Mobile Application Security Assessment Report

Training Module:

Web/Mobile Application Security

Practical Lab Exercises and Real-world Scenarios

Prepared By:

El-Fehri Samir
elfihrisami98@gmail.com

Training Date:

October 15-30, 2025
seedsecuritylabs.org

CONTENTS

CHAPTER 1

REVERSE ENGINEERING LAB

1.1 Lab Setup and ADB Operations

Task 1: Connect Device with Kali and Enter Shell

Description

Establish connection between Kali Linux and Genymotion Android emulator using ADB and access the device shell.

Solution

```
adb connect 192.168.58.113:5555
adb shell
```

Results

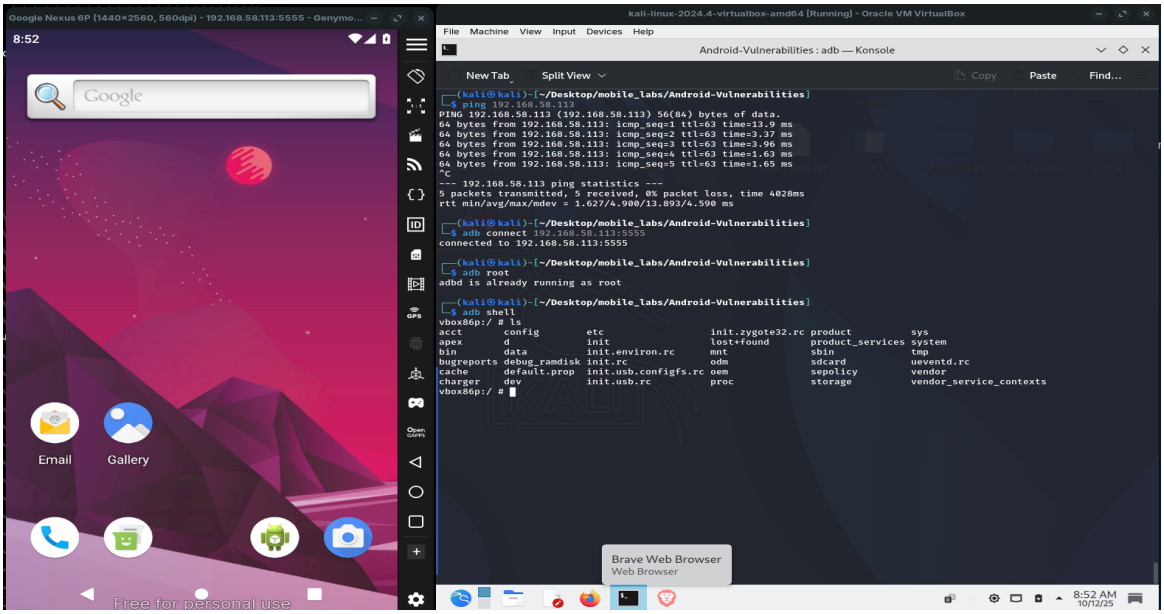


Figure 1.1: Connecting to device and accessing ADB shell

Task 2: Install Diva Application in Device

Description

Download and install the Diva (Damn Insecure and Vulnerable App) application on the Android emulator.

Solution

```
adb install diva-beta.apk
```

Results

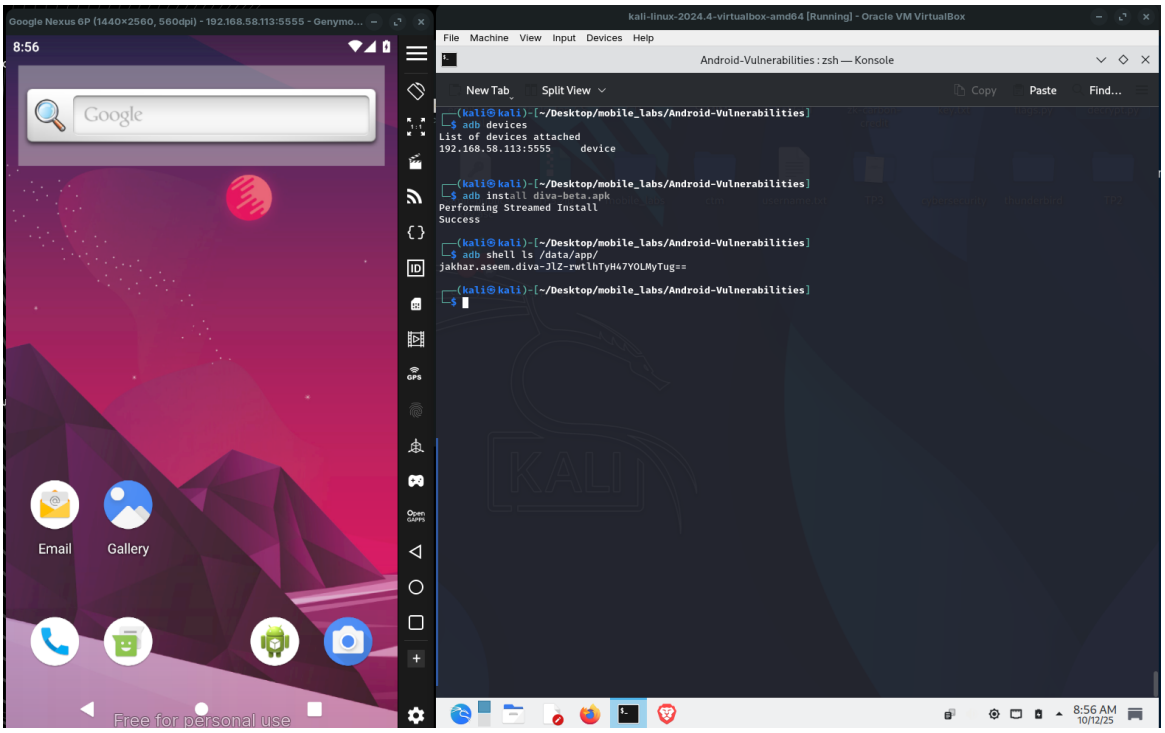


Figure 1.2: Installing Diva application via ADB

1.2 Android Process Analysis

Task 3: List Processes

Description

Display all running processes on the Android device to understand the system's process hierarchy.

Solution

```
adb shell
ps -ef
```

Results

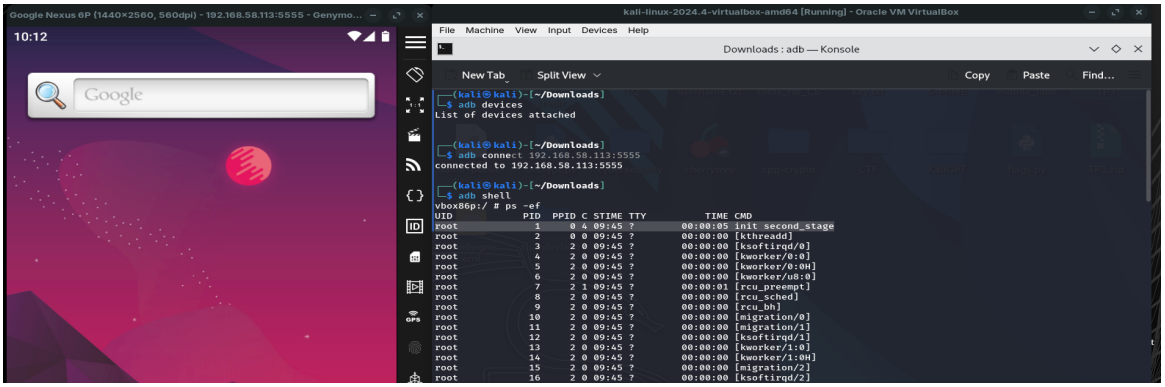


Figure 1.3: Listing all running processes

Task 4: Find Zygote PID

Description

Identify the Zygote process and determine its Process ID (PID) and Parent Process ID (PPID).

Solution

```
1 ps -ef | grep "zygote"
```

Question 1: What is his PID? What is the PID of his parent process?

The zygote process ID is 410, and the command `ps | grep "zygote"` shows that the parent ID is 1, that means the parent process of zygote is the init process.

Results

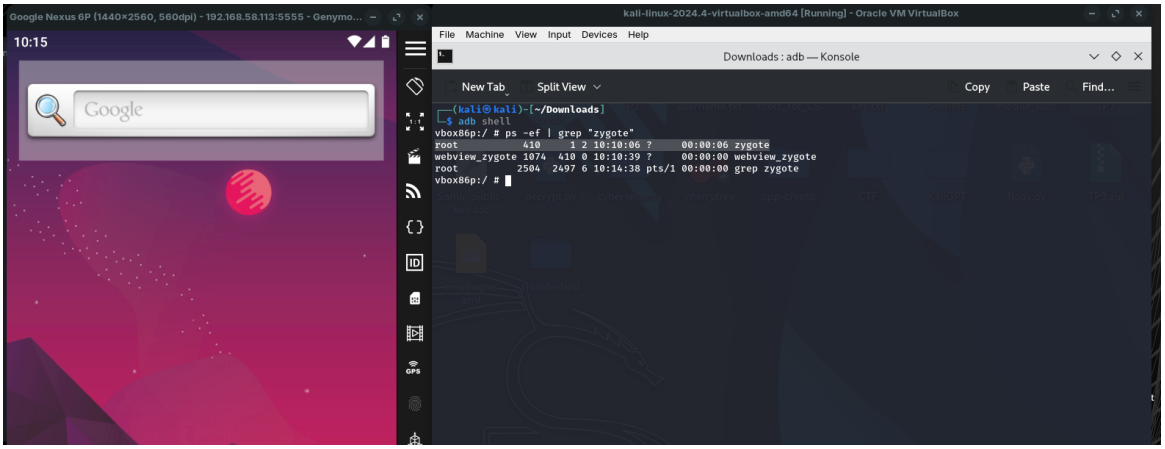


Figure 1.4: Finding Zygote process and its parent

1.3 Application Reverse Engineering

Task 5: Unzip Diva and List Results

Description

Extract the contents of the Diva APK file to analyze its internal structure.

Solution

```
unzip -d demo diva-beta.apk
ls -la demo/
```

Results

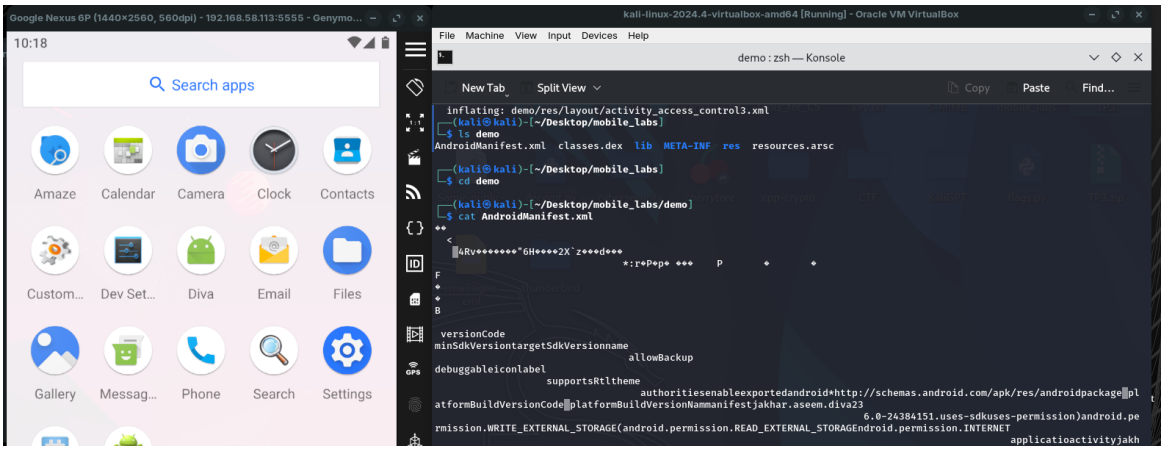


Figure 1.5: Unzipping APK and viewing contents

Task 6: Decompile Diva with Apktool

Description

Use Apktool to decompile the Diva application and examine the AndroidManifest.xml file.

Solution

```
apktool d diva-beta.apk
cd diva-beta
cat AndroidManifest.xml
```

Results

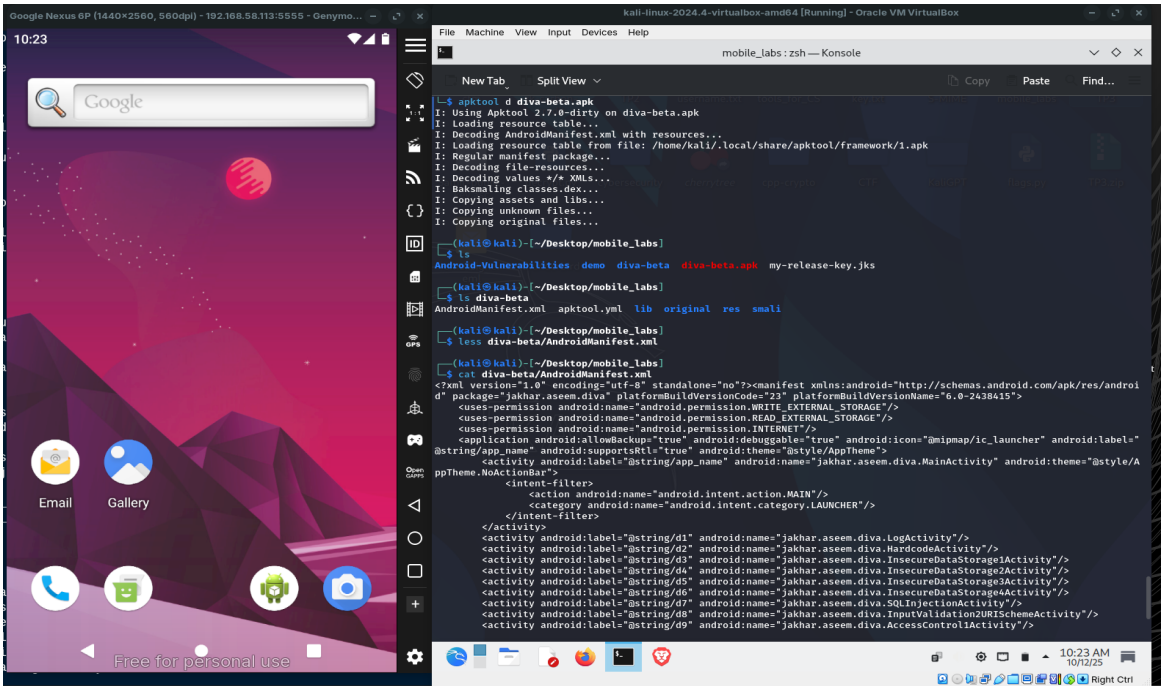


Figure 1.6: Decompiling with Apktool and viewing AndroidManifest.xml

Task 7: Decompile with JaDX and View Results

Description

Use JaDX decompiler to convert the APK into Java source code and analyze the output structure.

Solution

```
cd /usr/share/jadx/bin
sudo ./jadx -d diva diva-beta.apk
ls diva/
```

Results

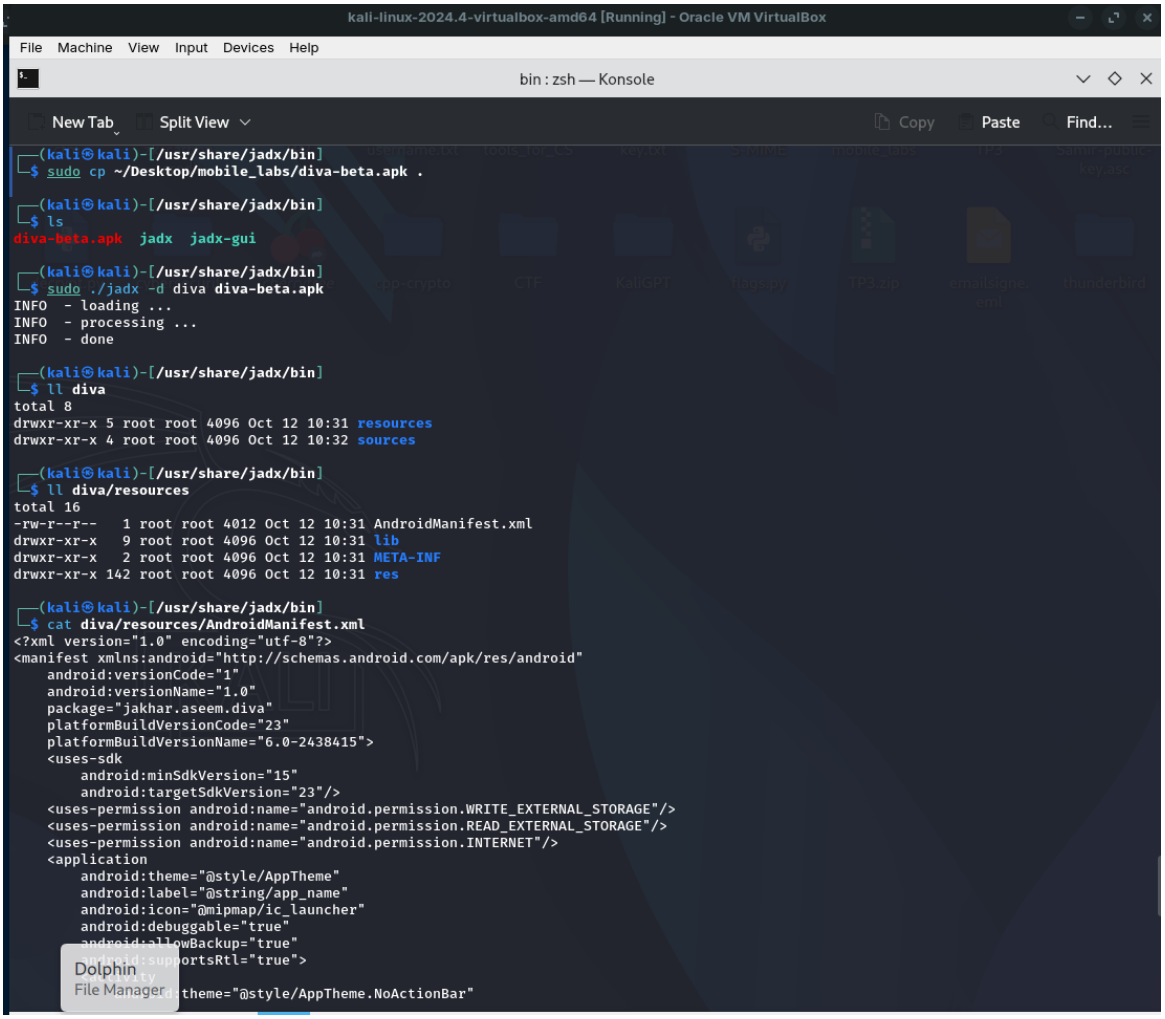


Figure 1.7: Decompiling with JaDX and viewing output structure

Task 8: List JaDX Results in Source Directory

Description

Navigate to the Java source code directory generated by JaDX and examine the decompiled classes.

Solution

```
cd diva/sources/jakhar/aseem/diva
ls -la
```

Results

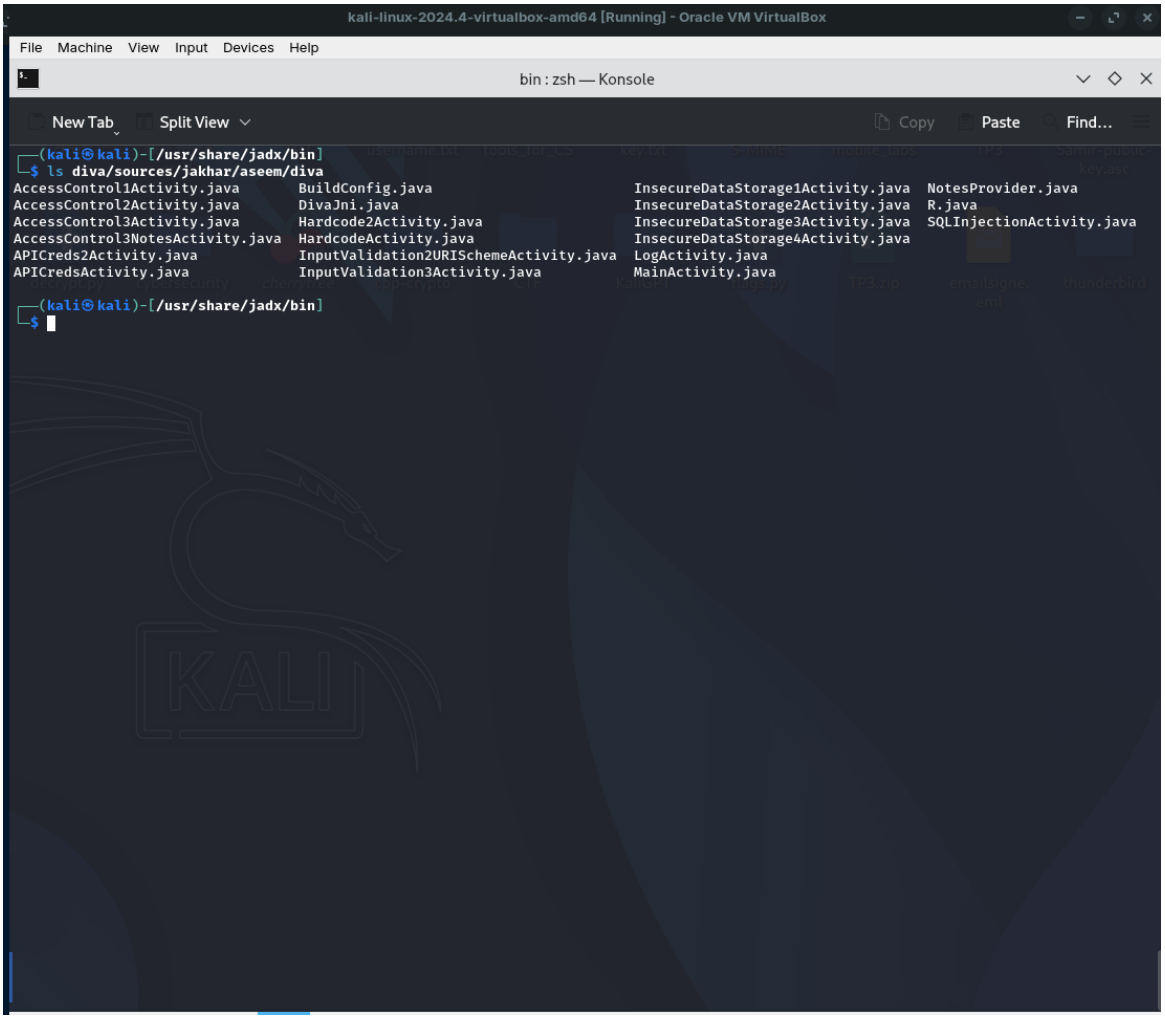


Figure 1.8: Viewing decompiled Java source files

1.4 DEX File Analysis

Task 9: DexDump Classes.dex in Plain Mode

Description

Analyze the DEX file content in plain text format using dexdump tool.

Solution

```
dexdump -l plain classes.dex | less
```

Results

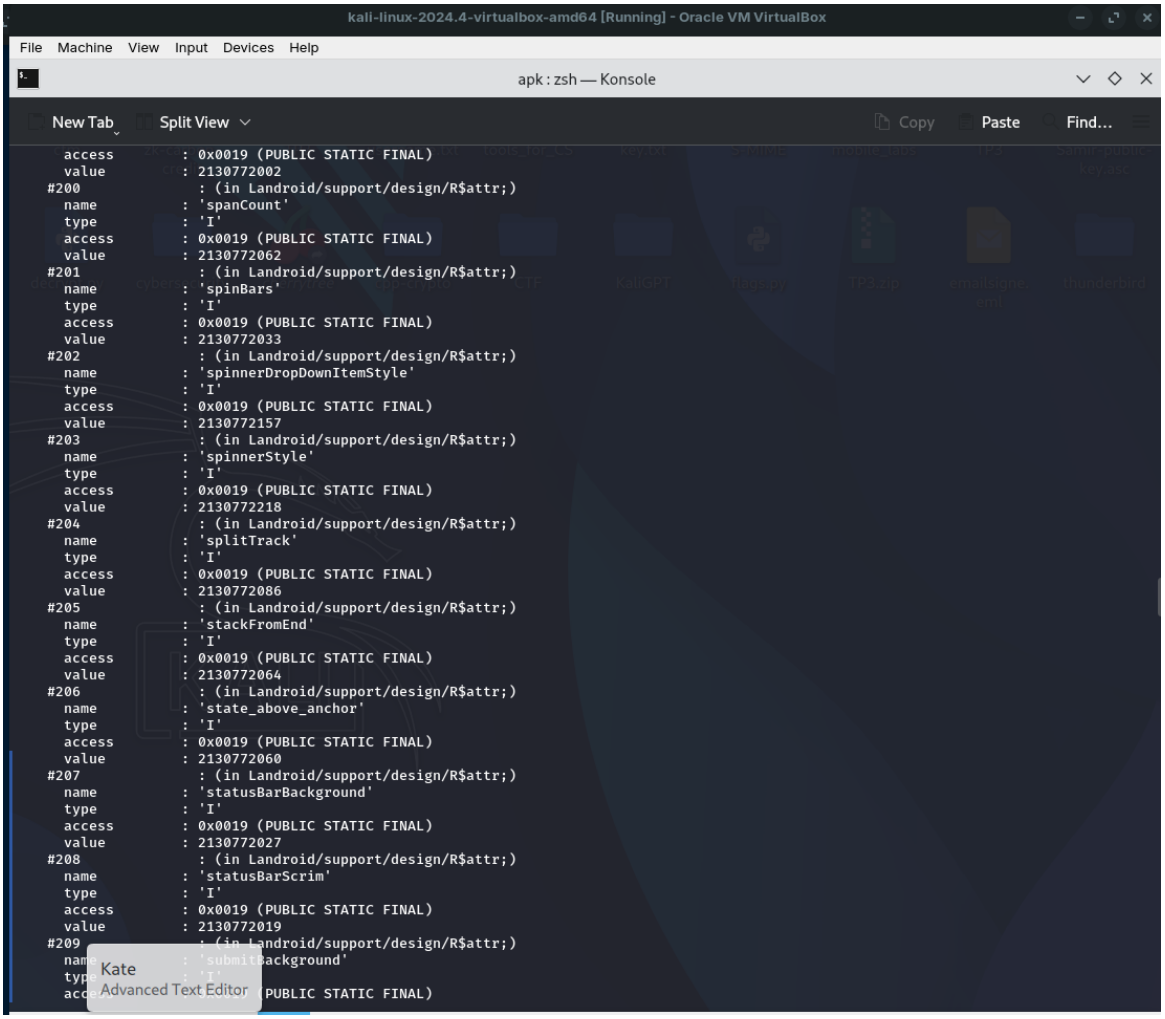


Figure 1.9: DexDump output in plain text format

Task 10: DexDump Classes.dex in XML Mode

Description

Analyze the DEX file content in XML format for structured data examination.

Solution

```
dexdump -l xml classes.dex | less
```

Results

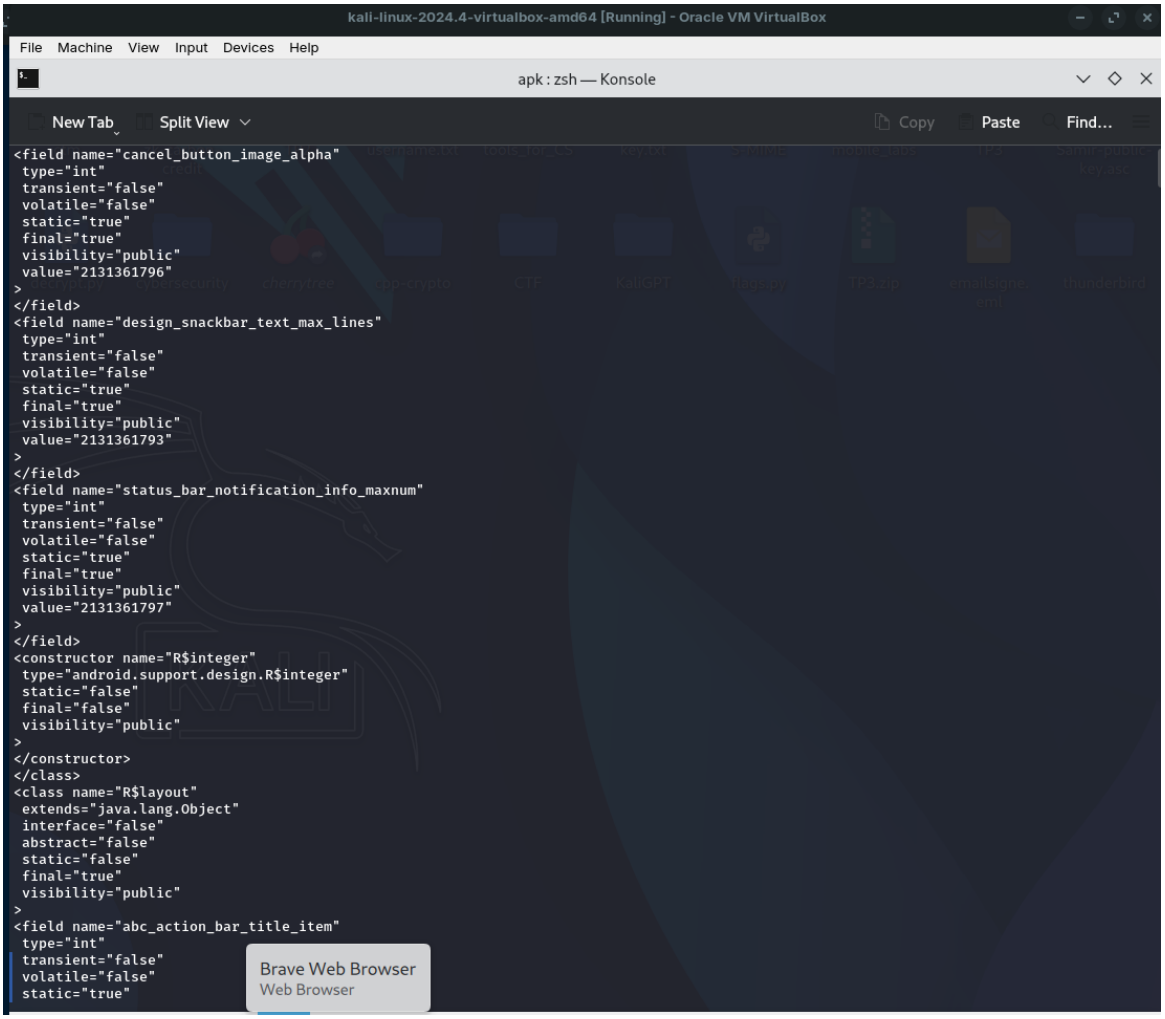


Figure 1.10: DexDump output in XML format

Task 11: Convert DEX to JAR and Analyze with JD-GUI

Description

Convert the DEX file to JAR format and use JD-GUI to visualize the Java bytecode.

Solution

```
d2j-dex2jar classes.dex
jd-gui classes-dex2jar.jar
```

Results

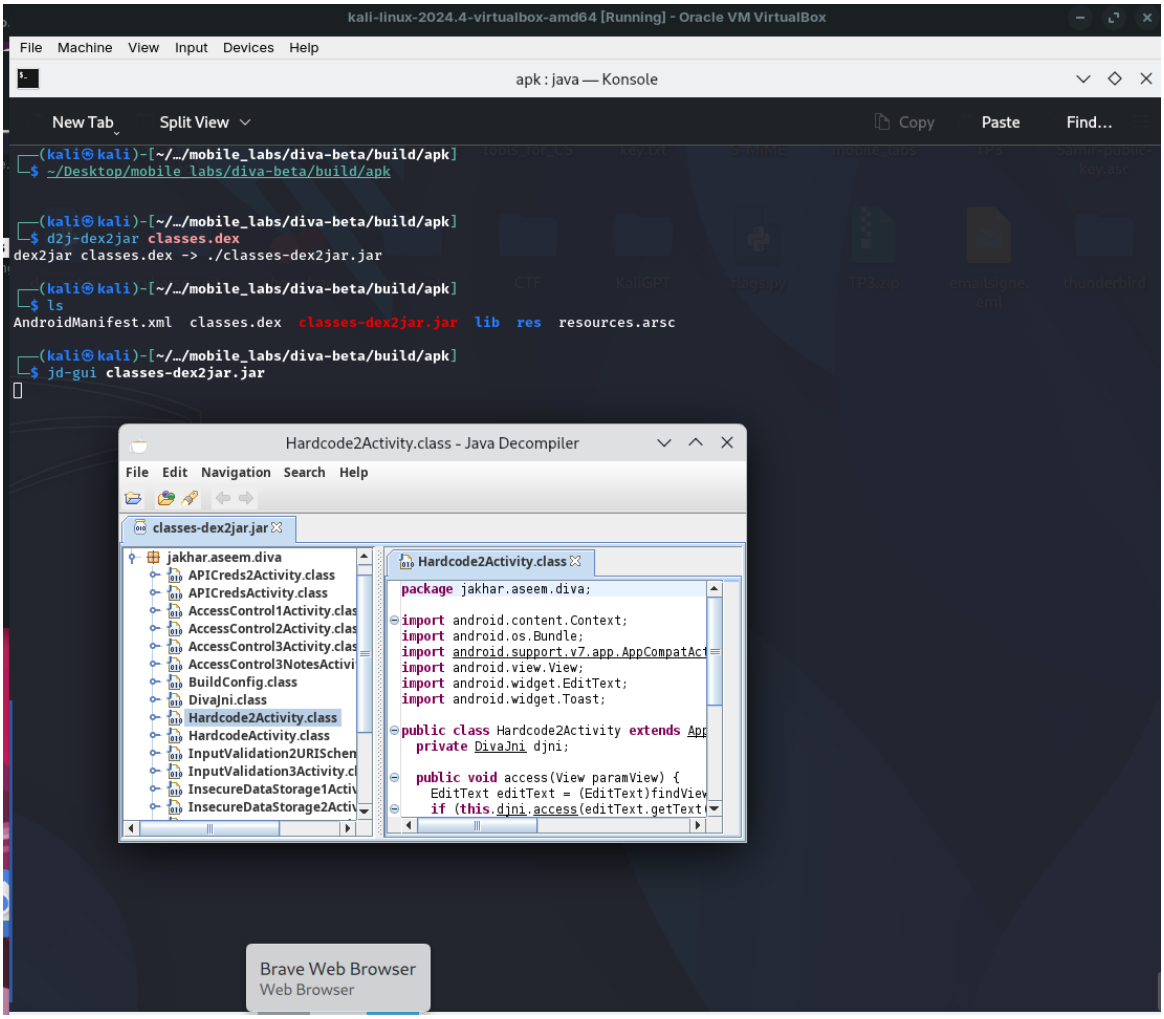


Figure 1.11: Converting DEX to JAR and analyzing with JD-GUI

1.5 Conclusion

This lab provided comprehensive hands-on experience with Android application reverse engineering. Key skills acquired include:

- Setting up Android testing environment with Genymotion
- Using ADB for device communication and application management
- Understanding Android process hierarchy and startup sequence
- Decompiling APK files using both Apktool and JaDX
- Analyzing DEX files using various tools including dexdump and dex2jar
- Visualizing Java bytecode using JD-GUI

CHAPTER 2

PENTESTING LAB

2.1 Lab Preparation

Getting Ready for Analysis

Before diving into the security assessment, we need to prepare our tools and decompile the application to examine its inner workings.

Preparation Steps

```
sudo cp diva-beta.apk /usr/share/jadx/bin
sudo ./jadx -d diva diva-beta.apk
sudo d2j-dex2jar classes.dex
mv diva ~/Desktop/mobile_labs/
jd-gui classes-dex2jar.jar
```

Environment Ready

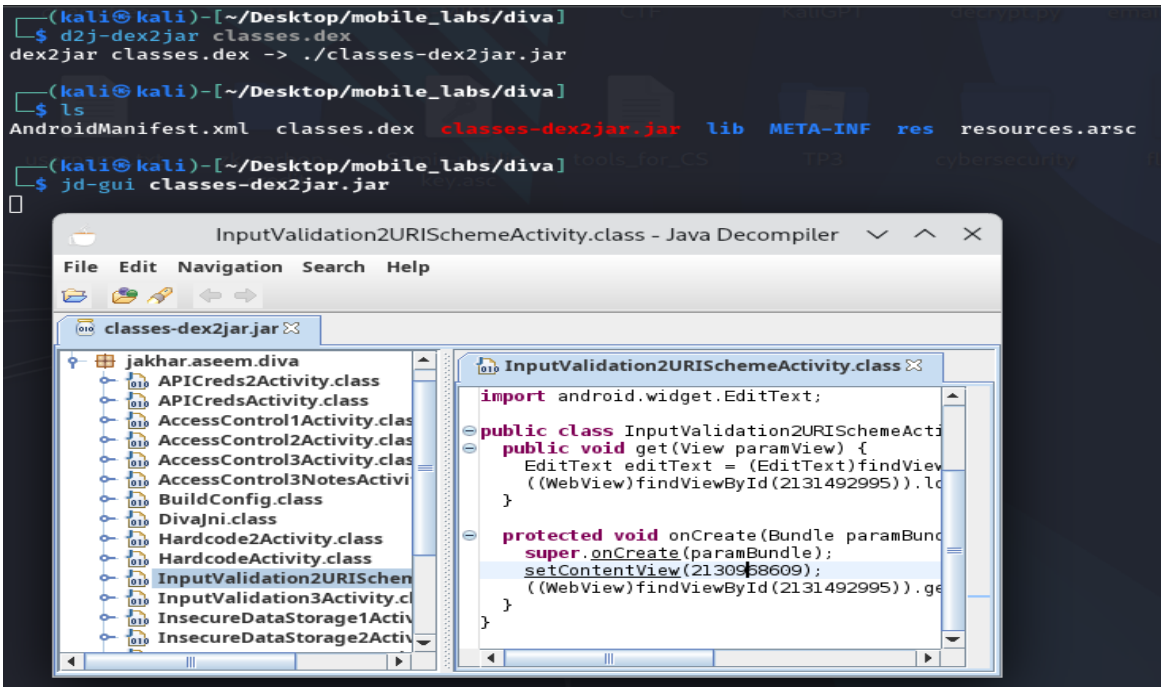


Figure 2.1: Application successfully decompiled and ready for source code analysis

2.2 Insecure Logging Vulnerability

The Problem

Applications sometimes log sensitive information that can be accessed by other applications or malicious actors. This is like writing your password on a sticky note for everyone to see.

```
adb shell ps -ef | grep "diva"
adb shell logcat | grep "2651"
```

Result

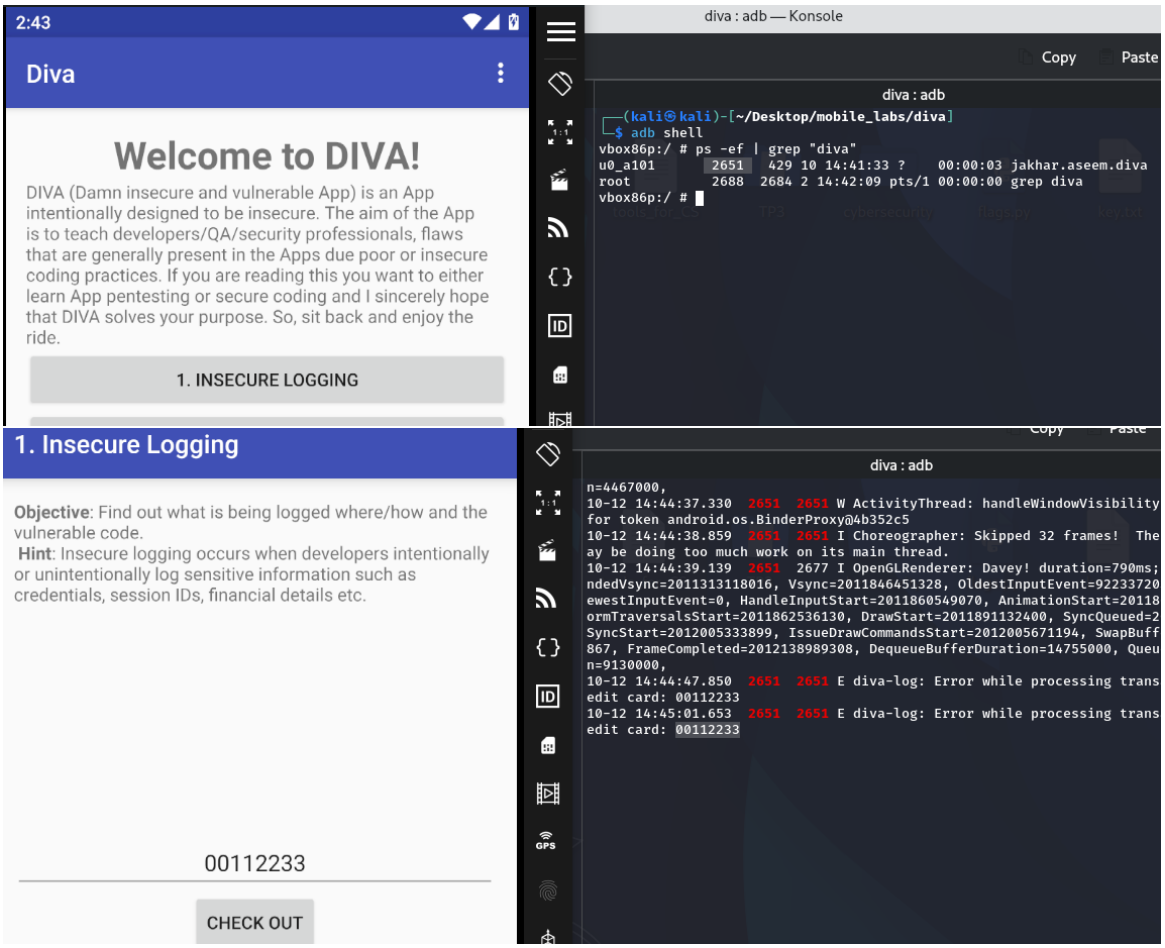


Figure 2.2: Credit card numbers visible in plain text

2.3 Hardcoded Credentials Issue

The Risk

Developers sometimes embed passwords and keys directly in the code, thinking they're hidden. However, when the app is decompiled, these secrets become clearly visible.

Solution

By examining the HardCodeActivity.class file, we uncovered the hidden secret key.

Result

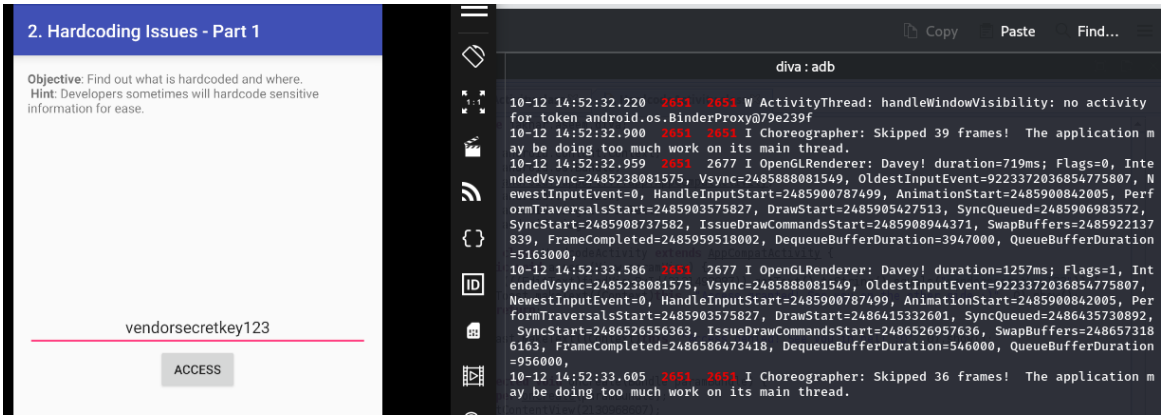


Figure 2.3: Accessed protected features using the discovered hardcoded creds

2.4 Shared Preferences Security Flaw

The Storage Problem

Android apps often use Shared Preferences for storing simple data, but when used for sensitive information without encryption, it becomes a security risk.

Locating the Data

```
cd /data/data/jakhar.aseem.diva/shared_prefs
cat jakhar.aseem.diva_preferences.xml
```

Exposed Information

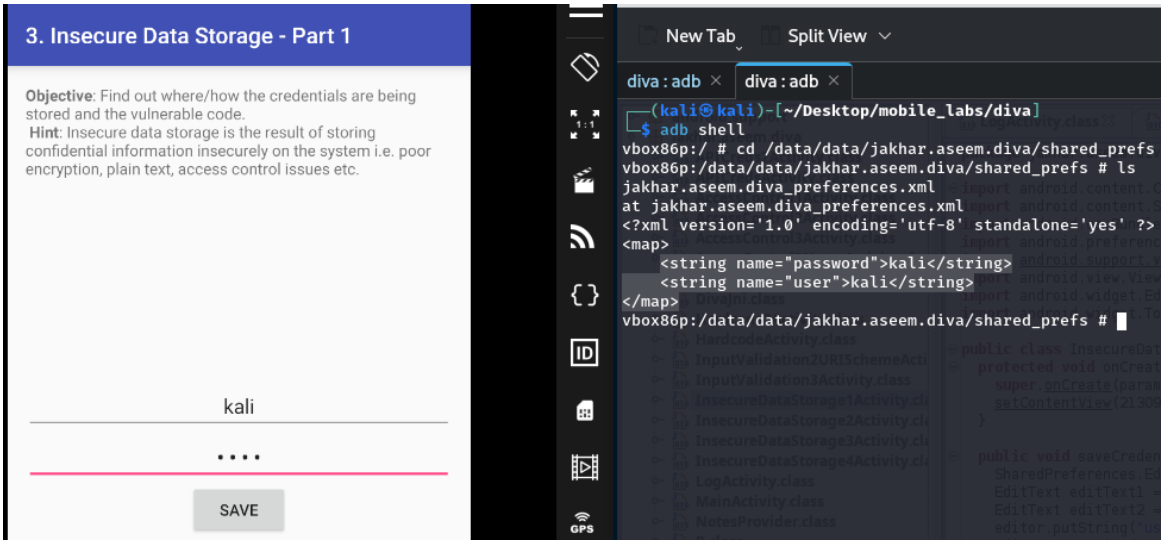


Figure 2.4: User credentials stored in plain text within an XML file

2.5 Database Security Weakness

Database Exposure

SQLite databases are commonly used in Android apps, but without proper encryption, stored data remains vulnerable to extraction.

Database Exploration

```
cd /data/data/jakhar.aseem.diva/databases
sqlite3 ids2
.tables
select * from myuser;
```

Database Contents Revealed

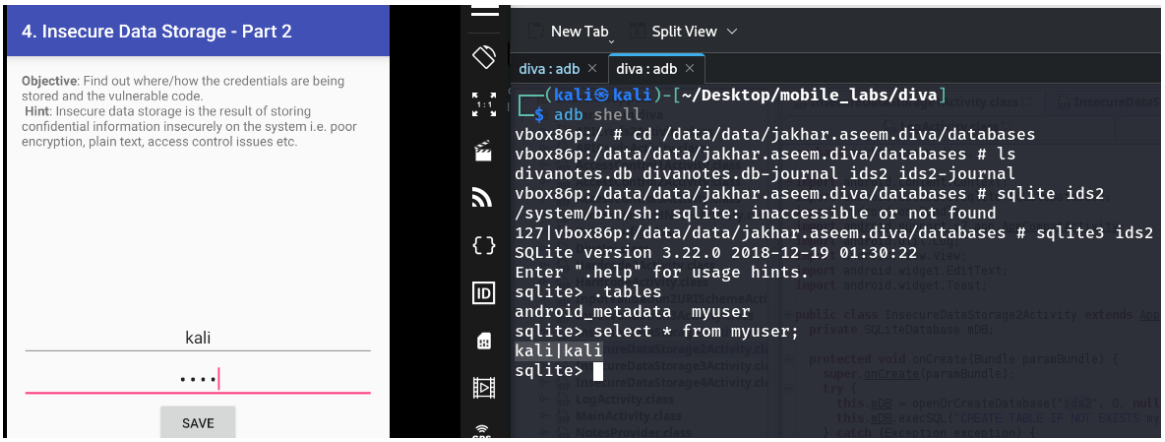


Figure 2.5: User credentials clearly visible in the database without any protection

2.6 Temporary File Mishandling

Temporary Storage Risks

Temporary files might seem harmless, but when they contain sensitive data and aren't properly secured, they become another attack vector.

Finding Temporary Data

```
cd /data/data/jakhar.aseem.diva
cat uinfo
```

Temporary File Exposure

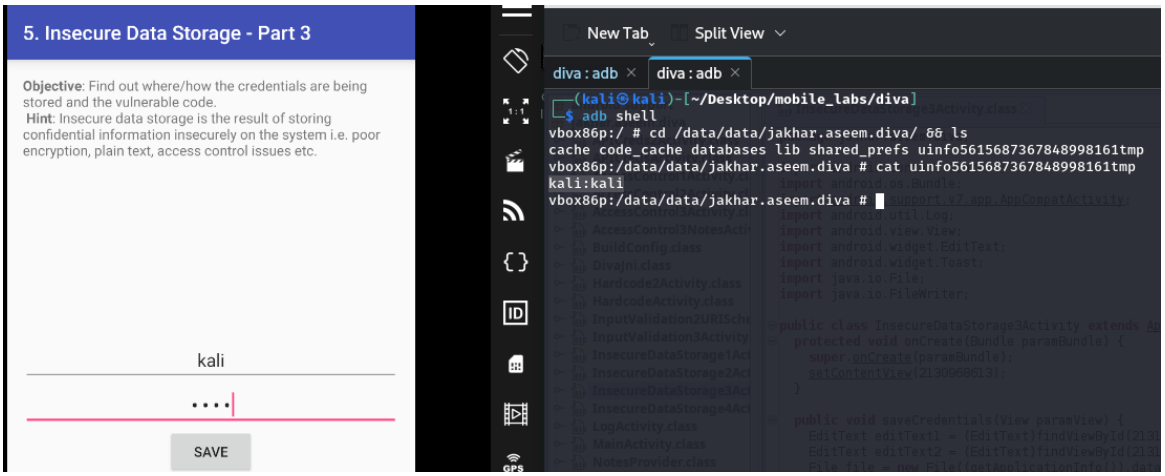


Figure 2.6: Sensitive information stored in temporary files without protection

2.7 External Storage Vulnerability

SD Card Security Issues

Storing data on external storage (like SD cards) is risky because it's accessible to all applications and users.

External Storage Examination

```
cd /mnt/sdcard
ls -a
cat .uinfo.txt
```

External Storage Findings

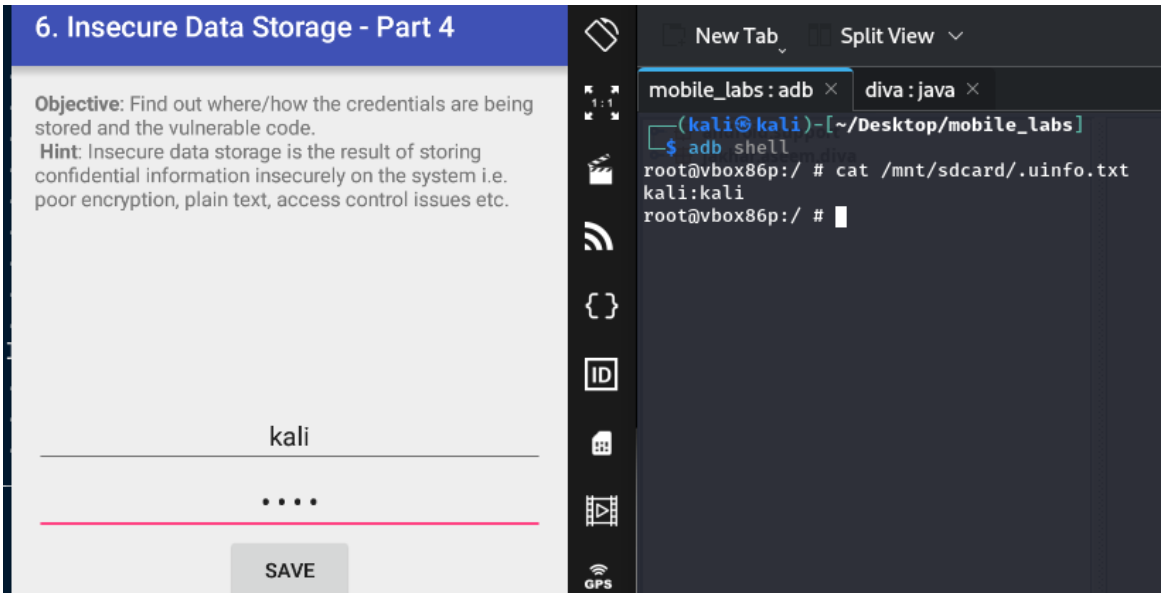


Figure 2.7: Credentials stored on external storage in a hidden file

2.8 SQL Injection Exploitation

Input Validation Failure

SQL injection occurs when applications don't properly validate user input, allowing attackers to manipulate database queries.

Crafting the Attack

Using a simple SQL injection payload: `john' or '1'='1`

Successful Data Extraction

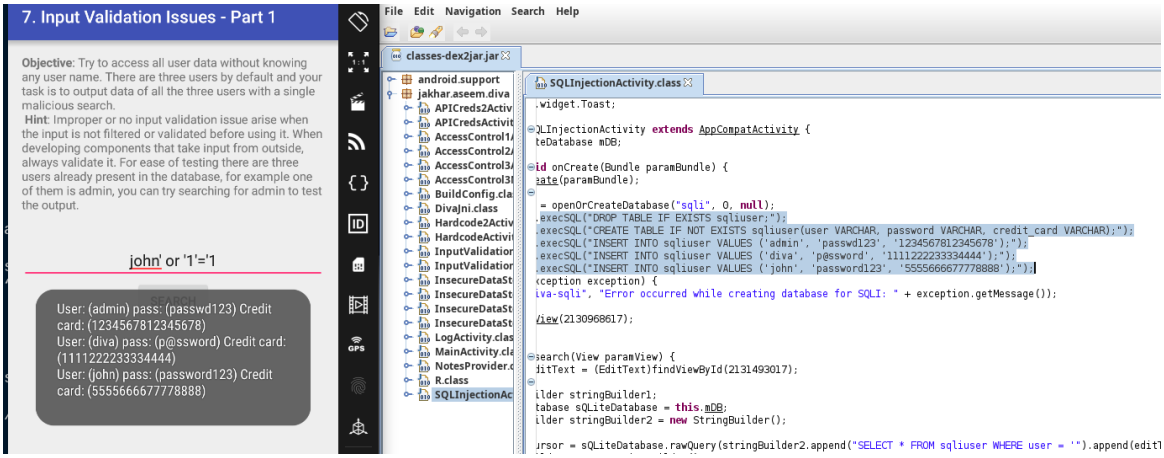


Figure 2.8: All user data extracted from the database using SQL injection

2.9 WebView Security Bypass

WebView Misconfiguration

WebViews can be dangerous when they allow access to local files, potentially exposing sensitive device information.

Exploiting File Access

Using URL: `file:/etc/hosts` to access system files

File Access Demonstrated

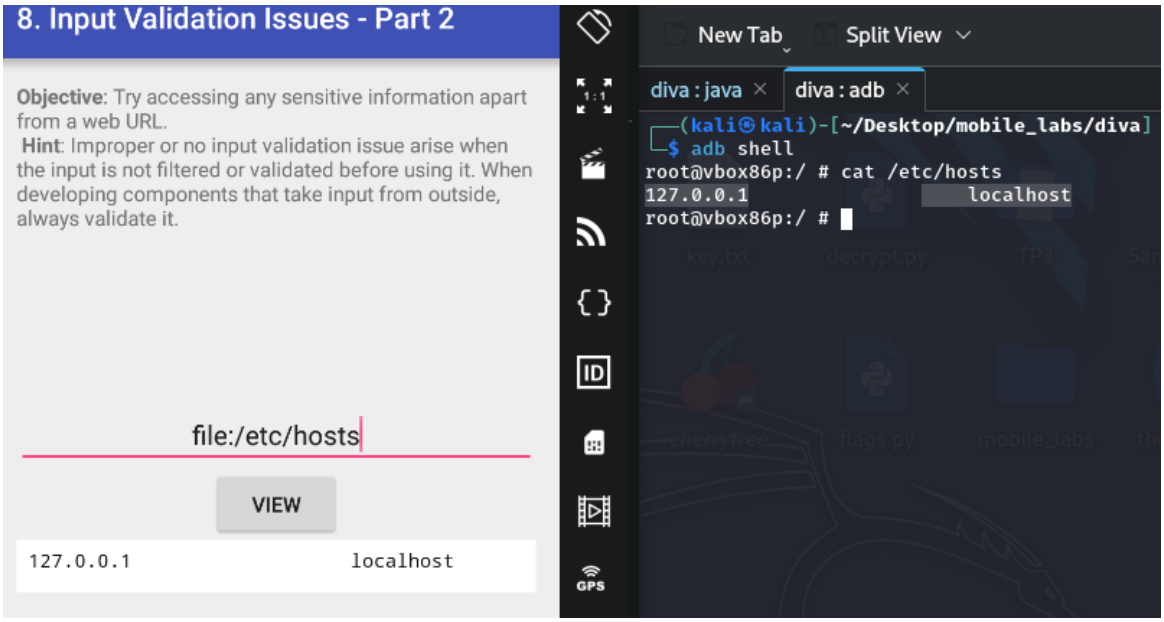


Figure 2.9: Successfully accessed local system files through the WebView component

2.10 Access Control Bypass

Intent Filter Exploitation

Android intents can be invoked directly, sometimes bypassing normal application flow and security checks.

Direct Intent Invocation

```
adb shell am start -a jakhar.aseem.diva.action.VIEW_CREDS
```

Bypass Successful

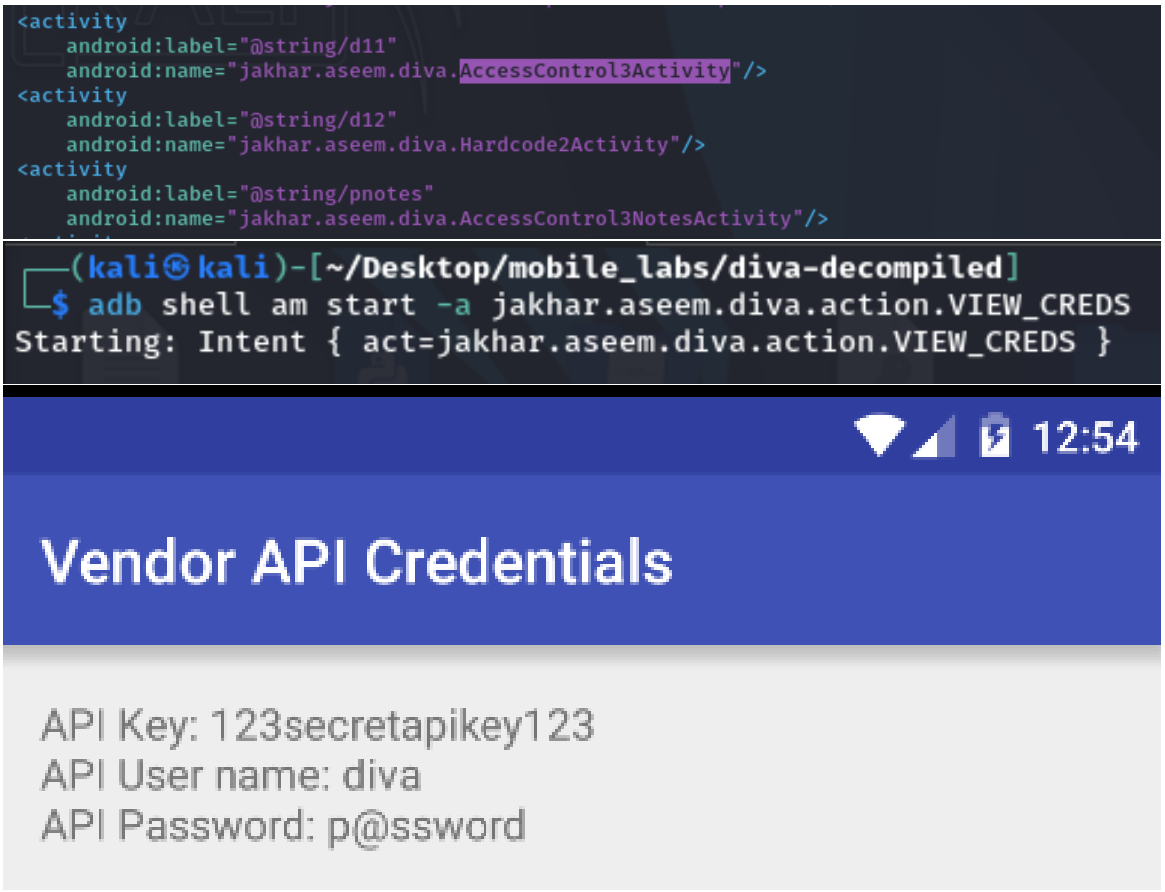


Figure 2.10: Successfully accessed API credentials without going through proper authentication steps

2.11 Authentication Mechanism Bypass

PIN Protection Circumvention

Even when applications implement PIN-based protection, implementation flaws can allow bypassing the security.

Exploiting the Flaw

```
adb shell am start -a jakhar.aseem.diva.action.VIEW_CREDS2 --ez check_pin false
```

Protection Defeated

Figure 2.11: TVEETER API credentials accessed without providing a valid PIN

2.12 Content Provider Data Leak

Exported Content Providers

Content providers that are improperly exported can leak sensitive data to other applications.

Querying the Provider

```
1 adb shell content query --uri content://jakhar.aseem.diva.provider.notesprovider/
    notes
```

Private Data Exposed

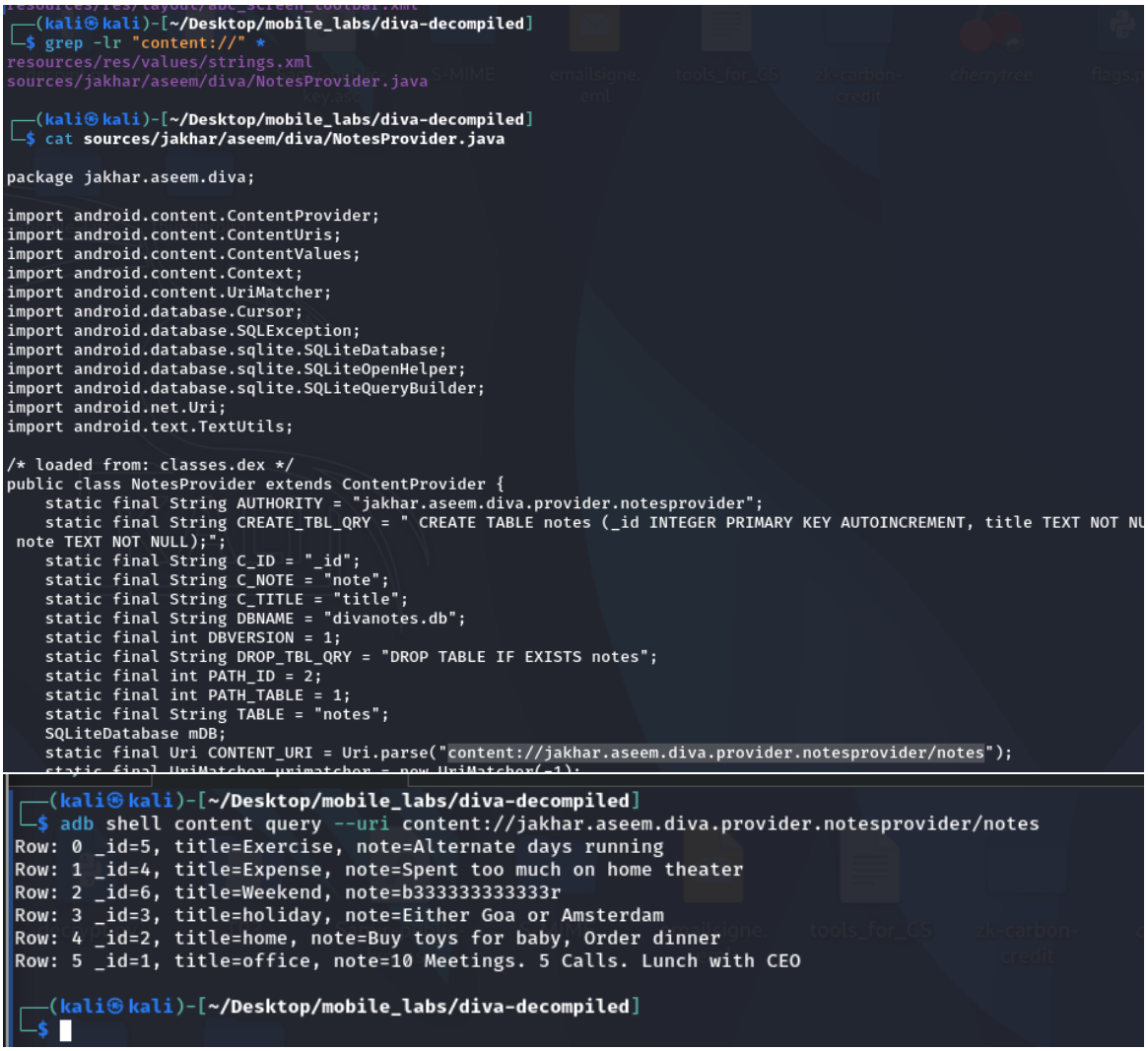


Figure 2.12: Private notes successfully extracted through the exported content provider

2.13 Native Code Hardcoded Secrets

JNI Security Risks

Even when developers move sensitive code to native libraries, hardcoded secrets remain vulnerable to extraction.

Extracting Native Secrets

```
strings libdivajni.so
```

Native Secrets Revealed

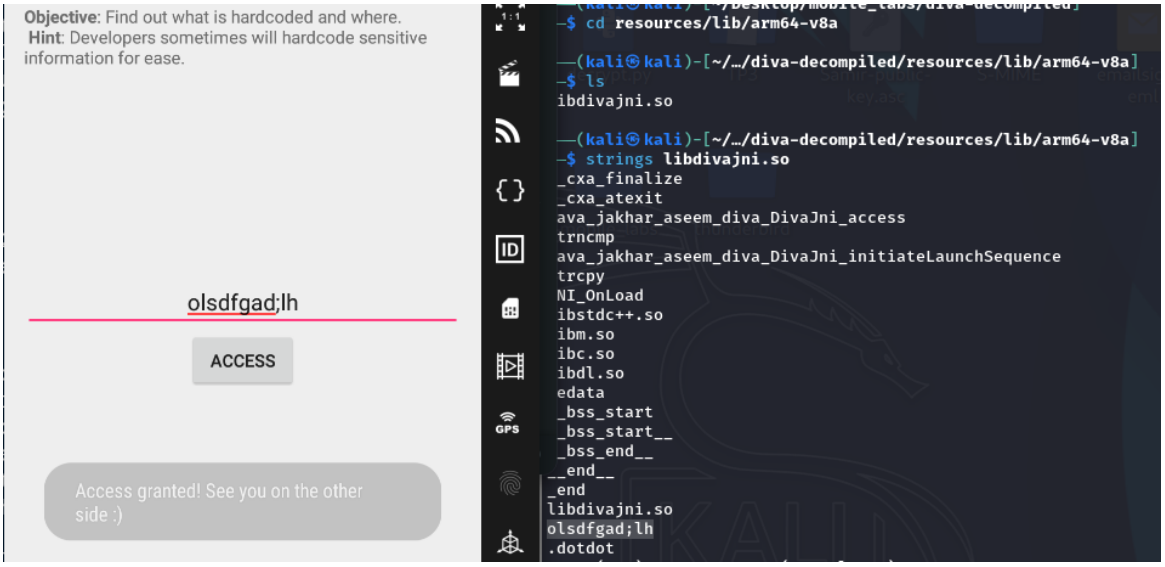


Figure 2.13: Hardcoded key "olsdfgad;lh" discovered in native library and used to gain access

2.14 Denial of Service Vulnerability

Input Size Validation Missing

When applications don't validate input size, they become vulnerable to crashes through buffer overflow attacks.

Triggering the Crash

By providing input longer than the expected 20 characters.

Application Crash

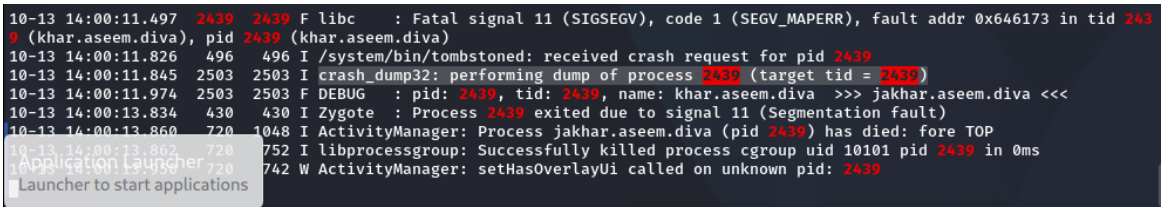


Figure 2.14: Application successfully crashed by providing oversized input

2.15 Conclusion

This hands-on lab provided valuable insights into common mobile application security vulnerabilities. The DIVA application served as an excellent learning tool, demonstrating how seemingly small implementation oversights can lead to significant security breaches.

For security professionals, this lab reinforces the importance of comprehensive security testing covering all aspects of mobile application architecture, from user interface to data storage and network communications.

CHAPTER 3

MOBILE APPLICATION ANALYSIS WITH MOBSF

3.1 MobSF Environment Setup

Setting up MobSF using Docker provides the easiest and most reliable installation method, ensuring all dependencies are properly configured by following these commands :

```
sudo apt-get update && apt install docker.io
sudo docker pull opensecurity/mobile-security-framework-mobsf
sudo docker run -it --rm -p 8000:8000 opensecurity/mobile-security-framework-mobsf:
    latest
```

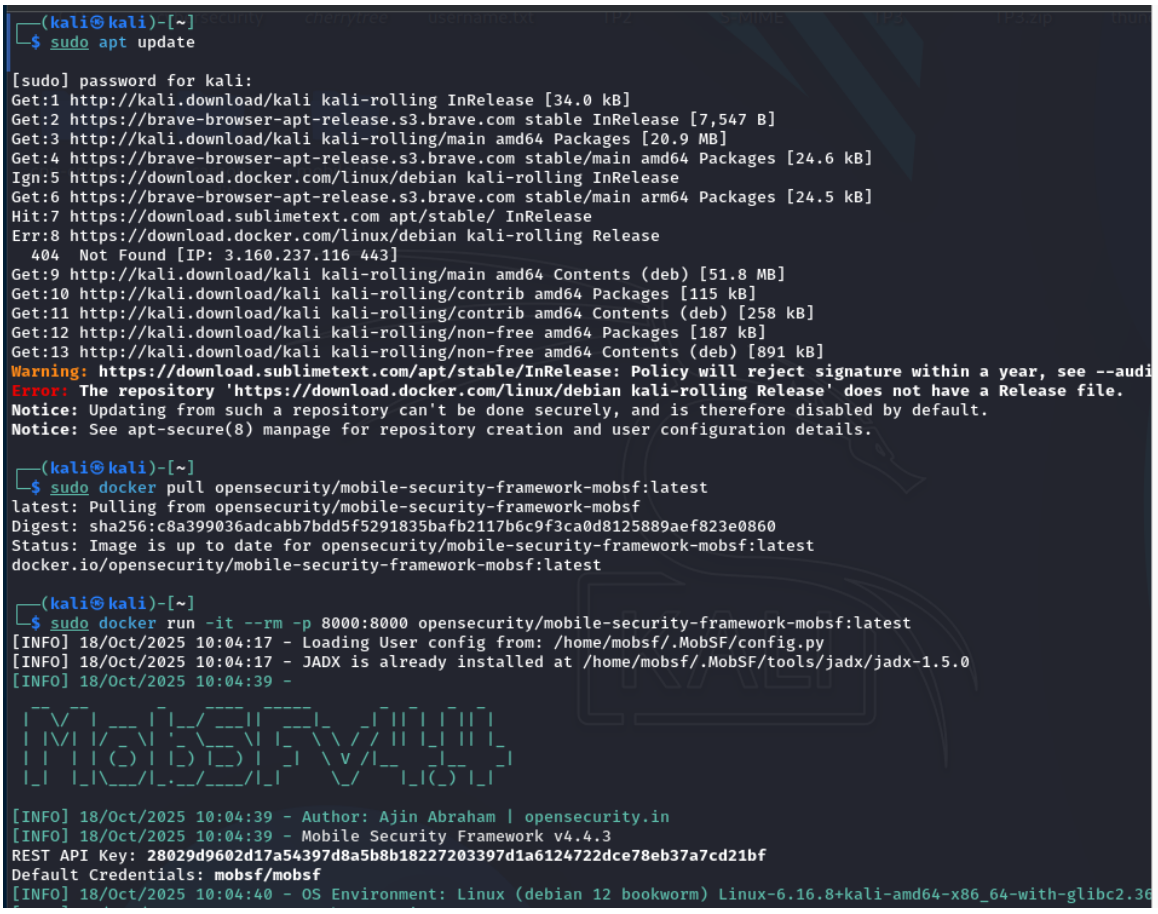


Figure 3.1: Setup MobSF using docker

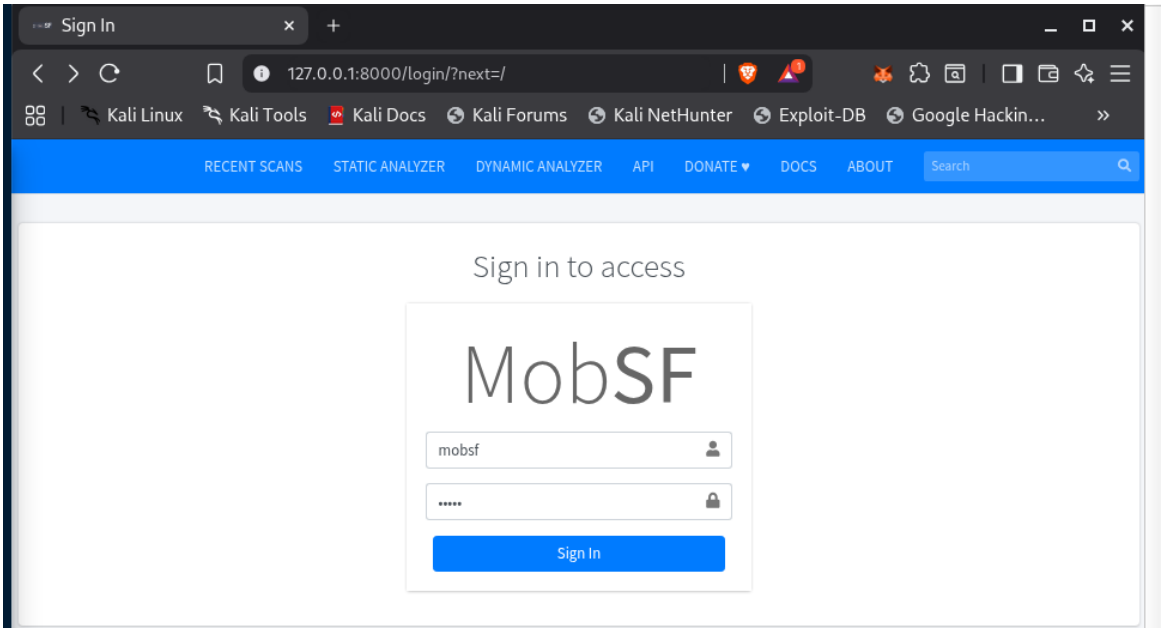


Figure 3.2: MobSF web interface on http://127.0.0.1:8000

3.2 Static Analysis - Android Application

Static analysis examines the application's code without executing it, revealing potential security vulnerabilities, code quality issues, and configuration problems.

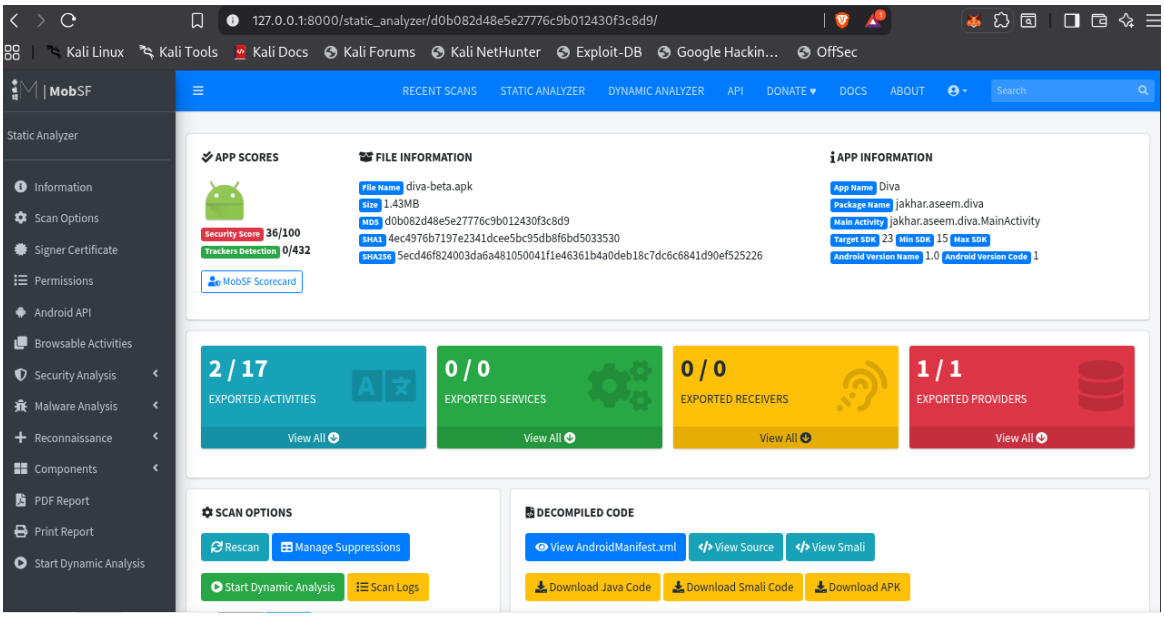


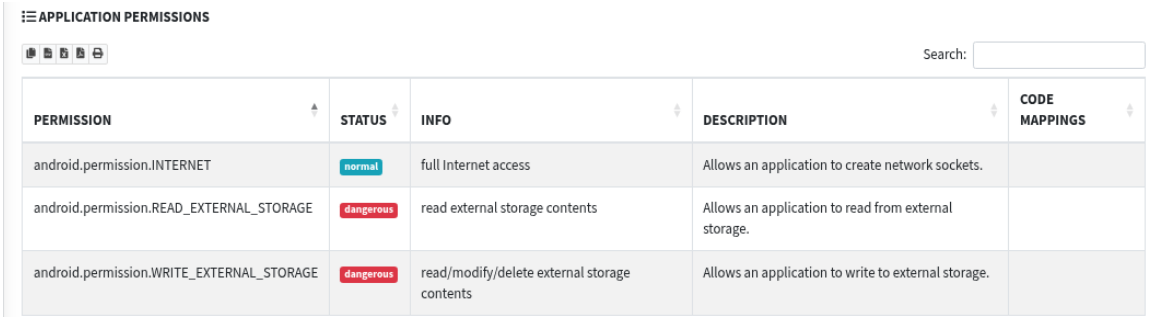
Figure 3.3: Diva Static analysis results

Answers :

1. **Application Components Analysis** The analysis revealed:
   - 17 Activities , No Services , No Receivers and One Provider
   - Exported Components: 2 Activities and 1 Provider are exported, making them accessible to other applications

2. **Certificate Security**
   - **Status:** Valid but uses debug certificate
   - **Validity:** October 12, 2025 to February 27, 2053
   - **Signature:** RSASSA-PKCS1-v1.5 with SHA-384
   - **Issue:** Debug certificate used (not suitable for production)

3. **Permission Analysis** The application requests both normal and dangerous permissions:
   - **WRITE_EXTERNAL_STORAGE** : Dangerous
   - **READ_EXTERNAL_STORAGE** : Dangerous
   - **INTERNET** : Normal



APPLICATION PERMISSIONS

Search:

PERMISSION	STATUS	INFO	DESCRIPTION	CODE MAPPINGS
android.permission.INTERNET	normal	full Internet access	Allows an application to create network sockets.	
android.permission.READ_EXTERNAL_STORAGE	dangerous	read external storage contents	Allows an application to read from external storage.	
android.permission.WRITE_EXTERNAL_STORAGE	dangerous	read/modify/delete external storage contents	Allows an application to write to external storage.	

Figure 3.4: Android apk permissions analysis

4. **API Usage** The app utilizes standard Android APIs for:
   - External storage operations (read/write)
   - SQLite database management
   - Network communication and internet access

ANDROID API

Search:

API	FILES
Content Provider	Show Files
Inter Process Communication	Show Files
Loading Native Code (Shared Library)	Show Files
Local File I/O Operations	Show Files
Starting Activity	Show Files

Figure 3.5: Android APIs analysis

5. **Security Vulnerabilities Identified** Major security concerns include:
   - **Certificate:** Uses only v1 signature vulnerable
   - **Manifest:** Debuggable enabled, backup allowed, low minSdk version
   - **Code:** Insecure logging, temporary file usage, potential SQL injection

6. Hardcoded Information One of the most issues in application development is hardcoding entities, like in this case we have a hardcoded key that used for security reasons:

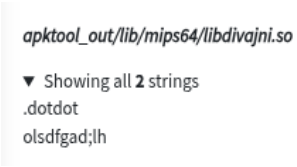


Figure 3.6: Hardcoded issues

7. Sensitive Strings Discovery String analysis revealed potential security concerns including hardcoded keys and debug information that could be exploited.

8. Security Scoring

- **Score:** 36/100
- **Grade:** C (High Risk)
- Multiple critical issues requiring immediate attention

3.3 Static Analysis - iOS Application

Analyzing iOS applications presents different challenges and considerations compared to Android, focusing on binary security and iOS-specific vulnerabilities.

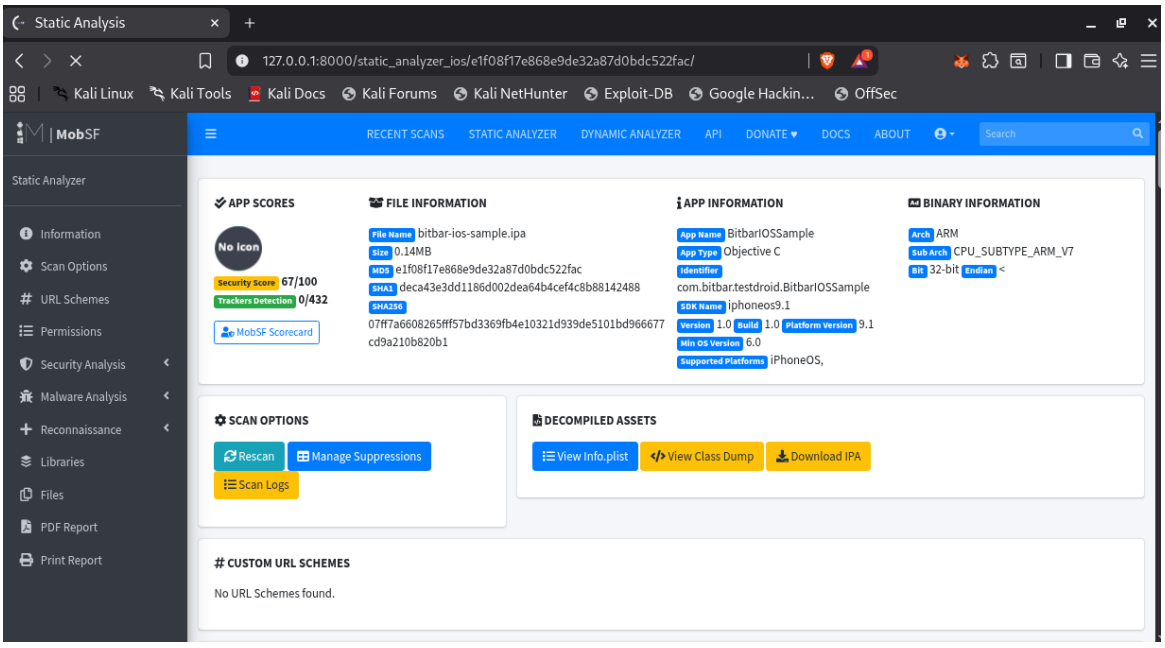


Figure 3.7: Static analysis results for the iOS IPA file

Answers :

1. Development Language : Objective-C

2. Code Signing and Certificates

- **Status:** Binary is properly code signed
- **Encryption:** Binary is not encrypted

- **Signature:** Standard iOS code signing applied

3. iOS Permissions Model Unlike Android, iOS uses a different permission system based on entitlements and usage descriptions rather than dangerous/normal classification.

4. API Usage Patterns

- Uses low-level C functions: _memcpy, _strlen, malloc
- Potential memory safety concerns with manual memory management
- Standard iOS frameworks and libraries

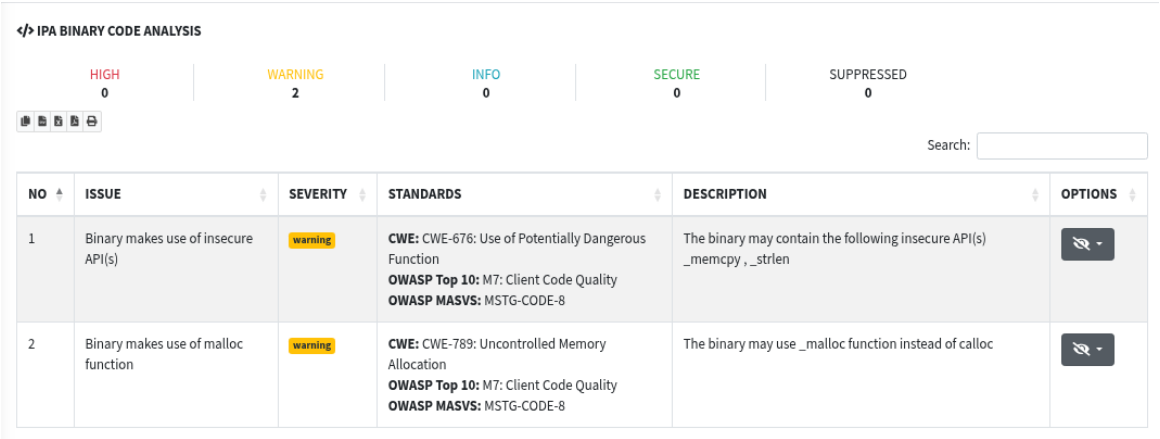
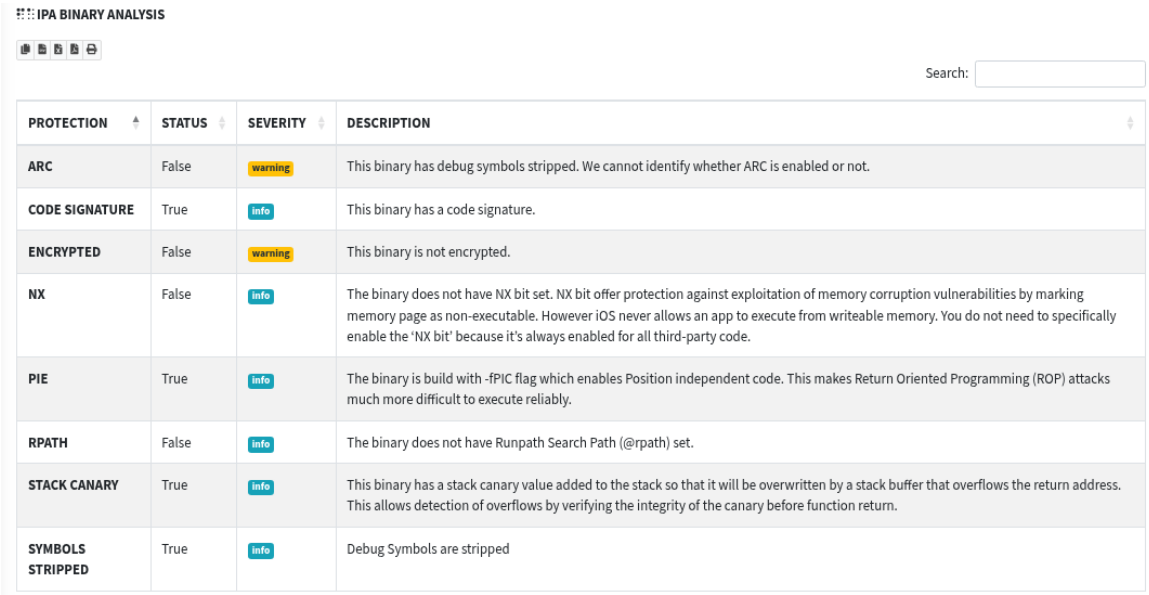


Figure 3.8: iOS binary security assessment results

5. Binary Security Analysis Security assessment revealed:

- **PIE:** Enabled (good practice)
- **Stack Canary:** Present (good practice)
- **NX:** False (potential concern)
- **Binary Encryption:** Not encrypted (App Store requirement)
- **Debug Symbols:** Stripped (good practice)



IPA BINARY ANALYSIS

Search:

PROTECTION	STATUS	SEVERITY	DESCRIPTION
ARC	False	warning	This binary has debug symbols stripped. We cannot identify whether ARC is enabled or not.
CODE SIGNATURE	True	info	This binary has a code signature.
ENCRYPTED	False	warning	This binary is not encrypted.
NX	False	info	The binary does not have NX bit set. NX bit offer protection against exploitation of memory corruption vulnerabilities by marking memory page as non-executable. However iOS never allows an app to execute from writeable memory. You do not need to specifically enable the 'NX bit' because it's always enabled for all third-party code.
PIE	True	info	The binary is build with -fPIC flag which enables Position independent code. This makes Return Oriented Programming (ROP) attacks much more difficult to execute reliably.
RPATH	False	info	The binary does not have Runpath Search Path (@rpath) set.
STACK CANARY	True	info	This binary has a stack canary value added to the stack so that it will be overwritten by a stack buffer that overflows the return address. This allows detection of overflows by verifying the integrity of the canary before function return.
SYMBOLS STRIPPED	True	info	Debug Symbols are stripped

Figure 3.9: iOS binary code analysis

6. Hardcoded URLs : No hardcoded secrets

7. String Analysis String reconnaissance didn't reveal critical secrets, but identified various application strings and URLs that could be useful for further analysis.

8. Security Scoring

- **Score:** 67/100
- **Grade:** A (Low Risk)
- Generally good security posture with minor concerns

3.4 Dynamic Analysis Setup

Dynamic analysis requires proper configuration with an Android device or emulator to monitor runtime behavior and network activity.

```
1 sudo docker run -e MOBSF_ANALYZER_IDENTIFIER=192.168.58.113:5555 -p 8000:8000
    opensecurity/mobile-security-framework-mobsf:latest
```

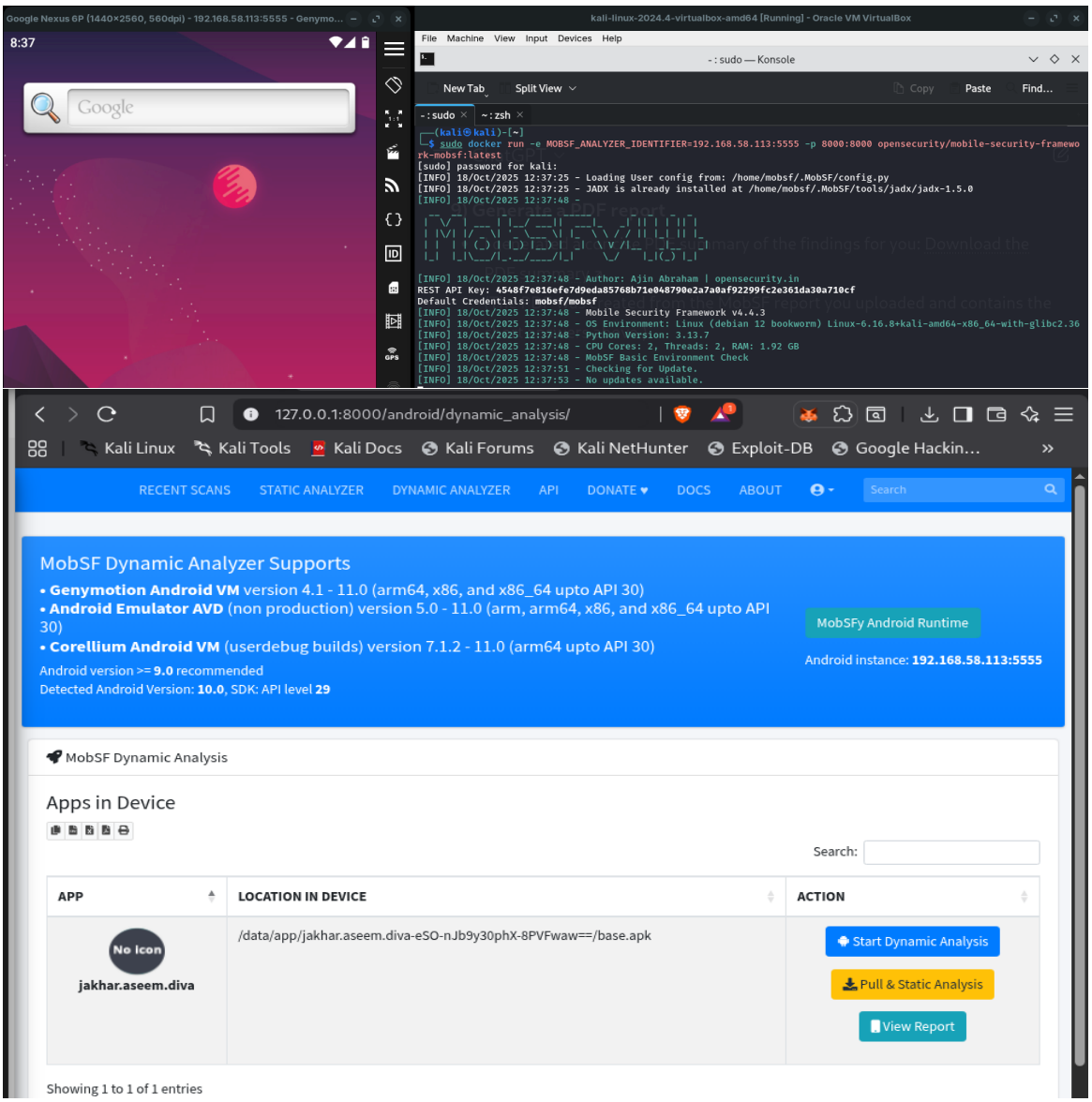


Figure 3.10: MobSF dynamic analysis interface

3.5 Dynamic Analysis Features

Dynamic analysis provides real-time monitoring of application behavior during execution. The MobSF provides much capabilities when moving to dynamic analysis as follows:

- **Screen Mirroring:** Real-time device screen display with interactive controls
- **Certificate Management:** Install/remove Root CA for traffic inspection
- **Proxy Configuration:** Set up HTTP/HTTPS interception proxy
- **TLS/SSL Testing:** Evaluate network connection security
- **Dependency Analysis:** Identify runtime dependencies and external resources
- **Screenshot Capture:** Document application states and screens
- **Logcat Monitoring:** Real-time system and application logs
- **Report Generation:** Comprehensive PDF reports of findings

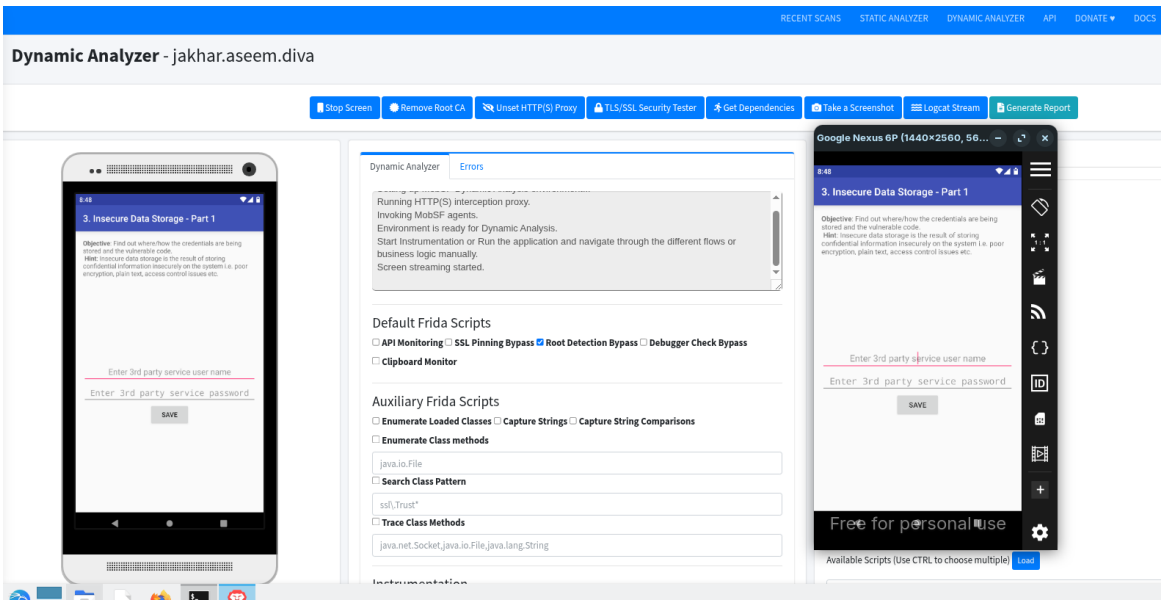


Figure 3.11: Dynamic analysis of Diva application showing runtime behavior

3.6 Report Generation

MobSF present one of the most security analysis important steps that is generating detailed reports summarizing all findings from both static and dynamic analysis, which gives an advancement for developers and security teams by providing features like :

- Executive summary with security score
- Detailed vulnerability breakdown
- Risk assessment and prioritization
- Technical details for developers
- Remediation recommendations

3.7 Conclusion

MobSF proves to be an invaluable tool for security professionals, providing comprehensive mobile application security assessment capabilities that help identify and remediate vulnerabilities before they reach production environments.

CHAPTER 4

REPACKAGING ATTACK LAB

4.1 Environment Setup

Before beginning the repackaging process, we need to establish communication between our Kali Linux machine and the Android virtual machine.

```
adb connect 192.168.58.113
adb install InsecureBankv2.apk
```

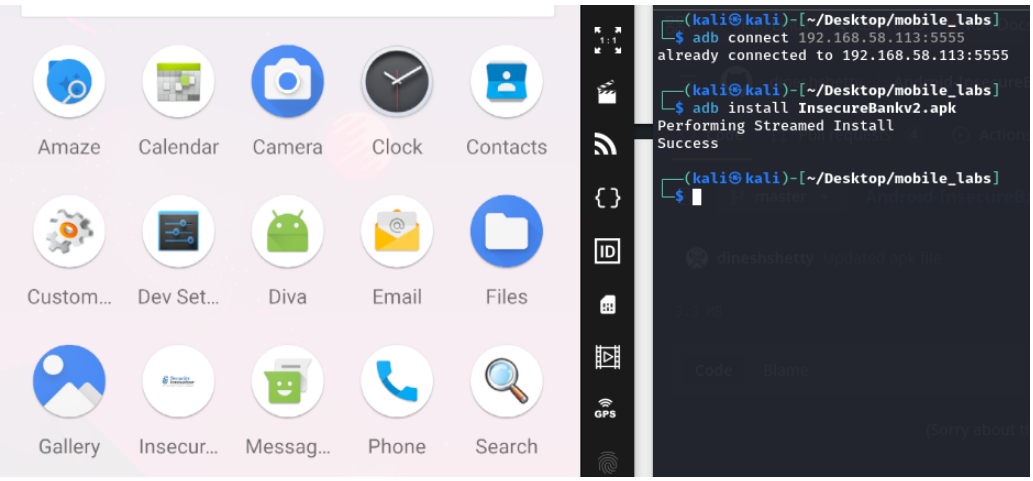


Figure 4.1: InsecureBankv2 application successfully installed on Android VM

4.2 Application Disassembly

To modify the application, we first need to decompile it into human-readable code using APKTool, which converts Dalvik bytecode into Small format.

```
apktool d InsecureBankv2.apk
```

4.3 Malicious Code Injection

We inject a malicious broadcast receiver that will delete all contacts when triggered by system events like volume changes or ringer mode modifications by creating new smali file `MaliciousCode.smali` under the path `smali/con/android/insecurebankv2/`.

```
MaliciousCode.smali                    ViewStatement.smali
MyBroadCastReceiver.smali              WrongLogin.smali

┌──(kali㉿kali)-[~/Desktop/mobile_labs/InsecureBankv2]
└─$ head smali/com/android/insecurebankv2/MaliciousCode.smali
.class public Lcom/MaliciousCode;
.super Landroid/content/BroadcastReceiver;
.source "MaliciousCode.java"


# direct methods
.method public constructor <init>()V
    .registers 1

    .prologue
```

Figure 4.2: Malicious Small code added to the decompiled application

Next we need to add necessary permissions and register our broadcast receiver in the `AndroidManifest.xml` file.

```
    <!— Permissions injection —>
    <uses-permission android:name="android.permission.READ_CONTACTS" />
    <uses-permission android:name="android.permission.WRITE_CONTACTS" />
    <!— End Permission injection—>

    <android:uses-permission android:name="android.permission.READ_PHONE_STATE"/>
    <android:uses-permission android:maxSdkVersion="18" android:name="android.permission.READ_EXTERNAL_STORAGE"/>
    <android:uses-permission android:name="android.permission.READ_CALL_LOG"/>
    <uses-permission android:name="android.permission.ACCESS_NETWORK_STATE"/>
    <uses-permission android:name="android.permission.ACCESS_COARSE_LOCATION"/>
    <uses-feature android:glEsVersion="0×00020000" android:required="true"/>
    <application android:allowBackup="true" android:debuggable="true" android:icon="@mipmap/ic_launcher"
android:label="@string/app_name" android:theme="@android:style/Theme.Holo.Light.DarkActionBar">

        <!— Begin Receiver injection —>
        <receiver android:name="com.android.insecurebankv2.MaliciousCode">
                <intent-filter>
                        <action android:name="android.media.VOLUME_CHANGED_ACTION" />
                        <action android:name="android.media.RINGER_MODE_CHANGED" />
                </intent-filter>
        </receiver>
        <!— End receiver injection—>
```

Figure 4.3: Permissions and receiver added to AndroidManifest.xml

4.4 Application Repackaging

After injecting our malicious code, we repackage the application into a new APK file by using apktool with build option and the result is apk `MaliciousBank.apk`.

```
apktool b InsecureBankv2 -o MaliciousBank.apk
```

```
┌──(kali㉿kali)-[~/Desktop/mobile_labs]
└─$ apktool b InsecureBankv2 -o malicious_bank.apk
I: Using Apktool 2.7.0-dirty
I: Checking whether sources has changed...
I: Smaling smali folder into classes.dex...
I: Checking whether resources has changed...
I: Building resources...
W: aapt: brut.common.BrutException: brut.common.BrutException: Could not extract resource: /prebuilt/linux/aapt_64 (de
faulting to $PATH binary)
I: Building apk file...
I: Copying unknown files/dir...
I: Built apk into: malicious_bank.apk
```

Figure 4.4: Successful repackaging of the malicious application

4.5 APK Signing

Android requires all applications to be digitally signed before installation. We create a self-signed certificate for our repackaged app.

```
keytool -genkey -v -keystore debug.keystore -alias androiddebugkey -storepass
   android -keypass android -dname "CN=Android Debug, O=Android, C=US" -keyalg RSA -
   keysize 2048 -validity 10000
```

```
jarsigner -verbose -sigalg SHA1withRSA -digestalg SHA1 -keystore debug.keystore -
   storepass android -keypass android malicious_bank.apk androiddebugkey
```

Figure 4.5: Successful signing of the malicious APK with debug certificate

4.6 Installation and Testing

We have to uninstall the original application and install our repackaged version with the malicious code.

```
adb uninstall com.android.insecurebankv2
adb install MaliciousBank.apk
```

Checking permissions

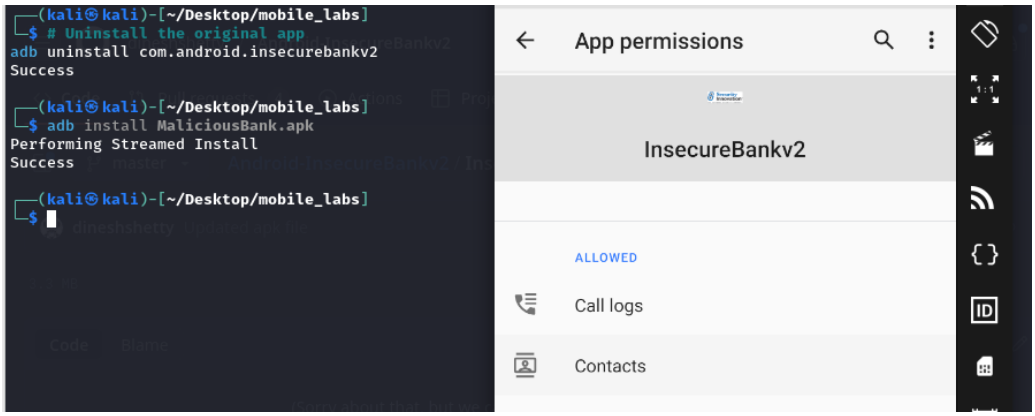


Figure 4.6: Repackaged app installed with contacts permissions granted

4.7 Attack Demonstration

To demonstrate the attack, we create test contacts and trigger the malicious code through system events.

Creating Test Data

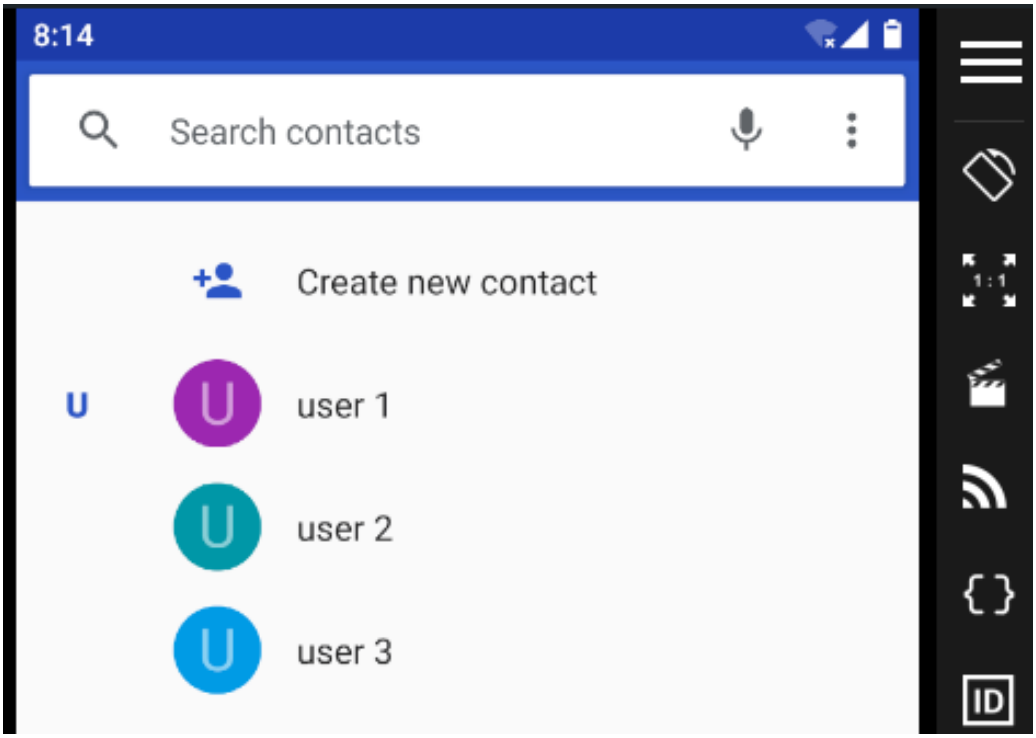


Figure 4.7: Test contacts created before triggering the malicious code

Application Launch

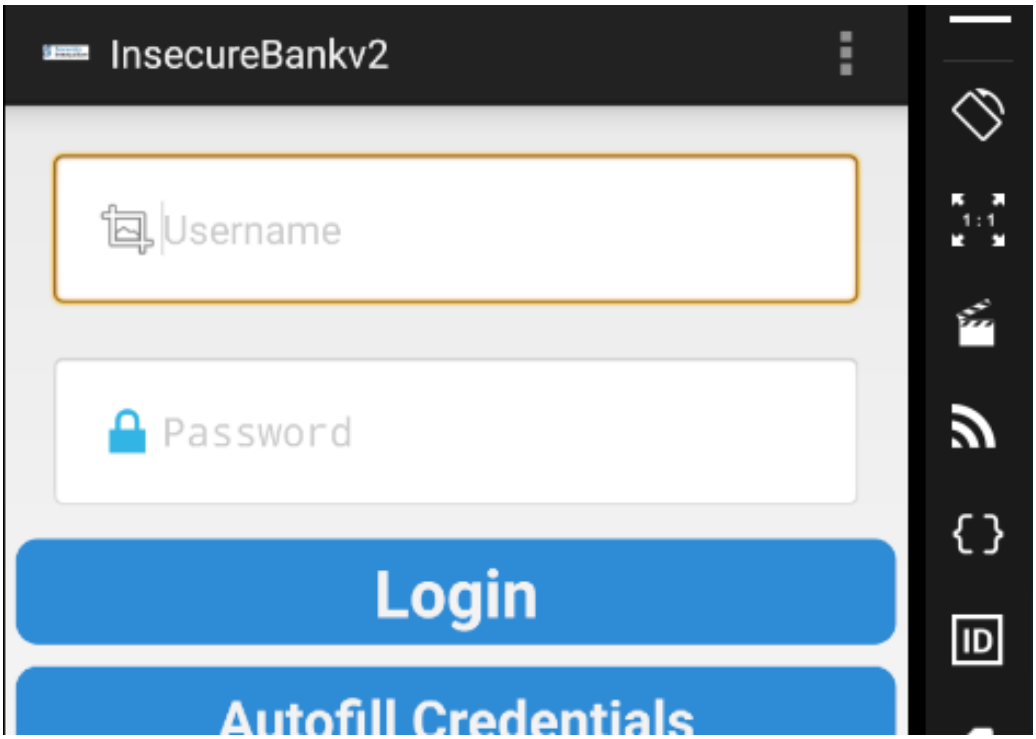


Figure 4.8: Repackaged banking application running normally

Results

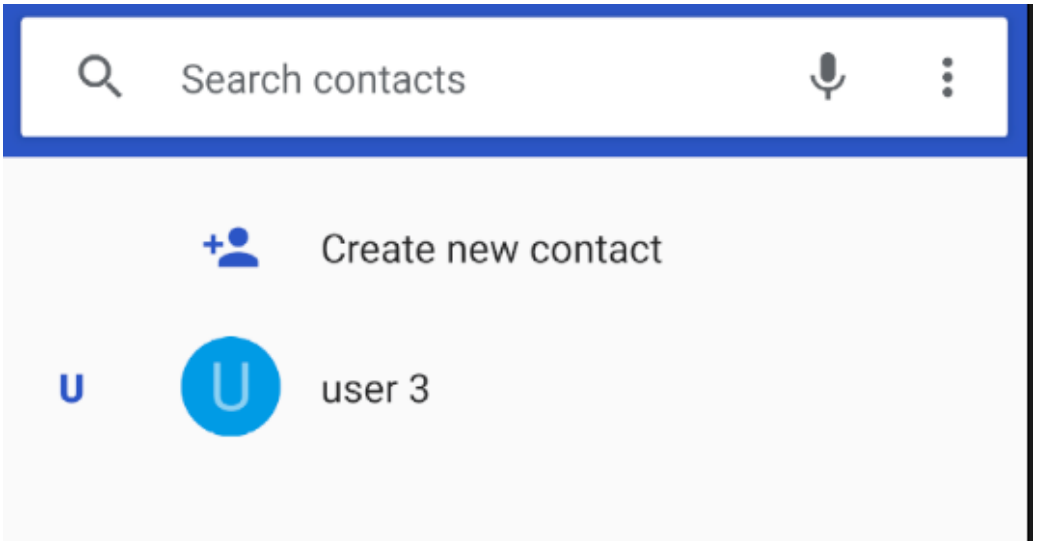


Figure 4.9: Deleting contacts after launching the app

4.8 Attack mecanism

- **Trigger:** System broadcast events (VOLUME_CHANGED_ACTION, RINGER_MODE_CHANGED)
- **Action:** Automatic deletion of all contact records
- **Stealth:** No visible indication to the user
- **Persistence:** Registered receiver remains active

4.9 Questions replay

a) iOS vs Android Security : iOS benefits from a closed ecosystem with strict code signing and App Store exclusivity, making repackaging extremely difficult. Android's open nature, while promoting innovation, creates more vulnerability to such attacks.

b) Google's Improvement Opportunities : Google could enhance security through better app verification, runtime integrity checks, and advanced machine learning detection to identify repackaged applications more effectively.

c) Official Store Limitations : While Google Play Store is safer than third-party markets, its massive scale and automated review processes mean some malicious apps can still slip through undetected.

d) User Protection Strategies : When using untrusted sources, users should verify developer credibility, scrutinize requested permissions, use security software, and monitor device behavior for anomalies.