**Walkthrough: Pirate’s Trove CTF**

**Team members: 21pc03 – Akhil SM, 21pc04 – Aparnaa T, 21pc05 – Ashwanth L**

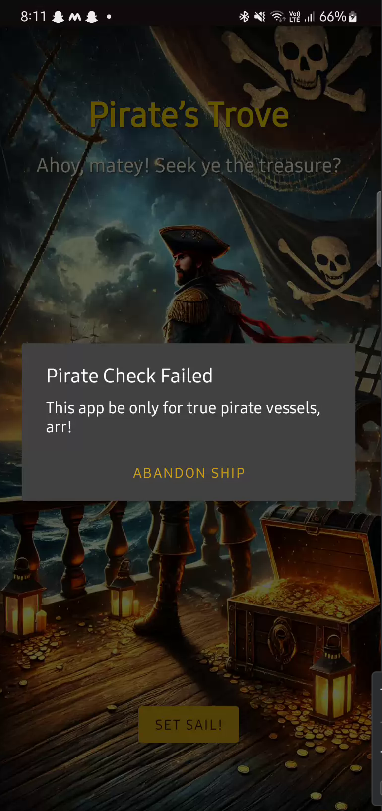
Pirate-themed Android app where you’re a treasure hunter seeking Captain Flint’s lost loot. The app has seven levels, each hiding a flag Except the first flag. Tools needed: ADB, APKTool, Frida, scripting knowledge and a bit of pirate cunning to find them all.

**Prerequisites**

* **Tools:** Install ADB (Android Debug Bridge), APKTool, Frida, and an emulator (e.g., Android Studio’s AVD) or a rooted device.
* **App:** Install PiratesTrove.apk on your emulator/device with USB debugging enabled.
* **Setup:** Connect your device/emulator to your computer via ADB (adb devices to confirm).

**Level 1 : Opening the application**

**Screen:** You will see a screen where the application will try to open , but an error message displays that the app is meant only for pirates and the user is presented with a screen that displays the message that is described above and a button titled abandon ship upon pressing it the app exits.



**Steps:**

**Decompile the APK:**

* Use APKTool and decompile the application
* Analyse the code and we can find that the system property value of “ship.deck” is compared with “PirateShip”
* So, open the adb shell with root access and set the property of ship.deck as PirateShip by using the command **setprop ship.deck PirateShip**
* Now restarting the application, the error gets resolved and we will be moving on.

**Alternate way:**

* Either we can decompile and change the smali code to bypass the checking condition and recompile it and sign it .

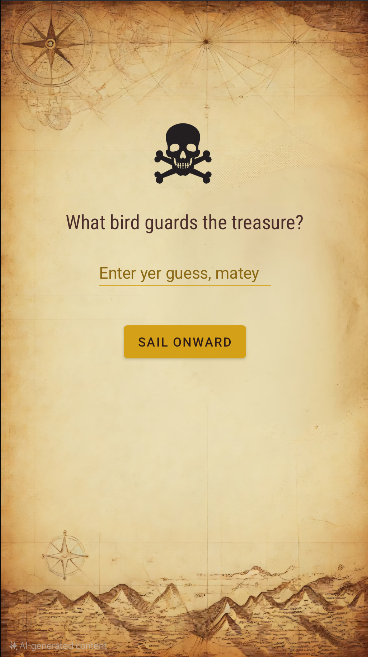
**Evaluation metrics:**

The knowledge of the solver to set the system properties is expected



**Level 2: The Parrot’s Riddle**

**Screen:** You see a skull icon, the text "What bird guards the treasure?" and an input field.  
**Hint:** The answer seems to be "parrot," but entering it does nothing. The real treasure’s hidden deeper.



**Steps:**

1. **Decompile the APK:**
   * Use APKTool: apktool d PiratesTrove.apk -o piratestrove\_decompiled
   * Navigate to piratestrove\_decompiled/res/values/strings.xml.
2. **Find the Flag:**
   * Look for <string name="secret\_clue">FLAG{ParrotSquawks}</string>.
   * The flag is in plain sight—no decryption needed!

**Flag 1:** FLAG{ParrotSquawks}

**Explanation:** The flag is hardcoded in the app’s resources, a classic insecure storage mistake. Decompiling reveals it instantly.

**Evaluation metrics:**

The knowledge of the solver to check and exploit the insecure resources storage mistake

**Next Move:** Click "Sail Onward" to proceed to Level 2.

**Level 3: Captain’s Loot**

**Screen:** A coin icon with "What’s the captain’s favorite loot?" and an input field.  
**Hint:** Guessing "gold" or "doubloons" doesn’t work. Maybe the captain’s secrets are stored elsewhere.



**Steps:**

1. **Access App Data:**
   * Use ADB to explore the app’s files: adb shell.
   * Navigate to /data/data/com.example.piratestrove/shared\_prefs/.
2. **Extract SharedPreferences:**
   * Run cat pirate\_prefs.xml or pull it: adb pull /data/data/com.example.piratestrove/shared\_prefs/pirate\_prefs.xml.
   * Open the file and find: <string name="captains\_secret">FLAG{GoldDoubloons}</string>.
3. **Note:** If the file isn’t readable without root, it’s because MODE\_WORLD\_READABLE was used (a vulnerability), but emulators often allow access.

**Flag 2:** FLAG{GoldDoubloons}

**Explanation:** The flag is stored in SharedPreferences with world-readable permissions, making it easy to extract with basic file access.

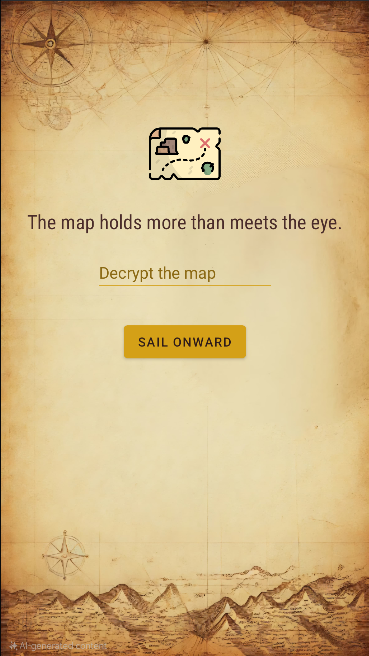
**Evaluation metrics:**

The knowledge of the solver to check the shared preferences is noted

**Next Move:** Hit "Sail Onward" for Level 3.

**Level 4: The Hidden Map**

**Screen:** A map icon says, "The map holds more than meets the eye," with a hint to "Decrypt the map."  
**Hint:** No obvious answer works. The map must be a file we can find and crack.



**Steps:**

1. **Pull the File:**
   * Use ADB: adb pull /data/data/com.example.piratestrove/files/treasure\_map.txt.
   * You’ll get a binary file with encrypted data.
2. **Analyze the Code:**
   * Decompile the APK again (apktool d) and check Level3Activity.smali or the Kotlin source if you have it.
   * Notice the XOR encryption with the key "pirate".
3. **Decrypt the File:**
   * Use a Python script:

Python

# Use raw string with r prefix  
with open(r"C:\Users\csuser\Downloads\piratestrove\piratestrove\treasure\_map.txt", 'rb') as f:  
    encrypted = f.read()  
  
key = "pirate"  
  
# Decrypt by XORing with the same key  
decrypted = "".join(chr(encrypted[i] ^ ord(key[i % len(key)])) for i in range(len(encrypted)))  
  
print(decrypted)  # Will print the decrypted flag

* + The XOR operation reverses the encryption, revealing the flag.

**Flag 3:** FLAG{SunkenChest}

**Explanation:** The flag is encrypted with a weak, predictable key ("pirate") and stored in a file. Basic reverse-engineering and decryption uncover it.

**Evaluation metrics:**

The knowledge of the solver to analyse the code and find the appropriate reverse engineering technique and writing a basic python script is expected

**Next Move:** Click "Sail Onward" to Level 4.

**Level 5: Flint’s Blade**

**Screen:** A sword icon asks, "Where did Flint hide his blade?"  
**Hint:** No input seems right. Maybe the app’s chattier than it looks.



**Steps:**

1. **Check Logs:**
   * Run the app and monitor logs: adb logcat | grep TREASURE.
   * Look for: D/TREASURE: FLAG{CutlassCove}.
2. **Alternative:** If you miss it, decompile and search Level4Activity.smali for Log.d("TREASURE", "FLAG{CutlassCove}").

**Flag 4:** FLAG{CutlassCove}

**Explanation:** The flag is carelessly logged to Logcat, a common debug oversight. Watching the logs spills the secret.

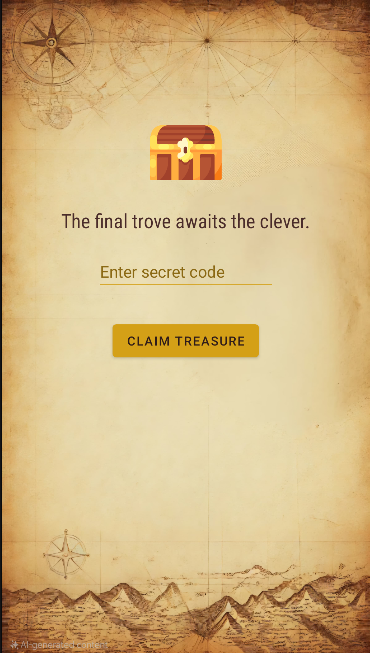
**Evaluation metrics:**

The knowledge of the solver to check the log and finding the flag is expected

**Next Move:** Press "Sail Onward" to Level 5.

**Level 6: The Final Trove**

**Screen:** A chest icon with "The final trove awaits the clever" and a "Claim Treasure" button.  
**Hint:** Entering a code does nothing special. The treasure’s in the journey, not the input.



**Steps (Using Frida):**

1. **Setup Frida:**
   * Install Frida (pip install frida-tools) and push frida-server to your device/emulator.
   * Run it: adb shell "/data/local/tmp/frida-server &".
2. **Hook the Intent:**
   * Create hook\_intent.js:

Javascript

Java.perform(function () {  
    // Hook into the Level5Activity class  
    var Level5ActivityClass = Java.use("com.example.piratestrove.Level5Activity");  
  
    // Function to try calling logSensitiveData  
    function callLogSensitiveData() {  
        Java.choose("com.example.piratestrove.Level5Activity", {  
            onMatch: function (instance) {  
                console.log("[\*] Found Level5Activity instance:", instance);  
  
                // Use reflection to access and call the private logSensitiveData method  
                // Specify no parameters with null  
                var logSensitiveDataMethod = Level5ActivityClass.class.getDeclaredMethod("logSensitiveData", null);  
                logSensitiveDataMethod.setAccessible(true); // Make private method accessible  
                logSensitiveDataMethod.invoke(instance, null); // No arguments for invoke  
                console.log("[\*] logSensitiveData() called successfully!");  
            },  
            onComplete: function () {  
                console.log("[\*] Search for Level5Activity instances completed.");  
            }  
        });  
    }  
  
    // Check periodically until the activity is found  
    console.log("[\*] Waiting for Level5Activity to be instantiated...");  
    var interval = setInterval(function () {  
        Java.choose("com.example.piratestrove.Level5Activity", {  
            onMatch: function (instance) {  
                console.log("[\*] Level5Activity detected, proceeding to call logSensitiveData...");  
                callLogSensitiveData();  
                clearInterval(interval); // Stop checking once found  
            },  
            onComplete: function () {}  
        });  
    }, 1000); // Check every second  
});

* + Inject it: frida -U -f com.example.piratestrove -l hook\_intent.js
  + There is a function that logs the obfuscated flag , which is never called by the application logic
  + This js script will call the function , which remains uncalled by the application logic

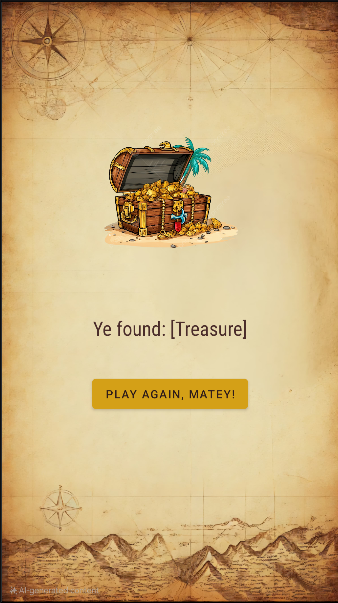
1. **Trigger the Intent:**
   * Navigate to Level 5 by completing all the challenges again
   * Now the function call is made and the obfuscated log is found in the logs, along with the key
   * Then it is done base64 decoding to extract the original flag

**Flag 5:** FLAG{FlintsFortune}

**Explanation:** The flag is passed via an Intent extra to an exported activity. Frida hooks the runtime data, or ADB exploits the exposed component.

**Evaluation metrics:**

The knowledge of the solver to analyse the code and use the Frida tool appropriately by writing the correct and appropriate js script



**All Flags**

1. **Flag 1:** FLAG{ParrotSquawks} (Strings resource)
2. **Flag 2:** FLAG{GoldDoubloons} (SharedPreferences)
3. **Flag 3:** FLAG{SunkenChest} (Encrypted file)
4. **Flag 4:** FLAG{CutlassCove} (Logcat)
5. **Flag 5:** FLAG{FlintsFortune} (Intent extra)

**Final Thoughts**

You’ve plundered all five flags! Each level exploits a different Android vulnerability:

* System property that could be changed by adb shell with root access
* Insecure resource storage (Level 1).
* World-readable preferences (Level 2).
* Weak encryption (Level 3).
* Debug log leakage (Level 4).
* Function with sensitive data (That could be extracted by Frida tool)(Level 5).