Android Application Framework: Beginner's Guide

December 2, 2020 By Raj Chandel

Android is a mobile operating system based on a modified version of the Linux kernel and other open-source software, designed primarily for touchscreen mobile devices such as smartphones and tablets. Android is developed by a consortium of developers known as the Open Handset Alliance and commercially sponsored by Google. In this article series we'll demonstrate various tools to conduct pentest on Android APK, manual analysis of the application as well as the major vulnerabilities; how to detect and fix them. But before beginning that, it is important to understand the architecture of an Android device, how is an application compiled and most importantly the composition of an Android application.

Table of Content

1.1 Brief history

Android Inc. was founded in Palo Alto, California, in October 2003 by Andy Rubin, Rich Miner, Nick Sears, and Chris White.

In July 2005, Google acquired Android Inc. for at least $50 million. Its key employees, including Rubin, Miner and White, joined Google as part of the acquisition. At Google, the team led by Rubin developed a mobile device platform powered by the Linux kernel. Google marketed the platform to handset makers and carriers on the promise of providing a flexible, upgradeable system.

The first commercially available smartphone running Android was the HTC Dream, also known as T-Mobile G1, announced on September 23, 2008.

1.2 Hardware

Android's main hardware platform for Android is ARM with x86 and x86-64 architectures also supported in the later releases. Since Android 5.0 Lollipop, 64-bit variants of all platforms are supported in addition to 32-bit variant.

1.3 Kernel

As of 2020, Android uses versions 4.4, 4.9 or 4.14 of Linux Kernel. Android Kernel is based on Linux Kernel's Long Term Support (LTS) branch.

1.4 File System

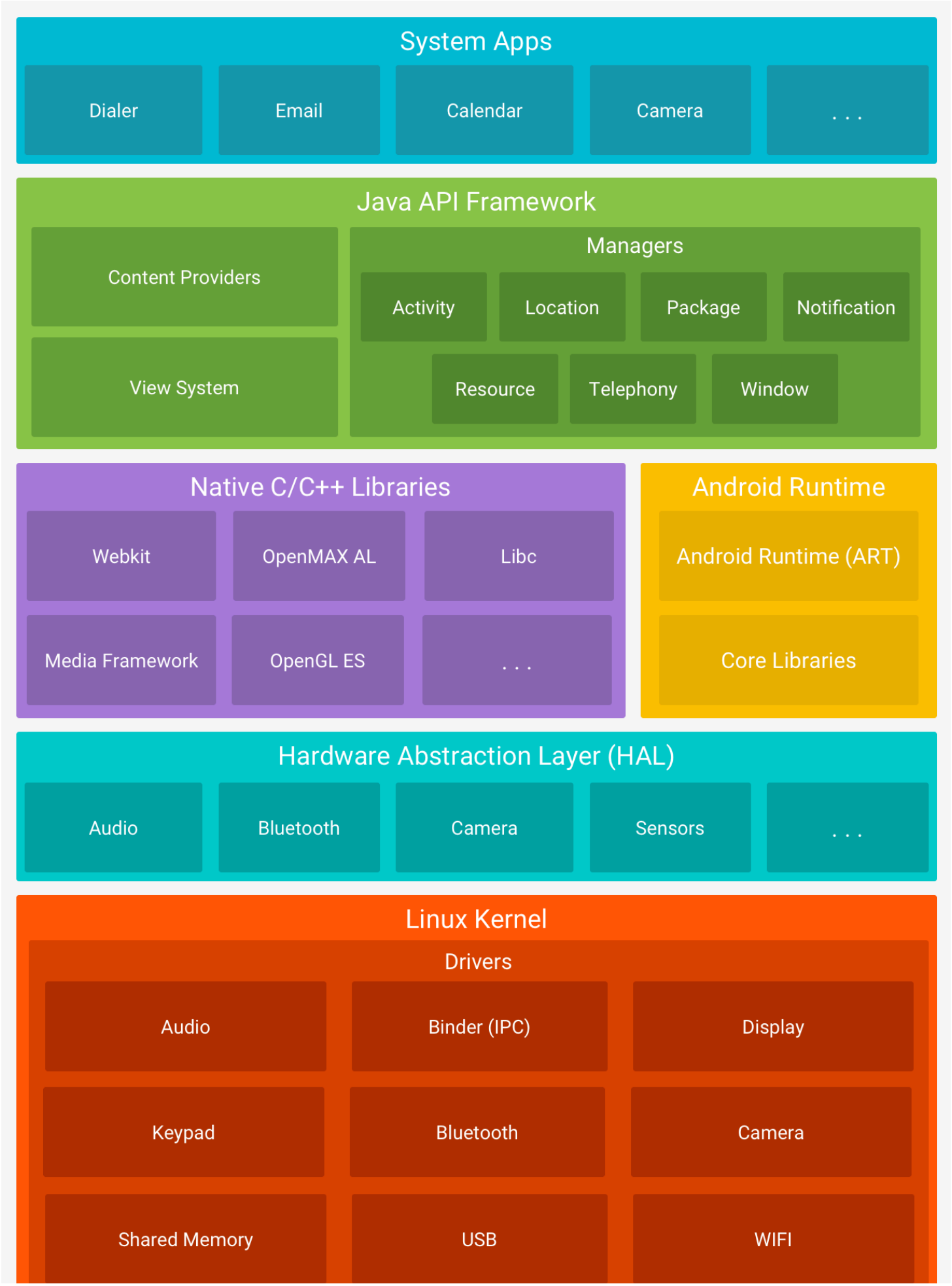
Ever since the release of Android, it has made use of the YAFFS2. After Android 2.3 it used EXT4 file system, although many OEMs have experimented with F2FS. The following directories are there by default in any installation of Android:

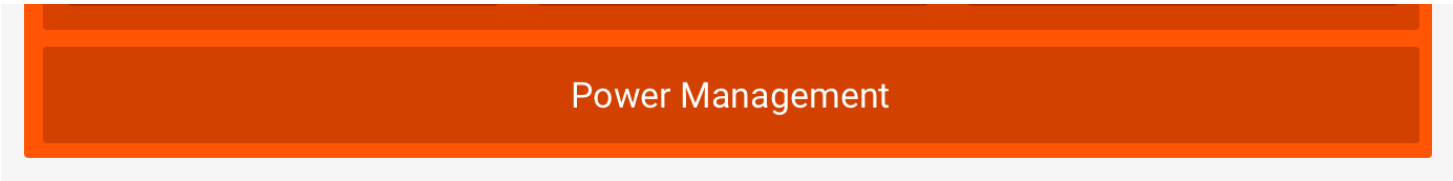
- Boot: The partition contains the kernel, ramdisk etc. which is required for the phone to boot when powered on.
- System: Contains OS files which include Android UI and pre-installed apps
- Recovery: Alternative option to booting into OS, allows recovery and backup of partitions.
- Data: Saves user data, apps data, messaging, music etc. Can be traversed through the file browser. This is wiped when the factory reset is pressed. Sub-folders are:
    - Android – Default for app cache and saved data.
    - Alarms – Custom audio files for alarms.
    - Cardboard – Contains data for VR files.
    - DCIM – Stores pictures and videos were taken by Camera app.
    - Downloads – Stores downloaded files from the internet.
    - Notifications – Custom tones for notifications of some apps.
    - Music, Movies – Default folders to store songs and videos from third-party apps.
    - Pictures – Default folder to store pictures taken by third-party apps.
    - Podcasts – Stores podcasts files when you use a podcast app.
    - Videos – Stores downloaded videos from third-party apps.

- Cache: Storage of frequently used data and app components.
- Misc: Contains other important system setting information. Like USB config, carrier ID.

2.

Android Architecture


System Apps
Dialer
Email
Calendar
Camera
. . .
Java API Framework
Content Providers
View System
Managers
Activity
Location
Package
Notification
Resource
Telephony
Window
Native C/C++ Libraries
Webkit
OpenMAX AL
Libc
Media Framework
OpenGL ES
. . .
Android Runtime
Android Runtime (ART)
Core Libraries
Hardware Abstraction Layer (HAL)
Audio
Bluetooth
Camera
Sensors
. . .
Linux Kernel
Drivers
Audio
Binder (IPC)
Display
Keypad
Bluetooth
Camera
Shared Memory
USB
WIFI



(Source: android.com)

2.1 Kernel

The kernel is the most important part of the Android OS that provides an interface for the user to communicate with the hardware. It contains the essential drivers that are used by programs to instruct a hardware component a perform a specific function. These drivers are audio, display, Bluetooth etc.

2.2 Hardware Abstraction Layer (HAL)

A **hardware abstraction layer** (**HAL**) is a logical division of code that serves as an abstraction layer between a computer's physical hardware and its software. It provides a device driver interface allowing a program to communicate with the hardware.

2.3 Libraries

Sitting on the top of the kernel, libraries provide developer support to develop applications, resource file and even manifest. There are some native libraries like SSL, SQLite, Libc etc that are required by native codes to effectively perform a task.

2.4 Android Runtime

Android Runtime (ART) is an application runtime environment used by the Android OS. ***A runtime environment is a state in which program can send instructions to the computer's processor and access the computer's RAM. Android apps programmed in (let's say) Java, will be first converted to byte code during compilation packaged as an APK and run-on runtime.***

Android uses a Virtual Machine to execute any application so as to isolate the execution of the program from OS and protecting malicious code from the affecting system.

Before Android 4.4, that runtime used to be DVM (Dalvik Virtual Machine) which has since been replaced by Android Runtime (ART).

DVM used JIT (Just in time) compilation which worked by taking application code, analyzing it and actively translating it during runtime. ART uses AOT (ahead of time) compilation that compiles *.dex files (Dalvik Bytecode) before they are even needed. Usually, it is done during installation and stored in phone storage.

To be noted: After Android N, JIT compilation was reintroduced along with AOT and an interpreter in ARt making it hybrid to tackle against problems like installation time and memory.

2.5 Application Framework

The entire feature-set of the Android OS is available to the developer through APIs written in the Java language. These APIs make the most important components that are needed to create Android apps. These are:

- **View System:** Used to build an app's UI, including lists, text boxes, buttons etc.
- **Resource Manager:** Provides access to non-code resources like layout files, graphics etc.
- **Notification manager:** Allows the app to display custom alerts in the status bar.
- **Activity Manager:** Manages the lifecycle of apps and provides a common navigation back stack.
- **Content Providers:** Enables apps to access data from other apps like WhatsApp access data from the Contacts app.

2.6 System Applications

The pre-installed set of core applications used for basic functions like SMS, calendars, internet browsing, contacts etc.

3.

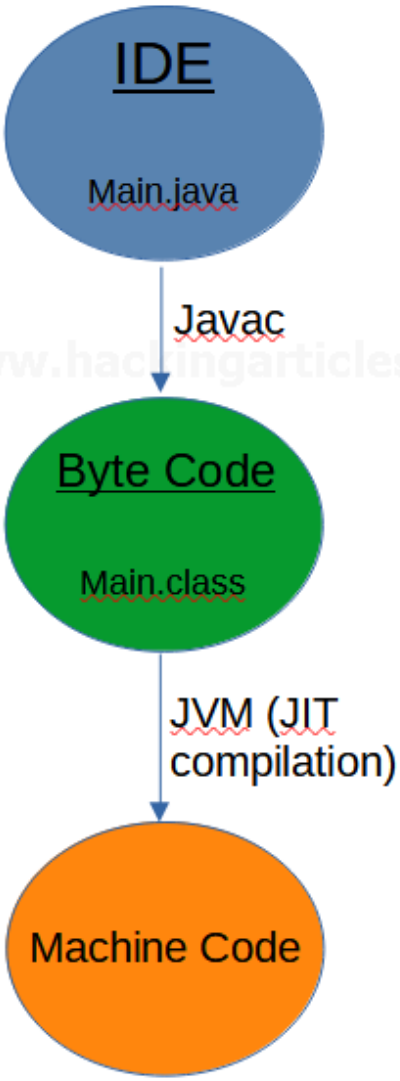
APK Compilation and De-compilation

3.1 Compilation

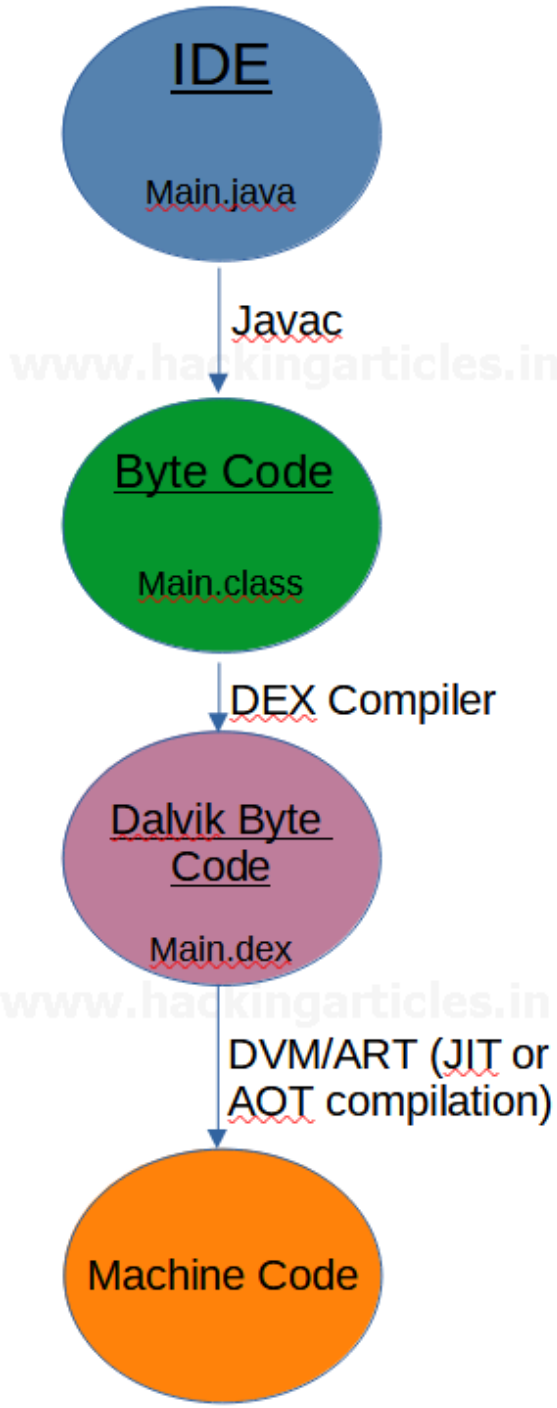
Android compilation process could be understood better when comparing to the java compilation process. To compile a java code the following steps are performed:

- A code written in main.java is compiled by javac (Java compiler).
- Javac compiles main.java into java byte code (main.class)
- A Java Virtual Machine (JVM) understands and converts Java byte code file (main.class) into machine code using JIT (Just-in-time) compiler.
- The machine code is then run by CPU.

It can be demonstrated in the following diagram:



Now, the main difference between Java and Android compilation process is that Android doesn't work with JVM because a JVM doesn't enable a variety of services with a limited processor speed and RAM. So, Android introduced a Dalvik Virtual Machine (DVM). However, in the latest Android releases ART is being used. The process is now as follows:



It is to be noted that even though Dalvik was replaced as the default runtime, Dalvik bytecode format (*.dex) is still in use.

3.2 Decompiling APK

While conducting reverse engineering, APK de-compilation and re-compilation is done. We'll look after proper recompilation and reverse engineering in further articles, let's have a look at de-compilation to understand how an APK is essentially structured when packed in the format.

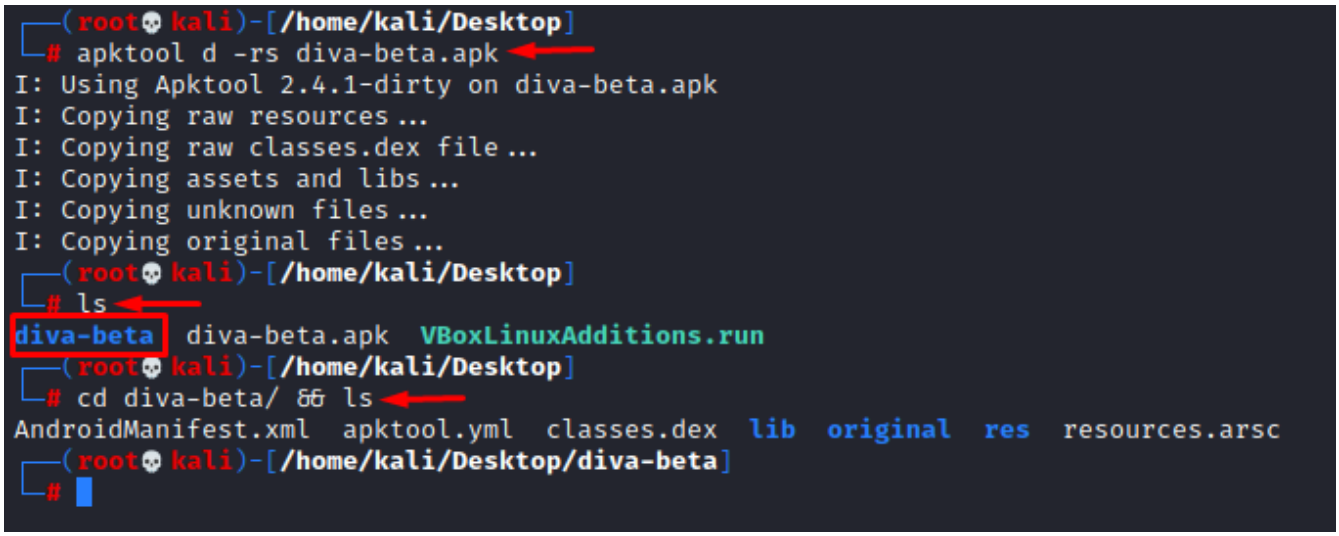
It is to be noted that an APK file is just a ZIP archive that contains XML files, dex code, resource files and other files.

So, to decompile it we can either:

Unzip <name>.apk

Or we can take aid of a great tool called apktool. So, we'll run the following command:

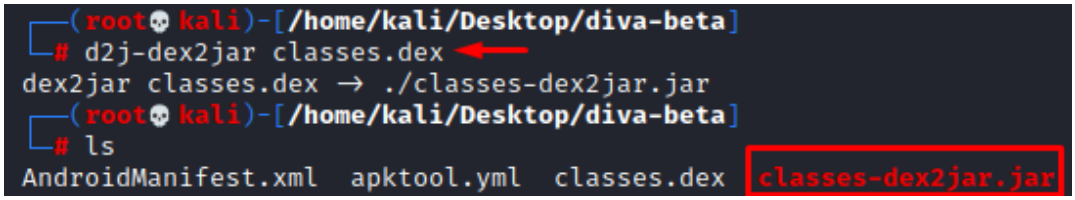
```
apktool -d -rs diva-beta.apk
ls
cd diva-beta/ && ls
```



Here, we see that a folder has been created that essentially packs Manifest file, yml file, resources and a file named classes.dex

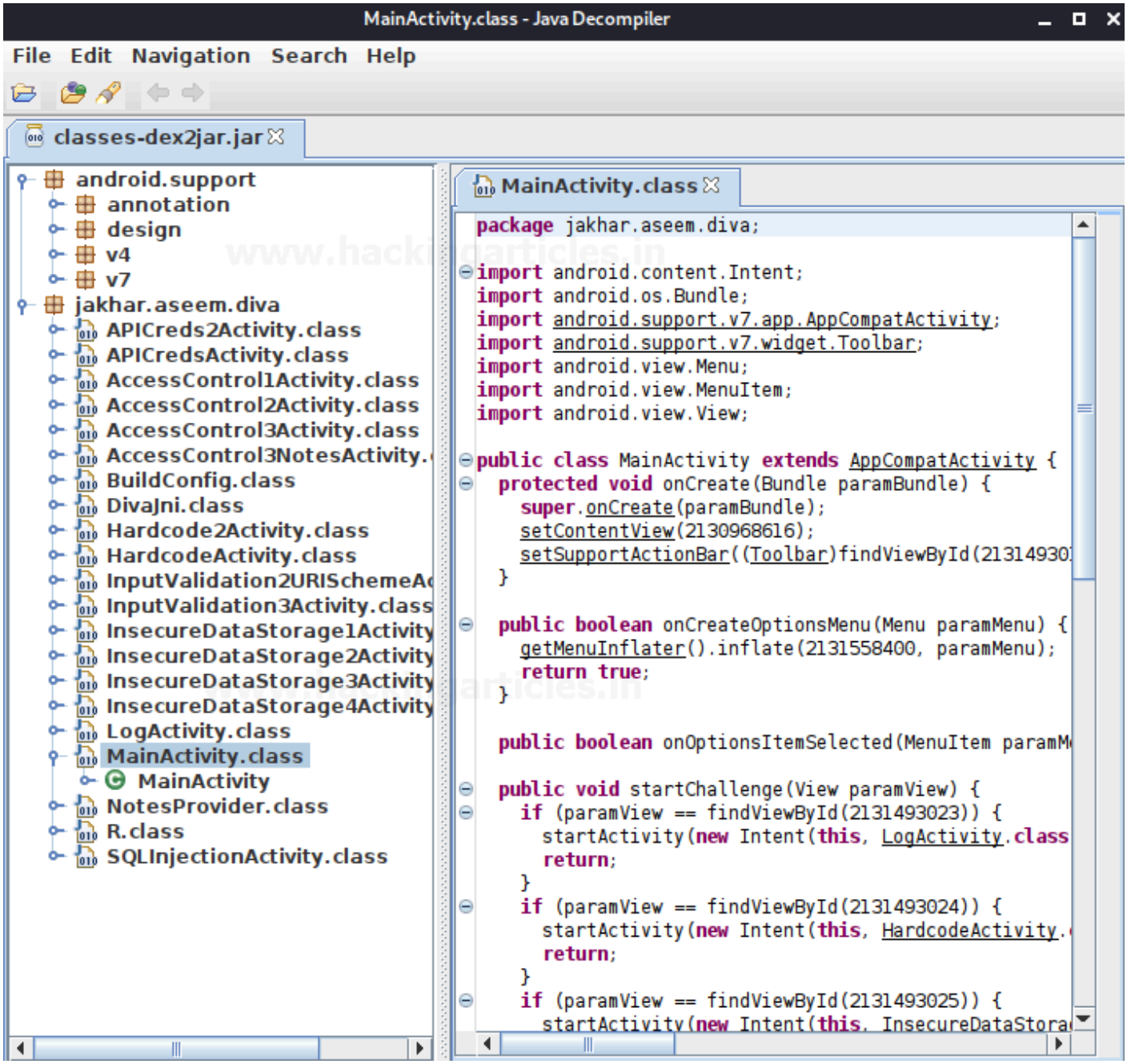
This dex file contains the dalvik byte code and we'll further disassemble this file into standard class files using the following command:

```
d2j-dex2jar classes.dex
```



This command further creates a .jar code structure. We will use a java decompiler to inspect the source code:

```
Jd-gui classes-dex2jar.jar
```

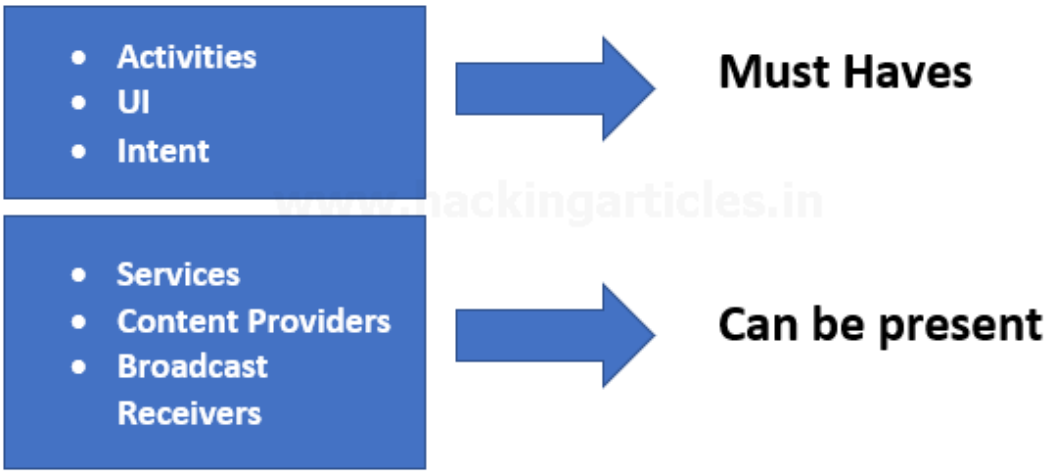


It is to be noted that the readability of this code entirely depends on whether or not an obfuscator (like ProGuard) was used while creating an APK file.

4. **Composition of an APK**

Any android application comprises of 6 essential things which are bound by the application manifest file that contains the description of how these interact with each other. Out of these, activities, UI and intents are a must

have in any app while services, content providers and broadcast receivers can be present depending on needs. The components are:



- **Activities**: Activity is a class in Android which, when implemented, represents a single screen with a user interface just like window or frame of Java. Every application has a ***MainActivity (java*** if code is in Java*)* which automatically runs by default for the first time when an application is run. All the actions performed in Activity pages are done using callbacks. There are 7 callbacks:

Sr.No	Callback & Description
1	**onCreate()** This is the first callback and called when the activity is first created.
2	**onStart()** This callback is called when the activity becomes visible to the user.
3	**onResume()** This is called when the user starts interacting with the application.
4	**onPause()** The paused activity does not receive user input and cannot execute any code and called when the current activity is being paused and the previous activity is being resumed.
5	**onStop()** This callback is called when the activity is no longer visible.
6	**onDestroy()** This callback is called before the activity is destroyed by the system.
7	**onRestart()** This callback is called when the activity restarts after stopping it.

- **Services**: A **service** is a component that runs in the background to perform long-running operations without needing to interact with the user and it works even if application is destroyed. A service can

essentially take two states:

1. **Started:** A service is **started** when an application component, such as an activity, starts it by calling *startService()*. Once started, a service can run in the background indefinitely, even if the component that started it is destroyed.
2. **Bound:** A service is **bound** when an application component binds to it by calling *bindService()*. A bound service offers a client-server interface that allows components to interact with the service, send requests, get results, and even do so across processes with interprocess communication (IPC).

The **Service** base class defines various callback methods and the most important are given below:

Sr. No.	Callback and Description
1	**onStartCommand():** The system calls this method when another component, such as an activity, requests that the service be started, by calling *startService()*.
2	**onBind():** The system calls this method when another component wants to bind with the service by calling *bindService()*
3	**onUnbind():** The system calls this method when all clients have disconnected from a particular interface published by the service.
4	**onRebind():** The system calls this method when new clients have connected to the service, after it had previously been notified that all had disconnected in its *onUnbind(Intent)*.
5	**onCreate():** The system calls this method when the service is first created using *onStartCommand()* or *onBind()*. This call is required to perform one-time set-up.
6	**onDestroy():** The system calls this method when the service is no longer used and is being destroyed.

- **Content Providers**: Content providers provide the content/data to Android applications from an Android system or other Android applications. To create a service, you create a Java class that extends the Service base class or one of its existing subclasses. Content providers provide the following four basic operations. These are also known as **CRUD operations**, where:

C – Create: It is used for the creation of data in content providers.
R – Read: It reads the data stored in the content provider.
U – Update: It lets the editing in existing data in content providers.
D – Delete: It deletes the existing data stored in its Storage.

To implement content providers, we'd use the following callbacks in Java:

Sr. No.	Callbacks and Description
1.	**onCreate() –** This method in Android initializes the provider as soon as the receiver is created.
2.	**query() –** It receives a request in the form of a query from the user and responds with a cursor in Android
3.	**insert() –** This method is used to insert the data into our content provider.
4.	**update() –** This method is used to update existing data in a row and return the updated row data.
5.	**delete() –** This method deletes existing data from the content provider.
6.	**getType() –** It returns the Multipurpose Internet Mail Extension type of data to the given Content URI.

- **Broadcast Receivers**: Broadcast Receivers simply respond to broadcast messages from other applications or from the system itself. These messages are sometime called events or intents. For example, applications can also initiate broadcasts to let other applications know that some data has been downloaded to the device and is available for them to use, so this is a broadcast receiver who will intercept this communication and will initiate appropriate action. Some of the broadcast receivers are:
    - **intent.action.REBOOT**
    - **intent.action.DATE_CHANGED**
    - **intent.action.BOOT_COMPLETED**

- **Intents**: An Android **Intent** is an abstract description of an operation to be performed. It can be used with **startActivity** to launch an Activity, **broadcastIntent** to send it to any interested BroadcastReceiver components, and **startService(Intent)** or **bindService(Intent, ServiceConnection, int)** to communicate with a background Service.
- **Views:** All the interaction of a user with the Android application is through the user interface(UI). **View** is the basic building block of UI (User Interface) in android. View refers to the android.view.View class, which is the **super class** for all the GUI components like TextView, ImageView, Button etc

5.

Conclusion

In this article, we saw basics of android and development necessary to know before beginning android pen-testing. In the next few articles in the series, we'll be learning complete Android Penetration Testing that will cover both aspects static and dynamic analysis, especially covering the Mobile OWASP Top 10 and completing challenges in DIVA application.

6. developer.android.com
7. wikipedia.com
8. tutorialpoint.com/android/index.htm