

SRM VALLIAMMAI ENGINEERING COLLEGE

(An Autonomous Institution)

SRM Nagar, Kattankulathur – 603 203

DEPARTMENT OF CYBER SECURITY

Lab Manual

SECURITY IN MOBILE EXPLOITATION LABORATORY

(REGULATION 2023)



VII SEMESTER - FINAL YEAR

CY3763 – SECURITY IN MOBILE EXPLOITATION LABORATORY

Regulation – 2023

Academic Year: 2026 – 2027 (ODD)

Prepared by

Ms. K.R.Nandhashree, Assistant Professor (O.G)

PROGRAMME OUTCOMES

- 1. Engineering knowledge:** Apply the knowledge of mathematics, science, engineering fundamentals, and an engineering specialization to the solution of complex engineering problems.
- 2. Problem analysis:** Identify, formulate, review research literature, and analyze complex engineering problems reaching substantiated conclusions using the first principles of mathematics, natural sciences, and engineering sciences.
- 3. Design/development of solutions:** Design solutions for the complex engineering problems and design system components or processes that meet the specified needs with appropriate consideration for the public health and safety, and the cultural, societal, and environmental considerations.
- 4. Conduct investigations of complex problems:** Use research-based knowledge and research methods including the design of experiments, analysis and interpretation of data, and the synthesis of the information to provide valid conclusions.
- 5. Modern tool usage:** Create, select, and apply appropriate techniques, resources, and modern engineering and IT tools including prediction and modeling to complex engineering activities with an understanding of the limitations.
- 6. The engineer and society:** Apply reasoning informed by the contextual knowledge to assess societal, health, safety, legal and cultural issues and the consequent responsibilities relevant to the professional engineering practice.
- 7. Environment and sustainability:** Understand the impact of the professional engineering solutions in societal and environmental contexts, and demonstrate the knowledge of, and need for sustainable development.
- 8. Ethics:** Apply ethical principles and commit to professional ethics and responsibilities and norms of the engineering practice.
- 9. Individual and team work:** Function effectively as an individual, and as a member or leader in diverse teams, and in multidisciplinary settings.
- 10. Communication:** Communicate effectively on complex engineering activities with the engineering community and with society at large, such as, being able to comprehend and write effective reports and design documentation, make effective presentations, and give and receive clear instructions.
- 11. Project management and finance:** Demonstrate knowledge and understanding of the engineering and management principles and apply these to one's own work, as a member and leader in a team, to manage projects and in multidisciplinary environments.
- 12. Life-long learning:** Recognize the need for, and have the preparation and ability to engage in independent and life-long learning in the broadest context of technological change.

PROGRAM SPECIFIC OUTCOMES (PSOs)

PSO1: Exhibit proficiency in planning, implementing and evaluating team oriented-software programming solutions to specific business problems and society needs.

PSO2: Demonstrate professional skills in applying programming skills, competency and decision making capability through hands-on experiences.

PSO3: Apply logical thinking in analyzing complex real world problems, and use professional and ethical behaviors to provide proper solutions to those problems.

PSO4: Demonstrate the ability to work effectively as part of a team in applying technology to Business and personal situations.

PROGRAM EDUCATIONAL OBJECTIVES (PEOs)

PEO1: To mould students to exhibit top performance in higher education and research and to become a State-of –the-art technocrat.

PEO2: To impart the necessary background in Computer Science and Engineering by providing solid foundation in Mathematical, Science and Engineering fundamentals.

PEO3: To equip the students with the breadth of Computer Science and Engineering innovate novel solutions for the benefit of common man.

CO – PO and PSO MAPPING:

Course Outcomes	Programme Outcomes (PO)											PSO		
	1	2	3	4	5	6	7	8	9	10	11	1	2	3
CO 1	-	-	1	-	1	-	-	-	2	1	2	1	2	-
CO 2	-	-	-	-	1	-	-	-	2	1	2	1	2	-
CO 3	-	-	1	-	-	-	2	-	1	-	3	2	2	-
CO 4	1	-	-	-	1	-	-	-	2	-	3	1	2	-
CO 5	1	-	1	-	1	-	-	-	2	-	2	2	2	-

CY3763

SECURITY IN MOBILE EXPLOITATION LABORATORY

L T P C
0 0 3 1.5

OBJECTIVES:

- Learn how to analyze the structure of an Android APK file.
- Perform static code analysis on decompiled Android APK code
- Monitor and analyze network traffic from a mobile app.
- Reverse engineer Android applications to understand their functionality.
- Identify insecure coding practices in mobile apps.
- Automate security testing of mobile applications.

LIST OF EXPERIMENTS

1. Basic Android APK Analysis
2. Static Code Analysis
3. Dynamic Analysis Setup
4. Network Traffic Analysis
5. Reverse Engineering
6. Exploiting Vulnerable Components
7. Automating Mobile Malware Analysis
8. Secure Code Practices
9. Automated Testing of Mobile Apps
10. Building a Mobile App Security Scanner

TOTAL: 45 PERIODS

COURSE OUTCOMES:

At the end of the course, the student should be able to:

- Use Python to extract the contents of an APK file.
- Use tools like Androguard to decompile APKs. Use adb (Android DebugBridge) commands in Python to interact with an Android device.
- Set up a proxy (e.g., mitmproxy) to capture network traffic. Write a Python script to extract and analyze APKs for known malware indicators.
- Use Appium or Selenium with Python to automate UI testing.
- Integrate the lab exercises into a single Python tool. Create a user interface (CLI or GUI) for the tool.

LIST OF EQUIPMENTS FOR A BATCH OF 30 STUDENTS:

SOFTWARE: Kali Linux, Virtual Box, C / C++ / Java or equivalent compiler

HARDWARE: Standalone desktops – 30 Nos. (or) Server supporting 30 terminals or more.

INDEX

S.No	Name of the Experiments	Page No.
1.	Basic Android APK Analysis	1
2.	Static Code Analysis	4
3.	Dynamic Analysis Setup	7
4.	Network Traffic Analysis	10
5.	Reverse Engineering	13
6.	Exploiting Vulnerable Components	16
7.	Automating Mobile Malware Analysis	19
8.	Secure Code Practices	22
9.	Automated Testing of Mobile Apps	25
10.	Building a Mobile App Security Scanner	28

AIM:

To analyze the structure and contents of an Android Application Package (APK) file using APK analysis tools and understand the essential components required for Android application deployment and security assessment.

PRE-LAB DISCUSSION:

Android is an open-source mobile operating system developed by Google that powers billions of smartphones, tablets, wearable devices, and embedded systems worldwide. Android applications are distributed in the form of Android Application Package (APK) files. An APK file contains everything required to install and execute an application on an Android device, including the compiled application code, resources, assets, libraries, certificates, and configuration files.

An APK file is essentially a compressed archive similar to a ZIP file. It consists of several important files and directories such as AndroidManifest.xml, classes.dex, META-INF, assets, res, and lib. The AndroidManifest.xml file stores information about the application, including its package name, application components, requested permissions, minimum Android version, and other configuration details. The classes.dex file contains the compiled Dalvik bytecode that is executed by the Android Runtime (ART). The META-INF directory contains digital certificates and signature files used to verify the authenticity and integrity of the application.

APK analysis is one of the fundamental activities performed during mobile application security assessment. Before installing or executing an Android application, security professionals inspect the APK to understand its internal structure and identify potential security issues. APK analysis helps detect excessive permissions, insecure configurations, hardcoded credentials, embedded malware, hidden functionalities, vulnerable third-party libraries, and unauthorized modifications.

Two major approaches are used for APK analysis: Static Analysis and Dynamic Analysis. Static analysis involves examining the APK file without executing the application. It includes inspecting the manifest file, application resources, permissions, source code, certificates, and executable files. Dynamic analysis involves executing the application in a controlled environment to monitor its runtime behavior, including network communication, file access, memory usage, and interactions with the operating system.

Several tools are available for APK analysis. APKTool is widely used to decode APK resources and reconstruct application files into a readable format. ADB (Android Debug Bridge) provides communication between a computer and an Android device for installing applications, debugging, and file transfer. Python libraries such as zipfile and Androguard enable automated inspection of APK files, making security analysis faster and more efficient.

Understanding the APK structure forms the foundation for advanced topics such as reverse engineering, malware analysis, penetration testing, vulnerability assessment, and secure Android application development. This experiment introduces students to the basic concepts of APK analysis and familiarizes them with the tools used throughout the remaining laboratory exercises.

ALGORITHM:

Step 1: Start the Kali Linux operating system.

Step 2: Obtain the Android APK file for analysis.

Step 3: Create a working directory.

Step 4: Copy the APK file into the working directory.

Step 5: Decode the APK using APKTool.

Step 6: Open the decoded project folder.

Step 7: Examine the AndroidManifest.xml file.

Step 8: Identify the package name, permissions, activities, and application components.

Step 9: Execute the Python program to list the contents of the APK file.

Step 10: Analyze the extracted files.

Step 11: Stop.

PROGRAM:**PYTHON CODE:**

```
import zipfile

apk_file = "sample.apk"

try:
    with zipfile.ZipFile(apk_file, 'r') as apk:
        print("APK CONTENTS")
        print("-----")
        for file in apk.namelist():
            print(file)
except FileNotFoundError:
    print("APK file not found.")
```

LINUX COMMANDS:

```
mkdir APK_Analysis
cd APK_Analysis
cp ~/Downloads/sample.apk .
apktool d sample.apk
cd sample
ls
cat AndroidManifest.xml
```

OUTPUT**APK CONTENTS**

```
-----
AndroidManifest.xml
classes.dex
resources.arsc
META-INF/
assets/
lib/
res/
kotlin/
I: Using Apktool 2.9.x
I: Decoding AndroidManifest.xml
I: Loading resource table...
I: Decoding resources...
I: Copying assets...
I: Copying original files...
I: Decoding completed successfully.
```

RESULT:

Thus, the Android APK file was successfully analyzed using APKTool and Python. The internal structure of the APK, including the manifest file, executable code, resources, libraries, and certificates, was examined successfully.

AIM:

To perform static code analysis on an Android application by decompiling an APK file and examining its source code, manifest file, permissions, application components, and security vulnerabilities without executing the application.

PRE-LAB DISCUSSION:

Static Code Analysis is the process of examining the source code or compiled code of an application without actually executing it. It is one of the most important techniques used in mobile application security testing because it enables security analysts to understand an application's internal structure, identify vulnerabilities, and detect malicious functionality in a safe environment. Since the application is never executed during static analysis, there is no risk of triggering malicious behavior or affecting the analysis system.

Android applications are distributed as APK (Android Package Kit) files. These files contain the application's compiled code, resources, assets, certificates, and configuration files. During static analysis, the APK file is decompiled using specialized tools to recover readable source code and configuration information. Analysts then inspect these files to understand how the application works and whether it follows secure coding practices.

One of the primary objectives of static analysis is to inspect the AndroidManifest.xml file. This file contains valuable information about the application, including its package name, version number, requested permissions, application components such as activities, services, broadcast receivers, content providers, and the minimum Android SDK version required for execution. Applications requesting dangerous permissions such as camera access, microphone access, SMS reading, location tracking, or contact access require careful examination because they may introduce privacy or security risks.

Another important aspect of static analysis is examining the application's source code. Decompiled Java or Smali code allows analysts to identify insecure coding practices such as hardcoded passwords, API keys, authentication tokens, encryption keys, insecure cryptographic algorithms, SQL injection vulnerabilities, insecure WebView implementations, and improper certificate validation. Detecting these weaknesses early helps developers improve application security before deployment.

Several tools are commonly used for Android static analysis. APKTool is used to decode APK resources and reconstruct application files into a human-readable format. JADX converts Dalvik bytecode into Java source code, making it easier to understand the application's implementation. Androguard, a Python framework, provides automated APK analysis capabilities including permission extraction, certificate analysis, API inspection, and malware detection. These tools help security professionals perform efficient and comprehensive analysis of Android applications.

Static code analysis plays a significant role in secure software development, malware analysis, vulnerability assessment, penetration testing, digital forensics, and mobile application auditing. It provides valuable insights into the application's architecture and security posture without requiring execution on a physical device or emulator.

ALGORITHM:

Step 1: Start the Kali Linux operating system.

Step 2: Obtain the Android APK file for analysis.

Step 3: Decode the APK using APKTool.

Step 4: Open the APK using JADX.

Step 5: Examine the AndroidManifest.xml file.

Step 6: Identify the application package name.

Step 7: Analyze the requested permissions.

Step 8: Inspect the Java source code for insecure coding practices.

Step 9: Execute the Python program to extract APK information.

Step 10: Record the analysis results.

Step 11: Stop.

PROGRAM:**PYTHON CODE:**

```
from androguard.misc import AnalyzeAPK
apk, dex, analysis = AnalyzeAPK("sample.apk")
print("Package Name :", apk.get_package())
print("Application Name :", apk.get_app_name())
print("Version Name :", apk.get_androidversion_name())
print("Version Code :", apk.get_androidversion_code())
print("\nPermissions")
for permission in apk.get_permissions():
    print(permission)
```

LINUX COMMANDS

```
apktool d sample.apk
```

```
jadx-gui sample.apk
```

```
cat sample/AndroidManifest.xml
```

OUTPUT

Package Name : com.example.demo

Application Name : Demo Application

Version Name : 1.0

Version Code : 1

Permissions

android.permission.INTERNET

android.permission.ACCESS_NETWORK_STATE

android.permission.CAMERA

android.permission.READ_EXTERNAL_STORAGE

I: Using Apktool 2.9.x

I: Loading resource table...

I: Decoding AndroidManifest.xml

I: Decoding resources...

I: Copying assets...

I: Copying original files...

I: Copying META-INF...

I: Copying lib...

I: Decoding completed successfully.

RESULT:

Thus, the Android APK was successfully decompiled and analyzed using static code analysis techniques. The application package information, permissions, source code, and configuration files were examined without executing the application.

AIM:

To set up a dynamic analysis environment for Android applications using an Android Emulator, Android Debug Bridge (ADB), and Frida to observe the runtime behavior of mobile applications.

PRE-LAB DISCUSSION:

Dynamic Analysis is the process of examining an application's behavior while it is executing. Unlike static analysis, which inspects the application's code without running it, dynamic analysis allows security analysts to monitor how an application behaves in real time. This technique helps identify runtime vulnerabilities, malicious activities, insecure network communications, file operations, API calls, memory modifications, and interactions with the Android operating system.

Dynamic analysis plays an important role in Android application security because many malicious applications conceal their behavior until execution. Malware may encrypt its payload, download additional code from remote servers, detect virtual environments, or perform malicious actions only after specific user interactions. These behaviors cannot always be detected through static analysis alone. Therefore, executing the application inside a secure testing environment provides valuable insights into its actual functionality.

To safely analyze Android applications, security professionals use an isolated testing environment consisting of an Android Emulator or a physical test device. The Android Emulator simulates a real Android device, allowing applications to be installed and executed without affecting production systems. It enables analysts to monitor application behavior while preventing potential malware from compromising personal devices.

Communication between the host computer and the Android device is established using Android Debug Bridge (ADB). ADB is a command-line utility included with the Android SDK Platform Tools. It allows developers and security analysts to install applications, execute shell commands, capture logs, transfer files, inspect running processes, and debug Android applications remotely. ADB serves as the primary interface for interacting with Android devices during security analysis.

Another essential tool for dynamic analysis is Frida. Frida is an open-source dynamic instrumentation framework that enables analysts to inject JavaScript code into running applications. Using Frida, security professionals can monitor function calls, intercept API requests, bypass SSL pinning, inspect memory contents, modify application behavior, and analyze sensitive operations without modifying the original APK. Frida is widely used in penetration testing, malware analysis, and reverse engineering because it provides powerful runtime inspection capabilities.

A properly configured dynamic analysis environment consists of Kali Linux, Android Studio Emulator, Android SDK Platform Tools, ADB, Python, Frida Server, and Frida Tools. Together, these components provide a secure platform for monitoring Android applications and understanding their runtime behavior. This experiment focuses on configuring the analysis environment and verifying communication between the host machine and the Android emulator.

ALGORITHM:

Step 1: Start the Kali Linux operating system.

Step 2: Launch the Android Emulator.

Step 3: Open the terminal window.

Step 4: Verify that ADB is installed.

Step 5: Start the ADB server.

Step 6: Connect to the Android Emulator.

Step 7: Verify the connected device.

Step 8: Install Frida Tools using Python.

Step 9: Verify the Frida installation.

Step 10: Test communication between the host machine and the Android device.

Step 11: Stop.

PROGRAM:**PYTHON CODE:**

```
import subprocess
print("Connected Android Devices")
print("-----")

subprocess.run(["adb", "devices"])

print("\nAndroid Version")
print("-----")

subprocess.run(["adb", "shell", "getprop", "ro.build.version.release"])

print("\nDevice Model")
print("-----")

subprocess.run(["adb", "shell", "getprop", "ro.product.model"])
```

LINUX COMMANDS:

adb version

adb start-server

adb devices

```
adb shell
```

```
adb install sample.apk
```

```
pip install frida-tools
```

```
frida --version
```

```
adb shell getprop ro.build.version.release
```

```
adb shell getprop ro.product.model
```

OUTPUT

```
Android Debug Bridge version 1.0.41
```

```
Version 35.0.2
```

```
List of devices attached
```

```
emulator-5554    device
```

```
Android Version
```

```
14
```

```
Device Model
```

```
Android SDK built for x86
```

```
Frida 16.3.3
```

RESULT:

Thus, the dynamic analysis environment was successfully configured using the Android Emulator, Android Debug Bridge (ADB), Python, and Frida. Communication between the host system and the Android device was established successfully, enabling runtime analysis of Android applications.

AIM:

To capture, monitor, and analyze the network traffic generated by an Android application using mitmproxy and identify HTTP/HTTPS requests, responses, and sensitive information exchanged between the mobile application and the server.

THEORY:

Network Traffic Analysis is one of the most important techniques used in mobile application security testing. It involves monitoring and inspecting the data exchanged between an Android application and remote servers during execution. Most Android applications communicate with web servers to authenticate users, retrieve data, upload files, synchronize information, or access cloud services. Monitoring this communication helps security analysts understand how an application exchanges sensitive information and whether appropriate security mechanisms are implemented.

Modern mobile applications primarily communicate using HTTP and HTTPS protocols. While HTTP transmits information in plain text, HTTPS uses Secure Socket Layer (SSL) or Transport Layer Security (TLS) protocols to encrypt communication between the client and the server. Although HTTPS provides confidentiality, security analysts often use proxy tools to inspect encrypted traffic in controlled environments for vulnerability assessment and penetration testing.

A proxy server acts as an intermediary between the mobile application and the destination server. By routing application traffic through the proxy, analysts can capture every request and response generated by the application. This enables them to inspect request headers, response headers, authentication tokens, cookies, API endpoints, JSON data, file uploads, and other information exchanged during communication.

mitmproxy is an open-source interactive HTTPS proxy widely used for mobile application security testing. It supports HTTP, HTTPS, HTTP/2, WebSocket, and TCP protocols. mitmproxy allows analysts to intercept, inspect, modify, replay, and save network requests in real time. It also provides scripting support using Python, making it highly suitable for automated traffic analysis and penetration testing.

During network traffic analysis, security professionals verify whether sensitive information such as usernames, passwords, authentication tokens, session identifiers, API keys, personal information, and financial data are transmitted securely. They also examine certificate validation, SSL pinning implementation, encryption mechanisms, and communication with third-party services. Applications transmitting sensitive information in plain text or using weak encryption methods are considered vulnerable to Man-in-the-Middle (MITM) attacks.

Android applications frequently communicate using RESTful APIs. These APIs exchange data in JSON or XML format over HTTP or HTTPS. Capturing these requests enables analysts to understand backend communication, identify exposed endpoints, and verify proper authentication and authorization mechanisms.

Network traffic analysis is widely used during penetration testing, malware analysis, vulnerability assessment, secure software development, and digital forensics. Understanding application network behavior enables security professionals to detect insecure communication channels, unauthorized data

transmission, information leakage, and malicious network activities. This experiment demonstrates how mitmproxy can be used to capture and analyze network traffic generated by an Android application..

ALGORITHM:

Step 1: Start the Kali Linux operating system.

Step 2: Launch the Android Emulator.

Step 3: Start the mitmproxy server.

Step 4: Configure the Android device to use the proxy server.

Step 5: Install the mitmproxy CA certificate on the Android device.

Step 6: Launch the Android application.

Step 7: Perform application operations that generate network requests.

Step 8: Observe the captured HTTP and HTTPS requests in mitmproxy.

Step 9: Analyze request headers, response headers, cookies, JSON data, and API endpoints.

Step 10: Save the captured traffic for further analysis.

Step 11: Stop.

PROGRAM:

PYTHON CODE:

```
from mitmproxy import http
```

```
def request(flow: http.HTTPFlow):
```

```
    print("=====")
    print("Request URL :", flow.request.pretty_url)
    print("Request Method :", flow.request.method)
    print("Client IP :", flow.client_conn.address)
    print("=====")
```

```
def response(flow: http.HTTPFlow):
```

```
    print("Status Code :", flow.response.status_code)
    print("Content Type :", flow.response.headers.get("Content-Type"))
```

LINUX COMMANDS:

```
mitmproxy
```

```
mitmweb
```

```
adb devices
```

```
adb shell settings get global http_proxy
```

```
adb shell settings put global http_proxy 10.0.2.2:8080
```

```
adb shell settings get global http_proxy
```

OUTPUT

Proxy Started Successfully

Listening at

http://*:8080

=====

Request URL :

<https://api.example.com/login>

Request Method :

POST

Client IP :

127.0.0.1

Status Code : 200

Content Type : application/json

POST /login HTTP/1.1

Host: api.example.com

User-Agent: Android

Content-Type: application/json

Response

HTTP/1.1 200 OK

RESULT:

Thus, the network traffic generated by the Android application was successfully captured and analyzed using mitmproxy. The HTTP/HTTPS requests, responses, API endpoints, and communication details between the application and the server were monitored successfully.

AIM:

To perform reverse engineering on an Android application using APKTool and JADX to understand its internal architecture, source code, application components, and security implementation.

PRE-LAB DISCUSSION:

Reverse Engineering is the process of analyzing a software application to understand its internal design, functionality, and implementation without having access to its original source code. In Android security, reverse engineering is widely used by security researchers, penetration testers, malware analysts, and forensic investigators to inspect application behavior, discover vulnerabilities, and identify malicious functionality. It is also used by software developers to verify application security, recover lost code, and evaluate third-party applications.

Android applications are distributed as APK (Android Package Kit) files containing compiled Dalvik Executable (DEX) bytecode rather than the original Java or Kotlin source code. During reverse engineering, the APK file is decompiled into a human-readable format using specialized tools. Decompiled source code allows analysts to understand application logic, inspect implemented algorithms, identify sensitive API calls, examine authentication mechanisms, and verify secure coding practices.

One of the primary tools used for Android reverse engineering is APKTool, which decodes APK resources and reconstructs the application directory structure. APKTool extracts important files such as AndroidManifest.xml, application resources, assets, and Smali code. Smali is an assembly-like language representing Dalvik bytecode and provides a low-level view of application execution.

Another important tool is JADX, which converts Dalvik bytecode stored in the classes.dex file into readable Java source code. Although the decompiled code may not exactly match the original source, it provides a clear understanding of application functionality and business logic. Analysts can examine activities, services, API calls, database operations, encryption methods, authentication procedures, and file handling routines using the decompiled source code.

Reverse engineering also helps identify security weaknesses such as hardcoded passwords, API keys, database credentials, authentication tokens, insecure cryptographic implementations, weak certificate validation, improper permission handling, and vulnerable third-party libraries. Understanding these weaknesses enables developers to improve application security and helps security professionals perform comprehensive vulnerability assessments.

In malware analysis, reverse engineering is used to understand malicious payloads, command-and-control communication, data exfiltration techniques, privilege escalation methods, and persistence mechanisms. Analysts examine suspicious code segments, encrypted strings, network endpoints, and hidden functionalities to determine whether an application exhibits malicious behavior.

Reverse engineering should always be performed responsibly and only for educational purposes, authorized penetration testing, malware research, or software security assessment. Unauthorized reverse engineering of copyrighted or proprietary software may violate legal and ethical guidelines. This

experiment introduces students to Android reverse engineering using industry-standard tools and demonstrates how application logic can be inspected without executing the application.

ALGORITHM:

Step 1: Start the Kali Linux operating system.

Step 2: Obtain the Android APK file.

Step 3: Decode the APK using APKTool.

Step 4: Open the APK using JADX.

Step 5: Examine the AndroidManifest.xml file.

Step 6: Analyze the application package structure.

Step 7: Inspect the decompiled Java source code.

Step 8: Identify application activities, services, and API calls.

Step 9: Search for hardcoded credentials or sensitive information.

Step 10: Record the observations.

Step 11: Stop.

PROGRAM:

PYTHON CODE:

```
from androguard.misc import AnalyzeAPK
apk, dex, analysis = AnalyzeAPK("sample.apk")
print("Application Activities")
print("-----")
for activity in apk.get_activities():
    print(activity)
print("\nApplication Services")
print("-----")
for service in apk.get_services():
    print(service)
print("\nBroadcast Receivers")
```

```
print("-----")

for receiver in apk.get_receivers():

    print(receiver)
```

LINUX COMMANDS:

```
apktool d sample.apk
```

```
jadx-gui sample.apk
```

```
grep -R "password" sample/
```

```
grep -R "apikey" sample/
```

```
grep -R "token" sample/
```

```
find sample/
```

OUTPUT

```
Application Activities
-----
com.example.demo.MainActivity
com.example.demo.LoginActivity
com.example.demo.SettingsActivity
Application Services
-----
com.example.demo.SyncService
Broadcast Receivers
-----
com.example.demo.NetworkReceiver
Hardcoded API Key Found
AlzaSy*****
Decompiled Source Generated Successfully.
AndroidManifest.xml Loaded Successfully.
Reverse Engineering Completed Successfully.
```

RESULT:

Thus, the Android application was successfully reverse engineered using APKTool, JADX, and Python. The application structure, source code, activities, services, broadcast receivers, and security-sensitive information were successfully analyzed.

AIM:

To identify and analyze vulnerable Android application components such as exported Activities, Services, Broadcast Receivers, and Content Providers, and verify whether they can be exploited through Android Debug Bridge (ADB).

PRE-LAB DISCUSSION:

Android applications are composed of several fundamental components that work together to provide application functionality. These components include Activities, Services, Broadcast Receivers, and Content Providers. They communicate with each other using Intents, allowing applications to exchange information and interact with system resources. While these components enable flexible application design, improper configuration may expose security vulnerabilities that attackers can exploit.

An Activity represents a single user interface screen within an Android application. Activities that are declared as exported can be launched by external applications or users through explicit intents. If an exported activity performs privileged operations without proper authentication or authorization, attackers may bypass application logic and gain unauthorized access to sensitive features.

A Service is a background component that performs long-running tasks without user interaction. Exported services may expose privileged functionality if they accept commands from unauthorized applications. Improper validation of incoming requests can lead to unauthorized execution of sensitive operations such as file access, network communication, or database manipulation.

A Broadcast Receiver listens for system or application-generated broadcast messages. Attackers may send forged broadcast intents to exported receivers if they are not protected by permissions. This may trigger unintended application behavior, privilege escalation, or execution of sensitive code.

A Content Provider manages structured data shared between applications. Improperly secured content providers may expose sensitive user information such as contacts, messages, application databases, or configuration files. Security analysts verify whether exported content providers enforce appropriate permissions before allowing access to confidential data.

Android provides the `android:exported` attribute to determine whether an application component can be accessed by external applications. Components marked as `android:exported="true"` require careful security validation because they become accessible outside the application. If authentication and authorization mechanisms are absent, attackers may exploit these components using ADB commands or malicious applications.

Android Debug Bridge (ADB) is commonly used during penetration testing to launch exported activities, start services, send broadcast intents, and query content providers. Security analysts use ADB to verify whether exported components expose unintended functionality or allow unauthorized access.

Identifying vulnerable application components is an essential part of mobile penetration testing and secure application development. Proper permission enforcement, input validation, and component protection significantly reduce the attack surface of Android applications. This experiment demonstrates how exported components can be identified and tested using Android security tools.

ALGORITHM:

Step 1: Start the Kali Linux operating system.

Step 2: Launch the Android Emulator.

Step 3: Install the target APK on the emulator.

Step 4: Decode the APK using APKTool.

Step 5: Examine the AndroidManifest.xml file.

Step 6: Identify exported Activities, Services, Broadcast Receivers, and Content Providers.

Step 7: Use ADB commands to launch exported components.

Step 8: Observe the application response.

Step 9: Record the identified vulnerabilities.

Step 10: Stop.

PROGRAM:**PYTHON CODE:**

```
from androguard.misc import AnalyzeAPK

apk, dex, analysis = AnalyzeAPK("sample.apk")

print("Exported Activities")
print("-----")

for activity in apk.get_activities():
    print(activity)

print("\nExported Services")
print("-----")

for service in apk.get_services():
    print(service)

print("\nBroadcast Receivers")
print("-----")
```

```
for receiver in apk.get_receivers():
```

```
    print(receiver)
```

```
print("\nContent Providers")
```

```
print("-----")
```

```
for provider in apk.get_providers():
```

```
    print(provider)
```

LINUX COMMANDS:

```
adb devices
```

```
adb install sample.apk
```

```
adb shell am start -n com.example.demo/.MainActivity
```

```
adb shell am startservice -n com.example.demo/.SyncService
```

```
adb shell am broadcast -a com.example.demo.TEST
```

```
adb shell dumpsys package com.example.demo
```

OUTPUT

```
Exported Activities
```

```
-----
```

```
com.example.demo.MainActivity
```

```
com.example.demo.SettingsActivity
```

```
Exported Services
```

```
-----
```

```
com.example.demo.SyncService
```

```
Broadcast Receivers
```

```
-----
```

```
com.example.demo.NetworkReceiver
```

```
Content Providers
```

```
-----
```

```
com.example.demo.DatabaseProvider
```

```
Starting: Intent
```

```
Status: OK
```

```
Activity launched successfully.
```

RESULT:

Thus, the vulnerable Android application components were successfully identified and analyzed. Exported Activities, Services, Broadcast Receivers, and Content Providers were examined using Python and ADB commands, demonstrating how improperly configured components may expose security vulnerabilities during mobile application security assessment.

AIM:

To develop a Python-based automation tool for analyzing Android APK files and identifying potential malware indicators such as dangerous permissions, suspicious API calls, embedded URLs, and malicious strings.

PRE-LAB DISCUSSION:

Mobile malware has become one of the fastest-growing cybersecurity threats due to the widespread use of smartphones and mobile applications. Attackers continuously develop malicious Android applications that steal sensitive information, monitor user activities, encrypt files, send premium SMS messages, perform banking fraud, or establish remote access to compromised devices. Detecting such applications manually is a time-consuming process; therefore, automated malware analysis has become an essential part of modern mobile security.

Mobile malware analysis is the process of examining Android applications to determine whether they exhibit malicious behavior. The analysis may involve inspecting permissions, application components, source code, embedded resources, network communication, cryptographic functions, and suspicious strings. Automation significantly reduces analysis time by performing repetitive tasks without human intervention and generating reports for security analysts.

Python is widely used for malware analysis because of its simplicity, extensive libraries, and automation capabilities. Python libraries such as Androguard provide APIs for extracting APK metadata, permissions, certificates, package information, and application components. Regular expressions (Regex) can be used to search for suspicious URLs, IP addresses, email addresses, API keys, or hardcoded credentials inside application files.

One of the primary indicators of malicious applications is the excessive use of dangerous Android permissions. Permissions such as READ_SMS, SEND_SMS, READ_CONTACTS, RECORD_AUDIO, ACCESS_FINE_LOCATION, READ_CALL_LOG, and SYSTEM_ALERT_WINDOW may indicate potentially malicious behavior if they are unnecessary for the application's intended functionality. Automated tools compare requested permissions against known dangerous permissions to identify suspicious applications.

Another important malware indicator is the presence of suspicious API calls or embedded URLs. Malware frequently communicates with remote command-and-control (C2) servers to receive commands or transmit stolen information. Detecting HTTP or HTTPS URLs, IP addresses, suspicious domains, shell commands, encryption routines, and dynamic code loading techniques helps analysts understand the application's behavior.

Automated malware analysis is widely used in antivirus software, mobile security solutions, malware research laboratories, and Security Operations Centers (SOC). It enables analysts to quickly prioritize suspicious applications for detailed investigation and improves the efficiency of large-scale malware detection systems. In this experiment, Python is used to automate APK inspection and identify common malware indicators from Android applications.

ALGORITHM:

Step 1: Start the Kali Linux operating system.

Step 2: Obtain the Android APK file.

Step 3: Import the required Python libraries.

Step 4: Load the APK using Androguard.

Step 5: Extract package information and requested permissions.

Step 6: Compare permissions with the predefined list of dangerous permissions.

Step 7: Display suspicious permissions detected.

Step 8: Generate the malware analysis report.

Step 9: Save the analysis results.

Step 10: Stop.

PROGRAM:**PYTHON CODE:**

```
from androguard.misc import AnalyzeAPK

dangerous_permissions = [

    "android.permission.READ_SMS",

    "android.permission.SEND_SMS",

    "android.permission.READ_CONTACTS",

    "android.permission.RECORD_AUDIO",

    "android.permission.ACCESS_FINE_LOCATION",

    "android.permission.READ_CALL_LOG"

]

apk, dex, analysis = AnalyzeAPK("sample.apk")

print("Package Name :", apk.get_package())

print("\nRequested Permissions")

print("-----")
```

```

permissions = apk.get_permissions()

for permission in permissions:

    print(permission)

print("\nSuspicious Permissions")

print("-----")

for permission in permissions:

    if permission in dangerous_permissions:

        print(permission)

```

LINUX COMMANDS:

```
python3 malware_analysis.py
```

```
apktool d sample.apk
```

```
jadx-gui sample.apk
```

```
adb install sample.apk
```

OUTPUT

Package Name :

com.example.demo

Requested Permissions

android.permission.INTERNET

android.permission.ACCESS_NETWORK_STATE

android.permission.READ_SMS

android.permission.RECORD_AUDIO

android.permission.READ_CONTACTS

Suspicious Permissions

android.permission.READ_SMS

android.permission.RECORD_AUDIO

android.permission.READ_CONTACTS

Malware Analysis Completed Successfully.

RESULT:

Thus, the Android APK was successfully analyzed using Python and Androguard. The application permissions were extracted, dangerous permissions were identified, and a basic automated malware analysis report was generated successfully.

AIM:

To identify insecure coding practices in Android applications and implement secure coding techniques to protect sensitive information, validate user input, and improve overall application security.

PRE-LAB DISCUSSION:

Secure coding is the practice of developing software by following security principles that minimize vulnerabilities and protect applications from cyber attacks. In Android application development, secure coding is particularly important because mobile applications frequently process sensitive information such as user credentials, financial data, personal information, and authentication tokens. Failure to implement secure coding practices may expose applications to attacks such as data theft, privilege escalation, code injection, insecure data storage, and unauthorized access.

One of the most common security issues in Android applications is the use of hardcoded credentials. Developers sometimes embed API keys, passwords, encryption keys, database credentials, or authentication tokens directly within the application source code. During reverse engineering, attackers can easily extract these values and misuse them to gain unauthorized access to backend systems. Sensitive information should instead be stored securely using Android Keystore or retrieved securely from trusted servers.

Input validation is another critical aspect of secure coding. Mobile applications receive input from users, external applications, APIs, QR codes, sensors, and network services. Improper validation of input may lead to security vulnerabilities such as SQL Injection, Command Injection, Cross-Site Scripting (XSS), and Buffer Overflow attacks. Developers should always validate, sanitize, and verify user input before processing it.

Secure communication between mobile applications and servers is equally important. Applications should use HTTPS instead of HTTP to ensure encrypted communication using SSL/TLS protocols. Developers should validate server certificates properly and avoid disabling certificate verification, as improper implementation may expose applications to Man-in-the-Middle (MITM) attacks.

Android provides secure storage mechanisms such as Android Keystore and Encrypted SharedPreferences for protecting sensitive information. Storing passwords or confidential information in plain text files, SharedPreferences, or SQLite databases without encryption significantly increases the risk of data leakage if the device is compromised.

Developers should also follow the Principle of Least Privilege, requesting only the permissions required for application functionality. Excessive permissions unnecessarily increase the attack surface and may expose sensitive user information. Proper exception handling, logging, code obfuscation, and regular security testing further improve application resilience against attacks.

Secure coding practices form the foundation of secure software development and significantly reduce the likelihood of security vulnerabilities. This experiment demonstrates how Python can be used to identify insecure coding practices within Android applications by searching for hardcoded credentials and insecure URLs.

ALGORITHM:

Step 1: Start the Kali Linux operating system.

Step 2: Decode the Android APK using APKTool.

Step 3: Open the decoded project directory.

Step 4: Execute the Python program.

Step 5: Search the application source files for hardcoded passwords, API keys, authentication tokens, and HTTP URLs.

Step 6: Display the detected insecure coding practices.

Step 7: Record the identified security issues.

Step 8: Stop.

PROGRAM:**PYTHON CODE:**

```
import os

keywords = [
    "password",
    "apikey",
    "token",
    "secret",
    "http://"
]

directory = "sample"

print("Potential Security Issues")
print("-----")

for root, dirs, files in os.walk(directory):
    for file in files:
        path = os.path.join(root, file)
        try:
```

```
with open(path, "r", errors="ignore") as f:
    data = f.read().lower()
    for keyword in keywords:
        if keyword in data:
            print(keyword, "found in", path)
except:
    pass
```

LINUX COMMANDS:

```
apktool d sample.apk
python3 secure_code_check.py
grep -R "password" sample/
grep -R "apikey" sample/
grep -R "token" sample/
grep -R "http://" sample/
```

OUTPUT

Potential Security Issues

```
-----
password found in sample/smali/Login.smali
apikey found in sample/assets/config.txt
token found in sample/smali/Auth.smali
http:// found in sample/res/raw/config.xml
Security Scan Completed Successfully.
```

RESULT:

Thus, insecure coding practices such as hardcoded passwords, API keys, authentication tokens, and insecure HTTP URLs were successfully identified using Python and APK analysis techniques. The experiment demonstrated the importance of adopting secure coding practices to improve the security of Android applications.

AIM:

To automate the functional testing of an Android application using Appium and Python by performing UI interactions and validating the application's behavior.

PRE-LAB DISCUSSION:

Mobile application testing is an essential phase of the software development life cycle that ensures Android applications function correctly, securely, and efficiently across different devices and operating system versions. Manual testing of mobile applications is often repetitive, time-consuming, and prone to human error. Automation testing addresses these limitations by executing predefined test cases automatically, thereby improving testing speed, consistency, and accuracy.

Appium is an open-source mobile automation framework widely used for testing Android and iOS applications. It supports multiple programming languages such as Python, Java, JavaScript, and C#. Appium automates mobile applications by interacting with user interface elements in the same manner as a real user, making it suitable for functional testing, regression testing, smoke testing, and acceptance testing.

Appium follows a client-server architecture. The Appium Server receives automation commands from the test script and translates them into platform-specific commands using the UIAutomator2 driver for Android applications. The driver interacts directly with the Android device or emulator to perform user actions such as clicking buttons, entering text, scrolling screens, and validating application responses.

Before executing automation scripts, the testing environment must be configured with Android Studio, Android Emulator, Android SDK Platform Tools, Appium Server, Python, and the Appium Python Client library. Android Debug Bridge (ADB) establishes communication between the host computer and the Android device. Once the emulator is connected, Appium installs the application and executes the automation script without requiring any modifications to the application source code.

During automated testing, UI elements are identified using locators such as Accessibility ID, Resource ID, XPath, Class Name, or Android UI Automator. Resource ID is generally preferred because it provides better performance and reliability compared to XPath. Automation scripts verify application behavior by comparing expected results with actual outputs after performing various user interactions.

Automated mobile testing significantly reduces testing effort, improves software quality, and enables continuous integration and continuous deployment (CI/CD). It allows developers and testers to validate application functionality after every code change, ensuring early detection of defects and maintaining application stability. This experiment demonstrates how Appium and Python can be used to automate the testing of Android applications by launching an application, interacting with UI elements, and validating successful execution.

ALGORITHM:

- Step 1: Start the Kali Linux operating system.
- Step 2: Launch the Android Emulator.
- Step 3: Start the Appium Server.
- Step 4: Verify the connected Android device using ADB.
- Step 5: Configure the Appium Desired Capabilities.
- Step 6: Establish a connection with the Android Emulator.
- Step 7: Launch the Android application.
- Step 8: Locate UI elements and perform user interactions.
- Step 9: Validate the application response.
- Step 10: Close the application.
- Step 11: Stop.

PROGRAM:**PYTHON CODE:**

```
from appium import webdriver

from appium.options.android import UiAutomator2Options

from selenium.webdriver.common.by import By

import time

options = UiAutomator2Options()

options.platform_name = "Android"

options.device_name = "Android Emulator"

options.automation_name = "UiAutomator2"

options.app = "/home/kali/sample.apk"

driver = webdriver.Remote(

    "http://127.0.0.1:4723",

    options=options

)
```

```
time.sleep(5)

print("Application Launched Successfully")

driver.quit()
```

LINUX COMMANDS:

```
adb devices

appium

python3 mobile_test.py
```

OUTPUT

```
List of devices attached
emulator-5554    device
Appium REST http interface listener started
Server listening on
http://127.0.0.1:4723
Application Launched Successfully
Test Execution Completed Successfully.
```

RESULT:

Thus, the Android application was successfully automated using Appium and Python. The application was launched, the automation session was established, and the test execution completed successfully, demonstrating automated functional testing of a mobile application.

AIM:

To develop a Python-based Mobile App Security Scanner that automatically analyzes an Android APK file and generates a security report containing application information, permissions, exported components, and potential security vulnerabilities.

PRE-LAB DISCUSSION:

A Mobile App Security Scanner is an automated software tool used to evaluate the security posture of Android applications. It combines various security analysis techniques to inspect APK files, identify vulnerabilities, and generate comprehensive reports that assist security analysts and developers in improving application security. Modern security scanners integrate multiple analysis techniques such as APK inspection, permission analysis, static code analysis, reverse engineering, malware detection, and secure coding verification into a single automated solution.

During the software development life cycle, security testing should be integrated into every stage to identify vulnerabilities before deployment. Automated security scanners significantly reduce manual effort by performing repetitive security checks consistently and accurately. These tools enable organizations to analyze a large number of applications quickly while maintaining high analysis quality.

A typical mobile application security scanner begins by extracting APK metadata such as package name, version information, application name, certificates, and SDK versions. It then analyzes AndroidManifest.xml to identify requested permissions, exported Activities, Services, Broadcast Receivers, and Content Providers. Applications requesting unnecessary dangerous permissions or exposing critical components without adequate protection are flagged for further investigation.

The scanner also performs static analysis to identify hardcoded passwords, API keys, authentication tokens, insecure URLs, weak cryptographic implementations, and suspicious strings within the application source code. Additional checks may include identifying insecure network communication, improper certificate validation, outdated third-party libraries, and vulnerable coding practices. These findings help developers remediate vulnerabilities before the application is released.

Python is widely used to build security automation tools due to its extensive library support and simplicity. The Androguard framework provides APIs for extracting APK metadata, application components, permissions, and certificates. Additional Python modules such as os, re, and json can be used to automate report generation and perform customized security analysis. Combining these capabilities allows developers to build lightweight and efficient mobile application security scanners.

Automated security scanners are extensively used in vulnerability assessment, malware analysis, penetration testing, Security Operations Centers (SOC), DevSecOps pipelines, and secure software development. Integrating automated security testing into Continuous Integration and Continuous Deployment (CI/CD) pipelines ensures that security vulnerabilities are detected early, reducing the overall cost of software maintenance and improving application reliability.

This experiment integrates the concepts learned throughout the previous experiments and demonstrates how multiple security checks can be combined into a single Python-based Mobile App Security Scanner capable of generating a basic security assessment report for Android applications.

ALGORITHM:

Step 1: Start the Kali Linux operating system.

Step 2: Obtain the Android APK file.

Step 3: Import the required Python libraries.

Step 4: Load the APK using Androguard.

Step 5: Extract package information and application permissions.

Step 6: Identify exported application components.

Step 7: Compare permissions with predefined dangerous permissions.

Step 8: Generate the security report.

Step 9: Display the analysis results.

Step 10: Stop.

PROGRAM:**PYTHON CODE:**

```
from androguard.misc import AnalyzeAPK

dangerous_permissions = [

    "android.permission.READ_SMS",

    "android.permission.SEND_SMS",

    "android.permission.READ_CONTACTS",

    "android.permission.RECORD_AUDIO",

    "android.permission.ACCESS_FINE_LOCATION"

]

apk, dex, analysis = AnalyzeAPK("sample.apk")

print("\n===== MOBILE APP SECURITY REPORT =====\n")

print("Package Name :", apk.get_package())

print("Application Name :", apk.get_app_name())

print("Version :", apk.get_androidversion_name())
```

```

print("\nPermissions")
print("-----")
permissions = apk.get_permissions()
for permission in permissions:
    print(permission)
print("\nPotential Security Issues")
print("-----")
for permission in permissions:
    if permission in dangerous_permissions:
        print("Dangerous Permission :", permission)
print("\nActivities :", len(apk.get_activities()))
print("Services :", len(apk.get_services()))
print("Receivers :", len(apk.get_receivers()))
print("Content Providers :", len(apk.get_providers()))
print("\nSecurity Scan Completed Successfully.")

```

LINUX COMMANDS:

```

python3 security_scanner.py
apktool d sample.apk
jadx-gui sample.apk
adb install sample.apk

```

OUTPUT

```

===== MOBILE APP SECURITY REPORT =====

```

Package Name : com.example.demo

Application Name : Demo Application

Version : 1.0

Permissions

android.permission.INTERNET

android.permission.ACCESS_NETWORK_STATE

android.permission.READ_SMS

android.permission.RECORD_AUDIO

Potential Security Issues

Dangerous Permission :
android.permission.READ_SMS

Dangerous Permission :
android.permission.RECORD_AUDIO

Activities : 4

Services : 1

Receivers : 2

Content Providers : 1

Security Scan Completed Successfully.

RESULT:

Thus, a Python-based Mobile App Security Scanner was successfully developed to analyze Android APK files. The scanner extracted application information, identified permissions, detected potential security issues, analyzed application components, and generated a basic security report, demonstrating an integrated approach to Android mobile application security assessment.