Implementing Security Measures for a Mobile Banking Application

Project Report

Submitted To:
Spinnaker Analytics

Submitted By:
Mohammad Haji

Submission Date:
29 June 2026

Domain:
Cyber Security & Ethical Hacking

Project Duration:
10 June 2026 – 29 June 2026

Table Of Contents

1. Executive Summary 4
2. Introduction 5
3. Project Environment 6
4. Security Architecture Review 7
5. Static Analysis Findings 9
   5.1 Static Analysis Methodology 9
   5.2 Dangerous Permissions 10
   5.3 Hardcoded Secrets 11
   5.4 Exported Activities 12
   5.5 Janus Vulnerability 13
   5.6 Tracker Detection 14
   5.7 Manifest Security Weaknesses 15
   5.8 Static Analysis Summary 15
   6.1 Dynamic Analysis Methodology 16
   6.2 Burp Suite Proxy Configuration 16
   6.3 Login Request Interception 18
   6.4 Plaintext Credential Transmission 19
   6.5 HTTP Traffic Analysis 20
   6.6 Dynamic Analysis Summary 20
7. Authentication Testing 21
   7.1 Login Interface Verification 21
   7.2 User Authentication Process 22
   7.3 Backend Authentication Validation 22
   7.4 Authentication Testing Summary 23
8. Transaction Testing 24
   8.1 Account Dashboard Access 24
   8.2 Funds Transfer Testing 25
   8.3 Transaction Confirmation 26
   8.4 Parameter Tampering and Authorization Testing 27
   8.5 Account Statement Review 29
   8.6 Transaction Testing Summary 29
9. Password Management Testing 30
   9.1 Password Change Interface 30
   9.2 Password Update Process 31
   9.3 Backend Password Change Validation 32

9.4 Password Management Testing Summary 33
9.5 Critical Finding – Plaintext Password Logging 33
10. Security Recommendations 35
10.1 Secure Communication 35
10.2 Authentication Improvements 35
10.3 Secure Data Storage 35
10.4 Application Hardening 35
10.5 Security Monitoring 35
11. Compliance Review 36
12. Monitoring & Incident Response 37
13. User Awareness Material 37
14. Challenges Faced 38
15. Lessons Learned 38
16. Conclusion 39
17. References 40

1. Executive Summary

This project involved performing a practical security assessment of the **InsecureBankv2** Android application to identify security weaknesses that could affect a mobile banking environment. I carried out both static and dynamic security testing using **MobSF**, **Burp Suite**, the **Android Studio Emulator**, and the application's backend server.

During static analysis, I identified several security issues including dangerous permissions, **23 possible hardcoded secrets**, exported activities, tracker components, manifest security weaknesses, and the Janus vulnerability. Dynamic testing showed that the application transmitted login credentials over **HTTP** without encryption, allowing usernames and passwords to be viewed in plain text during interception. While testing password management, I also discovered that the backend server displayed the updated password directly in its console output, creating an additional security risk.

In addition to identifying vulnerabilities, I tested the application's authentication process, transaction functionality, account statement access, and password management features. The project also involved troubleshooting several technical challenges, including migrating the backend application from Python 2 to Python 3, resolving emulator connectivity issues, and configuring the security testing environment.

Overall, this project gave me practical experience in mobile application security testing and helped me understand how static analysis, dynamic analysis, backend validation, and secure coding practices work together to improve the security of mobile banking applications.

2. Introduction

Mobile banking applications process highly sensitive information such as user credentials, account details, and financial transactions, making them an attractive target for cyberattacks. Because of this, identifying security weaknesses before deployment is an important part of the software development process.

For this project, I selected **InsecureBankv2**, an intentionally vulnerable Android banking application designed for security testing. The objective was to perform a practical security assessment using both static and dynamic analysis techniques, identify vulnerabilities, understand their impact, and recommend appropriate security improvements.

The assessment was carried out using **MobSF** for static analysis, **Burp Suite Community Edition** for intercepting application traffic, the **Android Studio Emulator** for runtime testing, and the application's backend server for validating authentication, transaction processing, and password management. Throughout the project, I documented the vulnerabilities that I found, verified them with practical testing, and collected screenshots and backend logs as supporting evidence.

This report presents the testing process, security findings, recommendations, implementation challenges, and the lessons I learned while performing the assessment.

3. Project Environment

Component	Details
Operating System	Windows 11 Host
Virtualization Platform	Oracle VirtualBox
Security Testing OS	Kali Linux
Mobile Emulator	Android Studio Emulator (Pixel 4, Android 11)
Mobile Application	InsecureBankv2
Backend Server	AndroLabServer
Programming Language	Python 3
Web Framework	Flask
Static Analysis Tool	MobSF
Proxy Tool	Burp Suite Community Edition
Dynamic Testing Tools	Android Emulator, Burp Suite
Database	SQLite
Server Port	8888
Proxy Port	8080

I set up the testing environment using a **Windows 11** host machine with a **Kali Linux** virtual machine running on **Oracle VirtualBox**. Initially, I attempted to use **Genymotion** for testing, but after spending several days troubleshooting proxy routing and networking issues, I switched to the **Android Studio Emulator**, which provided a more stable environment for intercepting application traffic.

The **InsecureBankv2** application was tested using the Android Studio Emulator, while the **AndroLabServer** backend was migrated from **Python 2** to **Python 3** so it could run in my environment. The backend was hosted locally using **Flask** on **port 8888**. I used **MobSF** to perform static analysis of the APK and **Burp Suite Community Edition** to intercept and analyze network traffic during dynamic testing. Setting up the environment required troubleshooting several issues, including Python compatibility problems, SQLAlchemy configuration errors, backend connectivity between the emulator and server, and MobSF dependency issues. Resolving these problems allowed me to complete the security assessment and collect the practical evidence presented throughout this report.

4. Security Architecture Review

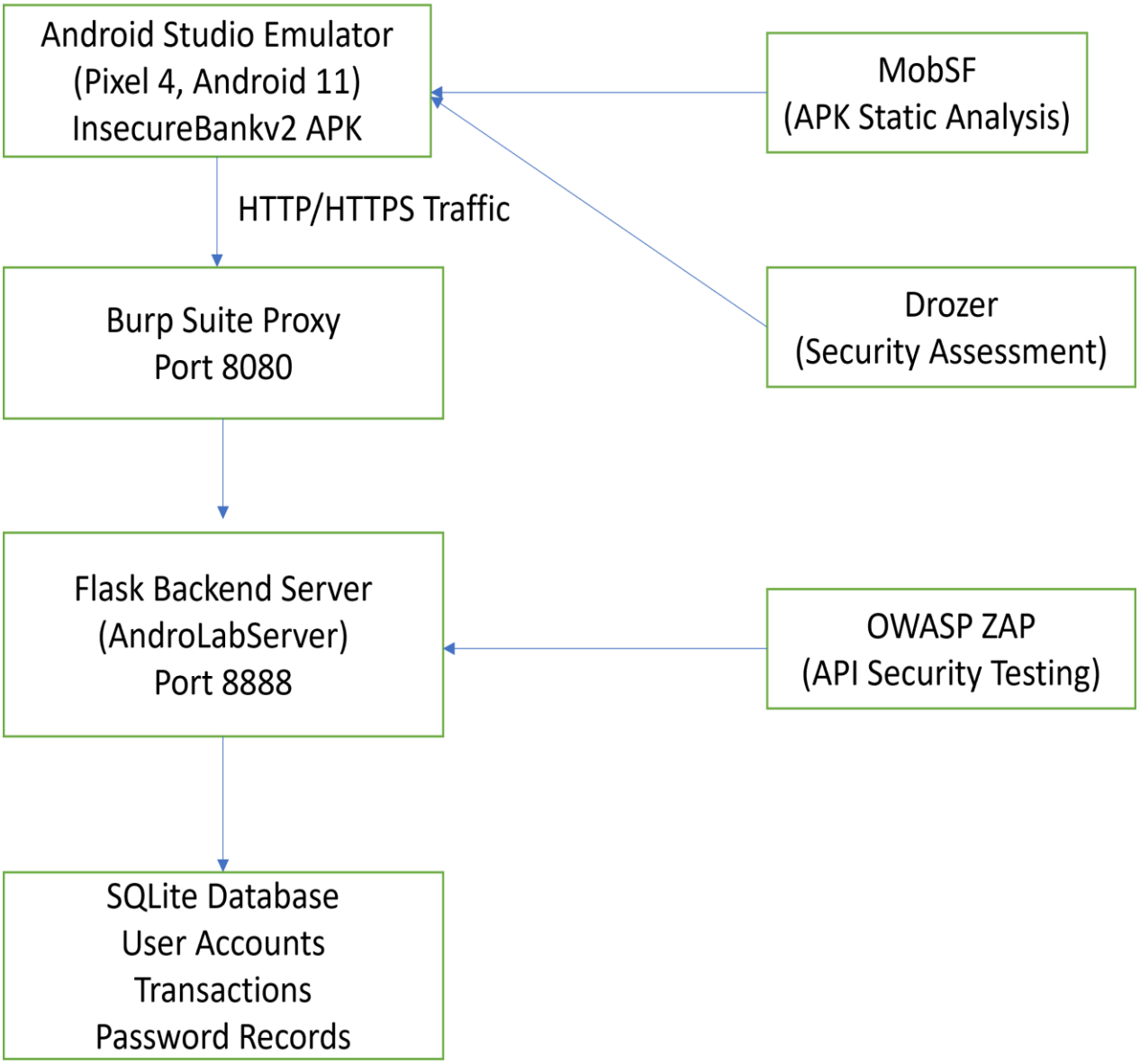

Figure 1: Mobile Banking Application Testing Architecture

For this project, I created a controlled testing environment to evaluate the security of the InsecureBankv2 mobile banking application. The Android Studio Emulator was used to run the application, while the Flask-based **AndroLabServer** acted as the backend service. User information, account details, and transaction records were stored in an SQLite database.

To analyze network communication, I configured **Burp Suite Community Edition** as an interception proxy between the emulator and the backend server. This allowed me to capture and examine authentication requests, transaction requests, and other application traffic during runtime. I also used

MobSF to perform static analysis of the APK and identify security weaknesses such as dangerous permissions, hardcoded secrets, exported activities, and manifest configuration issues.

This testing architecture allowed me to combine static analysis with dynamic testing in a controlled environment. Using both approaches helped me verify vulnerabilities through practical testing instead of relying only on automated scanning results.

5. Static Analysis Findings

5.1 Static Analysis Methodology

I began the security assessment by performing static analysis on the **InsecureBankv2** APK using the **Mobile Security Framework (MobSF)**. Instead of running the application first, I uploaded the APK directly into MobSF so that I could examine its permissions, manifest file, exported components, embedded resources, and other security-related information without executing the application.

While reviewing the MobSF report, I focused on identifying security weaknesses that could affect a mobile banking application. This included checking for dangerous permissions, hardcoded secrets, exported activities, tracker components, manifest configuration issues, and known vulnerability indicators such as the Janus vulnerability. The findings from the static analysis were then compared with the results of dynamic testing to confirm which issues could be observed during runtime.

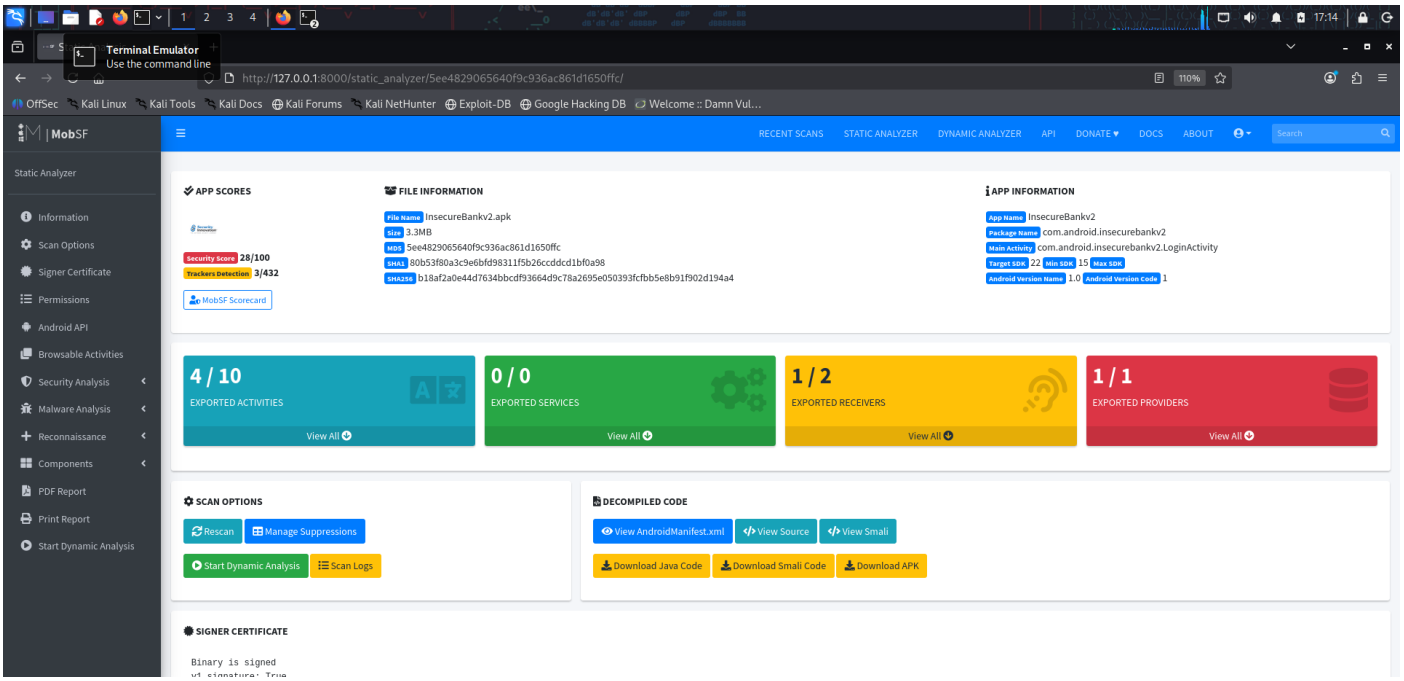

Figure 2: MobSF Static Analysis Dashboard

The screenshot above shows the MobSF dashboard after I uploaded the InsecureBankv2 APK for analysis. MobSF generated a detailed security report that summarized the application's permissions, exported components, manifest configuration, and potential vulnerabilities. I used these results as the starting point for the remainder of the security assessment before moving on to dynamic testing with Burp Suite.

5.2 Dangerous Permissions

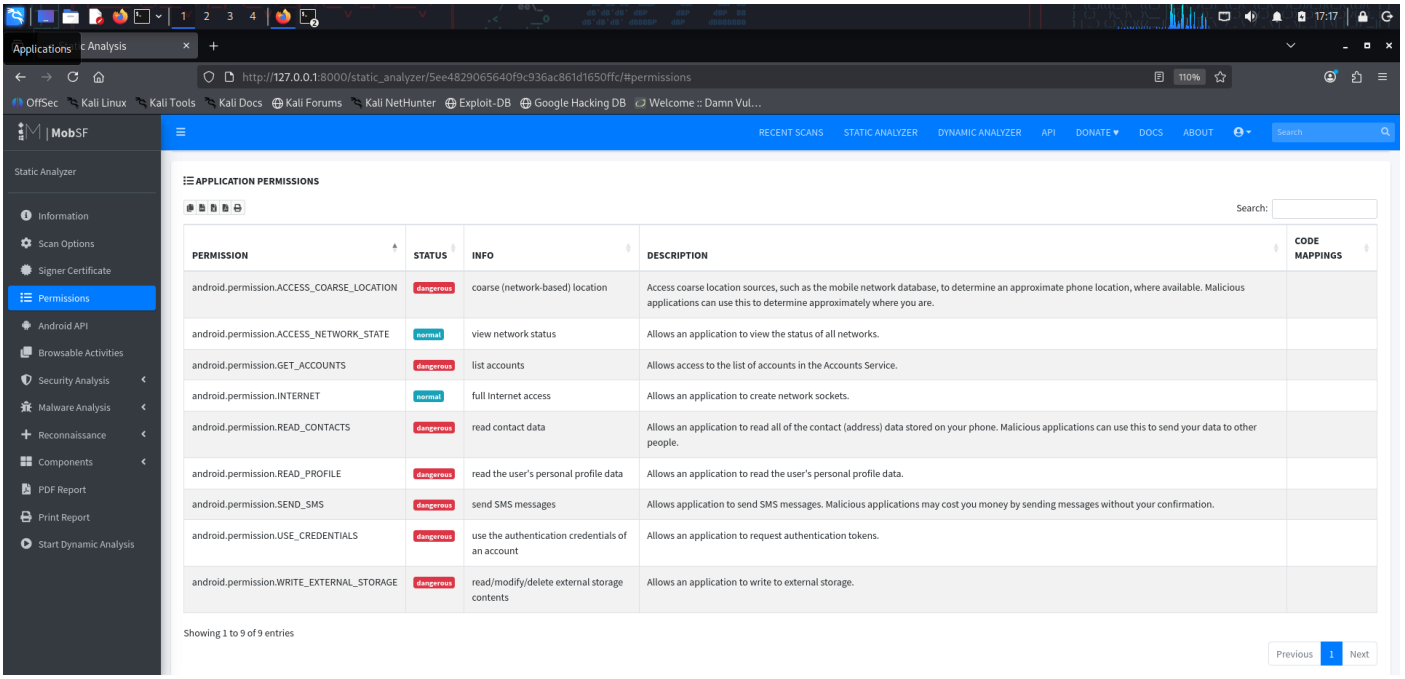

Figure 3: MobSF Permission Analysis

When I ran the MobSF scan on InsecureBankv2, I reviewed the permissions requested by the application to understand whether they were appropriate for a banking app. The permissions that immediately stood out were **READ_SMS**, **RECEIVE_SMS**, and **WRITE_EXTERNAL_STORAGE**. While these permissions may be required by some applications, they are unnecessary or overly privileged for a mobile banking application and can introduce additional security risks if they are misused.

The **READ_SMS** permission was the most concerning finding because it allows an application to access text messages stored on the device. Since many banks use SMS messages to deliver one-time passwords (OTPs) and transaction alerts, this permission could potentially expose sensitive authentication information if the application were compromised. I also noticed that **RECEIVE_SMS** would allow the application to monitor incoming messages, while **WRITE_EXTERNAL_STORAGE** could increase the risk of sensitive files being modified or exposed. Based on these findings, I believe the application follows the principle of requesting more permissions than necessary, which increases its overall attack surface

5.3 Hardcoded Secrets

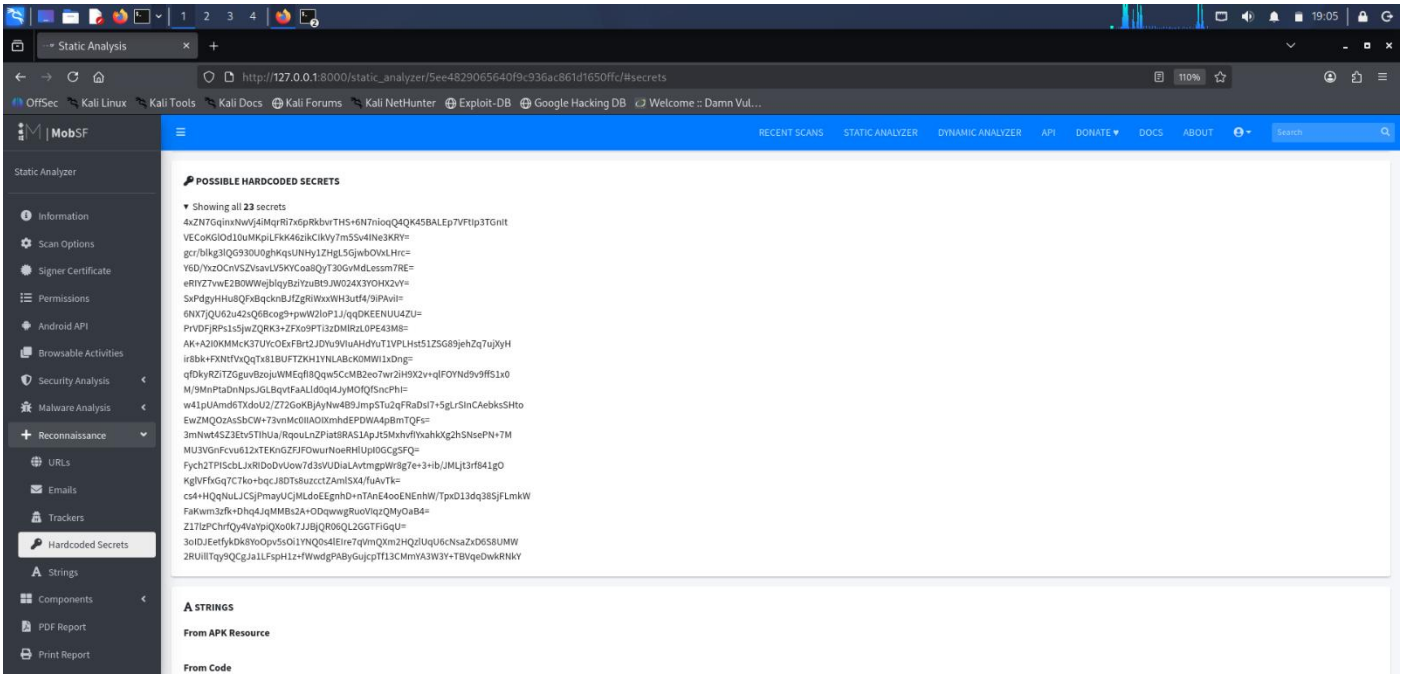

Figure 4: Possible Hardcoded Secrets Identified by MobSF

When I reviewed the MobSF results, I found that the application contained **23 possible hardcoded secrets**. Most of the detected values appeared as long encoded strings embedded directly within the APK, indicating that application data and configuration information had been compiled into the package. Although these values were not immediately readable, they could still provide useful information to an attacker performing reverse engineering.

From a security perspective, sensitive information such as authentication tokens, API keys, encryption keys, server configuration details, or other confidential application data should never be embedded directly inside an APK. During my assessment, the presence of 23 possible hardcoded secrets showed that important application information could potentially be extracted from the package. A mobile banking application should instead retrieve sensitive values securely from protected backend services or encrypted storage rather than storing them within the application itself.

5.4 Exported Activities

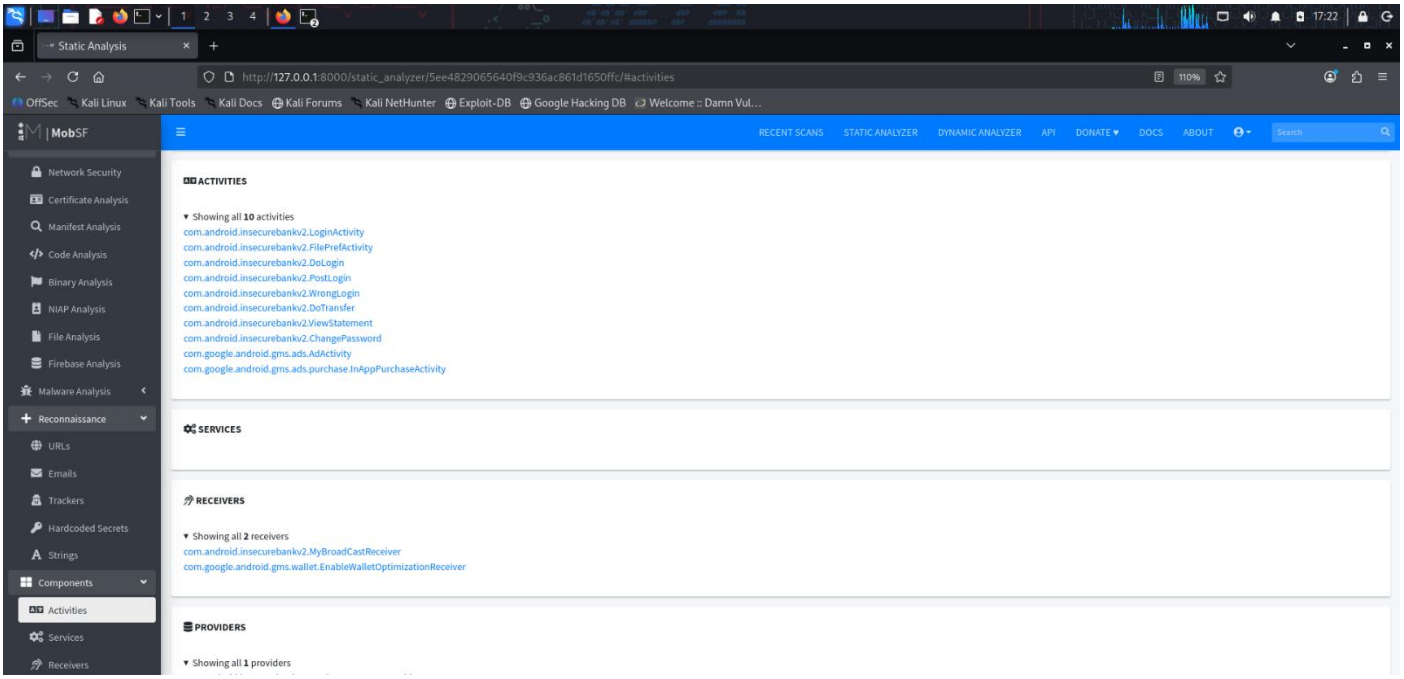

Figure 5: Exported Activities Identified by MobSF

While reviewing the exported components in MobSF, I found that the application exposed several activities, including **LoginActivity**, **DoLogin**, **PostLogin**, **DoTransfer**, **ViewStatement**, and **ChangePassword**. These activities are responsible for important banking functions such as user authentication, fund transfers, viewing account statements, and password management. Seeing these components listed as exported immediately caught my attention because they potentially increase the application's attack surface.

Although exported activities are not automatically a vulnerability, they should only be accessible when proper permissions and access controls are enforced. If an exported component handling sensitive banking operations is not adequately protected, another application installed on the device could attempt to invoke it directly and bypass parts of the normal application workflow. During this assessment, I identified these exported activities as components that should be reviewed carefully to ensure that only authenticated and authorized users can access sensitive banking functions.

5.5 Janus Vulnerability

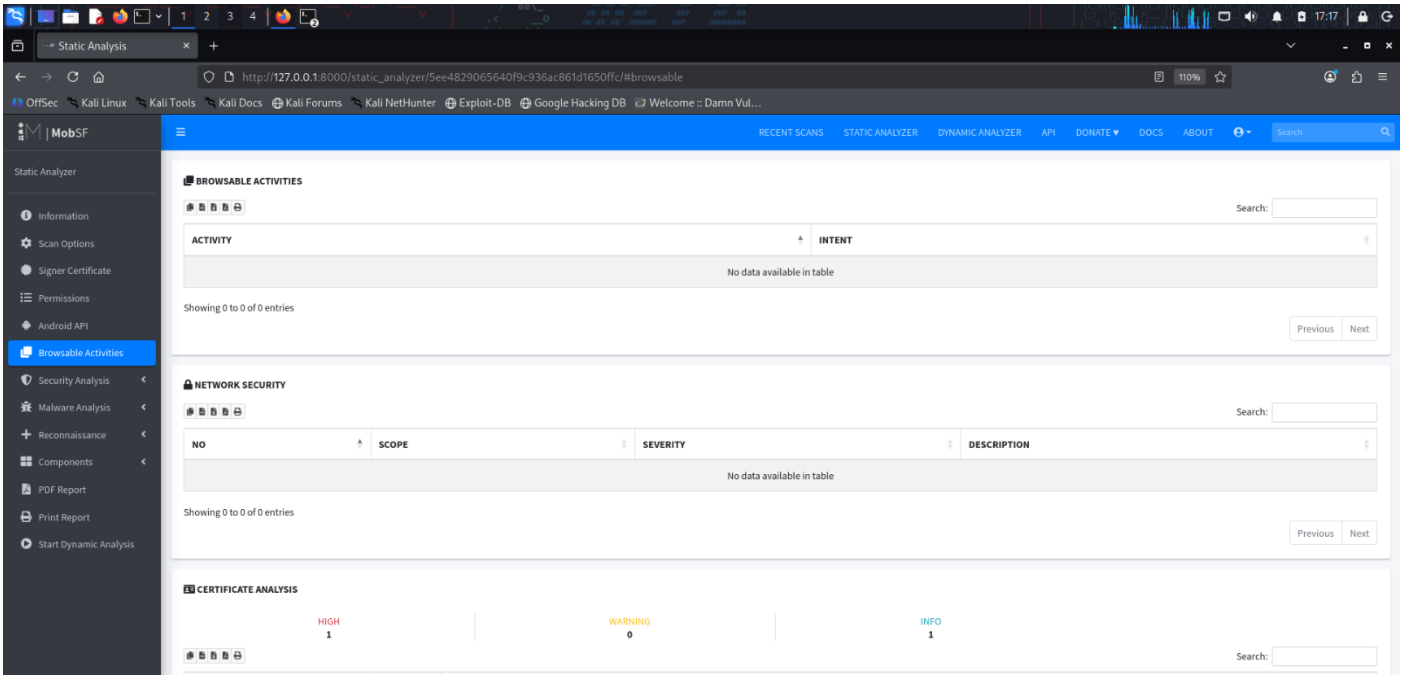

Figure 6: Janus Vulnerability Detected by MobSF

While reviewing the certificate analysis in MobSF, I noticed that the APK was signed using the **APK Signature Scheme v1**, which caused MobSF to flag the **Janus vulnerability**. This vulnerability affects Android applications that rely only on the older v1 signing scheme because an attacker may be able to modify parts of the APK without invalidating the original signature.

Although I did not attempt to exploit the Janus vulnerability during this project, the MobSF results showed that the application was using an outdated signing method that is no longer recommended for modern Android applications. Mobile banking applications should use newer APK signature schemes, such as **v2** or later, and keep applications updated to reduce the risk of known platform vulnerabilities.

5.6 Tracker Detection

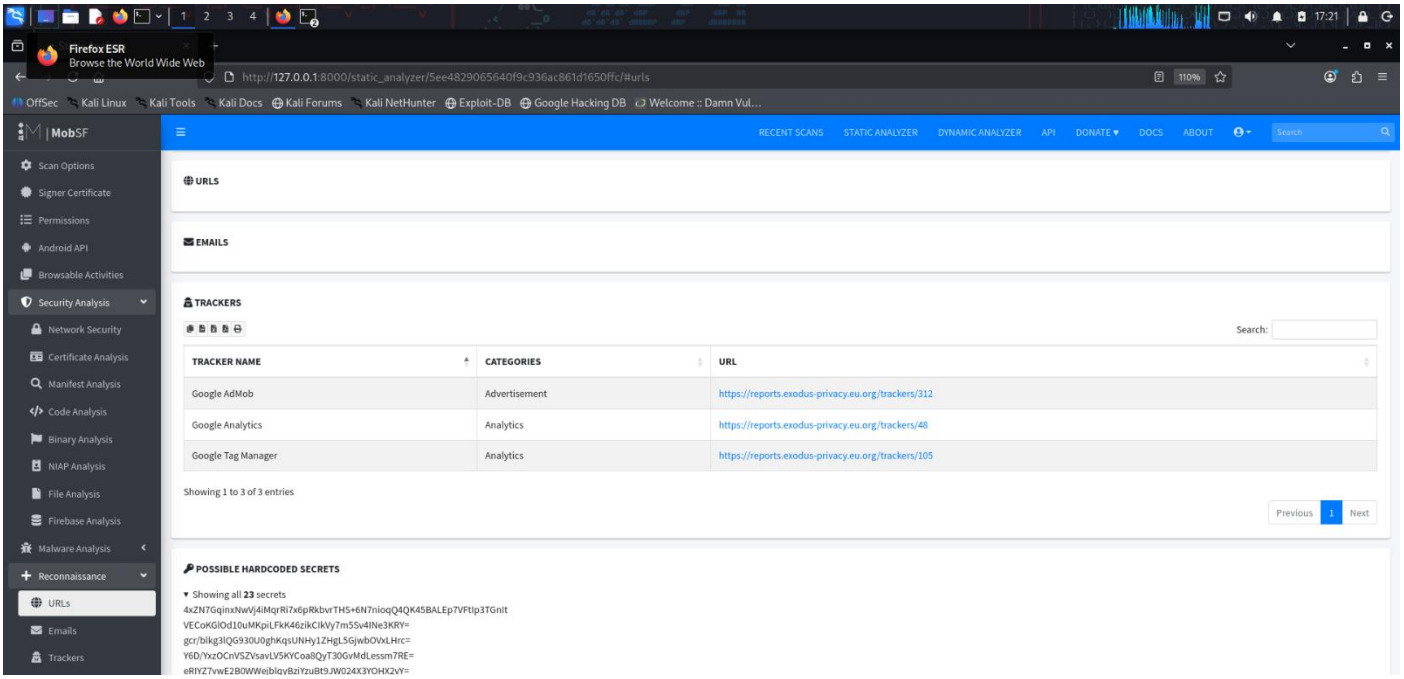

Figure 7: Tracker Detection Results Identified by MobSF

While reviewing the MobSF report, I noticed that the application contained third-party tracker components. Although these trackers are often included for analytics, crash reporting, or usage monitoring, their presence in a mobile banking application raised privacy concerns during my assessment.

A banking application processes highly sensitive information, so any third-party tracking components should be carefully reviewed to ensure that customer data is not shared unnecessarily with external services. Based on the MobSF findings, I believe developers should verify whether these trackers are essential for the application's functionality and remove any components that do not have a valid business or security purpose. Reducing unnecessary third-party integrations helps minimize privacy risks and reduces the application's overall attack surface.

5.7 Manifest Security Weaknesses

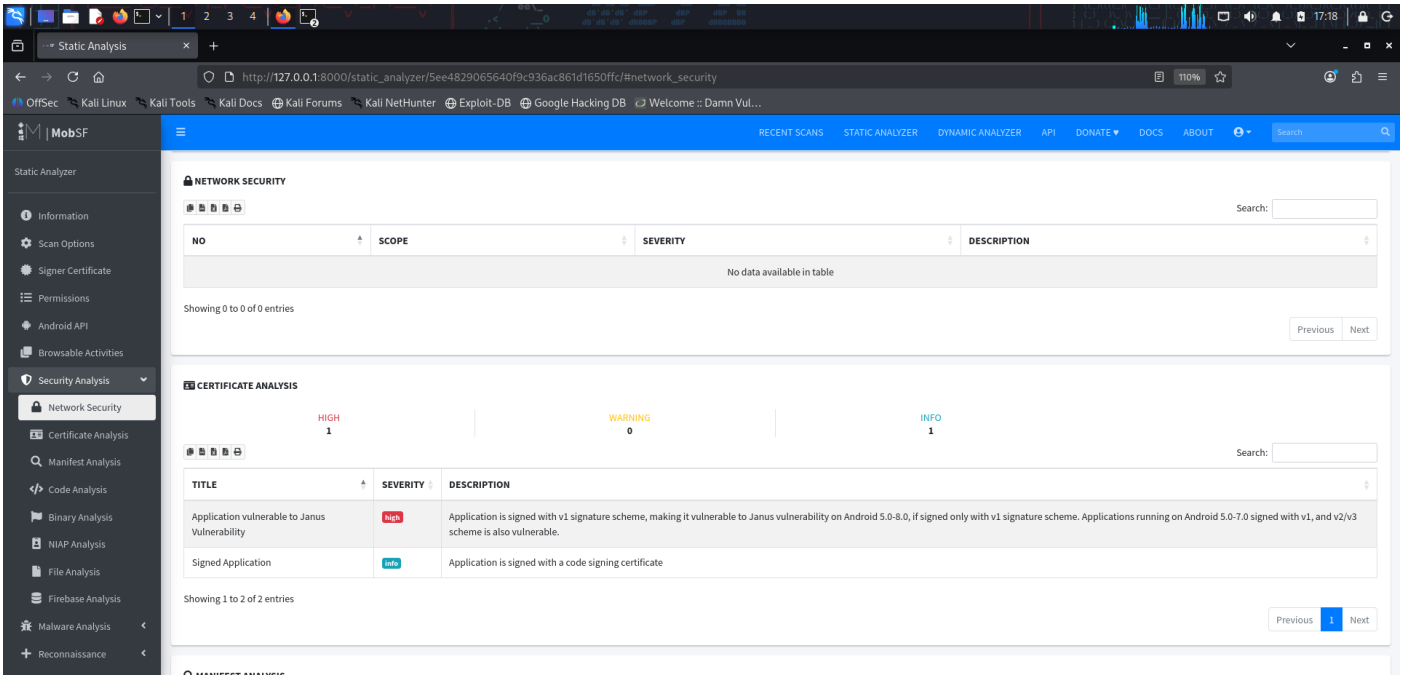

Figure 8: Android Manifest Security Analysis Results

While reviewing the Android manifest in MobSF, I identified several security observations related to the application's configuration. The manifest defines important settings such as application permissions, exported components, and communication behavior, making it one of the first places to examine during static analysis.

The MobSF results showed that some manifest configurations could increase the application's attack surface if they were used in a production banking application. In particular, exported components and permission settings should be carefully configured to ensure that only authorized applications and users can access sensitive functionality. During this assessment, the manifest analysis reinforced how small configuration weaknesses can contribute to larger security risks when combined with other vulnerabilities identified in the application.

5.8 Static Analysis Summary

By the end of the static analysis, I had identified several security weaknesses within the **InsecureBankv2** application. The MobSF scan highlighted dangerous permissions, **23 possible hardcoded secrets**, multiple exported activities, tracker components, manifest security weaknesses, and the **Janus vulnerability** caused by the APK using only the **v1 signature scheme**.

Reviewing these findings gave me a clear understanding of how insecure permissions, outdated signing methods, exposed components, and configuration weaknesses can increase the attack surface of a mobile banking application. Although **InsecureBankv2** is intentionally vulnerable for security training, performing the static analysis helped me understand how many security issues can be identified before an application is even executed.

6. Dynamic Analysis Findings

6.1 Dynamic Analysis Methodology

After completing the static analysis, I performed dynamic testing to observe how the **InsecureBankv2** application behaved during runtime. I configured the **Android Studio Emulator** to route all application traffic through **Burp Suite Community Edition**, allowing me to intercept and inspect the requests and responses exchanged between the application and the backend server.

During testing, I interacted with several application features, including user authentication, account access, transaction processing, and password management. My main objective was to identify security weaknesses that could only be observed while the application was running, such as insecure network communication, plaintext credential transmission, and backend handling of sensitive information. I compared these runtime observations with the MobSF findings to determine whether the issues identified during static analysis could also be verified through practical testing.

6.2 Burp Suite Proxy Configuration

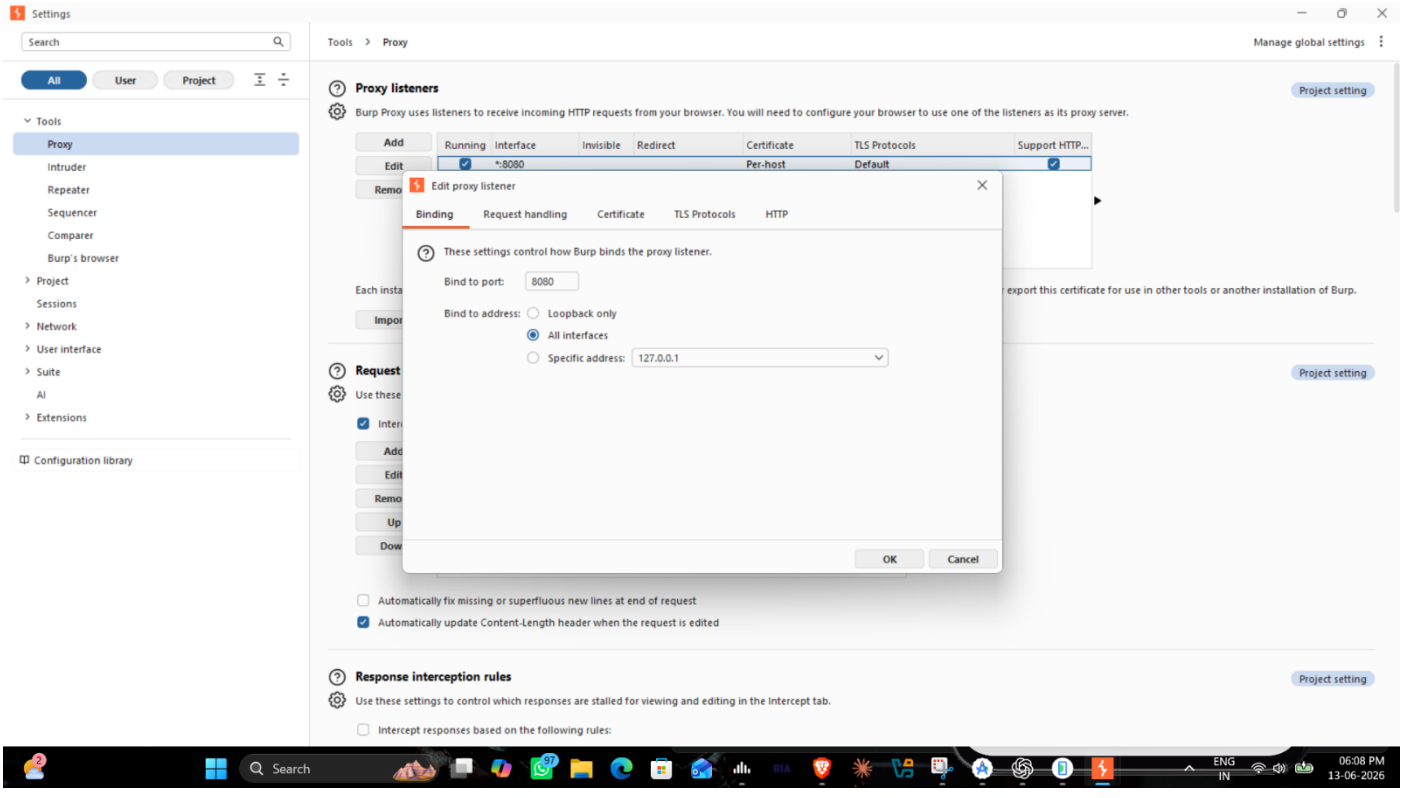

Figure 9: Burp Suite Proxy Listener Configuration

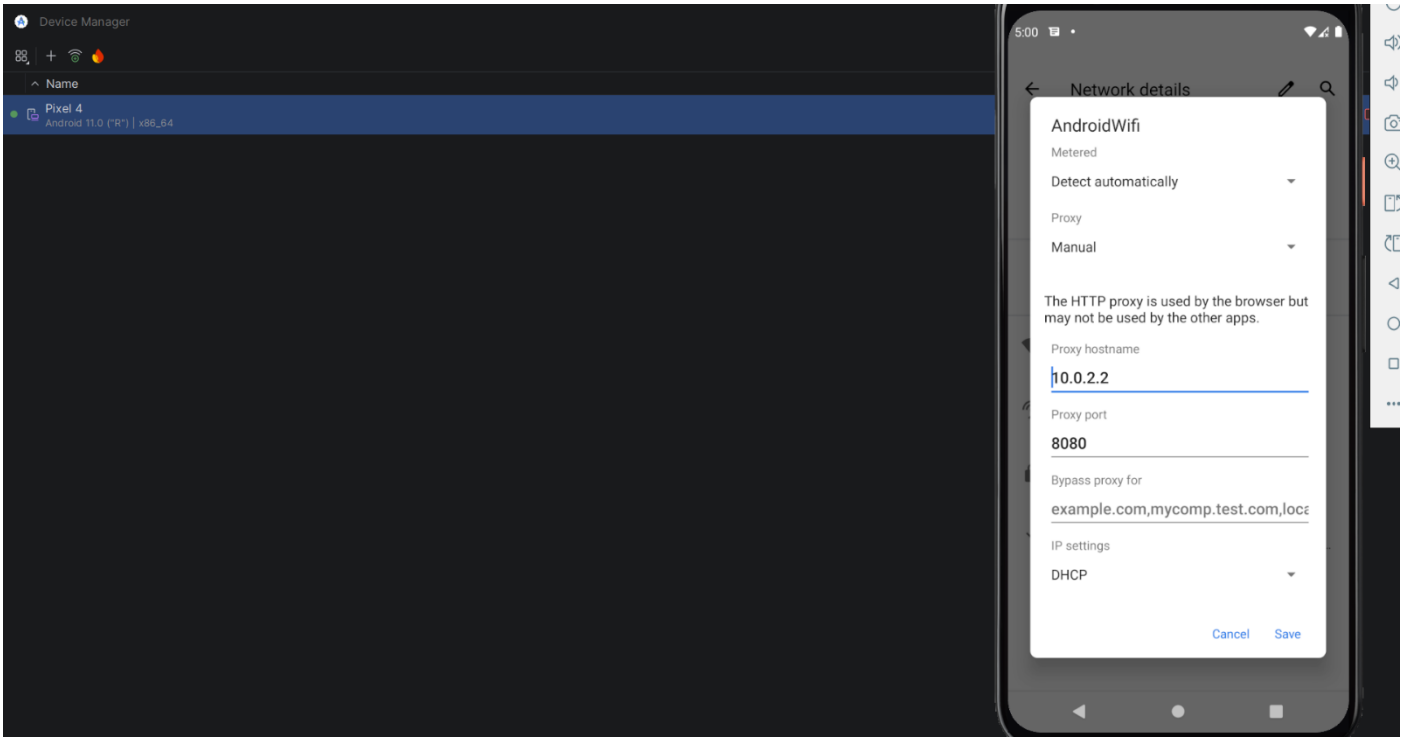

Figure 10: Android Emulator Proxy Configuration

To inspect the application's network traffic, I configured **Burp Suite Community Edition** as an interception proxy. The proxy listener was configured on **port 8080**, and I updated the Android Studio Emulator's Wi-Fi proxy settings so that all HTTP requests generated by the application were routed through Burp Suite.

After completing the proxy configuration, I verified that the setup was working correctly by monitoring the **HTTP History** in Burp Suite while interacting with the application. This allowed me to capture login requests, transaction requests, and password management traffic, providing the evidence used during the dynamic security assessment.

6.3 Login Request Interception

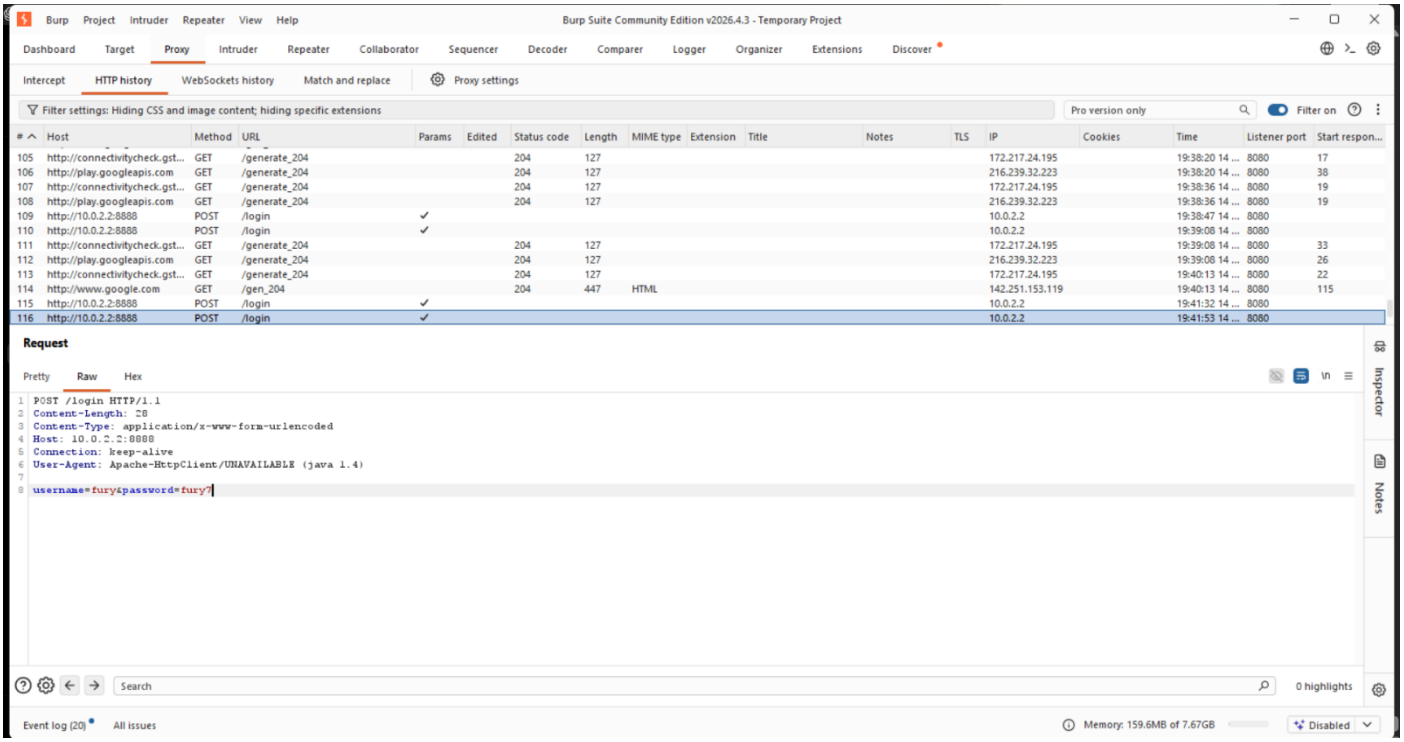

Figure 11: Login Request Captured in Burp Suite

To verify how the application handled user authentication, I entered valid login credentials into **InsecureBankv2** and intercepted the authentication request using **Burp Suite** before it reached the backend server. Capturing the request allowed me to examine the HTTP method, request headers, destination server, and the parameters submitted during the login process.

This step was important because it provided the evidence needed to analyze how the application transmitted authentication data during runtime. After confirming that the request had been successfully intercepted, I examined the captured traffic in more detail to determine whether sensitive information was adequately protected. The detailed security findings from this analysis are discussed in the following section.

6.4 Plaintext Credential Transmission

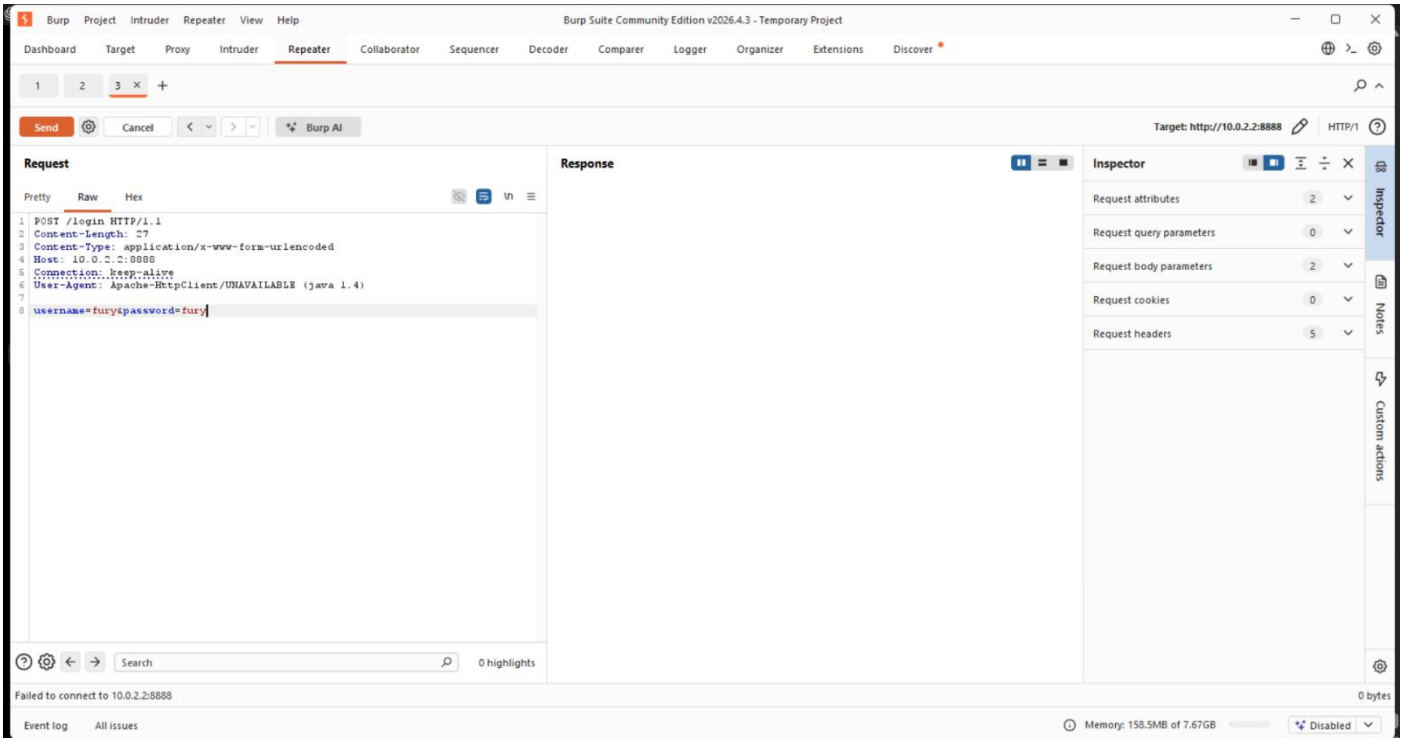

Figure 12: Plaintext Credentials Observed in Login Request

During dynamic testing with Burp Suite, I examined an HTTP POST request sent to the backend server running on **10.0.2.2:8888**. While reviewing the request in **Burp Repeater**, I observed that the credentials were transmitted directly in the request body as **username=fury** and **password=fury**. The values were fully visible in plain text without any encryption or masking.

This was one of the most serious security issues I identified during testing. The application communicated over HTTP instead of HTTPS, meaning the credentials were transmitted without TLS protection. If an attacker were connected to the same network, they could potentially intercept the traffic and read the username and password using common network monitoring or proxy tools. I also observed that no additional protection, such as encrypted request parameters or authentication tokens, was applied before the request was sent to the backend server.

Based on these findings, I concluded that the application's authentication process does not adequately protect user credentials during transmission. A mobile banking application should always transmit authentication requests over HTTPS using TLS to ensure that usernames, passwords, and other sensitive information cannot be intercepted while travelling across the network.

The request examined in Burp Repeater contained the following parameters:

username=fury&password=fury

The request was transmitted over HTTP without any transport layer encryption, allowing the credentials to be viewed directly within Burp Suite. This demonstrated that the application did not protect sensitive authentication data during transmission, making it vulnerable to interception by anyone with access to the same network.

6.5 HTTP Traffic Analysis

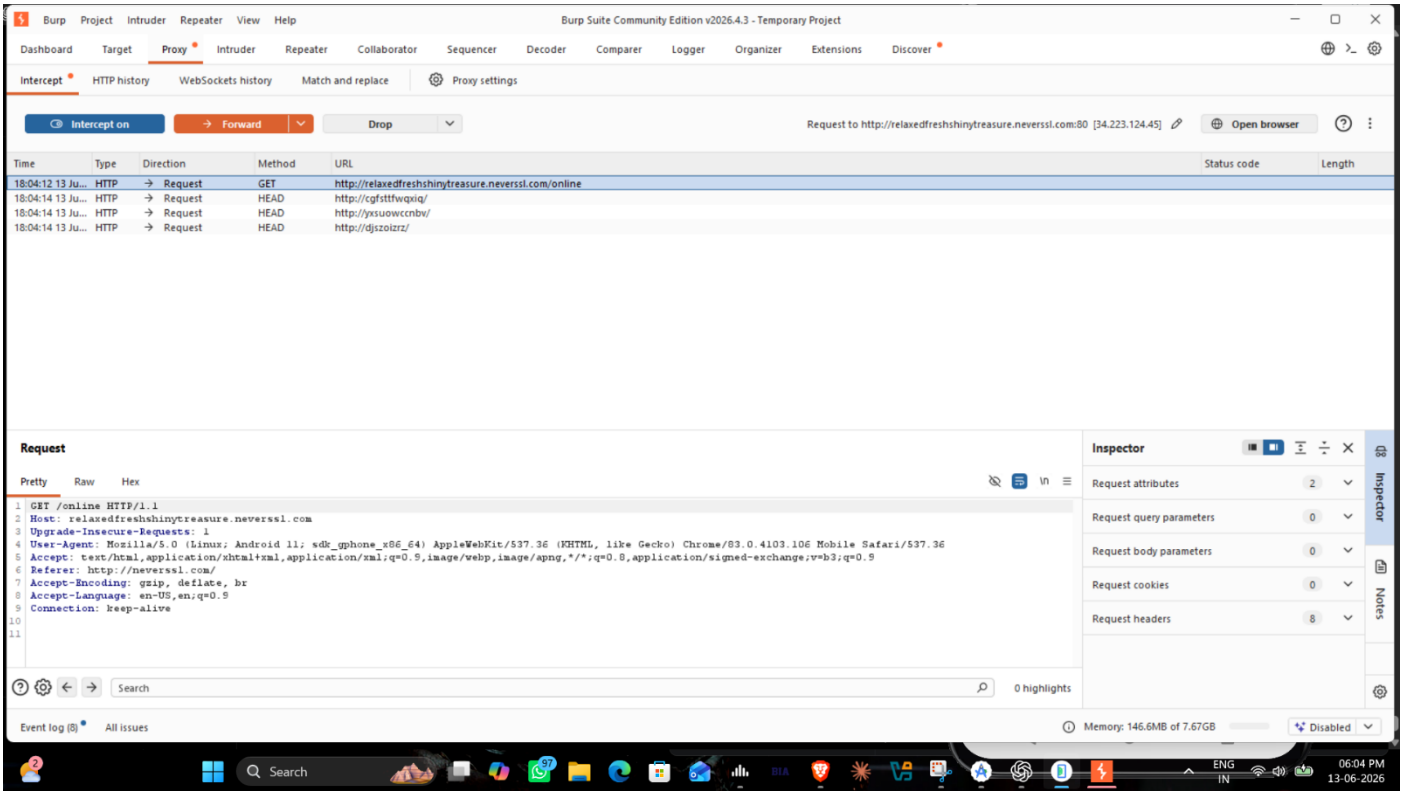

Figure 13: HTTP Traffic Captured During Application Testing

While monitoring the application's network traffic in Burp Suite, I observed multiple HTTP requests and responses exchanged between the Android Studio Emulator and the backend server. Capturing this traffic allowed me to verify how the application communicated during user authentication, account access, transaction processing, and password management.

During my analysis, I noticed that sensitive information was transmitted over **HTTP** without transport layer encryption. This made it possible to inspect the contents of the requests and responses directly within Burp Suite. Reviewing the captured traffic confirmed the insecure communication practices identified earlier and provided practical evidence to support the security findings documented in this report.

6.6 Dynamic Analysis Summary

By the end of the dynamic assessment, I had successfully verified how the **InsecureBankv2** application behaved during runtime. Using **Burp Suite**, I intercepted authentication requests, monitored network communication, and examined how sensitive information was transmitted between the mobile application and the backend server.

The dynamic testing confirmed several important security weaknesses, including plaintext credential transmission over **HTTP**, insecure handling of sensitive information, and weak communication security. Comparing these findings with the MobSF results showed that several vulnerabilities identified during static analysis could also be observed during runtime. This phase of the project helped me understand how combining static and dynamic analysis provides a more complete picture of an application's security posture

7. Authentication Testing

7.1 Login Interface Verification

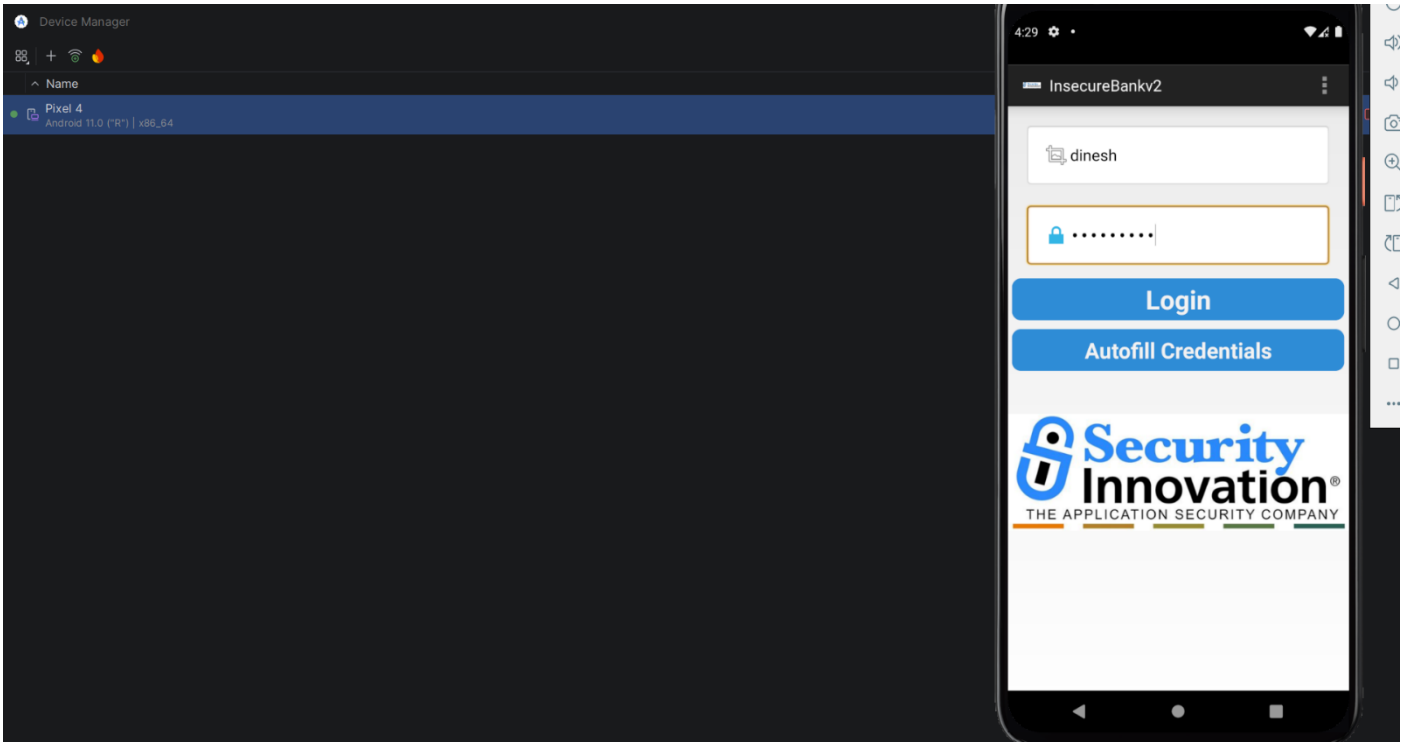

Figure 14: InsecureBankv2 Login Screen

I started the authentication testing by launching the **InsecureBankv2** application in the Android Studio Emulator and reviewing the login interface. The application required a username and password before granting access to the banking features. This screen served as the starting point for testing the authentication workflow and verifying how user credentials were processed.

Before attempting to log in, I confirmed that the backend server was running and that Burp Suite was configured to intercept the application's network traffic. This allowed me to observe both the application's behavior and the corresponding backend communication during the authentication process.

7.2 User Authentication Process

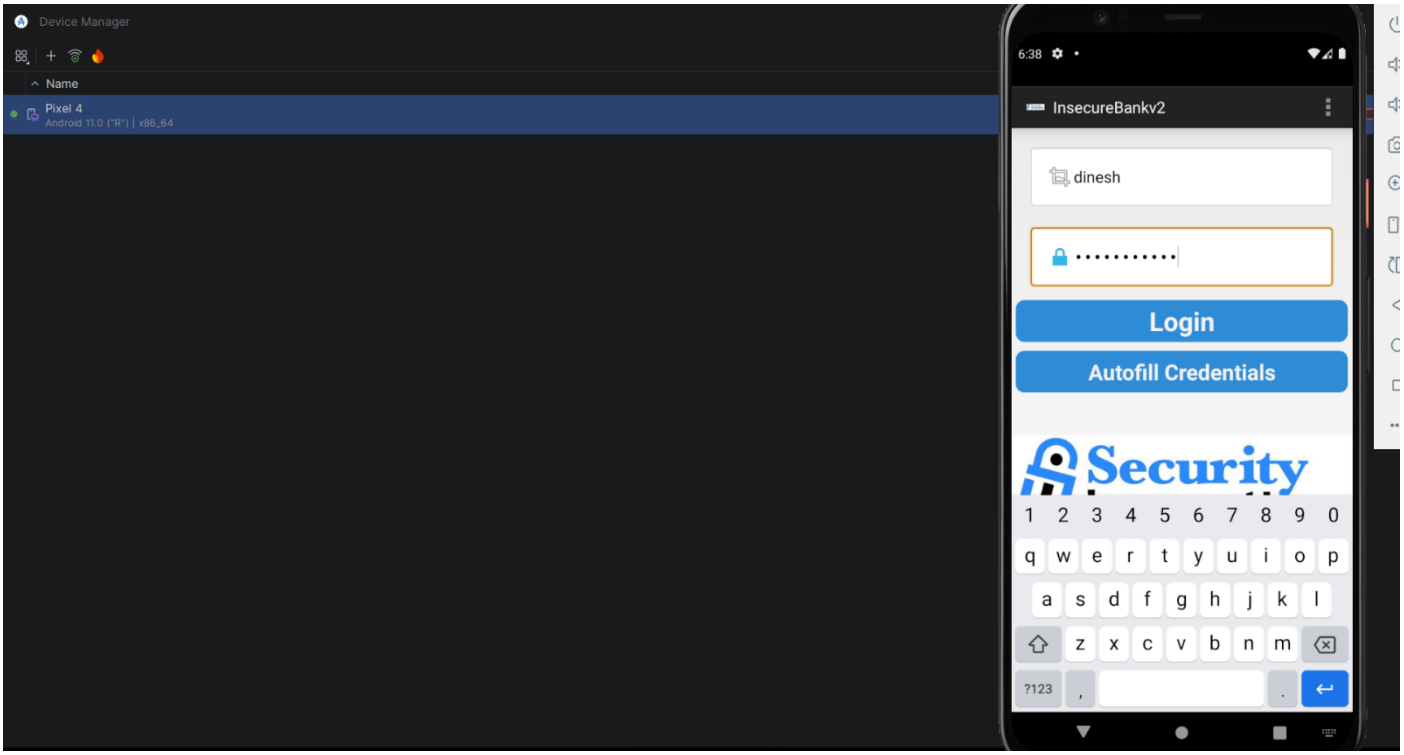

Figure 15: Login Attempt Using Valid Credentials

To verify the authentication process, I logged in using the valid test account provided by the application (**username: dinesh**). After submitting the credentials, the application successfully authenticated the user and communicated with the backend server. I confirmed the successful login by monitoring the backend console, which returned the **"Correct Credentials"** message and allowed access to the application's dashboard.

This test confirmed that the authentication workflow between the mobile application and the backend server was functioning correctly. At the same time, the intercepted network traffic showed that the credentials were transmitted over **HTTP**, which was investigated further during the dynamic security assessment.

7.3 Backend Authentication Validation

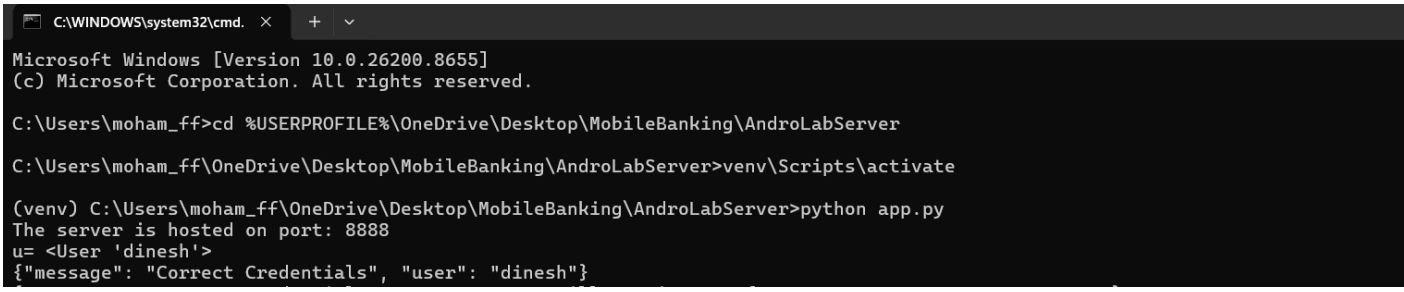

Figure 16: Backend Server Authentication Success Log

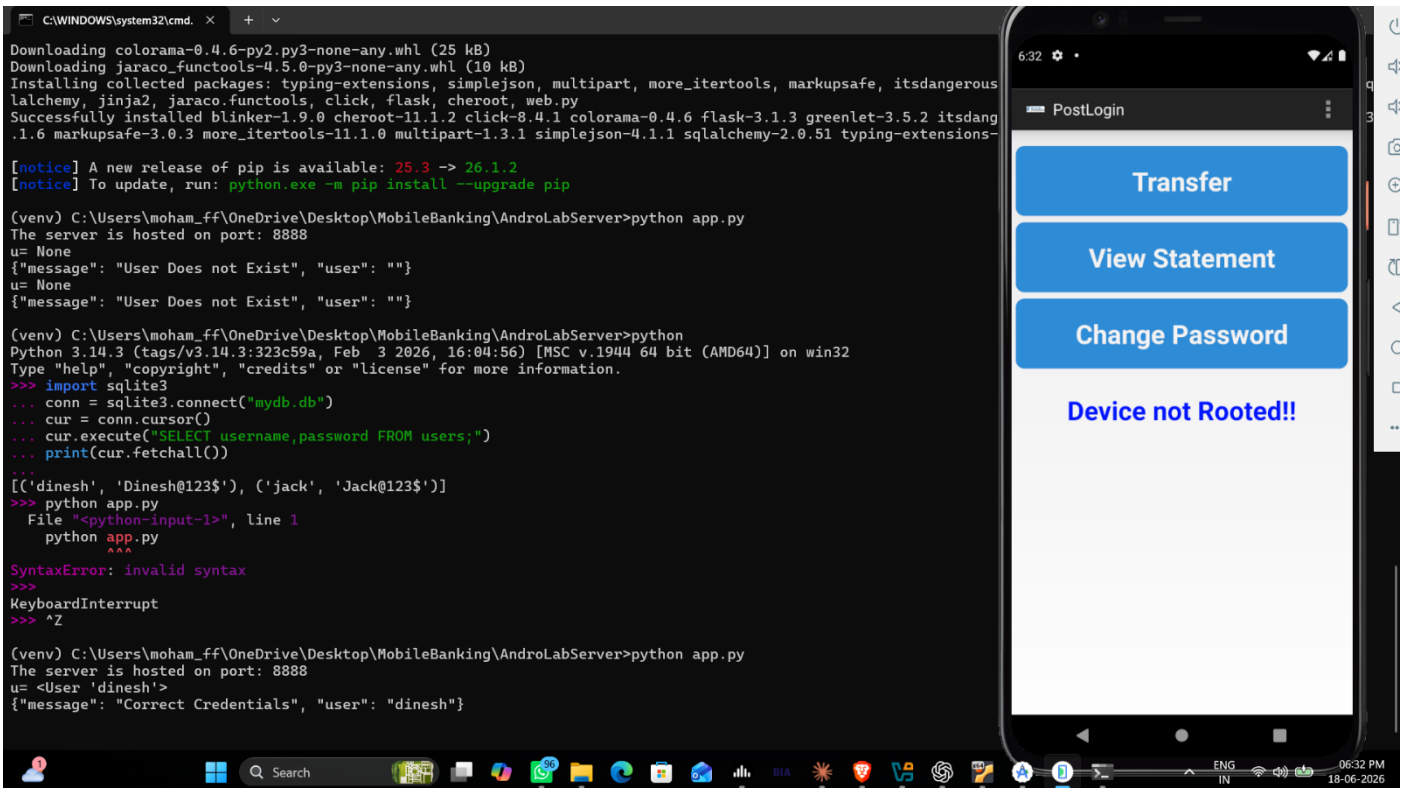

Figure 17: Login Verification Through Backend Server

After submitting the login request, I monitored the backend server console to verify that the authentication request had been processed correctly. The backend successfully validated the supplied credentials and returned the **"Correct Credentials"** message, confirming that the authentication process was handled by the server rather than only by the Android application.

I also verified that the backend returned the user's account information after successful authentication. The account numbers **888888888** and **666666666** were displayed during testing, confirming that the application retrieved user data only after the backend accepted the login request. This demonstrated successful communication between the Android application and the backend server throughout the authentication process.

7.4 Authentication Testing Summary

The authentication testing confirmed that the **InsecureBankv2** application successfully communicated with the backend server during the login process. Using the valid test account, I verified that the backend authenticated the user, returned the **"Correct Credentials"** response, and provided access to the application's banking features.

Although the authentication workflow operated correctly from a functional perspective, my dynamic analysis showed that the login credentials were transmitted over **HTTP** without encryption. This means that, despite successful authentication, sensitive user credentials could still be intercepted during transmission. Based on my testing, secure authentication should always be combined with **HTTPS using TLS** to protect user credentials while they travel between the mobile application and the backend server.

8. Transaction Testing

8.1 Account Dashboard Access

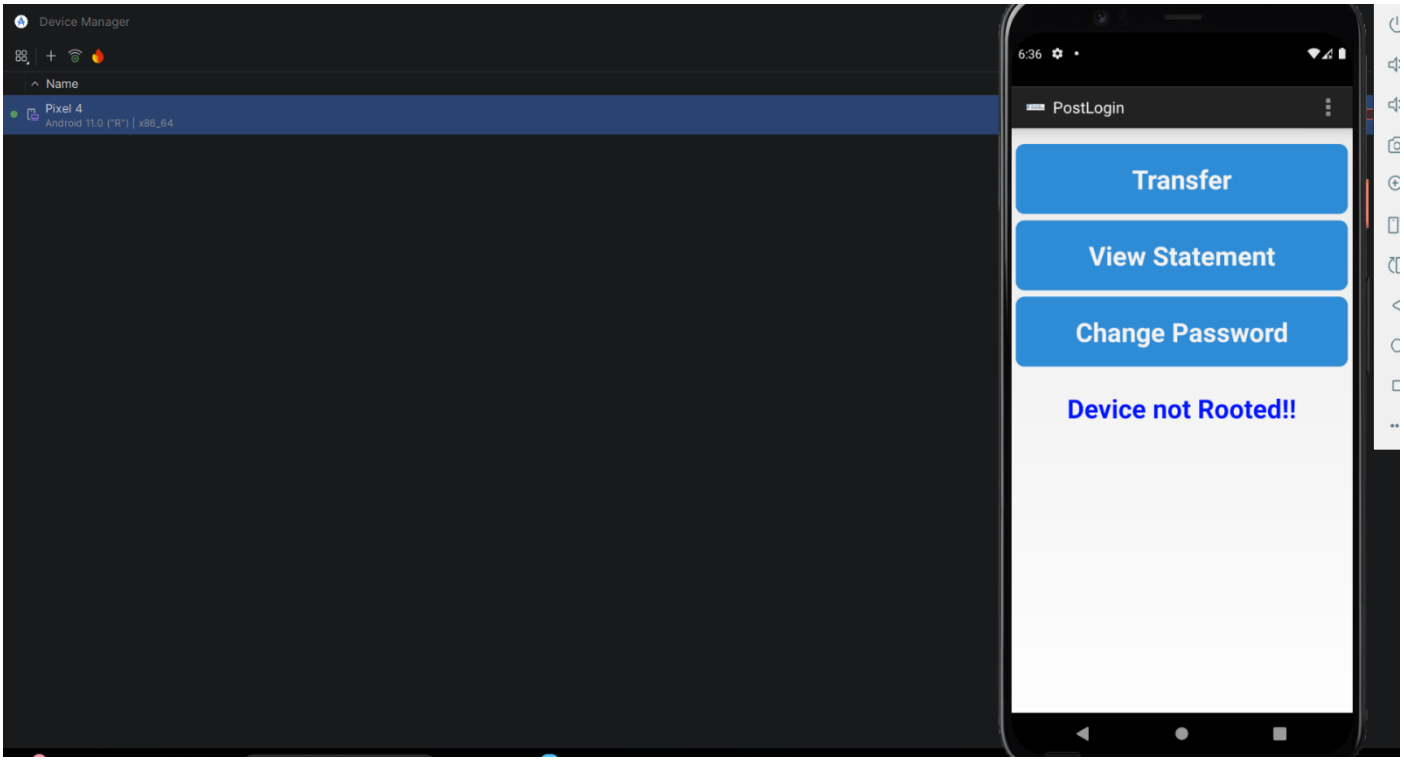

Figure 18: Post-Login Banking Dashboard

After successfully logging in, I was redirected to the application's banking dashboard. The dashboard provided access to features such as fund transfers, account statement viewing, and password management. Reaching this screen confirmed that the authentication process had completed successfully and that the application had established a valid user session.

I also confirmed that the dashboard was displayed only after the backend accepted the login request. This verified that access to the banking functions depended on successful backend authentication rather than only on client-side validation.

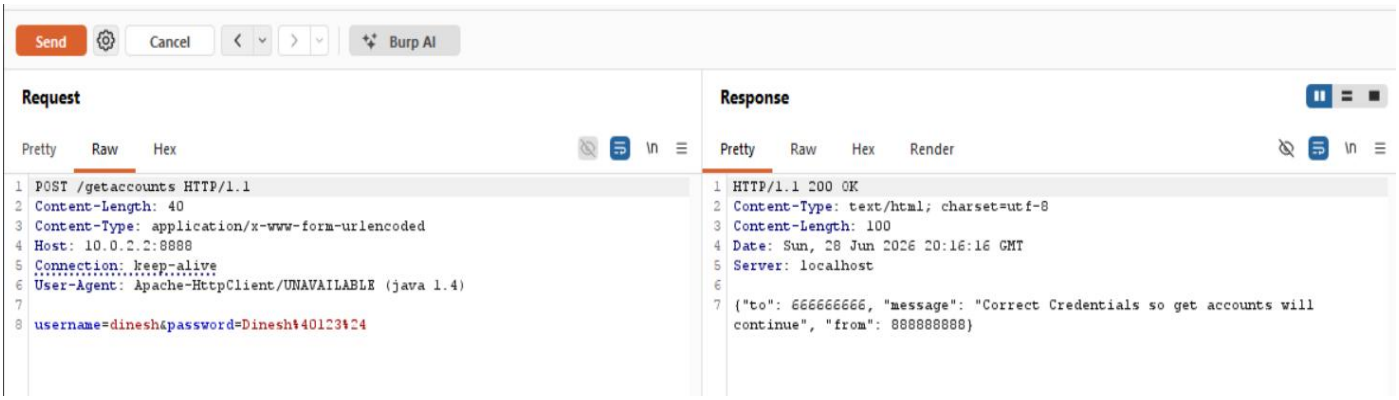

Figure 19: Successful /getaccounts Request and Server Response

After reaching the dashboard, I intercepted the **POST /getaccounts** request using Burp Suite Repeater. The server returned an HTTP 200 OK response containing the account information, confirming that the application successfully retrieved banking data after authentication. This request served as the baseline before performing parameter tampering and authorization testing.

8.2 Funds Transfer Testing

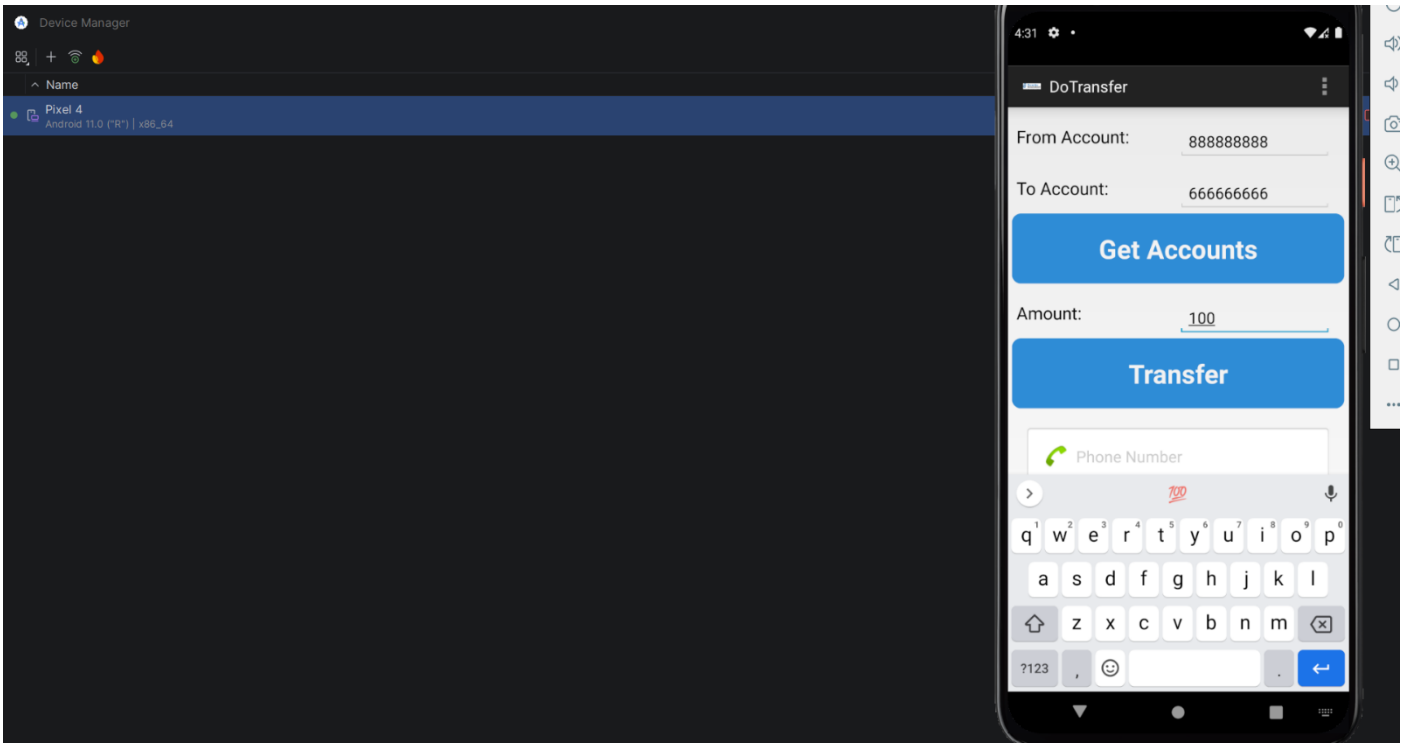

Figure 20: Funds Transfer Interface

To test the transaction functionality, I opened the fund transfer module after logging into the application. The transfer screen displayed the available account numbers and allowed me to enter the required transaction details before submitting the request.

During testing, I observed that the transfer feature was fully accessible after authentication and that the application communicated with the backend server to process the transaction. I also reviewed the application's behavior during this process to understand how transaction requests were handled and whether any additional security controls were applied.

8.3 Transaction Confirmation

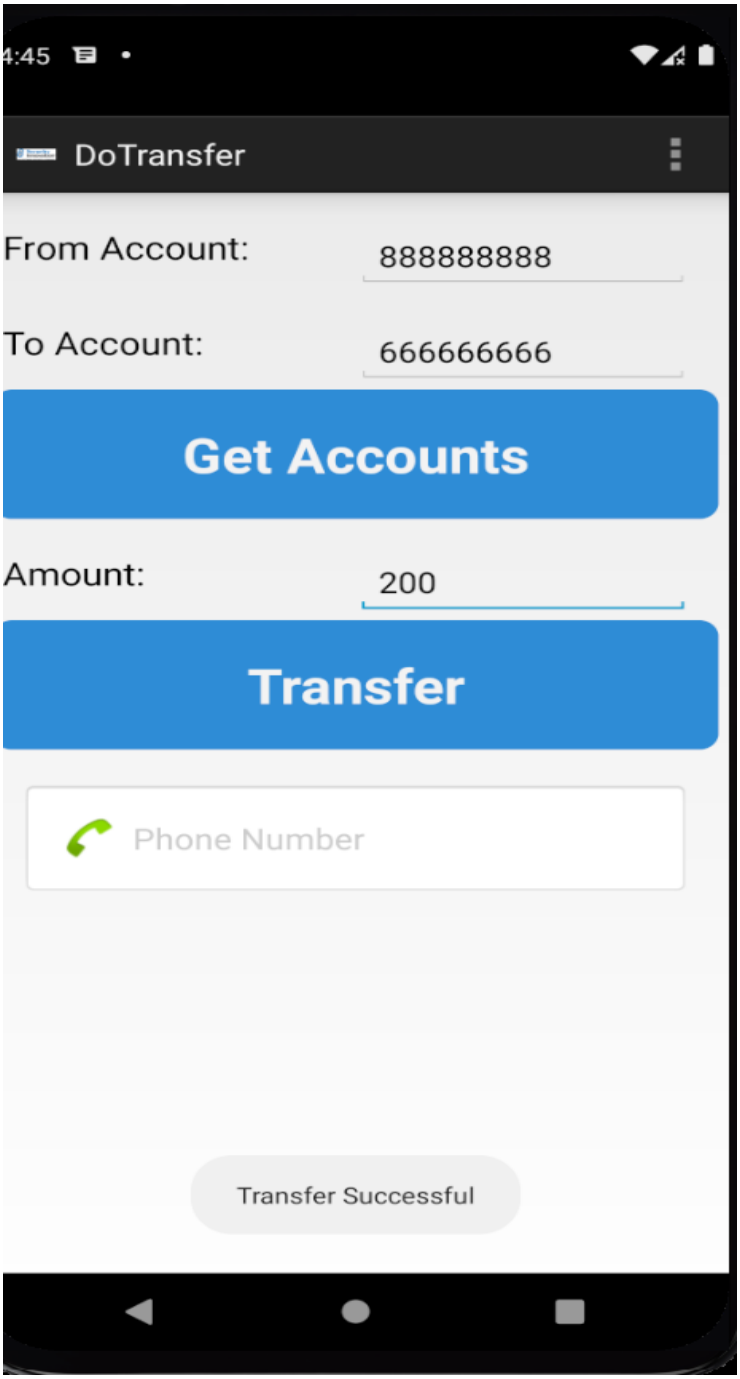

Figure 21: Successful Transaction Confirmation

After submitting the transfer request, the application displayed a successful transaction confirmation message, indicating that the backend had accepted and processed the request. This confirmed that the communication between the Android application and the backend server was functioning correctly during transaction processing.

While the transfer completed successfully from a functional perspective, my testing also showed that the application did not appear to perform additional authorization checks beyond the authenticated session. In a production mobile banking application, transaction processing should include stronger authorization controls and secure communication to reduce the risk of unauthorized or manipulated transactions.

8.4 Parameter Tampering and Authorization Testing

To evaluate the application's server-side authorization and input validation, I intercepted the **POST /dotransfer** request using Burp Suite Repeater. After capturing the request, I manually modified the transaction parameters **(from_acc, to_acc, and amount)** before replaying it to the backend server. The objective of this test was to determine whether the server properly validated transaction parameters and enforced authorization checks.

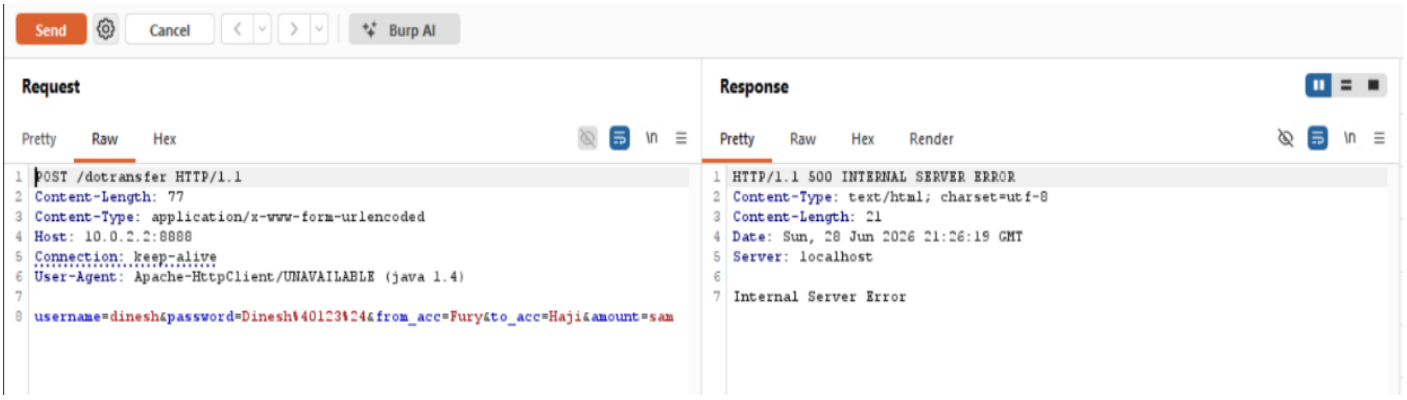

Figure 22: Modified POST /dotransfer request showing manually altered transaction parameters in Burp Suite Repeater

After replaying the modified request, the application returned an **HTTP 500 Internal Server Error** instead of processing the request or returning a controlled validation message. This indicates that the unexpected input caused an unhandled server exception. Although the manipulated transaction was not completed successfully, the server failed to validate the manipulated parameters correctly and instead generated an unhandled exception., demonstrating insufficient server-side input validation and improper exception handling.

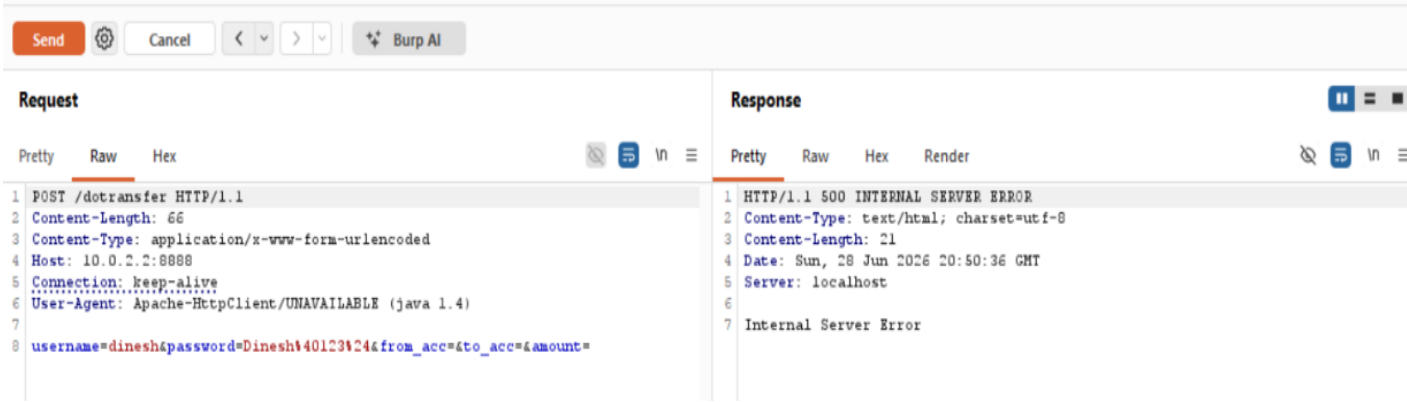

Figure 23: Server response displaying HTTP 500 Internal Server Error after replaying the modified transaction request.

During this assessment, I did not observe evidence that unauthorized account data could be accessed or modified successfully. However, the server's response revealed that malformed transaction parameters were not handled securely. In a production banking application, invalid requests should be rejected gracefully using appropriate input validation and generic error responses without exposing internal application failures.

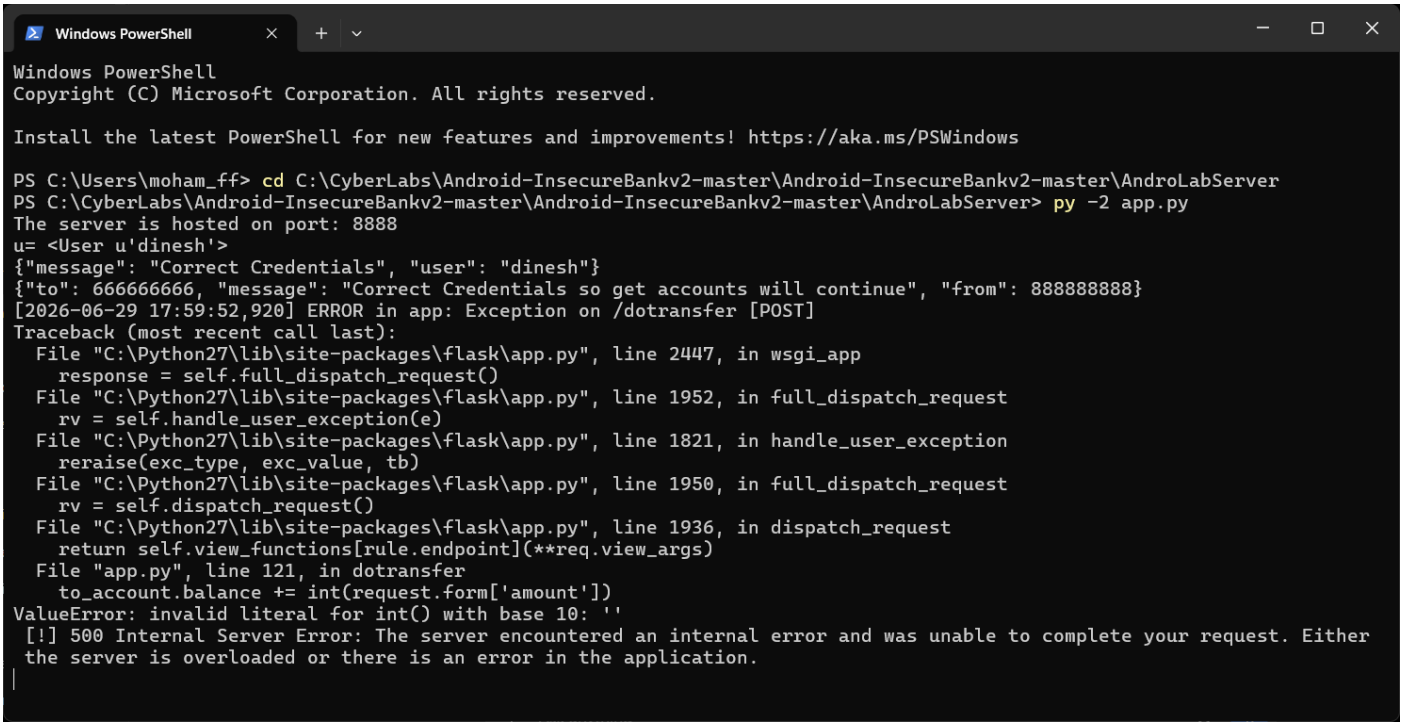

Figure 24: Backend server console displaying the unhandled exception generated after replaying the modified /dotransfer request with manipulated transaction parameters.

Observation: After replaying the modified request, I observed that the application did not process the manipulated transaction successfully. Instead, the server returned an HTTP 500 Internal Server Error because the modified transaction parameters were not properly validated before processing the request. The backend server also generated an unhandled exception, indicating that insufficient input validation and exception handling were implemented. In a secure banking application, invalid or manipulated requests should be rejected gracefully with an appropriate validation message rather than causing an internal server error.

8.5 Account Statement Review

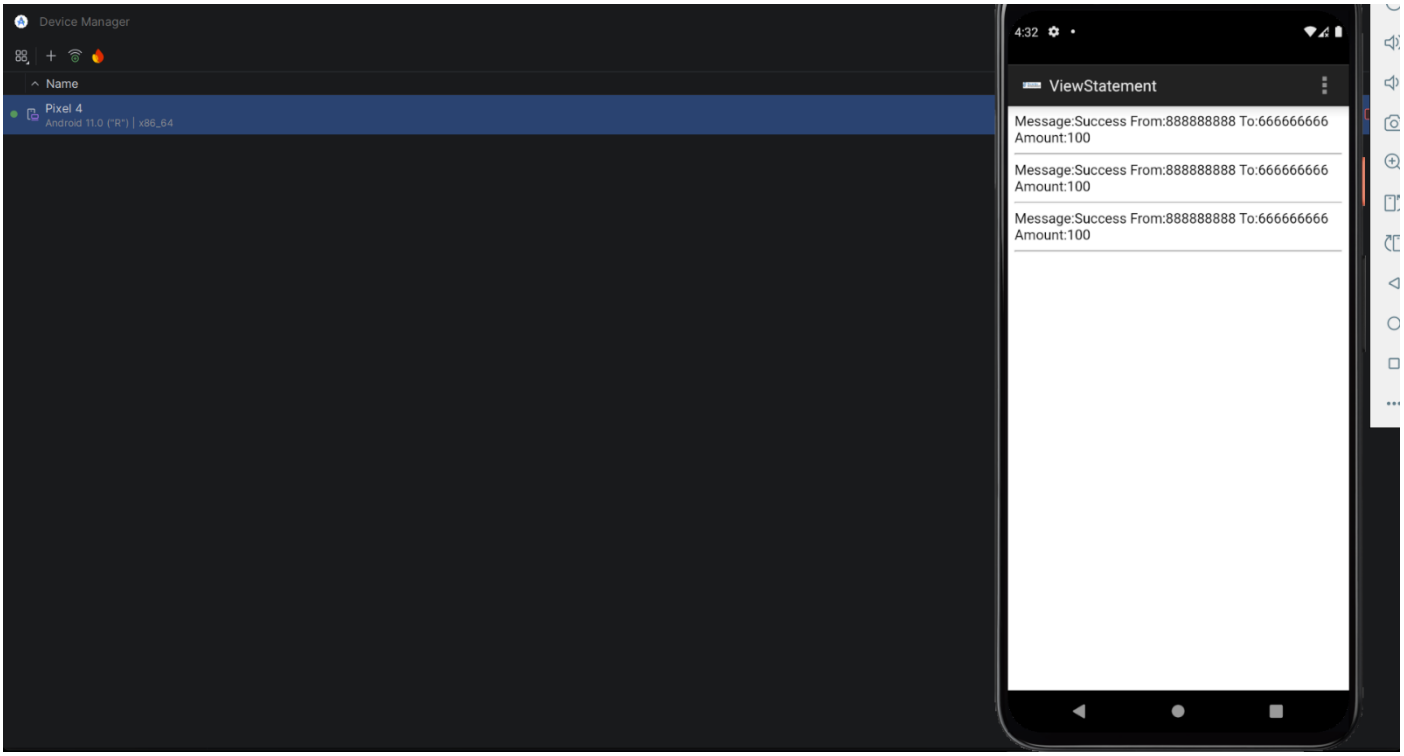

Figure 25: Account Statement Viewing Functionality

After completing the transaction testing, I opened the account statement feature to verify whether transaction records could be accessed successfully. The application displayed the available account information and transaction history without any errors, confirming that the backend was returning the requested data correctly.

While reviewing the statement information, I noticed that the transaction details were displayed in plain text within the application. During my testing, I did not observe any additional authorization checks before the statement information was presented. In a production banking application, access to account statements should be protected through strong authorization controls to ensure that only the authenticated account owner can view sensitive financial information.

8.6 Transaction Testing Summary

The transaction testing confirmed that the **InsecureBankv2** application successfully supported the core banking functions that I tested, including dashboard access, fund transfers, transaction confirmation, and account statement viewing. Each feature communicated successfully with the backend server after user authentication, demonstrating that the application's basic functionality was working as expected.

Although the application completed these operations successfully, my testing identified several security concerns. The application communicated over **HTTP**, transaction and account information were displayed in plain text, and I did not observe additional authorization checks during the transaction workflow. Based on these findings, I believe that stronger authorization controls, secure communication using **HTTPS with TLS**, and improved protection of sensitive banking data are necessary before an application of this type could be considered secure for real-world use.

9. Password Management Testing

9.1 Password Change Interface

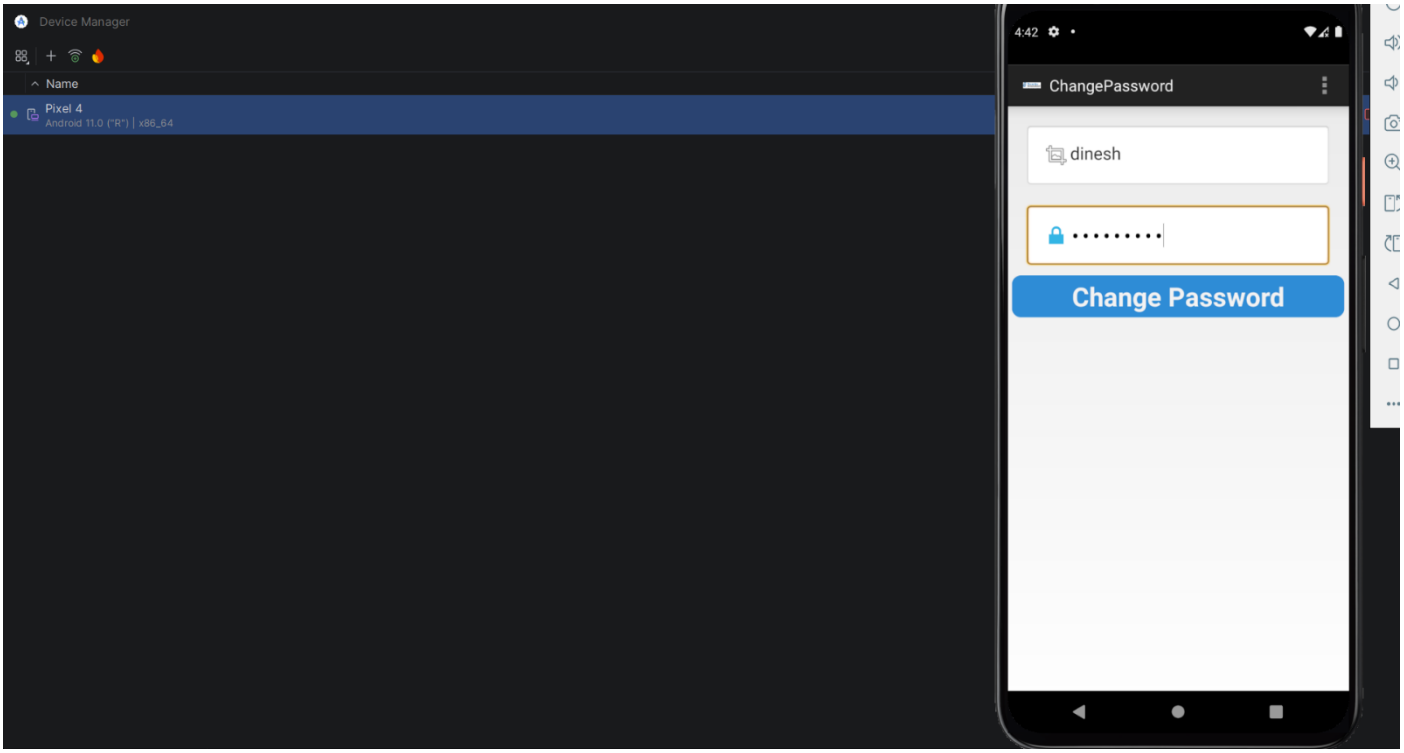

Figure 26: Password Change Interface

After successfully logging into the application, I opened the password management feature to verify whether an authenticated user could update their account password. The application provided a dedicated password change screen where I entered a new password and submitted the request to the backend server.

Before testing the password change process, I confirmed that the backend server was running correctly so that I could monitor how the request was handled. This allowed me to verify both the application's behavior and the backend processing during the password management workflow.

9.2 Password Update Process

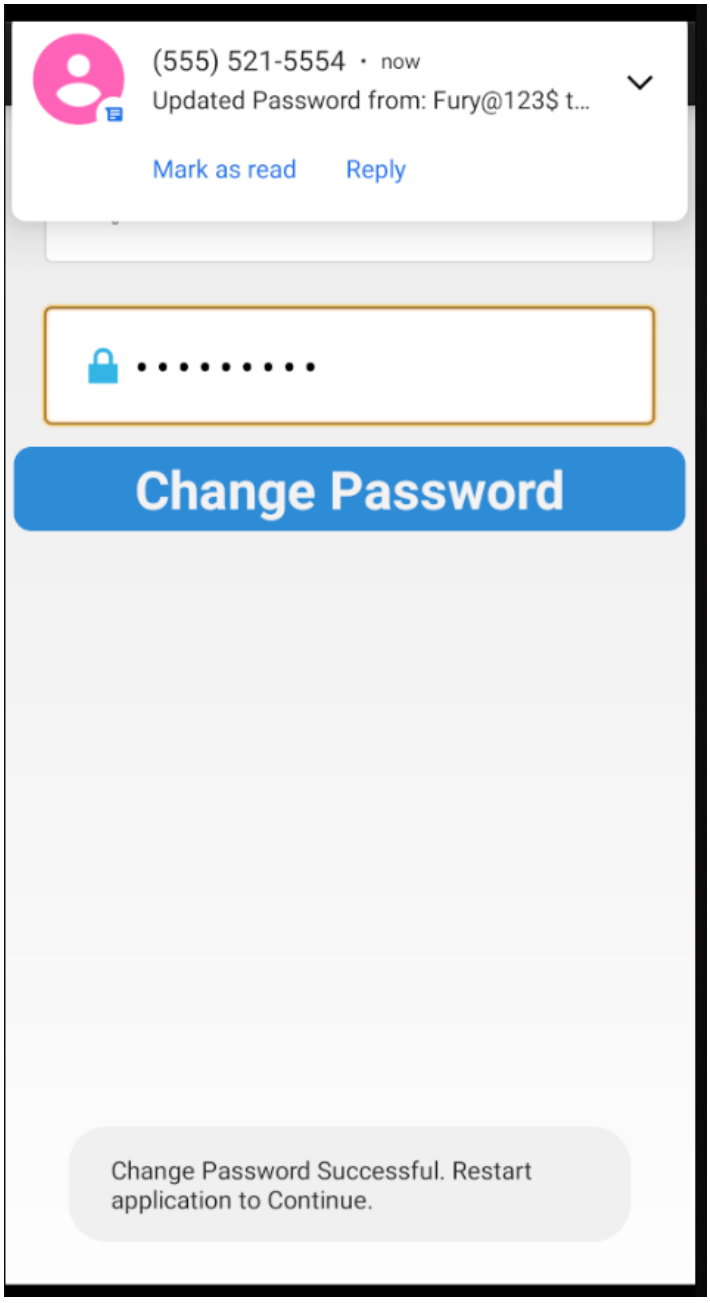

Figure 27: Password Change Confirmation

To verify the password update functionality, I changed the account password from **Dinesh@123$** to **Fury@123$** using the application's password management feature. After submitting the request, the application displayed a **"Change Password Successful"** confirmation message, indicating that the backend had accepted and processed the password update.

To confirm that the change had been applied correctly, I attempted to log in using both the old and the new passwords. The old password was no longer accepted, while the updated password successfully authenticated the user. This confirmed that the password change had been applied successfully and that the backend server had updated the account credentials.

9.3 Backend Password Change Validation

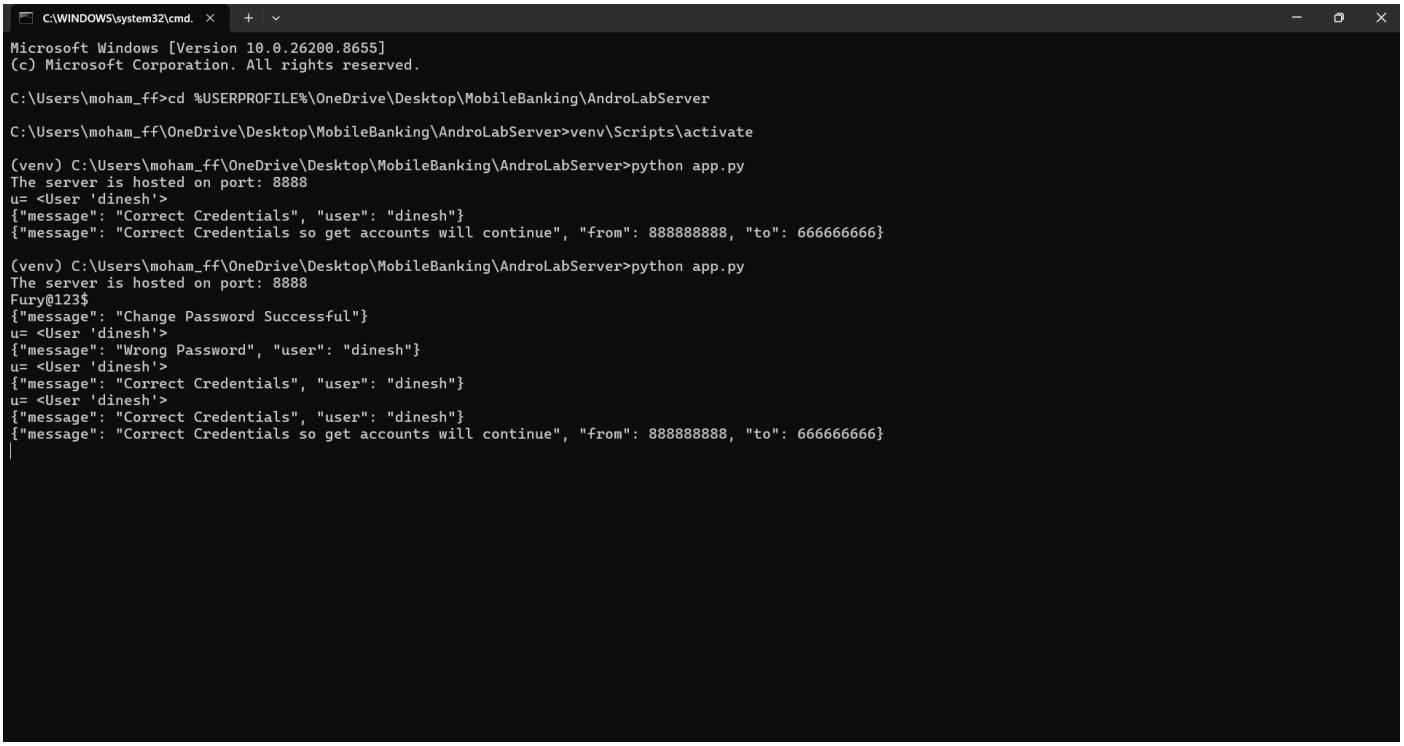

Figure 28: Backend Password Change Log

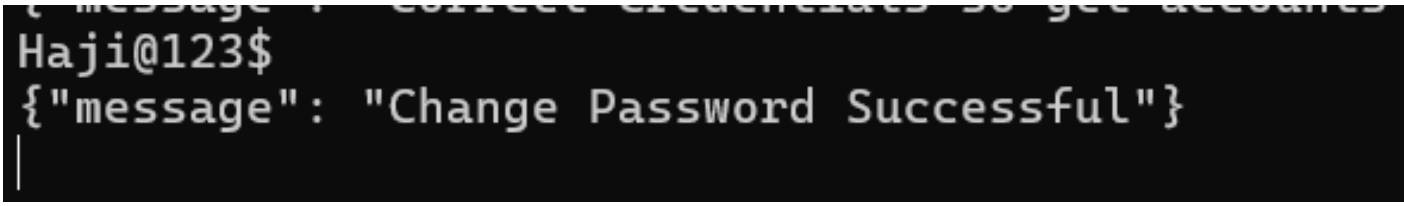

Figure 29: Backend Password Update Verification

After changing the password through the application, I monitored the backend server console to verify that the request had been processed correctly. While reviewing the server logs, I noticed that the backend displayed information related to the password change operation directly in its console output. This immediately caught my attention because backend logs should never expose sensitive authentication data.

Although the password change itself was successful, the way the backend handled logging raised a security concern. In a production banking environment, passwords and other sensitive credentials should never be written to application logs in plain text because anyone with access to those logs could potentially view confidential user information. During my testing, this demonstrated that protecting user credentials is not only important during network transmission but also after the server receives and processes the request.

9.4 Password Management Testing Summary

The password management testing confirmed that the **InsecureBankv2** application successfully allowed authenticated users to update their account credentials. During testing, I changed the account password from **Dinesh@123$** to **Fury@123$**, verified that the old password was no longer accepted, and confirmed that the new password successfully authenticated the user. These results showed that the password update process was functioning correctly from a functional perspective.

Although the password change workflow operated successfully, my testing also identified an important security concern. While reviewing the backend server during password processing, I found that sensitive credential information was exposed in plain text, which is discussed in **Section 9.5**. Based on these findings, password management features should not only support secure password updates but also protect sensitive information throughout processing, storage, logging, and network communication.

9.5 Critical Finding – Plaintext Password Logging

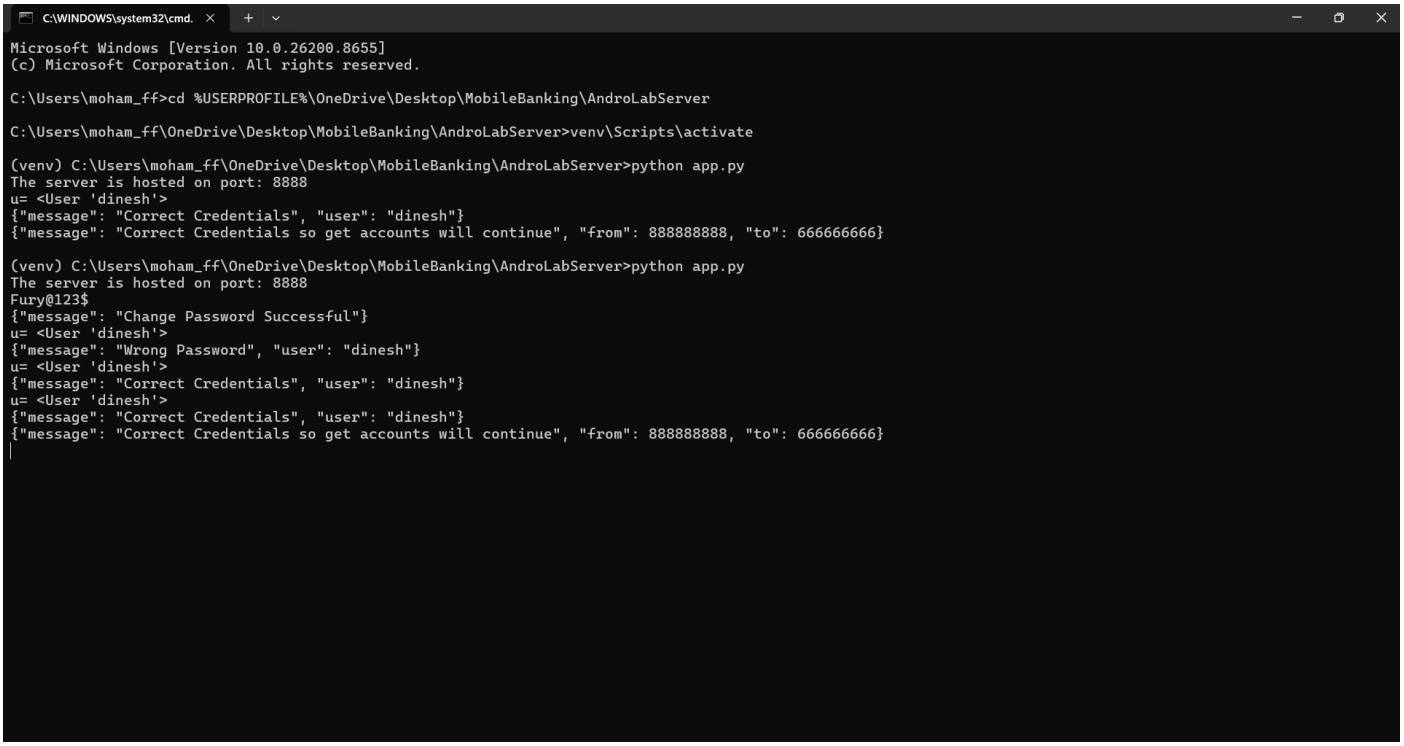

Figure 30: Backend Console Showing Plaintext Password Logging During Password Change

During password management testing, I discovered a separate security issue that was not related to network communication. After submitting a password change request through the application, I monitored the backend server console to verify that the request had been processed successfully. While reviewing the backend console, I noticed that the updated password (Fury@123$) was displayed directly in plain text before the server returned the **"Change Password Successful"** response. The account being updated remained **dinesh**, but the new password value was exposed in the server console. The console displayed the password before the successful response was returned to the application, demonstrating that the backend itself

was responsible for exposing the sensitive credential rather than the mobile application interface. While reviewing the backend console, I clearly saw the newly submitted password displayed in plain text before the application returned the "Change Password Successful" response. This confirmed that the backend was logging sensitive authentication data without masking or hashing the password, creating an unnecessary exposure of user credentials.

Unlike the plaintext credential transmission identified during dynamic analysis, this issue occurs **after the server receives the request**. Even if HTTPS were implemented, storing or displaying passwords in server logs or console output would still expose sensitive user information. Anyone with access to the backend console or server logs, including system administrators or an attacker who compromises the server, could view user passwords without any additional effort.

Based on my testing, I consider this a **High Severity** vulnerability because passwords should never be written to application logs or console output in plain text. Backend applications should avoid recording sensitive authentication data and should only log operational information required for troubleshooting and auditing. This finding demonstrated to me that secure password handling must be maintained throughout the entire authentication process, not only while data is being transmitted over the network.

#	**Vulnerability**	**Severity**	**Discovery Method**
1	Plaintext Credential Transmission	Critical	Burp Suite
2	Plaintext Password Logging	High	Backend Testing
3	Hardcoded Secrets (23)	High	MobSF
4	Dangerous Permissions	Medium	MobSF
5	Exported Activities	Medium	MobSF
6	Janus Vulnerability	Medium	MobSF
7	Tracker Detection	Low	MobSF
8	Manifest Security Weaknesses	Low	MobSF
9	Insufficient Input Validation (HTTP 500 on parameter tampering)	Medium	Parameter Tampering Test

10. Security Recommendations

10.1 Secure Communication

Based on my dynamic testing, the most critical issue was the transmission of login credentials over **HTTP** without encryption. To prevent this, the application should enforce **HTTPS using TLS 1.3** for all communication between the mobile application and the backend server. Certificate validation should also be implemented correctly to prevent man-in-the-middle attacks. Sensitive information such as usernames, passwords, account details, and transaction data should never be transmitted over unencrypted communication channels.

10.2 Authentication Improvements

During authentication testing, I confirmed that the login process functioned correctly, but stronger authentication controls are still required to better protect user accounts. Implementing **Multi-Factor Authentication (MFA)** would add an additional layer of security by requiring users to verify their identity using more than one authentication factor. Strong password policies, account lockout mechanisms, and rate limiting should also be implemented to reduce the risk of brute-force and credential-based attacks.

10.3 Secure Data Storage

My testing showed that sensitive information was exposed during both network communication and backend processing. Mobile banking applications should avoid storing or displaying credentials, authentication tokens, or other confidential information in plain text. Sensitive data stored locally should be protected using strong encryption algorithms such as **AES-256**, while encryption keys should be managed securely and never embedded directly within the application.

10.4 Application Hardening

The MobSF assessment identified several security weaknesses, including dangerous permissions, exported activities, **23 possible hardcoded secrets**, tracker components, manifest configuration issues, and the **Janus vulnerability**. Based on these findings, the application should be hardened by removing unnecessary permissions, restricting exported components, eliminating hardcoded sensitive information, adopting modern APK signing schemes, and implementing secure coding practices. Additional protection mechanisms such as code obfuscation, root detection, and runtime integrity checks would further reduce the application's attack surface.

10.5 Security Monitoring

During this project, I found that reviewing backend logs and monitoring application behavior helped identify security issues that were not immediately visible through static analysis alone. Production mobile banking applications should implement continuous security monitoring to detect suspicious login attempts, abnormal transaction activity, unauthorized access, and application misuse. Logging should record operational events without exposing sensitive information such as passwords or authentication credentials.

11. Compliance Review

The findings from this project were compared with commonly accepted security practices used in the financial services industry. Based on my testing, I identified several security weaknesses that would need to be addressed before an application like **InsecureBankv2** could be considered suitable for a production banking environment. This included plaintext credential transmission, plaintext password logging in the backend, dangerous permissions, exported activities, hardcoded secrets, and other application security weaknesses identified during the assessment.

Although **InsecureBankv2** is intentionally vulnerable for security training, performing this assessment helped me understand why mobile banking applications are expected to implement secure communication, strong authentication, proper access controls, secure coding practices, and effective protection of sensitive customer information to meet industry security expectations.

Compliance Assessment Summary

Standard / Framework	**Security Requirement**	**Assessment Result**
PCI DSS	Encrypt sensitive data during transmission	**Not Compliant** (HTTP used instead of HTTPS)
PCI DSS	Do not log sensitive authentication data	**Not Compliant** (Password logged in plaintext)
OWASP MASVS / MSTG	Secure authentication and credential protection	**Partially Compliant** (Authentication works, but credentials are transmitted without encryption)
OWASP MASVS / MSTG	Secure application configuration	**Not Compliant** (Dangerous permissions, exported activities, manifest weaknesses)
NIST Cybersecurity Framework	Identify and assess security risks	**Assessment Performed** (Static and dynamic security testing completed)

The compliance review shows that the application does not satisfy several important security requirements expected for a production mobile banking application. Although InsecureBankv2 is intentionally designed as a vulnerable training application, comparing the identified findings against recognized security frameworks helped me understand how security weaknesses affect regulatory compliance and industry best practices.

12. Monitoring & Incident Response

One of the most important lessons from this project was that security issues can be identified not only through application testing but also by reviewing backend activity. During password management testing, monitoring the backend console helped me discover that sensitive credential information was being displayed in plain text, demonstrating the value of effective security monitoring.

Production mobile banking applications should continuously monitor authentication events, transaction activity, application logs, and network communications to detect suspicious behavior and potential security incidents. At the same time, logging mechanisms should never expose sensitive information such as passwords or authentication credentials. A well-defined incident response process should include procedures for identifying, containing, investigating, recovering from, and documenting security incidents so that organizations can respond quickly and reduce potential impact.

13. User Awareness Material

Although secure application development is essential, users also play an important role in protecting their banking accounts. Based on the security weaknesses identified during this project, users should always verify that they are using secure banking applications that communicate over **HTTPS**, avoid sharing their credentials with anyone, and enable **Multi-Factor Authentication (MFA)** whenever it is available.

Users should create strong and unique passwords, install banking applications only from trusted sources such as official app stores, keep their devices updated with the latest security patches, and remain alert to phishing attacks and other social engineering attempts. If any suspicious account activity is detected, it should be reported to the bank immediately so that appropriate action can be taken.

14. Challenges Faced

This project involved considerably more practical troubleshooting than I initially expected. Before I could begin the actual security assessment, I had to resolve several environment and compatibility issues that prevented the testing tools from working correctly.

The first major challenge was configuring **Genymotion** to communicate with Burp Suite. I spent almost **three days** troubleshooting proxy routing problems caused by VirtualBox networking. Even though the proxy settings appeared correct, the emulator traffic was not reaching Burp Suite consistently. After trying multiple network configurations without success, I decided to switch to the **Android Studio Emulator**, which immediately resolved the proxy routing issue and allowed me to intercept application traffic successfully.

Another significant challenge involved the **AndroLabServer** backend application. The server was originally written for **Python 2**, but my environment used **Python 3**, which caused multiple compatibility errors. I migrated the project to Python 3 by updating print statements, correcting indentation problems, and fixing several deprecated syntax issues. During testing, I also encountered an SQLAlchemy compatibility error caused by the deprecated convert_unicode=True parameter. Removing this parameter resolved the database initialization problem and allowed the backend server to start successfully.

I also experienced connectivity problems between the Android emulator running in Kali Linux and the backend server. After several unsuccessful attempts to establish reliable communication, I moved the backend server to the Windows host machine while continuing to use Kali Linux for security testing. This provided stable communication between the emulator, Burp Suite, and the backend application.

Finally, I experienced installation issues while setting up **MobSF**. The application failed because of Poetry dependency conflicts and missing Gunicorn components. I reinstalled the required dependencies and verified that MobSF was functioning correctly before performing the static analysis.

Although these issues delayed the testing process, resolving them improved my understanding of Android application testing environments, backend compatibility, and security assessment workflows.

15. Lessons Learned

Working on this project taught me that performing a practical mobile application security assessment involves much more than simply running security tools. I learned that setting up a stable testing environment is often the most time-consuming part of the assessment. Solving the Genymotion proxy issue showed me that understanding networking and virtualization is just as important as understanding the security tools themselves.

Migrating the AndroLabServer backend from Python 2 to Python 3 helped me better understand application compatibility issues. I learned how deprecated syntax and library changes can prevent older applications from functioning correctly, and I became more comfortable troubleshooting Python errors, dependency problems, and SQLAlchemy compatibility issues.

Using MobSF and Burp Suite together helped me understand the difference between static and dynamic analysis. MobSF identified weaknesses inside the APK such as dangerous permissions, hardcoded secrets, exported activities, and manifest issues, while Burp Suite allowed me to observe how the application behaved during execution and how sensitive information was transmitted over the network.

One of the biggest lessons I learned was that not every serious vulnerability is visible in network traffic. During password management testing, I discovered that the backend server printed the updated password directly in the console. This showed me that sensitive information can still be exposed even if network communication is protected. Secure software development requires protecting sensitive data throughout the entire application, including backend logging and server-side processing.

Overall, this project improved my practical understanding of Android application security, backend communication, vulnerability assessment, and troubleshooting. More importantly, it showed me how real security assessments involve investigating problems, validating findings with evidence, and understanding the impact of each vulnerability rather than simply relying on automated scanning tools.

16. Conclusion

Through this project, I performed a practical security assessment of the **InsecureBankv2** mobile banking application using **MobSF**, **Burp Suite**, the **Android Studio Emulator**, and the application's backend server. Combining static and dynamic analysis allowed me to identify several important security weaknesses, including dangerous permissions, **23 hardcoded secrets**, exported activities, manifest security issues, the **Janus vulnerability**, plaintext credential transmission over **HTTP**, and plaintext password logging in the backend server.

In addition to identifying vulnerabilities, I verified the application's authentication process, transaction workflow, account statement access, and password management functionality through practical testing. The challenges I encountered while configuring the testing environment, migrating the backend from Python 2 to Python 3, and troubleshooting connectivity issues gave me valuable hands-on experience that extended beyond simply using automated security tools.

Overall, this project strengthened my understanding of mobile application security testing and demonstrated how static analysis, dynamic analysis, backend validation, and secure coding practices work together to protect sensitive banking applications. The findings and recommendations documented in this report can help developers understand common security weaknesses and implement stronger security controls before deploying mobile banking applications in real-world environments.

17. References

- OWASP Mobile Security Testing Guide (MSTG)
- OWASP Mobile Top 10
- MobSF Documentation
- Burp Suite Documentation
- Android Developers Security Documentation
- NIST Cybersecurity Framework
- InsecureBankv2 GitHub Repository
- Oracle VirtualBox Documentation
- Android Studio Documentation