MOBILE SECURITY PACKAGE WALKTHROUGH

21PC09 – FENI AUXILY A

**Vuln CTF App**

**Introduction**:

This project is a Vulnerable Android CTF Application built with Kotlin/Java in Android Studio targeting API level 21, making it compatible with 99.7% of Android devices. The app includes multiple challenges, each demonstrating a common Android security flaw. Users can explore various vulnerabilities, and each vulnerability has a hidden flag retrievable through tools in the format FLAG{\*content\*}. The goal of this project is to help security learners practice reverse engineering, static and dynamic analysis, and mobile penetration testing realistically.

**Goal**:

The goal is to extract 6 flags using vulnerabilities in the vulnctfapp application.

**Preparation**:

1. Install APK on an emulator
2. Extract the APK to inspect contents
3. Decompile the APK for code inspection using jadx (Optional)

**Challenge 01**:

This challenge uses the Vulnerability, **Static Analysis Vulnerability**. The Static Analysis Vulnerability can easily expose hardcoded secrets like sensitive data, flags/API keys. Attackers can decompile and inspect the code to retrieve secrets. In Android applications, these values often end up in compiled .dex files, which can be easily reverse-engineered using tools like JADX or apktool.

Here, the application contains a flag hardcoded in a Java class file as a public static string. This is a classic vulnerability. When the APK is decompiled using a tool like JADX, the hardcoded string is visible in HardcodedFlag.java under the com.example.vulnctfapp.flags package, we can easily find it by searching for the term “FLAG”. This exposes the value of the flag without any runtime effort, showing why hardcoding secrets is dangerous.

**FLAG**: FLAG{h@rdc0ded\_&ecret}

**Challenge 02**:

This challenge demonstrates an **Insecure Storage Vulnerability**. Sensitive data like flags are stored in Android’s SharedPreferences, which can be accessed without any encryption or decryption if the device is rooted or emulated. This reflects poor handling of private information on the device.

Here, the flag is written into the app’s SharedPreferences using a helper class. We can easily retrieve it using Android’s Device File Explorer in Android Studio or using adb shell to access /data/data/com.example.vulnctfapp/shared\_prefs/ctf.xml. The flag is stored in plain text and can be directly read from the XML.

**FLAG**: FLAG{!n$ecure\_&h@red\_pref$}

**Challenge 03**:

This challenge leverages **Information Disclosure via logs** output. Logging sensitive information can allow anyone to retrieve it without needing access to the app's logic. We can also access these logs using adb logcat or other tools, especially during app runtime or debugging. Logging flags or internal logic to Logcat can unintentionally expose secrets, making this an information disclosure vulnerability.

Here, the flag is printed using the Log.d() function in a helper class. By running the app and monitoring Logcat output using Android Studio or adb logcat, we can search for the tag “FLAG” and find the printed value.

**FLAG**: FLAG{l0gc@t\_le@k}

**Challenge 04**:

This challenge features a **Logic Vulnerability**. The app requires and checks for a weak PIN so that it can reveal a flag upon successful validation. Implementing weak or easily reversible authentication logic can allow anyone to bypass checks by reverse-engineering the condition and directly supplying the expected input. Such predictable or hardcoded secrets/PINs are a serious security issue.

Here, we can either easily guess common PINs or decompile the app using JADX and inspect the logic inside PinValidator.java. It returns true only if the PIN equals "1234". Entering this into the app triggers the flag output on the screen.

**FLAG**: FLAG{@\_we@k\_p!n}

**Challenge 05**:

The vulnerability is the **Client-side Exposure via Assets**. This vulnerability arises from exposing sensitive content in client-side resources such as HTML files stored in the assets folder. Storing sensitive information in local WebView assets can expose data, as we can inspect or extract these files from the APK package without any authentication.

Here, the application loads a local HTML file into a WebView from the assets/ directory. Opening the APK with a file archiver or inspecting it using jadx or Android Studio's assets browser reveals vuln.html, which contains a commented flag in the source. Viewing the source of the page shows the flag clearly and openly without requiring interaction.

**FLAG**: FLAG{webv!ew\_vuln\_fl@g}

**Challenge 06**:

This challenge demonstrates a **Binary-level Vulnerability**. Sensitive data can be stored in native code and exposed via JNI functions. We can easily reverse engineer the compiled .so libraries using tools like Ghidra, IDA Pro, or simply use static analysis. Returning hardcoded secrets from native libraries compiled in C/C++ does not entirely ensure security.

Here, a native method returns a hardcoded flag from native-lib.cpp. When the app is decompiled, the call to getNativeFlag() is visible, but the flag is only found by disassembling libnative-lib.so. Only a string analysis or function tracing reveals the flag.

**FLAG**: FLAG{n@t!ve\_s0\_fl@g}

**Conclusion**:

The project reinforces how easily flaws like Hardcoded data, Insecure storage, Weak pins, Insecure WebView, and Native leaks can be exploited. It mainly highlights the importance of secure coding and thorough testing in Android app development.