

Walkthrough for Notesdroid CTF App

21PC18 - Kaavya S K
21PC34 - Shri Meenaakshi S

Introduction

About the App

NotesDroid CTF is an Android application designed as a **Capture The Flag (CTF) challenge**, where players exploit security flaws to extract hidden flags. It looks like a simple **note-taking app**, but beneath the surface, it hides **six security challenges** that require reverse engineering, cryptography, and hacking skills to solve.

Game Rules

- **Unlock Challenges:** Add a note with the title as the challenge number (e.g., "1", "2") by clicking the start game button. This will unlock the corresponding challenge screen. Then click on view challenges to see the list of them.
- **Find the Flag:** Each challenge hides a **flag**, which needs to be extracted and entered into the challenge screen. If correct, the app confirms your success!
- **Think Like a Hacker!** Use tools like **JADX**, **Frida**, **SQLite**, and **file explorers** to analyze the app and retrieve the flags.

Install apk into the device: **adb install notesdroidctf.apk**

Challenge 1: XOR Mystery!

Description

The flag in this challenge is obfuscated using a simple **XOR-like operation** on each character. Our goal is to reverse the obfuscation and extract the correct flag.

1. Decompile the APK

- Use **JADX** or **apktool** to decompile the APK and analyze the app's source code.

1. With **apktool**, decompile the app by running:

```
apktool d notesdroid.apk -o notesdroid_decompiled
```

- Navigate to the decompiled folder to explore the **smali** files.

2. Locate the Obfuscated Flag

- In the decompiled folder, check the **"smali\com\practice\notesapp\ChallengeDetailActivity.smali"** file.
- Inside this file, you will find the string **"rddapasi"**—this appears to be the obfuscated flag.

```
.method private final getXORObfuscatedFlag()Ljava/lang/String;
    .locals 11

    const-string v0, "rddapasi"

    .line 57
    check-cast v0, Ljava/lang/CharSequence;

    .line 82
    new-instance v1, Ljava/util/ArrayList;
```

3. Understanding the Obfuscation Logic

In the original Kotlin source code (**ChallengeDetailActivity.kt**), there is a function:

```

private final String getXORObfuscatedFlag() {
    ArrayList arrayList = new ArrayList(r0.length());
    for (int i = 0; i < r0.length(); i++) {
        arrayList.add(Character.valueOf((char) (r0.charAt(i) - 3)));
    }
    return CollectionsKt.joinToString$default(arrayList, "", null, null, 0, null, null, 62, null);
}

```

- This function **subtracts 3** from each character's ASCII value to encode the flag.
- To **reverse it**, we simply add 3 to each character.

4. Decoding the Flag

- The encoded string "**rddapas i**" is found in **smali/kotlin/text/FlagEnum.smali**.
- Reverse the transformation:
 - 'r' → 'o'
 - 'd' → 'a'
 - 'd' → 'a'
 - 'a' → '^' (*Possible encoding artifact*)
 - 'p' → 'm'
 - 'a' → '^' (*Possible encoding artifact*)
 - 's' → 'p'
 - 'i' → 'f'

The decoded flag is:

Flag: oaa^m^pf

Challenge 2: Lost in Preferences!

Description:

The flag is stored in Android's SharedPreferences.

Solution:

Use **adb** to access the app's SharedPreferences:

adb shell

cd /data/data/com.practice.notesapp/shared_prefs/

cat CTF_PREFS.xml

```
PS C:\Users\Shri Meenaakshi\Desktop> adb shell
emu64xa:/ # cd /data/data/com.practice.notesapp/shared_prefs/
emu64xa:/data/data/com.practice.notesapp/shared_prefs # ls
CTF_PREFS.xml
emu64xa:/data/data/com.practice.notesapp/shared_prefs # cat CTF_PREFS.xml
<?xml version='1.0' encoding='utf-8' standalone='yes' ?>
<map>
  <set name="unlocked_challenges">
    <string>1</string>
  </set>
  <string name="flag2">Shared{Pref}</string>
</map>
emu64xa:/data/data/com.practice.notesapp/shared_prefs #
```

Flag: Shared{Pref}

Challenge 3: SQL Secrets!

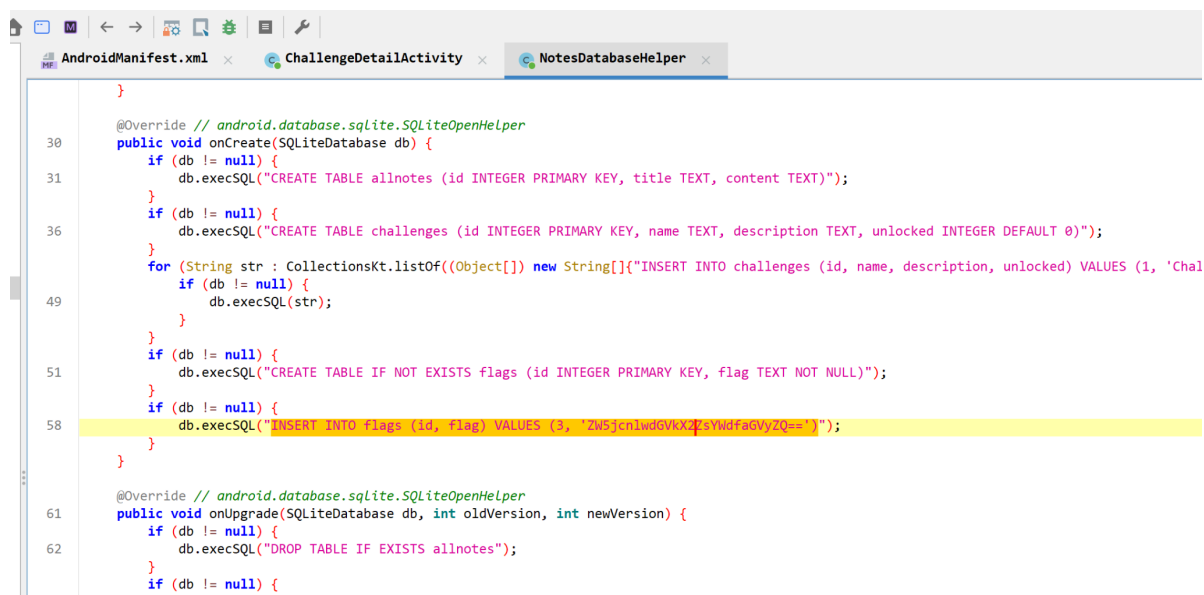
Description

The flag is stored inside the **app's SQLite database**. Extracting it requires accessing and analyzing the database file.

1. Locate Database Handling Code

- **Decompile APK** using JADX
- Search for **INSERT INTO flags** then go to NotesDatabaseHelper.

Found the query:



```

}

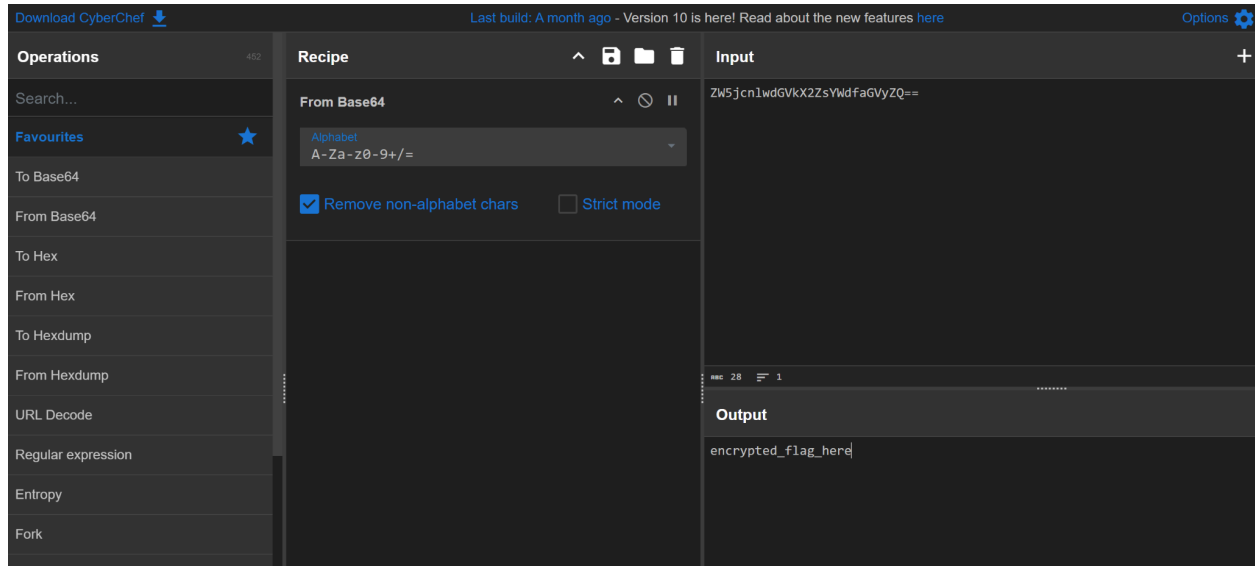
@Override // android.database.sqlite.SQLiteOpenHelper
30 public void onCreate(SQLiteDatabase db) {
31     if (db != null) {
32         db.execSQL("CREATE TABLE allnotes (id INTEGER PRIMARY KEY, title TEXT, content TEXT)");
33     }
34     if (db != null) {
35         db.execSQL("CREATE TABLE challenges (id INTEGER PRIMARY KEY, name TEXT, description TEXT, unlocked INTEGER DEFAULT 0)");
36     }
37     for (String str : CollectionsKt.listOf((Object[]) new String[]{"INSERT INTO challenges (id, name, description, unlocked) VALUES (1, 'Chal
48         if (db != null) {
49             db.execSQL(str);
50         }
51     }
52     if (db != null) {
53         db.execSQL("CREATE TABLE IF NOT EXISTS flags (id INTEGER PRIMARY KEY, flag TEXT NOT NULL)");
54     }
55     if (db != null) {
56         db.execSQL("INSERT INTO flags (id, flag) VALUES (3, 'ZW5jcmlwdGVkX2ZsYWdfaGVyZQ==')");
57     }
58 }

@Override // android.database.sqlite.SQLiteOpenHelper
61 public void onUpgrade(SQLiteDatabase db, int oldVersion, int newVersion) {
62     if (db != null) {
63         db.execSQL("DROP TABLE IF EXISTS allnotes");
64     }
65     if (db != null) {
66         db.execSQL("DROP TABLE IF EXISTS challenges");
67     }
68     onCreate(db);
69 }
```

- The flag is **Base64** encoded.
- The flag entry: **ZW5jcmlwdGVkX2ZsYWdfaGVyZQ==**

2. Decode the Flag

- Use **CyberChef** or any Base64 decoder:



Decoded flag → **encrypted_flag_here**

Challenge 4: Caesar's Lost Scroll!!

Description:

The flag is encrypted using a **Caesar cipher** (ROT13 shift). The function **decryptCaesarCipher()** attempts to decode it.

Step 1: Decompiled the APK using JADX

- Initially, we inspected the **decryptCaesarCipher()** function in JADX, but it didn't show the actual encrypted flag.
- Instead, the function processes a variable (**r0**), whose value was **not directly visible**.

Step 2: Decompiled the APK using Apktool

Since the flag was not visible in JADX, we used Apktool to extract the Smali code:

```
apktool d notesdroid.apk -o notesapp_decompiled
```

Step 3: Searching for the Encrypted Flag in Smali

We searched for the **string constant** in the Smali files:

The flag "XUJ{Znoyl_Zlfgrel}" was found in:

`\smali\com\practice\notesapp\ChallengeDetailActivity.smali`

```
208
209 .method private final decryptCaesarCipher()Ljava/lang/String;
210     .locals 11
211
212     const-string v0, "XUJ{Znoyl_Zlfgrel}"
213
214     .line 71
215     check-cast v0, Ljava/lang/CharSequence;
216
217     .line 100
218     new-instance v1, Ljava/util/ArrayList;
219
220     invoke-interface {v0}, Ljava/lang/CharSequence; ->length()I
221
222     move-result v2
```

Step 4: Decrypting the Flag

The function suggests a **ROT13 transformation** (Caesar cipher shift by 13).

We used a Python script to decode it:

main.py	Output
<pre>1 import codecs 2 print(codecs.decode("XUJ{Znoyl_Zlfgrel}", "rot_13")) 3</pre>	<pre>KHW{Mably_Mystery} === Code Execution Successful ===</pre>

Flag: XHQ{Almost_Mystery}

Challenge 5: Hidden File Treasure!

Step 1: Analyzing the Code in JADX

While decompiling the APK using **JADX**, we found the following function inside **ChallengeDetailActivity**:

```
private final String readFlagFromFile() {
    String str = "";
    try {
        File file = new File(getFilesDir(), "flag5.txt");
        if (file.exists()) {
            str = StringsKt.trim((CharSequence) FilesKt.readText$default(file, null, 1, null)).toString();
        } else {
            Log.e("FLAG", "Flag file not found");
        }
    } catch (IOException e) {
        Log.e("FLAG", "Error reading flag file", e);
    }
    return str;
}
```

This confirms that the app **reads the flag from flag5.txt** inside the internal storage.

Step 2: Check if the Flag File Exists

The app attempts to **read flag5.txt**, but if the file **was never created**, the challenge can be bypassed by submitting an **empty flag**.

Step 3: Gaining Access to Internal Storage

Since this directory is **protected**, we need **root access** or a **debuggable APK** to explore it. To navigate to the correct location, run:

```
adb shell "cat /data/data/com.practice.notesapp/files/flag5.txt"
```

Step 4: Extracting the Flag

```
PS C:\Users\Shri Meenaakshi\Desktop> adb shell "cat /data/data/com.practice.notesapp/files/flag5.txt"
Hola{path_traversal_ownz}
PS C:\Users\Shri Meenaakshi\Desktop> 
```

Flag : Hola{path_traversal_ownz}

Challenge 6: Hidden Intent!

Description:

The flag is **passed between activities using an Intent extra**.

Step 1: Analyzing the Code in JADX

Since the challenge involves **hidden intents**, we search for `putExtra("flag6", ...)` in **JADX**.

In another activity, we find:

```
        invoke2(challenge);
        return Unit.INSTANCE;
    }

    /* renamed from: invoke, reason: avoid collision after fix types in other method */
    public final void invoke2(Challenge challenge) {
        Intrinsic.checkNotNullParameter(challenge, "challenge");
        Intent intent = new Intent(ChallengeListActivity.this, (Class<?>) ChallengeDetailActivity.class);
        intent.putExtra("challenge_id", challenge.getId());
        intent.putExtra("flag6", "Final{flag}");
        ChallengeListActivity.this.startActivity(intent);
    }
}
```

This confirms that the **flag is directly embedded in the Intent**.

Step 2: Extracting the Flag

Since we now know the **hardcoded flag**, we can directly retrieve:

Flag: Final{Flag}

Alternatively, we can **extract it dynamically** using **Frida**.

Step 3: Using Frida to Intercept the Intent (Optional)

If the flag was **not hardcoded**, we could hook into `getStringExtra("flag6")` using **Frida**:

```
Java.perform(function () {  
  var Intent = Java.use("android.content.Intent");  
  Intent.getStringExtra.overload("java.lang.String").implementation = function (key) {  
    var result = this.getStringExtra(key);  
    console.log("[*] Intercepted Intent Extra: Key=" + key + ", Value=" + result);  
    return result;  
  };  
});
```

Run it using:

```
frida -U -n com.practice.notesapp -s intent_hook.js --no-pause
```

Conclusion

The **NotesDroid CTF App** highlights key **Android security flaws** through hands-on challenges. Players learn to exploit and defend against **common vulnerabilities** like weak encryption, SQL injection, and intent-based attacks.