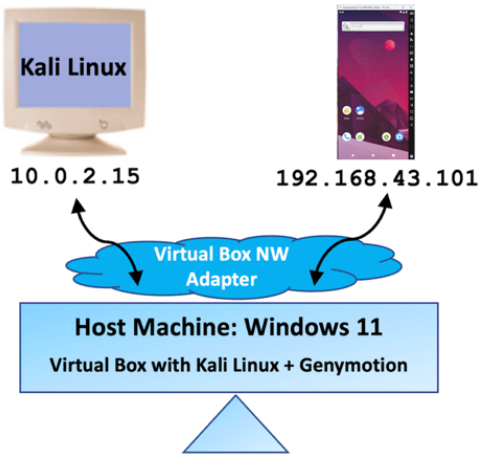
Department of Computer Science
FC College University

Department of Data Science
University of the Punjab

HO# 2.13 Android App Pen-Testing- II

Setting-up the Lab Environment



Reverse Engineering & Static Analysis

Setting up Lab Environment for Pen Testing Android Apps

Option 1: (Download/Install a Pre-built AVM from osboxes) https://www.osboxes.org/android-x86/

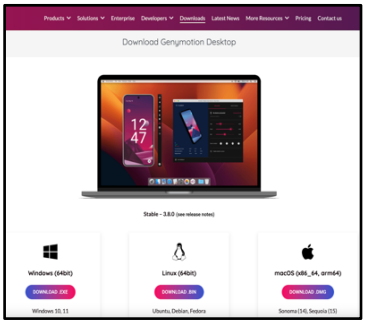
You can download a pre-built virtual machine image (`.vdi` file) of your choice from **osboxes**, install and run it inside Virtual Box or VMware in an isolated environment. Its limitation is that these `.vdi` files are typically x86/x64 builds, and ARM based Android images require additional emulation layer to run on emulators like Qemu.

Option 2: (Use AVD inside Android Studio) https://developer.android.com/studio/run/managing-avds

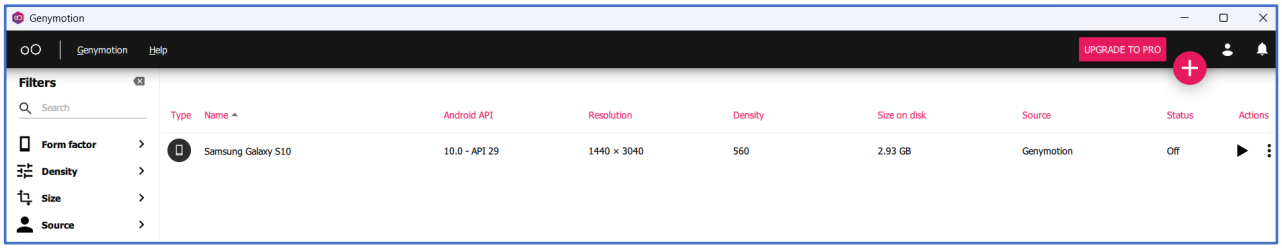
You can use *Android Studio Device Manager*, having a built-in *Android Emulator* that allows you to emulate different Android Virtual Devices (AVDs), each having different Android OS versions, screen sizes and so on for a controlled testing environment. This allows developers and security researchers to test apps or perform penetration testing without needing a physical Android device.

Option 3: (Use Genymotion) https://genymotion.com/product-desktop/

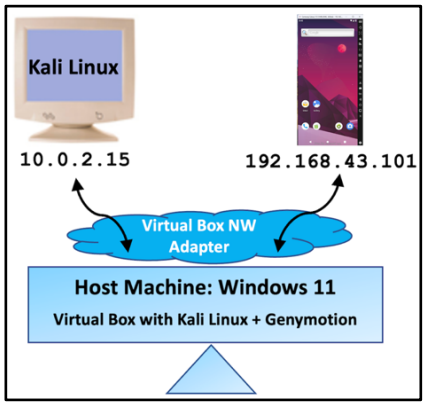
- *Genymotion* is a user-friendly Android emulator primarily used by developers and security researchers to simulate Android devices for app development, debugging, and pen-testing. It offers a variety of pre-configured virtual devices running different Android versions. Genymotion can run on desktop (Windows, macOS, Linux) or in the cloud, and it integrates well with tools like Android Studio and Android Debug Bridge (ADB). Unlike Android Emulator, it is known for its speed, performance, and ease of use, especially when testing on multiple device types w/o needing physical hardware.
- Download Genymotion for Personal Use for your host Windows-11 machine from above link. When you click Download, it will give you two options: *Genymotion + VirtualBox* and *Genymotion Desktop only*. Since we already have our VirtualBox installed, so we will select Genymotion Desktop only. Once you install it, do create an account and do select personal use for free.



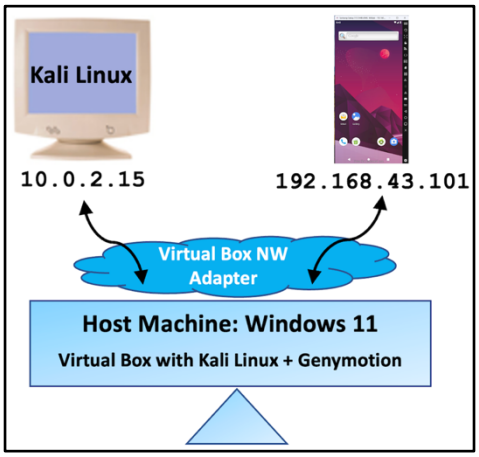
- Once installed, start Genymotion, and create a new virtual device of your choice (Samsung Galaxy S10), the Android version (Android 10.0) and other options like h/w resources, display options, android options. In the Hypervisor option do not forget to select the Network mode to NAT.
- Once done you will see a new virtual device inside the left panel of your Virtual Box. You may have to make a bit of NW settings for the Android virtual device inside your Virtual Box. It will have Adapter1 set to Host-only Adapter, let it be like that. You need to set the Adapter2 to the same NAT Network which your Kali Linux machine is using.



- Start the Galaxy S10 virtual machine from Virtual Box.
- Start Genymotion, and boot the virtual device (Samsung Galaxy S10). This may take a while and finally will display the UI of the Android device on your host Windows 11 OS.
- By default, Google Play Store is not installed on your virtual device. To install it click Open **GApps** icon on the right panel of your virtual device, install it, sign in on Google using some temporary account and then you can download and install apps from Play Store like chrome and others on this virtual device.
- Click Settings icon on device to check the IP of your phone: **`192.168.43.101:5555`**. Now from Kali Linux machine try pinging the Android device to check connectivity.



Android Debug Bridge (adb) and its Commands



ADB is a command-line tool (client-server program) that lets you communicate with Genymotion emulator, Android studio emulator as well as with physical Android device. It is mainly used for debugging apps, install/uninstall APKs, file transfers, accessing system logs, and execute shell commands directly on the device.

Overview of ADB Architecture:

- **ADB Client** (on Kali Linux)**:** It is a tool used to interact with the Android device by issuing commands from Kali Linux terminal to ADB server.
- **ADB Server** (on Kali Linux)**:** It is a server process that runs and listens on a local port usually 5037, waiting for connections from the ADB client. Its job is to manage the communication between the ADB client on Kali and the ADB daemon on Android device. When we issue a command via ADB client, it forwards the request to the ADB server running on Kali, which further forwards the request to ADB daemon.
- **ADB Daemon** (on Android emulator)**:** The ADB daemon (`adbd`), runs on every connected Android device (physical or virtual). It listens for incoming connections from the ADB server at port 5555 and processes commands from the client.

Hands-On Practice of adb Commands:

- I assume that both our virtual Kali Linux machine and the virtual Android device (Samsung Galaxy S10) are both up and accessible to each other. This can be verified by running the ping command from Kali Linux:
  ```
  $ ping 192.168.43.101
  ```
- To install `adb` on our Kali Linux machine using the following command:
  ```
  $ sudo apt install adb
  $ adb --version

  Android Debug Bridge version 1.0.41
  Running on Linux 6.8.11-amd64 (x86_64)
  ```
- After successfully installing `adb` on your Kali Linux machine, now let us start the ADB server, which will also run a listener daemon at tcp port 5037:
  ```
  $ adb kill-server
  $ adb start-server
  $ netstat -antlp | grep adb

  tcp    0   0    127.0.0.1:5037   0.0.0.0:*          LISTEN      <PID>/adb
  ```
  The output shows that ADB server is running at local host and listening for incoming requests at port 5037.
- Let us use the `adb connect` command and pass it the socket address of the virtual android device. The ADB client will forward this request to ADB server running on Kali, which will further send this request to the ADB Daemon running on Android device.
  ```
  $ adb connect 192.168.43.101:5555
  ```
- To check the connectivity, we can use the `adb devices` command as shown:
  ```
  $ adb devices
  List of devices attached
  192.168.43.101:5555      device        (in case of a virtual device you get the socket address)
  7f2a756529862a6b         device        (in case of a physical device you get the serial number)
  ```

- The connection can further be verified using the netstat command:

```
$ netstat -antlp | grep adb
tcp    0   0  127.0.0.1:5037     0.0.0.0:*              LISTEN       <PID>/adb
tcp    0   0  10.0.2.15:57590    192.168.43.101:5555    ESTABLISHED  <PID>/adb
```

- This time in the output of netstat, you can see one established connection, while the ADB server is still listening for incoming requests.
- Now, it is time to open an interactive shell from our Kali Linux machine to the connected emulated Android device. In case of multiple connected devices, you need to pass the Serial no of the device as well:

```
$ adb get-serialno
192.168.43.101:5555
$ adb [-s 192.168.43.101:5555] shell
vbox86p:/ #
```

- In case of a rooted device, you get a root prompt and have access to entire file system, otherwise, you may be sandboxed in some other directory. Remember regular apps can only access their own directories inside `/data/data/<package>/` and only `root` user can access/modify files in other directories. Even `root` may have restrictions on modifying some directories due to SELinux enforcement.
- By default, the `ls` command inside `adb` gives you a black/white output, which is difficult to differentiate between directories and files. So, you can add an alias to change this default settings as shown in the screenshot below (`echo $KSH_VERSION`):

```
vbox86p:/ # echo "alias ls='ls --color=auto'" >> /data/local/.bashrc
vbox86p:/ # source /data/local/.bashrc
vbox86p:/ # ls
acct        config      etc                 init.zygote32.rc  product          sys
apex        d           init                lost+found        product_services system
bin         data        init.environ.rc     mnt               sbin             tmp
bugreports  debug_ramdisk init.rc           odm               sdcard           ueventd.rc
cache       default.prop  init.usb.configfs.rc oem             sepolicy         vendor
charger     dev         init.usb.rc         proc              storage          vendor_service_contexts
vbox86p:/ #
```

To Do: A brief summary of Android File Hierarchy Standard and `adb` commands are given on next page. Please spend some time and make your hands dirty to have a practical understanding of the file system of the virtual Android device connected with your Kali Linux machine. Do remember that our virtual device is by default rooted so you can move around anywhere inside the file system. This will not be the case if you are connected to a non-rooted physical device. More on this later ☺

Summary of adb Commands

Commands	Description
`$ adb version`	Display the version of installed `adb` version
`$ adb --help`	Display help about different adb commands
`$ adb kill/start-server`	Stops/Start the `adb` server running on your Kali machine
`$ adb connect/disconnect 192.168.43.101:5555`	Connects/disconnect to a specific device over WiFi
`$ adb devices`	List connected devices
`$ adb push <Kali> <device>`	Copies a file from Kali Linux to the device
`$ adb pull <device> <kali>`	Copies a file from device to Kali Linux
`$ adb install <app-name>.apk`	Install an application
`$ adb uninstall <package-name>`	Uninstall an application
`$ adb logcat -v color`	Display logs of all apps. To get help about different options use `adb logcat -h`
The following commands should either be given from Android Korn Shell or use **`adb shell`** followed by command	
`$ adb shell` **`vbox86p:`**	Will spawn MIRBSD KSH for Android, from where you can give the Linux/Android commands directly without preceding it with the adb shell prefix.
`$ adb shell <cmd>`	Run individual commands like `ls, pwd, mkdir, cp, rm, mv, uname, ps, reboot` inside Android OS
`$ adb shell screencap /sdcard/screen.png`	Captures a screenshot and save it in a local file
`$ adb shell screenrecord /sdcard/video.mp4`	Record the screen and save it in a local file

`$ adb shell pm list packages`
`$ adb shell pm path <pkg-name>`
`$ adb shell pull <path-to-apk-file>`
	The package manager command is a versatile command used for many purposes. For example: `list packages` will list all the installed packages, `dump` will display package information like permissions, activities, services, etc.

`$ adb shell am start -n <pkg-name>/<activity>`	The activity manager command is also a versatile command used for many purposes like launching an app

Android File Hierarchy Standard

Directory	Description
`/data/`	Stores user-installed apps, app data, databases, and settings. Usually requires root to access.
`/data/data/`	Each application has its own directory inside `/data/data/`, named by the app's package name, e.g., `com.arif.login`. Used to store private data related to each app. To access it the device must be rooted.
`/data/app/`	This directory stores APK files for installed applications on the device.
`/system/`	Contains system apps, core files, libraries, and configurations that the system needs to function (similar to `/usr` in Linux).
`/system/app/`	This directory contains pre-installed system apps that are bundled with the Android OS.
`/mnt/`	Itis a general-purpose mount point for various file systems, including external storage devices, network file systems, and temporary storage. However, it is not the primary storage directory for user data.
`/sdcard/`	This is the primary location for user data (downloads, photos, and media) on Android devices. Despite the name, it doesn't necessarily refer to an external SD card, rather can also refer to a partition of internal storage. Remember this is world readable.
`/bin/`	Stores binaries and essential shell commands (like ls, cp, mv).
`/storage/`	Contains external storage devices (like SD cards or USB drives).
`/etc/`	Stores system configuration files (though most Android settings are in /system/etc/).
`/lib/`	Holds shared libraries (.so files) needed for system binaries (on some devices).
`/boot/`	Contains the bootloader and kernel (usually not visible in all devices).
`/init/`	This directory contains a soft link named `init` to `/system/bin/init` program (first process when Android boots).
`/proc/`	Virtual filesystem with system and process information (like /proc in Linux).
`/dev/`	Virtual directory for device files (e.g., input devices, sensors, cameras).
`/sys/`	Exposes hardware and kernel parameters (used for tuning system performance).
`/vendor/`	Holds vendor-specific drivers, firmware, and HALs (Hardware Abstraction Layers).

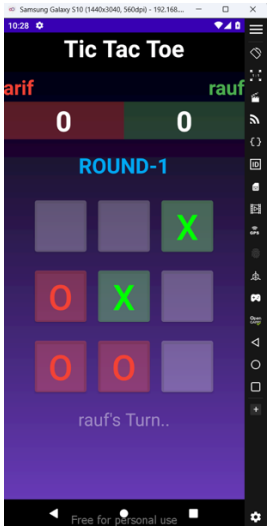
Installing/Uninstalling Android Apps on AVD via ADB

- You can try installing all the Android apps that you have built yourself in Android Studio (in HO#2.12), or you can download APKs from the following links and install them on your Android devices:
  - https://f-droid.org/
  - https://apkpure.com/
  - https://www.apkmirror.com/
  - https://androidapks.com/
- You can also download publicly available `apk` files along with their source codes from `github`:
  - https://github.com/payatu/diva-android
  - https://github.com/JimishKhokhar/Tic-Tac-Toe-APK/blob/master/app-debug.apk
- It is always recommended to download apps from Google Play Store. To do this on your AVD inside Genymotion, click **Open GApps** icon on the right panel of your AVD, install it, sign-in on Google using some temporary account and then you can download and install apps from Play Store on your AVD directly.

```
$ ls ~/IS/module2/2.13/apk-files/
helloworld.apk   login-app.apk   tictactoe.apk   diva-beta.apk   flappy-bird.apk
```

- Connect your Kali machine with the AVD running inside Genymotion, and let me install the famous game of my childhood `tictactoe`

```
$ adb start-server
$ adb connect 192.168.43.101:5555
$ adb install tictactoe.apk
Performing Streamed Install
Success
```



- Remember to the `adb install` command, we need to give the `apk` name, while to the `adb uninstall` command, we need to give the package name. From an apk file, you can check the package name using the Android Asset Packaging Tool (`aapt`), that shows the app's metadata, permissions, resources, and package information. For already installed applications, you can view the contents of the `/data/data/` subdirectory, having one directory each, named by the app's package name, that is used to store private data related to each app, or use the package manager command.

```
$ aapt dump permissions| badging|resources|strings tictactoe.apk
package: com.example.tictactoe

$ adb shell ls /data/data/
$ adb shell pm list packages | grep tictactoe
```

- Once installed, you can run the app by either clicking its icon or directly from `adb` by giving the following command:

```
$ adb shell am start -n com.example.tictqctoe/.MainActivity
Starting: Intent {cmp=com.example.tictactoe/.MainActivity}
```

Play the game & break your childhood records ☺

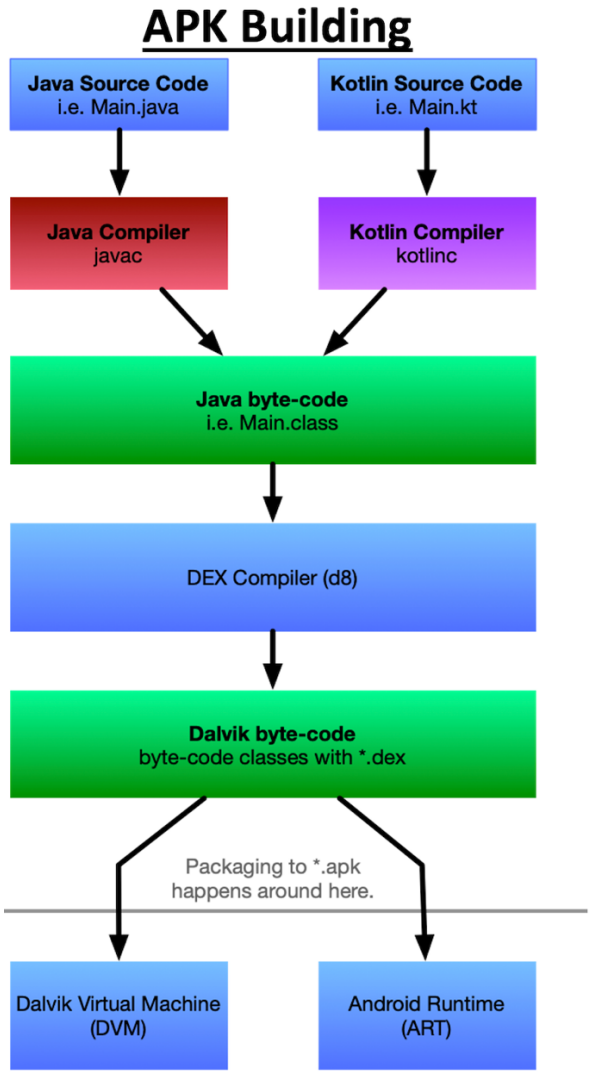
- Once done, you may like to uninstall the app and verify:

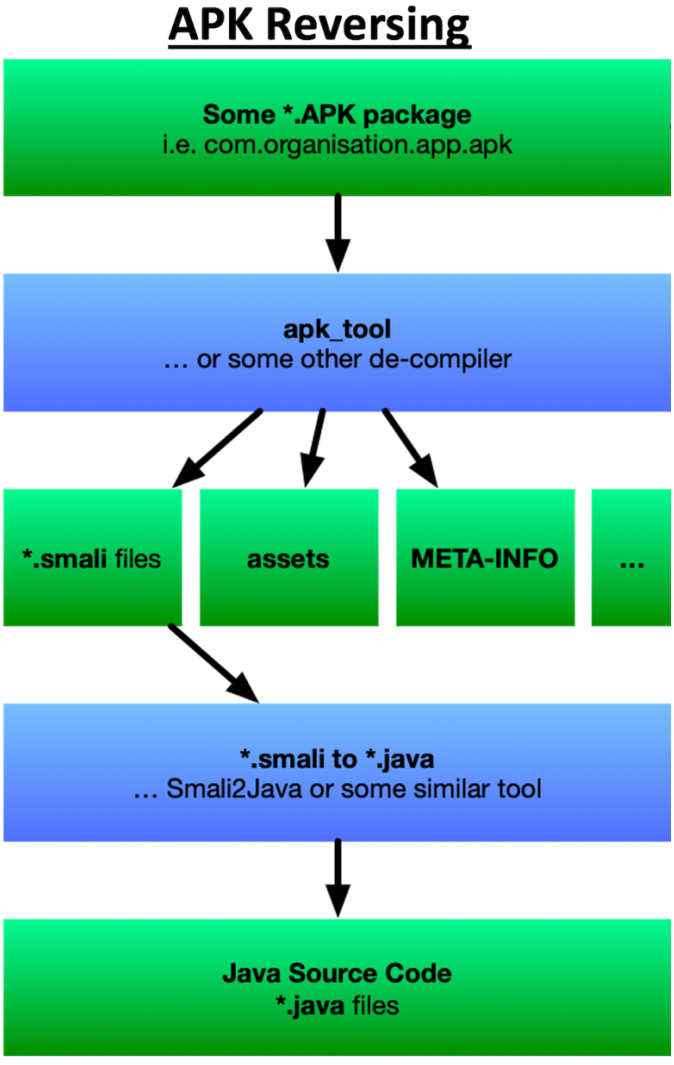
```
$ adb uninstall com.example.tictactoe
Success
```

- To verify that the app is successfully uninstalled from the AVD:

```
$ adb shell ls /data/data/ | grep tictactoe
$ adb shell pm list packages | grep tictactoe
```

Reverse Engineering and Static Analysis of apk Files





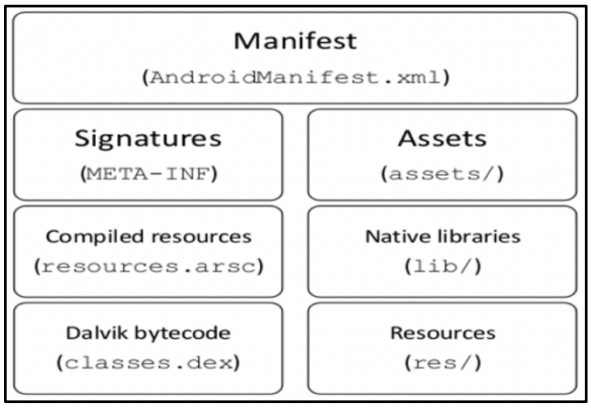
Reverse Engineering Android Apps

Reverse engineering in cybersecurity refers to the process of analyzing and disassembling binaries to understand their inner working, with the goal of extracting valuable information, identifying vulnerabilities, and malicious behaviour.

- **Static analysis** involves decompiling, reverse engineering and analyzing the APK file without executing it. It allows you to understand the app's structure, code, resources, and potential vulnerabilities. Some static analysis tools are `apktool`, `jadx`, `dex2jar`, `jd-gui` and `MobSF`. If the `apk` is obfuscated (using tools like `R8`, `Proguard`), you may need to de-obfuscate using tools like `Zguard` and `Procyon`.
- **Dynamic analysis** involves running the APK and observing its runtime behaviour, which includes network traffic, memory manipulation, API calls etc. Some dynamic analysis tools are `Frida`, `Drozer`, `Wireshark` and `BurpSuite`.

Unzipping APK, & Decompiling using `d2j-dex2jar`, `jd-gui`

By now, we all know that an APK is the package format used by Android to distribute & install applications. It is a compressed archive (similar to .zip) that contains all the necessary files for an Android app, including the compiled code (DEX files), resources, assets, and manifest file. Let us use the `unzip` utility to extract files. Without any option, the `unzip` command will `unzip` the file in the `pwd`, however, you can use the `-d` option to the `unzip` command.




```
$ unzip diva-beta.apk -d diva-unzip
$ ls diva-unzip
AndroidManifest.xml  classes.dex  resources.arsc  /res  /lib  META-INF/
```

A brief description of some important files/directories is given below:

- **`AndroidManifest.xml`:** Every APK file includes an `AndroidManifest.xml` file that describes the essential information required by the Android build tools, Android OS and Google Play. It contains the application's package name, permissions, activities, services, content providers, intents and metadata/configurations the Android device needs to run the app. When you unzip an `apk` file it appears to be in a binary format, so you can't view its contents ☹.
- **`classes.dex`:** This is a **D**alvik **EX**ecutable file, which is compiled Java/Kotlin code. Some large apps may have multiple `classesX.dex` files (e.g., `classes2.dex`, `classes3.dex`). These files are executed by the Android Runtime (ART) or Dalvik VM.
- **`resources.arsc`:** This will also be a binary file that contains precompiled resources like strings, colors, and layouts in binary format. It is used by Android to efficiently load resources.
- **`res/`:** This directory contains uncompiled raw resources in different sub-directories containing app icons, images, audio, video, xml files (defining strings, colors, styles) and so on.
- **`lib/`:** This directory contains compiled native libraries (`.so` files) for different CPU architectures like `x86`, `x86_64`, `mips`, `mips64`, `arm64-v8a`, and so on.
- **`assets/`:** This directory contains additional app files like fonts, JSON files, SQLite databases, and so on.
- **`META-INF/`:** This directory contains signature related files used to verify the integrity and authenticity of the app.
  - `MANIFEST.MF` is a text file containing the app's files and their checksums, to check if any file is changed after build.
  - `CERT.SF` is a text file containing finger-prints of lines inside `.MF` file to ensure that `.MF` is not tempered.
  - `CERT.RSA` file contains digital signature that authenticate the signer (`keytool -printcert -file CERT.RSA`)

Convert `.dex` files to `.jar` file using `d2j-dex2jar`:

- The `d2j-dex2jar` (part of `dex2jar` toolset) is a tool that is passed `classes.dex` file and it will convert the Android DEX (Dalvik Executable) files into Java bytecode `classes-dex2jar.jar` file. We already unzipped the `diva-beta.apk` file, which contains the `classes.dex` file:

```
$ ls
AndroidManifest.xml  classes.dex  resources.arsc  /res  /lib  META-INF/
```

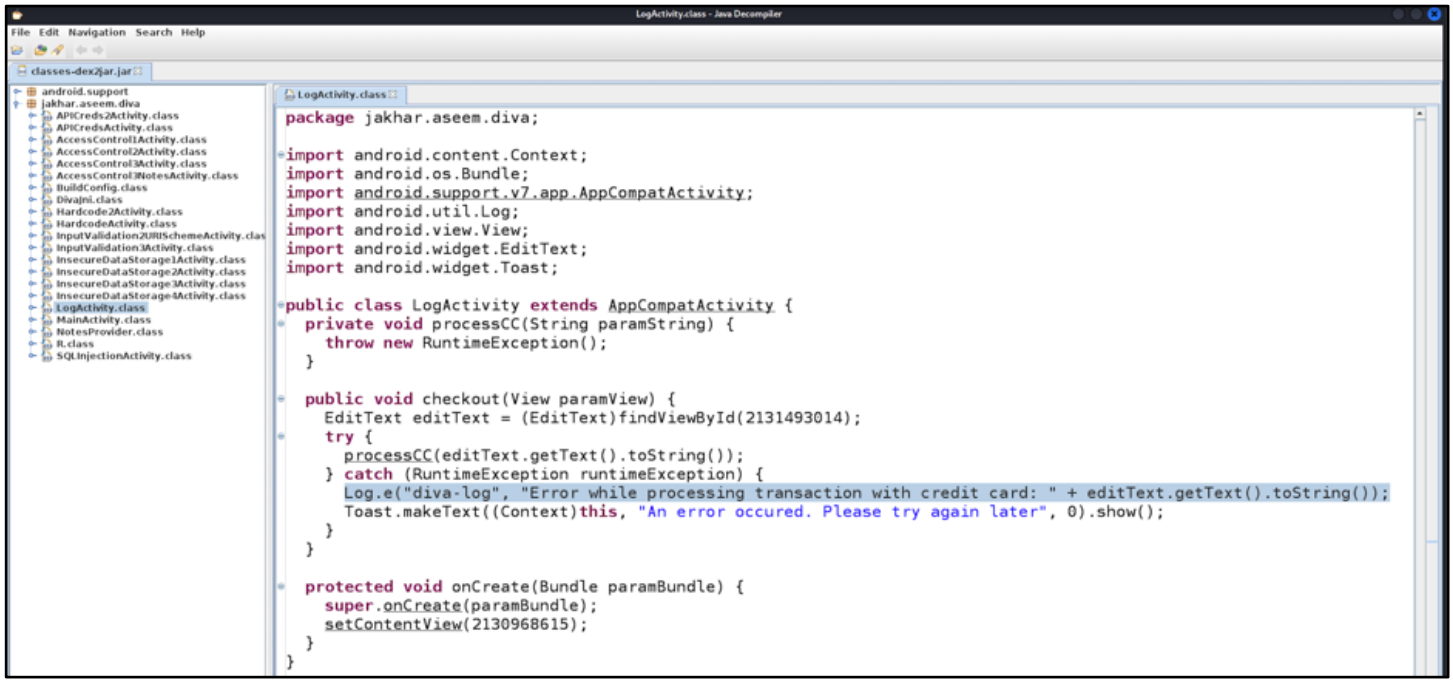
- Now we pass the `classes.dex` file to `d2j-dex2jar` tool and it will convert the Android DEX (Dalvik Executable) files into Java bytecode in the form of JAR files (Java Archive files).

```
$ d2j-dex2jar classes.dex
$ ls
AndroidManifest.xml  classes.dex  classes-dex2jar.jar resources.arsc  res/ lib/ META-INF/
```

Decompile all the .class files inside `.jar` file using `jd-gui`:

- Now, we can use `jd-gui` (a Java byte code de-compiler) that will take the `.class` files from the `.jar` archive and decompiles them into Java source code. At the same time, it will open all the files in a read-only GUI environment as shown in the screenshot below:

```
$ jd-gui classes-dex2jar.jar
```



Decompilation using apktool

Check Installed Java versions:

Before installing apktool, remember a minimum of Java8 is required to run it. So do checkout the version of java on your Kali Linux machine:

- Check for installed JDKs:

```
┌──(kali㉿kali)-[~]
└─$ update-java-alternatives --list
java-1.11.0-openjdk-amd64       1111       /usr/lib/jvm/java-1.11.0-openjdk-amd64
java-1.17.0-openjdk-amd64       1711       /usr/lib/jvm/java-1.17.0-openjdk-amd64
java-1.21.0-openjdk-amd64       2111       /usr/lib/jvm/java-1.21.0-openjdk-amd64
```

- Switch to appropriate JDK: Run the following command for java as well as for javac

```
┌──(kali㉿kali)-[~]
└─$ sudo update-alternatives --config java
There are 3 choices for the alternative java (providing /usr/bin/java).

  Selection    Path                                            Priority   Status
------------------------------------------------------------
  0            /usr/lib/jvm/java-21-openjdk-amd64/bin/java   2111      auto mode
  1            /usr/lib/jvm/java-11-openjdk-amd64/bin/java   1111      manual mode
  2            /usr/lib/jvm/java-17-openjdk-amd64/bin/java   1711      manual mode
* 3            /usr/lib/jvm/java-21-openjdk-amd64/bin/java   2111      manual mode

Press <enter> to keep the current choice[*], or type selection number: 3
```

- Verify and check the current version of java and javac

```
┌──(kali㉿kali)-[~]
└─$ java --version
openjdk 21.0.7-ea 2025-04-15
OpenJDK Runtime Environment (build 21.0.7-ea+7-Debian-1)
OpenJDK 64-Bit Server VM (build 21.0.7-ea+7-Debian-1, mixed mode, sharing)
```

Note: If JAVA_HOME variable is set then it takes priority on above settings.

Install apktool:

Try installing apktool using apt install, if it does not work somehow, then visit https://apktool.org/docs/install and follow the instructions. For Linux the instructions are:

1. Download the Linux wrapper script. (Right click, Save Link As `apktool`)
2. Download the latest version of Apktool.
3. Rename the downloaded jar to `apktool.jar`.
4. Move both `apktool.jar` and `apktool` to `/usr/local/bin`. (root needed)
5. Make sure both files are executable. (`chmod +x`)
6. Try running `apktool` via CLI.

Decompile diva-beta.apk

- Now let us decompile diva-beta.apk using apktool. You can create decompiled code inside a directory as the name of the apk file in the pwd. If you want to create a directory with a different name and at specified location, use the -o option. The following command will create a directory named diva in pwd and place all the app data inside it. Make time to view and analyze the contents of this directory:

```
$ apktool d diva-beta.apk -o diva-apktool
$ ls diva-apktool
AndroidManifest.xml   apktool.yml   smali/  lib/ res/  original/
$ tree lib
$ tree original
$ less smali/jakhar/aseem/diva/MainActivity.smali
$ subl smali/jakhar/aseem/diva/*.smali
```

```
└─$ tree lib
lib
├── arm64-v8a
│   └── libdivajni.so
├── armeabi
│   └── libdivajni.so
├── armeabi-v7a
│   └── libdivajni.so
├── mips
│   └── libdivajni.so
├── mips64
│   └── libdivajni.so
├── x86
│   └── libdivajni.so
└── x86_64
    └── libdivajni.so

8 directories, 7 files
```

```
└─$ tree original
original
├── AndroidManifest.xml
└── META-INF
    ├── CERT.RSA
    ├── CERT.SF
    └── MANIFEST.MF

2 directories, 4 files
```

Decompilation using `jadx`:

- The `jadx` is an open-source tool for decompiling Android APK files into Java source code. It converts the app's DEX (Dalvik Executable) files into readable Java code, providing an easier way to analyze, debug, or modify Android applications. Unlike `apktool` that generate smali code, JADX produces a more human-readable Java code, which is often useful for reverse engineering, security analysis, or learning about app internals.
- Download **`jadx-1.5.1.zip`** from https://github.com/skylot/jadx/releases/tag/v1.5.1
- Now unzip the file, and add the path of its `bin/` subdirectory inside your PATH environment variable, so that you can access the `jadx` command without giving the full path. By default, without any option, it will create a directory with the same name as that of the `apk` file containing the decompiled code. You can use the `-d` option and specify location and name of a directory where to place the decompiled code:

  `$ jadx diva-beta.apk -d diva-jadx`
- Make time to view and analyze the contents of this directory:

```
$ ls diva-jadx
resources/     sources/
$ ls resources/
AndroidManifest.xml    classes.dex    lib/      res/    META-INF/
$ ls sources/
android/ jakhar/
$ subl diva-jadx/sources/jakhar/aseem/diva/*.java
```

Comparison between `apktool` and `jadx`

Both `apktool` and `jadx` are a must tools you should know being reverse engineer. The `apktool` is ideal if you want to recompile APKs after making certain changes to the code like removing checks, modifying permissions, changing resources, injecting malware etc.). On the contrary `jadx` is focused on decompiling DEX bytecode into Java source code. It's great for understanding the app's logic but does not support recompiling APKs. The following table gives a detailed comparison between the two tools for your reference:

Functionality	apktool	jadx
Main Purpose	Reverse engineering APK resources and smali code. Smali is a human readable version of Dalvik Bytecode	Decompiling DEX bytecode to Java source code
Recompiling APK	It allows recompilation/rebuilding of the APK after modifying resources or smali code	No, it only decompiles DEX to Java code, and cannot rebuild APK
Handling Resources	Extracts and analyzes resources, including images, XML files, layout, and manifest	Limited. Displays some resources, but focuses on Java code
View Manifest File	It fully decompiles the AndroidManifest.xml	It partially decompiles
Obfuscation Resistance	Can still read smali code even when obfuscated	Obfuscated Java code is hard to understand due to name mangling

Example: Reverse Engineer, Modify, Build and Test an APK

Step 1: Decompile an Android app using `apktool`:

Decompile our friendly `login-app.apk` using `apktool`, and then view/analyze the decompiled contents

```
$ apktool d login-app.apk -o login-app
$ ls login-app
AndroidManifest.xml   apktool.yml   res/  original/ META-INF/
smali/  smali_classes2/   smali_classes3/
```

Step 2: Modify the Decompiled Source Code (APK Modding):

Let us search and change the login credentials. But for this first we have to search where the credentials are hardcoded inside the app, normally locating the "`const-string`" string. Let us recursively search for this string in all the files inside pwd:

```
$ grep -r "const-string" .
```

Once you find the files where the credentials are hardcoded, just open the file in `vim` and make changes to it say from `arif:kakamanna` to `arif1:kakamanna1`:

```
$ vim smali_classes3/rbee/examples/loginapp/MainActivity\$1.smali
const-string v1, "arif"
const-string v2, "kakamanna"
```

Step 3: Repackage / Rebuild the Source Files using `apktool`:

Let us now rebuild the modified source of the app using `apktool's -b` option followed by the directory name where the original `apk` has been decompiled. The `-o` option specifies the new name of the new `apk` file. Remember, while giving the following command, your `pwd` should not be somewhere inside the package decompiled directory, otherwise you may encounter directory path errors.

```
$ apktool b -f login-app/ -o login-app-new.apk
```

Step 4: Generate your APK Signing Key using `keytool`:

App signing is the process that ensures the security, integrity and authenticity of an Android app. Let us generate our own key using the **`keytool`**, which is a command-line utility provided by Java to create keystores for signing APKs. The following command will generate a file (`my-apk-key.keystore`) in the `pwd`, where the private key and certificate will be saved. The `-validity` option specifies the validity period of the key pair, in days. It will prompt you for password and some other required info, while generating the key. Once created, you can view the key using the second command.

```
$ keytool -genkey -v -keystore my-apk-key.keystore -alias mykey -keyalg RSA -keysize
2048 -validity 10000
$ keytool -list -v -keystore my-apk-key.keystore
```

Step 5: Re-sign the APK using `apksigner`:

Now we have the key, we can use the `apksigner/jarsigner` tool to self-sign the manipulated `apk`.

```
$ sudo apt install apksigner
$ which apksigner
$ apksigner --version
$ apksigner sign --ks my-apk-key.keystore --ks-key-alias mykey login-app-new.apk
```

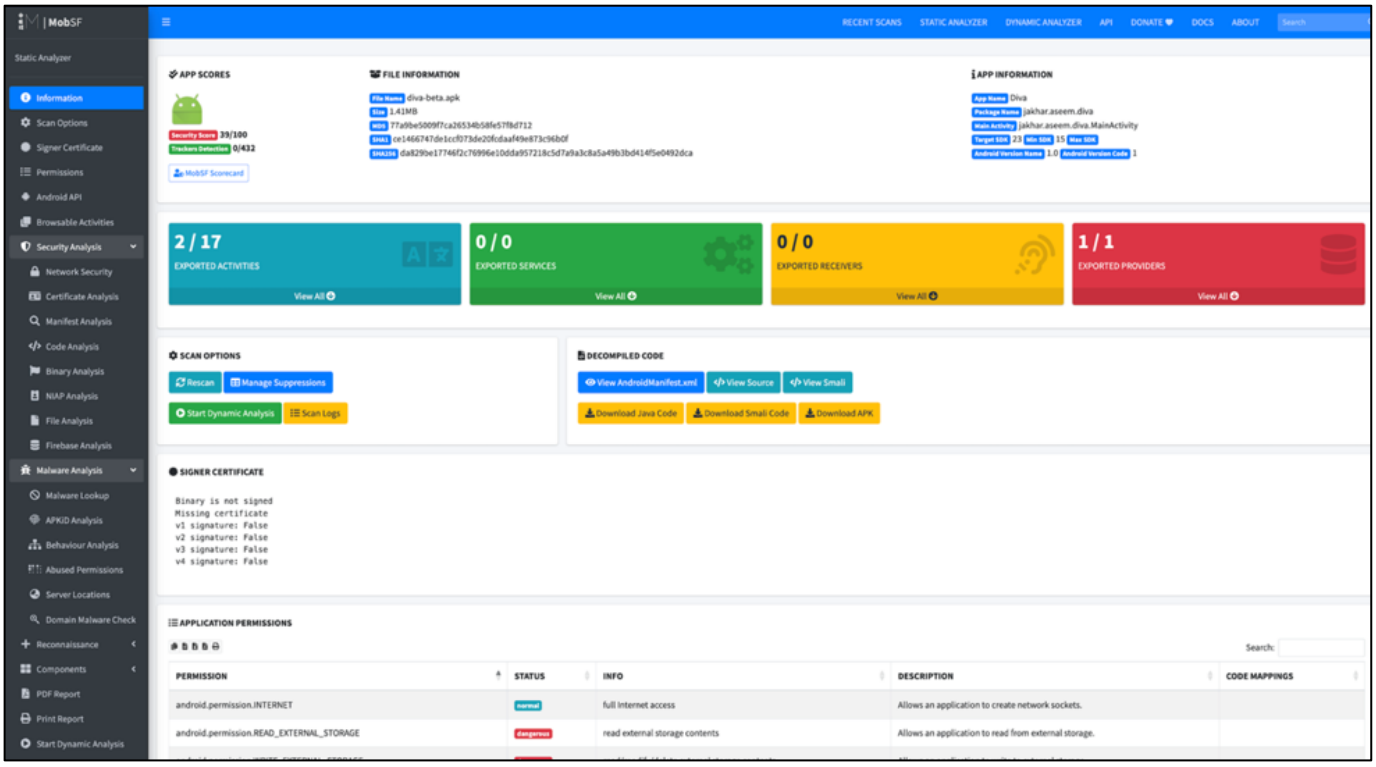
Once we will run the above command, it will prompt for the passphrase for the keystore, that we have entered while generating the keystore file in previous step.

Step 6: Install/Run/Test the new `apk` on Android device:

```
$ adb install login-app-new.apk
```

Static Analysis of APK Files using MobSF

- Mobile Security Framework (**MobSF**) is a popular open-source security research platform for static analysis of Android `.apk`, iOS `.ipa` and Windows `.appx` files, with limited dynamic analysis support as well.
- **Using Online MobSF:** The easiest way is to use the online version available at https://mobsf.live.
- **Installation on Kali Linux:**
```
$ git clone https://github.com/MobSF/Mobile-Security-Framework-MobSF.git
$ cd Mobile-Security-Framework-MobSF
$ sudo ./setup.sh
$ sudo ./run.sh 127.0.0.1:8000
```
- **Installation using Docker:**
```
$ docker pull opensecurity/mobile-security-framework-mobsf:latest
$ docker run -it --rm -p 8000:8000 opensecurity/mobile-security-framework-mobsf:latest
```
- Fire-up your Firefox inside Kali and type http://127.0.0.1:8000, to access MobSF web interface (Default Credentials: `mobsf:mobsf`).
- Let us perform the static analysis of `diva-beta.apk` by uploading the file using the web UI. MobSF decompiles the application and scan through its code, permissions, manifest files, and resources to look for coding mistakes, security flaws, privacy issues, misconfigurations, and malicious behaviors (all without executing the app). After that it provides a detailed report on security vulnerabilities, malware indicators, and best practice violations. Here is a screenshot of static analysis report of `diva-beta.apk` file.



Key Headings in MobSF's Report

- At the very top you can view the MobSF Score-card, File and App Information.
- Then it displays the **exported** activities, services, broadcast receivers, and content providers that may lead to security risks like component hijacking.
- You can view the **Decompiled code** including the `AndroidManifest.xml`, and the Java/smali source code.
- The **Signer Certificate information** tells us about vulnerabilities (like janus) that have to do with forging signatures.
- Check out if the app is over-privileged and asks for permissions that probably it shouldn't have asked for. (READ_SMS, WRITE_EXTERNAL_STORAGE)
- Next, we have the **Android API** along with the path of the files, and by clicking them you can view the code as well.

- **Browsable Activities and Network Security** are blank for this specific `apk`, as it is a standalone app and do not interact with the Internet
- **Certificate analysis** shows that the app is signed with debug certificate and is signed with v1 signature scheme, making it vulnerable to Janus vulnerability.
- Next will be the **Manifest analysis** section, that specifies 2 high and 4 warning issues. For example, Debugging is enabled on the app, which makes it easier for reverse engineers to hook a debugger to it, allowing dumping a stack trace and accessing debugging helper classes.
- Then is the **Code Analysis** section that tells us issues like logging sensitive information into log files and so on.
- After this we have the **Shared Library Binary Analysis**, that tells you about different shared object files and whether different binary protection mechanisms are in place or not (NX, PIE, stack canary, RELRO, FORTIFY, stripping of symbols etc.)
- The **Firebase Database analysis** section checks if the app connects to a Firebase database and whether it's properly secured or not. It looks for misconfigured Firebase URLs that might allow unauthorized read/write access, leading to data leakage or database tampering.
- The **Malware lookup** actually scans app components (files, strings, permissions) and matches them against malware signatures using public or internal databases (like `VirusTotal` if configured). It helps to identify, if the APK shows signs of known malware, trojans, spyware, or suspicious activities.
- **APKiD** analysis detects *packers* (tools that compress & encrypt the code and include a small loader that decrypts it at runtime), *obfuscators* (tools that scramble the names of classes methods and variables, so it's hard to understand when decompiled), and *protectors* (tools that prevent debugging, tempering and running the app on emulators or rooted devices) used inside the APK. Combined together, this makes it very hard for hackers or analysts to reverse-engineer. Common tools used for these tasks are `R8`, `DexGuard`.
- For the **Behaviour analysis** section, MobSF simulates and inspects the app's behaviour without running it by analyzing code patterns and permissions. It flags risky behaviours like SMS sending, background recording, contact theft, file system access, and system command execution.
- **Abused permission**s section highlights dangerous Android permissions the app requests, especially ones that are often misused by malware (`READ_SMS`, `WRITE_EXTERNAL_STORAGE`, `RECORD_AUDIO` etc)
- **Server locations** section maps the IP addresses and domains contacted by the app to their geographic locations that helps you see if the app is connecting to risky regions (e.g., unknown servers in high-risk countries).
- **Domain Malware Check:** MobSF checks all app-related domains against public malware blacklists. Flags if the app connects to a malicious or compromised server. (malicious-update-center.com, hacker-c2-server.ru)
- The URLs section extracts all URLs hardcoded inside the app (from code and resources), that helps spot C2 servers, API endpoints, or suspicious network calls.
- **Trackers:** MobSF detects third-party tracking SDKs or libraries (like Google Analytics, Facebook Graph API, Firebase Analytics) that helps identify user tracking and privacy violations.
- **Strings** section contains all human-readable strings from the app (code, smali, resources) to find API keys, URLs, IPs, commands, and hidden messages.

Conceptual Understanding of Android Booting Process

When an Android device is first powered on, there is a sequence of steps that are executed, helping the device to load necessary firmware, OS, application data, and so on into memory. Android boot process consists of multiple stages, starting from powering on the device to loading the home screen.

1. **The Boot ROM:** The Boot ROM is a tiny piece of read-only memory hardwired into Android's processor chip. It contains the code that runs when you press the power button, which turns on the device hardware (CPU, minimal RAM, storage controllers). It searches for the bootloader from device's storage and hands over control to the bootloader.

2. **The Bootloader:** The Bootloader is a piece of program that is executed before the OS starts to function having two parts: Initial Boot Loader (IPL) and Second Program Loader (SPL). The IPL detect and set up the external RAM, copies the SPL into it and transfers the control of execution to SPL. The SPL will look for Android kernel in the boot media and loads it in external RAM. If Secure Boot is enabled, it verifies the integrity of the boot image, and then hands over control to the Kernel.

3. **The Linux Kernel:** The Linux Kernel is the heart of Android OS, and is responsible for process management, memory management, and enforcing security on the device. Once the Linux kernel is loaded into the external RAM, it mounts the root file system (`rootfs`) and provides access to system and user data. The Kernel will look for the `init` program in the `rootfs` and launch it as a first user-space process.

4. **The `init` Process:** The `init` process will parse the `init.rc` script and launch the system service processes as mentioned in this file. At this stage you will see the Android logo on the device screen.

5. **Zygote and Dalvik:** The `Zygote` is the first Android process spawned by `init`, which initializes the Dalvik virtual machine and tries to create multiple instances, one for each Android process.

6. **The System Server:** Finally, the `system_server` process starts, which is responsible for running and managing all core Android services some of which are:
    1. Activity Manager (Manages app lifecycles, i.e., launch, pause, kill)
    2. Package Manager (Manages installed apps and permissions)
    3. Window Manager (Manages UI rendering and window interactions)
    4. Input Manager (Manages touchscreen, keyboard, and gesture inputs)
    5. Telephony Service (Manages calls, SIM and network connectivity)
    6. Battery Service, Sensor Service, Power Manager, etc.

7. **Home Screen is Displayed:** Once system services are running, the Launcher app (Home Screen app) is started by the Activity Manager. The user interface (UI) is drawn and displayed. The device is now ready for user interaction.

Hands-On Understanding of Android Booting Process

With the `adb` running between your Kali Linux and virtual Android device run the following commands to practically understand how booting process work:

- Let us first connect our Kali machine with Android device:
  `$ adb start-server`
  `$ adb connect 192.168.43.101:5555`
  `$ adb devices`
  ```
  List of devices attached
  192.168.43.101:5555    device
  ```

- Run the following command, which will reboot the device and perform the above discussed steps:
  `$ adb reboot bootloader`

- Once the bootloader has initialized the hardware, it loads the Android kernel. To monitor only the kernel logs that were generated during bootup use `-b` option with `logcat` command. The following output contains some of the important actions that I have captured from the actual output (which was quite long):
  `$adb logcat -b kernel`

  ```
  03-29 08:22:56.999     0     0 I         : Initializing cgroup subsys cpuset
  03-29 08:22:56.999     0     0 I         : Linux version 4.4.157-genymotion-ga887da7 (gcc version 4.9.3…
  03-29 08:22:56.999     0     0 I e820    : BIOS-provided physical RAM map:
  03-29 08:22:56.999     0     0 I         : NX (Execute Disable) protection: active
  03-29 08:22:56.999     0     0 I RAMDISK : [mem 0x37e30000-0x37f0ffff]
  03-29 08:22:56.999     0     0 I         : Kernel command line: BOOT_IMAGE=/kernel init=/init console=tty0
  androidboot.hardware=vbox86 androidboot.console=tty0 enforcing=0 selinux=1 androidboot.selinux=permissive
  quiet mac80211_hwsim.channels=2 buildvariant=userdebug
  03-29 08:22:56.999     0     0 I Memory  : 8159216K/8388152K available (8724K kernel code, 1258K rwdata, 3880K
  rodata, 1324K init, 876K bss, 228936K reserved, 0K cma-reserved)
  03-29 08:22:57.002     0     0 I SELinux : Initializing.
  03-29 08:22:57.002     0     0 I AppArmor: AppArmor disabled by boot time parameter
  03-29 08:22:57.415     0     0 I         : Trying to unpack rootfs image as initramfs...
  03-29 08:22:57.911     0     0 I init    : init first stage started!
  03-29 08:22:58.169     0     0 I init    : Added '/init.zygote32.rc' to import list
  03-29 08:22:58.170     0     0 I init    : Parsing file /init.environ.rc...
  03-29 08:22:58.723     0     0 I init    : starting service 'init-first-stage'...
  03-29 08:23:09.934     0     0 I init    : Received message 'start' for 'idmap2d' from system_server
  03-29 08:34:15.055     0     0 D logd    : logdr: UID=0 GID=0 PID=4895 b tail=0 logMask=80 pid=0 ...
  ```

- The `dmesg` command in Linux (and Android) displays the kernel ring buffer messages, which include diagnostic messages generated by the kernel during boot and runtime. It primarily provides information about the hardware initialization, driver loading, system errors, and other low-level kernel activities.
  `$ adb shell dmesg`

- Once the OS is up and running, you can view the related processes. The given output contains just some important related processes (VSZ is the virtual memory size in KB, RSS is the resident set size in KB, WCHAN is the waiting channel specifying the system call or resource for which the process is waiting, and S specifies the state sleeping, running, zombie):
  `$ adb shell ps`

USER	PID	PPID	VSZ	RSS	WCHAN	S	NAME
root	1	0	29388	8416	ep_poll	S	init
root	426	1	1191364	142388	poll_schedule	S	zygote
bluetooth	431	1	17468	5436	binder_thread_read	S	android.hardware.bluetooth@1.0-service
system	443	1	12296	4444	binder_thread_read	S	android.hardware.power@1.0-service
system	703	426	1590272	242744	ep_poll	S	system_server
bluetooth	851	426	1025436	105180	ep_poll	S	android.hardware.power@1.0-service
radio	1009	426	1036432	145272	ep_poll	S	com.android.phone

- View the contents of the `init.rc`, which is executed by the `init` process to initialize the system by setting system properties, mounting file systems, and starting essential services.
  `$ adb shell ls -l / | grep init`
  ```
  lrwxr-x---  1 root  shell   16      DTG     init -> /system/bin/init
  -rwxr-x---  1 root  shell   33623   DTG     init.rc
  -rwxr-x---  1 root  shell   563     DTG     init.zygote32.rc
  ```

  `$ adb shell cat /init.rc`

Rooting an Android Device

- Rooting is the process of gaining superuser access or administrator privileges on an Android device. By rooting the device, the user can gain full control over the operating system, allowing him/her to perform tasks like:
    - Modify system files
    - Change the kernel
    - Install custom ROMs or kernels
    - Remove bloatware (unwanted pre-installed apps)
- Rooting a modern Android device requires bypassing security features like *secure boot* (check Bootloader firmware), *verified boot* (check Kernel), *SELinux*, and *bootloader locking* (if you try to flash a custom ROM or Kernel it blocks the boot process). For example, if you attempt to root a modern Android device with Secure Boot enabled, you need to sign the customized kernel with the Original Equipment Manufacturer (OEM) cryptographic key, which if not known will give you an error: "boot image signature checks failed". The tools to root an Android device are **`Magisk`**, **`SuperSU`** and **`TWRP`** (Team Win Recovery Project).
- Risks and Consequences of rooting an Android device are:
    - Rooting typically voids the manufacturer's warranty.
    - Rooting makes the device more vulnerable to malware and malicious apps because it bypasses Android's security model and may disable security features like SELinux.
    - If not done correctly, rooting can brick (completely disable) the device, rendering it unusable.
    - Rooting might prevent users from receiving official Over-The-Air (OTA) updates from the device manufacturer, or the device might be "unrooted" when updating.
- After you have done, what you wanted to do with the rooted device, you can unroot it. Unrooting is the process of reversing or removing root access, and restores the device to the default settings. The changes you have made **may still be there**, depending on how the unrooting process is done.

Jailbreaking is the process of removing software restrictions imposed by Apple on iOS devices (iPhones, iPads, iPods). It allows users to install apps and tweaks from unofficial sources (such as Cydia), customize the interface, and bypass Apple's strict app installation policies, giving more freedom than what is allowed in the official Apple App Store.

Disclaimer

The series of handouts distributed with this course are only for educational purposes. Any actions and or activities related to the material contained within this handout is solely your responsibility. The misuse of the information in this handout can result in criminal charges brought against the persons in question. The authors will not be held responsible in the event any criminal charges be brought against any individuals misusing the information in this handout to break the law.