CT126-3-2-CTF-PRACTICAL CTF STRATEGIES

GROUP ASSIGNMENT (40%)

Team Name: Exploit Expert

Imaginary CTF 2025

No.	Name	TP Number	Workload
1.	Chew Kai Xuan	TP076457	25%
2.	Chua Yi Hua	TP099596	25%
3.	Fhang Ying Ying	TP076307	25%
4.	Nicholas Beh	TP070075	25%

Table of Content

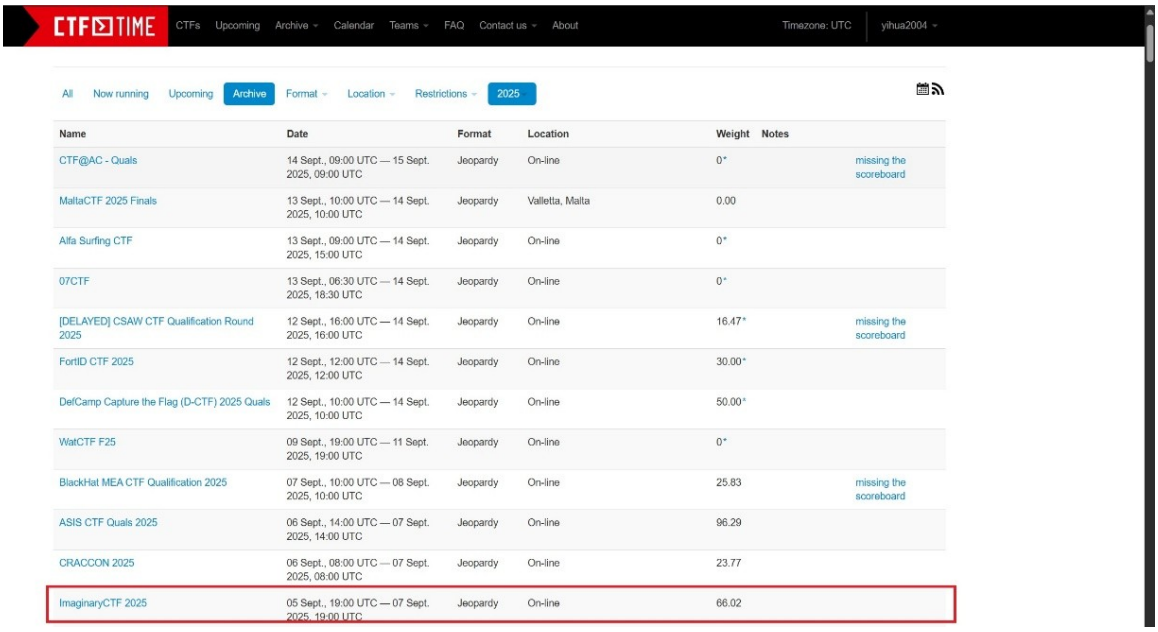
1. Introduction.....
   1.1 Overview of the competition.....
   1.2 Prizes and Winning Awards.....
   1.3 Our Team: ExploitExperts.....
   1.4 Educational Alignment and Approval.....
2. Our Challenge.....
   2.1 Challenge Details.....
   2.2 Accessing the challenge.....10
3. Step-by-Step Solution.....12
   3.1 Initial Problem Understanding.....12
   3.2 Inspiration on Solving the Challenges and Teamwork.....14
   3.3 Explanation on steps to solve the challenge.....15
   3.4 Capture The Flag.....23
   3.5 Our Teamwork to solve this challenge.....24
   3.6 Problems that we faced.....32
4. Tools / Commands we used.....34
   4.1 ChatGPT.....34
   4.2 Github (For Reference Only).....37
   4.3 Jadx.....39
   4.4 Python.....41
5. Vulnerability.....43
6. Challenges Faced.....44
   6.1 Lessons Learned.....45
7. Explain Like I'm Five (ELIF).....46
8. Group Photo.....47
9. Workload Matrix.....48
10. References.....49

Our CTF challenges' category:

Reverse Engineering

CTF challenge write-up:

1. Introduction



CTF TIME | CTFs | Upcoming | Archive | Calendar | Teams | FAQ | Contact us | About | Timezone: UTC | yihua2004

All | Now running | Upcoming | Archive | Format | Location | Restrictions | 2025

Name	Date	Format	Location	Weight	Notes
CTF@AC - Quals	14 Sept., 09:00 UTC — 15 Sept. 2025, 09:00 UTC	Jeopardy	On-line	0*	missing the scoreboard
MaltaCTF 2025 Finals	13 Sept., 10:00 UTC — 14 Sept. 2025, 10:00 UTC	Jeopardy	Valletta, Malta	0.00	
Alfa Surfing CTF	13 Sept., 09:00 UTC — 14 Sept. 2025, 15:00 UTC	Jeopardy	On-line	0*	
07CTF	13 Sept., 06:30 UTC — 14 Sept. 2025, 18:30 UTC	Jeopardy	On-line	0*	
[DELAYED] CSAW CTF Qualification Round 2025	12 Sept., 16:00 UTC — 14 Sept. 2025, 16:00 UTC	Jeopardy	On-line	16.47*	missing the scoreboard
FortID CTF 2025	12 Sept., 12:00 UTC — 14 Sept. 2025, 12:00 UTC	Jeopardy	On-line	30.00*	
DefCamp Capture the Flag (D-CTF) 2025 Quals	12 Sept., 10:00 UTC — 14 Sept. 2025, 10:00 UTC	Jeopardy	On-line	50.00*	
WatCTF F25	09 Sept., 19:00 UTC — 11 Sept. 2025, 19:00 UTC	Jeopardy	On-line	0*	
BlackHat MEA CTF Qualification 2025	07 Sept., 10:00 UTC — 08 Sept. 2025, 10:00 UTC	Jeopardy	On-line	25.83	missing the scoreboard
ASIS CTF Quals 2025	06 Sept., 14:00 UTC — 07 Sept. 2025, 14:00 UTC	Jeopardy	On-line	96.29	
CRACCON 2025	06 Sept., 08:00 UTC — 07 Sept. 2025, 08:00 UTC	Jeopardy	On-line	23.77	
ImaginaryCTF 2025	05 Sept., 19:00 UTC — 07 Sept. 2025, 19:00 UTC	Jeopardy	On-line	66.02	

Figure 1 – Competition Information

1.1 Overview of the competition

The ImaginaryCTF 2025 was a virtual online CTF competition held on the ImaginaryCTF platform - a rapidly emerging community hub in the cybersecurity community and competitive hackers. The competition was structured to refine technical skills, foster teamwork and render a competitive experience enabling the participants to learn and gain a professional experience in the competition *(ImaginaryCTF 2025 - Competition, 2025).*

The event was a completely online event that ran between 05 September, 19:00 UTC and 07 September, 19:00 UTC and international players could participate without limitations. With estimated 5,000 participants in 1,414 teams, which is one of the largest attendances at the platform so far. In a Jeopardy-like format, the competitors can pick challenges of various categories, where every solved task earned them points that ranked them on the live world board *(ImaginaryCTF 2025 - Competition, 2025).* Overall, there were 15 challenges organized in 7 categories, cryptography, binary exploitation, web exploitation, reverse engineering, forensics, steganography and OSINT. Each difficulty contained a digital flag as an indicator of a solution. Teams might specialize in what they were best at or diversify in categories to make as many

points as possible. Every flag taken brought them on an even greater scoreboard, escalating the competition to the summit *(ImaginaryCTF 2025 - Competition, 2025)*.

ImaginaryCTF 2025 – Event Type and Details:

- **Event Type:** Public
- **Format:** Jeopardy
- **Platform:** Online
- **Participants:** Approximate 5,000 players joined
- **Teams:** 1414 teams joined
- **Challenges:** 48 questions across 7 categories
- **Wave:** 3 (In total)
- **Difficulty level:** 1-10 (Only 1-8 in this CTF competition)

Reversing

Challenge	Difficulty (1-10)	Wave
nimrod	2	1
weird-app	4	2
comparing	4	1
memory	5	1
illegal	6	1
stacked	6	3
questionably-distributed	8	1

Figure 2 – Reverse Engineering Difficulty Level

ImaginaryCTF 2025 – Event Organizers:

Event organizers

- [sqrt (-1) + 1]

ImaginaryCTF 2025 will now take place from September 5-7.

ImaginaryCTF 2025 is a cybersecurity CTF competition run by ImaginaryCTF with a variety of challenges for all skill levels, running from September 5-7. Teams will be challenged to discover vulnerabilities in websites, crack codes, and recover information through challenges in cryptography, binary exploitation, web exploitation, forensics, reversing, and more. We hope that you will learn something new from our challenges!

Figure 3 - Event Organizers

ImaginaryCTF is a CTF contest that will be conducted annually. This is an event arranged by **[sqrt (-1) + 1]**. In **ImaginaryCTF 2025**, question generation was done by **15 authors**. And **5 staffs/moderators** who have the oversight of the entire competition. Other personalities in the formation of this **CTF** competition are the **challenge developers**, **challenge tester** and **supporting partners**.

1.2 Prizes and Winning Awards

Prizes

$200, 2x Binary Ninja
$150, 2x Binary Ninja
$100, 1x Binary Ninja

$50 pool for best writeups
Binary Ninja for best writeup using Binja

Figure 4 - Prizes

ImaginaryCTF 2025 had **appealing prizes** to **increase** the **participation rate** and the **team** that **performs** the **best**. The **prize system** was meant to **reward** the **rankings** on the **leaderboard** at the **conclusion** of the event **(ImaginaryCTF 2025 - Competition, 2025)**. The **prizes** were as follows:

1st Place: 200 + 2x Binary Ninja Licenses.
2nd Place: 150 and 00:1121 2x Binary Ninja Licenses.
3rd Place: 100 + 1 Binary Ninja License.

Best Writeups: $50 prize pool

Special Award: The best writeup with the use of Binary Ninja license.

These prizes provided teams and the participants with a good incentive to compete fiercely and collaborate with each other and work towards the larger cybersecurity ecosystem by providing quality writeups (ImaginaryCTF 2025 - Competition, 2025).

1.3 Our Team: ExploitExperts

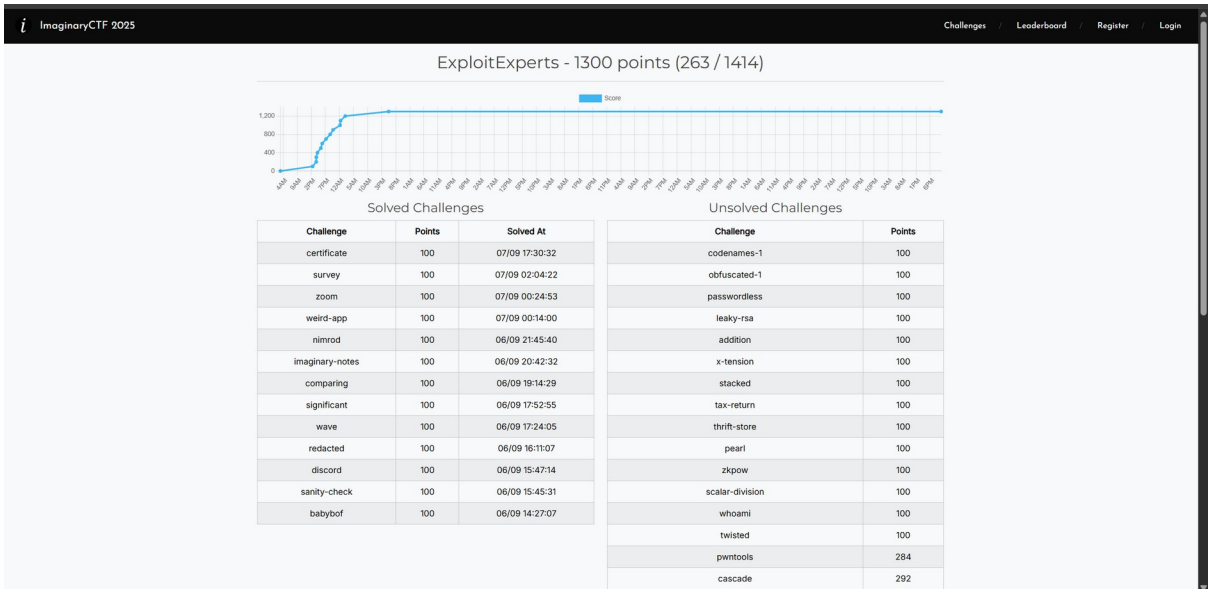


Figure 5 – Team ExploitExperts Dashboard

The attached image (*Figure 5*) is team page with our team's name **"Exploit Experts"** in ImaginaryCTF 2025. It shows our progress chart, solved challenges and unsolved challenges.

Figure 6 – Team members

The attached image is our team's page with our team member's name **ExploitExperts -** Nicholas Beh, **yyf** – Fhang Ying Ying, **Yi** – Chua Yi Hua, **xuan** – Chew Kai Xuan.

Introduction of Our Team – ExploitExperts

One day, in the world of the Internet and privileged secrets and puzzles, four adventurers came together to show their talents. Their names were **ExploitExperts** and they were named Chew Kai Xuan, Chua Yi Hua, Fhang Ying Ying and Nicholas Beh.

Their hunt started as they heard about ImaginaryCTF 2025, an event that resonated with their interests, namely, cybersecurity and the art of solving puzzles. The moment couldn't have been better: the challenge would be held on 5th to 7th September, just before their pending academic deadlines, offering them an opportunity to take this adventure that was so daring.

They were not only attracted by the excitement of competition which was reflected in their academic activities. The second wave of issues was also enchanted, requiring sharp objects, problem-solving magic, logical thinking, and technical accuracy, which was developed during their studies on reverse engineering and forensics.

But the magic of the whole was not in mediums, or in code, or in their combination. They found out that the real keys to winning were coordination, communication and time management. Thus, they thought of a brilliant plan: everybody would bring their strong points and play them as a knight playing a sword.

- Kai Xuan was crowned king of PWN, and she brought down binary defenses.

- Yi Hua got good in Reverse Engineering and OSINT and ripped off the layers of obfuscation.
- Ying Ying was on the gate of the Web and Forensics, where it was possible to find the hints in the plain sight.
- Nicholas unravelled the enigma of MISC and Cryptography and solved codes with accuracy.

They travelled collectively through trials that simulated the true-life trials of cybersecurity. With every challenge they overcame they got closer to the flag and became closer as a team.

Therefore, taking part in ImaginaryCTF 2025 was not only a competition, but an opportunity to bring knowledge into practice and put skills to the test on the international level, and to show that nothing was impossible with teamwork and planning.

And the tale of them is told, and in the ever-growing frontier of cybersecurity tales.

Figure 7 – ExploitExperts.jpg

1.4 **Educational Alignment and Approval**

We obtained permission from our lecturer, Mr Mohd Hanis Bin Jenalis, to enter ImaginaryCTF 2025. His green light given us a guarantee not only for a contest but also the continuation of what we studied. It was though the theory we had been dragging about in textbooks and lecture notes were now going to be implemented.

As the competition started, the classroom and practice boundaries were blurred. Things we talked about became riddle to be discovered. Every difficulty required problem-solving, reasoning and genuine cooperation. The theories had been waiting all along to this stage, and now they were alive, demanding, challenging, and exciting.

It took hours to hone our skills in Python, Kali Linux and cybersecurity tools and we can finally put them to the test. ImaginaryCTF with its Jeopardy-style format was virtually like it was custom-made. We were allowed to choose issues that worked to our advantage. Others broke coded, others hunted, others assembled logic which made the impossible seem suddenly solvable.

In retrospect, it was not just a battle of skill. It was a bridge--a passage out of theory into practice. In our case, ImaginaryCTF 2025 was not only the competition in cybersecurity, but also the tale of development, strength, and exploration.

*Figure 8 (Mr. Mohd Hanis bin Jenalis with Kuih Lapis – *GPT generated image*)*

2. Our Challenge

weird-app (100pts) - 272 solves

by cleverbear57

Description

I made this weird android app, but all it gave me was this .apk file. Can you get the flag from it?

Attachments

weird.zip

Close

Figure 9 – ImaginaryCTF 2025 | Challenge Question

2

2.1 Challenge Details

- **Category:** Reverse Engineering
- **Challenge Name:** weird-app
- **Difficulty Level:** 4 (Medium)
- **Score:** 100 points
- **Wave:** 2
- **Challenge Description:** We were given the challenge as weird.zip containing an APK, we downloaded it and transferred it to an isolated Kali Linux VM to analyze. Using jadx-gui to inspect the decompiled Java/Kotlin tree in its nature and searched the likely signs (flag, CTF {REDACTED, transform, encrypt, etc.), we narrowed down to suspicious functions called by the MainActivity. Transformation was reversed with a short Python script to the obfuscated flag-like string to de-obfuscate it. ChatGPT was useful in interpreting decompiled code and generating executable Python code, but all execution and verification occurred locally; we recorded commands, version and screenshots of tools to replicate.

Solved Challenges

Challenge	Points	Solved At
certificate	100	07/09 17:30:32
survey	100	07/09 02:04:22
zoom	100	07/09 00:24:53
weird-app	100	07/09 00:14:00
nimrod	100	06/09 21:45:40
imaginary-notes	100	06/09 20:42:32
comparing	100	06/09 19:14:29
significant	100	06/09 17:52:55
wave	100	06/09 17:24:05
redacted	100	06/09 16:11:07
discord	100	06/09 15:47:14
sanity-check	100	06/09 15:45:31
babybof	100	06/09 14:27:07

Figure 10 – Solved Challenges (Weird-App)

The challenge was about reverse engineering, an Android application type challenge and presented as a medium-level challenge. It was ranked at 100 points. The problem seemed simple, but it took time to overcome thousands of decompiled files and find out suspicious code blocks. Jadx tool was used to examine the APK and the obstacle eventually prepared our patience and diagnostic capabilities in the analysis of source code.

2.2 Accessing the challenge

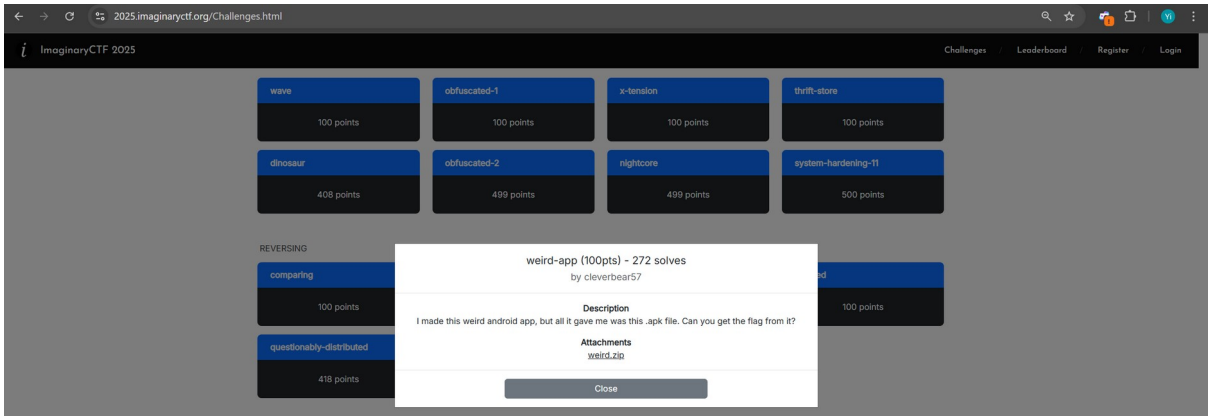


Figure 11 – Accessing the challenge

The corresponding Weird-App CTF page is hosted at:
https://2025.imaginaryctf.org/Challenges.html

Screenshot of Challenge Environment:

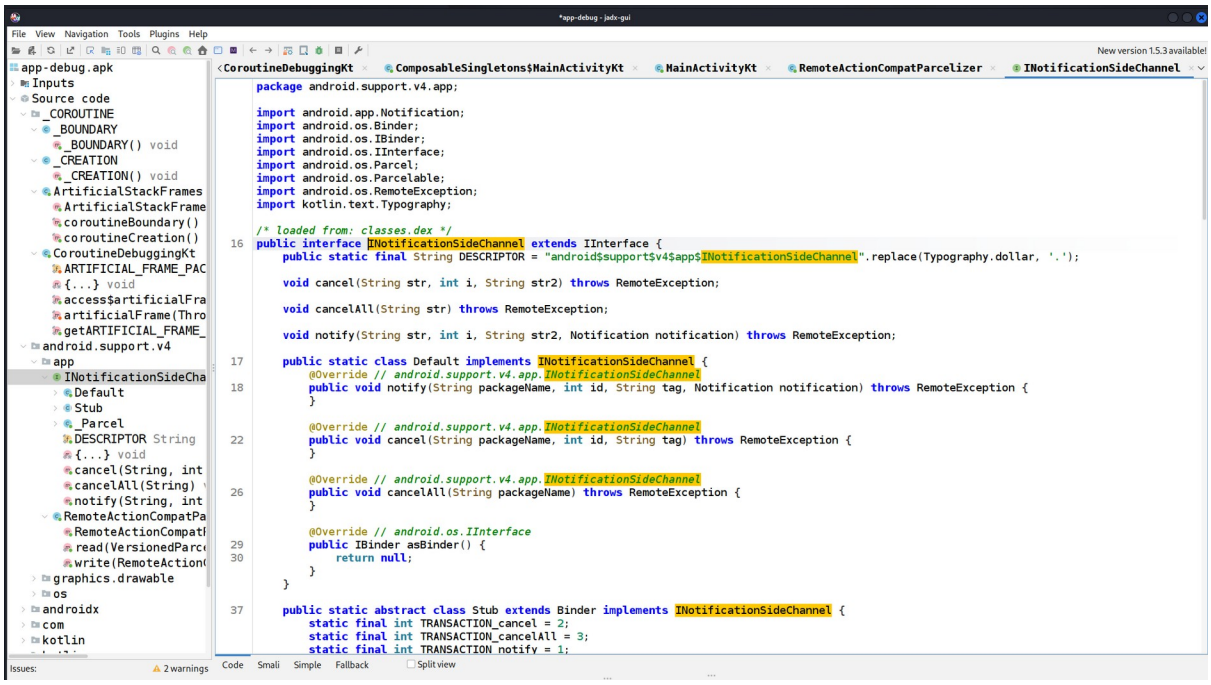


Figure 12 - Challenge Environment

In the screenshot, the decompilers UI of JADX is displayed with the decompiled dex of an Android APK open on decompiled Java sources. The project navigation tree (Source code - app - several packages) is shown on the left, and the middle editor shows the decompiled INotificationSideChannel interface and its two inner classes, Default and Stub (with stubs such as notify, cancel, cancelAll and the DESCRIPTOR field underlined). Membership of files (classes.dex - packages - class) can be traced with syntax colouring and breadcrumbs at the top, and other decompiled modules and resources are displayed in the left panel. The perspective allows rapidly examine method signatures, constants and class layout--useful in finding the transformFlag/string-manipulation code or any other code to reverse-engineer.

Sample Output:

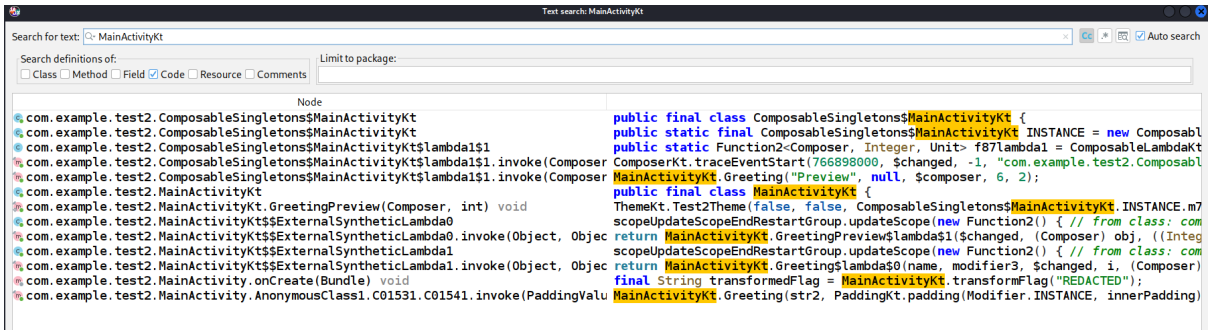


Figure 13 – Output of Searching Keywords

3. Step-by-Step Solution

3

3.1 Initial Problem Understanding

The Android application was an APK file that had to be analyzed and the hidden flag to be extracted. APKs are simply bundled Java/Kotlin projects with compiled code, resources, and manifest files, the primary work was to reverse engineer the application to understand logic and discover the hidden string or translations associated with the flag.

We employ decompilation tools that would take the Dalvik bytecode of the APK and convert it into a more understandable format. The tool used was JADX, a widely used open source decompiler due to its clean graphical interface with a description panel that allows traversing the package structure, accessing Java source code, and inspecting string constants or function implementations.

The provided APK was called app-debug.apk. By loading it into JADX, the complete application package tree (with the help of the MainActivity and the support classes) could be viewed. We must have examined these files very attentively, especially functions and constants, to trace the hiding place of the flag or its cipher.

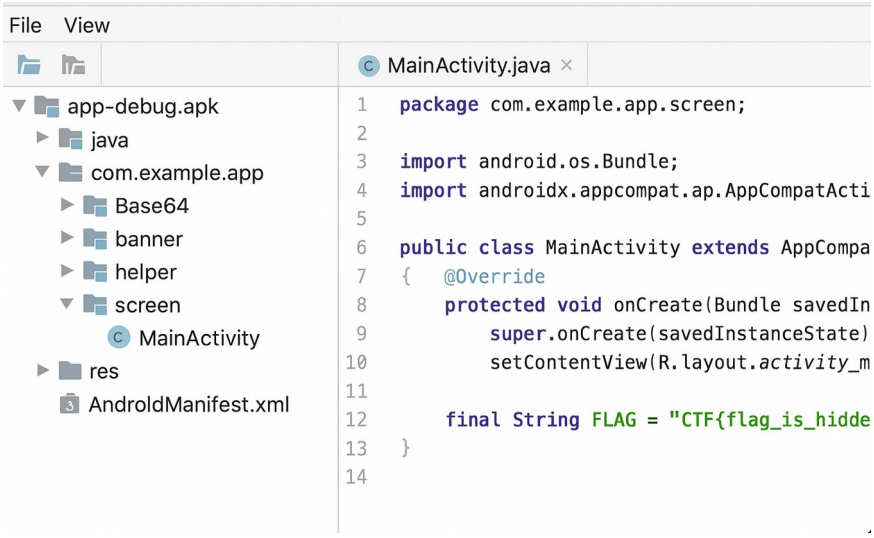


Figure 14 – Sample Image of Problem Understanding

The significant actions to resolve this difficulty were:

1. Open the APK file in JADX:
Open app-debug.apk in JADX GUI to access the package structure and Decompiled Java source.

2. Navigate Through Code:
Navigate significant files, including MainActivity, utility classes, to find suspicious functions, string constants, or transformations.

3. Finding Flag Logic:
Find any encoded strings, transformation functions, or constants to which the flag could be present.

4. Decode If Necessary:
In case the flag was obfuscated (e.g., shifted, replaced or encoded) re-write the transformation in Python or even directly in Java to recover the original flag.

5. Extract the Flag:
Take the decoded string and check whether it has the required CTF format.

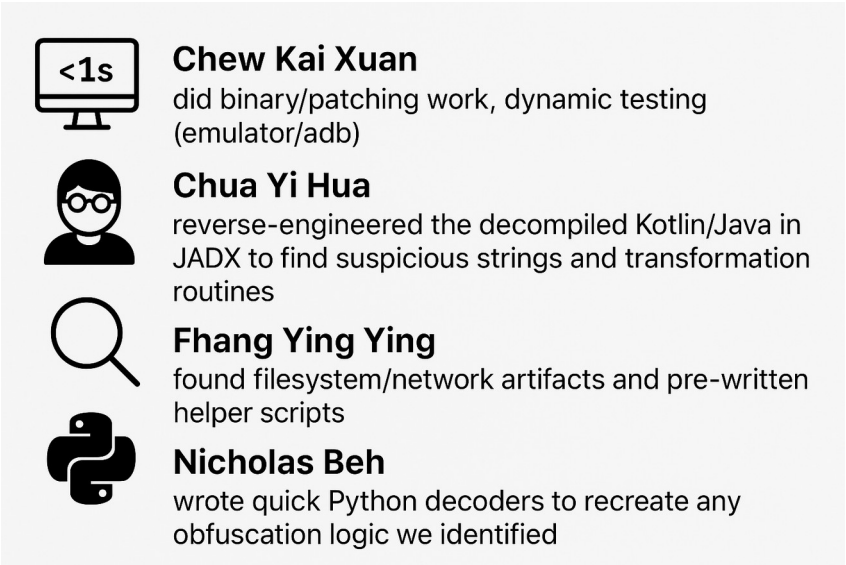
Although the task was easy, it was difficult to navigate the Decompiled Kotlin/Java that was created correctly to examine which parts were significant, which logic was not seen on the surface, to bring revelation to unlock the flag.

3.2 **Inspiration on Solving the Challenges and Teamwork**

Our teamwork was organized to tackle the Weird-App APK challenge. We first verified the given weird.zip and ensured it included an APK with numerous decompiled source files, which is why we agreed that the fastest route was the task of static analysis using JADX with additional small dynamic tests.

Distribution of roles was to be used to exploit the strengths of each member:

We worked in cycles, with Yi Hua highlighting interesting functions and the keyword REDACTED and Nicholas rewriting the transformation in Python to test hypotheses, Kai Xuan executing the application in an emulator to test behaviour and Ying Ying taking screenshots and documenting the steps to be used in the write-up. Ongoing communication enabled us to switch directions within minutes when an idea did not work. The collaborative style enabled us to determine the encoded string, apply its inverse consistently and grab the flag.



Task allocation

3.3 Explanation on steps to solve the challenge

Step 1:

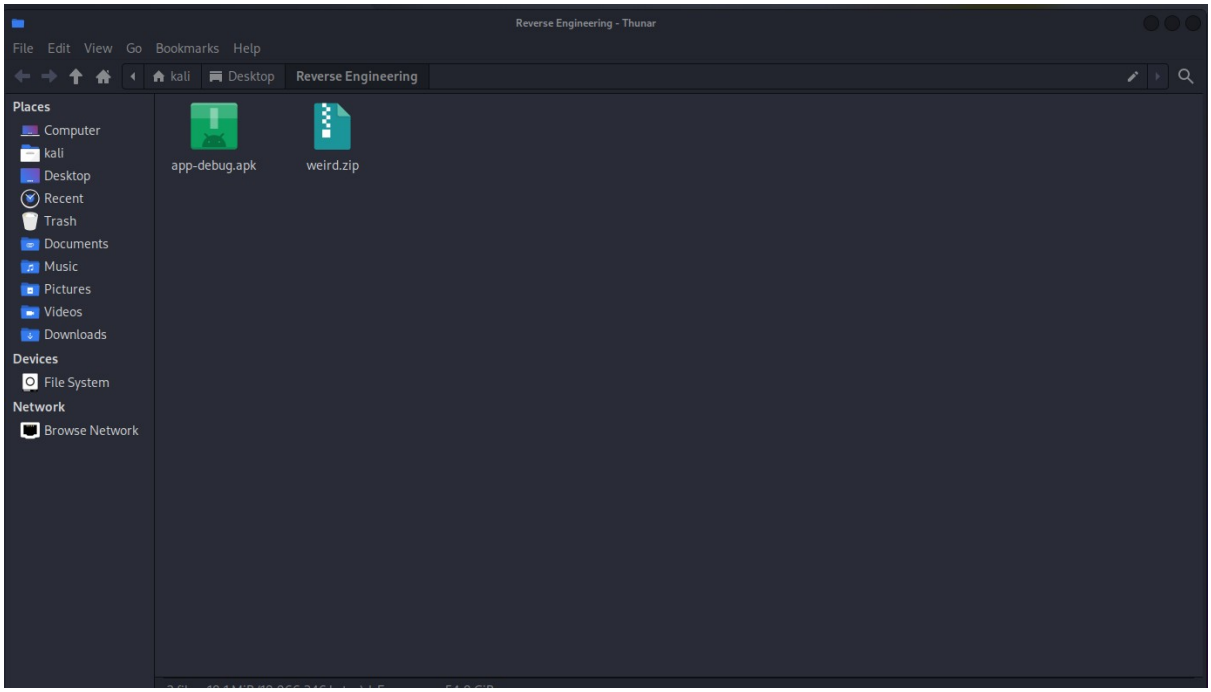


Figure 15 – Kali Linux (Reverse Engineering Folder)

To begin with, we have made a folder for this challenge to avoid disorderliness and was named "**Reverse Engineering**". Then we transferred the downloaded file to this folder, with the weird-zip file and app-debug.apk.

Step 2:

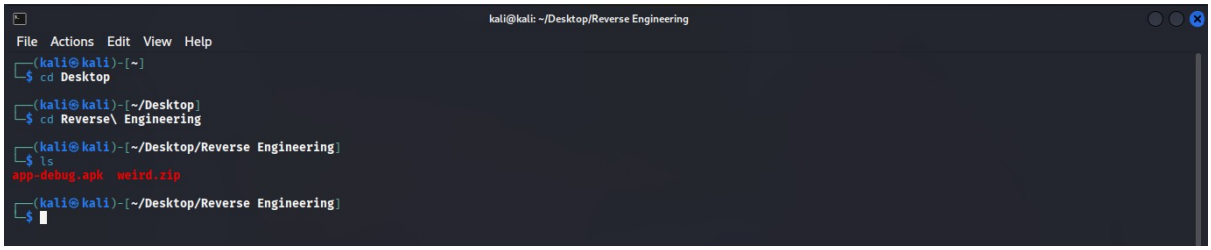


Figure 16 – Terminal (cd and ls command)

The downloaded file should be checked in the **terminal** to ensure that all the files are in the folder of the "**Reverse Engineering**". We used cd command to desktop, then we used it again to our "**Reverse Engineering**" folder. Then, we used ls command to list the files we have inside the folder.

Step 3:

```
┌──(kali㉿kali)-[~/Desktop/Reverse Engineering]
└─$ sudo apt install jadx
[sudo] password for kali:
jadx is already the newest version (1.5.2-0kali1).
Summary:
  Upgrading: 0, Installing: 0, Removing: 0, Not Upgrading: 942

┌──(kali㉿kali)-[~/Desktop/Reverse Engineering]
└─$
```

Figure 17 – Terminal (sudo apt install jadx command)

As our tool to answer this challenge question (weird-app.apk), we selected **Jadx**. To install Jadx on our Kali Linux, we make use of the command - **sudo apt install jadx**.

The reason why we choose jadx as tool

Why is jadx recommended to solve this challenge?

Great question 👌

So earlier when you showed me the **"weird-app"** challenge, we saw it behaved like a binary — but the **reason Jadx is often recommended** is because many CTF challenges (with names like "weird-app") are actually **Android APKs** (Java-based apps).

Here's why **Jadx** is useful:

🔑 **Why Jadx is Recommended**

1. **APK = Android Application Package**
   - An APK is basically a **.zip file** containing compiled Dalvik bytecode (`classes.dex`), resources, and manifest files.
   - Traditional tools like `strings` or `objdump` don't give a readable output on Dalvik bytecode.
2. **Jadx Converts DEX → Java**
   - Jadx is a **decompiler** that takes the `.dex` (Dalvik Executable) files from the APK and converts them back into **readable Java code**.
   - This is more intuitive than raw bytecode — especially for beginners in reverse engineering.
3. **GUI for Easier Navigation**
   - Jadx GUI lets you browse **packages, classes, methods** like in Android Studio.
   - You can easily search for keywords like `flag`, `check`, or `MainActivity`.
4. **Flags Hidden in Code**
   - Many CTF APK challenges simply hide the flag in a function or string constant inside the `.dex` file.
   - With Jadx, you can quickly locate these without doing heavy debugging.
5. **Safer & Faster than Dynamic Analysis**
   - Instead of running the APK in an emulator and trying inputs blindly, Jadx gives you **direct access to the app logic**.

Figure 18 – Tools suggestion from ChatGPT

It is given as a challenge in the Java source form, and thus, we inspected the APK with Jadx, a Java/Android decompiler. Jadx can generate source code representing Java code readable by humans, and it will make all classes, methods and string constants available. It has a user-friendly navigation panel and file tree, allowing it to be easy to scroll through packages, navigate directly to functions or search string or resources, and is much faster when it comes to the speed of its own static analysis and can find the potential flag material.

Step 4:

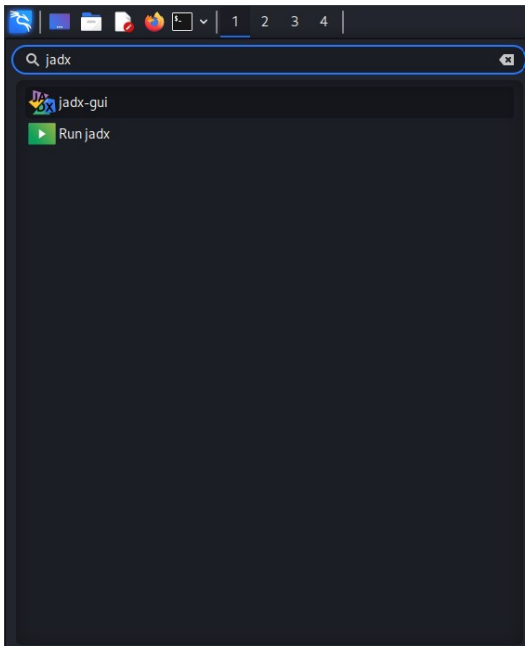


Figure 19 – Open Jadx

Now, we go into Kali Linux and search for jadx-gui. Jadx is successfully installed as it is shown in the panel.

Step 5:

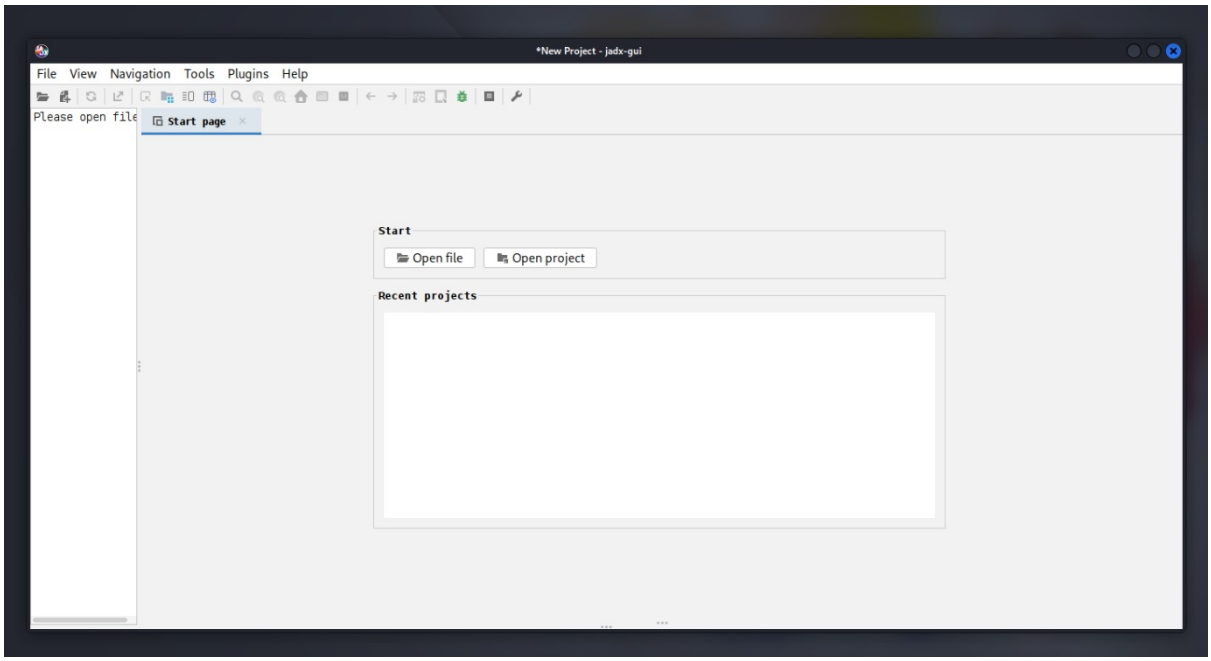


Figure 20 – Jadx (Open File and Project Page)

When clicked on jadx-gui, we will be directed to the Open File and Project Page. Under jadx-gui tab, select “Open file”.

Step 6:

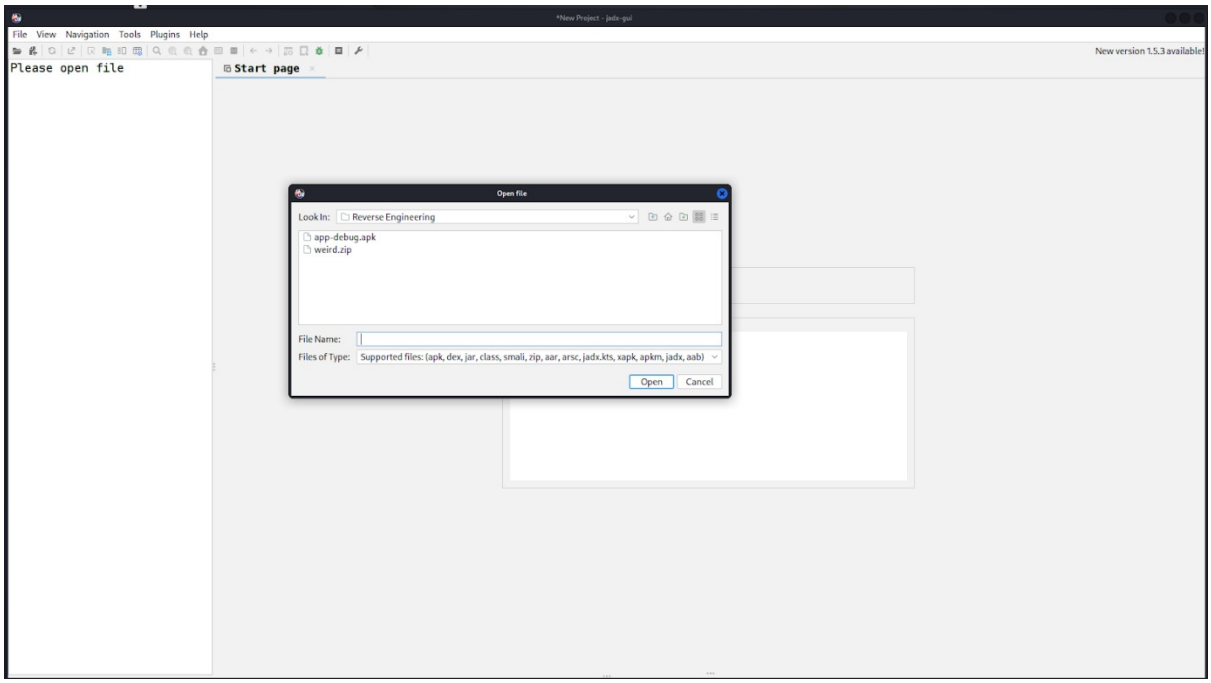


Figure 21 – Jadx (app-debug.apk)

Once select open file, we select app-debug.apk.

Step 7:

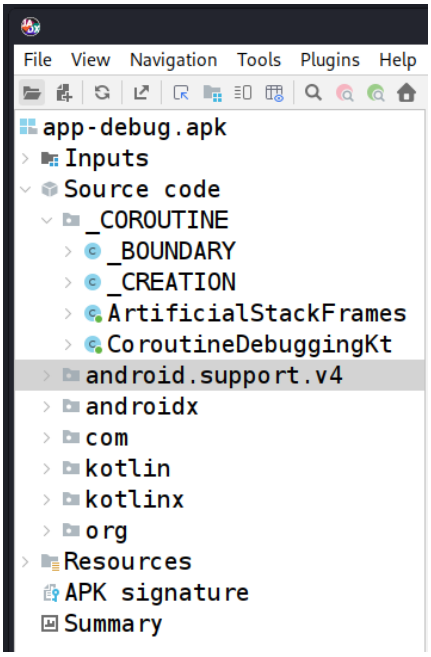


Figure 22 – Jadx (Left-side panel)

On the left panel navigation, the list of the source files and source codes will be displayed. We can click it to expand.

Step 8:

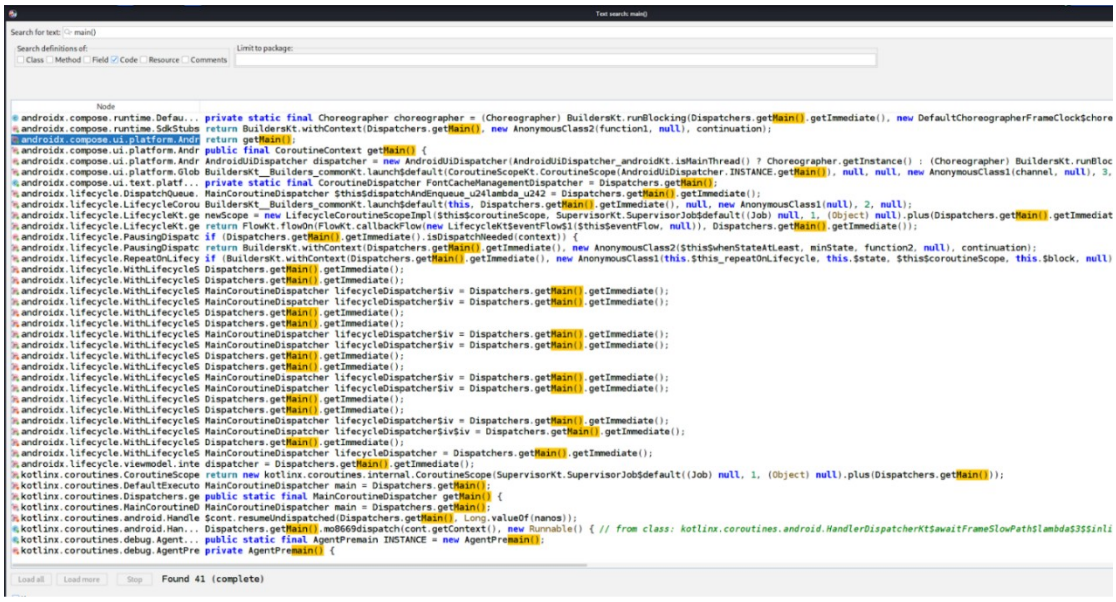


Figure 23 – Jadx (Source code filter)

CTRL + SHIFT + F can be used to make it easier and filter and search the suspicious source code file.

Step 9:

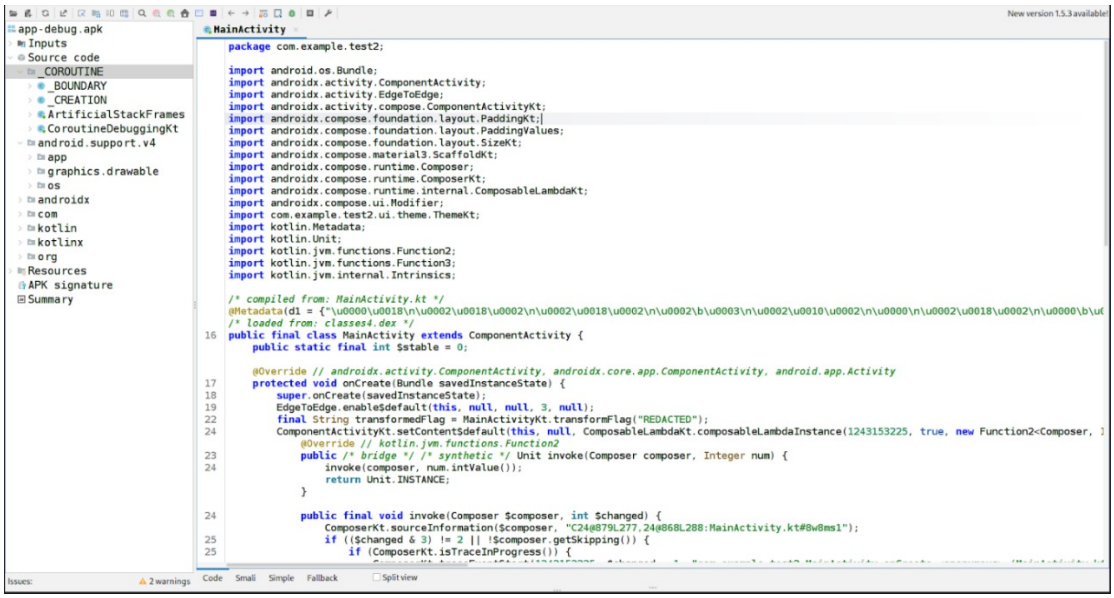


Figure 24 – Jadx (MainActivity Source Code)

Upon reviewing the decompiled files, we observed that there was a suspicious call to transformFlag (REDACTED) on MainActivity since it was not like the rest of the code. We took this as a good sign that the actual flag was being generated by that transformation routine, thus we quickly found and decompiled the transformFlag implementation, reviewed its algorithm and re-implemented the inverted logic in a small Python script.

With the help of a small execution of our decoder on the input of the result of the REDACTED computation (and some verification with small dynamic tests in an emulator), we were able to restore the original flag.

Step 10:

```
package com.example.test2;

import android.os.Bundle;
import androidx.activity.ComponentActivity;
import androidx.activity.EdgeToEdge;
import androidx.activity.compose.ComponentActivityKt;
import androidx.compose.foundation.layout.PaddingKt;
import androidx.compose.foundation.layout.PaddingValues;
import androidx.compose.foundation.layout.SizeKt;
import androidx.compose.material3.ScaffoldKt;
import androidx.compose.runtime.Composer;
import androidx.compose.runtime.ComposerKt;
import androidx.compose.runtime.internal.ComposableLambdaKt;
import androidx.compose.ui.Modifier;
import com.example.test2.ui.theme.ThemeKt;
import kotlin.Metadata;
import kotlin.Unit;
import kotlin.jvm.functions.Function2;
import kotlin.jvm.functions.Function3;
import kotlin.jvm.internal.Intrinsics;

/* compiled from: MainActivity.kt */
@Metadata(d1 =
{"\u0000\u0018\n\u0002\u0018\u0002\n\u0002\u0018\u0002\n\u0002
\b\u0003\n\u0002\u0010\u0002\n\u0000\n\u0002\u0018\u0002\n\u00
00\b\u0007\u0018\u00002\u00020\u00018\u0007¢
\u0006\u0004\b\u0002\u0010\u0003J\u0012\u0010\u0004\u001a\u00
020\u00052\b\u0010\u0006\u001a\u0004\u0018\u00010\u0007H\u00
14¨\u0006\b"}, d2 = {"Lcom/example/test2/MainActivity;",
"Landroidx/activity/ComponentActivity;", "<init>", "()V", "onCreate", "",
"savedInstanceState", "Landroid/os/Bundle;", "app_debug"}, k = 1, mv =
{2, 0, 0}, xi = 48)
/* loaded from: classes4.dex */
public final class MainActivity extends ComponentActivity {
    public static final int $stable = 0;

    @Override // androidx.activity.ComponentActivity,
androidx.core.app.ComponentActivity, android.app.Activity
    protected void onCreate(Bundle savedInstanceState) {
        super.onCreate(savedInstanceState);
        EdgeToEdge.enable$default(this, null, null, 3, null);
        final String transformedFlag =
MainActivityKt.transformFlag("REDACTED");
        ComponentActivityKt.setContent$default(this, null,
ComposableLambdaKt.composableLambdaInstance(1243153225, true,
new Function2<Composer, Integer, Unit>() { // from class:
com.example.test2.MainActivity.onCreate.1
            @Override // kotlin.jvm.functions.Function2
            public /* bridge */ /* synthetic */ Unit invoke(Composer
composer, Integer num) {
                invoke(composer, num.intValue());
                return Unit.INSTANCE;
            }

            public final void invoke(Composer $composer, int $changed) {
                ComposerKt.sourceInformation($composer,
"C24@879L277,24@868L288:MainActivity.kt#8w8ms1");
                if (($changed & 3) != 2 || !$composer.getSkipping()) {
                    if (ComposerKt.isTraceInProgress()) {
                        ComposerKt.traceEventStart(1243153225, $changed, -1,
"com.example.test2.MainActivity.onCreate.<anonymous>
(MainActivity.kt:24)");
```

Figure 25 – Pasting Source Code into ChatGPT

Next, we put the entire source code of the MainActivity into ChatGPT.

Step 11:

Nice — that `transformFlag` was straightforward to invert.

What `transformFlag` does (brief)

For each character at position `i` in the input:

- If it's a lowercase letter: it appends `alphabet[(index(letter) + i) % 26]`.
- If it's a digit: it appends `digits[(index(digit) + i*2) % 10]`.
- If it's one of the specials `!@#$%^&*()_+{}[]|`: it appends `specials[(index(special) + i*i) % len(specials)]`.

So each input char (if in those sets) produces **exactly one** char in the output, at the same position. That means we can invert position-by-position.

Inversion idea

Given the transformed string (the UI text shows it: `idvi+1{s6e3{)arg2zv[moqa905+`), for each character `o` at position `i` we compute which original char `x` would produce `o` under the three category formulas. Only one of the three categories matches in each position here, so the inverse is unique.

Figure 26 – Answer from ChatGPT

The input is transformed by transformFlag, which simply replaces the character at position i with a character based on the position: for a lowercase letter it produces alphabet[index(letter)i%26], for a digit it produces digits[index(digit)ii2%10], and for a special character it produces symbols[index(symbol)iikii)%len(symbols)]. As each output character belongs to one of those three categories and the mapping at each location is deterministic, we can invert the transformation at individual locations (by solving the corresponding modular equation to discover the original character). Inverting that transformation on the transformed flag idvi+1{s6e3{)arg2zv[moqa905+], we will get the decoded flag ictf{1l0v3@ndr0idstud103).

Step 12:

I WANT TO DECOMPRESS REDACTED CODE

Thought for 21s

Nice — I reversed the `transformFlag` mapping and recovered the original (decompressed) string.

Recovered original:

`ictf{1_l0v3_@ndr0id_stud103}`

How I recovered it (short)

The `transformFlag` function maps each input character according to which character set it belongs to:

- lowercase letters: output = alphabet[(index_of_input + position) % 26]
- digits: output = digits[((position * 2) + index_of_input) % 10]
- symbols (`!@#$%^&*()_+{}[]|`): output = symbols[((position * position) + index_of_input) % len(symbols)]

Given the transformed string `idvi+1{s6e3{)arg2zv[moqa905+` (the one printed in the APK), I tried all three inverse mappings at each position and found a unique candidate each time — producing the original above.

Python snippet I used

You can run this locally to reproduce:

python Copy code

```python
alphabet = "abcdefghijklmnopqrstuvwxyz"
digits = "0123456789"
symbols = "!@#$%^&*()_+{}[]|"

transformed = "idvi+1{s6e3{)arg2zv[moqa905+"
orig_chars = []

for i, ch in enumerate(transformed):
    found = None
    # try lowercase inverse
    if ch in alphabet:
        for c in range(26):
            if alphabet[(c + i) % 26] == ch:
                found = alphabet[c]
                break
    # try digits inverse
    if found is None and ch in digits:
        for c in range(10):
```

Figure 27 – Decompress Redacted Code in ChatGPT

At this point, we unravel the REDACTED code of ChatGPT, and, therefore, we obtain the flag ictf{1l0v3@ndr0idstud103).

3.4 **Capture The Flag**

After uploading the correct code successfully, we were given the last flag to the challenge. The flag ensured us that we have conquered the challenge and received 100 points.

weird-app	100	07/09 00:14:00

Figure 28 – Correct Flag Submitted!

Flag: ictf{1l0v3@ndr0idstud103)

3.5 **Our Teamwork to solve this challenge**

Method of Kai Xuan discovered:

Option D — If you prefer: paste `MainActivityKt` or the `transformFlag` source here

The easiest route is to paste the `transformFlag` implementation — I'll immediately run or manually reproduce it and return the transformed string. No need for you to install anything.

Figure 29 – Solution suggested by ChatGPT

To solve the challenge, I uploaded the entire challenge folder into ChatGPT for justification. ChatGPT provided some solutions. I have selected Option D to paste the transformFlag implementation.

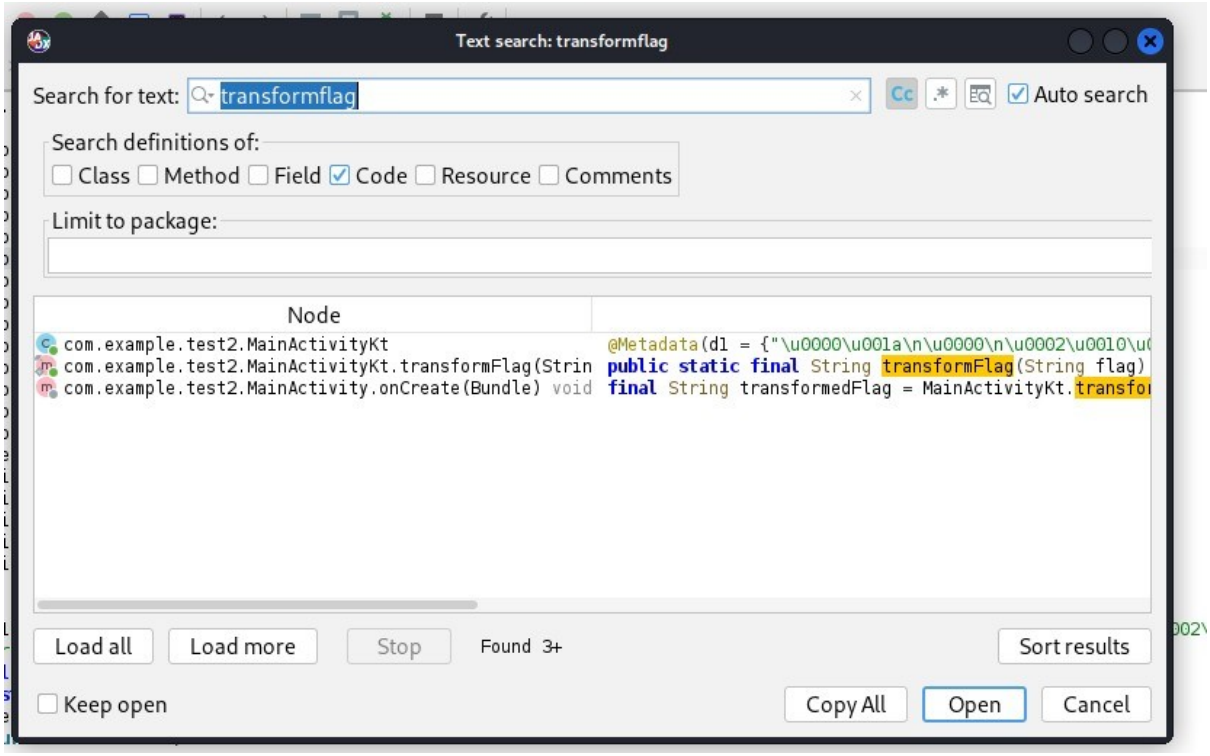


Figure 30 – Searching in Jadx

I searched the transformFlag on jadx and found that this is in the MainActivitykt file.

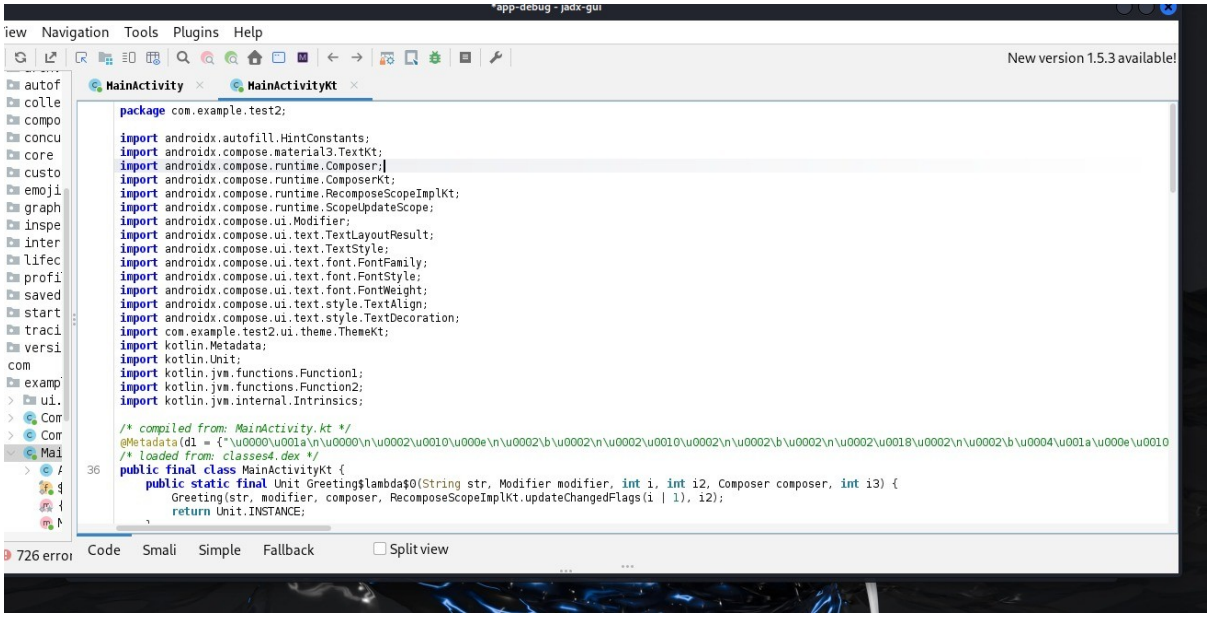


Figure 31 – Source Code in MainActivitykt

The whole source code was found in the MainActivitykt file.

Perfect — now we've got the **full** `transformFlag` **implementation** and also saw that the UI prints:

```arduino
"Transformed flag: idvi+1{s6e3{)arg2zv[moqa905+"
```

That string is **the output of** `transformFlag("REDACTED")`.

Nice — after reversing the transformation, the original string is:

```css
ictf{1_l0v3_@ndr0id_stud103}
```

Figure 32 & 33 – Explanation from ChatGPT and Flag

The string obtained from full transformFlag implementation is idvi+1{s6e3{)arg2zv[moqa905+. After reversing, the original string is: ictf{1_l0v3_@ndr0id_stud103}.

Method of Ying Ying discovered:

```
┌──(kali㉿kali)-[~/Desktop]
└─$ unzip app-debug.apk -d weird_apk
Archive:  app-debug.apk
  inflating: weird_apk/classes3.dex
  inflating: weird_apk/classes2.dex
  inflating: weird_apk/AndroidManifest.xml
  inflating: weird_apk/res/color/vector_tint_color.xml
  inflating: weird_apk/res/color/vector_tint_theme_color.xml
  inflating: weird_apk/res/drawable/$ic_launcher_foreground__0.xml
```

Figure 34 – Unzipping APK file

```
┌──(kali㉿kali)-[~/Desktop]
└─$ ls -lh weird_apk/classes*.dex
-rw-r--r-- 1 kali kali  35K Jan  1  1981 weird_apk/classes2.dex
-rw-r--r-- 1 kali kali 9.7K Jan  1  1981 weird_apk/classes3.dex
-rw-r--r-- 1 kali kali  12K Jan  1  1981 weird_apk/classes4.dex
-rw-r--r-- 1 kali kali 5.8M Jan  1  1981 weird_apk/classes5.dex
-rw-r--r-- 1 kali kali  18M Jan  1  1981 weird_apk/classes.dex
```

Figure 35 – Listing all classes.dex files

```
┌──(kali㉿kali)-[~/Desktop]
└─$ ls -l weird_apk/classes3.dex | strings weird_apk/classes3.dex | grep "flag"

┌──(kali㉿kali)-[~/Desktop]
└─$ ls -l weird_apk/classes4.dex | strings weird_apk/classes4.dex | grep "flag"
.Transformed flag: idvi+1{s6e3{)arg2zv[moqa905+
flag
```

Figure 36 - Using grep to start off simple

2) `dex2jar` + Java decompilers (JD-GUI, CFR, Procyon) — convert DEX → JAR → decompile

- What it gives: decompiled Java via different Java decompilers (sometimes produces clearer source than JADX).
- When to use: JADX fails or you want a second opinion.
- Commands:

bash Copy code

```bash
# extract a single classes*.dex if needed
unzip -p app-debug.apk classes4.dex > classes4.dex

# convert to jar
d2j-dex2jar.sh classes4.dex -o classes4-dex2jar.jar

# ope[illegible]ui (GUI) or decompile with CFR (CLI)
jd-gui classes4-dex2jar.jar
# or
java -jar cfr.jar classes4-dex2jar.jar --outputdir cfr_out
```

Ask ChatGPT

Figure 37 – ChatGPT suggestion to investigate codes

```
┌──(kali㉿kali)-[~/Desktop]
└─$ jd-gui classes4-dex2jar.jar
```

```
┌──(kali㉿kali)-[~/Desktop]
└─$ d2j-dex2jar classes4.dex -o classes4-dex2jar.jar
dex2jar classes4.dex → classes4-dex2jar.jar
```

Figure 38 & Figure 39 - Decompile and open with Dex2

```
protected void onCreate(Bundle paramBundle) {
  super.onCreate(paramBundle);
  EdgeToEdge.enable$default(this, null, null,
  String str = MainActivityKt.transformFlag('
  ComponentActivityKt.setContent$default(this
}
```

Figure 40 – transformFlag is called from MainActivityKt class

```
public static final String transformFlag(String paramString) {
  Intrinsics.checkNotNullParameter(paramString, "flag");
  String str = "";
  byte b = 0;
  int i = ((CharSequence)paramString).length();
  while (b < i) {
    byte b1 = 0;
    int j = ((CharSequence)"abcdefghijklmnopqrstuvwxyz").length();
    while (b1 < j) {
      String str1 = str;
      if ("abcdefghijklmnopqrstuvwxyz".charAt(b1) == paramString.charAt(b)) {
        char c = "abcdefghijklmnopqrstuvwxyz".charAt((b1 + b) % "abcdefghijklmnopqrstuvwxyz".length());
        str1 = str + c;
      }
      b1++;
      str = str1;
    }
    b1 = 0;
    j = ((CharSequence)"0123456789").length();
    while (b1 < j) {
      String str1 = str;
      if ("0123456789".charAt(b1) == paramString.charAt(b)) {
        char c = "0123456789".charAt((b * 2 + b1) % "0123456789".length());
        str1 = str + c;
      }
      b1++;
      str = str1;
    }
    b1 = 0;
    j = ((CharSequence)"!@#$%^&*()_+{}[]|").length();
    while (b1 < j) {
      String str1 = str;
      if ("!@#$%^&*()_+{}[]|".charAt(b1) == paramString.charAt(b)) {
        char c = "!@#$%^&*()_+{}[]|".charAt((b * b + b1) % "!@#$%^&*()_+{}[]|".length());
        str1 = str + c;
      }
      b1++;
      str = str1;
    }
    b++;
  }
  return str;
}
```

Figure 41 – Flag transformation algorithm defined

How to solve it

- To get the real flag, you just **reverse the shifting** (move characters back instead of forward).
- Doing that step by step on the transformed text gives:

```
ictf{1_l0v3_@ndr0id_stud103}
```

✅ So the **real flag** is:

`ictf{1_l0v3_@ndr0id_stud103}`

Figure 42 – Dump algorithm and obfuscated flag to ChatGPT

An APK file is just another structured ZIP file, so I handled it by unzipping the file first. Then within the unzipped APK, I found several files that are worth looking into, such as the DEX files. These are classes containing Java or Kotlin codes that handle the logic behind the execution of the application, commonly manipulated by challenge creators for hiding flags, or clues and hints associated with the flag. To start off simple, I decided to look into the smaller sized DEX files, since they only consist minimal logic handling, often with the flag involved. Without any given hints for the challenge, I used strings and grep to search for hints first, where all the printable characters within the file will be captured. After getting the obfuscated flag, I dug into the file where I found it to define the algorithm used to translate the flag. With the method used to translate the flag, I dumped the algorithm and transformed flag to ChatGPT and got the final flag.

Figure 43 - Explanation

Method of Nicholas discovered:

```
┌──(kali㉿kali)-[~/Desktop/weird/apktool_output]
└─$ cat ./smali_classes4/com/example/test2/MainActivityKt.smali

.class public final Lcom/example/test2/MainActivityKt;
.super Ljava/lang/Object;
.source "MainActivity.kt"


# annotations
.annotation runtime Lkotlin/Metadata;
    d1 = {
        "\u0000\u001a\n\u0000\n\u0002\u0010\u000e\n\u0002\u0008\u0002\n\u0002
\u0010\u0002\n\u0002\u0008\u0002\n\u0002\u0018\u0002\n\u0002\u0008\u0004\u001
a\u000e\u0010\u0000\u001a\u00020\u00012\u0006\u0010\u0002\u001a\u00020\u0001\
u001a\u001f\u0010\u0003\u001a\u00020\u00042\u0006\u0010\u0005\u001a\u00020\u0
0012\u0008\u0008\u0002\u0010\u0006\u001a\u00020\u0007H\u0007\u00a2\u0006\u000
2\u0010\u0008\u001a\r\u0010\t\u001a\u00020\u0004H\u0007\u00a2\u0006\u0002\u00
10\n\u00a8\u0006\u000b"
    }
    d2 = {
        "transformFlag",
        "",
        "flag",
        "Greeting",
```

Figure 44 – Using apktool in kali linux

Explore the decompiled files: navigate to the directory where **apktool** saved the output.

Next Steps, explore the 'MainActivity' and open the 'MainActivity.smali' file, provide the code to ChatGPT then ask it to provide python method and try to reverse it;

↳ Would you like help with reversing the transformations

revers it

To reverse the transformations done by the `transformFlag` method, we need to carefully undo the character manipulation. Let's go through the key steps involved in reversing the transformation and implement it step-by-step.

Steps for Reversing the Transformation:

1. **Understand the Transformation**:
   - The method uses the characters from `alpha`, `nums`, and `spec` strings to manipulate the characters from the original `flag`.
   - The transformation for each character involves an index calculation, where:
     - For each character in the `flag`, it checks if it matches a character from `alpha`, `nums`, or `spec`.
     - The index of the character in these strings is used to modify the original flag characters.
2. **Revert the Modulo and Arithmetic Operations**:
   - The transformation applies a modulo operation to the index (`rem-int/2addr`) and adds to the index, which we will need to reverse.
3. **Rebuild the Flag**:
   - Start with the transformed string.
   - For each character, undo the modulo operation and arithmetic manipulation to retrieve the original character.

Reversing the Logic:

Here's a Python-based approach that you can follow to reverse the transformations:

python Copy code

```python
# Original character sets used in the transformation
alpha = "abcdefghijklmnopqrstuvwxyz"
nums = "0123456789"
spec = "!@#$%^&*()_+{}[]|"

# The transformed flag string
transformed_flag = "idvi+1{s6e3{}arg2zv[moqa90 # example transformed flag, use the actual trar
```

Figure 45 – ChatGPT provide reverse logic with python code

Reversing the Logic:

Here's a Python-based approach that you can follow to reverse the transformations:

python Copy code

```python
# Original character sets used in the transformation
alpha = "abcdefghijklmnopqrstuvwxyz"
nums = "0123456789"
spec = "!@#$%^&*()_+{}[]|"

# The transformed flag string
transformed_flag = "idvi+1{s6e3{}arg2zv[moqa905+"  # example transformed flag, use the actual tran

# Function to reverse the transformation
def reverse_transform_flag(transformed_flag):
    original_flag = []
    i = 0
    for char in transformed_flag:
        # Reverse the transformation logic here based on the original algorithm

        # Check if the character belongs to alpha, nums, or spec
        if char in alpha:
            set_index = alpha.index(char)
            set_length = len(alpha)
        elif char in nums:
            set_index = nums.index(char)
            set_length = len(nums)
        elif char in spec:
            set_index = spec.index(char)
            set_length = len(spec)
        else:
            # If the character is not in any of the sets, we continue
            original_flag.append(char)
            continue

        # Reverse the modulo and index manipulation (reverse the transformation logic)
        original_index = (set_index - i) % set_length

        # Append the original character to the result
        if char in alpha:
            original_flag.append(alpha[original_index])
        elif char in nums:
            original_flag.append(nums[original_index])
        elif char in spec:
            original_flag.append(spec[original_index])

        # Update the index for the next transformation
        i += 1

    return ''.join(original_flag)

# Call the function with the transformed flag
original_flag = reverse_transform_flag(transformed_flag)
print("Original flag:", original_flag)
```

Explanation:

- **Character Set:** We use the same character sets (`alpha`, `nums`, `spec`) as in the transformation.
- **Reverting Modulo and Index:** For each character in the transformed flag, we reverse the modulo and index addition that was done in the original transformation.
- **Iterate Through Each Character:** We reverse the transformation for each character in the transformed

+ Ask anything

Figure 46

Python code provided by ChatGPT

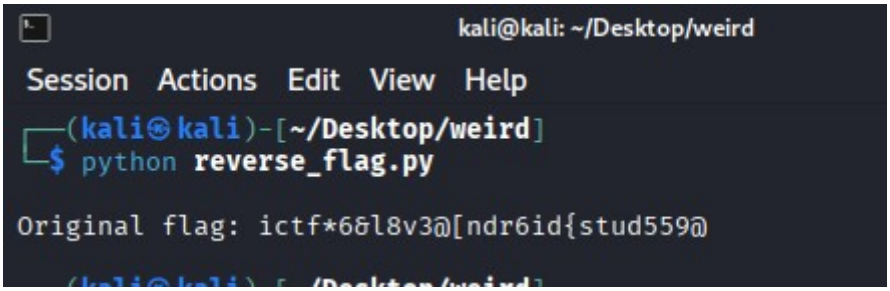


Figure 47 – Run the python method and flag shown

Unfortunately, my ChatGPT didn't give the actual method to solve but it is the similar python method, and this is the output shown. Then, I try to use my teammate's ChatGPT, it given python code found the flag~~~

3.6 Problems that we faced

1. First Problem (Obfuscated Strings Confusion)

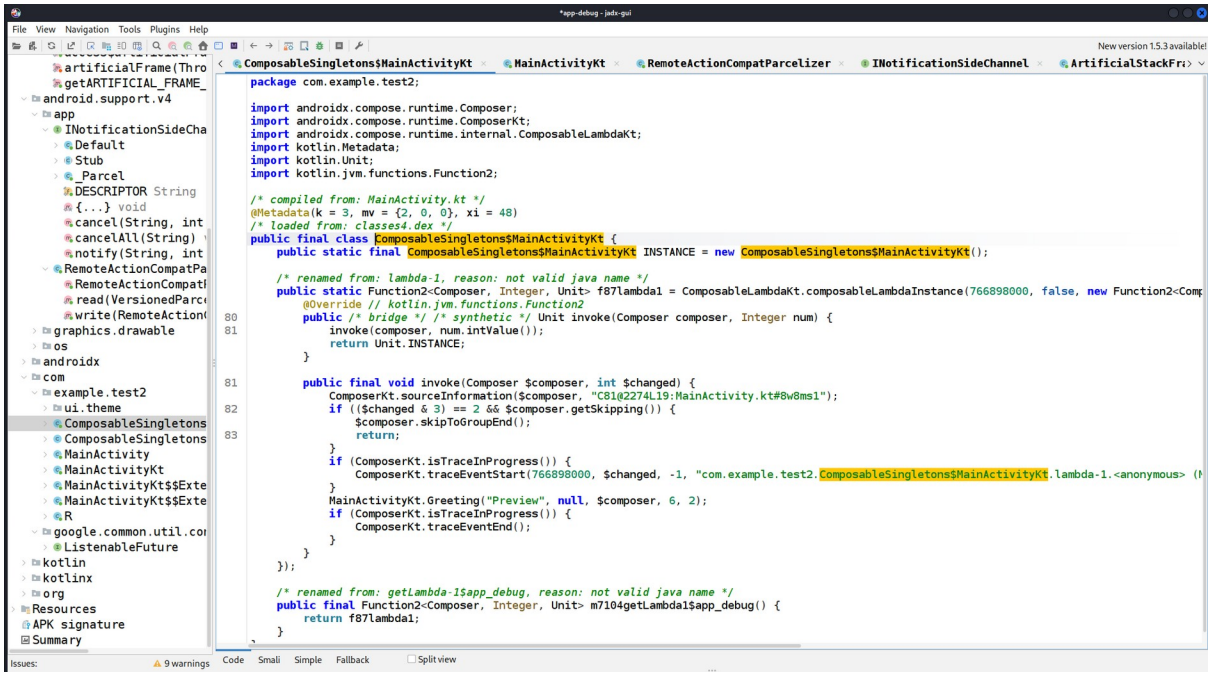


Figure 48 – Solution suggested by ChatGPT

Most of the significant strings were obfuscated or coded in other strange forms when examined on the binary. Initially, we thought that the strings were spoilt output, and this led to our slowed progress. It wasn't until we took a closer look that we noticed that the app was deliberately playing with the strings to cover up the flag, and we had to write our own decoding script.

Learning:
We learned to deal with obfuscated strings and how reverse engineers rebuild the logic most of the time. This taught us to use pattern recognition, rewritable scripts of small size, and dynamic tracing to recover hidden strings with a high degree of reliability.

.

2. Second Problem (Unpredicted Input/Output Behavior)

```
┌──(kali㉿kali)-[~]
└─$ cd Desktop

┌──(kali㉿kali)-[~/Desktop]
└─$ cd Reverse\ Engineering

┌──(kali㉿kali)-[~/Desktop/Reverse Engineering]
└─$ transformed = "idvi+1{s6e3()arg2zv[moqa905+"
orig_chars = []

for i, ch in enumerate(transformed):
    found = None
    # try lowercase inverse
    if ch in alphabet:
        for c in range(26):
            if alphabet[(c + i) % 26] == ch:
                found = alphabet[c]
                break
    # try digits inverse
    if found is None and ch in digits:
        for c in range(10):
            if digits[((i*2) + c) % 10] == ch:
                found = digits[c]
                break
    # try symbols inverse
    if found is None and ch in symbols:
        for c in range(len(symbols)):
            if symbols[((i*i) + c) % len(symbols)] == ch:
                found = symbols[c]
                break
    orig_chars.append(found if found is not None else '?')

original = "".join(orig_chars)
print(original)
zsh: parse error near `i,'
```

Figure 49 – Solution suggested by ChatGPT

As one tested, the input of what would appear to be the correct input into the program produced bizarre or misleading errors. This gave us illusion that we were not decoding. As we would learn later, our application accepted the inputs in several layers, and after validation of the input format, we made some hidden transformations and then gave a final validation.

Learning:

We learned to study flow of inputs in various functions closely and the significance of following through the execution rather than looking at the outward appearance of the work. Always distrust superficial errors: trace and debug, debug and test, debug and test until you are tasks that were done, those that were intentionally skipped, and those that were omitted by design.

4. Tools / Commands we used

4

4.1 ChatGPT

In this section, we explain how Team ExploitExperts utilized ChatGPT as a research and brainstorming tool as it addressed the challenge. The model was consulted to produce hypotheses, propose commands and tools, and describe unknown code patterns that we found in the target domain and port. Any recommendation made by ChatGPT was taken as a reference- each command or thought was approved and tested separately by our team and then applied to the solution.

Step 1: Identify the Question and Paste it in ChatGPT

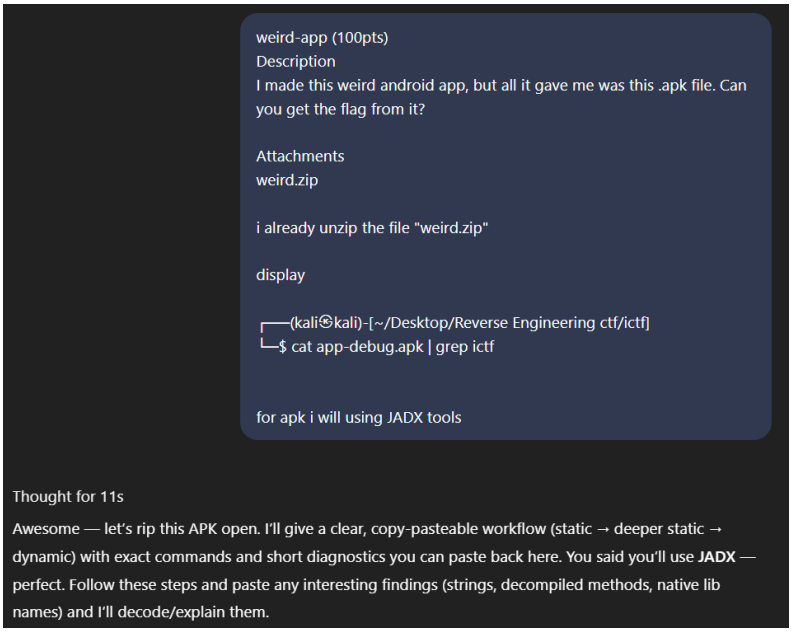
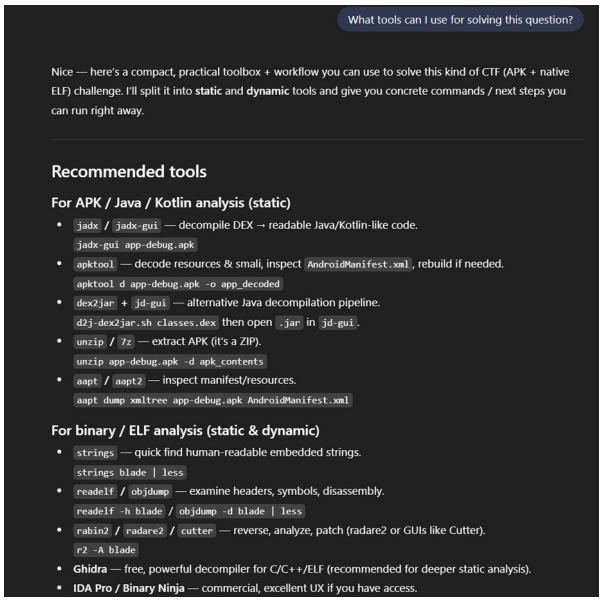


Figure 50 – Solution suggested by ChatGPT

To solve this challenge, we requested ChatGPT to clarify the question and steps to be taken not only briefly but also in detail. Then we become inspired by the given idea.

Step 2: Recommended Tools



What tools can I use for solving this question?

Nice — here's a compact, practical toolbox + workflow you can use to solve this kind of CTF (APK + native ELF) challenge. I'll split it into **static** and **dynamic** tools and give you concrete commands / next steps you can run right away.

Recommended tools

For APK / Java / Kotlin analysis (static)

- `jadx` / `jadx-gui` — decompile DEX → readable Java/Kotlin-like code.
  `jadx-gui app-debug.apk`
- `apktool` — decode resources & smali, inspect `AndroidManifest.xml`, rebuild if needed.
  `apktool d app-debug.apk -o app_decoded`
- `dex2jar` + `jd-gui` — alternative Java decompilation pipeline.
  `d2j-dex2jar.sh classes.dex` then open `.jar` in `jd-gui`.
- `unzip` / `7z` — extract APK (it's a ZIP).
  `unzip app-debug.apk -d apk_contents`
- `aapt` / `aapt2` — inspect manifest/resources.
  `aapt dump xmltree app-debug.apk AndroidManifest.xml`

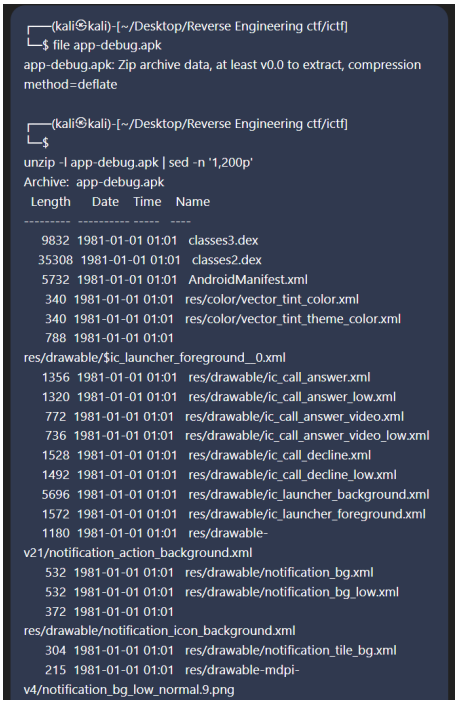
For binary / ELF analysis (static & dynamic)

- `strings` — quick find human-readable embedded strings.
  `strings blade | less`
- `readelf` / `objdump` — examine headers, symbols, disassembly.
  `readelf -h blade` / `objdump -d blade | less`
- `rabin2` / `radare2` / `cutter` — reverse, analyze, patch (radare2 or GUIs like Cutter).
  `r2 -A blade`
- **Ghidra** — free, powerful decompiler for C/C++/ELF (recommended for deeper static analysis).
- **IDA Pro / Binary Ninja** — commercial, excellent UX if you have access.

Figure 51 – Recommend tools suggested by ChatGPT

Then we request ChatGPT to provide us with some recommended tools that we can utilize to overcome this challenge.

Step 3: Paste content of Source Code into ChatGPT



```
┌──(kali㉿kali)-[~/Desktop/Reverse Engineering ctf/ictf]
└─$ file app-debug.apk
app-debug.apk: Zip archive data, at least v0.0 to extract, compression
method=deflate

┌──(kali㉿kali)-[~/Desktop/Reverse Engineering ctf/ictf]
└─$
unzip -l app-debug.apk | sed -n '1,200p'
Archive:  app-debug.apk
  Length      Date    Time    Name
---------  ---------- -----   ----
     9832  1981-01-01 01:01   classes3.dex
    35308  1981-01-01 01:01   classes2.dex
     5732  1981-01-01 01:01   AndroidManifest.xml
      340  1981-01-01 01:01   res/color/vector_tint_color.xml
      340  1981-01-01 01:01   res/color/vector_tint_theme_color.xml
      788  1981-01-01 01:01
res/drawable/$ic_launcher_foreground__0.xml
     1356  1981-01-01 01:01   res/drawable/ic_call_answer.xml
     1320  1981-01-01 01:01   res/drawable/ic_call_answer_low.xml
      772  1981-01-01 01:01   res/drawable/ic_call_answer_video.xml
      736  1981-01-01 01:01   res/drawable/ic_call_answer_video_low.xml
     1528  1981-01-01 01:01   res/drawable/ic_call_decline.xml
     1492  1981-01-01 01:01   res/drawable/ic_call_decline_low.xml
     5696  1981-01-01 01:01   res/drawable/ic_launcher_background.xml
     1572  1981-01-01 01:01   res/drawable/ic_launcher_foreground.xml
     1180  1981-01-01 01:01   res/drawable-
v21/notification_action_background.xml
      532  1981-01-01 01:01   res/drawable/notification_bg.xml
      532  1981-01-01 01:01   res/drawable/notification_bg_low.xml
      372  1981-01-01 01:01
res/drawable/notification_icon_background.xml
      304  1981-01-01 01:01   res/drawable/notification_tile_bg.xml
      215  1981-01-01 01:01   res/drawable-mdpi-
v4/notification_bg_low_normal.9.png
```

Figure 52 – Paste of Source Code to ChatGPT

We allowed gpt to recover the material of apk file.

Step 4: Explanation from ChatGPT

Nice work extracting the contents of the APK 👍

From what you've done so far:

- ✅ You listed all files with `unzip -l` and `jar tf`.
- ✅ You ran `strings` on the APK and searched for `ictf`, `{`, `}`, `flag`, etc. You found a lot of junky `{` symbols but nothing obvious like `ictf{...}` yet.
- ✅ The APK has **multiple DEX files** (`classes.dex`, `classes2.dex`, `classes3.dex`, `classes4.dex`, `classes5.dex`). That's common when an app is obfuscated or uses multi-dex.

Next Steps (Reverse with JADX)

1. **Decompile APK with JADX:**

```bash
jadx-gui app-debug.apk
```

   - Open `app-debug.apk` in the GUI.
   - On the left sidebar, expand the `sources` tree.
   - Look at the package names — usually you'll see something like `com.example.weird` or similar.
2. **Search inside JADX:**
   - Use **Ctrl+F** in JADX and search for:
     - `ictf`
     - `flag`
     - `secret`
     - `"{"` (sometimes flags are stored as string concatenations).
3. **Check MainActivity & Utilities:**
   - Look at `MainActivity.java`. Many CTF APKs either:
     - Hardcode the flag as a string, or
     - Store it in a function that does some simple decoding (base64, XOR, stringbuilder).
   - If not in `MainActivity`, check helper classes like `Utils`, `FlagUtils`, `Decryptor`, etc.

Figure 53 – Sample Output from ChatGPT

We read and perceive output which gpt returned to us.

4.2 **Github (For Reference Only)**

Author Write-Up Link:
https://github.com/ImaginaryCTF/ImaginaryCTF-2025-Challenges/tree/main/Reversing/weird-app

We also get inspired by author write-ups:

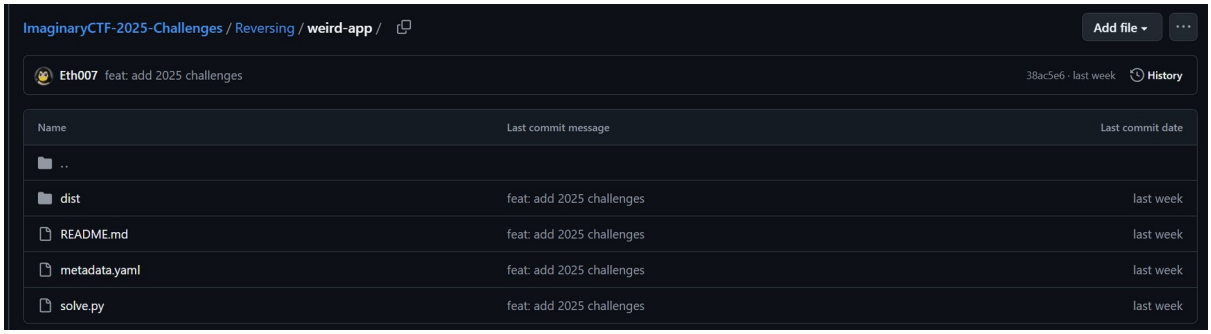


Figure 54 – GitHub Solutions (Weird-app)

README.md

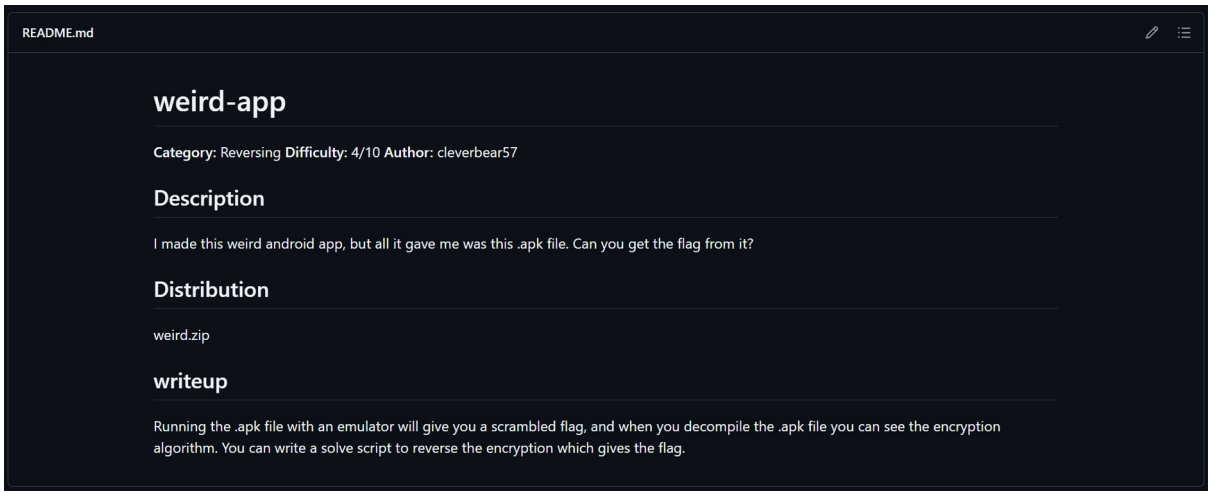


Figure 55 – GitHub README.md (Weird-app)

metadata.yaml

Figure 56 – GitHub metadata.yaml (Weird-app)

Here's showed the flag after the CTF challenge ended.

solve.py

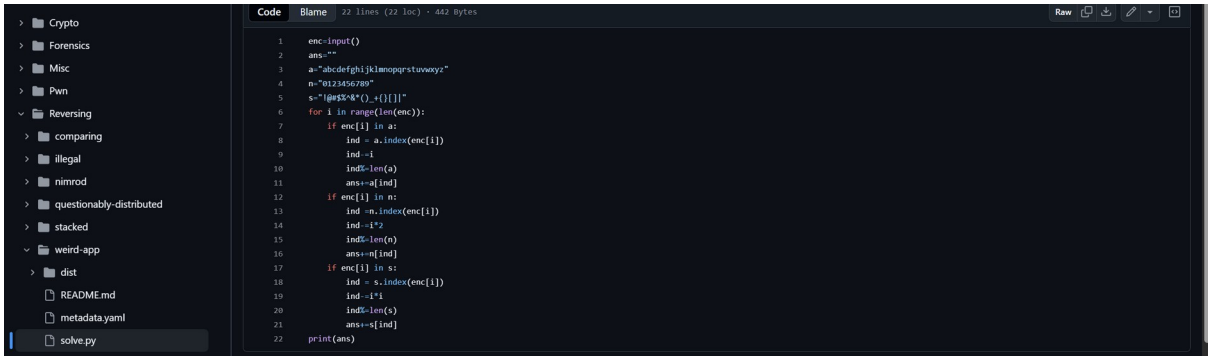


Figure 57 – GitHub solve.py (Weird-app)

The python code we can use to solve the CTF challenge (weird-app).

4.3 **Jadx**

Figure 58 – GitHub solve.py (Weird-app)

Our primary tool in this CTF challenge was Jadx, which is an open-source APK decompiler and a GUI to investigate Android packages. Jadx allows us to call Dalvik bytecode to readable Java/ Kotlin-style source, browse the resource files and manifests, and search through thousands of decompiled files to find suspects of suspicious functions or hard-coded strings - just what we wanted in an APK-style puzzle, like Weird-App.

Why Jadx?

1. **Direct Interpretability into app logic:**
   Jadx deconstructs methods, classes and string constants to enable us to locate transformation routines (e.g. transformFlag("REDACTED")) and trace the construction of the flag.

2. **Intelligent code navigation & search:**
   The GUI includes a tree view and global search, and it is quick to navigate to related files, trace call chains and locate the specific code paths that are important.

3. **Unreadable output:**
   The code that has been decompiled is readable and can be reimplemented in Python or Java to test - again vital when we required to replicate and reverse obfuscation schemes.

4. **Cross-platform and lightweight:**

Jadx is Windows/Linux/macOS, and it is easy to execute on a Kali workstation or a personal laptop, thus our workflow remained the same between team members.

5. **Interactions with dynamic testing:**
   After identifying candidate functions by Jadx, we verified the behaviour of them by emulator executions or small test harnesses to verify our decoders.

6. **Export & evidencing:**
   Jadx will give us the option to print snippets and capture screenshots of the source, which are convenient to document and write up.

In general, Jadx was the quickest and most viable to statically analyse the APK and reduce the flag logic.

4.4 Python

Figure 59 – GitHub solve.py (Weird-app)

```
Type "help", "copyright", "credits" or "license" for more information.
>>> # inverse_transform.py
... alphabet = "abcdefghijklmnopqrstuvwxyz"
... digits   = "0123456789"
... symbols  = "!@#$%^&*()_+{}[]|"
...
... def inverse_transform(transformed):
...     orig_chars = []
...     for i, ch in enumerate(transformed):
...         found = None
...         # if transformed char is in alphabet → original was a lowercase letter
...         if ch in alphabet:
...             t_idx = alphabet.index(ch)
...             # orig_index = (t_idx - i) mod 26
...             orig_idx = (t_idx - i) % len(alphabet)
...             found = alphabet[orig_idx]
...         # if transformed char is in digits → original was a digit
...         elif ch in digits:
...             t_idx = digits.index(ch)
...             orig_idx = (t_idx - (i * 2)) % len(digits)
...             found = digits[orig_idx]
...         # if transformed char is in symbols → original was symbol
...         elif ch in symbols:
...             t_idx = symbols.index(ch)
...             orig_idx = (t_idx - (i * i)) % len(symbols)
...             found = symbols[orig_idx]
...         else:
...             found = '?'  # unknown/unsupported char
...         orig_chars.append(found)
...     return "".join(orig_chars)
...
... def forward_transform(orig):
...     res = []
...     for i, ch in enumerate(orig):
...         if ch in alphabet:
...             c = alphabet.index(ch)
...             ind = c + i
...             res.append(alphabet[ind % len(alphabet)])
...         elif ch in digits:
...             c = digits.index(ch)
...             ind = (i * 2) + c
...             res.append(digits[ind % len(digits)])
...         elif ch in symbols:
...             c = symbols.index(ch)
...             ind = (i * i) + c
...             res.append(symbols[ind % len(symbols)])
...         else:
...             res.append('?')
...     return "".join(res)
...
... if __name__ == "__main__":
...     # Example transformed string (from your earlier post)
...     transformed = "idvi+1{s6e3{)arg2zv[moqa905+"
...     original = inverse_transform(transformed)
...     print("Transformed:", transformed)
```

Figure 60 – Sample of python source code used

Inverse transform takes the transformed string and, one after another, finds out what character transformed character is in and calculates the original character by modular subtraction. Forward transform rewrites the transform used by the app, allowing you to check the original that was recovered.

5. Vulnerability

Weaknesses:

1. The application leaked confidential information through the deterministic, position-based obfuscation (transformFlag), which, however, can be reversed using only simple modular arithmetic - security by obscurity, not security.

2. The absence of integrity/anti-tamper checks and hard-coded secrets ensured that the static analysis was a trivial matter. The hosting (web editor in a Docker case) also was leaking helpful test vectors.

How we exploited it:

Find transform flag and transformed string with reverse-engineering using Jadx (without a runtime exploit) and invert the position-based mapping with a small Python program.

Security implications:

Storing flags / secrets in distributed binaries should never be stored without cryptographic protection or validation on the server-side, trivial transformations can easily be reversed by any motivated analyst.

Skills gained:

Jadx code inspection, reverse-engineering logic (modular changes), writing deconstructing scripts in Python, and orchestrated group working (roles + verification).

Mitigations:

Secrets should be kept on the server-side, proper cryptography and authenticated requests, and detective measures should be implemented in case client-side validation is needed.

5

6. Challenges Faced

Although the Weird-App challenge looked simple at first, we encountered several obstacles while analyzing and solving it. These challenges tested both our technical skills and our ability to coordinate under time pressure.

1. Overwhelming Decompiled Code (Navigation Problem):
When first opened the APK in JADX, we face thousands of lines of decompiled code, which was overwhelming as beginners. Many functions appeared meaningless, and we wasted time exploring irrelevant sections before realizing we should focus only on suspicious logic like transformFlag.

Solution:
We split up roles immediately--one member scanning for suspicious functions, another reviewing constants, a third testing ideas in Python, and the last documenting. This division helped us focus on useful sections and avoid dead ends.

2. Communication Under Time Pressure:
The live, timed format created pressure that sometimes-caused duplication of work, such as two members analysing the same function unknowingly. Poor communication during rushed moments slowed progress.

Solution:
We set a strict rule of a quick sync every 15 minutes. Each member reported their progress, which reduced overlap and improved efficiency. This taught us how essential structured communication is in a live CTF.

3. Probability Change of transformFlag Logic:
The rotation of transformFlag function was initially confusing as index based. Decoding our first attempts failed and we did not know whether the algorithm was incorrect or our implementation was incorrect. This demoralized in the middle of the time-timed game.

Solution:
Well, we did not give up but tried hypotheses systematically with small Python scripts by trial-and-error. This trial-and-error method ultimately came up with the right reverse transformation and we were able to retrieve the flag.
Automation scripting Writing a decoder in Python to decode fast and reliably.

6

6.1 Lessons Learned

Through these experiences, we have been able to learn that technical skill is just important as teamwork and communication. There are role division and periodic synchronization that save time during pressure. We also knew the importance of being patient and tenacious because trial and error is very likely to result in breakthroughs in the reverse engineering process. Lastly, we learned that such tools as JADX and Python are tools that very effective to navigate and decode obfuscated logic easily in APK challenges.

7. Explain Like I'm Five (ELIF)

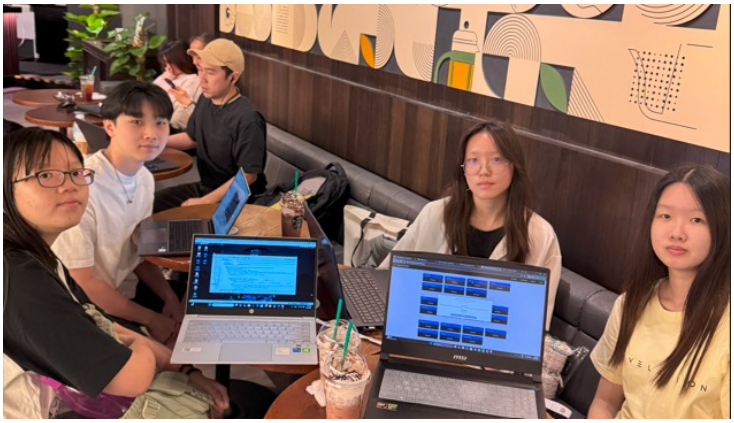
Imagine that you have accidentally discovered a weird treasure-chest (the APK) that has some secret code on it. However, there is no ordinary lock, all letters and numbers on the chest are mixed up in a magic trick, which varies according to the location. We were to determine how the chest scrambled its letters and then get the trick undone so that we could open it.

We have used a tool called JADX (a key) to open the APK (the treasure) and had a glimpse of what the chest contained and what was inside its instructions. We hunted around the numerous scrolls therein, and eventually found a spell named transform Flag (REDACTED). The process by which the letters were moved was elucidated by this spell: the letters moved on by their place, the numbers severed by their numbers, and the symbols sprang about in their forms.

We had initially guessed what the scrambled message was, but we were unsuccessful. Thus, a little Python code was created to reverse the spell, character by character. After we got there the jumbled letters were made out in a readable flag again, -- as when you open the chest with the right key.

The challenge also helped us learn the lesson of being patient, working as a team, and having the right tools. There were those of us who coded, those who trialled an idea and those who coded the decoding. At last, unitedly we lifted the spell, and opened the chest, and got the flag which made our victory.

8. Group Photo



9. Workload Matrix

No	Name	TP Number	Workload (%)
1	Chew Kai Xuan	TP076457	25
2	Chua Yi Hua	TP099596	25
3	Fhang Ying Ying	TP076307	25
4	Nicholas Beh	TP070075	25

10. References

1. GitHub repository (challenge solution)
ImaginaryCTF. (2025). ImaginaryCTF-2025-Challenges: Reversing/weird-app/solve.py [Computer software]. GitHub. https://github.com/ImaginaryCTF/ImaginaryCTF-2025-Challenges/blob/main/Reversing/weird-app/solve.py

2. CTF challenges list page
ImaginaryCTF. (2025). Challenges. https://2025.imaginaryctf.org/Challenges.html

3. ChatGPT website (for reference / explanation)
OpenAI. (2025). ChatGPT. https://chatgpt.com

4. YouTube video
YouTube. (2025, Month Day). [Video title] [Video]. YouTube. https://www.youtube.com/watch?v=wMMKqj9Srqg

5. Apriorit blog — reverse engineering tools
Apriorit. (n.d.). Software reverse engineering tools. Apriorit Dev Blog. https://www.apriorit.com/dev-blog/366-software-reverse-engineering-tools

6. W3Schools — Python tutorial
W3Schools. (n.d.). Python tutorial. W3Schools. https://www.w3schools.com/python/

7. YouTube video (ID: OWEZQMVLMPs)
YouTube. (n.d.). YouTube video [Video]. https://www.youtube.com/watch?v=OWEZQMVLMPs

8. YouTube video (ID: 5vG6lbBUT9U)
YouTube. (n.d.). YouTube video [Video]. https://www.youtube.com/watch?v=5vG6lbBUT9U

9. HTTP Toolkit — Android reverse engineering blog post
HTTP Toolkit. (n.d.). Android reverse engineering: an intro and practical guide. HTTP Toolkit Blog. https://httptoolkit.com/blog/android-reverse-engineering/

10. Gorkem Kara — jadx tutorial
Kara, G. (n.d.). Reverse engineering Android apps — JADX GUI tutorial. GorkemKara.net. https://gorkemkara.net/reverse-engineering-android-apps-jadx-gui/