mytashicell_apk_analysis

Executive Summary

This report presents a comprehensive static analysis of the “MyTashiCell” Android application (package name: `com.tashicell.selfcareapp`). The APK was reverse engineered using Mobile Security Framework (MobSF) and tools like JADX. The analysis revealed critical and warning-level security issues, including cleartext traffic usage and insecure encryption practices. Despite the absence of trackers, the application scored 42/100 on MobSF’s security scale, indicating a moderate risk profile. The report covers architecture insights, permissions, cryptographic implementations, vulnerabilities, and ethical reflections.

Introduction

MyTashiCell is the self-care mobile application provided by Bhutan Telecom company, TashiCell. This application allows users to manage their mobile accounts, recharge, check usage, and more. The aim of this analysis is to evaluate its internal structure, permissions, security features, and highlight areas for improvement using APK reverse engineering techniques.

Tools and Environment

Tools Used:
- MobSF (Static Analyzer)
- JADX
- APKTool

Configuration:
All tools were configured on aWindows 11 machine and Java Runtime Environment (JRE) for decompilation tools.

APK Analysis Process

1. **APK Extraction**: `My_Tashicell_App_2025_05_06_16_14_32.apk` (30.54 MB)

2. **Decompilation**:
    - Used APKTool to extract and decode resources.
    - Decompiled `.dex` files to readable `.java` using JADX .
3. **MobSF Static Scan**: Run on local MobSF instance.

Key Artifacts:

- `AndroidManifest.xml`
- `MainActivity` : `com.tashicell.selfcareapp.MainActivity`
- DEX signature hashes: SHA256 - `04ba3829493cb3ded3a7ec303c62cac343a36e1a59bce5aceba326ce705a9e4e`

Challenges:

- Flutter obfuscation reduced visibility of logic.
- Interpreting native libraries required examining shared object `.so` binaries.

Application Architecture

Key Components:
- Activities:
`MainActivity` , `BarcodeCaptureActivity` , `WebViewActivity` , `GoogleApiActivity`
- No exported receivers, services, or providers
- Flutter/Dart based architecture

Data Flow:
- User Input → Activity Intent → API Request (HTTP/HTTPS) → Server Response → Render via Flutter UI

Permissions and Security Analysis

Permissions Requested:
-
`CAMERA`
-
`INTERNET`

-
`ACCESS_NETWORK_STATE`
-
`READ/WRITE_EXTERNAL_STORAGE`
-
`FLASHLIGHT`

Security Concerns:

-
`usesCleartextTraffic=true` : exposes app to MITM attacks
- Application is installable on Android 4.4 (SDK 20), a vulnerable OS version
- Backup flag not explicitly disabled (privacy risk)

Security Mechanisms:
- Stack canary, NX bit, and PIE enabled in native binaries
- No code obfuscation or control flow integrity features found

Code Analysis Findings

Issues Identified:

- **CBC mode with PKCS7 padding** (padding oracle risk)
- **Insecure PRNG** (`Random`) used
- **Sensitive info copied to clipboard**
- **Logging of sensitive info**
- **Temp file creation with weak permissions**
- **External storage read/write** (risk of data leakage)

Code Quality:

- Logic split between native C++ Flutter libraries and Dart glue code
- API communication endpoints are hardcoded
- Use of multiple libraries from AndroidX

Conclusion and Reflection

Summary:

- The MyTashiCell app, while functional, has several areas that require immediate security improvements.
- The APK analysis exposed risky practices like logging, insecure encryption, and poor permission hygiene.

Learning Outcomes:

- Understood the static analysis workflow using MobSF and reverse engineering tools
- Gained experience in interpreting manifest configurations, native binaries, and Flutter structures
- Developed technical skills in code analysis, threat identification, and mitigation suggestions

Recommendations:

- Disable backup by default
- Avoid cleartext traffic
- Replace insecure encryption (CBC) with AES-GCM
- Use secure random generator (e.g., `SecureRandom`)

References

- OWASP Mobile Security Testing Guide
- Android Developer Documentation