OmniSaga

Shonen Walkthrough (Hardcoded Secrets)

Step 1: Decompiling the APK

- Use tools like `jadx-gui`, `apktool`, or `MobSF` to analyze the application.
- Extract and inspect the `AndroidManifest.xml` file for any suspicious permissions or exported activities.
- Open the `classes.dex` file using `jadx` to review the Java source code.

Hint: The application's core logic resides in `MainActivity2.java` and `SecretManager.java`.

Step 2: Identifying the Secret Retrieval Mechanism

- `MainActivity2` retrieves a hidden flag using the method `getHiddenFlag()`.
- This function attempts to invoke `getSecret()` from `SecretManager` using Java Reflection.
- `SecretManager` loads a native library (`getme.so`) and calls `getEncodedSecret()`.

Hint: The actual flag is not stored directly in Java but in the C++ library. You'll need to analyze the native library to extract it.

Step 3: Extracting the Encoded Secret from Native Code

- Use `strings` and `objdump` on the `libgetme.so` file to inspect its contents:

```
strings libgetme.so | grep -i "encoded"
objdump -D libgetme.so | less
```

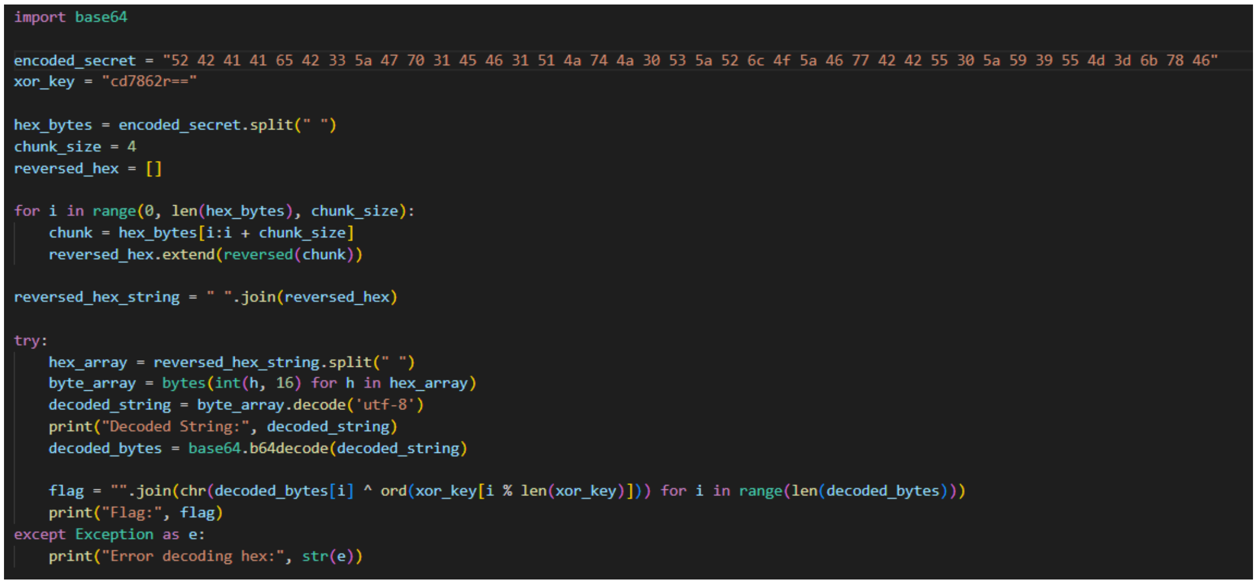
- You should find a hardcoded encoded secret in hexadecimal format.

Hint: The secret string is encoded in a reversible way. Understanding how `xorDecrypt()` works is key to decoding it.

Step 4: Decoding the Secret

- The encoded string is structured in a hex format that needs reversing.
- Convert the hex string into readable bytes.

- Decode the resulting Base64 string.
- Perform an XOR decryption using the key `cd7862r==`



```python
import base64

encoded_secret = "52 42 41 41 65 42 33 5a 47 70 31 45 46 31 51 4a 74 4a 30 53 5a 52 6c 4f 5a 46 77 42 42 55 30 5a 59 39 55 4d 3d 6b 78 46"
xor_key = "cd7862r=="

hex_bytes = encoded_secret.split(" ")
chunk_size = 4
reversed_hex = []

for i in range(0, len(hex_bytes), chunk_size):
    chunk = hex_bytes[i:i + chunk_size]
    reversed_hex.extend(reversed(chunk))

reversed_hex_string = " ".join(reversed_hex)

try:
    hex_array = reversed_hex_string.split(" ")
    byte_array = bytes(int(h, 16) for h in hex_array)
    decoded_string = byte_array.decode('utf-8')
    print("Decoded String:", decoded_string)
    decoded_bytes = base64.b64decode(decoded_string)

    flag = "".join(chr(decoded_bytes[i] ^ ord(xor_key[i % len(xor_key)])) for i in range(len(decoded_bytes)))
    print("Flag:", flag)
except Exception as e:
    print("Error decoding hex:", str(e))
```

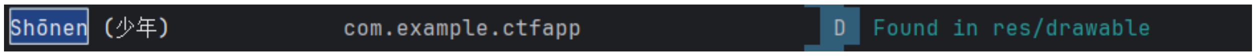
Hint: If your result looks like gibberish, ensure you are reversing the hex string correctly.

Step 5: Verifying the Flag

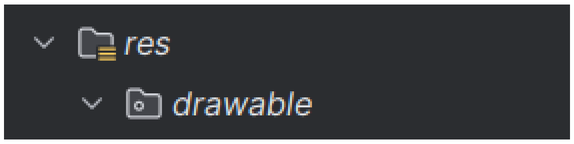
- Once you obtain the correct flag, enter it into the application's UI.

Hint: The flag format typically follows `cdf_Flag{...}`.

- If correct, the hidden image button should appear, confirming the solution.
- Inspect logs and hardcoded hints in UI elements.



```
Shōnen (少年)          com.example.ctfapp          D  Found in res/drawable
```



Isekai Walkthrough (Vulnerable Android IPC Components)

Step 1: Decompile the APK

- Use `jadx-gui` or `apktool` to extract the source code.
- Inspect `AndroidManifest.xml` for exported services and suspicious permissions.
- Identify inter-process communication (IPC) components like bound services, broadcast receivers, or content providers.

Step 2: Analyze the Vulnerable IPC Service

- Locate `MyService.java`, which uses `FileObserver` to monitor `flag.txt`.
- Check if it allows external interaction, such as reading/writing sensitive files.
- Investigate `MyObserver.validateFlag()` for insecure flag retrieval mechanisms.

Step 3: Exploit the IPC Vulnerability

Approach 1: Intent Manipulation

- If `MyService` is exported, interact with it via `adb`:

  ```
  adb shell am startservice -n com.example.ctfapp/.MyService
  ```
- Check log outputs for sensitive information leaks.

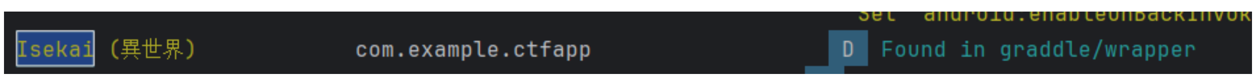
Approach 2: Extracting the Flag

- The flag is Base64-encoded inside `flag.txt`.
- Locate `MyObserver.storeFlag()`, which encodes and writes the flag.
- Extract and decode the flag

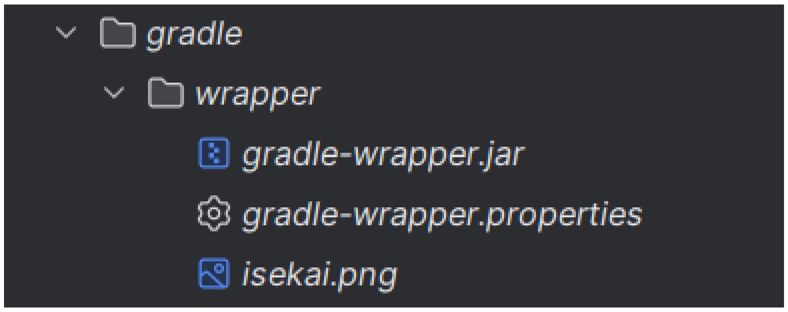
```python
import base64

encoded_text = "U0VDUkVUX0ZMQUdfVUlCQVNFQ1JFVA=="
decoded_text = base64.b64decode(encoded_text).decode("utf-8")

print("Decoded Text:", decoded_text)
```

Step 4: Reverse Engineer UI Clues

- `VulnerableServiceActivity` has an `ImageView` revealed upon correct flag entry.
- Inspect logs and hardcoded hints in UI elements.



```
Isekai (異世界)          com.example.ctfapp          D  Found in graddle/wrapper
```



```
gradle
  wrapper
    gradle-wrapper.jar
    gradle-wrapper.properties
    isekai.png
```

Seinen Walkthrough (Insecure Deeplinks)

Step 1: Reverse Engineering the APK

- Decompile the APK using tools like `jadx` or `apktool`:

  ```
  jadx -d output CTFApp.apk
  ```

- Locate `DeeplinkHandlerActivity.kt` and `SecurityUtils.kt` to understand the deep link processing.

Step 2: Extracting the Secret Key

- In `SecurityUtils.kt`, we see:

  ```
  val expectedHash = "5d41402abc4b2a76b9719d911017c592"
  ```

  - This is an MD5 hash of the secret key.
- Use an online MD5 decryption service or hash lookup tool to reveal the plaintext key:

  ```
  echo -n "hello" | md5sum
  ```

  - The hash corresponds to `hello`.

Step 3: Constructing the Exploit Deep Link

Now that we have the secret key (`hello`), we craft a deep link:

```
ctfapp://challenge?action=getFlag&key=hello
```

Open this deep link in an emulator or real device:

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

This should display the flag in the `flagTextView`.

Step 4: Retrieving the Flag from Storage (Alternative Approach)

If deep link exploitation is patched, we can retrieve the flag from the storage:

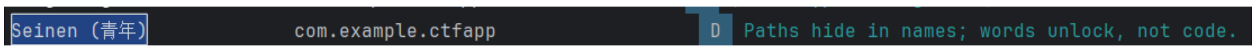
- The flag is split across four files (`.hiddenA`, `sysconfig.tmp`, `logcache.dat`, `cache_meta.bin`).
- Extract and decode them manually:
```
adb pull /data/data/com.example.ctfapp/files/.hiddenA
```
- Decode using Base64 and XOR decryption.

Step 5: Verifying the Flag

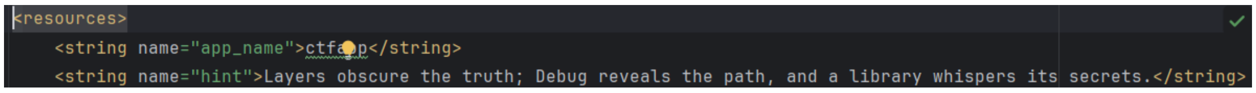
- Once you obtain the correct flag, enter it into the application's UI.

Hint: The flag format typically follows `CTF{...}`.

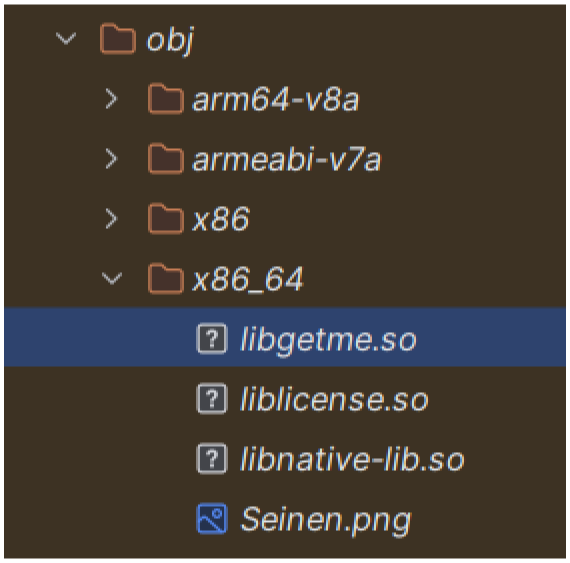
- If correct, the hidden image button should appear, confirming the solution.
- Inspect logs and hardcoded hints in UI elements.



```
Seinen (青年)          com.example.ctfapp               D  Paths hide in names; words unlock, not code.
```



```
<resources>
    <string name="app_name">ctfapp</string>
    <string name="hint">Layers obscure the truth; Debug reveals the path, and a library whispers its secrets.</string>
```



Slice of Life Walkthrough (Insecure)

Step 1. Static Analysis

Reviewing the Login Logic

- The login credentials are hardcoded in `InsecureActivity.kt`:

  ```
  if (username == "admin" && password == "password123")
  ```

- Using the above credentials (`admin` / `password123`), the attacker can log in and proceed to `DashboardActivity`.

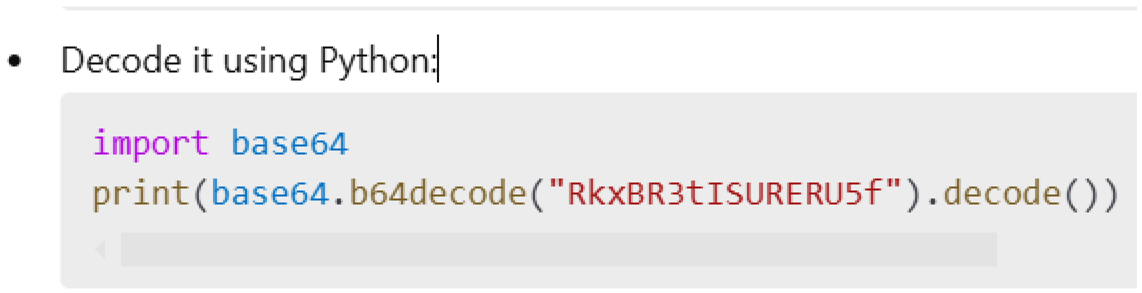
Step 2. Dynamic Analysis

Examining Flag Storage Locations

From the code, we see that the flag is split and stored in:

1. **SharedPreferences** (Base64 Encoded)
2. **SQLite Database**
3. **Internal Storage (XOR Encrypted File)**

Extracting Flag Part 1 (Stored in SharedPreferences)

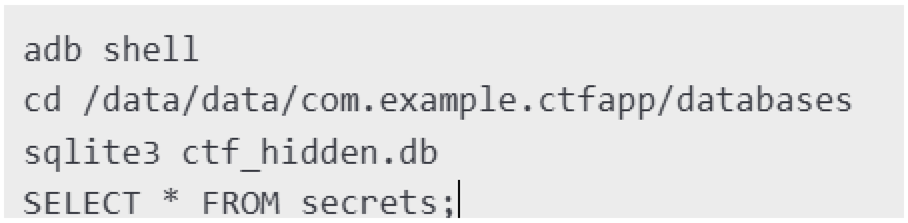


- Decode it using Python:

```
import base64
print(base64.b64decode("RkxBR3tISURERU5f").decode())
```

Extracting Flag Part 2 (Stored in SQLite Database)

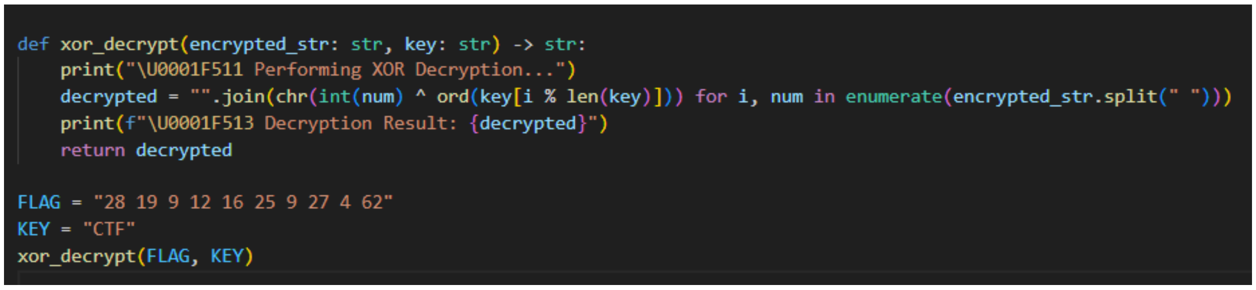
Dump the database from the Android device using ADB:



```
adb shell
cd /data/data/com.example.ctfapp/databases
sqlite3 ctf_hidden.db
SELECT * FROM secrets;
```

Extracting Flag Part 3 (XOR Encrypted File)

- The file `hidden_flag2.txt` contains an XOR-encrypted string.
- cThe file contents can be decrypted using the reverse XOR operation in Python:



```
def xor_decrypt(encrypted_str: str, key: str) -> str:
    print("\U0001F511 Performing XOR Decryption...")
    decrypted = "".join(chr(int(num) ^ ord(key[i % len(key)])) for i, num in enumerate(encrypted_str.split(" ")))
    print(f"\U0001F513 Decryption Result: {decrypted}")
    return decrypted

FLAG = "28 19 9 12 16 25 9 27 4 62"
KEY = "CTF"
xor_decrypt(FLAG, KEY)
```

Step 3. Reconstructing the Full Flag

By combining the extracted part

Step 4. Alternative Flag Extraction Methods

Memory Dumping

- If the app is running on a rooted device, use `frida` to dump memory:

```
frida -U -n com.example.ctfapp -i
```

- Search for the flag using `grep`:

```
grep -a "FLAG{" /proc/mem
```

Network Traffic Analysis

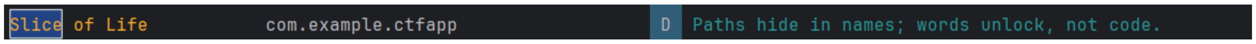
- If the app sends the flag over the network, use `mitmproxy` or `Burp Suite` to intercept traffic.

Step 5: Verifying the Flag

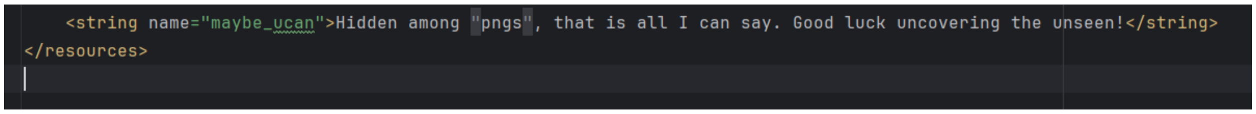
- Once you obtain the correct flag, enter it into the application's UI.

Hint: The flag format typically follows `FLAG{...}`.

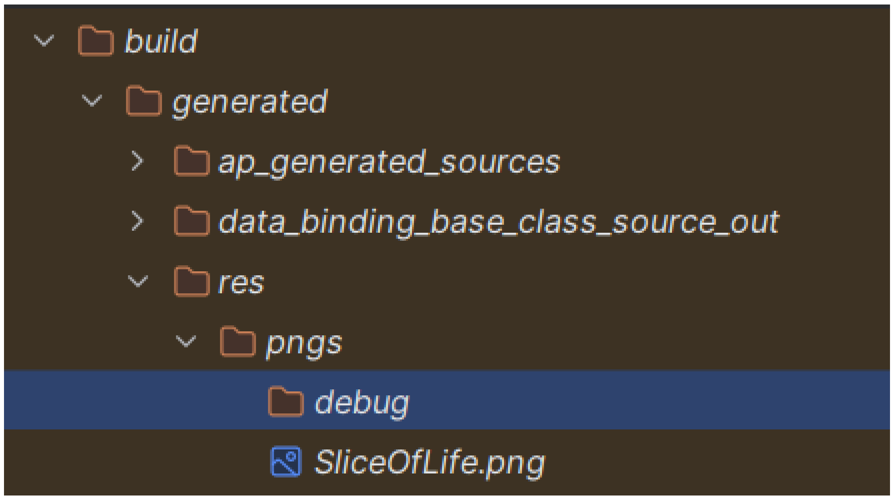
- If correct, the hidden image button should appear, confirming the solution.
- Inspect logs and hardcoded hints in UI elements.



```
Slice of Life          com.example.ctfapp          D  Paths hide in names; words unlock, not code.
```



```
    <string name="maybe_ucan">Hidden among "pngs", that is all I can say. Good luck uncovering the unseen!</string>
</resources>
```



Psych&Thrill Walkthrough (Ghost Queries)

Step 1: Examining the Login Logic

The `LoginActivity.kt` file contains the login logic. The key function in this class is `checkLogin()`, which interacts with the SQLite database (`UserDatabaseHelper`).

Vulnerability Identified:

- The SQL query used for authentication is **vulnerable to SQL Injection**:
  val query = "SELECT password FROM users WHERE username = '$username' AND password = '$password'"
  Since the input is directly concatenated into the query, an attacker can manipulate the input to bypass authentication.

Exploitation:

- A common SQL Injection payload such as:
  ```
  ' OR '1'='1' --
  ```
  can be used to bypass authentication, as it results in the query always returning true.

Step 2: Extracting the Flag via SQL Injection

Since the database contains user credentials, executing:

```
' OR '1'='1' --
```

Bypassing Authentication

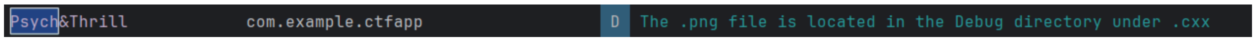
sys_user' OR '1'='1

as the username (with any password) should allow access.

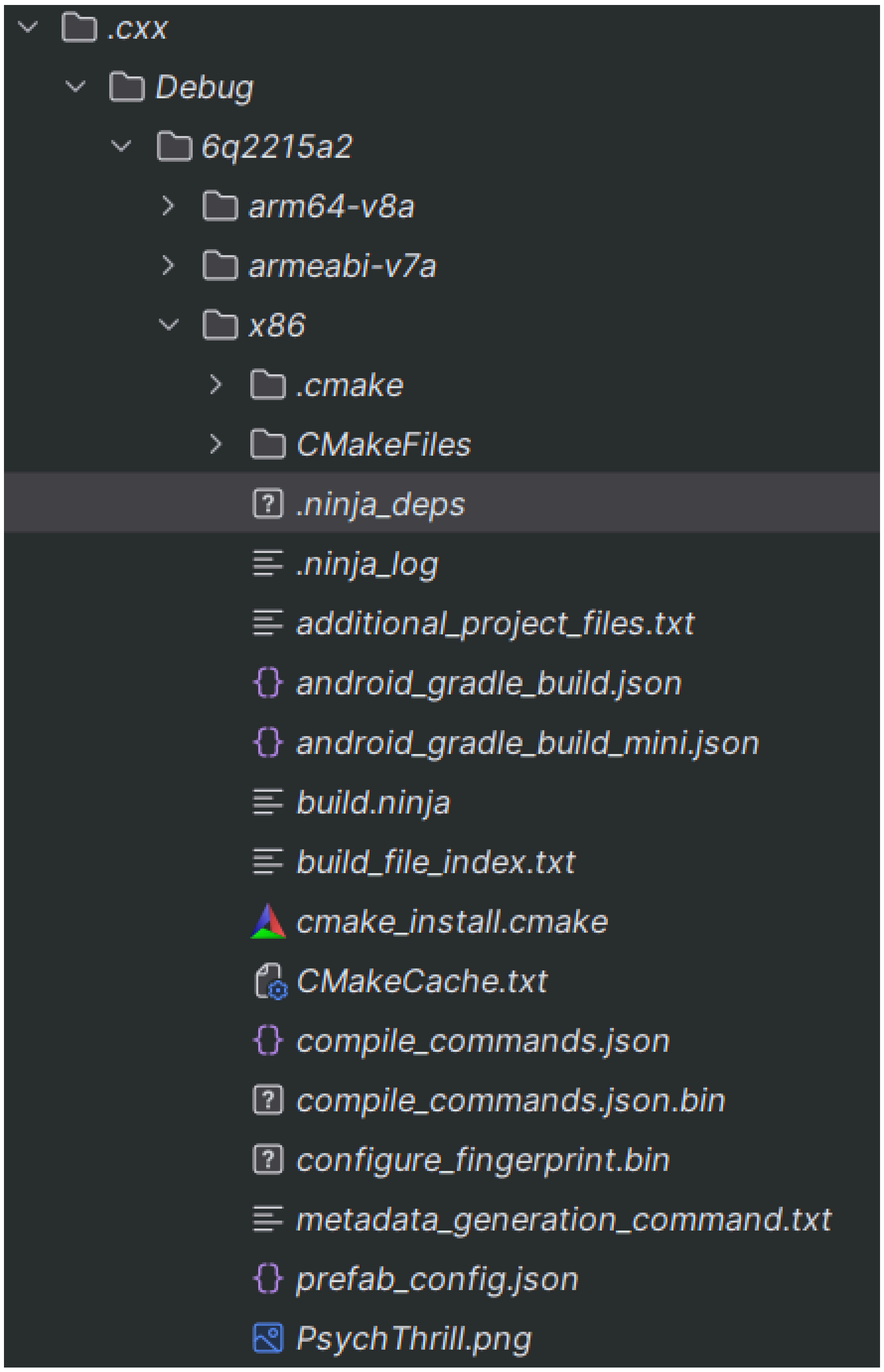
Once logged in, the flag is retrieved using AES decryption. However, the decryption key and salt are **hardcoded** in `AESUtils.kt`, making it susceptible to static analysis or reverse engineering.

If correct, the hidden image button should appear, confirming the solution.

Inspect logs and hardcoded hints in UI elements.



```
Psych&Thrill          com.example.ctfapp          D  The .png file is located in the Debug directory under .cxx
```



```
.cxx
  Debug
    6q2215a2
      arm64-v8a
      armeabi-v7a
      x86
        .cmake
        CMakeFiles
        .ninja_deps
        .ninja_log
        additional_project_files.txt
        android_gradle_build.json
        android_gradle_build_mini.json
        build.ninja
        build_file_index.txt
        cmake_install.cmake
        CMakeCache.txt
        compile_commands.json
        compile_commands.json.bin
        configure_fingerprint.bin
        metadata_generation_command.txt
        prefab_config.json
        PsychThrill.png
```

For the glory of humanity! Walkthrough (Steganography)

Step 1: Gathering Data from Previous Challenges

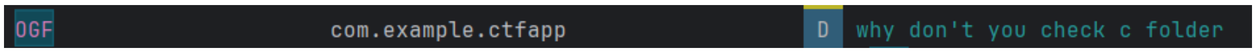
- The player must collect images from the previous five challenges.
- Using the steganography tool, extract hidden data from each image.
- Rearrange the extracted text from all five images to form the final flag.
- The final reconstructed flag is then used to unlock the main hidden image.

Step 2: Verifying the Flag

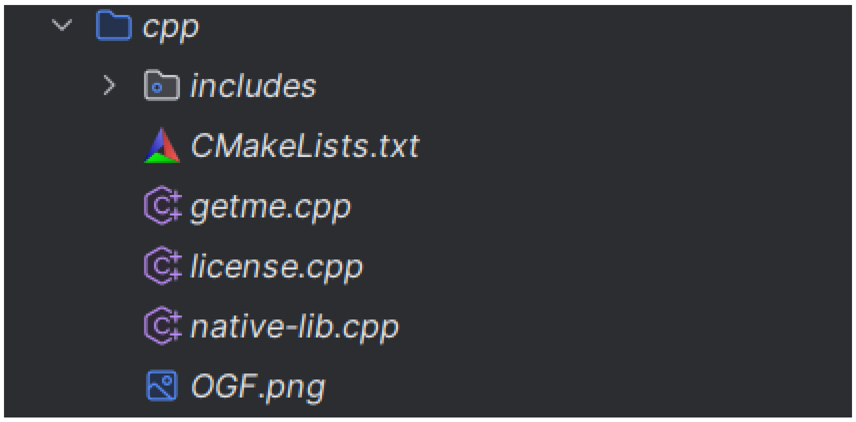
- Once you obtain the correct flag, enter it into the application's UI.

Hint: The flag format typically follows `ShingekiNoKyojin{...}`.

- If correct, the hidden image button should appear, confirming the solution.
- Inspect logs and hardcoded hints in UI elements.



```
OGF          com.example.ctfapp          D  why don't you check c folder
```



```
cpp
  includes
  CMakeLists.txt
  getme.cpp
  license.cpp
  native-lib.cpp
  OGF.png
```

Fight. Fight. Walkthrough (Small i Functions)

Step 1: Navigating to FightActivity

- **Button Redirection Issue in `WelcomeActivity`**
    - The "Fight Fight" button was initially set to navigate to `AmigoActivity`, which was incorrect.
    - This was fixed by changing the intent to navigate to `FightActivity` instead.
- **Incorrect Function Call in `FightActivity`**
    - The button responsible for checking the flag previously called `checkFlag()`, which was not the correct function.
    - This was fixed by modifying the button's `setOnClickListener` to call `checkProcess()` instead.

Step 2: Flag Verification Process

1. Select an encoded image using the provided button and upload the main image from the Previous flow
2. The application extracts the hidden message from the image.
3. Enter the extracted flag into the input field.
4. Click the "Way to the End!!" button to verify the flag.
5. If the entered flag matches the extracted message, the user is taken to the next challenge screen.

Frida Challenge-1 Walkthrough (Bypassing Security)

Step 1: Static Analysis of the APK

1. Decompile the APK using tools like JADX or APKTool
2. Analyze `NextActivity` and `RetrieveDB` classes to locate flag generation and security mechanisms.
3. Identify the encryption mechanism (`AES/ECB/PKCS5Padding`) and the hardcoded key.

Step 2: Dynamic Analysis Using Frida

- **Attach Frida to the Running App:**

```
frida -U -n com.example.ctfapp -s exploit.js --no-pause
```

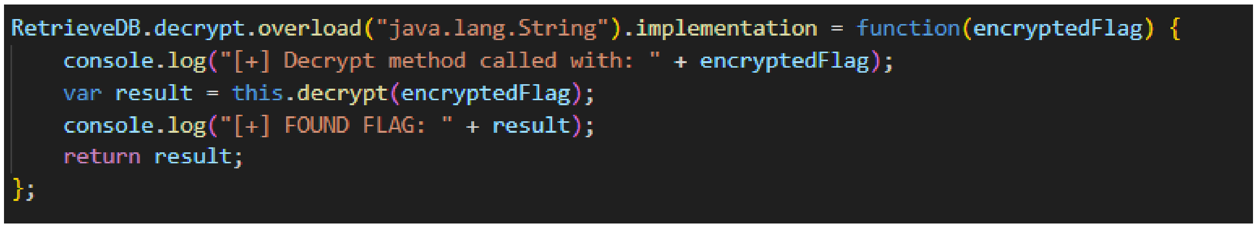
- **Script Execution:**
  - The script hooks into the app and starts capturing function calls.
  - The decrypted flag is printed to the Frida console.

Step 3: Bypassing Security Checks

1. **Disable Root Detection:**
   - Hook `isDeviceRooted()` to return `false`.
2. **Disable Debugger Detection:**
   - Hook `isDebuggerAttached()` to return `false`.
3. **Bypass Flag Extraction Check:**
   - Hook `isFlagExtracted()` to return `true`, enabling the proceed button

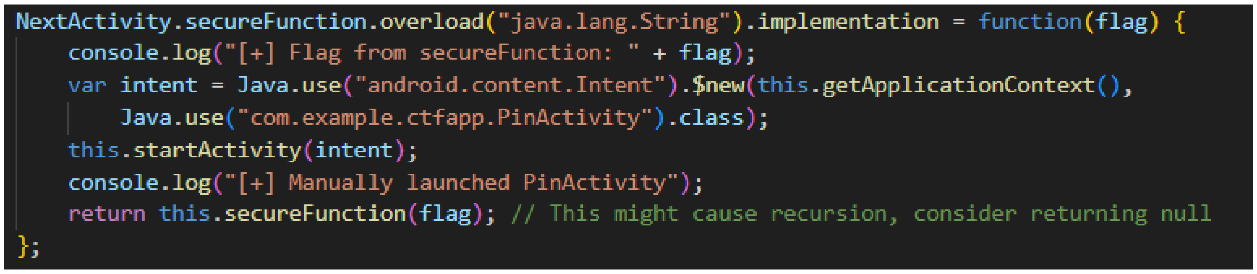
Step 4: Extracting and Using the Flag

1. **Intercept Decryption Function:**



```
RetrieveDB.decrypt.overload("java.lang.String").implementation = function(encryptedFlag) {
    console.log("[+] Decrypt method called with: " + encryptedFlag);
    var result = this.decrypt(encryptedFlag);
    console.log("[+] FOUND FLAG: " + result);
    return result;
};
```

2. **Trigger Navigation to the Next Activity:**



```
NextActivity.secureFunction.overload("java.lang.String").implementation = function(flag) {
    console.log("[+] Flag from secureFunction: " + flag);
    var intent = Java.use("android.content.Intent").$new(this.getApplicationContext(),
        Java.use("com.example.ctfapp.PinActivity").class);
    this.startActivity(intent);
    console.log("[+] Manually launched PinActivity");
    return this.secureFunction(flag); // This might cause recursion, consider returning null
};
```

- Hook `secureFunction()` to automatically proceed.

Frida Challenge-2 Walkthrough (Validate Pin)

Step 1: Static Analysis of the APK

By inspecting the decompiled source, we find the `PinActivity` class, which interacts with the `SecurityManager` JNI methods `validatePIN` and `isFridaDetected`.

Step 2: Analyzing the Native Code

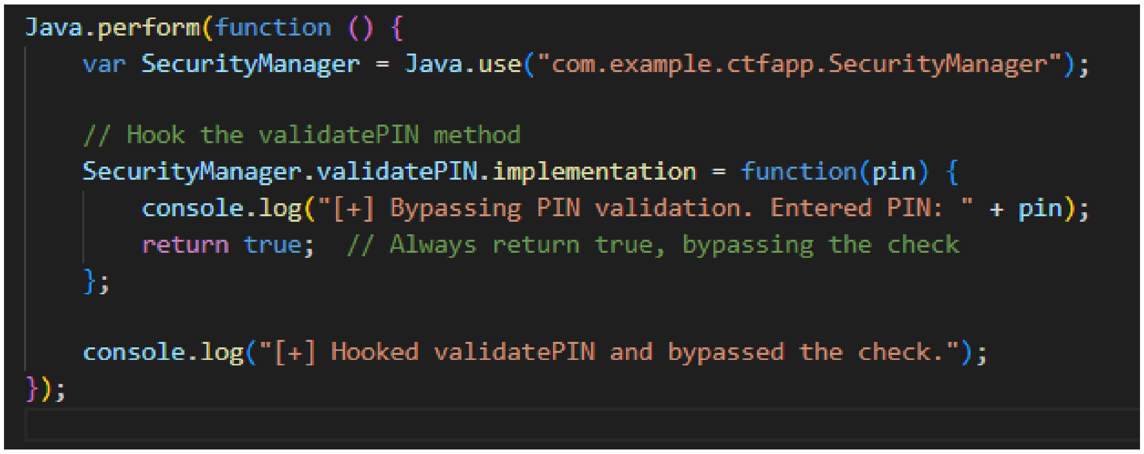
The native library, `native-lib`, is loaded at runtime and contains two key functions:

- `isFridaDetected()`: Checks for Frida-related libraries in memory using `dlopen()`.
- `validatePIN()`: Compares the entered PIN against a hardcoded correct PIN (`7357`).

Decompiling or examining the native library confirms that the correct PIN is hardcoded. This means an attacker can directly extract it and use it to obtain the flag.

Step 3: Bypassing PIN Validation with Frida

Instead of entering the correct PIN, an attacker can directly hook into the `validatePIN()` function and override its implementation using Frida:



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

Step 5: Extracting and Using the Hardcoded PIN

The `validatePIN()` function in the native library contains:

```
const char *correctPin = "7357";
```

Since this is hardcoded, the PIN `7357` can be directly used in the application to unlock the flag.

Frida Challenge-3 Walkthrough (License Verification)

Step 1: Understanding the Android Application

The provided Android application contains a `LicenseActivity` class that performs a license check using a native function:

```
external fun checkLicense(): Boolean
```

This function is implemented in C++ and is loaded from `liblicense.so`. When `checkLicense()` returns `true`, the application navigates to `CreditsActivity`. Otherwise, it displays "Invalid License".

Step 2: Analyzing the Native C++ Code

The native function `Java_com_example_ctfapp_LicenseActivity_checkLicense` is responsible for license validation. The key portions of the C++ code are:

```
std::string encryptLicense(const std::string& license) {
    std::string encrypted = license;
    int key = static_cast<int>(std::time(0)) % 256;  //

    for (size_t i = 0; i < encrypted.length(); i++) {
        encrypted[i] ^= key;  // XOR with key
    }

    return encrypted;
}
```

The `encryptLicense` function encrypts a given license key using an XOR operation with a dynamic key derived from the system time. The decryption function follows the same logic:

```cpp
std::string decryptLicense(const std::string& encryptedLicense)
    std::string decrypted = encryptedLicense;
    int key = static_cast<int>(std::time(0)) % 256;

    for (size_t i = 0; i < decrypted.length(); i++) {
        decrypted[i] ^= key;
    }

    return decrypted;
}
```

The decryption relies on the current system time, making it challenging to recover the original key unless the exact encryption time is known.

The license validation logic is implemented as:

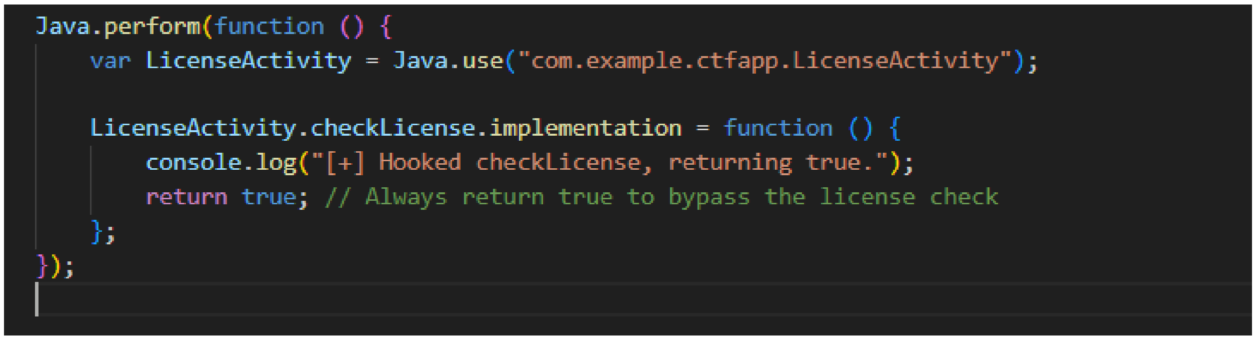
```cpp
std::string encryptedKey = "1C2C1F3F141F040604010B141B070F";
std::string decryptedKey = decryptLicense(encryptedKey);
std::string correctKey = "VALID-123-SECRET";

if (decryptedKey == correctKey) {
    return JNI_TRUE;
} else {
    return JNI_FALSE;
}
```

Since the correct license key is hardcoded (`VALID-123-SECRET`), the goal is to bypass this check.

Step 3: Using Frida to Bypass License Verification

Frida is a dynamic instrumentation toolkit that allows us to hook and modify the behavior of Android applications at runtime.

We can use the following Frida script to hook `checkLicense()` and force it to always return `true`:



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

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

Step 4: Verifying the Bypass

After running the Frida script, launch the application. Instead of displaying "Invalid License," the application should now navigate to `CreditsActivity`, confirming the successful bypass.

If the script is successful and not redirecting to `CreditsActivity,` then propagate back to Validate pin Activity and once again press “Submit” which will redirect to the `CreditsActivity`