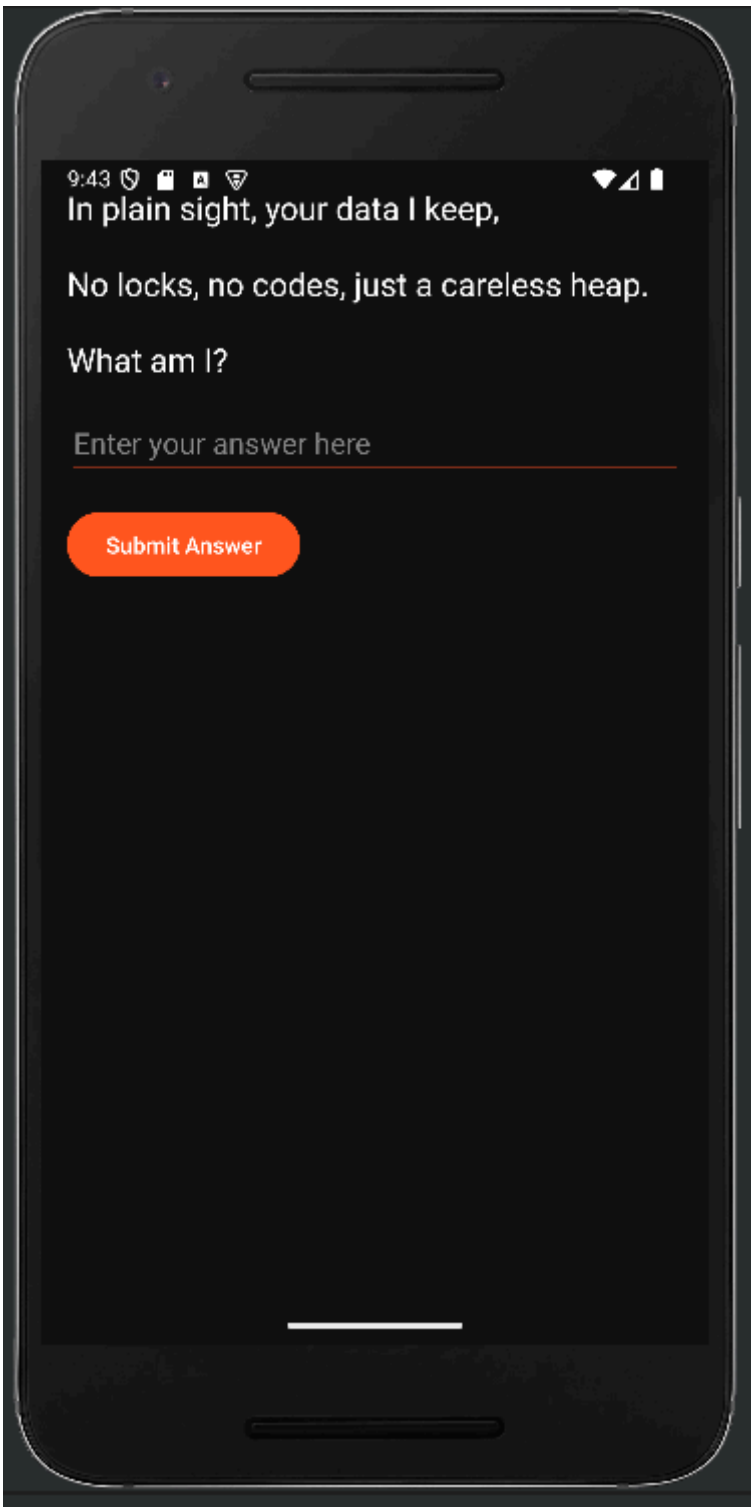


Once the flag is entered in the input box, and submit is clicked, we are presented with our first question.

FLAG 2:

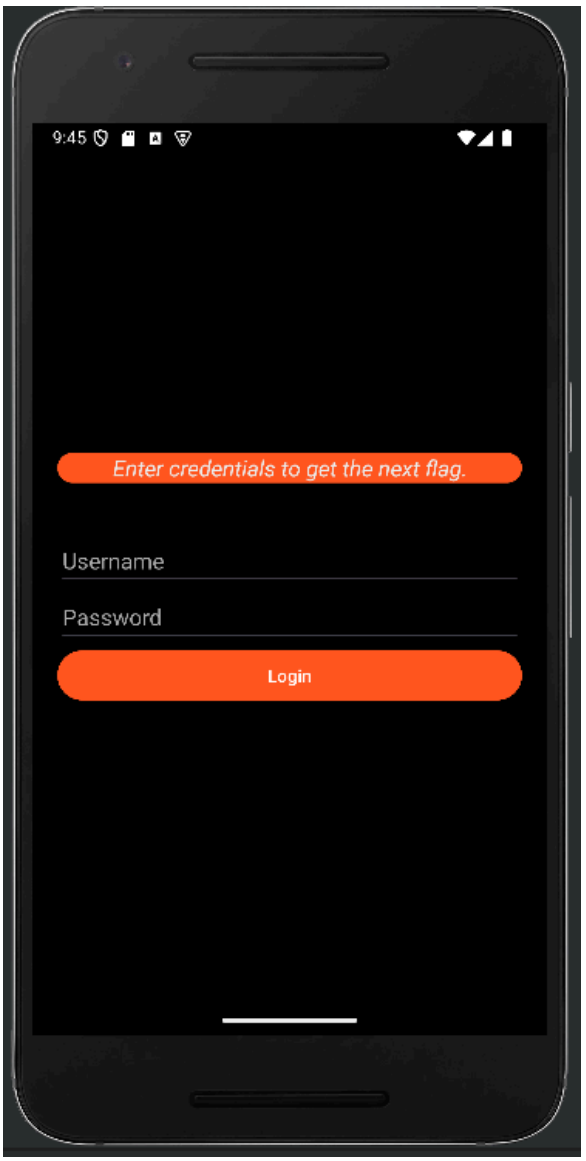
The first question is

"In plain sight, your data I keep, No locks, no codes, just a careless heap. What am I?"



The answer to this question is "**Insecure Data Storage**"

Once that answer is entered in the given text box, the app moves on to the next part of this challenge which presents us with a login page. We are required to find the credentials to move forward.



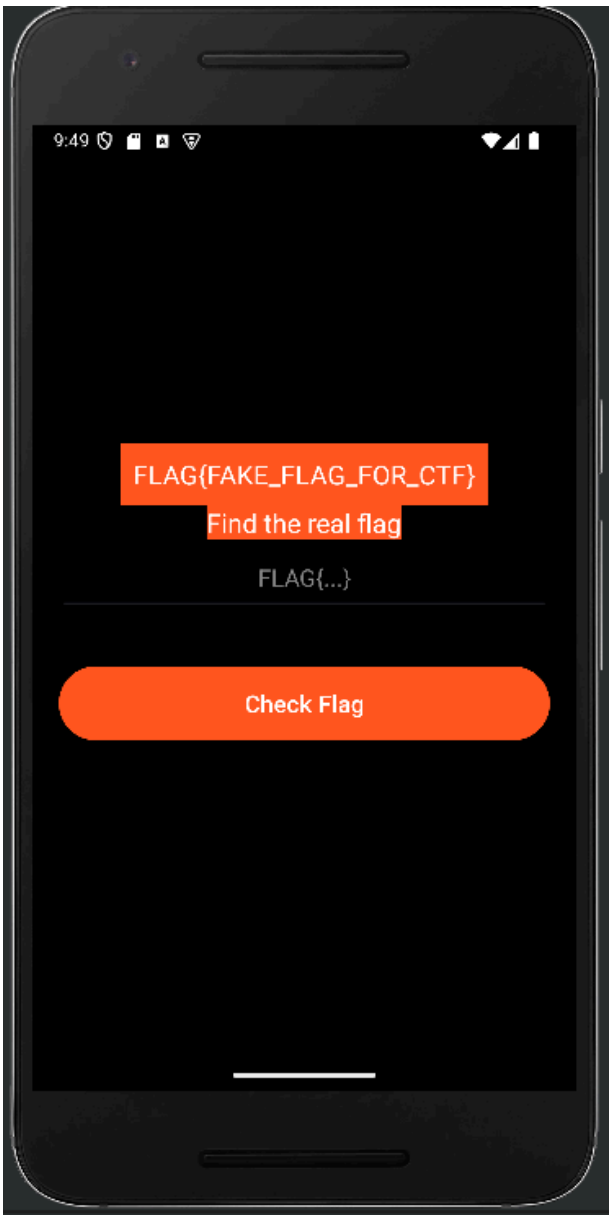
Inspecting the code flow, we can see that the if condition that checks for the username and password has it hardcoded in the code.

```
if (username == "admin" && password == "password123") {
    Log.d(TAG, msg: "Login successful. Redirecting to Dashboard.")

    // Simulating storing flag parts
    Log.d(TAG, msg: "Storing flag parts in various locations")
    StorageHelper.storeFlagParts( context: this)
      InSecureDB.storeFlagPart(this)
      XorUtils.storeEncryptedFlag(this)

    val intent = Intent( packageContext: this, DashBoard::class.java)
    startActivity(intent)
```

Once we enter the credentials above it moves on to the following screen which displays a fake flag

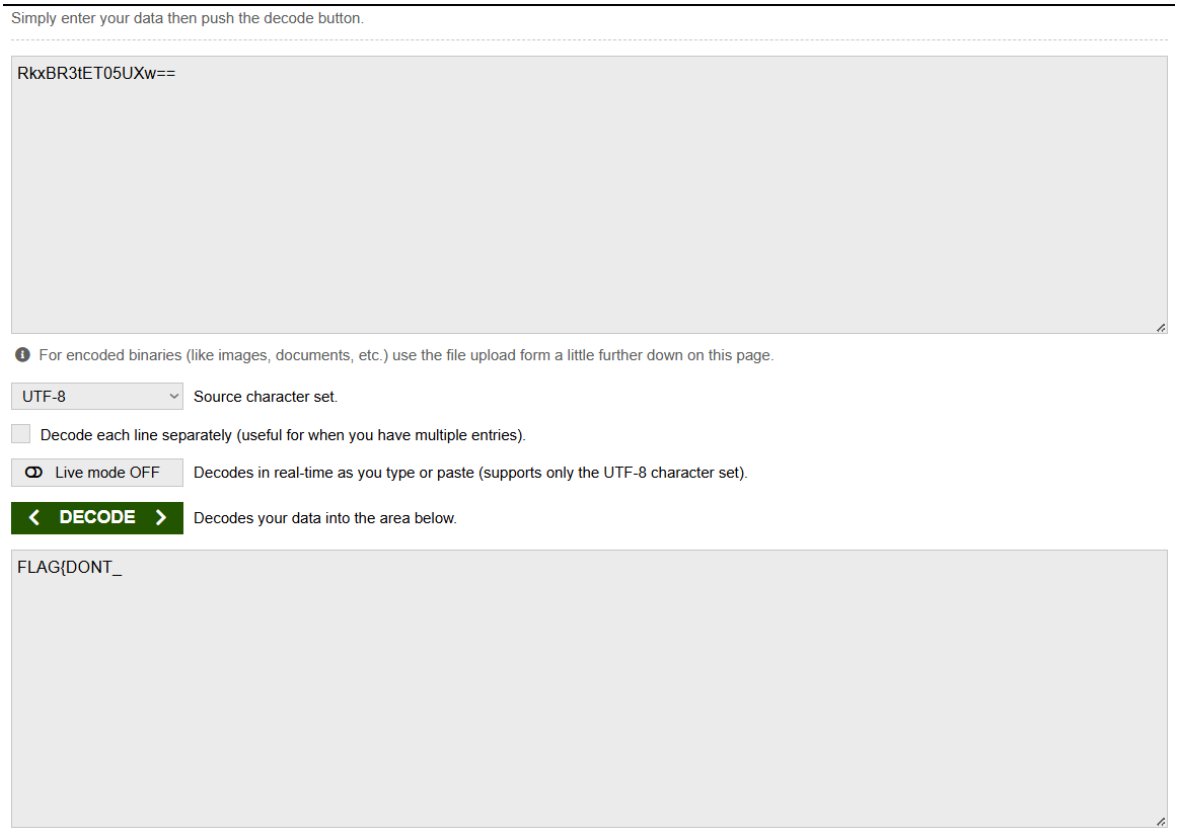


We need to find the real flag which exploits the “Insecure Data Storage”.

Inspecting code flow, we can see that the flag has been split up into 3 parts.

We go to the device explorer, and we navigate to the directory **data/data/com.example.malware_package/shared_prefs** and we see that there is a base64 encoded string that represents the first part of the string.

When base64 decoding the string, we get the following decoded string



First part of the flag: **FLAG{DON'T_**

```
val encodedFlagPart1 = StorageHelper.retrieveFlagPart( context: this)
val flagPart2 = dbHelper.getFlagPart()?.removeSuffix( suffix: "}")
val encryptedFlagPart3 = XorUtils.retrieveEncryptedFlag( context: this)
```

We can see that the second part of the flag is stored in an sql database. We can see from above that the second part of the flag points to the DB Helper. So while inspecting the code in DBHelper, we can see that the relevant database is "**ctf_hidden.db**".

```
companion object {
    private const val DATABASE_NAME = "ctf_hidden.db"
    private const val DATABASE_VERSION = 1
    private const val TAG = "InSecureDB"
}
```

We can also see that the table used is "**Secrets**".

```
override fun onCreate(db: SQLiteDatabase) {
    Log.d(TAG, msg: "Creating database table: secrets")
    db.execSQL( sql: "CREATE TABLE IF NOT EXISTS secrets (id INTEGER PRIMARY
}
```

We navigate again to the **data/data/com.example.malware_package/databases**.

We then open the **ctf_hidden.db** with **sqllite3** and check the secrets table which gives us the second part of the flag, which is
"STORE_DATA"

```
emu64xa:/data/data/com.example.malware_package/databases # sqlite3 ctf_hidden.db
SQLite version 3.44.3 2024-03-24 21:15:01
Enter ".help" for usage hints.
sqlite> SELECT * FROM secrets;
1|STORE_DATA}
```

Second part of the flag: **STORE_DATA**

As given above, if we check the XORUtils for the third part of the flag, we can see that at the start we get that the key is "**CTF**".

```
private const val KEY = "CTF"
```

We can see that this is the key used to encrypt the flag and the encrypted third part is saved in the hidden_flag2.txt file.

```
fun retrieveEncryptedFlag(context: Context): String? {
    return try {
        Log.d(TAG, msg: "Opening hidden_flag2.txt from assets")
        val inputStream = context.assets.open( fileName: "hidden_flag2.txt")
        val encryptedText = inputStream.bufferedReader().use(BufferedReader::readText)
        Log.d(TAG, msg: "Encrypted flag retrieved: $encryptedText")
        xorDecrypt(encryptedText, KEY)
    } catch (e: Exception) {
        Log.e(TAG, msg: "Error retrieving encrypted flag", e)
        null
    }
}
```

Thus we go to the **data/data/com.example.malware_package/files directory** and check the contents of the **hidden_flag2.txt** file.

We write a script in python to decrypt the flag as follows:

```
def xor_decrypt(input_str: str, key: str) -> str:
    print("Decrypting flag using XOR")
    return ''.join(
        chr(int(s) ^ ord(key[i % len(key)]))
        for i, s in enumerate(input_str.split())
    )


encrypted = "28 29 8 16 17 5 22 6 3 15 13 59"
key = "CTF"
decrypted = xor_decrypt(encrypted, key)
print(decrypted)
```

The result of this program is the decrypted third part of the flag.

```
Decrypting flag using XOR
_INSECURELY}
```

Third part of the flag: **_INSECURELY}**

Putting all 3 parts together, the second flag is

FLAG{DON'T_STORE_DATA_INSECURELY}

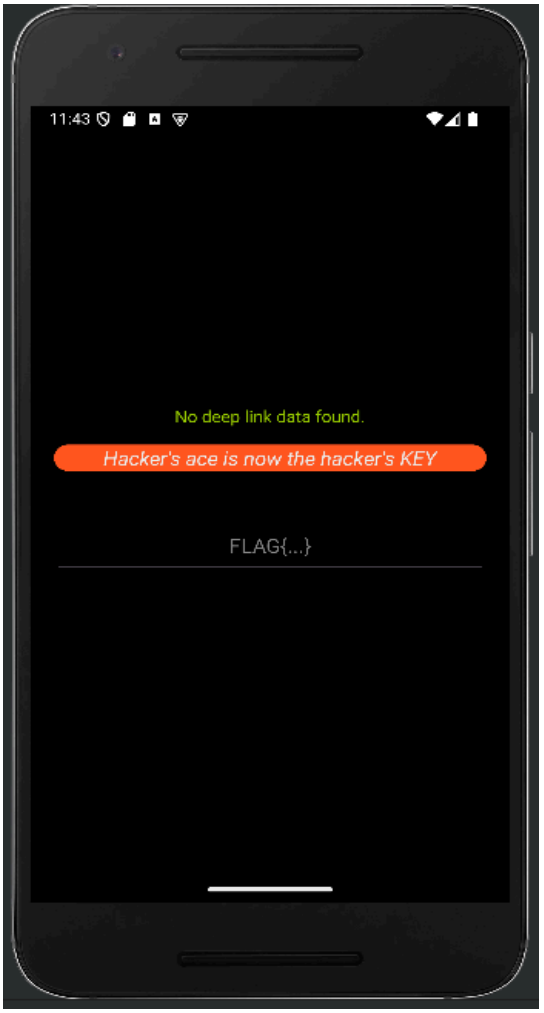
FLAG 3:

The question for flag 3 is

"I'm a shortcut to a hidden place, but unchecked, I'm a hacker's ace. Who am I?"

The answer to the question is "**Deeplinks**".

Once this answer is entered, we are presented with the below screen. The vulnerability that need to be exploited to find the flag at this stage is Deeplinks.



Insepcting the code, we see that there are 2 parameters passed as “**action**” and “**key**”

```
val actionParam = queryParams["action"]
val secretKey = queryParams["key"]
  FlagManager.initializeFlagStorage(this)
if (actionParam == "getFlag" && SecurityUtils.validateSecretKey(secretKey)) {
    val flag = FlagManager.retrieveFlag( context: this)
    flagTextView.text = flag  // Show the flag
    checkFlagButton.visibility = View.VISIBLE  // Show "Check Flag" button

    Log.d( tag: "DeeplinkHandler",  msg: "🎉 Flag retrieved successfully!")
} else {
    flagTextView.text = "Unauthorized access attempt detected!"
}
```

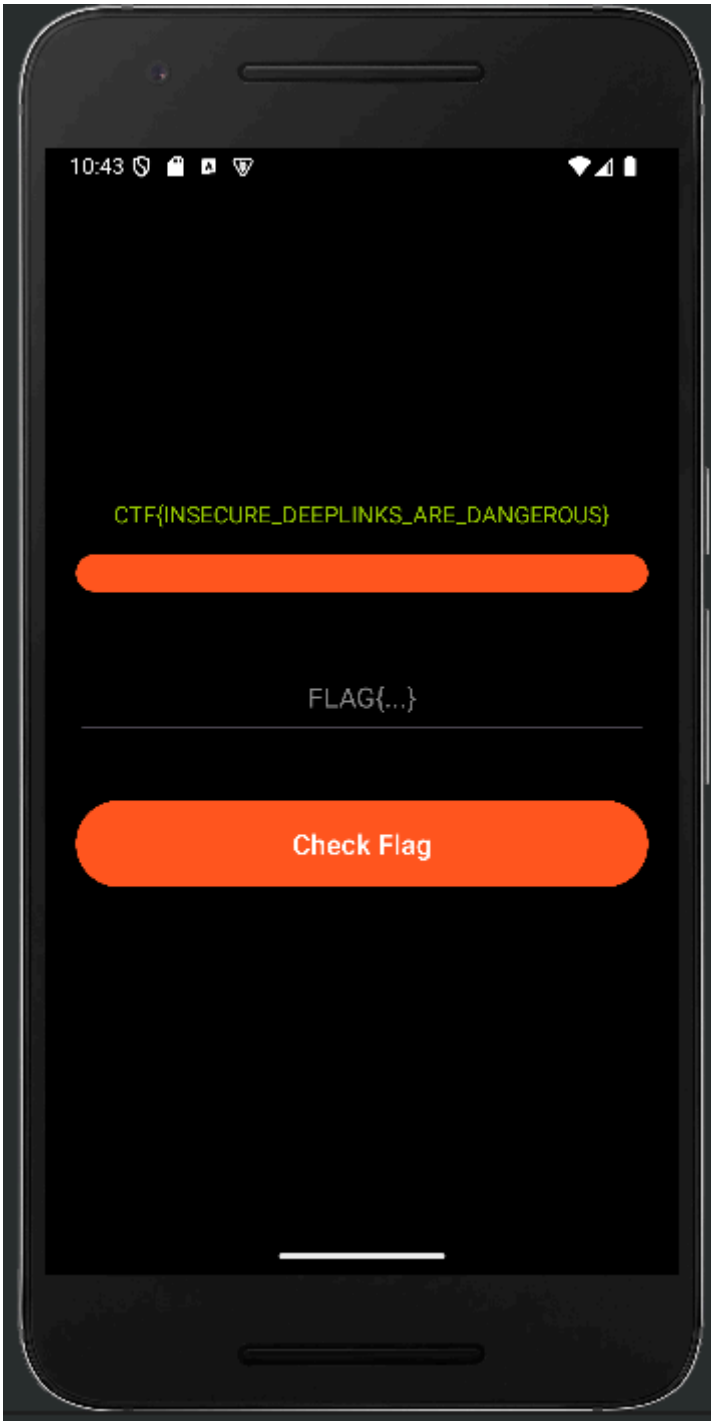
The action as we can see above must be “getFlag” and the key should be a value that is associated with the SecurityUtils file. Thus, on inspecting this file, we can see that it is MD5 hashed. The key is not explicitly mentioned, but the question says that Deeplinks are the hacker’s ace. Thus, it is the key.

Since the vulnerability is deeplinks, and the attacker now also knows the action and key param required to trigger an action in this intent, he uses the following adb command

adb shell am start -a android.intent.action.VIEW -d '''malware_package://flag?action=getFlag&key=Deeplinks'''

```
PS C:\Users\Varun\Downloads\Malware_Package\Malware_Package> adb shell am start -a android.intent.action.VIEW -d '''malware_package://flag?ac
tion=getFlag&key=Deeplinks'''
Starting: Intent { act=android.intent.action.VIEW dat=malware_package://flag/... }
```

We can now see that once this command is executed, the flag is displayed.

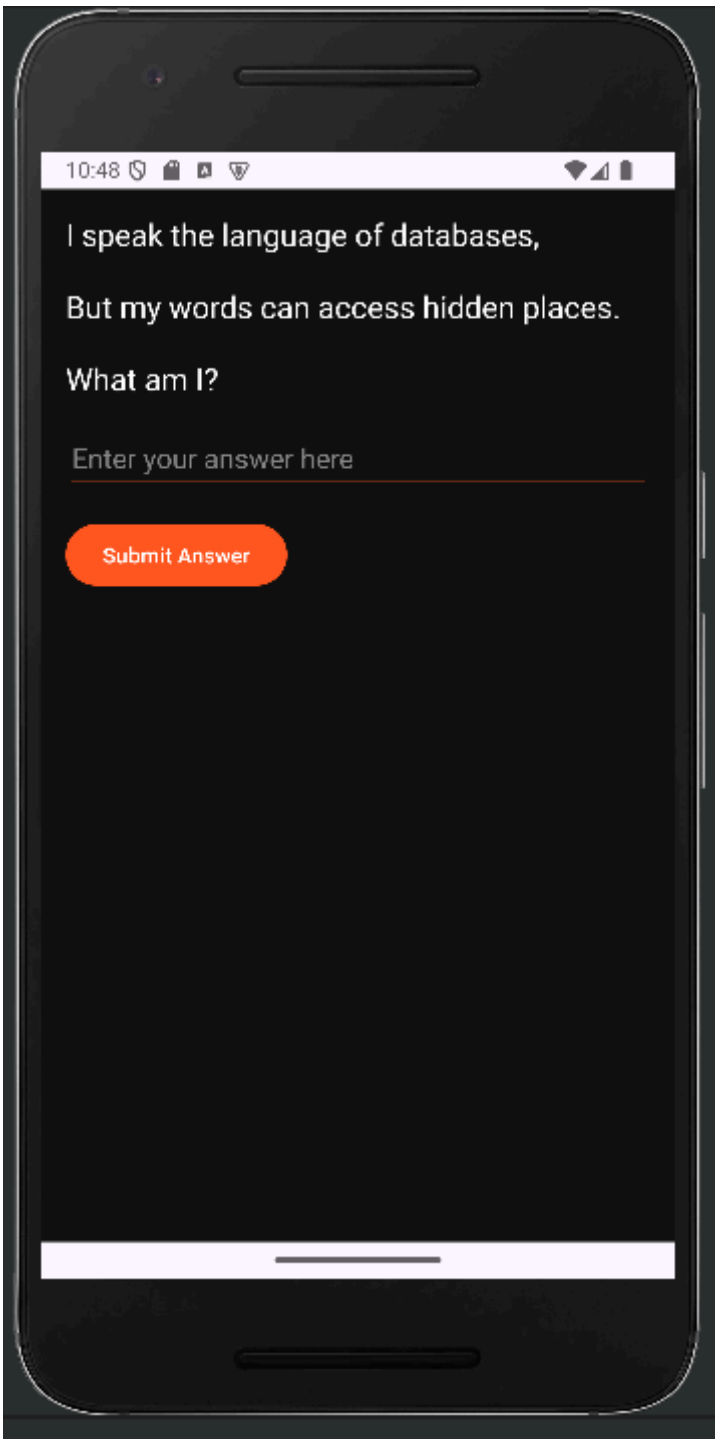


Thus, Flag 3 is **FLAG{INSECURE_DEEPLINKS_ARE_DANGEROUS}**

FLAG 4:

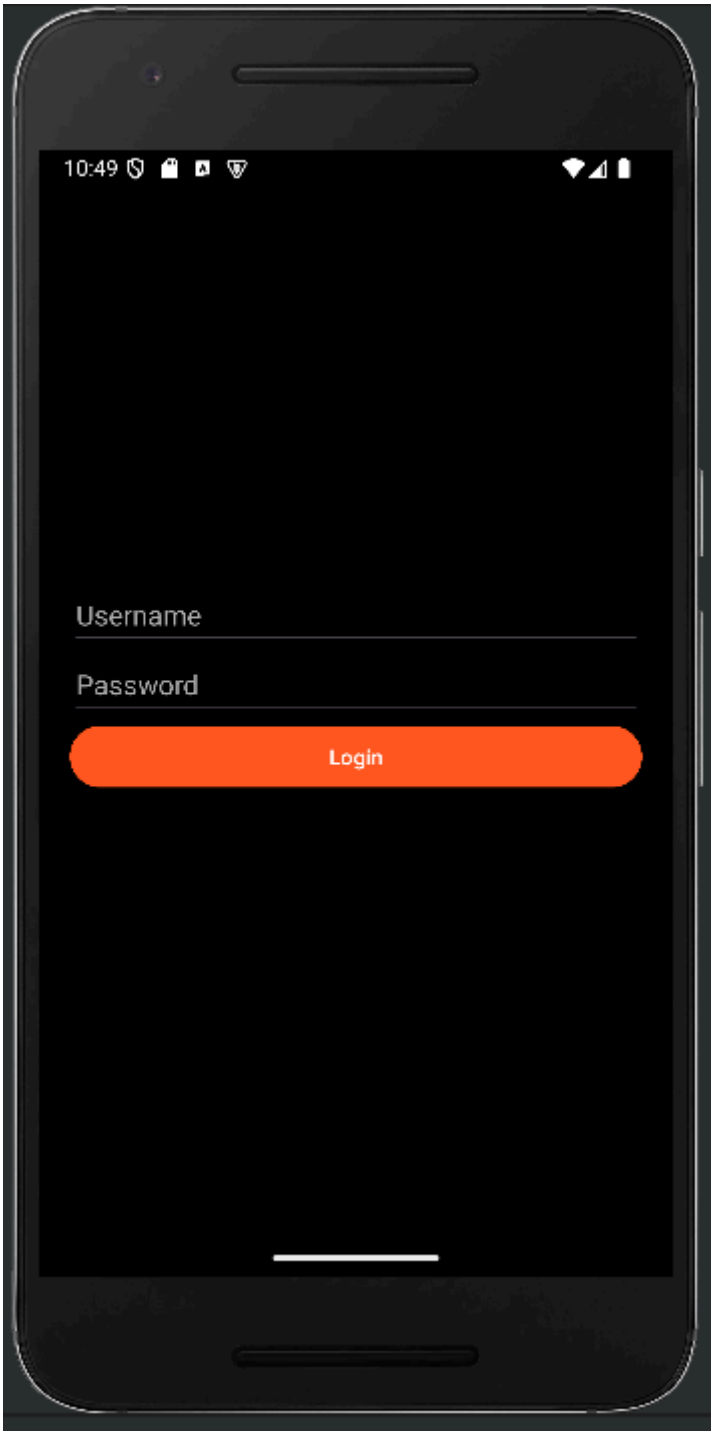
The question for this vulnerability is,

“I speak the language of databases, but my words can access hidden places. What am I?”



The answer to this is "**SQL Injection**"

Once this answer is typed we are presented with the following login page.



This stage has no flag.

Exploiting the SQL Injection vulnerability will make the NEXT FLAG button visible and let us move on to the next challenge.

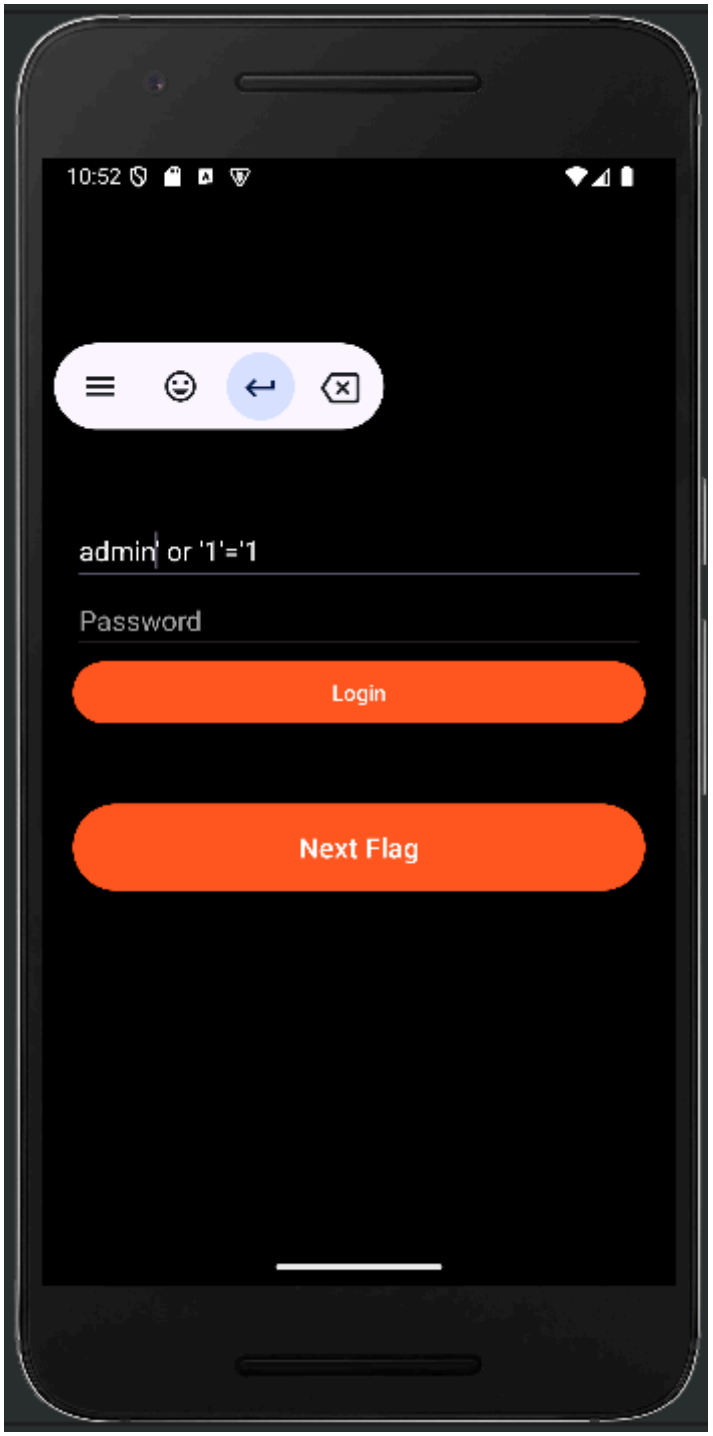
We know that we can use the command template

<random_username>' OR '1' = '1

To get access to the database. This query will give the results always as 1=1 is always true.

We use the already seen before username “**admin**” in the above template and as we can see below the NEXT FLAG option has been enabled without entering the password or the right username.

Thus SQL Injection vulnerability has been exploited.



FLAG 5:

```
emu64xa:/ # cd data/local/tmp
emu64xa:/data/local/tmp # ./frida-server-16.7.4-android-x86_64
```

Here the screen asks for a 4 digit pin to get to the next stage.

On inspecting the code, we can see that it works on 2 basic Boolean functions defined in the SecurityManager file

```
if (securityManager.validatePIN(enteredPin)) {
```

Thus looking at the securitymanager file, we can see that there are 2 functions as shown below.

```
class SecurityManager {
    external fun isFridaDetected(): Boolean
    external fun validatePIN(pin: String): Boolean
}
```

Thus we have to make this function validate pin return True to bypass this stage.

For this we first run the Frida server using the following command in the adb shell.

As we can see, now the Frida server is running.

Now we find out the PID of our running app which can be found using the following command,

```
PS C:\Users\Varun\Downloads\Malware_Package\Malware_Package> adb shell pidof com.example.malware_package
18697
```

Now we write a script named bypass.js with the objective of targeting the validate pin function and defaulting its result to True.

bypass.js > ...

```
Java.perform(function () {
    var SecurityManager = Java.use("com.example.malware_package.SecurityManager");

    // Hook the validatePIN method
    SecurityManager.validatePIN.implementation = function(pin) {
        console.log("[+] Bypassing PIN validation. Entered PIN: " + pin);
        return true;  // Always return true, bypassing the check
    };

    console.log("[+] Hooked validatePIN and bypassed the check.");
});

```

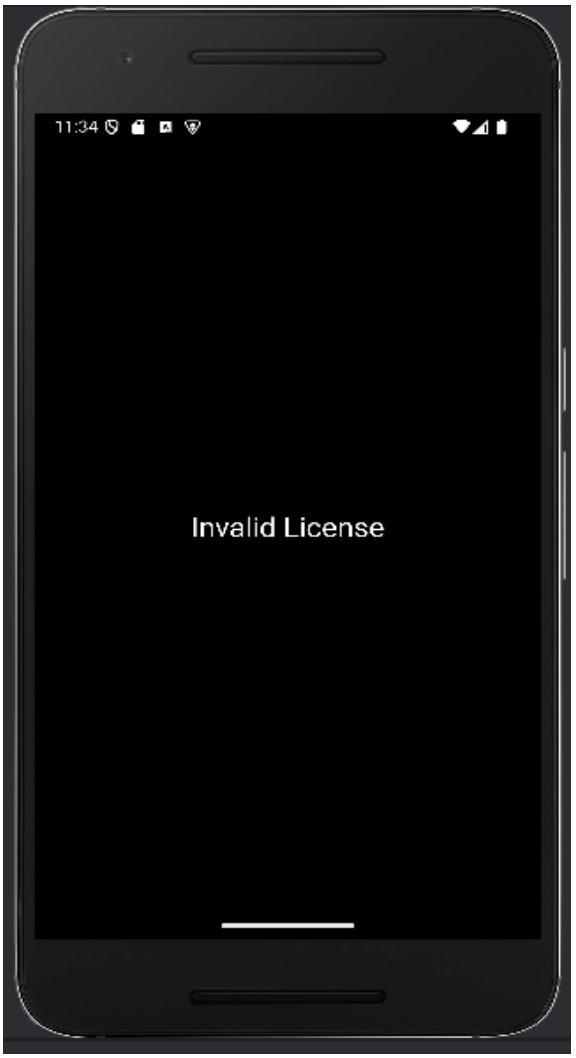
Now we use this script and the pid of the running app in the following Frida command to bypass this challenge.

```
PS C:\Users\Varun\Downloads\Malware_Package\Malware_Package> frida -U -p 18697 -l "C:\Users\Varun\Downloads\bypass.js"
     ____
    / _  |   Frida 16.7.4 - A world-class dynamic instrumentation toolkit
   | (_| |
    > _  |   Commands:
   /_/ |_|       help      -> Displays the help system
   . . . .       object?   -> Display information about 'object'
   . . . .       exit/quit -> Exit
   . . . .
   . . . .   More info at https://frida.re/docs/home/
   . . . .
   . . . .   Connected to Android Emulator 5554 (id=emulator-5554)
Attaching...
[+] Hooked validatePIN and bypassed the check.
[Android Emulator 5554::PID::18697 ]->
```

We can see that the bypass has worked and we get the toast FLAG{you-cracked-it}.

FLAG 6:

We now reach the Licence Invalid Screen.



Now we can see that this is the function that needs to be returned to true

```
val isLicenseValid = checkLicense()
```

We again use Frida to target this function to default its return to true using the following license.js script shown below.

license.js > ...

```
Java.perform(function () {
    var LicenseActivity = Java.use("com.example.malware_package.License");

    LicenseActivity.checkLicense.implementation = function () {
        console.log("[+] Hooked checkLicense, returning true.");
        return true; // Always return true to bypass the license check
    };
});
```

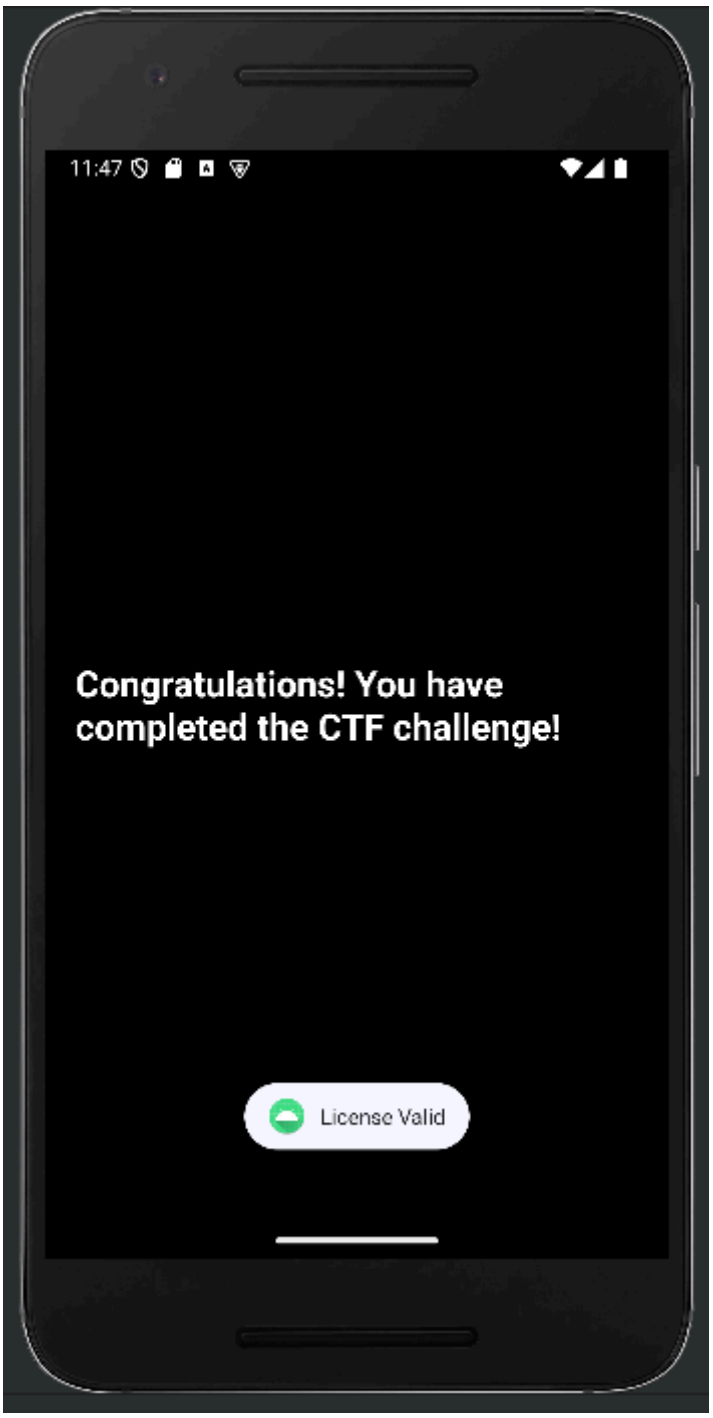
We now use this script in the following Frida command

```
PS C:\Users\Varun\Downloads\Malware_Package\Malware_Package> frida -U -p 18697 -l "C:\Users\Varun\Downloads\license.js"
     ____
    / _  |   Frida 16.7.4 - A world-class dynamic instrumentation toolkit
   | (_| |
    > _  |   Commands:
   /_/ |_|       help      -> Displays the help system
   . . . .       object?   -> Display information about 'object'
   . . . .       exit/quit -> Exit
   . . . .
   . . . .   More info at https://frida.re/docs/home/
   . . . .
   . . . .   Connected to Android Emulator 5554 (id=emulator-5554)

[Android Emulator 5554::PID::18697 ]->
```

As seen above it is bypassed.

When we go back to the previous screen and click on next flag again, we get the below success screen.



Thus this box is solved with 6 different challenges.