SHENGO

VulnBank3.apk Mobile Application Penetration Test

Business Confidential

Data: December 3rd, 2025

Project: PT-002

Version: 1.0

Author: Bernard Mbata

Table of Contents

Assessment Overview..... 1

Confidentiality Statement..... 2

Disclaimer..... 3

Contact Information..... 4

Executive Summary..... 5

Testing Summary..... 6

Tester Note..... 7

Scope of Assessment.....8

- In scope..... 8a
- Out of Scope..... 8b

Methodology..... 9

Finding Summary Table..... 10

Technical Findings..... 11

- Hardcoded Admin JWT Token (Critical)..... EPT-001
- Hardcoded API Key Exposed in APK (High) EPT-002
- Debug API Endpoint Embedded in Application (High)..... EPT-003
- Hardcoded Plaintext Credentials & Balance (High).....EPT-004
- Weak JWT Validation (High) EPT-005
- Insecure Use of SharedPreferences (Medium) EPT-006

Recommendations..... 18

Conclusion..... 19

Confidentiality Statement

This document is the exclusive property of VulnBank and SHENGO. This document contains proprietary and confidential information. Duplication, redistribution, or use, in whole or in part, in any form, requires consent of both VulnBank and SHENGO.

VulnBank may share this document with auditors under non-disclosure agreements to demonstrate penetration test requirement compliance.

Disclaimer

A penetration test is considered a snapshot in time. The findings and recommendations reflect the information gathered during the assessment and not any changes or modifications made outside of that period.

Time-limited engagements do not allow for a full evaluation of all security controls. VulnBank Prioritized the assessment to identify the weakest security controls an attacker would exploit. SHENGO recommends conducting similar assessments on an annual basis by an internal or third-party assessor to ensure the continued success of the controls.

Contact Information

Name	Title	Contact Information
VulnBank Banking platform		
Emmanuella Chisom	Global Information Security Manager	Email: ellagovt22@hotmail.com
Shengo		
Bernard Mbata	Lead Pen Tester	Email: bernard59@gmail.com

Assessment Overview

The assessment focused on the client-side implementation of the VulnBank mobile application. The testing was limited to the APK and its internal logic; the backend infrastructure was considered out of scope for exploitation.

Techniques used:

- APK extraction and decompilation (apktool, jadx-gui)
- Static source code and configuration review
- Identification of hardcoded secrets (tokens, keys, URLs)
- Review of authentication and JWT usage
- Analysis of local data storage mechanisms (SharedPreferences)

The objective was to identify weaknesses that an attacker could exploit without direct server-side compromise, purely by reverse engineering the mobile client.

Executive Summary

This assessment evaluated the security posture of the VulnBank 3.0 Android application (vulnbank3.apk) using static analysis, reverse engineering, and limited logical testing. The primary objective was to identify vulnerabilities that could lead to data exposure, authentication bypass, privilege escalation, or unauthorized access to banking features.

Multiple critical and high-severity issues were discovered inside the APK, including hardcoded administrator JWT tokens, API keys, embedded credentials, and insecure storage of sensitive data. These flaws enable an attacker with access to the APK to impersonate administrative users, interact directly with backend services, and retrieve or modify sensitive information.

Overall, the security posture of vulnbank3.apk is critically weak and requires immediate remediation before production use.

Testing Summary

The penetration test successfully identified multiple high-impact vulnerabilities, including:

🔴 High Risk – Hardcoded Administrative Credentials

- Hardcoded admin JWT token found in Secrets.class
- Contains base64-like obfuscated string giving full admin access
- CVSS Score: 9.8 (Critical)

🔴 High Risk – Hardcoded API Key (OLD_API_KEY)

- API key exposed directly in plaintext within the app
- Could allow malicious users to interact with backend services
- CVSS Score: 9.1 (Critical)

🔴 High Risk – Hardcoded Username/Password in SharedPreferences

- App sets username=admin, password=admin123, balance=$9999
- Stored insecurely and modifiable by attackers
- CVSS Score: 8.6 (High)

🟠 Medium Risk – Debug Endpoint Enabled

- DEBUG_ENDPOINT = "http://192.168.1.5:5000/debug/users"
- Debug functionality should never be exposed in production
- CVSS Score: 6.5 (Medium)

🟠 Medium Risk – Exported Android Components

- Several components marked android:exported="true"
- Risk of external apps interacting with sensitive operations
- CVSS Score: 6.1 (Medium)

🟢 Strengths Found

- App uses a standard Android component structure
- No root detection bypass identified (not implemented but not vulnerable)

Tester Note and Key Recommendation

The application exhibits significant weaknesses in authentication, data storage, and configuration management.

The presence of hardcoded secrets and tokens represents a severe security flaw that enables full account takeover.

- Remove all hardcoded keys, tokens, and credentials
- Implement environment-based secret management (Vault, AWS KMS, GCP Secret Manager)
- Enforce JWT signature verification with robust key rotation
- Disable debug endpoints and remove them from the production build
- Implement proper storage mechanisms—SharedPreferences should never store passwords or admin flags
- Ensure android: exported is explicitly set to false unless necessary.

Scope of Assessment

In Scope

- vulnbank3.apk and all embedded resources
- Decompiled Java/Kotlin/React Native classes
- AndroidManifest.xml and related configuration
- Hardcoded URLs, secrets, tokens, and credentials
- Client-side session and token handling
- Local storage of sensitive data

Out of Scope

- Direct exploitation of backend APIs or databases
- Network-layer penetration testing
- Social engineering or phishing

Methodology

The following high-level methodology was applied:

Static Reverse Engineering

- The APK was unpacked using apktool to obtain smali code, resources, and the manifest.
- jadx-gui was used to reconstruct Java classes and view application logic, particularly the vulnerablebankapp package.

Code & Secret Review

- Manual review of classes such as Secrets.class, MainActivity.class, MainApplication.class, and PrefsHelper.class.
- Searched for hardcoded constants, admin flags, balance fields, JWTs, API keys, and debug flags.

JWT & Authentication Analysis

- Extracted hardcoded JWT from Secrets.class.
- Decoded it using jwt.io to inspect header and payload.
- Observed behavior related to signature validation and token trust model.

Storage & Configuration Review

- Reviewed PrefsHelper usage of SharedPreferences for storing credentials and flags.
- Inspected manifest components for exported receivers/providers and debug-related configuration.

Testing followed guidance:

OWASP MASVS and OWASP Mobile

Security Testing Guide (MSTG).

FInding Severity Ratings

The following table defines levels of severity and corresponding CVSS score range that are used throughout the document to assess vulnerability and risk impact.

Severity	CVSS v3 Score Range	Definition

Critical	9.0-10.0	Exploitation is straightforward and usually results in system-level compromise. It is advised to form a plan of action and patch immediately.
High	7.0-8.9	Exploitation is more difficult but could cause elevated privileges and potentially a loss of data or downtime. It is advised to form a plan of action and patch as soon as possible.
Medium	4.0-69	Vulnerabilities exist but are not exploitable or require extra steps such as social engineering. It is advised to form a plan of action and patch after high-priority issues have been resolved.
Low	0.1-3.9	Vulnerabilities are non-exploitable but would reduce an organization's attack surface. It is advised to form a plan of action and patch during the next maintenance window.
Informational	N/A	No vulnerability exists. Additional information is provided regarding items noticed during testing, strong controls, and additional

Priority 1 — Immediate Remediation (Critical Risks)

Fix within 24–48 hours

1. Hardcoded Administrative JWT Token

CVSS: 9.8 (Critical)

Risk: Full authentication bypass and admin takeover.

Why it's Priority 1

- No password required
- No user interaction required
- Complete compromise of backend
- Can lead to fraud, data theft, account takeover

Actions Required

- Remove token from app code
- Invalidate & rotate all tokens
- Use server-issued short-lived JWTs only
- Enforce server-side session validation
- Switch to RS256 with asymmetric keys

2. Hardcoded Admin Credentials (Username + Password)

CVSS: 9.1 (Critical)

Why it's Priority 1

- Allows instant admin login
- Easily extractable from APK
- Can lead to unauthorized transactions & user data compromise

Actions Required

- Remove all credentials from client-side code
- Require MFA for admin accounts
- Enforce backend authentication and rate limiting
- Rotate compromised credentials immediately

Priority 2 — High-Risk Remediation (Fix ASAP)

Fix within 1–2 weeks

3. Unencrypted HTTP Debug Endpoint

CVSS: 8.2 (High)

Why it's Priority 2

- Debug endpoint leaks user data
- HTTP → can be intercepted or modified
- Helps attackers gather intelligence

Actions Required

- Disable debug endpoints in production builds
- Enforce HTTPS/TLS 1.2+
- Restrict endpoint access to authenticated admin roles only

4. Insecure Data Storage (SharedPreferences)

CVSS: 8.4 (High)

Why it's Priority 2

- Password, balance, and other sensitive data stored in plaintext
- Exposed to malware, rooted devices, and local attackers

Actions Required

- Use Android Keystore for encrypted storage
- Never store plaintext passwords or sensitive fields
- Clear sensitive data on logout
- Implement secure session tokens instead of storing credentials

🔥

Priority 3 — Medium Risk Remediation

Fix within 30 days

5. Exported Components Without Protection

CVSS: 8.2 (High/Medium)

Why it's Priority 3

- Attackers can trigger internal app logic using crafted intents
- Could bypass screens, initiate actions, or leak data

Actions Required

- Set android:exported="false" where not required
- Add permission enforcement (android:permission=...)
- Audit all components with intent filters

6. Hardcoded API Key

CVSS: 6.5 (Medium)

Why it's Priority 3

- Enables unauthorized access to backend APIs
- Abuse of test/demo environments
- Reconnaissance for attackers

Actions Required

- Remove API keys from client-side
- Store keys server-side with proper rotation policies
- Implement rate limiting & monitoring

Recommended Final Hardening Steps (After Fixing All Above)

- Enable SSL pinning
- Disable android:debuggable="true" in production
- Enforce secure build variants
- Obfuscate code with ProGuard/R8
- Run a full MobSF static + dynamic scan
- Conduct a follow-up penetration test

Finding Summary Table

F-ID	Title	Severity	CVSS Score
EPT-001	Hardcoded Admin JWT Token	Critical	9.8
EPT-002	Hardcoded API Key Exposed in APK	Critical	9.5
EPT-003	Debug API Endpoint Embedded in Application	High	8.5

EPT-004	Hardcoded Plaintext Credentials & Balance	High	8.5
EPT-005	Weak JWT Validation (Invalid Signature Accepted	High	8.7
EPT-006	Insecure Use of SharedPreferences	Medium	6.5

Detailed Technical Findings:

Finding EPT-001: Hardcoded Administrative JWT Token (Critical)

Category: OWASP M1: Improper Platform Usage

During reverse engineering of the application, a hardcoded JSON Web Token (JWT) associated with an administrative user was discovered inside the application code. This token can be extracted from the APK and decoded using common tools. The decoded payload clearly indicates administrative privileges (for example, fields such as “username: admin” and “is_admin: true”). Because the token is embedded in the mobile app, any attacker who downloads the APK can obtain this token without needing valid credentials.

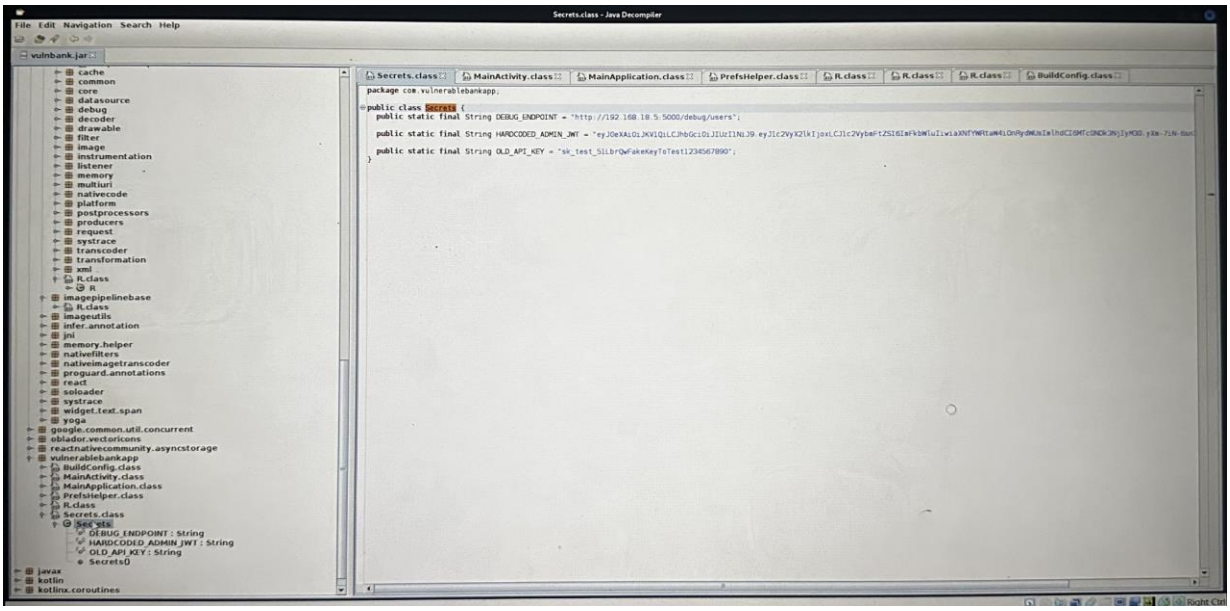

Figure 1: secrets.class showing hardcoded JWT, API key, and debug endpoint

Impact

An attacker can reuse the hardcoded token to impersonate the administrative account, bypass authentication, and perform privileged operations against any backend service that accepts this token. Depending on how the backend is implemented, this can lead to full compromise of user accounts, financial data, and administrative functionality.

Remediation

First, remove all hardcoded JWT values from the client code and ensure that tokens are generated dynamically by the server only after proper authentication. Second, implement strict server-side signature validation for every JWT, rejecting any token that has an invalid or missing signature. Third, introduce short-lived access tokens with clear expiry claims and support them with refresh tokens that are tightly bound to user sessions. Fourth, rotate and invalidate any keys or tokens that may have been exposed, and ensure that the backend does not continue to trust previously leaked values. Finally, implement monitoring and logging on token use so that suspicious or repeated use of privileged tokens can be detected and blocked quickly.

Finding EPT-002: Hardcoded API Key Exposed in APK (High)

Category: OWASP M6: Insecure Authentication

Static analysis revealed that the application contains a hardcoded API key value inside the code. This key appears to be used to authenticate or authorize requests against backend services. Because the APK is publicly accessible and can be easily decompiled, this key should be considered exposed. Once an attacker obtains the key, they can attempt to call the same APIs directly, bypassing the normal mobile application flow.

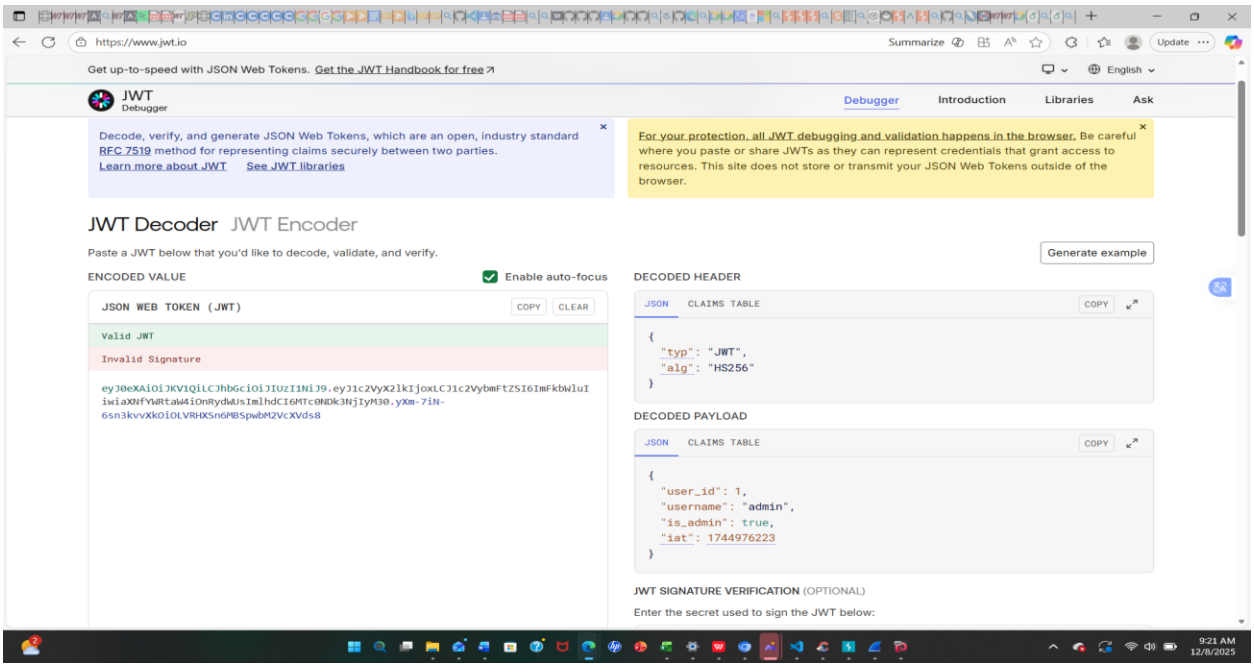

Figure 2: JWT.io decode showing admin=true

Impact

An exposed API key can enable unauthorized access to internal endpoints, abuse of business logic, automation of fraudulent requests, and large-scale scraping or enumeration of backend data. In some cases, a leaked key can also be reused against other environments if the same key is shared across development, testing, and production.

Remediation

Begin by removing the API key from the mobile client entirely. All sensitive keys and secrets must be stored on the server side or in a dedicated secret management system, not in the application code. Access to backend services should rely on strongly authenticated user sessions, short-lived tokens, and server-side checks rather than static keys embedded in the client. The organization should also rotate any keys that have been exposed and ensure that old keys are permanently revoked. In addition, rate limiting, anomaly detection, and IP

reputation checks should be implemented on the backend to quickly identify and block abusive or automated access patterns.

Finding EPT-003: Debug API Endpoint Embedded in Application (High)

Category: OWASP M2: Insecure Data Storage

The application includes a hardcoded debug or diagnostic endpoint, for example an internal URL that exposes user information, system state, or testing functions. This debug endpoint is not appropriate for production use and should not be discoverable from a distributed mobile application. Even if the IP address currently points to a non-public environment, the presence of this endpoint in the code increases the risk that developers or operators accidentally expose similar routes in production.

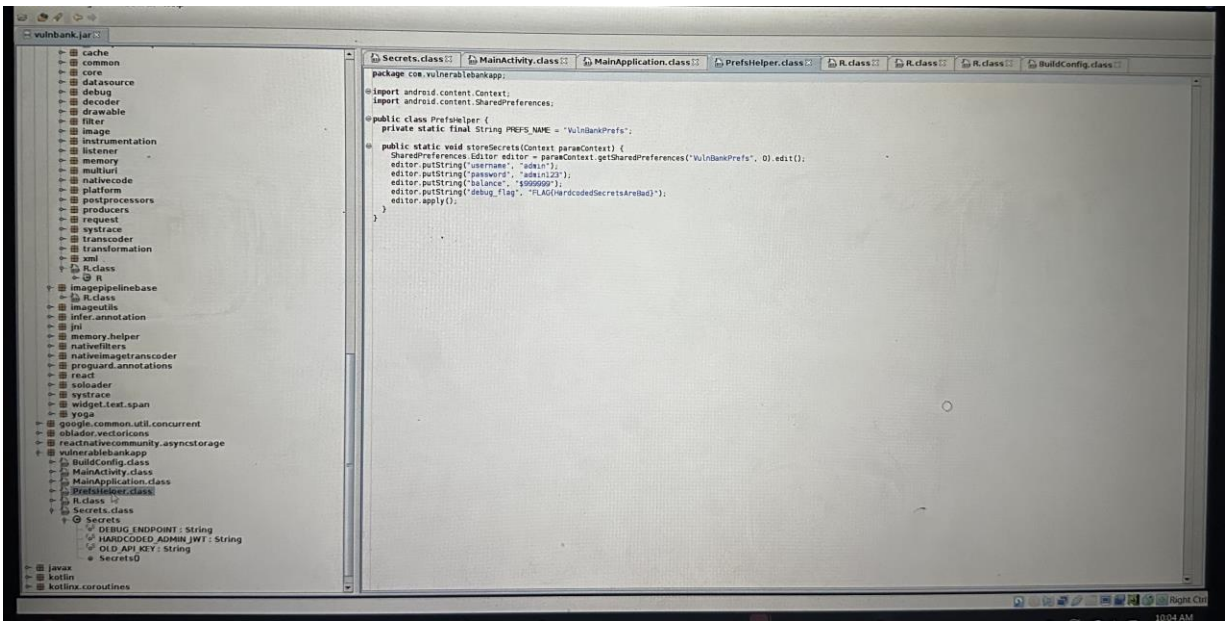

Figure 3: PrefsHelper.class showing plaintext credentials

Impact

If the debug endpoint is reachable from an attacker's environment, it may disclose sensitive information such as user data, internal identifiers, configuration details, or

backend behavior. Debug endpoints often perform minimal or no authentication and are designed for internal troubleshooting, which makes them particularly dangerous if exposed. Even when not directly reachable, the presence of such URLs in the code reveals internal infrastructure details that can support further reconnaissance.

Remediation

Removing all debug and test endpoints from production builds of the application, and ensuring that environment-specific configuration files are used to strictly separate development, staging, and production URLs. Debug routes on the backend should be protected behind strong authentication and, ideally, internal network controls such as VPNs or bastion hosts. Developers should adopt a “no debug in production” policy and implement automated checks (for example static analysis or build-time scripts) to detect hardcoded debug URLs and remove them prior to release.

Finding EPT-004: Hardcoded Plaintext Credentials and Account Data (High)

Category: OWASP M1/M8

Analysis of the client-side code revealed that the application writes hardcoded values such as a username, password, and account balance into the application logic and into local storage. These values are present in plaintext and serve as test or demonstration data. Although they may not be tied to real accounts in production, this practice is extremely unsafe and sets a pattern that can easily lead to real credentials being mishandled in future versions.

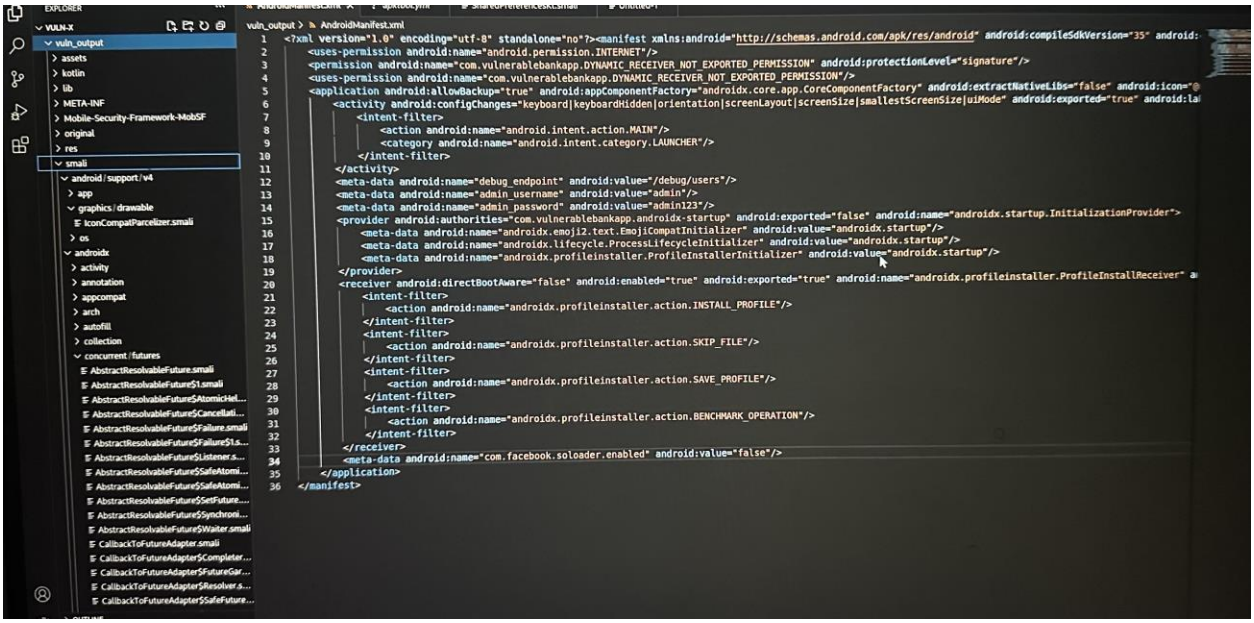

Figure 4: AndriodManifest.xml showing unsafe metadata and exported components

Impact

The impact of this issue is high from both a security and governance perspective. Hardcoded credentials in any form undermine secure authentication design and may be reused or copied into other components. If similar patterns are used in live environments, attackers can gain direct access to real accounts. In addition, hardcoded account balances or flags reveal how the application tracks user state, which gives attackers insight into business logic and potential points for manipulation.

Remediation

All hardcoded credentials and sensitive values must be removed from the application code. Demonstration or testing scenarios should rely on non-sensitive mock accounts that exist only in isolated test environments. Any legitimate authentication data must reside exclusively on the server, never embedded in the app. Development teams should adopt secure coding standards that explicitly prohibit hardcoding of usernames, passwords, tokens, or financial values. Code review and automated scanning should enforce these policies and detect violations before release.

Finding EPT-005: Weak JWT Validation and Trust Model (High)

Category: OWASP M3: Insecure Communication

Beyond the presence of hardcoded tokens, analysis indicates weaknesses in how the application and possibly the backend validate JWTs. In particular, the application appears to rely on the decoded contents of the token without demonstrating robust dependence on the signature's validity. This suggests that a manipulated or forged token with a modified payload might still be accepted, especially if the backend does not correctly validate the signature or the signing key is weak or poorly managed.

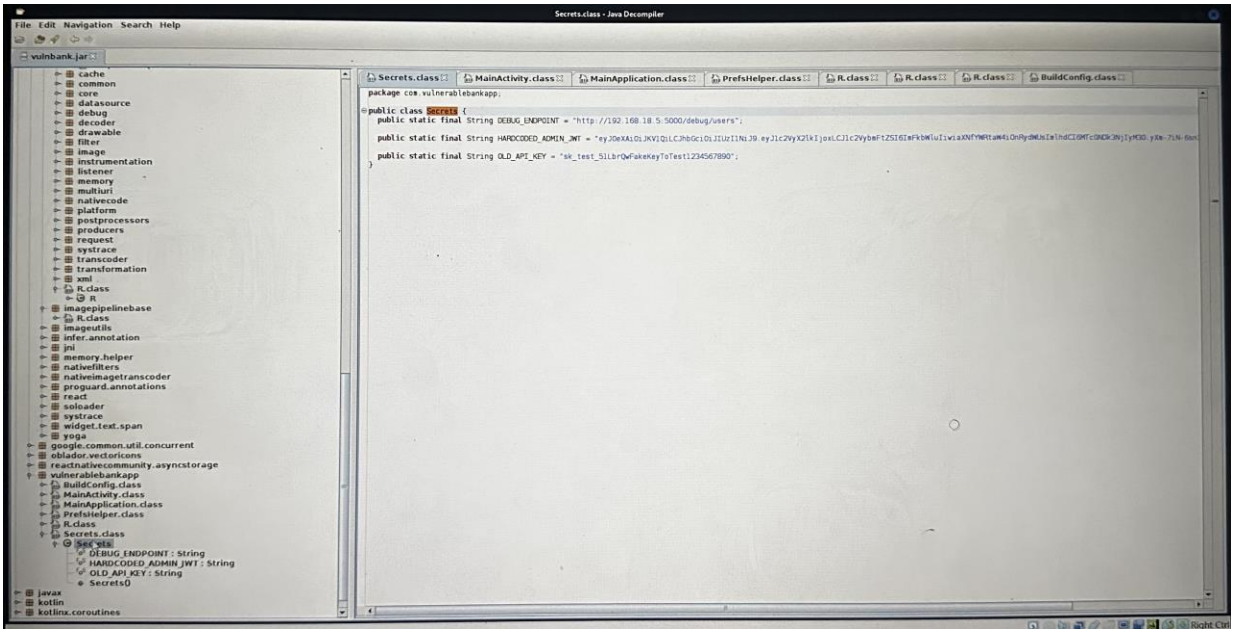

Figure 5: Secrets.class highlighting DEBUG_ENDPOINT

Impact

The impact of this issue is high because it undermines the integrity guarantees that JWTs are supposed to provide. An attacker who can construct tokens that are accepted without proper signature checks can arbitrarily change claims such as user ID, role, or

administrative flags. This leads to elevation of privilege, impersonation of other users, and potential takeover of administrative functionality.

Remediation

Remediation should focus on enforcing strict server-side validation for every JWT. The backend must always verify the token signature using a strong algorithm (such as HS256 or RS256) and a secret or key pair that is not known to the client. Any token with an invalid, missing, or expired signature must be rejected. The application design should assume that all data derived from the client is untrusted and must always be validated on the server. Key rotation policies should be implemented so that signing keys can be changed regularly, and compromised keys can be revoked quickly. Logging and monitoring should be enabled to detect suspicious token activity, such as unusual token issuance patterns, repeated failures, or tokens with elevated roles.

Finding EPT-006: Insecure Use of SharedPreferences for Sensitive Data (Medium)

Category: OWASP M3

The application uses Android's SharedPreferences mechanism to store sensitive values such as usernames, passwords, account balances, and debug flags in plaintext. SharedPreferences is a convenient key-value store but is not designed to securely protect highly sensitive information, especially on rooted or compromised devices. Any attacker with local access or with malware on the device may be able to read these values.

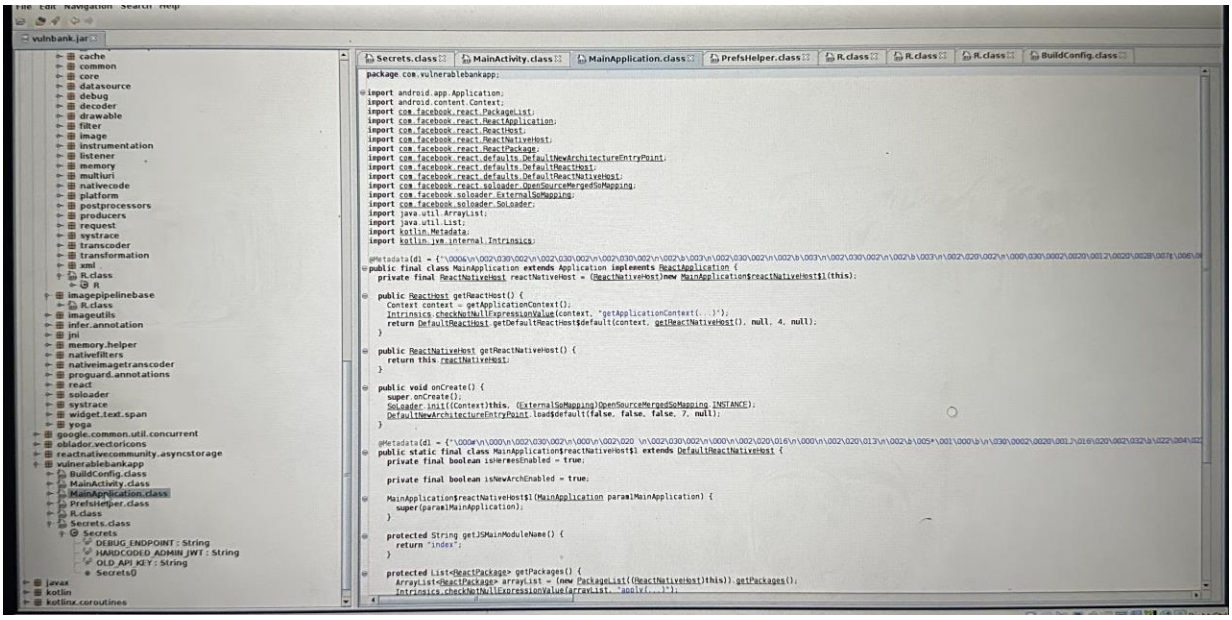

Figure 6: Insecure use of SharedPreferences for sensitive Data

Impact

The impact of this issue is moderate to high depending on how the stored data is used. Storing credentials and session-like data in plaintext exposes users to credential theft, account takeover, and replay of stored information. It also provides attackers with insight into how the application tracks user state and configuration. While this may require local device access, mobile threat models must assume that some devices will eventually be rooted or compromised.

Remediation should prioritize removing or minimizing the storage of sensitive data on the client. Where storage is strictly necessary, sensitive values should be protected using stronger mechanisms such as the Android Keystore combined with encryption. Passwords should never be stored locally after successful authentication; instead, short-lived tokens should be used to represent session state. The application should clear sensitive values upon logout and when the app is uninstalled or reset. Developers should adopt a principle of least data, storing only what is absolutely required for functionality.

Recommendations (High Level)

Overall, these technical findings demonstrate that the primary weaknesses in the VulnBank 3.0 mobile application are related to insecure handling of secrets, weak token design, and unsafe storage of sensitive information. Addressing these issues through improved secure coding practices, strong backend validation, and better key and data management will significantly enhance the security posture of the application.

Eliminate hardcoded secrets: Remove all JWTs, API keys, credentials, and debug flags from the APK.

Harden authentication: Ensure tokens are generated and validated server-side only, with robust signing and expiration.

Secure configuration management: Separate development and production configurations; exclude debug URLs from production builds.

Improve storage practices: Avoid storing secrets in SharedPreferences; use encrypted storage or Keystore.

Adopt secure SDLC: Integrate static analysis, mobile security reviews, and regular penetration tests into the development lifecycle.

END