eLearnSecurity Mobile Application Penetration Testing (eMAPT) Notes ANDROID by Joas

Sumário

Warning

These are notes focused on the eMAPT test, I didn't put it on iOS, because the test only covers Android content, but I'll prepare something just for iOS. Hope it helps in some way and of course the annotations were collected from public sources and all credits are given by the respective owners.

Android Architecture

Android architecture contains different number of components to support any android device needs. Android software contains an open-source Linux Kernel having collection of number of C/C++ libraries which are exposed through an application framework services.

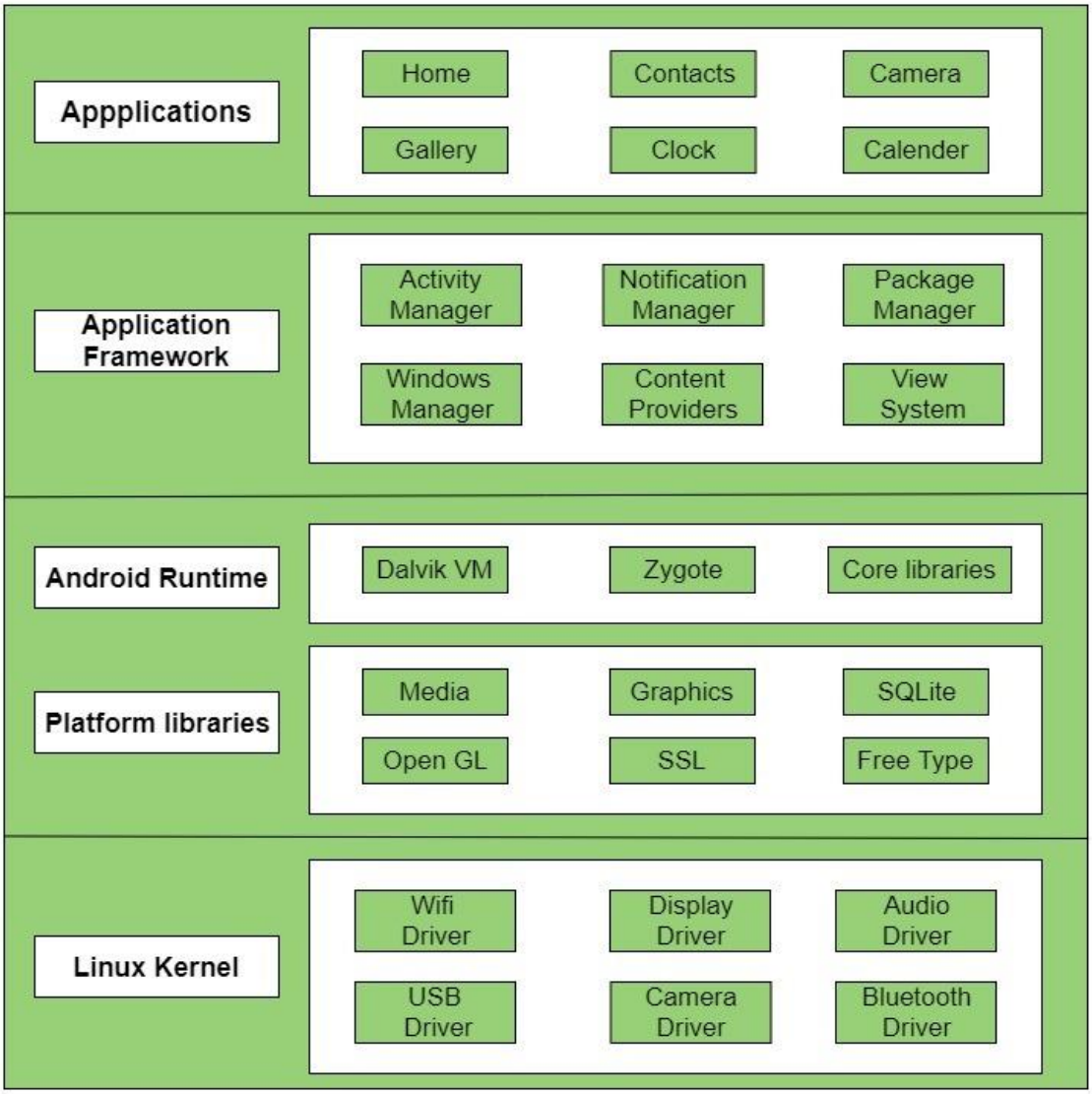
Among all the components Linux Kernel provides main functionality of operating system functions to smartphones and Dalvik Virtual Machine (DVM) provide platform for running an android application.

The main components of android architecture are following:-

- Applications
- Application Framework

- Android Runtime
- Platform Libraries
- Linux Kernel

Pictorial representation of android architecture with several main components and their sub components –



Applications –

Applications is the top layer of android architecture. The pre-installed applications like home, contacts, camera, gallery etc and third party applications downloaded from the play store like chat applications, games etc. will be installed on this layer only.
It runs within the Android run time with the help of the classes and services provided by the application framework.

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.

Application runtime –

Android Runtime environment is one of the most important part 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.

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 an 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 **FreeType** 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:

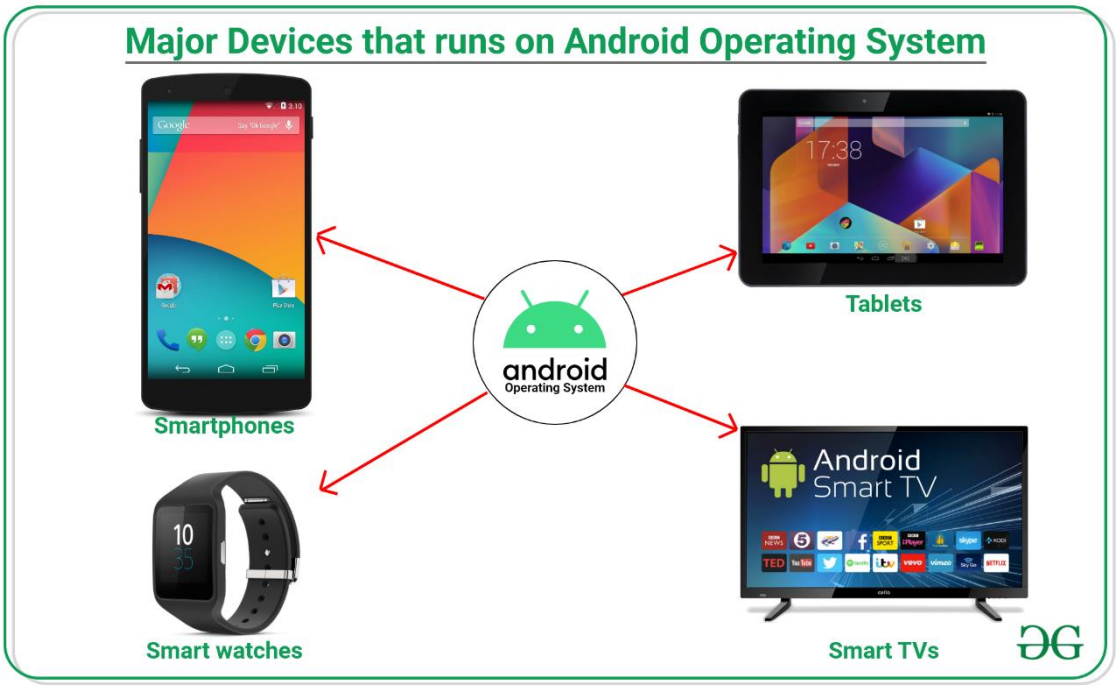
- **Security:** The Linux kernel handles the security between the application and the system.

- **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.

https://www.geeksforgeeks.org/android-architecture

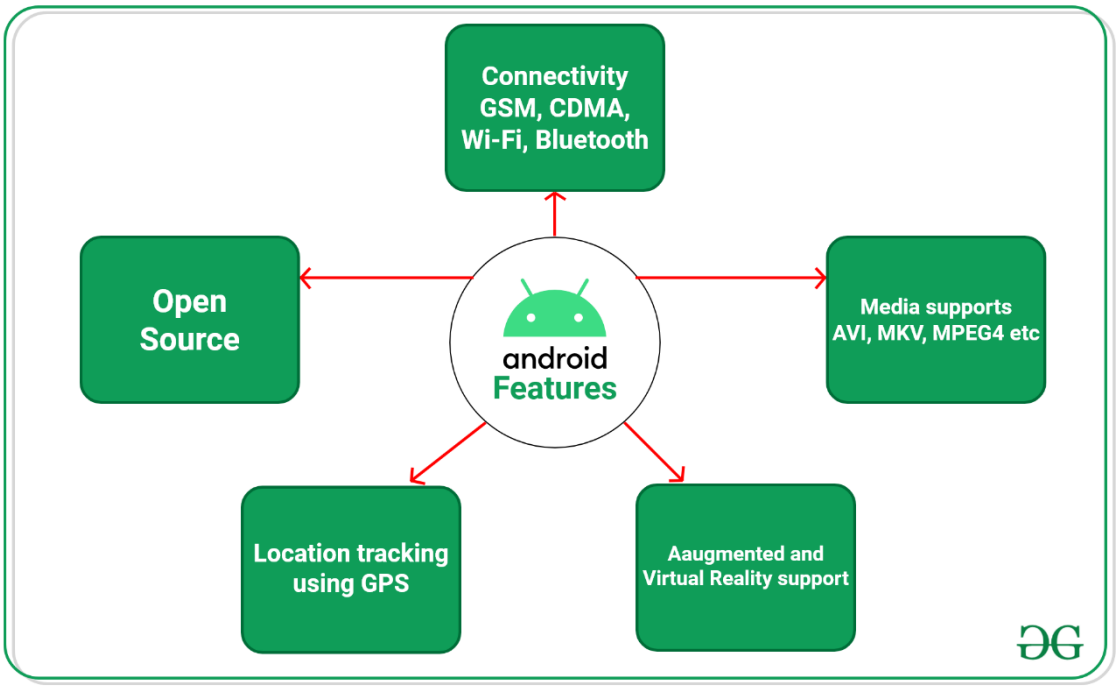
Introduction to Android Development

The **Android operating system** is the largest installed base among various mobile platforms across the globe. Hundreds of millions of mobile devices are powered by **Android** in more than 190 countries of the world. It conquered around **75%** of the global market share by the end of 2020, and this trend is growing bigger every other day. The company named **Open Handset Alliance** developed Android for the first time that is based on the modified version of the Linux kernel and other open-source software. **Google** sponsored the project at initial stages and in the year 2005, it acquired the whole company. In September 2008, the first Android-powered device launched in the market. Android dominates the mobile OS industry because of the long list of features it provides. It's user-friendly, has huge community support, provides a greater extent of customization, and a large number of companies build Android-compatible smartphones. As a result, the market observes a sharp increase in the demand for developing Android mobile applications, and with that companies need smart developers with the right skill set. At first, the purpose of Android was thought of as a mobile operating system. However, with the advancement of code libraries and its popularity among developers of the divergent domain, Android becomes an absolute set of software for all devices like tablets, wearables, set-top boxes, smart TVs, notebooks, etc.



Features of Android

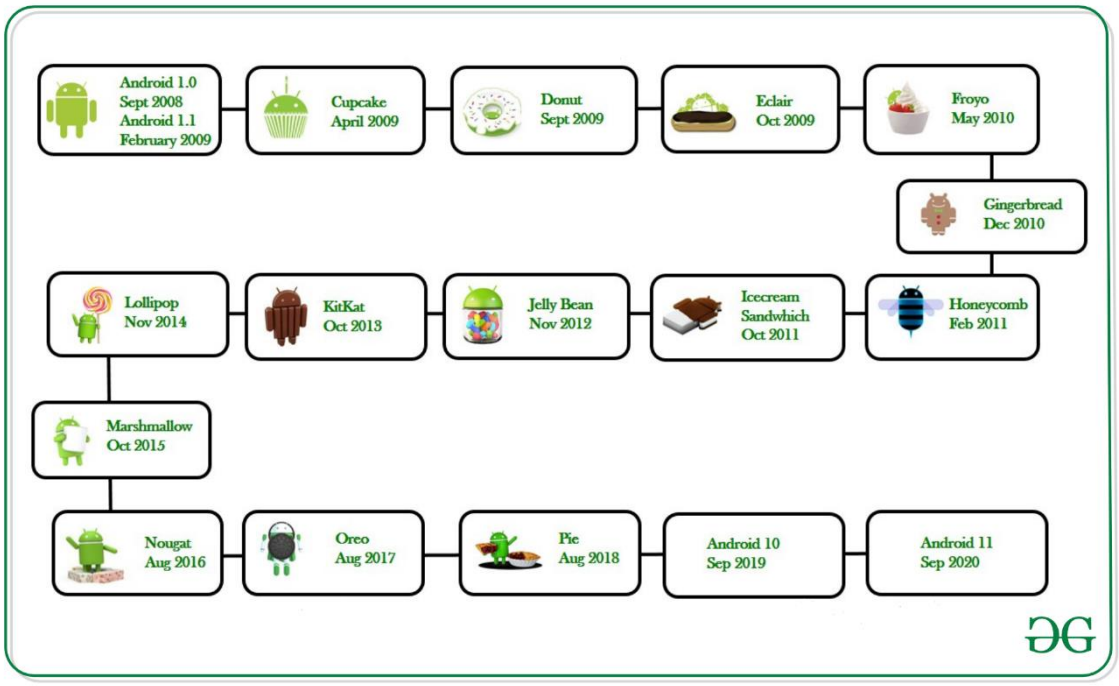
Android is a powerful open-source operating system that open-source provides immense features and some of these are listed below.



- Android Open Source Project so we can customize the OS based on our requirements.
- Android supports different types of connectivity for GSM, CDMA, Wi-Fi, Bluetooth, etc. for telephonic conversation or data transfer.
- Using wifi technology we can pair with other devices while playing games or using other applications.
- It contains multiple APIs to support location-tracking services such as GPS.
- We can manage all data storage related activities by using the file manager.
- It contains a wide range of media supports like AVI, MKV, FLV, MPEG4, etc. to play or record a variety of audio/video.
- It also supports different image formats like JPEG, PNG, GIF, BMP, MP3, etc.
- It supports multimedia hardware control to perform playback or recording using a camera and microphone.
- Android has an integrated open-source WebKit layout based web browser to support User Interface like HTML5, CSS3.
- Android supports multi-tasking means we can run multiple applications at a time and can switch in between them.
- It provides support for virtual reality or 2D/3D Graphics

Android Versions

Google launched the first version of the Android platform on Nov 5, 2007. Since then, Google released a lot of android versions such as Apple Pie, Banana Bread, Cupcake, Donut, Éclair, Froyo, Gingerbread, Jellybeans, Kitkat, Lollipop, marshmallow, Nougat, Oreo, etc. with extra functionalities and new features.



The following table shows the version details of android which is released by Google from 2007 to date.

Code Name	Version	API level	Release date
Apple Pie	Android 1.0	1	September 23, 2008
Banana Bread	Android 1.1	2	February 9, 2009
Cupcake	Android 1.5	3	April 30, 2009
Donut	Android 1.6	4	September 15, 2009
Eclair	Android 2.0 – 2.1	5-7	October 26, 2009
Froyo	Android 2.2 – 2.2.3	8	May 20, 2010

Code Name	Version	API level	Release date
Gingerbread	Android 2.3 – 2.3.4	9-10	December 6, 2010
Honeycomb	Android 3.0.x – 3.2.x	11 – 13	February 22, 2011
Ice Cream Sandwich	Android 4.0 – 4.0.4	14 – 15	October 18, 2011
Jelly Bean	Android 4.1 – 4.1.2	16 – 18	July 9, 2012
Kitkat	Android 4.4 – 4.4.4	19	July 9, 2012
Lollipop	Android 5.0 – 5.1	21 – 22	October 17, 2014
Marshmallow	Android 6.0 – 6.0.1	23	October 5, 2015
Nougat	Android 7.0 – 7.1	24 – 25	August 22, 2016
Oreo	Android 8.0	26	August 21, 2017
Pie	Android 9.0	27	August 6, 2018
Android Q	Android 10.0	29	September 3, 2019
Android 11	Android 11.0	30	September 8, 2020

Programming Languages used in Developing Android Applications

1. **Java**
2. **Kotlin**

Developing the Android Application using Kotlin is preferred by Google, as Kotlin is made an official language for Android Development, which is developed and maintained by JetBrains. Previously before the Java is considered the official language for Android Development. Kotlin is made official for Android Development in Google I/O 2017.

Advantages of Android Development

- The Android is an open-source Operating system and hence possesses a vast community for support.
- The design of the Android Application has guidelines from Google, which becomes easier for developers to produce more intuitive user applications.
- Fragmentation gives more power to Android Applications. This means the application can run two activities on a single screen.
- Releasing the Android application in the Google play store is easier when it is compared to other platforms.

Disadvantages of Android Development

- Fragmentation provides a very intuitive approach for user experience but it has some drawbacks, where the development team needs time to adjust with the various screen sizes of mobile smartphones that are now available in the market and invoke the particular features in the application.
- The Android devices might vary broadly. So the testing of the application becomes more difficult.
- As the development and testing consume more time, the cost of the application may increase, depending on the application's complexity and features.

Android UI Layouts

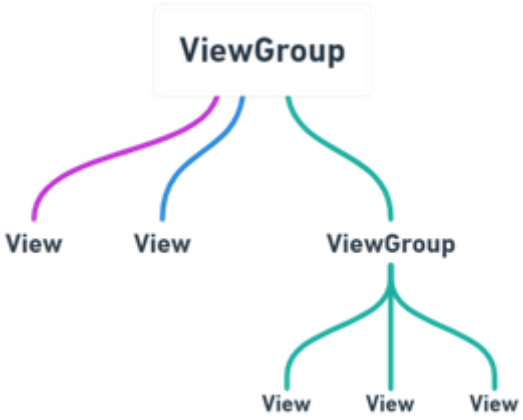
Android Layout is used to define the user interface that holds the UI controls or widgets that will appear on the screen of an android application or activity screen. Generally, every application is a combination of View and ViewGroup. As we know, an android application contains a large number of activities and we can say each activity is one page of the application. So, each activity contains multiple user interface components and those components are the instances of the View and ViewGroup. All the elements in a layout are built using a hierarchy of View and ViewGroup objects.

View

A View is defined as the user interface which is used to create interactive UI components such as TextView, ImageView, EditText, RadioButton, etc., and is responsible for event handling and drawing. They are Generally Called Widgets.

View

A ViewGroup act as a base class for layouts and layouts parameters that hold other Views or ViewGroups and to define the layout properties. They are Generally Called layouts.



ViewGroup

The Android framework will allow us to use UI elements or widgets in two ways:

- **Use UI elements in the XML file**
- **Create elements in the Kotlin file dynamically**

Types of Android Layout

- **Android Linear Layout: LinearLayout is a ViewGroup subclass, used to provide child View elements one by one either in a particular direction either horizontally or vertically based on the orientation property.**

- **Android Relative Layout: RelativeLayout is a ViewGroup subclass, used to specify the position of child View elements relative to each other like (A to the right of B) or relative to the parent (fix to the top of the parent).**
- **Android Constraint Layout: ConstraintLayout is a ViewGroup subclass, used to specify the position of layout constraints for every child View relative to other views present. A ConstraintLayout is similar to a RelativeLayout, but having more power.**
- **Android Frame Layout: FrameLayout is a ViewGroup subclass, used to specify the position of View elements it contains on the top of each other to display only a single View inside the FrameLayout.**
- **Android Table Layout: TableLayout is a ViewGroup subclass, used to display the child View elements in rows and columns.**
- **Android Web View: WebView is a browser that is used to display the web pages in our activity layout.**
- **Android ListView: ListView is a ViewGroup, used to display scrollable lists of items in a single column.**
- **Android Grid View: GridView is a ViewGroup that is used to display a scrollable list of items in a grid view of rows and columns.**

Use UI Elements in the XML file

Here, we can create a layout similar to web pages. The XML layout file contains at least one root element in which additional layout elements or widgets can be added to build a View hierarchy. Following is the example:

- **XML**

```
<?xml version="1.0" encoding="utf-8"?>
<LinearLayout
    xmlns:android="http:// schemas.android.com/apk/res/android"
    xmlns:tools="http:// schemas.android.com/tools"
    android:orientation="vertical"
    android:layout_width="match_parent"
    android:layout_height="match_parent"
    tools:context=".MainActivity">
```

```
    <!--EditText with id editText-->
    <EditText
        android:id="@+id/editText"
        android:layout_width="match_parent"
        android:layout_height="wrap_content"
        android:layout_margin="16dp"
        android:hint="Input"
        android:inputType="text"/>

    <!--Button with id showInput-->
    <Button
        android:id="@+id/showInput"
        android:layout_width="wrap_content"
        android:layout_height="wrap_content"
        android:layout_gravity="center_horizontal"
        android:text="show"
        android:backgroundTint="@color/colorPrimary"
        android:textColor="@android:color/white"/>

  </LinearLayout>
```

Load XML Layout File and its elements from an Activity

When we have created the layout, we need to load the XML layout resource from our activity onCreate() callback method and access the UI element from the XML using findViewById.

```
override fun onCreate(savedInstanceState: Bundle?) {
    super.onCreate(savedInstanceState)
    setContentView(R.layout.activity_main)

    // finding the button
```

```
    val showButton = findViewById<Button>(R.id.showInput)


    // finding the edit text
    val editText = findViewById<EditText>(R.id.editText)
```

Here, we can observe the above code and finds out that we are calling our layout using the setContentView method in the form of R.layout.activity_main. Generally, during the launch of our activity, the *onCreate()* callback method will be called by the android framework to get the required layout for an activity.

Create elements in the Kotlin file Dynamically

We can create or instantiate UI elements or widgets during runtime by using the custom View and ViewGroup objects programmatically in the Kotlin file. Below is the example of creating a layout using LinearLayout to hold an EditText and a Button in an activity programmatically.

- **Kotlin**

```
import android.os.Bundle
import android.widget.Button
import android.widget.EditText
import android.widget.LinearLayout
import android.widget.Toast
import android.appcompat.app.AppCompatActivity


class MainActivity : AppCompatActivity() {


   override fun onCreate(savedInstanceState: Bundle?) {
      super.onCreate(savedInstanceState)
      setContentView(R.layout.activity_main)


      // create the button
      val showButton = Button(this)
      showButton.setText("Submit")
```

```
        // create the editText
        val editText = EditText(this)

        val linearLayout = findViewById<LinearLayout>(R.id.l_layout)
        linearLayout.addView(editText)
        linearLayout.addView(showButton)

        // Setting On Click Listener
        showButton.setOnClickListener
        {
           // Getting the user input
           val text = editText.text

           // Showing the user input
           Toast.makeText(this, text, Toast.LENGTH_SHORT).show()
        }
     }
  }
```

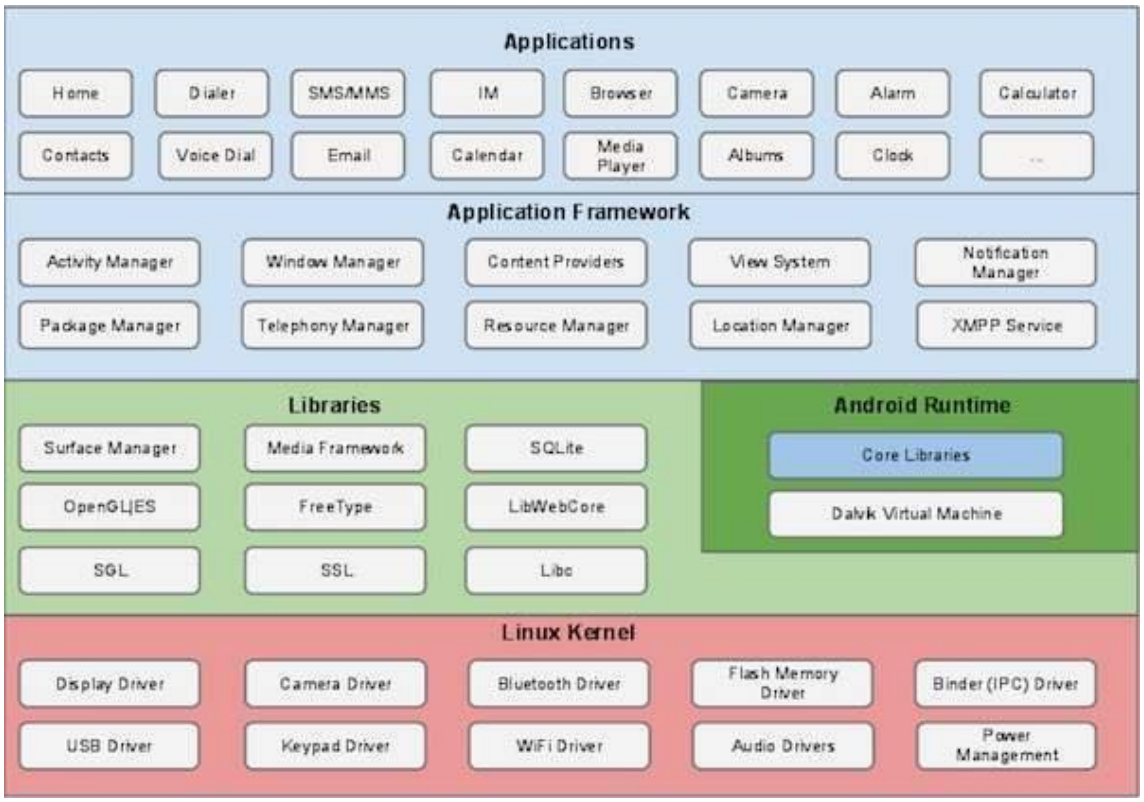
Different Attribute of the Layouts

XML attributes	Description
android:id	Used to specify the id of the view.
android:layout_width	Used to declare the width of View and ViewGroup elements in the layout.

XML attributes	Description
android:layout_height	Used to declare the height of View and ViewGroup elements in the layout.
android:layout_marginLeft	Used to declare the extra space used on the left side of View and ViewGroup elements.
android:layout_marginRight	Used to declare the extra space used on the right side of View and ViewGroup elements.
android:layout_marginTop	Used to declare the extra space used in the top side of View and ViewGroup elements.
android:layout_marginBottom	Used to declare the extra space used in the bottom side of View and ViewGroup elements.
android:layout_gravity	Used to define how child Views are positioned in the layout.

https://github.com/android/architecture-samples

Android operating system is a stack of software components which is roughly divided into five sections and four main layers as shown below in the architecture diagram.



Linux kernel

At the bottom of the layers is Linux - Linux 3.6 with approximately 115 patches. This provides a level of abstraction between the device hardware and it contains all the essential hardware drivers like camera, keypad, display etc. Also, the kernel handles all the things that Linux is really good at such as networking and a vast array of device drivers, which take the pain out of interfacing to peripheral hardware.

Libraries

On top of Linux kernel there is a set of libraries including open-source Web browser engine WebKit, well known library libc, SQLite database which is a useful repository for storage and sharing of application data, libraries to play and record audio and video, SSL libraries responsible for Internet security etc.

Android Libraries

This category encompasses those Java-based libraries that are specific to Android development. Examples of libraries in this category include the application framework libraries in addition to those that facilitate user interface building, graphics drawing and database access. A summary of

some key core Android libraries available to the Android developer is as follows −

- **android.app** − Provides access to the application model and is the cornerstone of all Android applications.
- **android.content** − Facilitates content access, publishing and messaging between applications and application components.
- **android.database** − Used to access data published by content providers and includes SQLite database management classes.
- **android.opengl** − A Java interface to the OpenGL ES 3D graphics rendering API.
- **android.os** − Provides applications with access to standard operating system services including messages, system services and inter-process communication.
- **android.text** − Used to render and manipulate text on a device display.
- **android.view** − The fundamental building blocks of application user interfaces.
- **android.widget** − A rich collection of pre-built user interface components such as buttons, labels, list views, layout managers, radio buttons etc.
- **android.webkit** − A set of classes intended to allow web-browsing capabilities to be built into applications.

Having covered the Java-based core libraries in the Android runtime, it is now time to turn our attention to the C/C++ based libraries contained in this layer of the Android software stack.

Android Runtime

This is the third section of the architecture and available on the second layer from the bottom. This section provides a key component called **Dalvik Virtual Machine** which is a kind of Java Virtual Machine specially designed and optimized for Android.

The Dalvik VM makes use of Linux core features like memory management and multi-threading, which is intrinsic in the Java language. The Dalvik VM

enables every Android application to run in its own process, with its own instance of the Dalvik virtual machine.

The Android runtime also provides a set of core libraries which enable Android application developers to write Android applications using standard Java programming language.

Application Framework

The Application Framework layer provides many higher-level services to applications in the form of Java classes. Application developers are allowed to make use of these services in their applications.

The Android framework includes the following key services −

- **Activity Manager** − Controls all aspects of the application lifecycle and activity stack.
- **Content Providers** − Allows applications to publish and share data with other applications.
- **Resource Manager** − Provides access to non-code embedded resources such as strings, color settings and user interface layouts.
- **Notifications Manager** − Allows applications to display alerts and notifications to the user.
- **View System** − An extensible set of views used to create application user interfaces.

Applications

You will find all the Android application at the top layer. You will write your application to be installed on this layer only. Examples of such applications are Contacts Books, Browser, Games etc.

https://www.tutorialspoint.com/android/android_architecture.htm

Testing Environment

Application fundamentals

Android apps can be written using Kotlin, Java and C++ languages. The Android SDK tools compile the code together with all the data and resource files into an APK, an *Android package* , which is a .apk. APK files contain all the content of an Android app and are the files that devices built for Android use to install the app.

Each Android app is activated in its own security sandbox, protected by the following Android security features:

- The Android operating system is a multi-user Linux system where each application is a different user.
- By default, the system assigns each application a unique Linux user code (the code is used only by the system and is unknown to the application). The system sets permissions for all files in an application so that only the user code assigned to that application can access them.
- Each process has its own virtual machine (VM), so an application's code runs in isolation from other applications.
- By default, each application runs in its own Linux process. Android starts the process when it needs to run some application component. It then shuts it down when it is no longer needed or when the system needs to reclaim memory for other applications.

The Android system implements the *principle of least privilege* . That is, each application, by default, has access only to the components it needs to do its job and nothing else. This creates a very secure environment where the application cannot access parts of the system that it does not have permission to. However, there is always a way for an app to share data with other apps and access system services:

- It is possible to have two applications share the same Linux user code, in which case they are able to access each other's files. To preserve system resources, applications with the same user code can also be combined to run in the same Linux process and share the same VM. You must also assign the same certificate to the applications.
- An app may request permission to access device data such as user contacts, SMS messages, mountable storage (SD card), camera, Bluetooth, etc. The user must explicitly grant these permissions. To learn more, see Working with System Permissions .

The remainder of this document introduces the following concepts:

- Fundamental library components that define the application.
- The manifest file where you declare the required device components and features for the application.
- Separate app code features that allow you to optimize the behavior of a variety of device configurations.

application components

App components are the building blocks of an Android app. Each component is an entry point through which the system or user can enter the application. Some components depend on others.

There are four different types of application components:

- Activities
- services

- Broadcast receivers
- content providers

Each type has a distinct purpose and has a specific lifecycle that defines how the component is created and destroyed. The following sections describe the four types of application components.

Activities

An *activity* is the entry point for user interaction. It represents a single screen with a user interface. For example, an email app might have an activity that shows a list of new emails, another activity that makes up an email, and another that reads emails. While these activities work together to form a cohesive user experience in the email app, they are independent of each other. Therefore, a different app can initiate any of these activities (if the email app allows it). For example, a camera app might initiate activity in the email app that composes the new email for the user to share a photo. An activity facilitates the following key interactions between system and application:

- Tracking what the user currently cares about (what is on the screen) to ensure the system remains running processes that host the activity.
- Knowledge of previously used processes that contain things the user can return to (interrupted activities) and therefore prioritization of maintaining those processes.
- Assistance to the application regarding interrupted processes so that the user can return to activities with the previous state restored.
- Providing a way for applications to implement user flows between themselves and for the system to coordinate those flows. Here goes the most classic example of all.

You implement an activity as a subclass of the Activity. For more information about the class , see the ActivitiesActivity developer guide .

services

the *service*it's an entry point to keep an app running in the background, whatever the reason. It is a component that runs in the background to perform long-running operations or work for remote processes. Services do not have a UI. For example, a service can play music in the background while the user is in a different application or fetch data on the network without blocking the user's interaction with an activity. Another component, such as an activity, can start the service and let it run or bind to it to interact. There are actually two very different semantic services that tell the system how to manage an application: started services tell the system to keep them running until their work is done. This could be background data syncing or music playing even after the user exits the app. Background data sync or music playback also represent two different types of services started that modify the way the system handles them:

- Playing music is something that is directly perceptible to the user, so the application informs the system about this, saying that it needs to be in the foreground, with a notification that warns the user about this situation. In this case, the system understands that it is necessary to try as much as possible to keep the service process running, because if it disappears, the user will be harmed.

- A common background service is not something that the user is directly aware of while running. In this way, the system has greater freedom to manage the processes. Interrupt (and restart of the process at some future time) can be allowed if RAM is needed for things immediately more important to the user.

Linked services run because some other application (or the system) tells you it needs to use them. It's basically the service providing an API to another process. The system then realizes that there is a dependency between these processes. So if process A is linked to a service in process B, it knows that it needs to keep B (and the service related to it) running for A. Also, if process A is important to the user, he knows how to treat process B as something also important. Because they have flexibility (for better or worse), services have become a very useful building block for all sorts of higher-level system concepts. Live wallpapers, notification detectors, screen savers, input methods,

Services are implemented as a subclass of Service. For more information about the class , see the ServicesService developer guide .

Note: If your app targets Android 5.0 (API level 21) or higher, use the class **JobScheduler**to program actions. By working with the Soneca API and scheduling jobs optimally, which reduces energy consumption, JobScheduler has the advantage of saving battery power. For more information on using this class, see the **JobScheduler**.

Broadcast receivers

the *broadcast receiver*it is a component that makes the system deliver events to the application outside the common user stream. This allows the app to respond to system-wide broadcast ads. Since broadcast receivers are more of a well-defined input to the application, the system is able to deliver broadcasts even to applications that are not currently running. For example, an application can schedule an alarm to post a notification that alerts the user of an upcoming event. By delivering this alarm to a broadcast receiver, the application does not need to remain running until the alarm is deactivated. Many streams originate from the system — for example, a stream that reports screen off, low battery, or image capture. Applications can also initiate broadcasts — for example, to let other apps know that some data has been saved on the device and is ready for use. Although broadcast receivers do not display any user interfaces, they cancreate a status bar notification to alert the user when a broadcast event occurs. More commonly, however, a broadcast receiver is just a *gateway* to other components and does a minimal amount of work. For example, it can program one JobServiceto run a job based on the event with JobScheduler.

Broadcast receivers are implemented as subclasses of BroadcastReceiverand each broadcast is delivered as an object Intent. For more information, see the BroadcastReceiver.

content providers

Content providers manage a shared set of application data that you can store on file systems, SQLite databases, the web, or any persistent storage location accessible to your application. Through the content provider, other applications can query or even modify the data, if the content provider allows it. For example, the Android system offers a content provider that manages the user's contact data. So any app with the proper permissions can query part of the content provider (such asContactsContract.Data) to read and write information about a specific person. It's tempting to think of content providers as an

abstraction in a database, because there are plenty of APIs and built-in support for them for this common case. However, from a systems development perspective, they have a different primary purpose. To the system, the content provider is an entry point into an application for publishing named data items, identified by a URI scheme. So an application can decide how it wants to map the data it contains to a namespace URI. This passes these URIs to other entities, which can then use them to access the data. There are a few things in particular that this allows the system to do in terms of managing an application:

- Assigning a URI does not require the application to remain running. That way, the URIs can persist after their own applications have been terminated. When it is necessary to retrieve application data at the corresponding URI, the system only needs to ensure that a belonging application is still running.
- URIs also provide an important security model for precise control. For example, an application can replace the URI with an image that is on the clipboard, but leave the content provider locked so that other applications cannot freely access it. When a second application tries to access the URI on the clipboard, the system may allow access by *temporarily granting the URI permission* . This allows access to data only behind that URI, but nothing else in the second application.

Content providers are useful for reading and writing data that is private in the application and not shared.

A content provider is implemented as a subclass of ContentProviderand must implement a standard set of APIs that allow other applications to transact. For more information, see the Content Providers developer guide .

A unique aspect of Android system design is that any application can launch a component of another application. For example, if you want the user to capture a photo with the device's camera, there will probably be another app that does this and your app can use it, that is, you don't need to develop an activity to capture a photo. No need to embed or even link to camera application code. Instead, you can simply launch the activity in the camera app that captures a photo. When complete, the photo even returns to the app in question for use. To the user, it feels like the camera is actually part of the app.

When the system starts a component, it starts the process for that application (if it is not already running) and instantiates the necessary classes for the component. For example, if the app starts the activity in the camera app that captures a photo, that app runs in the process that belongs to the camera app, not the app's process. So, unlike apps on most other systems, Android apps don't have any single entry point (there's no main(), for example).

Because the system runs each application in a separate process with file permissions that restrict access to other applications, the application cannot directly activate a component of another application. But the Android system manages to do that. Therefore, to activate a component in another application, you must send a message to the system that specifies the *intent* to launch a specific component. Then the system activates the component.

Activation of components

Three of the four types of components — activities, services, and broadcast receivers — are activated by an asynchronous message called an *intent* . Intents link individual components to

each other in the execution environment. Think of them as messengers that request an action from other components, whether the component belongs to the application or not.

The intent is created with an object Intentthat defines a message to activate a specific component or a specific *type* of component. Intents can be explicit or implicit, respectively.

For activities and services, intents define the action to perform (for example, *view* or *send* something) and can specify the URI of the data used in the action, among other things the initiating component needs to know. For example, an intent might broadcast a request for an activity to display an image or open a webpage. In some cases, you need to start an activity to receive a result. In this case, the activity also returns the result in a Intent. For example, you can issue an intent for the user to select a personal contact and return it to you. The return intent contains a URI that points to the selected contact.

For broadcast receivers, the intent simply defines the advertisement being broadcast. For example, a broadcast to indicate that the device's battery is running low contains a known action string that indicates *low battery level* .

Unlike activities, services, and broadcast receivers, content providers are not intent-activated. Instead, they are activated when targeted by a ContentResolver. The content resolver handles all direct transactions with the content provider so that the component performing transactions with the provider doesn't have to and instead calls the methods on the ContentResolver. This leaves an abstraction layer between the content provider and the component requesting information (for security purposes).

There are two methods to activate each component type:

- You can start an activity (or give it something new to do) by passing an Intenta startActivity()or startActivityForResult()(so that, when desired, the activity returns a result).
- With Android 5.0 (API level 21) and later, you can use the class JobSchedulerto program actions. For earlier versions of Android, you can start a service (or give new instructions to a service in progress) by passing an Intenta startService(). You can also link to the service by passing an Intenta bindService().
- You can start a broadcast by passing one Intentto methods like sendBroadcast(), sendOrderedBroadcast()or sendStickyBroadcast().
- You can initiate a query to a content provider by calling query()on a ContentResolver.

To learn more about intents, see the Intents and Intent Filters document . The following documents provide more information on enabling specific components: Activities , Services , BroadcastReceiverand Content Providers .

the manifest file

Before the Android system launches an application component, it must read the application *'s manifest file* AndroidManifest.xml so that the system knows that the component exists. The application must declare all of its components in this file, which must be in the root of the application's project directory.

The manifest does other things besides declaring the application's components, for example:

- Identifies all user permissions that the app needs, such as internet access or read-only access to the user's contacts.
- Declares the minimum API level required by the application based on the APIs the application uses.
- Declares the hardware and software features used or required by the application, such as camera, Bluetooth services, or multi-touch screen.
- Declares the API libraries the app needs to link to (in addition to the Android library APIs), such as the Google Maps Library .

Declaration of components

The main task of the manifest is to inform the system of the application's components. For example, a manifest file might declare an activity as follows:

<?xml version="1.0" encoding="utf-8"?><manifest ... ><applicationandroid:icon="@drawable/app_icon.png" ... ><activityandroid:name="com.example.project.ExampleActivity"android:label="@string/example_label" ... ></activity> ...</application></manifest>

In the element <application>, the attribute android:iconpoints to features of an icon that identify the application.

In the element <activity>, the attribute android:namespecifies the name of the fully qualified class of the subclass of, Activityand the attribute android:labelspecifies a string to use as the label for the user-visible activity.

You must declare all application components that use the following elements:

- Elements <activity>for activities
- Elements <service>for services
- Elements <receiver>for broadcast receivers
- Elements <provider>for content providers

Activities, services, and content providers included in the source, but not declared in the manifest, are not visible to the system and, as a result, may not run. However, broadcast receivers can be declared in the manifest dynamically in code (as objects BroadcastReceiver) and registered in the system by calling registerReceiver().

To learn more about the application manifest file structure, see the documentation The AndroidManifest.xml file .

Component Resource Declaration

As discussed above, in Activating Components , you can use one Intentto start activities, services, and broadcast receivers. You can use an Intentexplicitly naming the target component (using the component's class name) in the intent. You can also use an implicit intent. It describes the type of action to be performed and provides the option to enter the data on which you want to perform it. The implicit intent allows the system to find a component on the device that can perform the action and initiate it. If there is more than one component that can perform the action described by the intent, the user will select which one to use.

Caution: If you use an intent to launch an **Service**, use an explicit intent to verify that your app is secure. Using an implicit intent to start a service poses a security risk because it is not possible to determine which service will respond to the intent, and the user will not be able to see which service is started. Starting with Android 5.0 (API level 21), the system throws an exception when calling **bindService()**with an implicit intent. Do not declare intent filters for your services.

For the system to identify the components that can respond to an intent, the received intent is compared with the intent *filters* provided in the manifest file of other applications on the device.

When declaring an activity in your app's manifest, you can include intent filters that declare the activity's capabilities so that it responds to intents from other apps. To declare an intent filter in the component, an element is added <intent-filter>as a child of the component's declaration element.

For example, if you're building an email app with a compose a new email activity, you could declare an intent filter to respond to “send” intents (to send a new email), like this:

```
<manifest ... >
  ...
  <application ...
><activityandroid:name="com.example.project.ComposeEmailActivity"><intent-
filter><actionandroid:name="android.intent.action.SEND"/><dataandroid:type="*/*"/><categ
oryandroid:name="android.intent.category.DEFAULT"/></intent-
filter></activity></application></manifest>
```

Then, if another app creates an intent with the action ACTION_SENDand passes it to startActivity(), the system can start the activity so the user can draft and send an email.

To learn more about intent filters, see the Intents and Intent Filters document .

Application requirements statement

There are several devices developed for Android and not all of them have the same features and characteristics. To prevent the application from being installed on devices that do not contain the features the application requires, it is important to define a profile for the device types supported by the application. You must declare the device and software requirements in the manifest file. Most of these statements are informational only and the system does not read them, but external services such as Google Play do read them to provide a filter to users when they search for these apps for their device.

For example, if your app requires a camera and uses APIs introduced in Android 2.1 (API level 7), you would need to declare those requirements in the manifest file as follows:

<manifest ... ><uses-featureandroid:name="android.hardware.camera.any"android:required="true"/><uses-sdkandroid:minSdkVersion="7"android:targetSdkVersion="19"/> ...</manifest>

So devices that *do not* have a camera and have an Android version *older* than 2.1 will not be able to install the Google Play application. However, it is also possible to declare that the app uses the camera as a *non-mandatory* feature . In this case, the app needs to set the attribute requiredto falseand check at runtime if the device has a camera to disable camera features as needed.

For more information on managing application compatibility with different devices, see the Device Compatibility document .

app features

Android apps are made up of more than just code — they require resources separate from the source code, such as images, audio files, and anything else related to the visual presentation of the app. For example, you need to define animations, menus, styles, colors, and the layout of activity UIs with XML files. Using application features makes it easy to update many features of the application without having to modify the code. Providing the alternate feature sets allows you to optimize the application for various device configurations such as different languages and screen sizes.

For every resource included in the Android project, the SDK programming tools define a unique integer ID that the programmer can use to reference the resource from application code or other resources defined in XML. For example, if the app contains an image file named logo.png(saved in the directory res/drawable/), the SDK tools will generate a feature code called R.drawable.logo. This code maps to an application-specific integer value, which you can use to query the image and insert it into your UI.

One of the most important aspects of providing resources separate from the source code is the ability to provide alternate resources for different device configurations. For example, when defining UI strings in XML, you can convert the strings to other languages and save them in separate files. Then, based on a language *qualifier* appended to the resource directory name (such as res/values-fr/for French string values) and the user's language setting, the Android system applies the appropriate language strings to the UI.

Android supports multiple *qualifiers*for alternative resources. The qualifier is a short string added to the name of resource directories to define the device configuration on which these resources will be used. Another example: it is important to create different layouts for activities depending on the device's screen size and orientation. When the device screen is in portrait (vertical) orientation, a layout with vertical buttons can be useful, but when the screen is in landscape (horizontal) orientation, the buttons need to be aligned horizontally. To change the layout as per the orientation, you can define two different layouts and apply the appropriate qualifier to the directory name of each layout. Then the system automatically applies the proper layout as per the current device orientation.

For more information about the different types of features to include in the app and how to create alternate features for different device configurations, read Providing features . To learn more about best practices and developing robust, production-quality applications, see Guide to Application Architecture .

App manifest overview

Every app project needs to have a file AndroidManifest.xml(with that exact name) at the root of the project's source set . The manifest file describes essential app information for the Android build tools, the Android operating system, and Google Play.

Among many other things, the manifest file needs to declare the following items:

- The package name of the application, which normally corresponds to the namespace of your code. When the project is built, the Android build tools use this data to determine the location of code entities. When packaging the app, the build tools replace this value with the app ID from the Gradle build files. This code is used as the unique identifier of the app on the system and on Google Play. Read more about package name and application ID .
- Application components, which include all services, broadcast receivers, content providers and activities. Each component needs to define basic properties like the Kotlin or Java class name. In addition, it can declare features such as which device settings can be processed and the intent filters that describe how the component is initialized. Read more about application components .
- The permissions the app must have to access protected parts of the system or other apps. It also declares all permissions that other apps need to have to access content from that app. Read more about permissions .
- The hardware and software resources required by the app, which affect which devices can install the app from Google Play. Read more about device compatibility .

If you're using Android Studio to build the app, the manifest file is built for you, and most of the essential elements of the manifest are added during that build, particularly when using code templates .

file resources

The following sections describe how some of the most important characteristics of the application are reflected in the manifest file.

Package name and Application ID

The root element of the manifest file requires an attribute for the application's package name (which typically corresponds to the project's directory structure, the Java namespace).

For example, the following snippet shows the <manifest>root element with the package name "com.example.myapp":

<?xml version="1.0" encoding="utf-8"?><manifestxmlns:android="http://schemas.android.com/apk/res/android"**package="com.example.myapp"**android:versionCode="1"android:versionName="1.0"> ...</manifest>

When building your app into the final app package (APK), Android build tools use the attribute packagefor two things:

- This name is applied as the namespace of R.javathe application's generated class, which is used to access your application resources .

Example: with the above manifest, the class Ris created in com.example.myapp.R.

- This name is used to resolve all relative class names declared in the manifest file.

Example: With the above manifest, an activity declared as <activity android:name=".MainActivity">is resolved to be com.example.myapp.MainActivity.

So the name in packagethe manifest attribute must always match the name of the base package of the project where you keep the activities and the rest of the application code. Of course, you can have other subpackages in the project, but those files need to import the class R.javausing the package.

However, after the APK is built, the attribute will packagealso represent the universally unique app ID of your app. After the above tasks are performed by the build tools based on the name of package, the value of packageis replaced by the value determined in the property applicationIdin build.gradleyour project file (used in Android Studio projects). This final attribute value packageneeds to be universally unique because this is the only guaranteed way to identify the app on Google Play and the system.

The difference between the name of the packagein the manifest and the one applicationIdin the file build.gradlecan be a bit confusing. But if you keep the same name, there's nothing to worry about.

However, if you want the namespace of your code (and therefore the name of the packagein the manifest) to be different from applicationIdthat of the build file, make sure you fully understand the implications of setting the Application ID . packageThis page explains how to securely adjust the manifest name regardless of applicationIdthe file version and change the application ID for other build settings.

application components

For each application component that is created in your application, you need to declare a corresponding XML element in the manifest file:

- <activity> for each subclass of Activity.
- <service> for each subclass of Service.
- <receiver> for each subclass of BroadcastReceiver.
- <provider> for each subclass of ContentProvider.

If you subclass any of these components without declaring it in the manifest file, the system will not be able to launch it.

Your subclass name needs to be specified with the attribute name, using the entire package designation. For example, a subclass Activitymight be declared like this:

<manifest ... ><application ... ><activityandroid:name="com.example.myapp.MainActivity" ... ></activity></application></manifest>

However, if the first character in the value nameis a period, the application package name (from packagethe element attribute <manifest>) is prefixed to the name. For example, the following activity name resolves to `"com.example.myapp.MainActivity"`:

<manifest**package="com.example.myapp"** ... ><application ... ><activity**android:name=".MainActivity"** ... > ...</activity></application></manifest>

If you have application components that are located in subpackages (like com.example.myapp.purchases), the value nameneeds to add the missing subpackage names (like ".purchases.PayActivity") or use the fully qualified package name.

Intent filters

Application activities, services, and broadcast receivers are enabled by *intents* . The intent is a message defined by an object Intentthat describes an action to be performed, including data used in actions, the category of component that will perform the action, and other instructions.

When an app sends an intent to the system, the system finds an app component that can process the intent based on the intent *filter* declarations in the app's manifest file. The system launches an instance of the corresponding component and passes the object Intentto that component. If more than one application can handle the intent, the user can choose which one to use.

An application component can have multiple intent filters (defined with the element <intent-filter>), each describing a different feature of the component.

For more information, see the Intents and Intent Filters document .

Icons and Labels

Several manifest elements have iconand attributes labelto display, respectively, a small icon and a text label to users for the corresponding application component.

In all cases, the icon and label defined on a parent element become the default values icon for labelall child elements. For example, the icon and label defined on the element <application>are the default for all application components (like all activities).

The icon and label defined in <intent-filter>the component are displayed to the user whenever the component is presented as an option to fill an intent. By default, this icon inherits from any icon that is declared for the parent component (the <activity>or element <application>). However, you might want to change an intent filter's icon if it provides a single action that you want to better indicate in the selector dialog. For more information, see Allow other apps to start their activity .

permissions

Android apps need to ask for permission to access sensitive user data (like contacts and SMS) or certain system features (like camera and internet access). Each permission is identified by a unique label. For example, an application that needs to send SMS messages needs to have the following line in the manifest:

<manifest ... ><uses-permissionandroid:name="android.permission.SEND_SMS"/> ...</manifest>

Starting with Android 6.0 (API level 23), the user can approve or reject some application permissions in the runtime. But regardless of which version of Android your app supports, you need to declare all permissions requests with an element <uses-permission>in the manifest. If the permission is granted, the app will be able to use the protected resources. Otherwise, the attempt to access these resources will fail.

Your app can also protect the components themselves with permissions. It can use any permissions defined by Android as listed in android.Manifest.permission, or a permission declared in another app. Your app can also set its own permissions. The new permissions are declared with the <permission>.

For more information, see Permissions Overview .

device compatibility

The manifest file is also where you can declare what types of hardware or software resources the application requires and what types of devices it supports. The Google Play Store does not allow your app to be installed on devices that do not provide the required features or system version.

There are several manifest tags that define which devices the app supports. Below are some of the most common tags.

<uses-feature>

The element allows you to declare the hardware and software resources that the application needs. For example, if your app is unable to perform basic functionality on a device that does not have a compass sensor, you can declare the compass sensor as mandatory with the following manifest tag: <uses-feature>

<manifest ... ><uses-featureandroid:name="android.hardware.sensor.compass"android:required="true"/> ...</manifest>

Note : If you want your app to be available for Chromebooks, there are some important hardware and software resource limitations that you need to consider. For more information, see App Manifest Compatibility for Chromebooks .

<uses-sdk>

Each successive version of the platform often adds new APIs not available in the previous version. To indicate the minimum version supported by the application, the manifest must include the tag <uses-sdk>and the minSdkVersion.

However, the attributes in the element <uses-sdk>are replaced by the corresponding properties in the build.gradle. So if you are using Android Studio, you need to specify the minSdkVersione values targetSdkVersionin the file:

```
android {
  defaultConfig {
    applicationId 'com.example.myapp'// Defines the minimum API level required to run the app.minSdkVersion 15// Specifies the API level used to test the app.    targetSdkVersion 28...}}
```

For more information about the file build.gradle, read about configuring your build .

To learn more about declaring your app's support for different devices, see the Device Compatibility Overview .

file conventions

This section describes the conventions and rules that generally apply to all elements and attributes in the manifest file.

Elements

Only the elements <manifest>and <application>. Both must occur only once. Most other elements can occur zero or more times. However, some of them need to be present to make the manifest file useful.

All values are defined through attributes, not as character data within an element.

Peer elements are generally not ordered. For example, the <activity>, <provider>and elements <service>can be positioned in any order. There are two main exceptions to this rule:

- An element <activity-alias>must follow <activity>whoever it is an alias.
- The element <application>must be the last element inside the element <manifest>.

attributes

Technically, attributes are optional. However, many of them need to be specified in order for an element to fulfill its purpose. For truly optional attributes, see the reference documentation that states the default values.

Except for some attributes of the <manifest>root element, all attribute names begin with the prefix android:. For example: android:alwaysRetainTaskState. Because the prefix is universal, the documentation often omits it when referring to attributes by name.

various values

If more than one value is specified, the element will always be repeated instead of listing the multiple values within a single element. For example, an intent filter might list some actions:

<intent-filter ...
><actionandroid:name="android.intent.action.EDIT"/><actionandroid:name="android.intent.action.INSERT"/><actionandroid:name="android.intent.action.DELETE"/> ...</intent-filter>

resource values

Some attributes have values that are displayed to users, such as an activity title or an application icon. The value of these attributes may differ depending on the user's language or other device settings (such as to provide a different icon size based on the device's pixel density). So the values can be set from a resource or theme, rather than hard-coded in the manifest file. Actual value may change based on alternate features provided for different device configurations.

Resources are expressed as values with the following format:

"@[*package*:]*type*/*name*"

You can omit the *package* name if the resource is provided by your application, including a dependency on libraries, because library resources are merged into your). When you want to use a feature from the Android library, the only valid package name is android.

The *type* is a resource type, such as a stringor a drawable, and the *name* is the name that identifies the specific resource. Let's look at an example:

<activityandroid:icon="@drawable/smallPic" ... >

For more information on adding resources to the project, read Providing resources .

To apply a value set in a theme , the first character needs to be ?instead of @:

"?[*package*:]*type*/*name*"

string values

Where an attribute value is a string, you must use double backslashes (\\) for escape characters, such as \\nfor a newline or \\uxxxxfor a Unicode character.

Manifest element reference

The following table provides links to reference documents for all valid elements in the AndroidManifest.xml.

<action>	Adds an action to a filters intent.
<activity>	Declares a component of the activity.
<activity-alias>	Declares an alias for the activity.
<application>	It is the application declaration.
<category>	Adds a category name to an intent filter.
<compatible-screens>	Specifies each screen setting that the application supports.
<data>	Adds a data specification to an intent filter.

<grant-uri-permission>	Specifies the subsets of application data that the parent content provider is allowed to acc
<instrumentation>	Declares a class Instrumentationthat allows you to monitor an application's interaction wi
<intent-filter>	Specifies what types of intents an activity, service, or broadcast receiver can respond to.
<manifest>	It is the root element of the AndroidManifest.xml file.
<meta-data>	It is a name-value pair for an additional and arbitrary data item that can be supplied to the
<path-permission>	Defines the path and required permissions for a specific subset of data within a content pr
<permission>	Declares a security permission that can be used to limit access to specific components or f others.
<permission-group>	Declares a name for a logical grouping of related permissions.
<permission-tree>	Declares the base name for a permissions tree.
<provider>	Declares a content provider component.
<receiver>	Declares a broadcast receiver component.
<service>	Declares a service component.
<supports-gl-texture>	Declares a single application-compatible GL texture compression format.
<supports-screens>	Declares the screen sizes supported by the application and activates the screen compatibi larger than what the application supports.
<uses-configuration>	Indicates the specific input capabilities required by the application.
<uses-feature>	Declares a single hardware or software resource used by the application.
<uses-library>	Specifies a shared library that the application needs to link to.
<uses-permission>	Specifies a system permission that must be granted by the user for the application to func
<uses-permission-sdk-23>	Specifies that an app wants a specific permission, but only if the app is installed on a devic level 23) or higher.
<uses-sdk>	Allows you to express an application's compatibility with one or more versions of the Andr integer.

Example manifest file

The XML below is a simple example of AndroidManifest.xmldeclaring two activities for the application.

```
<?xml version="1.0" encoding="utf-8"?>
<manifest
   xmlns:android="http://schemas.android.com/apk/res/android"
   android:versionCode="1"
   android:versionName="1.0"
   package="com.example.myapp">
```

```
  <!-- Beware that these values are overridden by the build.gradle file -->
  <uses-sdk android:minSdkVersion="15" android:targetSdkVersion="26" />

  <application
    android:allowBackup="true"
    android:icon="@mipmap/ic_launcher"
    android:roundIcon="@mipmap/ic_launcher_round"
    android:label="@string/app_name"
    android:supportsRtl="true"
    android:theme="@style/AppTheme">

    <!-- This name is resolved to com.example.myapp.MainActivity
        based upon the package attribute -->
    <activity android:name=".MainActivity">
      <intent-filter>
        <action android:name="android.intent.action.MAIN" />
        <category android:name="android.intent.category.LAUNCHER" />
      </intent-filter>
    </activity>

    <activity
      android:name=".DisplayMessageActivity"
      android:parentActivityName=".MainActivity" />
  </application>
</manifest>
```

Kali Linux

The Industry Standard

Kali Linux is not about its tools, nor the operating system. Kali Linux is a **platform**.

-

Make Your Job Easier

You can take any Linux and install pentesting tools on it, but you have to set the tools up manually and configure them. Kali is optimized to reduce the amount of work, so a professional can just sit down and go.

-

Kali Everywhere

A version of Kali is always close to you, no matter where you need it. Mobile devices, Docker, ARM, Amazon Web Services, Windows Subsystem for Linux, Virtual Machine, bare metal, and others are all available.

-

Customization

With the use of metapackages, optimized for the specific tasks of a security professional, and a highly accessible and well documented ISO customization process, it's always easy to generate an optimized version of Kali for your specific needs.

-

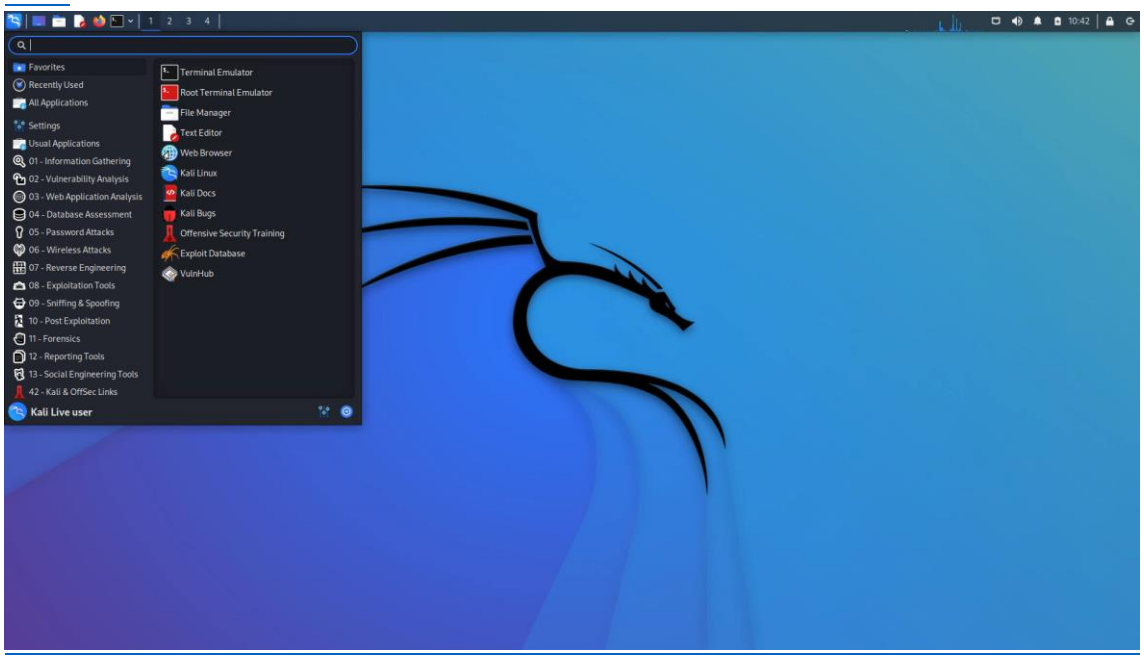
Documentation

Whether you are a seasoned veteran or a novice, our documentation will have all the information you will need to know about Kali Linux. Multiple tips and “recipes” are available, to help ease doubts or address any issues. All documentation is open, so you can easily contribute.

-

Community

Kali Linux, with its BackTrack lineage, has a vibrant and active community. There are active Kali forums, IRC Channel, Kali Tools listings, an open bug tracker system, and even community provided tool suggestions.



All the tools you need

The Kali Linux penetration testing platform contains a vast array of tools and utilities. From information gathering to final reporting, Kali Linux enables security and IT professionals to assess the security of their systems.

Mobile application specifics

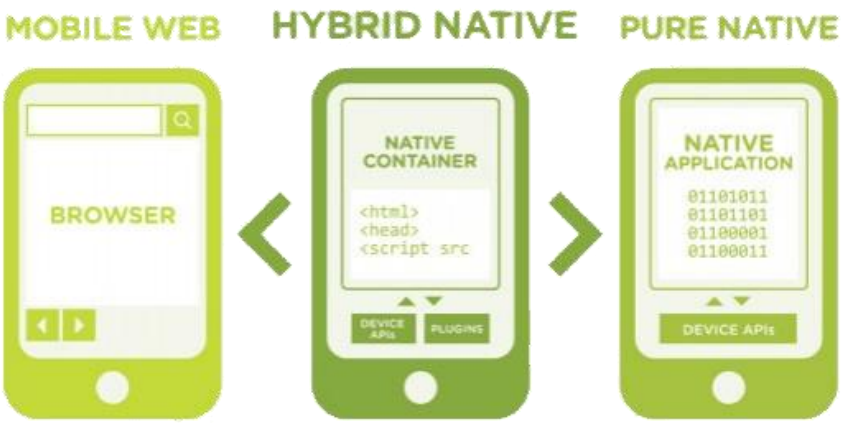
It is clear that the mobile application is very different from the desktop one. So, we should take this into account when planning the testing process.

So, let's consider the main **differences between mobile and desktop apps**:

- The mobile device is a system, that has not powerful stuffing. So, it can not work as a personal computer.
- The mobile application testing in provided on handsets (Apple, Samsung, Nokia, etc.), while the desktop app is tested on a central processor.
- Mobile devices screens variety, their extensions, and colors. Mobile phone screen size is smaller than desktop ones.
- Making and receiving calls is the main task of the phone, that is why the application should not interfere with this major function.
- A wide variety of specific operating systems and component configurations: Android, iOS, BlackBerry
  etc.
- Mobile phone OS quickly becomes obsolete. In addition, there is a limit to updating their OS.
- Mobile devices use network connections (3G, 4G, Wi-Fi), desktop use broadband connection or Wi-Fi.
- Mobile devices constantly search the network. That is why you should test the application at different data rates.
- Tools, which are good for the desktop apps testing, are not fully suitable for the mobile application testing.
- Mobile applications must support multiple input channels (keyboard, voice, gestures, etc.), multimedia technologies and other features that increase their usability.

Another important thing in mobile application testing process is the **type of application**.

Three main types of the mobile apps are divided: Mobile Web Apps, Native (Pure native) Apps, and Hybrid Apps.



Mobile Web application, in fact, is the website opened in the gadget (smartphone or tablet) with the help of the mobile browser.

Some merits of the Mobile Web Apps:

- Easy development.
- Easy access.
- Easy update.
- Mobile Web App requires no installation.

Some demerits of the Mobile Web Apps:

- No offline capabilities support.
- Limited functionality in the comparison with Hybrid and Native Apps. (no access to the file system and local resources).
- Problems with redistribution: Google Play and App Store don't support redistribution of the Mobile Web Apps.

Native App is the application, which has been developed specifically for one platform (Android, iOS, Tizen, Windows 10 M0bile, BlackBerry).

Some merits of the Native Apps:

- Native app works offline.
- It can use all features of its device.
- Advanced user experience.
- Push notifications can be used for users alert.

Some demerits of the Native Apps:

- Native Apps creation is expensive in comparison to the Mobile Web apps.
- It requires high costs for the maintenance.

Hybrid App is the mix of the Native App and Mobile Web App. It can be defined like mobile website content exposition in the application format.

Some merits of the Hybrid Apps:

- More cost effective in comparison to the Native App.
- Easy distribution.
- Embedded browser.
- Device features.

Some demerits of the Hybrid Apps:

- It works not so fast as Native App.
- Graphics are less accustomed to the OS in comparison to Native App.

Mobile site testing strategy key points

Now, we can think about our testing strategy. Let's consider the main points and challenges we should face to.

Devices selection

There is no doubt, that the real device is the best decision if you want to test mobile application. Testing on a real device always gives you the highest accuracy of results.

In fact, this is really not easy to choose the most appropriate device. Anyway, here are some actions you should do while selecting device for the mobile testing:

- Make the analysis to define the most popular and used gadgets in the market.
- Choose devices with different OS.
- Choose devices with different screen resolutions.
- Pay attention to the next factors: compatibility, memory size, connectivity etc.

As it was mentioned before you have lots of **advantages** for testing mobile apps on the real devices:

- High accuracy of the testing result.
- Simple bug replication.
- The points like battery drainage, geolocation, push notifications, devices built-in sensors are easy for testing.
- Ability to test incoming interrupts (calls, SMS).
- Ability to test mobile application in the real environment and conditions.
- No false positives.

And also some **disadvantages**:

- A huge number of the often used devices.
- Additional expenses for the maintenance of the devices.
- Limited access to the devices often used in the foreign countries.

As you can see testing on the real devices is the good decision, but also it has some limitations. You should overcome them to make mobile apps testing process real effective.

Emulators or simulators?

There is no difficult to guess, that they are special tools which emulate/simulate functionality and behavior of the mobile devices.

“Emulator” and “simulator meanings are often confused. Despite theirs almost similars pronunciation, they have no equal meaning.

In fact, an emulator is the original device replacement. Though you can run soft and apps on your gadget, you have no ability to modify them.

The simulator doesn’t replicate device’s hardware, but you have an ability to set up the similar environment as the original device’s OS.

So, it is better to use mobile simulators to test mobile application. Emulators are more appropriate for the mobile site testing.

Here, you can read more about emulators and simulators.

Some **advantages** of using the simulators to test mobile application:

- Easy setup.
- Fast working.
- Helps to verify and explore the behavior of your mobile app.
- Cost effective.

Some **disadvantages** of using the simulators to test mobile application:

- Device hardware is not taken into the consideration.
- False positives are possible.
- Incomplete data of the simulation results, which makes some difficulties for the complete analysis of the test results.

Cloud-based testing of the mobile application

Testing mobile applications with cloud-based tools seems to be the optimal choice. It can help you to overcome disadvantages of the real devices and simulators.

The main advantages of this approach:

- Easy availability.
- An ability to run mobile devices on multiple systems and networks.
- An ability not only to test, but also update and manage apps in the cloud.
- Cost effective.
- High scalability.
- The same script can be run on several devices in parallel.

Some weak points of the cloud mobile testing:

- Less of the control.
- No so high level of the security.
- Dependence of the Internet connection.

Some useful cloud-based tools, which can help you to test mobile application: **Xamarin Test Cloud**, **Perfecto Mobile Continuous Quality Lab**, **Keynote Mobile Testing**. Here you can read more about mobile testing tools.

Mobile manual and automated testing

Nowadays many specialists support the opinion that manual testing is going to die. Sure, it is not true. Of course, we can not do without test automation, but there also situations when manual testing is preferable.

Some **merits** of the manual mobile application testing:

- It is more cost effective in the short-termed period.
- Manual testing is more flexible.
- Better simulation of user actions.

Some **demerits** of the manual mobile application testing:

- Manual test cases are hard to be reused.
- Less effective of execution certain and constant task.
- Test running process is slow.
- Some kinds of test cases couldn’t be executed manually (load testing).

Some **advantages** of the app automation testing:

- Test running process is quite faster.
- Cost effective in the long-termed period.
- Automated test cases are easy to be reused.
- The only decision for some kind of testing (perfomance testing).
- Test results are easy to be shared.

Some **disadvantages** of the app automation testing:

- Automated mobile testing tools have some limitations.
- Time-consuming process.
- Automated testing has less efficiency in the defining user-friendliness or positive customer experience.

As you can see you should make different decisions creating your strategy for the mobile testing. Of course, there are no univocal answers on them.

The combination of different approaches seems to be the optimal way. For example, you can use simulators in the earliest stages of your testing process. But is better to use real devices (physical or cloud-based) in the final stages. Automated testing is preferable for the load and regression testing. But manual mobile testing tools are better to be used for usability and exploratory testing.

Mobile application testing stages

So, let's start to consider the main stages of the mobile app testing process. They more mostly similar to the website testing stages. Mostly, but not quite similar. As you have read before, there are some basic differences between mobile and desktop applications. Therefore, we need to pass some additional stages and make some additional verifications.

1. Documentation Testing

Documentation testing is the necessary preparatory stage of the mobile application testing process.

Actually, testing begins before software development process. Testers get navigational charts, screen layouts, other requirements invisible on the design. These requirements are analyzed

for completeness and inconsistency. Contradictions in the requirements must be resolved before the start of development.

Artifacts like Requirements (Specification, PRD), Test Plan, Test Cases, Traceability Matrix are created and analyzed on this stage.

2. Functional testing

Functional testing is aimed to ensure that it is working as per the defined requirements. In simple terms, we check whether the application performs the expected functions, which are usually described in the specification or correspond to the logic of business processes.Pay attention to the next important factors while providing functional testing of your mobile app:

- The application type, which is defined by its business functionality (social networks, banking, education, ordering and delivery of food, tickets, the game industry etc.).
- Target audience (companies, users, educational environment etc.).
- Distribution channels (direct delivery, Google Play, App Store, etc.)

Now, let's consider the main verifications, which should be passed to test mobile application functionality.

Installing and running the application

- The installation of the application should take place without significant errors, if the device meets the system requirements.
- Verify the application automatically starts correctly.
- Ensure the user manual is available.
- Ensure the application's operation during startup/exit meets the basic requirements.

Fields testing

- Verify the required fields work correctly.
- Make sure that mandatory and optional fields are displayed in different ways.

Business functionalities testing

- Verify the declared price and content correspond to the user got information.

- Ensure the user can perform typical operations: buying, adding goods to the cart, ordering goods etc.
- Make sure the application supports payment transactions through payment systems like Visa, Mastercard, Paypal etc.
- Check the recovery of the purchase regardless of the device, but with an account binding.

Interruptions testing

- Incoming and outgoing calls, SMS, and MMS.
- Battery discharge/removal.
- Disconnecting and connecting the network/Wi-Fi.
- Disconnecting and connecting the SD-card.
- Charging the device.

Constant users feedback testing

- Downloading content messages
- Progress bar.
- The appropriate reaction of the buttons on pressing.
- Network access error messages.
- Attempt to delete important information messages.
- Availability and synchronization of sound, vibration, and visual notifications.
- The appearance of a screen (message) at the end of the process (game).

Update testing

- All user data is saved after updates.
- Ensure the update progress is displayed properly.
- Make sure updates are supported by older operating systems.
- Testing various ways of installing updates (Wi-Fi, Bluetooth, USB)

Device resources testing

- Lack of space to install or run the application.
- Memory leaks. Pay attention to windows, with a lot of information, and tasks with long workflow.
- Installing/replacement the app on the SD-card.
- The absence of some functions supported by the application (3G, SD-card, etc.).
- Ensure the installed application does not interfere with the normal operation of other apps and does not consume their memory.

Some other verifications:

- Games concerned verifications: correctness of connecting/disconnecting players, players connection via different networks etc.
- Make sure the information error messages are correct on time and appropriate.
- Verify connection to the analytical tools like Google Analytics.
- Testing the power consumption.
- Verify the necessary options correct work with social networks – **Share, Publish, Navigation.**

Some useful tools to test mobile application functionality: **Appium, Selendroid, Robotium, Ranorex.**

3. Usability testing

Usability testing is aimed to ensure the convenience of using the application, creates an intuitive interface that conforms to accepted standards. It is performed to create fast and easy-to-use applications. Here are 3 main basic criteria for the apps evaluation:

- Satisfaction
- Efficiency
- Effectiveness

Let's consider the simple checklist to test mobile application usability:

- Make sure that the buttons are of the normal size and placed in one area of the screen
- Verify the app works in multitasking mode, when necessary.
- Check the navigation of the important application modules.
- Ensure the icons and pictures look natural in the app environment.
- Verify the color of the buttons that perform the same function is the same.

- The text should be simple, clear and visible to the user. Short sentences and paragraphs are possible to read.
- Define the optimal font size.
- Ensure correct operation of the Zoom-in and Zoom-out system.
- Verify the context menus are not overloaded.
- Make sure that the application can be terminated by any state and that it resumes operation in the same state.
- Ensure that the application components are synchronized with the user's actions.
- Verify the user can return or cancel the action if he/she pressed the wrong button.
- Verify the speed of response of the element is high enough

Some useful tools to test mobile application usability: **User Zoom, Reflector, Loop[11].**

4. UI (User Interface) testing

User Interface (UI) testing is performed to ensure the graphic user interface of your app meets the specifications.

Here are some verifications to test mobile application UI:

- Ensure the compliance with the standards of UI
- Check your app's UI with the standard screen resolutions: 640 × 480, 800 × 600, 1024 × 768, 1280 × 800, 1366 × 768, 1400 × 900, 1680 × 1050.
- Verify responsiveness of applications on different devices.
- Test the main design element: buttons, icons, colors, links, fonts, font sizes, layout, text boxes, text formatting, labels, captions, buttons, lists etc.
- Verify advertising does not overlap application control buttons.
- Ensure the advertising has an accessible closing button.
- Make sure the correct display of various elements on retina and non-retina screens.
- Verify all elements display with portrait and landscape page orientation.

Some useful tools to test mobile application interface: **FitNesse, iMacros, Coded UI, Jubula, LoadUI.**

5. Compatibility (Configuration) testing

Compatibility (Configuration) testing is conducted in order to ensure optimal application performance on different devices – taking into account their size, screen resolution, version, hardware, etc. You should pay attention to the next points:

- OS Configuration
- Browser Configuration
- Database Configuration
- Device Configuration
- Network Configuration

Cross-platform testing helps you to test mobile application in different OS: Windows, iOS, Android, and BlackBerry etc.

Cross-browser testing allows ensuring the correct work of the app in different browser configurations: Mozilla Firefox, Google Chrome, Opera Mini etc.

Database testing is aimed to verify the correct work of your application in different database configurations: Oracle, DB2, MySql, MSSQL Server, Sybase.

Device Configuration testing should take into account such parameters:

- Device type: smartphone, tablet, etc.
- Device configuration: RAM, processor type, screen resolution, battery capacity, etc.

Network configuration testing is performed to ensure the correct work in different network configurations (GSM, TDMA) and standards (2G, 3G, 4G).

Some tips to test your mobile application compatibility:

- Create a coverage matrix (the table in which all possible configurations are entered).
- Prioritize configurations.
- Check each configuration, step by step, in accordance with the set priorities.

Some useful tools to test mobile application perfomance compatibility: **BrowserStack, CrossBrowserTesting by Smart Bear, Litmus, Browsera, Rational Clearcase by IBM, Ghostlab.**

6. Perfomance testing

Performance testing is a set of types of testing, the purpose of which is to determine the operability, stability, resource consumption and other attributes of application quality under different usage scenarios and loads.

The main aims of the performance testing:

- Checking the response time of the application to various types of requests, in order to make sure that the application is working according to the requirements for the normal user load. **(Load testing).**
- Testing the working capacity of the application at loads exceeding the user's several times. **(Stress testing).**
- Examine the operability of the application for long time work, under normal load. **(Stability testing).**
- Check work in the conditions of the "expanded" database, under the normal time. **(Volume testing).**
- Determine the number of users who can simultaneously work with the application. **(Concurrency testing).**

Some verifications for perfomance testing your mobile app:

- Determine whether the application is running the same under different network conditions.
- Find various application and infrastructure bottlenecks that reduce application performance.
- Evaluate the ability of the app to cope with planned load volumes.
- Verify the response time of the application meets the requirements.
- Check the application stability under conditions of a hard user load.
- Ensure the performance of the application if it works under conditions of a non-permanent connection to the Internet.
- Make sure the existing client-server configuration provides optimal performance.

Some useful tools to test mobile application perfomance: **NeoLoad by Neotys**, **Apteligent (formerly Crittercism)**, **New Relic.**

7. Security testing

Security testing is aimed to check the security of the system, as well as to analyze the risks associated with providing a holistic approach to application protection, hackers, viruses, unauthorized access to sensitive data.

Some verifications you have to pass to test mobile application security:

- Ensure the data of users of the application **(logins, passwords, bank card numbers)** are protected from network attacks of automated systems and can not be found by selection.
- Verify the application security system requires a strong password and does not allow the attacker to seize the passwords of other users.
- Make sure that the application does not give access to sensitive content or functionality without proper authentication.
- Protect the application against attacks of the SQL injection type.
- Protect the application and the network from DoS Attacks.
- Protect the application from malicious attacks on clients.
- Protect the system from malicious implementations when the program is running.
- Provide session management to protect information from unauthorized users.
- Prevent possible malicious consequences of file caching.
- Examine user files and prevent their possible harmful effects.
- Analyze the interaction of system files, identify and correct vulnerabilities.
- Prevent possible malicious actions of cookies.

Some useful tools to test mobile application security: **Retina CS Community**, **OWASP Zed Attack Proxy**, **Veracode, Google Nogotofail,** and **SQL Map.**

8. Recovery testing

Recovery test verifies the app under test in terms of its ability to withstand and successfully recover from possible failures caused by software errors, hardware failures, or communication problems.

Here is the list of the verifications for the recovery testing:

- Verify the effective recovery of the application after unforeseen crash scenarios.
- Ensure the process of data recovery after a break in the connection.
- Test the recovery after a system failure and a transaction failure.
- Verify the ability of the application to process transactions in the event of a power failure (low battery, incorrect application shutdown etc.).

9. Localization testing

Localization testing allows you to test mobile application adaptation for a specific target audience in accordance with its cultural specifics.

Some verifications for the localization testing:

- Determine languages supported by the application.

- Ensure the correctness of the translation.
- Verify the correctness of the translation in accordance with the theme of the application
- Check the date formats.
- Check the delimiters in numbers.

Of course, the native speakers are preferred to perform localization testing of the mobile app.

Ubertesters, **eggPlant** can be useful to test mobile application localization.

10. Change related testing

So, you passed all mentioned stages and found some bugs. Therefore, some changes have been made to the code of your app.

The key goals of the change related testing:

- Verify your team has successfully fixed all detected bugs (**Re-testing or Confirmation testing**). Put it simply, the test cases that originally detected the bugs are run again. And this time they should be passed with no bugs.
- Verify the new changes did not lead to the appearance of new bugs. (**Regression testing**). Actually, providing regression testing, you should pass not only test cases with detected bugs, but also test cases checking all functionalities of your app.

Some useful tools for change related testing of your app: **Appium**, **Robotium**, **Ranorex.**

11. Beta testing

Finally, you have the prerelease full functionality version of your mobile app. It would be better to evaluate the possibilities and stability of the program in terms of its future users.

Beta testing is the stage of debugging and checking the beta version of the program. Its main purpose is identifying the maximum number of errors in its work for their subsequent elimination before the final release of the app to the market.

People who have experience with working with similar type apps, better yet, with the previous version of the application are chosen to the role of beta testers.

You should pay attention to the next factors before providing beta testing of your mobile app:

- A number of testing participants.
- Testing duration.
- Shipping
- Demographic coverage
- Testing costs.

Though you need to spend some money for beta testing, it could be a good investment in the quality of your mobile app.

Some popular platforms for beta testing of the mobile apps: **HockeyApp**, **Ubertesters**, **TestFlight** .

12. Certification testing

There are certain rules for organizing an installation file **(.apk)** and rules for applications design for each application store. Certification testing verifies the app meets the requirements of the most popular stores like Google Play, the App Store, and Windows Phone.

Let's consider the main criteria for application compliance with standards, licensing agreements and terms of use.

Android:

- The installation file for the application (.apk) matches with **Program Policies .**
- The application meets the requirements of the **UIG .**
- There are no viruses in the app. Android market semi-automatically checks the application for viruses and could block you account if detect them.
- You should follow the order of version control in the case of publishing an updated version of your app.

iOS:

- The application meets the requirements of the **Human Interface Guidelines.**
- The application must have a unique name.
- You need to provide a link for feedback from the developer.
- The application should be put to the determined particular category.
- App Store test the app for compatibility.
- App doesn't contains prohibited materials, unforeseen delays in work or repetition of existing functions.

Windows Phone

- The application meets the requirements of the **App certification requirements.**
- Clear description of the hardware and network requirements.
- The functions mentioned in the description or shown in the screenshots are fully realized
- Option to control auto-playable sound is required.

Tips to test mobile application

Let's systematize our knowledge, and try to determine the main tips for mobile application testing.

1. Learn the app you are going to test.
2. Remember the differencies between desktop and mobile apps.
3. Take into account the operating system and hardware specifics
4. Use real devices when it is possible.
5. Don't Try to Find the "Swiss Army Knife" of Testing. Use the tools you are familiar with.
6. Use the advantages of the cloud mobile testing.
7. Confirm your findings with screenshots, logs and videos.
8. Provide your mobile app testing both for portrait and landscape screen mode.
9. Use the development menu options for iOS and Android.
10. Do not neglect (but do not abuse) emulators and simulators for testing.
11. Verify perfomance of your app.
12. Don't automate everything
13. Get real users to test your app
14. Release the time to work out more complex, unconventional test scenarios (f.e. use test "monkeys").
15. Consider the human factor

1. Create a detailed plan

To yield the most effective results from a mobile app pentest, you need to first develop some sort of methodology as to how you plan to go about it. Obviously, each mobile app environment is going to be different from one another. Therefore, you should give careful thought and consideration to what exactly needs to be tested.

One of the best places to get started in this regard is this cheat sheet provided by OWASP. It is important to note it has been created for pentesting mobile apps in an iOS environment, but the same principles can still be applied to different situations.

2. Pick the right penetration testing tools

There are many pentesting tools available — some are vendor provided (for a cost), and also, many of them are free to download and use. Picking the right one(s) will depend primarily on the environment you are using. Here are a few of the most popular mobile pentesting tools available:

- Cydia
- Apktool
- Appcrack

- Burp Proxy
- Wireshark
- OWASP ZAP
- Tcdump

3. Prepare a thorough pentesting environment

You must plan your pentesting environment in great detail. For instance, since Apple theoretically has made it quite difficult to jailbreak an iPhone, it can still be done if the user knows what they are doing. Therefore, when pentesting in an iPhone environment, it will also be necessary to conduct a real word jail break in order to discover what the security ramifications will be. You can use the following resources:

- For iOS: www.evasi0n.com
- For Android: www.oneclickroot.com/

4. Manage your time wisely

Depending on the magnitude of the pentest you are conducting, you will need to have effective time management skills as well. For instance, there may be times when you are not testing the entire mobile app, just one portion of it. Therefore, use the right amount of time to do the test, and move onto the next item without sacrificing attention to detail.

5. Launch server attacks

It is equally important to test the server environment, as well as the server the app is hosted and downloaded from. In this regard, one of the more popular tools to use is Nmap. Some aspects that need to be pentested here include:

- The authentication mechanisms in place between the smartphone and the server (for example, Apple has numerous authentication steps a user has to take before he or she can download a mobile app from the Apple Store)
- Any authorized and unauthorized file uploads
- Any open redirects
- Cross origin resource sharing

6. Stay focused, be patient & above all be thorough

Remember, conducting a mobile app pentest can be a very tedious and laborious task, depending upon the actual magnitude of the test involved. It is often tempting to bypass a step, or even speed up the process of one segment of the plan. But, **never, ever do this**. Keep in mind it is your job to unearth any potential security vulnerabilities. For example, if it's discovered you purposely missed an important step in the pentesting plan, not only you, but the organization that you work for, could be held liable for any subsequent damages that may occur. In other words, follow this guiding principle: **Never assume a mobile app works. Always assume that it is broken in all ways possible.**

7. Launch network attacks

When pentesting network connectivity between the wireless device/smartphone and the server the mobile app will be downloaded from, always make use of network sniffers. These tools are used to collect important information and data not only about the network traffic itself, but also the data packets. From here, results can be used to ascertain and formulate the type of pentesting that needs to be done. No matter what, the following should be included:

- Examining the authentication, authorization and session management mechanisms deployed
- Examining the encryption protocols implemented

8. Make use of source instrumentation

This process involves creating a specialized piece of code and layering it onto the source code already being developed. The primary purpose of this is to create a "backdoor," to investigate the source code objects at a much more granular level. With this, you can diagnose any unknown errors or flaws in the source code that could prove to be a security vulnerability.

9. Keep your mobile app pentesting skills sharp by always practicing

It is very important to keep your skills sharp by constantly practicing as often as you can. The following websites offer tools in which you can further (and safely) hone in on your pentesting skills:

- The Androick Project Page
- Mobisec
- The Damn Vulnerable iOS Application (DVIA)

10. Conduct both binary and file-level analyses

In this regard, you are pentesting for specific application programming interface (API) calls that are inherently weak, and those files which have poor quality access controls embedded into them. In this instance, you should also check for the following:

- Buffer overflows
- Examining the potential for SQL Injection based attacks

Some popular tools to use here include the following:

- IDA
- The Hopper Disassembler

Android Development Java

https://developer.android.com/codelabs/build-your-first-android-app#0

https://www.udemy.com/course/master-android-7-nougat-java-app-development-step-by-step/

Java is one of the powerful general-purpose programming languages, created in 1995 by Sun Microsystems (now owned by Oracle). Java is Object-Oriented. However, it is not considered as pure object-oriented as it provides support for primitive data types (like int, char, etc). Java syntax is similar to C/C++. But Java does not provide low-level programming functionalities like

pointers. Also, Java code is always written in the form of classes and objects. Android heavily relies on the Java programming language all the SDKs required to build for android applications use the standard libraries of Java. If one is coming from a traditional programming background like C, C++, Java is easy to learn. So in this discussion, there is a complete guide to learn Java specifically considering Android App Development.

So in this article, we have covered the following things:

1. **Basics of Java**
2. **Decision Making Statements in Java**
3. **Type Conversion in Java**
4. **Comments in Java**
5. **Operators in Java**
6. **Strings in Java**
7. **Object-Oriented Programming Concepts in Java**
8. **Exception Handling in Java**
9. **Interfaces and Abstract Classes**
10. **Essential collections in Java required for Android Development**
11. **Miscellaneous**
12. **Complete Java Tutorial**

Step-by-Step Guide to Learn Java for Android App Development

Basics of Java

- How to start learning Java – understand the core introduction of the Java programming language.
- Setting up the environment – Setup IDE for writing programs in Java.
- The Hello World Example – The first Hello World program in Java.

- Java Class File – Basic entry point of Java programming, which is writing the main class.
- Java Identifiers – In Java, an identifier can be a class name, method name, variable name, or label.
- Data types in Java – Get to know what types of data types are supported by the Java programming language.
- Variables in Java – A variable is a name given to a memory location. It is the basic unit of storage in a program.
- Scope of Variables – The scope of a variable is the part of the program where the variable is accessible.
- Blank Final in Java – A final variable in Java can be assigned a value only once. We can assign a value either in the declaration or later.

Decision Making Statements in Java

- Decision Making in Java (if, if-else, switch, break, continue, jump) – A programming language uses control statements to control the flow of execution of a program based on certain conditions.
- Switch Statement in Java – The switch statement is a multi-way branch statement. It provides an easy way to dispatch execution to different parts of code based on the value of the expression.
- Loops in Java – Looping in programming languages is a feature that facilitates the execution of a set of instructions/functions repeatedly while some conditions are evaluated to be true.
- For-each loop in Java – For-each is another array traversing technique like for loop, while loop, do-while loop is introduced in Java5.

Type Conversion in Java

- Type conversion in Java with Examples – If the data types are compatible, then Java will perform the conversion automatically known as Automatic Type Conversion, and if not, then they need to be cast or converted explicitly.

Comments in Java

- Comments in Java – Comments take part in making the program become more human-readable by placing the details of code involved and proper use of comments makes maintenance easier and finding bugs easier.

Operators in Java

- Operators in Java – Java provides many types of operators which can be used according to the need. They are classified based on the functionality they provide.

Strings in Java

- String class in Java | Set 1 – String is a sequence of characters. In Java, objects of strings are immutable, which means constant and cannot be changed once created.

- StringBuffer class in Java – StringBuffer is a peer class of String that provides much of the functionality of strings.
- StringBuilder Class in Java with Examples – The StringBuilder in Java represents a mutable sequence of characters.

Object-Oriented Programming Concepts in Java

- Classes and Objects in Java – The basic OOPs components Class and Object in the java programming language.
- Different ways to create objects in Java – Get to know the various ways of creating objects in Java.
- Inheritance in Java – It is the mechanism in Java by which one class is allowed to inherit the features(fields and methods) of another class.
- Encapsulation in Java – Encapsulation is defined as the wrapping up of data under a single unit.
- Abstraction in Java – Data Abstraction is a property by virtue of which only the essential details are displayed to the user.
- Access Modifiers in Java – As the name suggests, access modifiers in Java help to restrict the scope of a class, constructor, variable, method, or data member.
- 'this' reference in Java – 'this' is a reference variable that refers to the current object.
- Overloading in Java – Overloading allows different methods to have the same name, but different signatures of methods.
- Overriding in Java – Overriding is a feature that allows a subclass or child class to provide a specific implementation of a method that is already provided by one of its super-classes or parent classes.
- Object class in Java – Object class is present in the java.lang package. Every class in Java is directly or indirectly derived from the Object class.
- Static class in Java – Some classes can be made static in Java. Java supports Static Instance Variables, Static Methods, Static Block, and Static Classes.

Exception Handling in Java

- Exceptions in Java – An exception is an unwanted or unexpected event that occurs during the execution of a program i.e at run time.
- Types of Exception in Java with Examples – Java also allows users to define their own exceptions.

Interfaces and Abstract Classes

- Interfaces in Java – Like a class, an interface can have methods and variables, but the methods declared in an interface are by default abstract.

- Access specifier of methods in interfaces – All methods in an interface are public, even if we do not specify public with method names. Also, data fields are public static final even if we do not mention them in field names.
- Access specifiers for classes or interfaces in Java – Methods and data members of a class/interface can have one of the following four access specifiers.
- Abstract Classes in Java – Java, a separate keyword abstract is used to make a class abstract.
- Difference between Abstract Class and Interface in Java – Get to know the differences between the interfaces and abstract classes.
- Anonymous Inner Class in Java – It is an inner class without a name and for which only a single object is created.

Essential collections in Java required for Android Development

- ArrayList in Java – ArrayList is a part of the collection framework and is present in the java.util package. It provides us with dynamic arrays in Java.
- HashMap in Java with Examples – It stores the data in (Key, Value) pairs, and you can access it via an index of another type.

Miscellaneous

- Java Naming Conventions – Naming conventions must be followed while developing software in Java for good maintenance and readability of code.
- Generics in Java – Generics mean parameterized types. The idea is to allow types (Ingers, strings, … etc, and user-defined types) to be a parameter for methods, classes, and interfaces.
- Annotations in Java – Annotations are used to provide supplemental information about a program.
- Lambda Expressions in Java 8 – Lambda expressions basically express instances of functional interfaces (An interface with a single abstract method is called a functional interface.

For a complete Java Tutorial, you may refer to this article: **Java Programming Language**

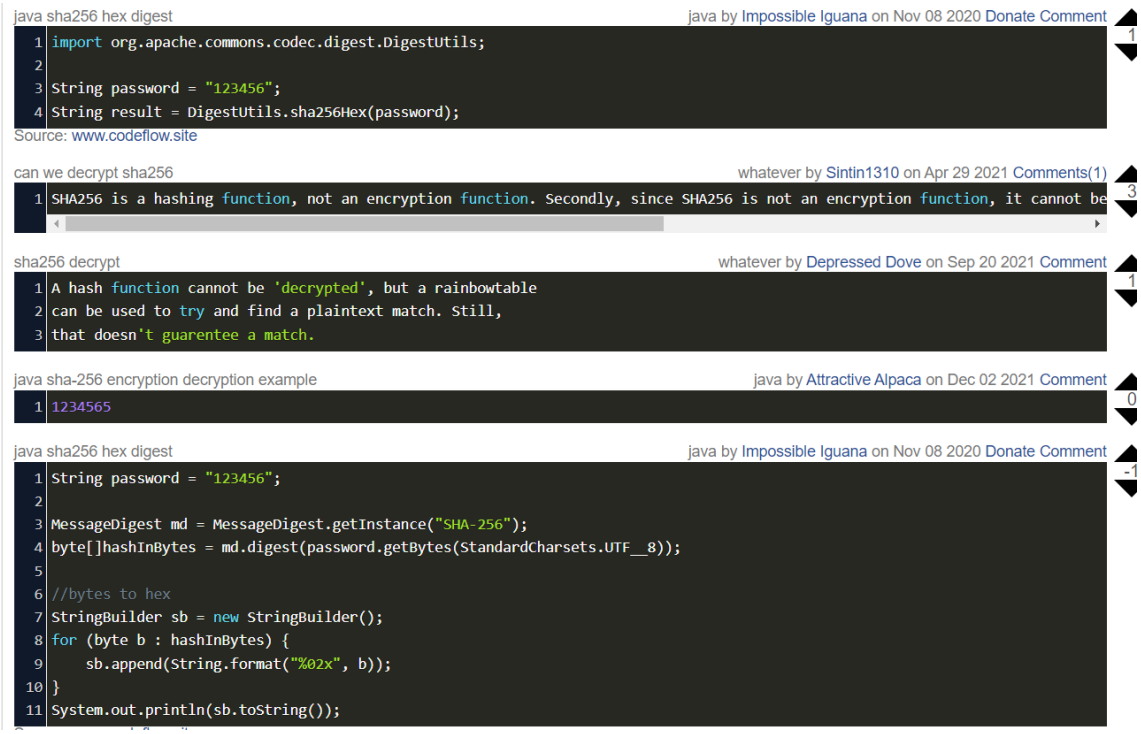
https://www.geeksforgeeks.org/learn-java-for-android-app-development-a-complete-guide/

Java Decrypt SHA256

https://xorbin.com/tools/sha1-hash-calculator

https://hashes.com/en/decrypt/hash

https://stackoverflow.com/questions/9316437/how-to-decrypt-a-sha-256-encrypted-string#:~:text=SHA%2D256%20is%20a%20cryptographic,and%20see%20if%20it%20matches.



https://www.codegrepper.com/code-examples/java/java+sha-256+encryption+decryption+example

1. Overview

The SHA (Secure Hash Algorithm) is one of the popular cryptographic hash functions. A cryptographic hash can be used to make a signature for a text or a data file.

In this tutorial, let's have a look at how we can perform SHA-256 and SHA3-256 hashing operations using various Java libraries.

The SHA-256 algorithm generates an almost unique, fixed-size 256-bit (32-byte) hash. This is a one-way function, so the result cannot be decrypted back to the original value.

Currently, SHA-2 hashing is widely used, as it is considered the most secure hashing algorithm in the cryptographic arena.

SHA-3 is the latest secure hashing standard after SHA-2. Compared to SHA-2, SHA-3 provides a different approach to generate a unique one-way hash, and it can be much faster on some hardware implementations. Similar to SHA-256, SHA3-256 is the 256-bit fixed-length algorithm in SHA-3.

NIST released SHA-3 in 2015, so there are not quite as many SHA-3 libraries as SHA-2 for the time being. It's not until JDK 9 that SHA-3 algorithms were available in the built-in default providers.

Now let's start with SHA-256.

2. *MessageDigest* Class in Java

Java provides inbuilt *MessageDigest* class for SHA-256 hashing:

MessageDigest digest = MessageDigest.getInstance("SHA-256");

```
byte[] encodedhash = digest.digest(
  originalString.getBytes(StandardCharsets.UTF_8));
```

However, here we have to use a custom byte to hex converter to get the hashed value in hexadecimal:

```
private static String bytesToHex(byte[] hash) {
    StringBuilder hexString = new StringBuilder(2 * hash.length);
    for (int i = 0; i < hash.length; i++) {
        String hex = Integer.toHexString(0xff & hash[i]);
        if(hex.length() == 1) {
            hexString.append('0');
        }
        hexString.append(hex);
    }
    return hexString.toString();
}
```

We need to be aware that the **MessageDigest is not thread-safe.** Consequently, we should use a new instance for every thread.

3. Guava Library

The Google Guava library also provides a utility class for hashing.

First, let's define the dependency:

```
<dependency>
    <groupId>com.google.guava</groupId>
    <artifactId>guava</artifactId>
    <version>31.0.1-jre</version>
</dependency>
```

Next, here's how we can use Guava to hash a String:

```
String sha256hex = Hashing.sha256()
  .hashString(originalString, StandardCharsets.UTF_8)
  .toString();
```

4. Apache Commons Codecs

Similarly, we can also use Apache Commons Codecs:

```
<dependency>
```

```
    <groupId>commons-codec</groupId>
    <artifactId>commons-codec</artifactId>
    <version>1.11</version>
</dependency>
```

Here's the utility class — called *DigestUtils* — that supports SHA-256 hashing:

5. Bouncy Castle Library

5.1. Maven Dependency

```
<dependency>
    <groupId>org.bouncycastle</groupId>
    <artifactId>bcprov-jdk15on</artifactId>
    <version>1.60</version>
</dependency>
```

5.2. Hashing Using the Bouncy Castle Library

The Bouncy Castle API provides a utility class for converting hex data to bytes and back again.

However, we need to populate a digest using the built-in Java API first:

```
MessageDigest digest = MessageDigest.getInstance("SHA-256");
byte[] hash = digest.digest(
  originalString.getBytes(StandardCharsets.UTF_8));
String sha256hex = new String(Hex.encode(hash));
```

6. SHA3-256

Now let's continue with SHA3-256. SHA3-256 hashing in Java isn't that different from SHA-256.

6.1. *MessageDigest* Class in Java

Starting from JDK 9, we can simply use the built-in SHA3-256 algorithm:

```
final MessageDigest digest = MessageDigest.getInstance("SHA3-256");
final byte[] hashbytes = digest.digest(
  originalString.getBytes(StandardCharsets.UTF_8));
String sha3Hex = bytesToHex(hashbytes);
```

6.2. Apache Commons Codecs

Apache Commons Codecs provides a convenient *DigestUtils* wrapper for the *MessageDigest* class.

This library began to support SHA3-256 since version 1.11, and it requires JDK 9+ as well:

String sha3Hex = **new DigestUtils**("SHA3-256").digestAsHex(originalString);

6.3. Keccak-256

Keccak-256 is another popular SHA3-256 hashing algorithm. Currently, it serves as an alternative to the standard SHA3-256. Keccak-256 delivers the same security level as the standard SHA3-256, and it differs from SHA3-256 only on the padding rule. It has been used in several blockchain projects, such as Monero.

Again, we need to import the Bouncy Castle Library to use Keccak-256 hashing:

Security.addProvider(**new BouncyCastleProvider**());

final MessageDigest digest = MessageDigest.getInstance("Keccak-256");

final byte[] encodedhash = digest.digest(

originalString.getBytes(StandardCharsets.UTF_8));

String sha3Hex = bytesToHex(encodedhash);

We can also make use of the Bouncy Castle API to do the hashing:

Keccak.**Digest256** digest256 = **new Keccak**.Digest256();

byte[] hashbytes = digest256.digest(

originalString.getBytes(StandardCharsets.UTF_8));

String sha3Hex = **new String**(Hex.encode(hashbytes));

https://www.baeldung.com/sha-256-hashing-java

https://code-examples.net/en/q/8e2855

https://mkyong.com/java/java-sha-hashing-example/

Receiving and Sending Data

How to send simple data to other apps

Android uses intents and associated extras to allow users to quickly and easily share information using their favorite apps.

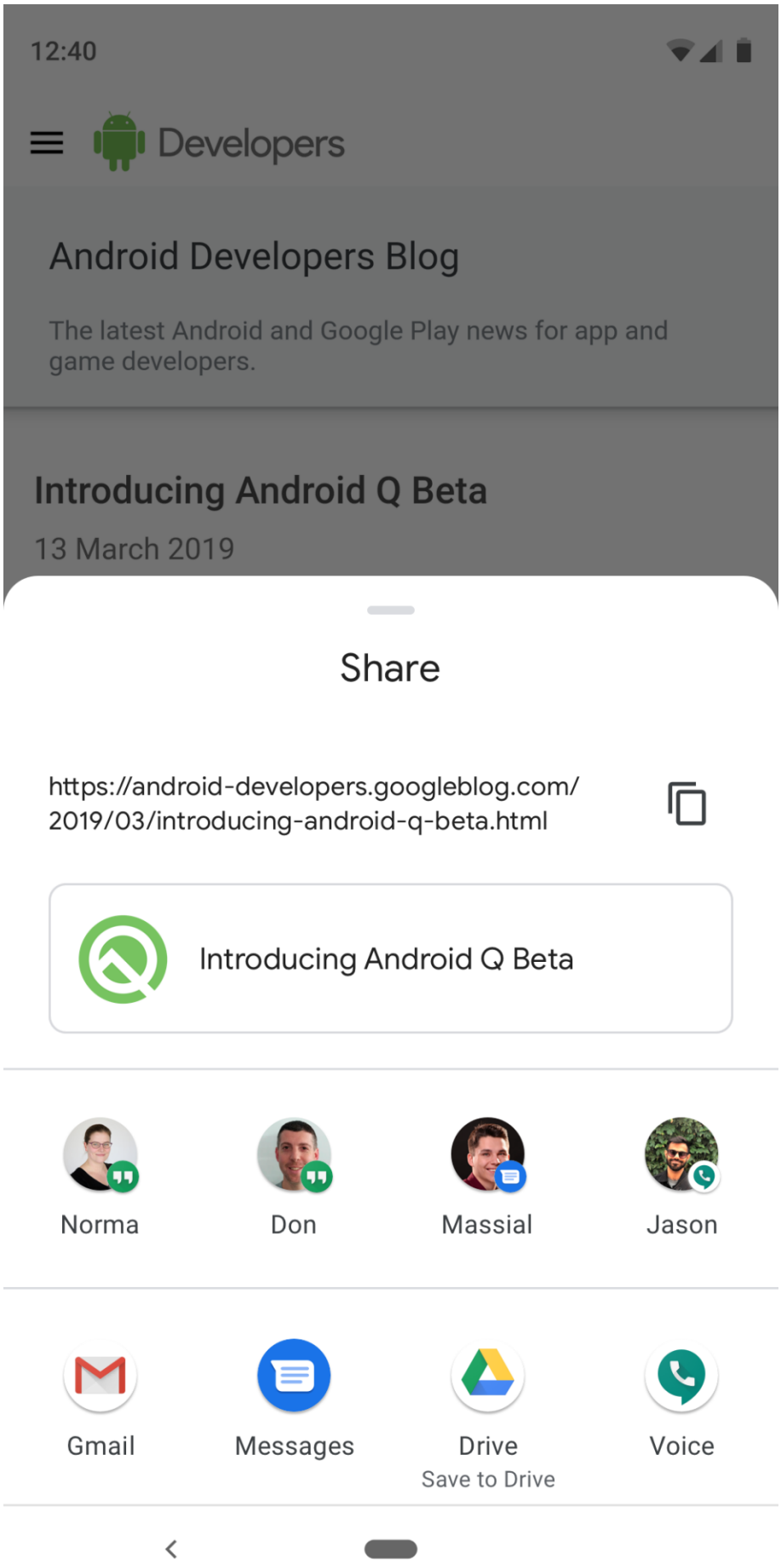
Android offers two ways for users to share data between apps:

- Android Sharesheet is primarily designed to send content outside of the app and/or directly to another user. For example, sharing a URL with a friend.
- Android's intent resolver is best suited for passing data to the next stage of a well-defined task. For example, opening a PDF from your app and letting users choose their favorite viewer.

When creating an intent, you must specify the action to be performed by it. Android uses action ACTION_SENDto send data from one activity to another, even across process boundaries. You must specify the data and related type. The system automatically identifies

compatible activities that can receive the data and displays them to the user. In the case of the intent resolver, if only one activity can handle the intent, that activity will start immediately.

Why Use Android Sharesheet



It is highly recommended to use Android Sharesheet to create consistency for your users across apps. Apps should not display their own list of share destinations or create their own Sharesheet variations.

Android Sharesheet gives users the ability to share information with the right person, including relevant app suggestions, all in a single tap. Sharesheet can suggest unavailable destinations for custom and consistently ranked solutions. This is because Sharesheet can take into account information about the app and user activity that is only available to the system.

Android Sharesheet also has many useful features for developers. For example, you can:

- find out when your users complete a share and where to;
- add your own ChooserTargetand custom app targets ;
- provide rich text content previews starting with Android 10 (API level 29);
- exclude destinations corresponding to ComponentNamespecific s .

Use Android Sharesheet

For all types of sharing, create an intent and set the action to Intent.ACTION_SEND. To display the Android Sharesheet, you need to call Intent.createChooser(), passing it your Intent. It will return a version of your intent that will always display the Android Sharesheet.

Send text content

The most direct and common use of Android Sharesheet is to send text content from one activity to another. For example, most browsers can share the URL of the currently displayed page as text with another app. This option is often used to share an article or website with friends via email or social networks. See how to do this below:

KotlinJava

```
val sendIntent: Intent = Intent().apply {
    action = Intent.ACTION_SEND
    putExtra(Intent.EXTRA_TEXT, "This is my text to send.")
    type = "text/plain"
}

val shareIntent = Intent.createChooser(sendIntent, null)
startActivity(shareIntent)
```

Optionally, you can add other elements to include more information, such as email recipients (EXTRA_EMAIL, EXTRA_CC, EXTRA_BCC), email subject (EXTRA_SUBJECT), and so on.

Note: Some email apps like Gmail expect one **String[]**for extra elements like **EXTRA_EMAIL**and **EXTRA_CC**, use **putExtra(String, String[])**to add them to your intent.

Send binary content

Share binary data using the ACTION_SEND. Set the appropriate MIME type and put a URI for the data in the extra element EXTRA_STREAM. In general, this option is used to share images, but it can be used to share any type of binary content:

KotlinJava

```
val shareIntent: Intent = Intent().apply {
    action = Intent.ACTION_SEND
    putExtra(Intent.EXTRA_STREAM, uriToImage)
    type = "image/jpeg"
}
startActivity(Intent.createChooser(shareIntent, resources.getText(R.string.send_to)))
```

The receiving application needs permission to access the data it Uripoints to. Recommended ways to do this are as follows:

- Store the data in the ContentProvider, ensuring that other apps have the correct permission to access your provider. The preferred mechanism for granting access is to use URI permissions , which are temporary and only grant access to the receiving application. An easy way to create one ContentProviderlike this is to use the helper class FileProvider.
- Use the system MediaStore. MediaStoreis primarily intended for video, audio, and image MIME types . However, as of Android 3.0 (API level 11) it can also store non-media types (see MediaStore.Filesfor more). You can insert files in MediaStoreusing scanFile(), followed by a content://style Urisuitable for sharing, which is passed to the onScanCompleted()given callback. Once added to MediaStorethe system, the content can be accessed by any app on the device.

Use the correct MIME type

Enter the most specific MIME type for the data you are sending. For example, use text/plainwhen sharing plain text. Here are some common MIME types for sending simple data on Android.

- text/plain, text/rtf, text/html, text/json: receivers must be registered totext/*
- image/jpg, image/png, image/gif: receivers must be registered toimage/*
- video/mp4, video/3gp: receivers must be registered tovideo/*
- application/pdf: Receivers must be registered for supported file extensions
- You can use the MIME type */*, but this is not recommended and will only match for activities capable of handling generic data streams.

Android Sharesheet can show a preview of content based on the given MIME type. Some preview features are only available for specific types.

See the official IANA registry of MIME media types.

Share multiple pieces of content

To share multiple pieces of content, use the action ACTION_SEND_MULTIPLEalong with a list of URIs that point to the content. The MIME type varies depending on the combination of content you are sharing. For example, if you share three JPEG images, the type will still be "image/jpg". For a mix of image types, the type will need to be "image/*"to match an activity that handles any image type. While it is possible to share a mix of types, this is not

recommended as it is not clear to the receiver what should be sent. If you need to send multiple types, use "*/*". It is up to the receiving application to analyze and process your data. See an example:

KotlinJava

```
val imageUris: ArrayList<Uri> = arrayListOf(
        // Add your image URIs here
        imageUri1,
        imageUri2
)

val shareIntent = Intent().apply {
    action = Intent.ACTION_SEND_MULTIPLE
    putParcelableArrayListExtra(Intent.EXTRA_STREAM, imageUris)
    type = "image/*"
}
startActivity(Intent.createChooser(shareIntent, "Share images to.."))
```

Make sure the URIsgiven point points to data that can be accessed by a receiving app.

Add rich content to text views

Starting with Android 10 (API level 29), the Android Sharesheet shows a preview of the text being shared. In some cases, shared text can be difficult to understand. Imagine a case of sharing a complicated URL like https://www.google.com/search?ei=2rRVXcLkJajM0PEPoLy7oA4. A more advanced view can reassure users about what is being shared.

If you are viewing text, you can set a title, a thumbnail image, or both. Add a description to Intent.EXTRA_TITLEbefore calling Intent.createChooser(). Add a relevant thumbnail via ClipData.

Note: The image content URI needs to be provided from a FileProvider, usually a **<cache-path>**configured one. See Share files . Sharesheet needs to have the correct permissions to read any image you want to use as a thumbnail. See Intent.FLAG_GRANT_READ_URI_PERMISSION .

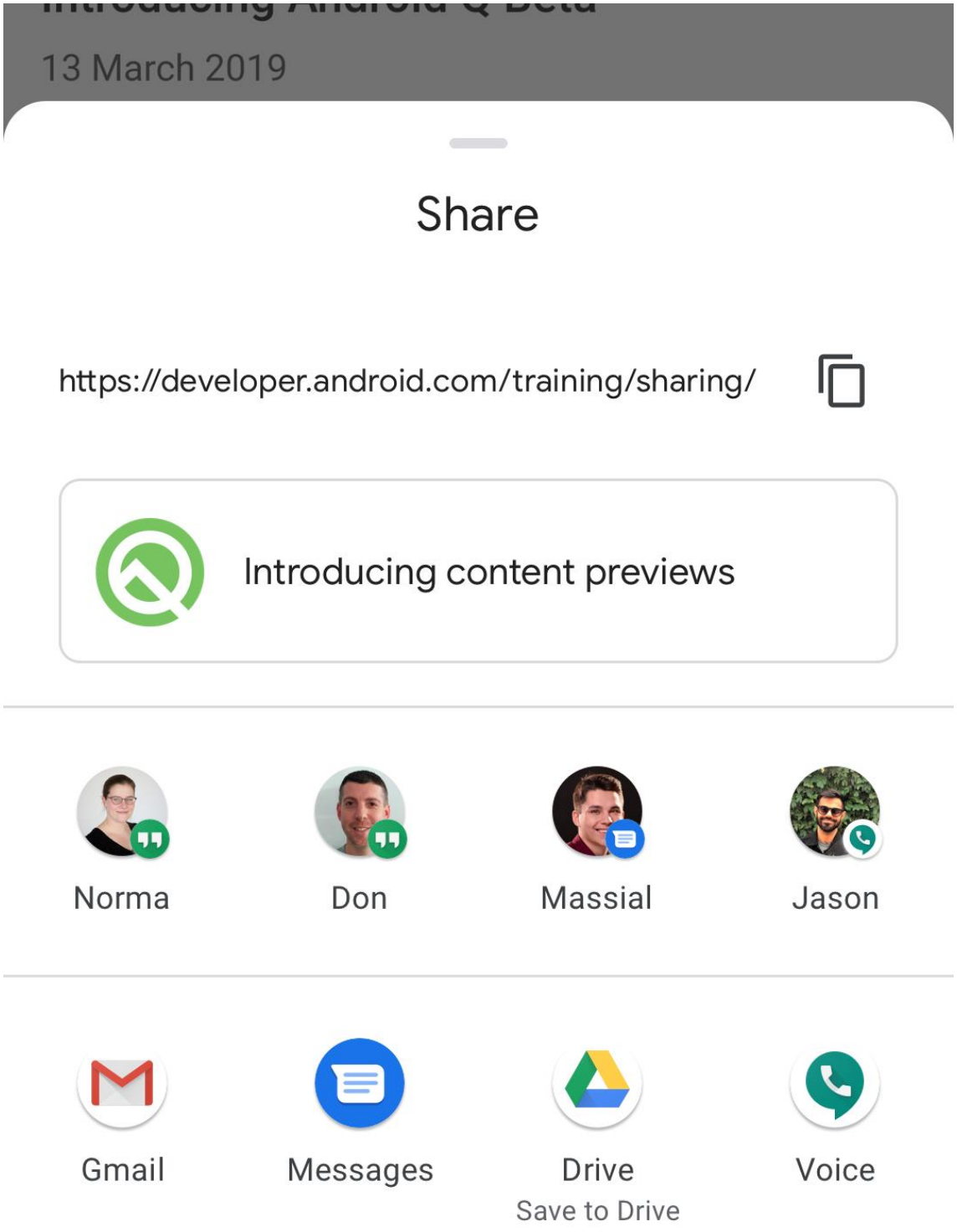
See an example:

KotlinJava

```
val share = Intent.createChooser(Intent().apply {
    action = Intent.ACTION_SEND
    putExtra(Intent.EXTRA_TEXT, "https://developer.android.com/training/sharing/")

    // (Optional) Here we're setting the title of the content
    putExtra(Intent.EXTRA_TITLE, "Introducing content previews")

    // (Optional) Here we're passing a content URI to an image to be displayed
    data = contentUri
```

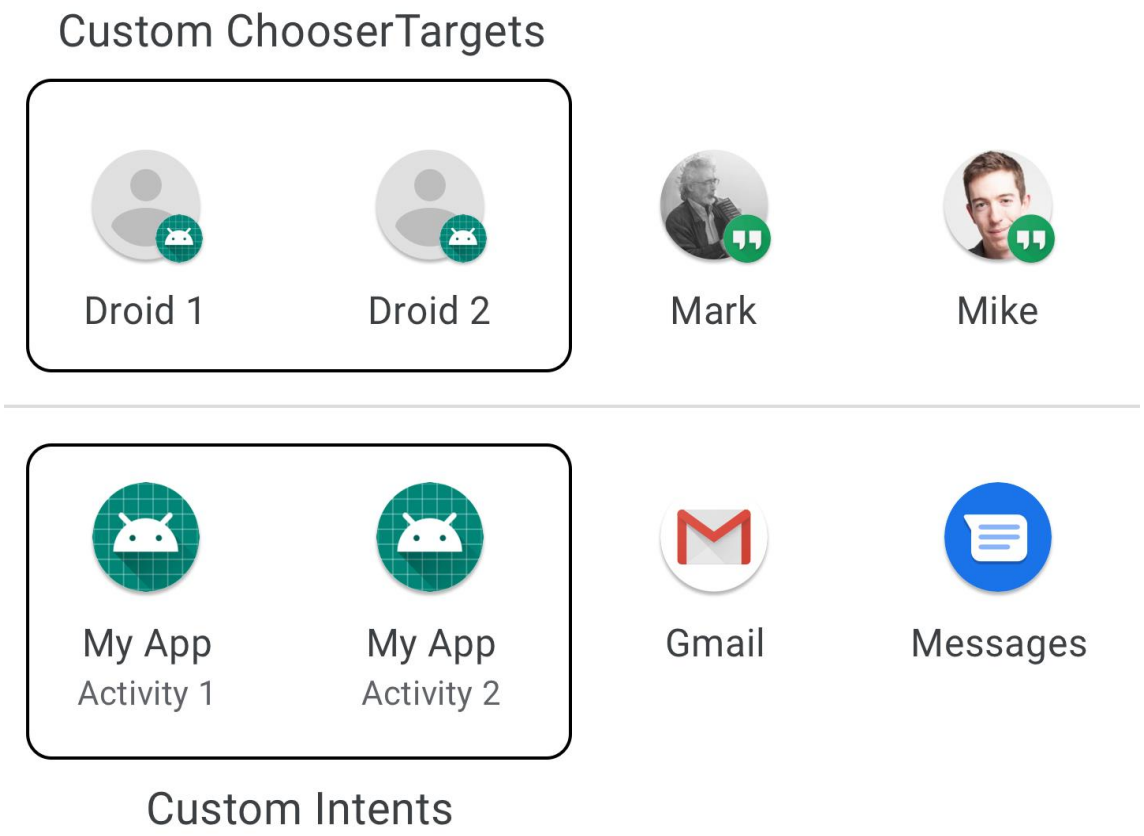
```
    flags = Intent.FLAG_GRANT_READ_URI_PERMISSION
}, null)
startActivity(share)
```

The view looks like this:



Add custom destinations

Android Sharesheet allows you to specify a limited number of objects ChooserTargetthat are shown before share shortcuts and ChooserTargets loaded from ChooserTargetServices. You can also specify a limited number of intents that point to activities listed before the app's suggestions.



Add Intent.EXTRA_CHOOSER_TARGETSand Intent.EXTRA_INITIAL_INTENTSto the share intent **after** calling Intent.createChooser().

KotlinJava

```
val share = Intent.createChooser(myShareIntent, null).apply {
    putExtra(Intent.EXTRA_CHOOSER_TARGETS, myChooserTargetArray)
    putExtra(Intent.EXTRA_INITIAL_INTENTS, myInitialIntentArray)
}
```

Use this feature carefully. Each Intente- ChooserTargetcustom you add reduces the number suggested by the system. Adding custom destinations is not recommended. A common example of adding Intent.EXTRA_INITIAL_INTENTSis making available other actions that users can perform on shared content. For example, the user shares images, and Intent.EXTRA_INITIAL_INTENTSit is used to provide the ability to send a link. A common example of adding Intent.EXTRA_CHOOSER_TARGETSis is showing relevant people or devices that your app is offered to.

Exclude specific destinations by component

You can exclude specific destinations by providing Intent.EXTRA_EXCLUDE_COMPONENTS.. This option only has to be used to remove

targets you control. A common use case is to hide the app's share destinations when your users share from inside the app, as the intent tends to share outside the app.

Add Intent.EXTRA_EXCLUDE_COMPONENTSto intent after calling Intent.createChooser().

KotlinJava

```
val share = Intent.createChooser(myShareIntent, null).apply {
  // Only use components you have control over
  share.putExtra(Intent.EXTRA_EXCLUDE_COMPONENTS, myComponentArray)
}
```

Receive sharing information

It can be helpful to know when your users are sharing and what destination is selected. Android Sharesheet makes this possible by providing the targeting ComponentNameof destinations that users click through a IntentSender.

First create one PendingIntentto one BroadcastReceiverand provide IntentSenderhis in Intent.createChooser().

KotlinJava

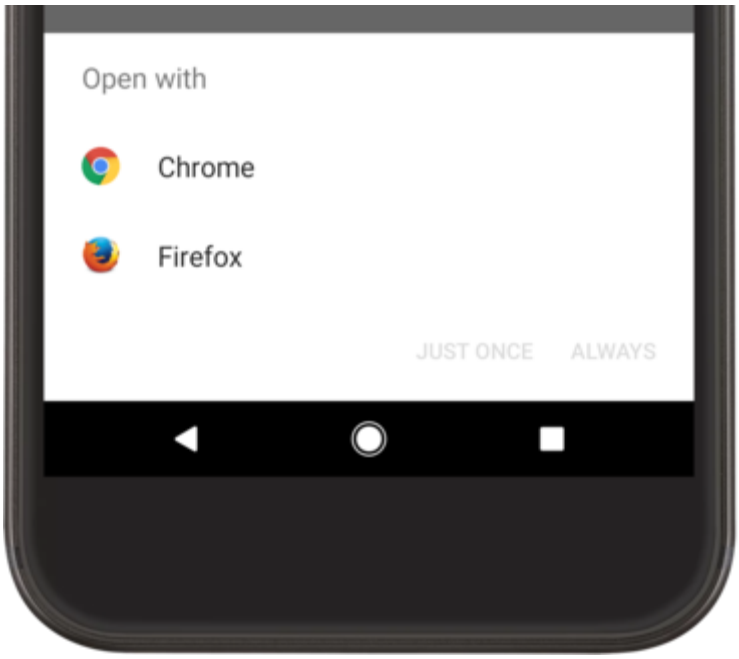
```
var share = new Intent(Intent.ACTION_SEND);
...
val pi = PendingIntent.getBroadcast(myContext, requestCode, Intent(myContext,
MyBroadcastReceiver.class),
Intent.FLAG_UPDATE_CURRENT)
share = Intent.createChooser(share, null, pi.intentSender);
```

Receive the callback on MyBroadcastReceiver and look for Intent.EXTRA_CHOSEN_COMPONENT.

KotlinJava

```
override fun onReceive(context: Context, intent: Intent) {
  ...
  val clickedComponent : ComponentName =
intent.getParcelableExtra(EXTRA_CHOSEN_COMPONENT);
}
```

Use the Android Intent Resolver



Screenshot of Intent Resolver ACTION_SEND.

Android's intent resolver is best suited for sending data to another app as part of a well-defined task flow.

To use the Android Intent Resolver, create an intent and add extra elements as you would if you called the Android Sharesheet. However, **do not callIntent.createChooser()** .

If there are multiple apps installed with filters that match ACTION_SENDand the MIME type, the system will display a disambiguation dialog called intent resolver, which allows the user to choose a share destination. If a single application matches, it will run.

Here's an example of how to use the Android Intent Resolver to send text:

KotlinJava

```
val sendIntent: Intent = Intent().apply {
    action = Intent.ACTION_SEND
    putExtra(Intent.EXTRA_TEXT, "This is my text to send.")
    type = "text/plain"
}
startActivity(sendIntent)
```

https://developer.android.com/training/sharing/send

How to receive simple data from other apps

Just as an app sends data to other apps, it can also receive data. Think about how users interact with your app and the types of data you want to receive from other apps. For example, for a social networking app, it might be interesting to receive text content from another app, such as a web URL.

Users often send data to the app through the Android Sharesheet or intent resolver. All incoming data has a MIME type defined by the provider app. Your app can receive data sent by another app in three ways:

- One Activitywith a corresponding tag intent-filterin the manifest
- One or more objects ChooserTargetreturned by theChooserTargetService
- Sharing shortcuts published by your app. They replace objects ChooserTarget. Share shortcuts are only available in apps for Android 10 (API level 29)

Sharing shortcuts and objects ChooserTargetare direct sharing direct links to a Activityspecific one in your app. They usually represent a person and are displayed by the Android Sharesheet. For example, a text messaging app might offer a sharing shortcut to someone who has a direct link to a conversation with that person.

Supporting MIME Types

Your app should be able to receive as many MIME types as possible. For example, a messaging app used to send text, images, and videos must support
receiving text/*, image/*and video/*. Here are some common MIME types for sending simple data on Android.

- text/*: in general, the sender user will send text/plain, text/rtf, text/html, text/json.
- image/*: in general, the sender user will send image/jpg, image/png, image/gif.
- video/*: in general, the sender user will send video/mp4, video/3gp.
- Recipients need to register for supported file extensions: in general, the sending user will send application/pdf.

Consult the official IANA registry of media MIME types. You can receive a MIME type of */*, but this is not recommended unless you are fully capable of processing **any** type of incoming content.

How to create great sharing targets

When a user taps a share target associated with a specific activity, they must be able to confirm and edit the shared content before using it. This is especially important for text message data.

Tapping any direct share target should take the user to an interface where an action can be taken directly on the target's subject. Avoid showing the user a disambiguation or taking them to an interface unrelated to the target being touched. In particular, don't take the user to a contact disambiguation interface where they need to confirm or reselect the contact for sharing, as this has already been done by tapping the target in the Android Sharesheet. For example, in a text messaging app, tapping a direct share target should take the user to a conversation view with the selected person. The keyboard must be visible, and the message must be pre-populated with the shared data.

How to receive data with an activity

Update your manifest

Intent filters tell the system which intents an app component tends to accept. In the same way that you created an intent with the action in the Sending simple data to other appsACTION_SEND lesson , create intent filters so that you can receive intent with this action. The intent filter is defined in the manifest using the . For example, if your app handles incoming text message content and a single image, or multiple images of any type, your manifest should look something like this:<intent-filter>

```
<activity android:name=".ui.MyActivity" >
  <intent-filter>
    <action android:name="android.intent.action.SEND" />
    <category android:name="android.intent.category.DEFAULT" />
    <data android:mimeType="image/*" />
  </intent-filter>
  <intent-filter>
    <action android:name="android.intent.action.SEND" />
    <category android:name="android.intent.category.DEFAULT" />
    <data android:mimeType="text/plain" />
  </intent-filter>
  <intent-filter>
    <action android:name="android.intent.action.SEND_MULTIPLE" />
    <category android:name="android.intent.category.DEFAULT" />
    <data android:mimeType="image/*" />
  </intent-filter>
</activity>
```

When another app tries to share any of these items by creating an intent and passing it to startActivity(), your app will be listed as an option in the Android Sharesheet or intent resolver. If the user selects your app, the corresponding activity (.ui.MyActivityin the example above) will be started. It's up to you to render the content correctly in your code and UI.

Note: For more information about intent filters and intent resolution, read Intents and intent filters

Process incoming content

To process the content sent by a Intent, call getIntent()to get the object Intent. Once you have the object, you can analyze its contents and determine what to do next. If it is possible to initiate this activity from other parts of the system, such as the splash screen, this characteristic will need to be considered when analyzing the intent.

Check incoming data very carefully, because you never know what another app might send. For example, the wrong MIME type might be set or the uploaded image might be too large. Also, remember to process binary data on a separate thread, not the main thread (UI).

KotlinJava

```
override fun onCreate(savedInstanceState: Bundle?) {
  ...
  when {
    intent?.action == Intent.ACTION_SEND -> {
```

```
            if ("text/plain" == intent.type) {
                handleSendText(intent) // Handle text being sent
            } else if (intent.type?.startsWith("image/") == true) {
                handleSendImage(intent) // Handle single image being sent
            }
        }
        intent?.action == Intent.ACTION_SEND_MULTIPLE
                && intent.type?.startsWith("image/") == true -> {
                handleSendMultipleImages(intent) // Handle multiple images being sent
        }
        else -> {
            // Handle other intents, such as being started from the home screen
        }
    }
    ...
}

private fun handleSendText(intent: Intent) {
    intent.getStringExtra(Intent.EXTRA_TEXT)?.let {
        // Update UI to reflect text being shared
    }
}

private fun handleSendImage(intent: Intent) {
    (intent.getParcelableExtra<Parcelable>(Intent.EXTRA_STREAM) as? Uri)?.let {
        // Update UI to reflect image being shared
    }
}

private fun handleSendMultipleImages(intent: Intent) {
    intent.getParcelableArrayListExtra<Parcelable>(Intent.EXTRA_STREAM)?.let {
        // Update UI to reflect multiple images being shared
    }
}
```

Updating the UI after receiving the data can be as simple as filling in one EditTextor more complex, such as applying an interesting photo filter to an image. It's up to your app to determine what happens next.

Ensure users recognize your app

Your app is represented by the corresponding icon and label in the Android Sharesheet and Intent Resolver. Both are defined in the manifest. You can define activity or intent filter labels to provide more context.

Starting with Android 10 (API level 29), Android Sharesheet will only use the icons defined in the application. Icons configured in intent-filterand tags activitywill be ignored.

Note: The best share targets do not need a label and icon in the related activity or intent filter. Just the name and icon of the target app should be enough for users to understand what will happen at the time of sharing.

Making direct share targets available

Direct Sharing was introduced in Android 6.0 (API level 23), allowing apps to serve objects ChooserTargetthrough a ChooserTargetService. Results were retrieved reactively on demand, leading to slow loading of targets.

In Android 10 (API level 29), the Direct Share APIs ChooserTargetServicehave been replaced by the new Sharing Shortcuts API . Rather than reactively retrieving results on demand, the Sharing Shortcuts API allows apps to publish direct sharing targets in advance. The direct sharing mechanism ChooserTargetServicewill continue to work, but targets provided in this way will be ranked lower in priority than any other targets that use the Sharing Shortcuts API.

ShortcutManagerCompat is an AndroidX API that provides backwards compatibility sharing shortcuts to the DirectShare API of ChooserTargetService. This is the preferred way to publish share shortcuts and ChooserTargets. See instructions below.

Publish direct share targets

You can only use dynamic shortcuts to publish direct share targets. Follow the steps below to publish direct share targets using the new API.

1. Declare share target elements in the app's XML resource file. For details, see Declare a Share Target below.
2. Publish dynamic shortcuts with categories corresponding to declared share targets. Shortcuts can be added, accessed, updated and removed using the ShortcutManager or ShortcutManagerCompat in AndroidX. The preferred method is to use the compatibility library in AndroidX, because it provides compatibility with previous versions of Android.

API Sharing Shortcuts

ShortcutInfo.Builderincludes new and improved methods that provide more information about the share target:

setCategories()

This is not a new method, but categories are now also used to filter shortcuts that can process share intents or actions. See Declaring a Share Target for more details. This field is required for shortcuts that are intended to be used as sharing targets.

setLongLived()

Specifies whether or not a shortcut is valid when the app has unpublished it or made it invisible (such as a dynamic or pinned shortcut). If a shortcut is long-lived, it may be cached by various system services, even after being unpublished as a dynamic shortcut.

Making a shortcut long lasting can improve its rating. See How to get the best rating for more details.

setPerson(), setPersons()

Associates one or more objects Personwith the shortcut. It can be used to better understand user behavior across different apps and to help the framework's prediction services improve the suggestions offered in a ShareSheet. Adding Person information to a shortcut is optional, but highly recommended if the share target is associated with a person. Some sharing targets, such as clouds, cannot be associated with a person.

Including a Personspecific object with a unique key in a share target and related notifications can improve your ranking. See How to get the best rating for more details.

For a typical messaging app, you need to publish a separate shortcut for each contact, and the field Personneeds to contain the contact's information. If the target is associated with multiple people (such as a group chat), add multiple objects Personto a single sharing target.

When posting a shortcut for a single person, include their full name in setLongLabel()and any abbreviations of the name, such as nickname or first name, insetShortLabel()

For an example of how to publish sharing shortcuts, see the Share Shortcuts Code Samples .

How to get the best rating

Android Sharesheet shows a fixed number of direct share targets. These suggestions are sorted by rating. You can improve the ranking of your shortcuts by doing the following:

- Make sure all shortcuts are unique and are never reused by different targets.
- Make sure the shortcut is long-lived by calling setLongLived(true).
- Provide a ranking that can be used to compare your app's shortcuts in the absence of training data. See setRank(). A lower rating means the shortcut is more important.

To further improve the ranking, we highly recommend that social apps do all of the above and:

- Provide relevant objects Personwith a key set in your shortcut (see setPerson(), setPersons()and setKey()).
- Link your shortcut to relevant notifications from the same person or group of people (see setShortcutId().). The shortcutId can be assigned to any previously published shortcut with setLongLived(true).

To get the best possible rating, social apps can do all of the above and:

- in defined objects Person, provide a valid URI for an associated contact on the device. SeesetUri()

See an example shortcut with all possible ranking improvements.

KotlinJava

```
val person = Person.Builder()
 ...
 .setName(fullName)
 .setKey(staticPersonIdentifier)
 .setUri("tel:$phoneNumber") // alternatively "mailto:$email" or CONTENT_LOOKUP_URI
 .build()

val shortcutInfo = ShortcutInfoCompat.Builder(myContext, staticPersonIdentifier)
```

```
  ...
  .setShortLabel(firstName)
  .setLongLabel(fullName)
  .setPerson(person)
  .setLongLived(true)
  .setRank(personRank)
  .build()
```

KotlinJava

```
val notif = NotificationCompat.Builder(myContext, channelId)
  ...
  .setShortcutId(staticPersonIdentifier)
  .build()
```

How to offer shortcut images

To create a sharing shortcut, you need to add an image via setIcon().

Sharing shortcuts can appear on all surfaces of the system and can be remodeled. To ensure your shortcut looks the way you intended, provide an adaptive bitmap via IconCompat.createWithAdaptiveBitmap().

Adaptive bitmaps must follow the same guidelines and dimensions established for adaptive icons . The most common way to accomplish this is to scale the intended square bitmap to 72 x 72 dp and center it on a 108 x 108 dp transparent canvas.

Do not offer images masked to a specific shape. For example, before Android 10 (API level 29), it was common to offer user avatars for ChooserTargetdirect share emails masked in the shape of a circle. Android Sharesheet and other system surfaces in Android 10 now shape and theme shortcut images. The preferred method for providing sharing shortcuts, via the ShortcutManagerCompat , will automatically wrap ChooserTargetdirect sharing s from previous versions into a circle.

Declare a share target

Share targets need to be declared in the app's resource file, just like static shortcut definitions . Add the share target definitions inside the root element <shortcuts>in the resource file, along with other static shortcut definitions. Each element <share-targets>contains information about the shared data type, the corresponding categories, and the target class that will process the share intent. The XML code looks like this:

```
<shortcuts xmlns:android="http://schemas.android.com/apk/res/android">
  <share-target
android:targetClass="com.example.android.sharingshortcuts.SendMessageActivity">
    <data android:mimeType="text/plain" />
    <category
android:name="com.example.android.sharingshortcuts.category.TEXT_SHARE_TARGET" />
  </share-target>
```

</shortcuts>

The data element in a share target is similar to specifying data in an intent filter . Each share target can have multiple categories, which are only used to match an app's published shortcuts with their share target definitions. Categories can have arbitrary values defined by the app.

If the user selects the share shortcut in Android ShareSheet that matches the share target example above, the app will receive the following share intent.

Action: Intent.ACTION_SEND
ComponentName: {com.example.android.sharingshortcuts /
com.example.android.sharingshortcuts.SendMessageActivity}
Data: Uri to the shared content
EXTRA_SHORTCUT_ID: <ID of the selected shortcut>

If the user opens the share target from the home screen shortcuts, the app will receive the intent created when adding the share shortcut to ShortcutManagerCompat . Because it's a different intent, Intent.EXTRA_SHORTCUT_IDit won't be available, and you'll have to manually pass the ID if you need it.

How to Use AndroidX to Offer Share Shortcuts and ChooserTargets

To work with the AndroidX compatibility library, your app manifest needs to contain the target selector service metadata set and intent filters. See current Direct Share API ChooserTargetService.

Since this service is already declared in the compatibility library, the user does not need to declare it in the app's manifest. However, the sharing activity link to the service needs to be considered as a target selector provider.

In the following example, the implementation of ChooserTargetServiceis androidx.core.content.pm.ChooserTargetServiceCompat, which is already defined in AndroidX:

```
<activity
  android:name=".SendMessageActivity"
  android:label="@string/app_name"
  android:theme="@style/SharingShortcutsDialogTheme">
  <!-- This activity can respond to Intents of type SEND -->
  <intent-filter>
    <action android:name="android.intent.action.SEND" />
    <category android:name="android.intent.category.DEFAULT" />
    <data android:mimeType="text/plain" />
  </intent-filter>
  <!-- Only needed if you import the sharetarget AndroidX library that
     provides backwards compatibility with the old DirectShare API.
     The activity that receives the Sharing Shortcut intent needs to be
     taken into account with this chooser target provider. -->
  <meta-data
    android:name="android.service.chooser.chooser_target_service"
```

```
        android:value="androidx.sharetarget.ChooserTargetServiceCompat" />
</activity>
```

Sharing Shortcuts FAQ

What are the main differences between the current Sharing Shortcuts API and the ChooserTargetServiceold Direct Share API?

The Sharing Shortcuts API uses a push model, while the old Direct Share API uses the pull model. This makes the process of retrieving share targets much faster during ShareSheet preparation. From an app developer's point of view, using the new API, the app needs to list direct share targets in advance. It may also be necessary to update the jump list whenever the internal state of the app changes (for example, if a new contact is added to a messaging app).

What happens if I don't migrate to the new Sharing Shortcuts APIs?

On Android 10 (API level 29) and higher, Android Sharesheet will prioritize sharing targets provided through the ShortcutManager using the Sharing Shortcuts API. So your published share targets can be hidden by other apps' share targets and never appear when shared.

Can I use ChooserTargetServicethe Sharing Shortcuts and Direct Share APIs in my app to be backwards compatible?

Do not do it. Instead, use the Support Library
APIs ShortcutManagerCompatprovided. Combining two sets of APIs can cause unwanted/unexpected behavior when retrieving share targets.

What's the difference between published shortcuts for share targets and home screen shortcuts (normal use of shortcuts by tapping and holding app icons on the home screen)?

All shortcuts published with the "share target" function are also home screen shortcuts and will be displayed in the menu when the app icon is tapped and held. The maximum shortcuts limit per activity also applies to the total number of shortcuts an app publishes (sharing targets and legacy Home shortcuts combined).

https://developer.android.com/training/sharing/receive

Reversing Apks

Prerequisites

In order to follow this introduction to APK reversing there're a few prerequisites:

- A working brain (I don't give this for granted anymore …).
- An Android smartphone (doh!).
- You have a basic knowledge of the Java programming language (you understand it if you read it).
- You have the JRE installed on your computer.
- You have adb installed.
- You have the Developer Options and USB Debugging enabled on your smartphone.

What is an APK?

An Android application is packaged as an **APK** (*Android Package*) file, which is essentially a ZIP file containing the compiled code, the resources, signature, manifest and every other file the software needs in order to run. Being it a ZIP file, we can start looking at its contents using the unzip command line utility (or any other unarchiver you use):

unzip application.apk -d application

Here's what you will find inside an APK.

/AndroidManifest.xml (file)

This is the binary representation of the XML manifest file describing what permissions the application will request (keep in mind that some of the permissions might be requested at runtime by the app and not declared here), what activities (GUIs) are in there, what services (stuff running in the background with no UI) and what receivers (classes that can receive and handle system events such as the device boot or an incoming SMS).

Once decompiled (more on this later), it'll look like this:

```
    <?xml version="1.0" encoding="utf-8" standalone="no"?>
1   <manifest xmlns:android="http://schemas.android.com/apk/res/android"
2   package="com.company.appname"
3                                           platformBuildVersionCode="24"
4                                           platformBuildVersionName="7.0">
5     <uses-permission
6   android:name="android.permission.WRITE_EXTERNAL_STORAGE"/>
7     <uses-permission android:name="android.permission.INTERNET"/>
8
9     <application android:allowBackup="true" android:icon="@mipmap/ic_launcher"
10      android:label="@string/app_name"
11      android:supportsRtl="true" android:theme="@style/AppTheme">
12      <activity android:name="com.company.appname.MainActivity">
13        <intent-filter>
14          <action android:name="android.intent.action.MAIN"/>
15          <category android:name="android.intent.category.LAUNCHER"/>
16        </intent-filter>
17      </activity>
18    </application>
19
    </manifest>
```

Keep in mind that this is the perfect starting point to isolate the application "entry points", namely the classes you'll reverse first in order to understand the logic of the whole software. In this case for instance, we would start inspecting the com.company.appname.MainActivity class being it declared as the main UI for the application.

/assets/* (folder)

This folder will contain application specific files, like wav files the app might need to play, custom fonts and so on. Reversing-wise it's usually not very important, unless of course you find inside the software functional references to such files.

/res/* (folder)

All the resources, like the activities xml files, images and custom styles are stored here.

/resources.arsc (file)

This is the "index" of all the resources, long story short, at each resource file is assigned a numeric identifier that the app will use in order to identify that specific entry and the resources.arsc file maps these files to their identifiers … nothing very interesting about it.

/classes.dex (file)

This file contains the Dalvik (the virtual machine running Android applications) bytecode of the app, let me explain it better. An Android application is (most of the times) developed using the Java programming language. The java source files are then compiled into this bytecode which the Dalvik VM eventually will execute … pretty much what happens to normal Java programs when they're compiled to .class files.

Long story short, this file contains the logic, that's what we're interested into.

Sometimes you'll also find a classes2.dex file, this is due to the DEX format which has a limit to the number of classes you can declare inside a single dex file, at some point in history Android apps became bigger and bigger and so Google had to adapt this format, supporting a secondary .dex file where other classes can be declared.

From our perspective it doesn't matter, the tools we're going to use are able to detect it and append it to the decompilation pipeline.

/libs/ (folder)

Sometimes an app needs to execute native code, it can be an image processing library, a game engine or whatever. In such case, those .so ELF libraries will be found inside the libs folder, divided into architecture specific subfolders (so the app will run on ARM, ARM64, x86, etc).

/META-INF/ (folder)

Every Android application needs to be signed with a developer certificate in order to run on a device, even debug builds are signed by a debug certificate, the META-INF folder contains information about the files inside the APK and about the developer.

Inside this folder, you'll usually find:

- A MANIFEST.MF file with the SHA-1 or SHA-256 hashes of **all** the files inside the APK.
- A CERT.SF file, pretty much like the MANIFEST.MF, but signed with the RSA key.
- A CERT.RSA file which contains the developer public key used to sign the CERT.SF file and digests.

Those files are very important in order to guarantee the APK integrity and the ownership of the code. Sometimes inspecting such signature can be very handy to determine who really

developed a given APK. If you want to get information about the developer, you can use the openssl command line utility:

openssl pkcs7 -in /path/to/extracted/apk/META-INF/CERT.RSA -inform DER -print

This will print an output like:

```
PKCS7:
 type: pkcs7-signedData (1.2.840.113549.1.7.2)
 d.sign:
  version: 1
  md_algs:
      algorithm: sha1 (1.3.14.3.2.26)
      parameter: NULL
  contents:
   type: pkcs7-data (1.2.840.113549.1.7.1)
   d.data: <ABSENT>
  cert:
      cert_info:
       version: 2
       serialNumber: 10394279457707717180
       signature:
         algorithm: sha1WithRSAEncryption (1.2.840.113549.1.1.5)
         parameter: NULL
       issuer: C=TW, ST=Taiwan, L=Taipei, O=ASUS, OU=PMD, CN=ASUS AMAX
Key/emailAddress=admin@asus.com
       validity:
         notBefore: Jul  8 11:39:39 2013 GMT
         notAfter: Nov 23 11:39:39 2040 GMT
       subject: C=TW, ST=Taiwan, L=Taipei, O=ASUS, OU=PMD, CN=ASUS AMAX
Key/emailAddress=admin@asus.com
       key:
         algor:
           algorithm: rsaEncryption (1.2.840.113549.1.1.1)
           parameter: NULL
```

public_key: (0 unused bits)

...

...

...

This can be gold for us, for instance we could use this information to determine if an app was really signed by (let's say) Google or if it was resigned, therefore modified, by a third party.

How do I get the APK of an app?

Now that we have a basic idea of what we're supposed to find inside an APK, we need a way to actually get the APK file of the application we're interested into. There are two ways, either you install it on your device and use adb to get it, or you use an online service to download it.

Pulling an app with ADB

First of all let's plug our smartphone to the USB port of our computer and get a list of the installed packages and their namespaces:

adb shell pm list packages

This will list all packages on your smartphone, once you've found the namespace of the package you want to reverse (com.android.systemui in this example), let's see what its physical path is:

adb shell pm path com.android.systemui

Finally, we have the APK path:

package:/system/priv-app/SystemUIGoogle/SystemUIGoogle.apk

Let's pull it from the device:

adb pull /system/priv-app/SystemUIGoogle/SystemUIGoogle.apk

And here you go, you have the APK you want to reverse!

Using an Online Service

Multiple online services are available if you don't want to install the app on your device (for instance, if you're reversing a malware, you want to start having the file first, then installing on a clean device only afterwards), here's a list of the ones I use:

- Apk-DL
- Evozi Downloader
- Apk Leecher

Keep in mind that once you download the APK from these services, it's a good idea to check the developer certificate as previously shown in order to be 100% sure you downloaded the correct APK and not some repackaged and resigned stuff full of ads and possibly malware.

Network Analysis

Now we start with some tests in order to understand what the app is doing while executed. My first test usually consists in inspecting the network traffic being generated by the application itself and, in order to do that, my tool of choice is bettercap … well, that's why I developed it in the first place :P

Make sure you have bettercap installed and that both your computer and the Android device are on the same wifi network, then you can start MITM-ing the smartphone (192.168.1.5 in this example) and see its traffic in realtime from the terminal:

sudo bettercap -T 192.168.1.5 -X

The -X option will enable the sniffer, as soon as you start the app you should see a bunch of HTTP and/or HTTPS servers being contacted, now you know who the app is sending the data to, let's now see **what** data it is sending:

sudo bettercap -T 192.168.1.5 --proxy --proxy-https --no-sslstrip

This will switch from passive sniffing mode, to proxying mode. All the HTTP and HTTPS traffic will be intercepted (and, if neeeded, modified) by bettercap.
If the app is correctly using public key pinning (**as every application should**) you will **not** be able to see its HTTPS traffic but, unfortunately, in my experience this only happens for a very small number of apps.

From now on, keep triggering actions on the app while inspecting the traffic (you can also use Wireshark in parallel to get a PCAP capture file to inspect it later) and after a while you should have a more or less complete idea of what protocol it's using and for what purpose.

Static Analysis

After the network analysis, we collected a bunch of URLs and packets, we can use this information as our starting point, that's what we will be looking for while performing static analysis on the app. "Static analysis" means that you will **not** execute the app now, but you'll rather just study its code. Most of the times this is all you'll ever need to reverse something.

There're different tools you can use for this purpose, let's take a look at the most popular ones.

apktool

APKTool is the very first tool you want to use, it is capable of decompiling the AndroidManifest file to its original XML format, the resources.arsc file and it will also convert the classes.dex (and classes2.dex if present) file to an intermediary language

called SMALI, an ASM-like language used to represent the Dalvik VM opcodes as a human readable language.

It looks like:

```
1  .super Ljava/lang/Object;
2  .method public static main([Ljava/lang/String;)V
3     .registers 2
4     sget-object v0, Ljava/lang/System;->out:Ljava/io/PrintStream;
5     const-string   v1, "Hello World!"
6     invoke-virtual {v0, v1}, Ljava/io/PrintStream;->println(Ljava/lang/String;)V
7     return-void
8  .end method
```

But don't worry, in most of the cases this is not the final language you're gonna read to reverse the app ;)

Given an APK, this command line will decompile it:

apktool d application.apk

Once finished, the application folder is created and you'll find all the output of apktool in there.

You can also use apktool to decompile an APK, modify it and then recompile it (like i did with the Nike+ app in order to have more debug logs for instance), but unless the other tools will fail the decompilation, it's unlikely that you'll need to read smali code in order to reverse the application, let's get to the other tools now ;)

jADX

The jADX suite allows you to simply load an APK and look at its Java source code. What's happening under the hood is that jADX is decompiling the APK to smali and then converting the smali back to Java. Needless to say, reading Java code is much easier than reading smali as I already mentioned :)

Once the APK is loaded, you'll see a UI like this:

One of the best features of jADX is the string/symbol search (the button) that will allow you to search for URLs, strings, methods and whatever you want to find inside the codebase of the app.

Also, there's the Find Usage menu option, just highlight some symbol and right click on it, this feature will give you a list of every references to that symbol.

Dex2Jar and JD-Gui

Similar to jADX are the dex2jar and JD-GUI tools, once installed, you'll use dex2jar to convert an APK to a JAR file:

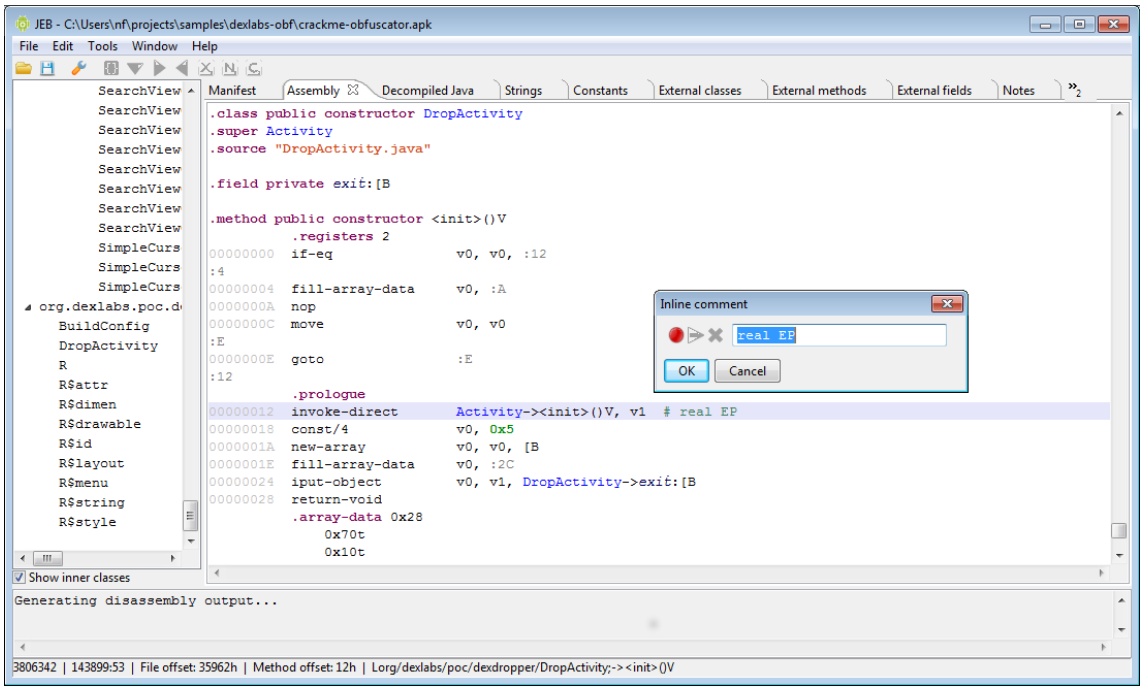
/path/to/dex2jar/d2j-dex2jar.sh application.apk

Once you have the JAR file, simply open it with JD-GUI and you'll see its Java code, pretty much like jADX:

Unfortunately JD-GUI is not as features rich as jADX, but sometimes when one tool fails you have to try another one and hope to be more lucky.

JEB

As your last resort, you can try the JEB decompiler. It's a **very** good software, but unfortunately it's not free, there's a trial version if you want to give it a shot, here's how it looks like:



JEB also features an ARM disassembler (useful when there're native libraries in the APK) and a debugger (**very** useful for dynamic analysis), but again, it's not free and it's not cheap.

Static Analysis of Native Binaries

As previously mentioned, sometimes you'll find native libraries (.so shared objects) inside the lib folder of the APK and, while reading the Java code, you'll find native methods declarations like the following:

```
1    public native String stringFromJNI();
```

The native keyword means that the method implementation is not inside the dex file but, instead, it's declared and executed from native code trough what is called a Java Native Interface or JNI.

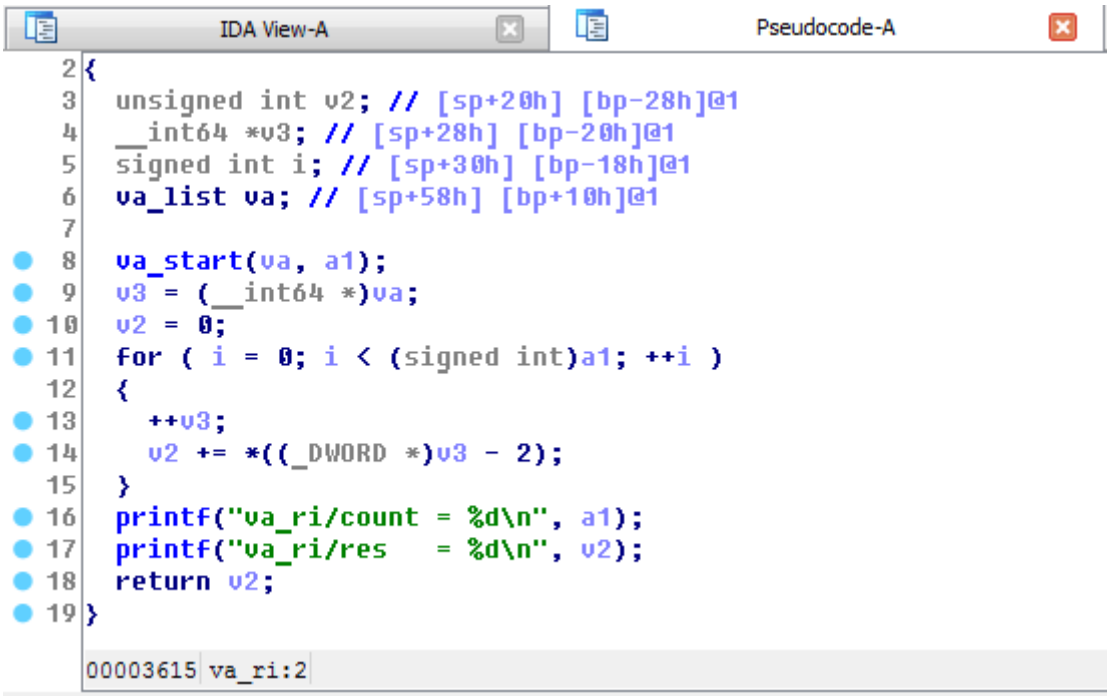
Close to native methods you'll also usually find something like this:

```
1    System.loadLibrary("hello-jni");
```

Which will tell you in which native library the method is implemented. In such cases, you will need an ARM (or x86 if there's a x86 subfolder inside the libs folder) disassembler in order to reverse the native object.

IDA

The very first disassembler and decompiler that every decent reverser should know about is Hex-Rays IDA which is the state of the art reversing tool for native code. Along with an IDA license, you can also buy a decompiler license, in which case IDA will also be able to rebuild pseudo C-like code from the assembly, allowing you to read an higher level representation of the library logic.

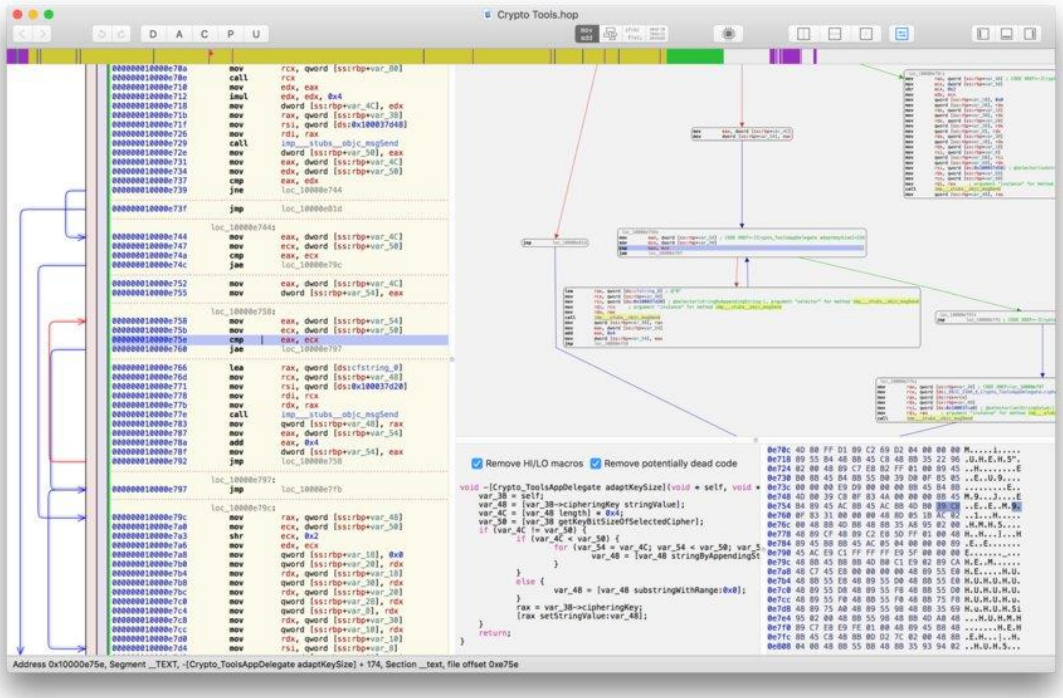


Unfortunately IDA is a very expensive software and, unless you're reversing native stuff professionaly, it's really not worth spending all those money for a single tool … warez … ehm … :P

Hopper

If you're on a budget but you need to reverse native code, instead of IDA you can give Hopper a try. It's definitely not as good and complete as IDA, but it's much cheaper and will be good enough for most of the cases.

Hopper supports GNU/Linux and macOS (no Windows!) and, just like IDA, has a builtin decompiler which is quite decent considering its price:



Dynamic Analysis

When static analysis is not enough, maybe because the application is obfuscated or the codebase is simply too big and complex to quickly isolate the routines you're interested into, you need to go dynamic.

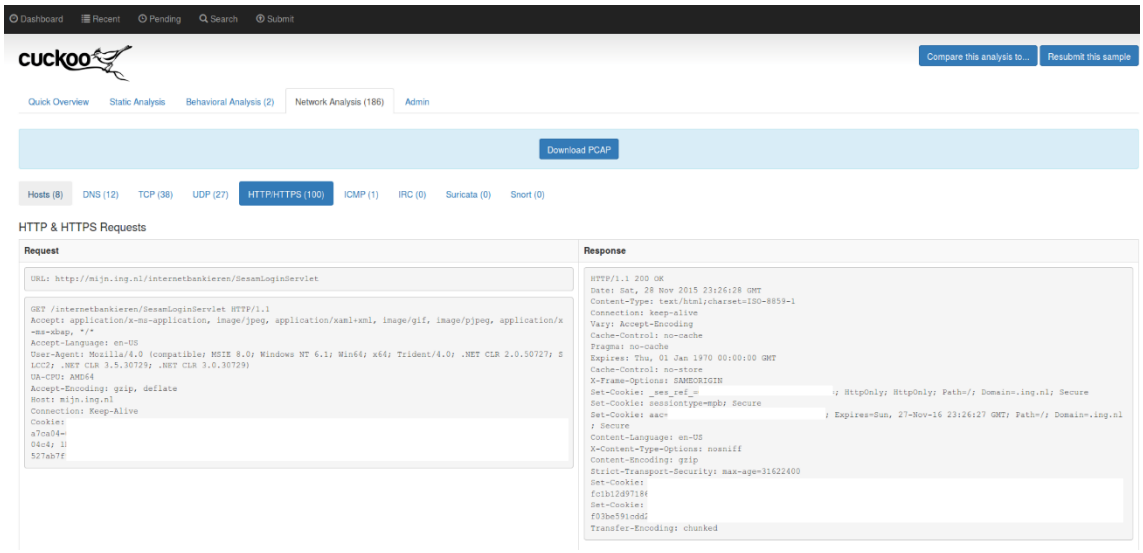
Dynamic analysis simply means that you'll execute the app (like we did while performing network analysis) and somehow trace into its execution using different tools, strategies and methods.

Sandboxing

Sandboxing is a black-box dynamic analysis strategy, which means you're not going to actively trace **into** the application code (like you do while debugging), but you'll execute the app into some container that will log the most relevant actions for you and will present a report at the end of the execution.

Cuckoo-Droid

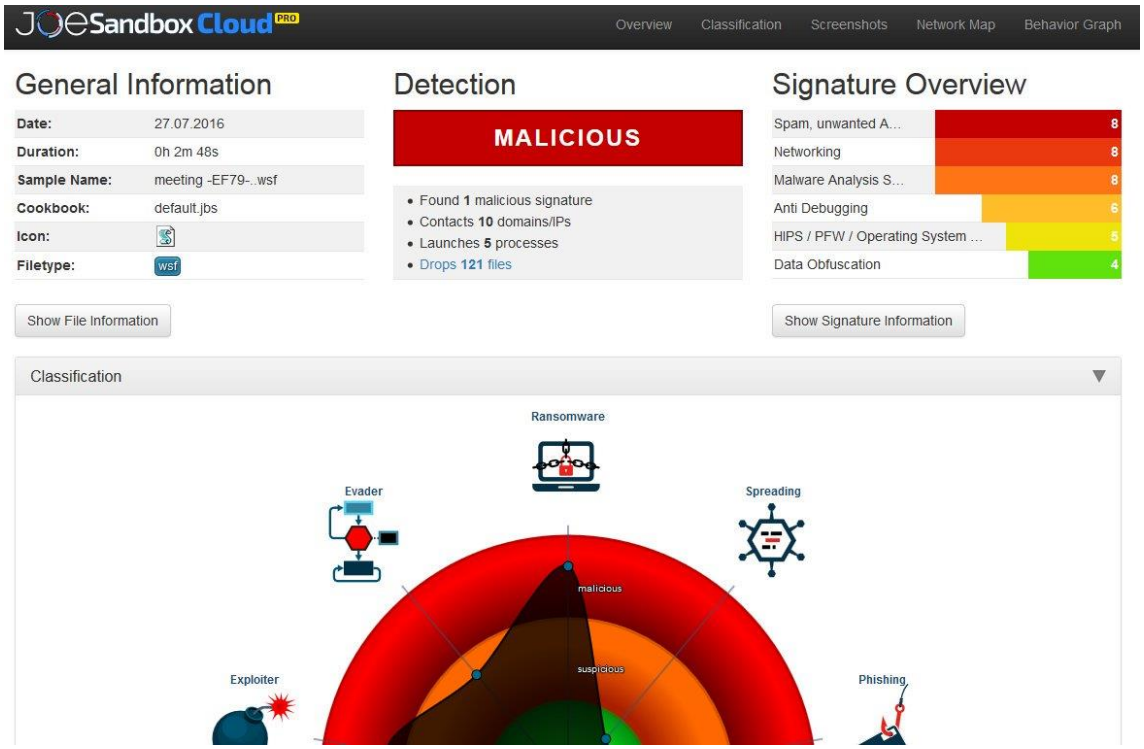
Cuckoo-Droid is an Android port of the famous Cuckoo sandbox, once installed and configured, it'll give you an activity report with all the URLs the app contacted, all the DNS queries, API calls and so forth:



Joe Sandbox

The mobile Joe Sandbox is a great online service that allows you to upload an APK and get its activity report without the hassle of installing or configuring anything.

This is a sample report, as you can see the kind of information is pretty much the same as Cuckoo-Droid, plus there're a bunch of heuristics being executed in order to behaviourally correlate the sample to other known applications.



Debugging

If sandboxing is not enough and you need to get deeper insights of the application behaviour, you'll need to debug it. Debugging an app, in case you don't know, means attaching to the running process with a debugger software, putting breakpoints that will allow you to stop the

execution and inspect the memory state and step into code lines one by one in order to follow the execution graph very closely.

Enabling Debug Mode

When an application is compiled and eventually published to the Google Play Store, it's usually its release build you're looking at, meaning debugging has been disabled by the developer and you can't attach to it directly. In order to enable debugging again, we'll need to use apktool to decompile the app:

apktool d application.apk

Then you'll need to edit the AndroidManifest.xml generated file, adding the android:debuggable="true" attribute to its application XML node:

```
<?xml version="1.0" encoding="utf-8" standalone="no"?>
<manifest xmlns:android="http://schemas.android.com/apk/res/android"
package="com.company.appname"
                                        platformBuildVersionCode="24"
                                        platformBuildVersionName="7.0">
  <uses-permission
android:name="android.permission.WRITE_EXTERNAL_STORAGE"/>
  <uses-permission android:name="android.permission.INTERNET"/>

  <application android:allowBackup="true" android:icon="@mipmap/ic_launcher"
    android:label="@string/app_name"
    android:supportsRtl="true"
    android:theme="@style/AppTheme"
    android:debuggable="true"> <-- !!! NOTICE ME !!! -->

    <activity android:name="com.company.appname.MainActivity">
      <intent-filter>
        <action android:name="android.intent.action.MAIN"/>
        <category android:name="android.intent.category.LAUNCHER"/>
      </intent-filter>
    </activity>
  </application>

</manifest>
```

Once you updated the manifest, let's rebuild the app:

apktool b -d application_path output.apk

Now let's resign it:

git clone https://github.com/appium/sign

java -jar sign/dist/signapk.jar sign/testkey.x509.pem sign/testkey.pk8 output.apk signed.apk

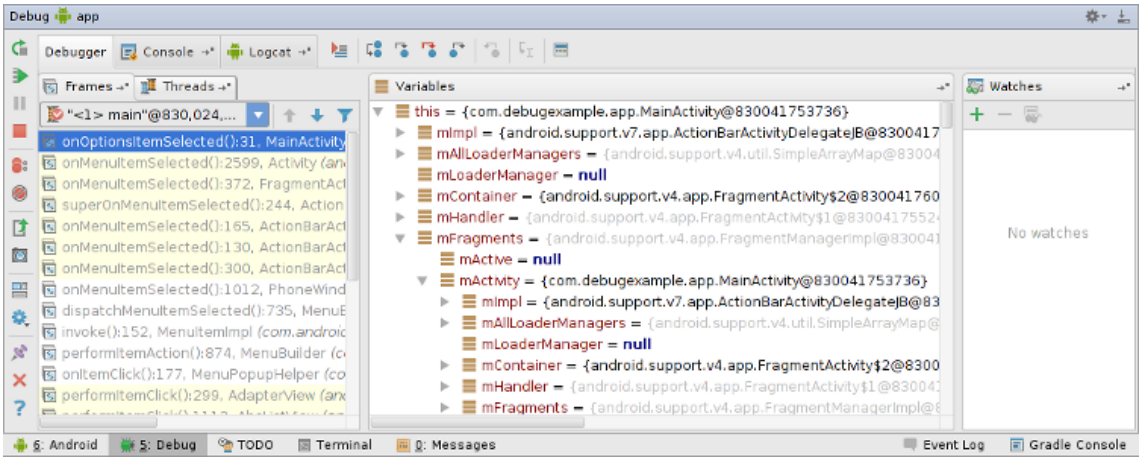
And reinstall it on the device (make sure you unistalled the original version first):

adb install signed.apk

Now you can proceed debugging the app ^_^

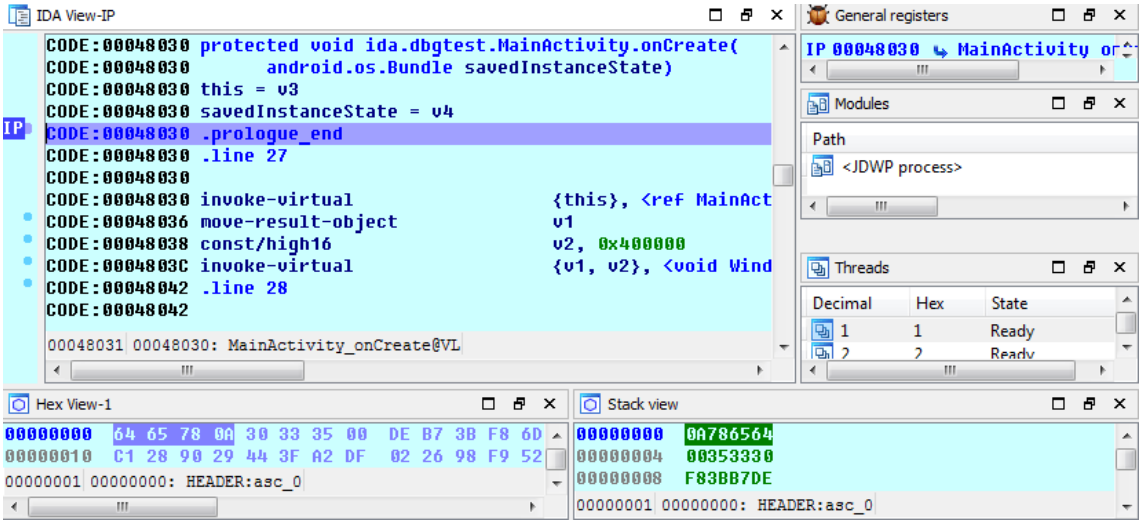
Android Studio

Android Studio is the official Android IDE, once you have debug mode enabled for your app, you can directly attach to it using this IDE and start debugging:



IDA

If you have an IDA license that supports Dalvik debugging, you can attach to a running process and step trough the smali code, this document describes how to do it, but basically the idea is that you upload the ARM debugging server (a native ARM binary) on your device, you start it using adb and eventually you start your debugging session from IDA.



Dynamic Instrumentation

Dynamic instrumentation means that you want to modify the application behaviour at runtime and in order to do so you inject some “agent” into the app that you’ll eventually use to instrument it.

You might want to do this in order to make the app bypass some checks (for instance, if public key pinning is enforced, you might want to disable it with dynamic instrumentation in order to easily inspect the HTTPS traffic), make it show you information it’s not supposed to show (unlock “Pro” features, or debug/admin activities), etc.

Frida

Frida is a great and free tool you can use to inject a whole Javascript engine into a running process on Android, iOS and many other platforms … but why Javascript?

Because once the engine is injected, you can instrument the app in very cool and easy ways like this:

```
1   from __future__ import print_function
2   import frida
3   import sys
4
5   # let's attach to the 'hello process
6   session = frida.attach("hello")
7
8   # now let's create the Javascript we want to inject
9   script = session.create_script("""
10  Interceptor.attach(ptr("%s"), {
11      onEnter: function(args) {
12          send(args[0].toInt32());
13      }
14  });
15  """ % int(sys.argv[1], 16))
16
17  # this function will receive events from the js
18  def on_message(message, data):
19      print(message)
20
21  # let's start!
22  script.on('message', on_message)
23  script.load()
24  sys.stdin.read()
```

In this example, we're just inspecting some function argument, but there're hundreds of things you can do with Frida, just RTFM! and use your imagination :D

Here's a list of cool Frida resources, enjoy!

XPosed

Another option we have for instrumenting our app is using the XPosed Framework. XPosed is basically an instrumentation layer for the whole Dalvik VM which requires you to to have a rooted phone in order to install it.

From XPosed wiki:

There is a process that is called "Zygote". This is the heart of the Android runtime. Every application is started as a copy ("fork") of it. This process is started by an /init.rc script when the phone is booted. The process start is done with /system/bin/app_process, which loads the needed classes and invokes the initialization methods.

This is where Xposed comes into play. When you install the framework, an extended app_process executable is copied to /system/bin. This extended startup process adds an additional jar to the classpath and calls methods from there at certain places. For instance, just after the VM has been created, even before the main method of Zygote has been called. And inside that method, we are part of Zygote and can act in its context.

The jar is located at /data/data/de.robv.android.xposed.installer/bin/XposedBridge.jar and its source code can be found here. Looking at the class XposedBridge, you can see the main method. This is what I wrote about above, this gets called in the very beginning of the process. Some initializations are done there and also the modules are loaded (I will come back to module loading later).

Once you've installed XPosed on your smartphone, you can start developing your own module (again, follow the project wiki), for instance, here's an example of how you would hook the updateClock method of the SystemUI application in order to instrument it:

```
package de.robv.android.xposed.mods.tutorial;

import static de.robv.android.xposed.XposedHelpers.findAndHookMethod;
import de.robv.android.xposed.IXposedHookLoadPackage;
import de.robv.android.xposed.XC_MethodHook;
import de.robv.android.xposed.callbacks.XC_LoadPackage.LoadPackageParam;

public class Tutorial implements IXposedHookLoadPackage {
   public void handleLoadPackage(final LoadPackageParam lpparam) throws
Throwable {
      if (!lpparam.packageName.equals("com.android.systemui"))
         return;

      findAndHookMethod("com.android.systemui.statusbar.policy.Clock",
lpparam.classLoader, "updateClock",
         new XC_MethodHook() {
         @Override
         protected void beforeHookedMethod(MethodHookParam param) throws
Throwable {
            // this will be called before the clock was updated by the original method
         }
         @Override
         protected void afterHookedMethod(MethodHookParam param) throws
Throwable {
            // this will be called after the clock was updated by the original method
         }
   });
   }
}
```

There're already a lot of user contributed modules you can use, study and modify for your own needs.

https://www.evilsocket.net/2017/04/27/Android-Applications-Reversing-101/

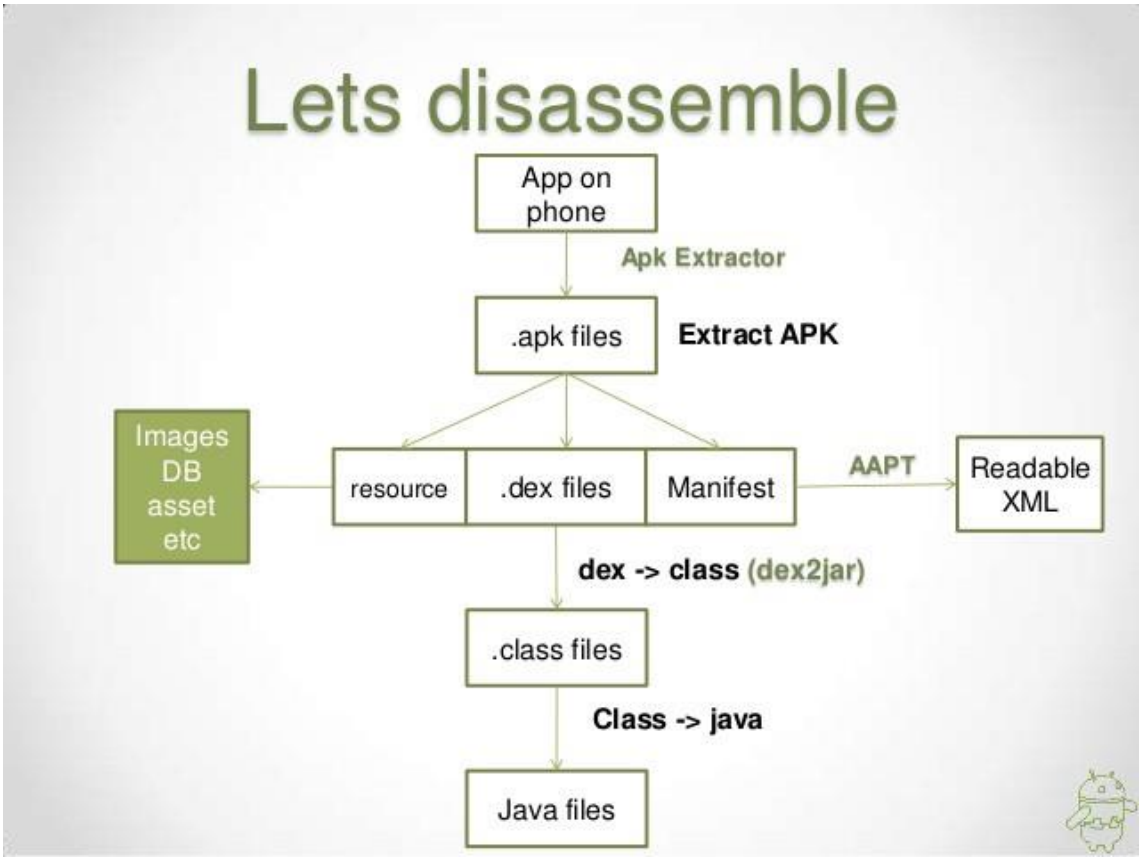
Many beginners or even intermediate Android developers fail to realize that the Android app they build and ship can be reverse engineered to a greater extent. If you are one of the developers who think hard-coding secret keys or even storing it in build.gradle file will prevent it from going into the hands of hackers or other developers, you are wrong.

Security has never been easy and the very first rule is to never trust the security on the client-side. The Client-side is not an environment we control and thus we should not rely on it by hard-coding or storing secrets that can disrupt our system. So the best way to ensure you do not end up getting caught by developers and hackers is to reverse engineer the application by yourself and fix the issues if possible.

Interesting Incident *— Once I was working on an Android Application which required a mathematical formula to be used in a feature. Disliking math to the core, I found it more easy and interesting to reverse engineer one of the competitors application and then I took out the Math formula from the code successfully! :p*

So, let's get started. For reverse engineering an application, we would need a few things beforehand —

1. APK of the application.
2. Some set of codes to execute.
3. Java Decompiler Tool (JD JUI in this article) to view the decompiled code.



Basic understanding of the Android app. Source — Pranay Airan.

First Step —

We would need the APK of the application we want to reverse-engineering. There are many ways to do that but I will suggest a simple way here. Download the app **Apk Extractor** on your device and select the application from the list inside the application. Once done, open any File Explorer and go to the **ExtractedApks** folder present in the Internal Storage directory. There, you will find the APK. Copy that APK to your system and we will proceed with the below steps.

Second Step —

Once we have the APK, we will reverse it to know and see the code. That will enable us and give us insights about the structuring of the code as well as find the security measures they have taken to avoid facing a reverse engineering attack.

Here, we will rename our {app}.apk file to {app}.zip and extract it. Inside the extracted folder, we will find the classes.dex file which contains the application code.

A DEX file is an executable file which contains the compiled code and runs on the Android platform.

Now, we will use the classes.dex file we took from the APK zip file and convert it to JAR. For doing that, we can use **'dex2jar'** open-source tool available **here.** Head over to the release section and download the latest available zip file and extract it. Copy the extracted classes.dex file and paste it inside the **'dex2jar-x.x'** directory.

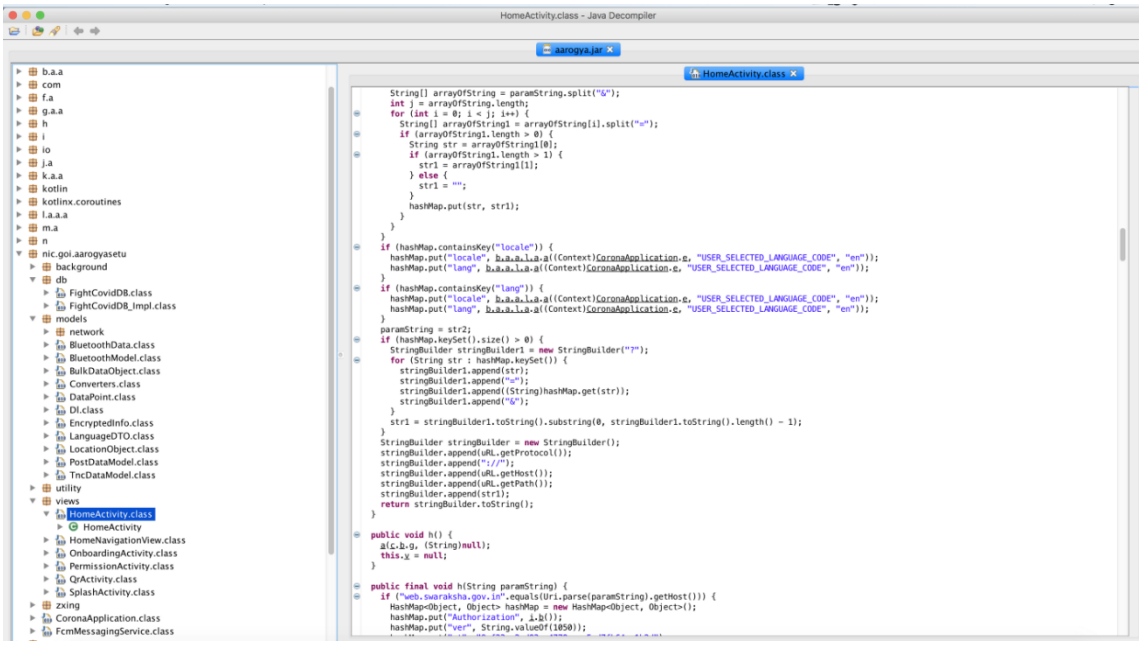
Open Terminal on your machine and head over to the **'dex2jar-x.x'** directory. Now we will run the command -

d2j-dex2jar.bat classes.dex

This will convert the classes.dex file to a JAR file which we can view using any Decompiler Tool.

Third Step —

We will use **JD JUI** which is a simple Java Decompiler tool. You can get it from **here**. Download and extract the zip. Run the **jd-gui.exe** and open the dex file and Voila! We reverse-engineered the application!!



Decompiled Android App Source Code

In case we are meeting for the first time, I am Varun — Founder of **Dwarsoft**. We at Dwarsoft turn ideas into reality with the speed of light and the same perfection of Dettol killing the germs. Hit me up with your ideas and let us make it a reality together!

https://medium.com/dwarsoft/how-to-reverse-engineer-an-android-application-in-3-easy-steps-dwarsoft-mobile-880d268bdc90

I had always wanted to learn how to reverse engineer Android apps. There were people out there who knew how to navigate and modify the internals of an APK file and I wasn't one of them. This had to be changed but it took a long time for that to happen. In this post, I will show you how I was able to reverse engineer an Android app, add some debug statements, and figure out how certain query parameters for API calls were being generated. It should give you a fairly good idea of how APK reverse engineering generally works.

Backstory

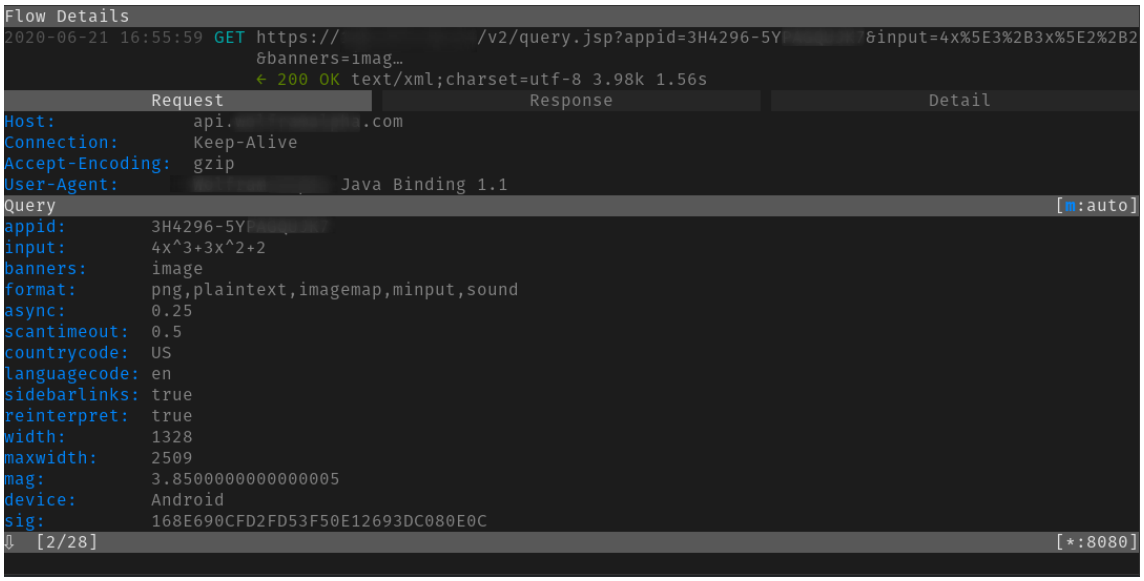
You might be wondering what fueled this curiosity so let me explain. I was in high school and was preparing for my advanced maths exam. I had recently learned that I could use a certain app to get step-by-step solutions to my problems. I was excited. I tried the app and it worked! It only required a one time purchase fee and that was it. I had a lot of questions about how the app worked under the hood:

- Was the processing happening on the phone?
- Were they making any API calls?
- If so then what were the calls?

Simple, innocent questions that led me into a rabbit hole. I tried reverse-engineering the app but couldn't get far. I eventually decided to put the project on the back burner and come back to it once I had more time and experience. It only took 3 years, a whole lot of learning, and a renewed interest in reverse-engineering for me to come back to this project.

I decided to have a fresh start at the problem and figure out if I even need to go as far as decompiling the APK. Maybe just a simple MITM attack would be enough to snoop the API calls and craft my own?

I currently have an iPhone so I installed the Android Emulator on my Linux machine and install the app on that. Then I launched mitmproxy and started intercepting the traffic from the emulator. Whenever I made a query, this API call was made:



So far so good. No need for learning how to reverse-engineer the app. Surely I can figure out what those query parameters are? As it turns out it was extremely hard to figure out how the sig parameter was being generated. Everything else seemed generic enough but sig was dynamic and changed with a change in input.

I tried modifying the input slightly just to check if the API was even checking the sig parameter. As it turns out, it was. The endpoint returned an invalid signature error even on the slightest change in input:

This XML file does not appear to have any style information associated with it. The document tree is shown below.

```
−<queryresult success="false" error="true" numpods="0" datatypes="" timedout="" timedoutpods="" timing="0.025"
 parsetiming="0." parsetimedout="false" recalculate="" id="" parseidserver="47" host="https:// .com"
 server="47" related="" version="2.6">
  −<error>
     <code>3</code>
     <msg>Invalid signature</msg>
   </error>
 </queryresult>
```

sig was some sort of hash but I wasn't sure what kind or how it was being generated and now this required a little bit of reverse engineering.

Note: While trying to proxy the android emulator requests via mitmproxy, you might see the error: Client connection killed by block_global. To fix this, make sure you run mitmproxy with the block_global flag set to false: mitmproxy --set block_global=false. You will also have to install the mitmproxy certificate on the device to decrypt the SSL traffic. Follow these instructions to do that.

Downloading and unpacking the APK

Disclaimer: I do not condone piracy. This is merely an exercise to teach you how something like this works. The same knowledge is used to reverse malware apps and to disable certificate pinning in APKs. There will be places throughout the article where I will censor the name of the application or the package I am reversing. Do not do anything illegal with this knowledge.

The very first step is to get your hands on the APK file. There are multiple ways to do that. You can either use ADB to get an APK from an Android device (emulated or real) or you can use an online APK website to download a working version of the app. I opted for the latter option. Search on Google and you should be able to find a way to download APKs pretty easily.

Let's suppose our APK is called application.apk. Now we need to figure out how to unpack the APK into a folder with all the resources and Dalvik bytecode files. We can easily do that using the apktool. You can easily download the apktool from this link.

At the time of writing, the most recent version was apktool_2.4.1.jar. Put this file wherever you want (in my case ~/Dev) and add an alias to it in your .bashrc for ease of use:

alias apktool='java -jar ~/Dev/apktool_2.4.1.jar'

I had to install JDK to get it to work so make sure you have it installed.

Now we can use apktool to actually unpack the APK:

apktool d application.apk

This should give you an application folder in the same directory where application.apk is located. The structure of the application folder should look something like this:

$ ls application

AndroidManifest.xml assets lib res smali_classes2

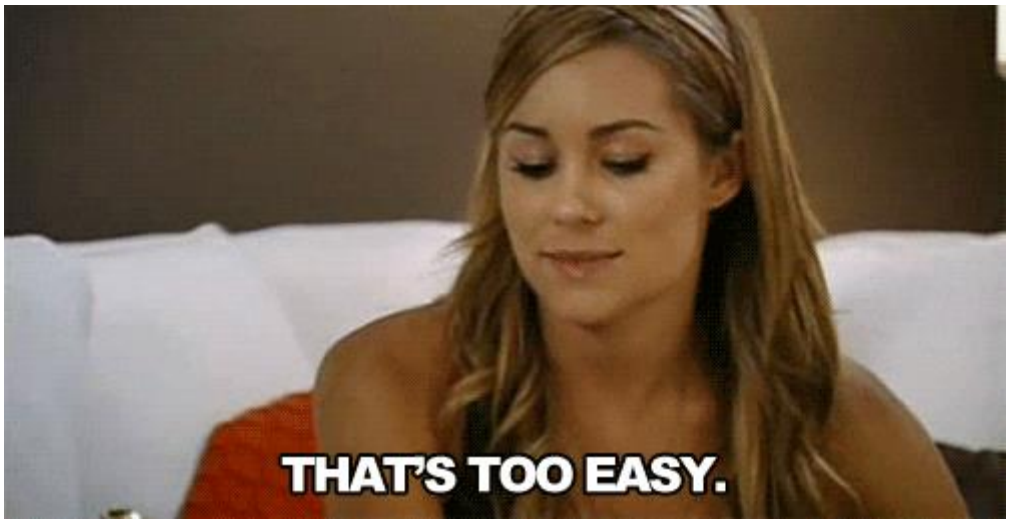
apktool.yml kotlin original smali unknown

Sweet!

What about JADX?

At the time of writing this article, the most famous tool for decompiling an APK is probably JADX. It converts an APK file into a human-readable Java output. Even though the decompilation of an APK using JADX usually gives you fairly readable Java code, the decompilation process is lossy and you can't simply recompile the app using Android Studio.

This is where I got stuck in the past as well. I used to assume that you can simply recompile a decompiled APK and it would work. If only APK reverse-engineering was this easy…



So wait! Does this mean we won't be using JADX at all? Quite the contrary. It is super useful to have the decompiled source code available even if it isn't in a functional state. It will help us in figuring out the internals of how the app works and which methods we need to modify in the smali code.

This is the perfect time to use JADX to decompile the APK. The hosted version of JADX is pretty neat. You can access it here. Just give it the APK and it will give you a zip file containing the decompiled source.

Seeing the following string in multiple places in the decompiled output gave me a good chuckle:

"ModGuard - Protect Your Piracy v1.3 by ill420smoker"

Introducing smali

So what are our options if JADX doesn't work? We are gonna do the next best thing and decompile the APK into smali code. Think of smali as assembly language. Normally when you compile your Java code into an APK, you end up with .dex files (inside the APK) which aren't human-readable. So we convert the .dex code into .smali code which is a human-readable representation of the .dex code. You can read more about where smali fits in the compilation life-cycle in this wonderful answer by Antimony on StackOverflow.

This is what smali code looks like:

invoke-**interface** {p1}, Ljava/util/Iterator;->hasNext()Z

move-result v2

This is equivalent to calling the hasNext method of java.util.Iterator. Z tells us that this call returns a boolean. p1 is called a parameter register and that is what the hasNext() is being called on. move-result v2 moves the return value of this method call to the v2 register.

It probably won't make a lot of sense right now but I will explain the required bits in a bit. This is just to give you an idea of what to expect. If you are interested, I highly recommend you to take a look at this wonderful presentation about Android code injection. It gives some useful details about smali code as well.

There is also a smali cheat-sheet that was super helpful for me to understand the basics of smali.

Finding the signature location

I had to find out where the &sig= query parameter was being added to in the smali code. It was fairly simple to figure this out using grep.

$ grep -r 'sig=' ./smali/com/

./simple/SimpleApi.smali: **const-string** v2, "&sig="

./impl/WAQueryImpl.smali: **const-string** v1, "&sig="

I started my exploration from there. I used the output of JADX to explore where this parameter was being populated. This is where having the decompiled source code was really useful. The file structure in the apktool output and jadx output is the same so we can explore the output of JADX to help us figure out where to insert the debug statements in smali.

After exploring the Java output for a while I found the method that was generating the signature. The signature was just an MD5 hash of the rest of the query parameters which were being sent to the server:

```
private String getMd5Digest(WAQueryParameters wAQueryParameters) {
  // ...
  StringBuilder sb = new StringBuilder(600);
  sb.append("vFdeaRwBTVqdc5CL");
  for (String[] strArr : parameters) {
    sb.append(strArr[0]);
    sb.append(strArr[1]);
  }
  try {
    MessageDigest instance = MessageDigest.getInstance("MD5");
    instance.update(sb.toString().getBytes());
    return String.format("%1$032X", new Object[]{new BigInteger(1, instance.digest())});
  } catch (NoSuchAlgorithmException unused) {
    return "";
  }
}
```

Now the only issue was I didn't know what query parameters were being passed to this method. I tried generating an MD5 hash in Python based on the parameters I saw in the URL but I was failing. If only I had a log statement that showed me the value of wAQueryParameters…

Adding logging in smali

The equivalent smali code for the getMd5Digest method was:

.method **private getMd5Digest**(Lcom._____/WAQueryParameters;)Ljava/lang/String;

.locals 6

const/4 v5, 0x1

.line 1196

const/4 v5, 0x2

invoke-**interface** {p1}, Lcom/____/_____/WAQueryParameters;->getParameters()Ljava/util/List;

move-result-**object** p1

..snipped for brevity

.end method

The move-result-object call was moving the output of getParameters to the p1 register. This is where I needed a log statement (or so I thought). I did some research and according to StackOverflow I could do something like this:

const-string v8, "log-tag"

invoke-**static** {v8, v9}, Landroid/util/Log;->d(Ljava/lang/String;Ljava/lang/String;)I

This would print a debug statement in the logcat output and print the value of the v9 register. There were a couple of things to take care of if I were to use this snippet. I had to make sure that v8 was actually a register declared for local use in the method (go through this PDF if you don't know what I mean) and that I was not over-writing a value in that register that was going to be used later in the method by the original code. And additionally, I wanted to print the value of p1 and it was not of the java.lang.String type.

The code wasn't all that hard to modify but it took me an embarrassingly long time to figure out the correct statements to insert in smali.

Firstly, I changed .locals 6 to .locals 7. This is useful because instead of tracing which register I could safely use for my custom code, why not allow the function access to a new register? That way we can be sure that no original code in the method is using the new register.

Then I inserted the following lines:

const-string v6, "YASOOB getMd5Digest"

invoke-**static** {v6, p1}, Landroid/util/Log;->d(Ljava/lang/String;Ljava/lang/String;)I

Repacking the unpacked APK

After the smali modifications, we have to repack the APK. This isn't terribly hard if you have the tooling already set up. We will do this in steps.

- Building the APK

If the output of apktool d application.apk was ~/application then simply go to ~ (your home folder) and run:

apktool b application

This will generate an application.apk file in the ~/application/dist folder.

- Signing the APK

Android doesn't allow you to install unsigned APKs. If you have the Android SDK installed then you already have a debug keystore that you can use to sign an APK. The command for doing that is this:

jarsigner -verbose -sigalg MD5withRSA -digestalg SHA1 -keystore ~/.android/debug.keystore -storepass android ~/application/dist/application.apk androiddebugkey

This will use the debug.keystore file in the ~/.android folder to sign the APK.

- Aligning the APK

You have to make sure to align your APK files using a tool called zipalign. It comes as part of the Android SDK. According to the Android docs:

zipalign is an archive alignment tool that provides important optimization to Android application (APK) files. The purpose is to ensure that all uncompressed data starts with a particular alignment relative to the start of the file.

You can use zipalign like this:

zipalign -v 4 ~/application/dist/application.apk ~/application/dist/application-aligned.apk

This will generate an application-aligned.apk file.

Installing the modified APK & Logging

I used an Android Emulator for this step. Specifically a Pixel 3XL emulator image with API 28 and Android Oreo. Make sure that you use an emulator Android image without Google Play. This is extremely important because otherwise in later steps ADB will give you an error saying adbd cannot run as root in production builds. You can find a detailed solution on StackOverflow.

Once you have an emulator set up you need to make sure that the original APK isn't installed on the device. It is fairly easy to uninstall an APK using adb. If the package name for the app is com.yasoob.app, you can uninstall it using the following command:

adb uninstall com.yasoob.app

Once it is uninstalled, you can install the modified version:

adb install -r ~/application/dist/application-aligned.apk

Now run the installed app in the emulator and run logcat in a terminal on the host machine:

adb logcat | grep 'YASOOB'

The output of logcat is super noisy and it outputs a lot of stuff we don't care about. That is why we use grep to only output the debug statements we actually care about.

In my case the output of logcat looked something like this:

YASOOB getMd5Digest: [[Ljava.lang.String;@ea7d7a1, [Ljava.lang.String;@8b00dc6, [Ljava.lang.String;@44c6187, [Ljava.lang.String;@fa6d8b4, [Ljava.lang.String;@f494cdd, [Ljava.lang.String;@78f4052, [Ljava.lang.String;@f038f23, [Ljava.lang.String;@232cc20, [Ljava.lang.String;@a92d9d9, [Ljava.lang.String;@5bd0f9e, [Ljava.lang.String;@e75fa7f, [Ljava.lang.String;@7c88a4c, [Ljava.lang.String;@d4e3a95]

This is exciting and disappointing at the same time. Exciting because the apk didn't crash (it crashed quite a few times before I figured out the correct smali code for logging a list) and disappointing because Java outputted the references to Strings and not the string values themselves. But hey! At least we made some progress and our repacked APK isn't crashing!

At this point I merged all of the APK building and installation commands into one huge command so that I don't have to continuously execute them one by one:

apktool b application && \

jarsigner -verbose -sigalg MD5withRSA -digestalg SHA1 -keystore ~/.android/debug.keystore -storepass android ~/application/dist/application.apk androiddebugkey && \

rm ~/application/dist/application-aligned.apk && \

zipalign -v 4 ~/application/dist/application.apk ~/application/dist/application-aligned.apk && \

adb uninstall com.yasoob.app && \

adb install -r ~/application/dist/application-aligned.apk && \

adb logcat | grep 'YASOOB'

Fixing the debug statement in smali

Ok, we had a working debug statement but it didn't give us the information we needed. I looked at the smali code again and saw the following statements:

```
.method private getMd5Digest(Lcom._____/WAQueryParameters;)Ljava/lang/String;

   #--snipped for brevity--

   aget-object v4, v2, v1


   invoke-virtual {v0, v4}, Ljava/lang/StringBuilder;-
>append(Ljava/lang/String;)Ljava/lang/StringBuilder;


   .line 1211

   const/4 v5, 0x0


   aget-object v2, v2, v3


   invoke-virtual {v0, v2}, Ljava/lang/StringBuilder;-
>append(Ljava/lang/String;)Ljava/lang/StringBuilder;


   #--snipped for brevity--

.end method
```

This is where the application was appending the query parameters to a StringBuilder which is eventually used to generate the MD5 hash. Why didn't I simply put a debug statement here? We know that StringBuilder expects a String so hopefully Java will output the value of String this time instead of the reference.

This is how the modified code looked like:

```
.method private getMd5Digest(Lcom._____/WAQueryParameters;)Ljava/lang/String;

   #--snipped for brevity--

   aget-object v4, v2, v1


   invoke-virtual {v0, v4}, Ljava/lang/StringBuilder;-
>append(Ljava/lang/String;)Ljava/lang/StringBuilder;


   # Custom Code YASOOB

   const-string v6, "YASOOB QueryTask->getMd5Digest::FirstAppend"
```

```
    invoke-static {v6, v4}, Landroid/util/Log;->d(Ljava/lang/String;Ljava/lang/String;)I

    # Custom Code end


    .line 1211
    const/4 v5, 0x0


    aget-object v2, v2, v3


    invoke-virtual {v0, v2}, Ljava/lang/StringBuilder;-
>append(Ljava/lang/String;)Ljava/lang/StringBuilder;



    # Custom Code YASOOB
    const-string v6, "YASOOB QueryTask->getMd5Digest::SecondAppend"
    invoke-static {v6, v2}, Landroid/util/Log;->d(Ljava/lang/String;Ljava/lang/String;)I
    # Custom Code end


    #--snipped for brevity--
.end method
```

While I was at it, I added some more debug statements in a couple of additional places in the same file but different methods. There was one method that was calling this getMd5Digest method and another that outputted the actual API URL with the query parameters. I added a debug statement in both of these.

I rebuilt the APK and started monitoring the logs in logcat:

```
YASOOB QueryTask->getMd5Digest::FirstAppend: appid
YASOOB QueryTask->getMd5Digest::SecondAppend: ****-*********
YASOOB QueryTask->getMd5Digest::FirstAppend: async
YASOOB QueryTask->getMd5Digest::SecondAppend: 0.25
YASOOB QueryTask->getMd5Digest::FirstAppend: banners
YASOOB QueryTask->getMd5Digest::SecondAppend: image
YASOOB QueryTask->getMd5Digest::FirstAppend: countrycode
YASOOB QueryTask->getMd5Digest::SecondAppend: US
```

YASOOB QueryTask->getMd5Digest::FirstAppend: device

YASOOB QueryTask->getMd5Digest::SecondAppend: Android

YASOOB QueryTask->getMd5Digest::FirstAppend: format

YASOOB QueryTask->getMd5Digest::SecondAppend: png,plaintext,imagemap,minput,sound

YASOOB QueryTask->getMd5Digest::FirstAppend: input

YASOOB QueryTask->getMd5Digest::SecondAppend: 1

YASOOB QueryTask->getMd5Digest::FirstAppend: languagecode

YASOOB QueryTask->getMd5Digest::SecondAppend: en

YASOOB QueryTask->getMd5Digest::FirstAppend: mag

YASOOB QueryTask->getMd5Digest::SecondAppend: 3.8500000000000005

YASOOB QueryTask->getMd5Digest::FirstAppend: maxwidth

YASOOB QueryTask->getMd5Digest::SecondAppend: 2509

YASOOB QueryTask->getMd5Digest::FirstAppend: reinterpret

YASOOB QueryTask->getMd5Digest::SecondAppend: **true**

YASOOB QueryTask->getMd5Digest::FirstAppend: scantimeout

YASOOB QueryTask->getMd5Digest::SecondAppend: 0.5

YASOOB QueryTask->getMd5Digest::FirstAppend: sidebarlinks

YASOOB QueryTask->getMd5Digest::SecondAppend: **true**

YASOOB QueryTask->getMd5Digest::FirstAppend: width

YASOOB QueryTask->getMd5Digest::SecondAppend: 1328

YASOOB QueryTask->setSignatureParameter: 7A1AE2AD7F5F81C85B8A4D0FC2723C8C

YASOOB WAQueryImpl->toString:
&input=1&banners=image&format=png,plaintext,imagemap,minput,sound&async=0.25&scantimeout=0.5&countrycode=US&languagecode=en&sidebarlinks=**true**&reinterpret=**true**&width=1328&maxwidth=2509&mag=3.8500000000000005&device=Android&sig=7A1AE2AD7F5F81C85B8A4D0FC2723C8C

This is amazing! Now I knew which parameters, and in what order, are being used to generate the MD5 hash. I quickly whipped out my trusty Visual Studio Code and wrote down a super simple Python script for generating this hash for me based on custom inputs. This is what I came up with:

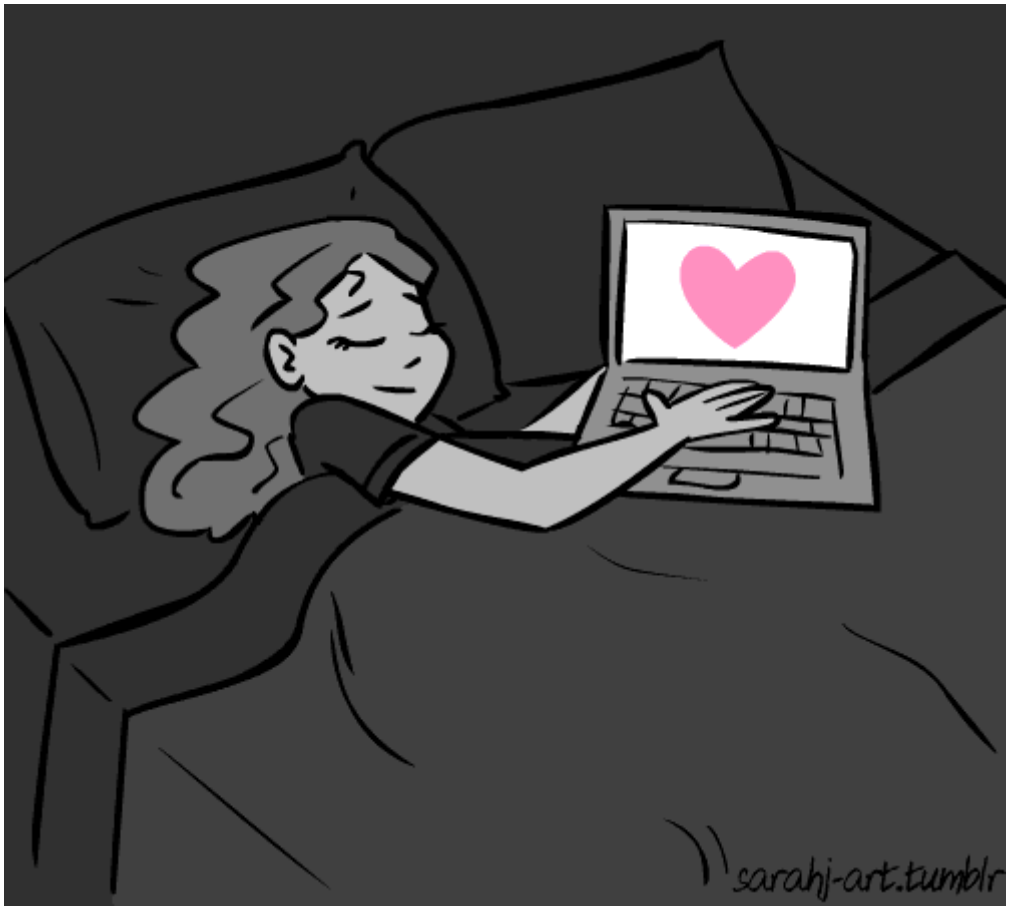
```
import hashlib


url = "https://api.********.com/v2/query.jsp?"

sb = hashlib.md5()
```

```
sb.update("vFdeaRwBTVqdc5CL".encode())


input = "4x^3+3x^2+2x"
data = {
   "appid":"*****-********",
   "async":"0.25",
   "banners":"image",
   "countrycode":"US",
   "device":"Android",
   "format":"png,plaintext,imagemap,minput,sound",
   "input": input,
   "languagecode":"en",
   "mag":"3.8500000000000005",
   "maxwidth":"2509",
   "reinterpret":"true",
   "scantimeout":"0.5",
   "sidebarlinks":"true",
   "width":"1328"
}


for k, v in data.items():
   sb.update(k.encode())
   sb.update(v.encode())
   base_url += f"&{k}={v}"


url += f"&sig={sb.hexdigest()}"
print(url)
```

I ran the program and the resulting URL was the same one I was seeing in mitmproxy. I modified the query, ran the Python program again and the resulting URL worked!

This is where I stopped my exploration. The original aims were to figure out how the sig hash was being generated and how to reverse-engineer the API to make custom query calls. I was able to accomplish both of those aims and my curiosity was satisfied.

I placed the code in an “old projects” folder, looked at the clock and sighed. I had promised myself that I would sleep at midnight but the clock was showing 4:30 am. Nevertheless, it was time for some hard-earned rest.



Useful Tips

- If you don’t know what the smali output for some Java code is supposed to be, create a new Android project, write down the code in Java and see the resulting smali using apktool. There is no better way to learn smali than actually seeing what your own Java (or Kotlin) code compiles to.
- There might be situations where a function/method is using the maximum allowed registers and you have no idea how to output a debug log. In those scenarios, you have to be a bit more creative and do some register shifting. The following commands will be super useful in those cases:
- move-**object** vx,vy
- move-**object** vy,vx

This is moving the object reference from vy to vx and back. This will allow you to temporarily shuffle register values, do your debugging, and then put the original register values back.

- If for some reason your repacked APK is giving you an error and not working, try to repack the APK without any modifications first. This will make sure that your modifications aren’t the reason why the repacked APK isn’t working. It is also really useful to use adb logcat without grep when debugging issues

https://www.youtube.com/watch?v=uc7eZGE07ps

https://www.youtube.com/watch?v=frrTYG6T_yw

- apktool — tool for reverse engineering Android apk files. In this case we are using to extract files from apk and rebuild.
- keytool — Java tool for creating keys/certs, that comes with the JDK.
- jarsigner Java tool for signing JAR/APK files, that comes with the JDK.
- zipalign — archive alignment tool, that comes with the Android SDK.
- JD-GUI — To view java code
- dex2jar — Converts Android dex files to class/jar files.

Instructions:

First, Take any apk file and unpack(decompile) it. This will create an “application” directory with assets, resources, compiled code, etc.

To decompile an apk

- apktool d -r -s application.apk

or

- apktool d application.apk

This below part is to see convert Dex files to java files.
You can skip this part if you don’t wish to check the Code. You can still make some changes in manifest.xml and res folder.

If you wish to decompile any java files, you can do the following:

Convert the Dex files into standard class files

- dex2jar application/classes.dex

Now use the JD (Java Decompiler) to inspect the source

- jd-gui classes-dex2jar.jar
- You can just change the orientation of the main activity in the Manifest file or you can change the app name from strings.xml file.(just to check the recompiled apk is working or not)

Once you have made your changes, you need to repack the APK. This will create a my_application.apkfile:

To recompile(build) the apk

- apktool b -f -d application

After recompiling (building) the apk the new apk will be generated in Dist folder.

Application — Dist- application.apk

The APK must be signed before you run on your device.

Before signing an apk, create a key if you don't have an existing one. If prompted for a password, create your own password.

To generate a key.

- keytool -genkey -v -keystore my-release-key.keystore -alias alias_name \

-keyalg RSA -keysize 2048 -validity 10000

Now sign the APK with the key:

Sign the apk

- jarsigner -verbose -sigalg SHA1withRSA -digestalg SHA1 -keystore my-release-key.keystore my_application.apk alias_name

Verify apk

- jarsigner -verify -verbose -certs my_application.apk

Finally, the apk must be aligned for optimal loading:

- ./zipalign -v 4 my_application.apk my_application-aligned.apk

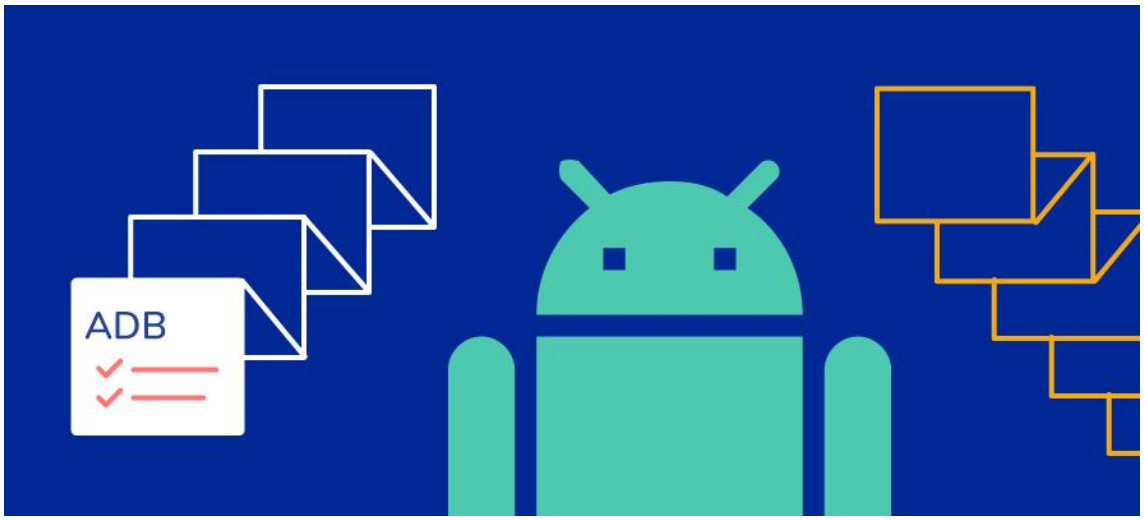
you have a my_application-aligned.apk file, which you can install onto your device.

Android ADB Cheatsheet

ADB, Android Debug Bridge, is a command-line utility included with **Google's Android SDK**. ADB can control your device over USB from a computer, copy files back and forth, install and uninstall apps, run shell commands, and more.

Sheet

MOBILE AUTOMATION



The next article from the **mobile test automation series** will be dedicated to the ADB. All you need to to know- the most basic operations to the most advanced configurations.

ADB, Android Debug Bridge, is a command-line utility included with **Google's Android SDK**. ADB can control your device over USB from a computer, copy files back and forth, install and uninstall apps, run shell commands, and more.

While ago when we were working on the first version of the **BELLATRIX test automation framework**, I did this research while I was working on a **similar features for our solution**.

QUICK NAVIGATION

ADB Basics

adb devices (lists connected devices)
adb root (restarts adbd with root permissions)
adb start-server (starts the adb server)
adb kill-server (kills the adb server)
adb reboot (reboots the device)

adb devices -l (list of devices by product/model)
adb shell (starts the backround terminal)
exit (exits the background terminal)
adb help (list all commands)
adb -s <deviceName> <command> (redirect command to specific device)
adb –d <command> (directs command to only attached USB device)
adb –e <command> (directs command to only attached emulator)

Package Installation

adb shell install <apk> (install app)
adb shell install <path> (install app from phone path)
adb shell install -r <path> (install app from phone path)
adb shell uninstall <name> (remove the app)

Paths

/data/data/<package>/databases (app databases)
/data/data/<package>/shared_prefs/ (shared preferences)
/data/app (apk installed by user)
/system/app (pre-installed APK files)
/mmt/asec (encrypted apps) (App2SD)
/mmt/emmc (internal SD Card)
/mmt/adcard (external/Internal SD Card)
/mmt/adcard/external_sd (external SD Card)

adb shell ls (list directory contents)
adb shell ls -s (print size of each file)
adb shell ls -R (list subdirectories recursively)

File Operations

adb push <local> <remote> (copy file/dir to device)
adb pull <remote> <local> (copy file/dir from device)
run-as <package> cat <file> (access the private package files)

Phone Info

adb get-state (print device state)
adb get-serialno (get the serial number)
adb shell dumpsys iphonesybinfo (get the IMEI)
adb shell netstat (list TCP connectivity)
adb shell pwd (print current working directory)
adb shell dumpsys battery (battery status)
adb shell pm list features (list phone features)
adb shell service list (list all services)
adb shell dumpsys activity <package>/<activity> (activity info)
adb shell ps (print process status)
adb shell wm size (displays the current screen resolution)
dumpsys window windows | grep -E 'mCurrentFocus|mFocusedApp' (print current app's opened activity)

Package Info

adb shell list packages (list package names)
adb shell list packages -r (list package name + path to apks)
adb shell list packages -3 (list third party package names)
adb shell list packages -s (list only system packages)
adb shell list packages -u (list package names + uninstalled)
adb shell dumpsys package packages (list info on all apps)
adb shell dump <name> (list info on one package)
adb shell path <package> (path to the apk file)

Configure Settings Commands

adb shell dumpsys battery set level <n> (change the level from 0 to 100)
adb shell dumpsys battery set status<n> (change the level to unknown, charging, discharging, not charging or full)
adb shell dumpsys battery reset (reset the battery)
adb shell dumpsys battery set usb <n> (change the status of USB connection. ON or OFF)
adb shell wm size WxH (sets the resolution to WxH)

Device Related Commands

adb reboot-recovery (reboot device into recovery mode)
adb reboot fastboot (reboot device into recovery mode)
adb shell screencap -p "/path/to/screenshot.png" (capture screenshot)
adb shell screenrecord "/path/to/record.mp4" (record device screen)
adb backup -apk -all -f backup.ab (backup settings and apps)
adb backup -apk -shared -all -f backup.ab (backup settings, apps and shared storage)
adb backup -apk -nosystem -all -f backup.ab (backup only non-system apps)
adb restore backup.ab (restore a previous backup)
adb shell am start|startservice|broadcast <INTENT>[<COMPONENT>]
-a <ACTION> e.g. android.intent.action.VIEW
-c <CATEGORY> e.g. android.intent.category.LAUNCHER (start activity intent)

adb shell am start -a android.intent.action.VIEW -d URL (open URL)
adb shell am start -t image/* -a android.intent.action.VIEW (opens gallery)

Logs

adb logcat [options] [filter] [filter] (view device log)
adb bugreport (print bug reports)

Permissions

adb shell permissions groups (list permission groups definitions)
adb shell list permissions -g -r (list permissions details)

https://technastic.com/adb-commands-list-adb-cheat-sheet/

https://gist.github.com/HugoMatilla/f92682b06068b06a6f2a

https://www.automatetheplanet.com/adb-cheat-sheet/

Rooting Device

https://www.xda-developers.com/root/

Rooting is the Android equivalent of jailbreaking, a means of unlocking the operating system so you can install unapproved apps, deleted unwanted bloatware, update the OS, replace the firmware, overclock (or underclock) the processor, customize anything and so on.

Of course, for the average user, this sounds like -- and can be -- a scary process. After all, "rooting" around in your smartphone's core software might seem like a recipe for disaster. One wrong move and you could end up with bricked handset.

Thankfully, there's a utility that makes rooting a one-click affair: KingoRoot. It's free and it works -- though not with all devices.

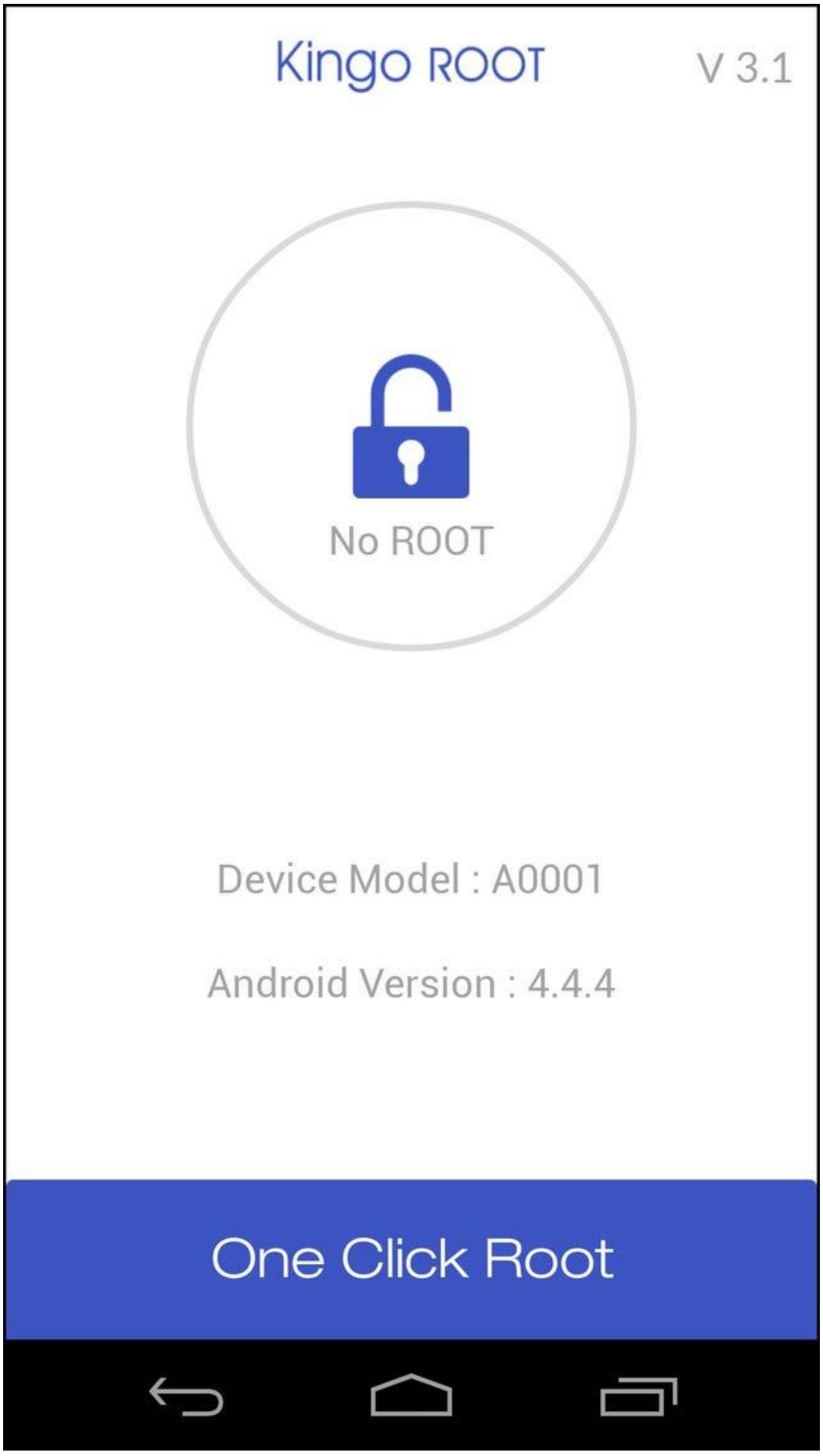
I originally tested Kingo on a Virgin Mobile Supreme and Asus Nexus 7; the process proved quick and easy. More recently, I used it to root a OnePlus One, and this time it was even easier -- because an app did all the work.

However, I couldn't get the utility to work on a Verizon Samsung Galaxy S6. Your mileage may vary, of course, and I definitely recommend checking the compatibility list before proceeding. (Even if your device isn't on it, the utility may work with it.) Here's how to get started.

The app version

The easiest way to use KingoRoot is to install the app version, which literally performs the root process with just one tap.

In fact, the only complicated part is actually getting that app onto your Android device. That's because it's not available in the Google Play Store; instead, you must download the KingoRoot APK and manually install it.



Screenshot by Rick Broida/CNET

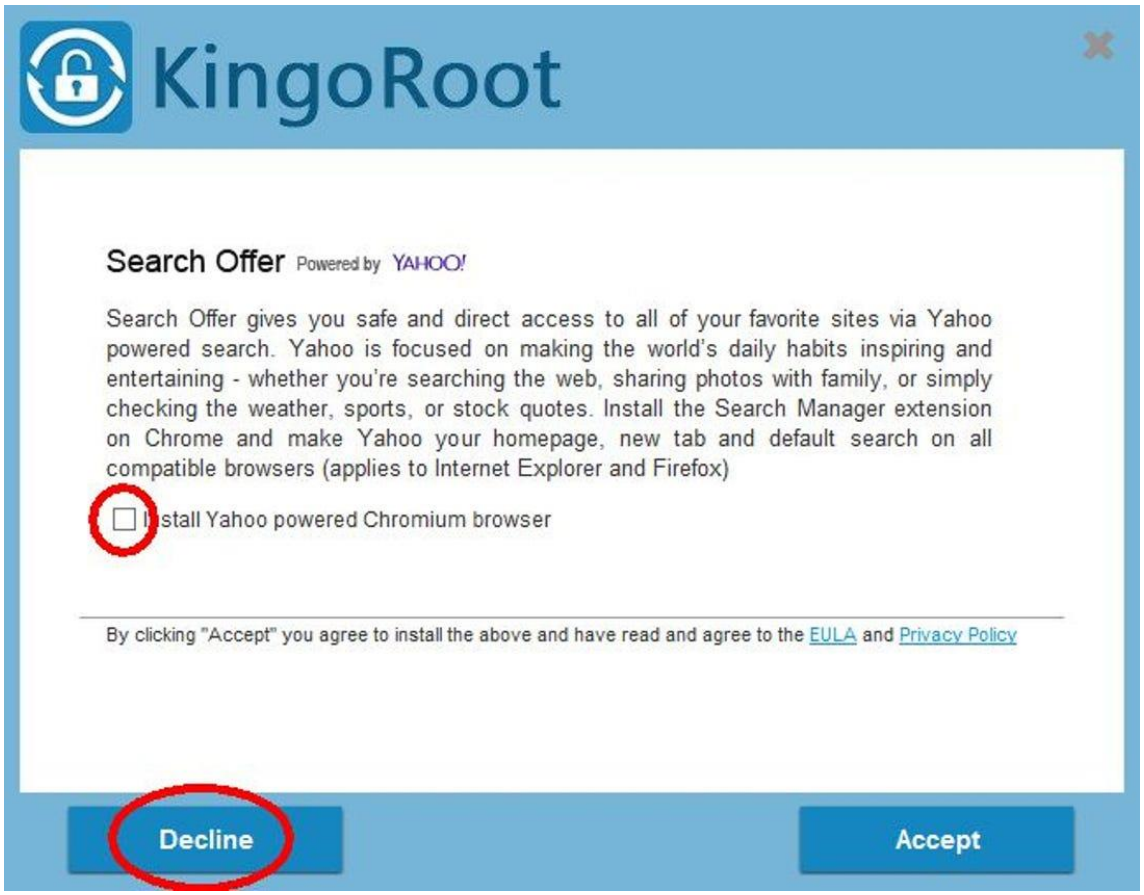
Ideally, you'll just point your device's mobile to the KingoRoot Android page and download it directly. If that doesn't work for some reason, or you're working from your PC, download the APK and email it to yourself as an attachment. Then, on your device, open that e-mail and download that attachment.

To install it, however, you'll need to make sure your device is set to allow apps from unknown sources. In most versions of Android, that goes like this: Head to **Settings**, tap **Security**, scroll down to **Unknown Sources** and toggle the switch to the on position.

Now you can install KingoRoot. Then run the app, tap **One Click Root**, and cross your fingers. If all goes well, your device should be rooted within about 60 seconds. (On my aforementioned Galaxy S6, the process made it to 90 percent, then the phone crashed and rebooted. Luckily, no harm done.)

The desktop version

Kingo's support pages suggested I might have better luck with the Galaxy S6 if I tried the Windows version of KingoRoot. Here's that process:

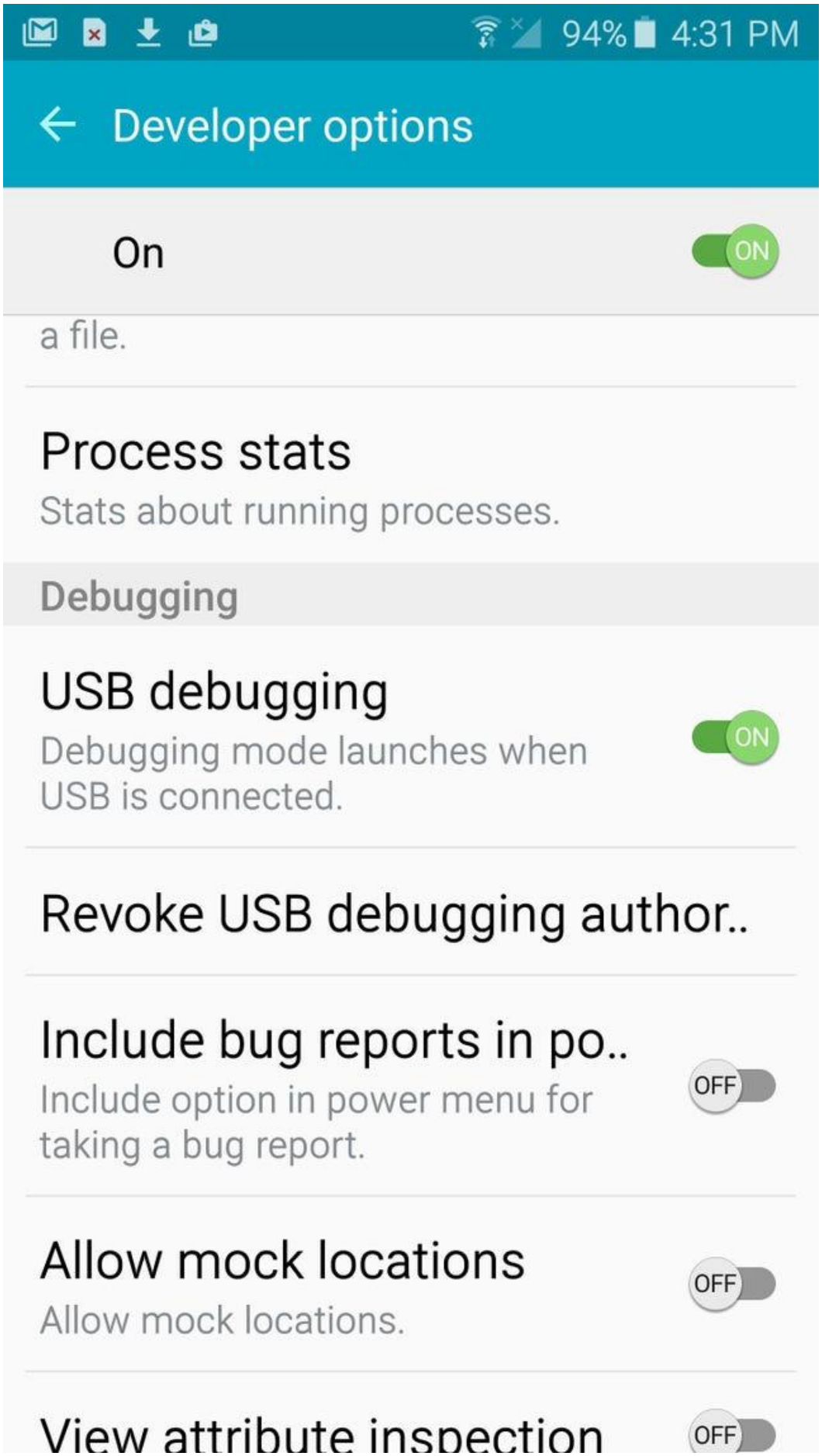


No adware! Leave this box unchecked and be sure to click Decline during installation.

Screenshot by Rick Broida/CNET

Step 1: Download and install KingoRoot for Windows, making sure to leave *unchecked* the option to "Install Yahoo powered Chromium browser" and then click **Decline** to prevent any other adware incursions.

Step 2: Enable USB debugging mode on your phone. If it's running Android 4.0 or 4.1, tap Settings, Developer Options, then tick the box for "USB debugging." (You may need to switch "Developer options" to On before you can do so.) On Android 4.2, tap Settings, About Phone, Developer Options, and then tick USB debugging." Then tap OK to approve the setting change.



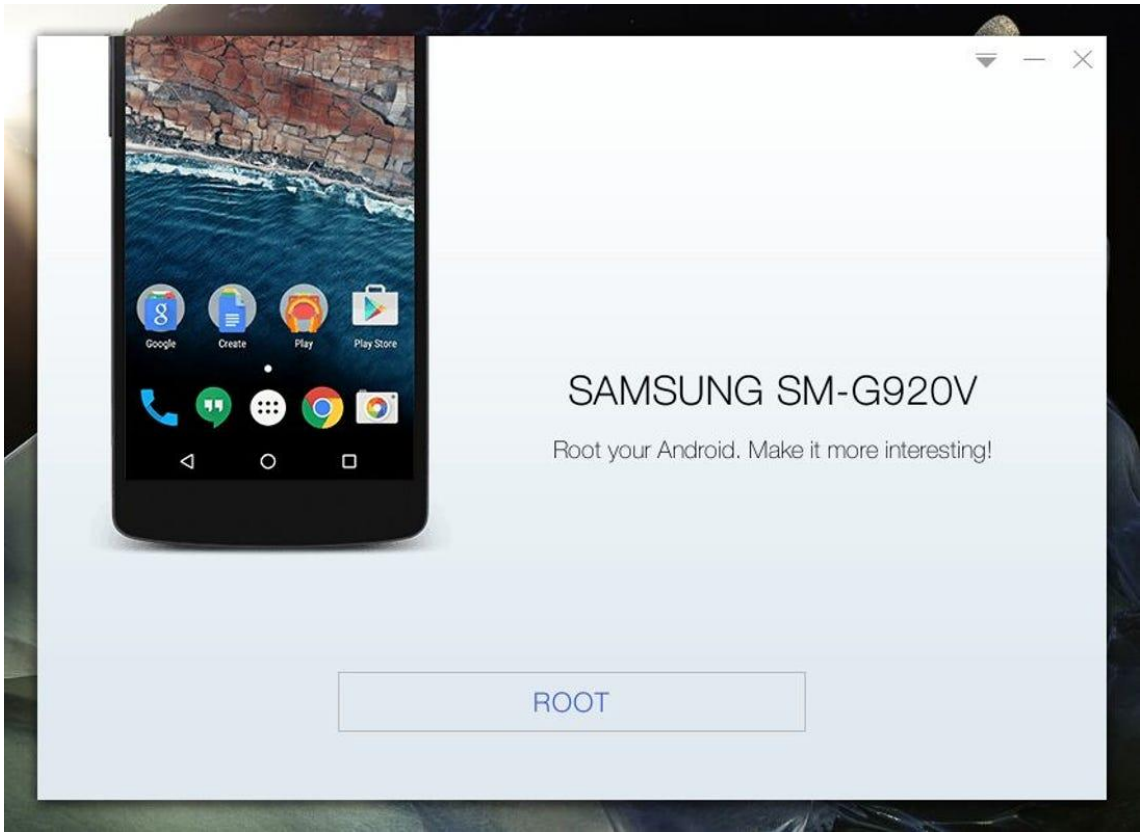
Enlarge Image

Screenshot by Rick Broida/CNET

On Android 4.3 and later (including 5.0, though this also applies to some versions of 4.2), tap Settings, About Phone, then scroll down to Build Number. Tap it seven times, at which point you should see the message, "You are now a developer!"

With that done, tap Settings, About Phone, Developer Options, and then tick USB debugging." Then tap OK to approve the setting change.

Step 3: Run Android Root on your PC, then connect your phone via its USB sync cable. After a moment, the former should show a connection to the latter. Your device screen may show an "Allow USB debugging?" pop-up. Tick "Always allow from this computer," then tap OK.



Enlarge Image

Screenshot by Rick Broida/CNET

Step 4: Click Root, then sit back and wait while the utility does its thing. After a few minutes, my Galaxy S6 got to 70 percent, and then the phone once again crashed and rebooted. Again, your mileage can (and most likely will) vary.

And that's all there is to it. If you decide you want to reverse the process, just run Android Root again, connect your phone, then click Remove Root. (Same goes for the app version, more or less.)

Now, what should you do with your rooted phone? Hit the comments to share your favorite options.

https://www.cnet.com/tech/services-and-software/how-to-easily-root-an-android-device/

Android rooting is the ideal way to get more control over your smartphone, opening up a world of unknown possibilities, but it's important to approach it with caution. Rooting isn't without its risks — and if something goes wrong, it can void your warranty, leave you with a broken smartphone or tablet, or worse.

CONTENTS

Before proceeding, it is important to understand that rooting is not always a straightforward process, and you may encounter hiccups along the way. If you decide that you absolutely need to root your Android device, continue below, but know that it isn't for the faint of heart or technology-inexperienced.

Manufacturers and carriers will dissuade you from rooting, and they aren't just scaremongering. If you don't follow instructions properly, the worst-case scenario can irreparably damage your device, but many people feel that the potential benefits are well worth it. With a rooted phone, you can remove bloatware, speed up your processor, and customize every element of your phone software's appearance.

This guide on how to root Android phones will walk you through the steps to root your device. While we can root some phones in minutes, others are going to take a little more research. But one thing is clear: rooting your phone is one of the best ways to tap into your Android device's true potential.

See more

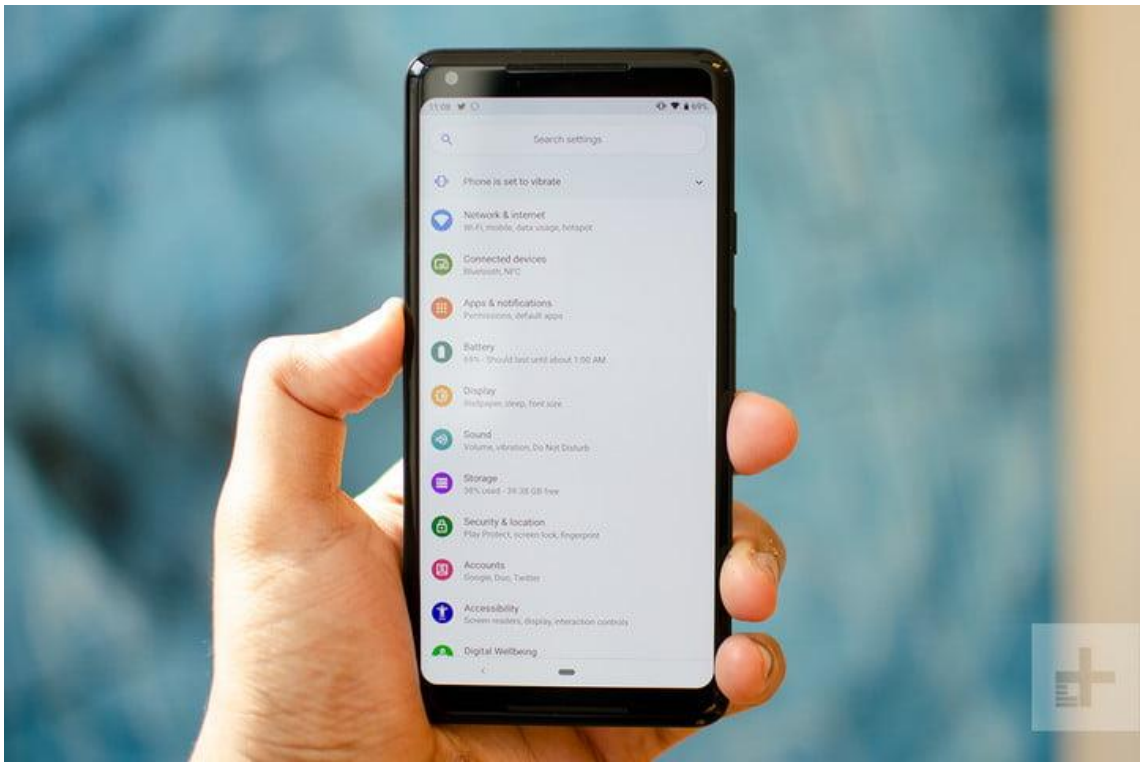
- Best root apps for rooted Android phones and tablets
- Best Android Phones
- Best Android 10 tips and tricks

What is rooting?

Rooting an Android phone or tablet is akin to jailbreaking an iPhone — basically, it allows you to dive deeper into a phone's sub-system. After rooting, you can access the entire operating system to customize just about anything on your Android device, and you can get around any restrictions that your manufacturer or carrier may have applied.

Rooting is best undertaken with caution. You must back up your phone's software before installing — or "flash," in rooting terms — a custom ROM (a modified version of Android).

Why would you root?



Julian Chokkattu/Digital Trends

One of the biggest incentives to root your Android phone is so you can strip away bloatware that's impossible to uninstall otherwise (although you can sometimes disable it — check out our guide on disabling bloatware). On some devices, rooting will enable previously disabled settings, like wireless tethering. Additional benefits include the ability to install specialized tools and flash custom ROMs, each of which can add extra features and improve your phone or tablet's performance.

There isn't an overabundance of must-have root apps, but there are enough to make it worthwhile. Some apps will let you automatically back up all of your apps and data to the cloud, block web and in-app advertisements, create secure tunnels to the internet, overclock your processor, or make your device a wireless hot spot. Look at the best root apps for rooted devices for a better idea of what is possible.

Why wouldn't you root?

There are essentially four potential cons to rooting your Android.

- **Voiding your warranty:** Some manufacturers or carriers will void your warranty if you root your device, so it is worth keeping in mind that you can always unroot. If you need to send the device back for repair, simply flash the software backup you made, and it'll be good as new.
- **Bricking your phone:** If something goes wrong during the rooting process, you risk bricking — i.e., corrupting — your device. The easiest way to prevent that from happening is to follow the instructions carefully. Ensure the guide you are following is up to date and that the custom ROM you flash is specifically for your phone. If you do your research, you won't have to worry about bricking your smartphone.

- **Security risks:** Rooting introduces some security risks. Depending on what services or apps you use on your device, it could create a security vulnerability. And certain malware takes advantage of rooted status to steal data, install additional malware, or target other devices with harmful web traffic.
- **Disabled apps:** Some security-conscious apps and services do not work on rooted devices — financial platforms like Google Pay and Barclays Mobile Banking do not support them. Apps that serve copyrighted TV shows and movies, like Sky Go and Virgin TV Anywhere, will not start on rooted devices, either — and neither will Netflix.

How to prepare your Android device for rooting

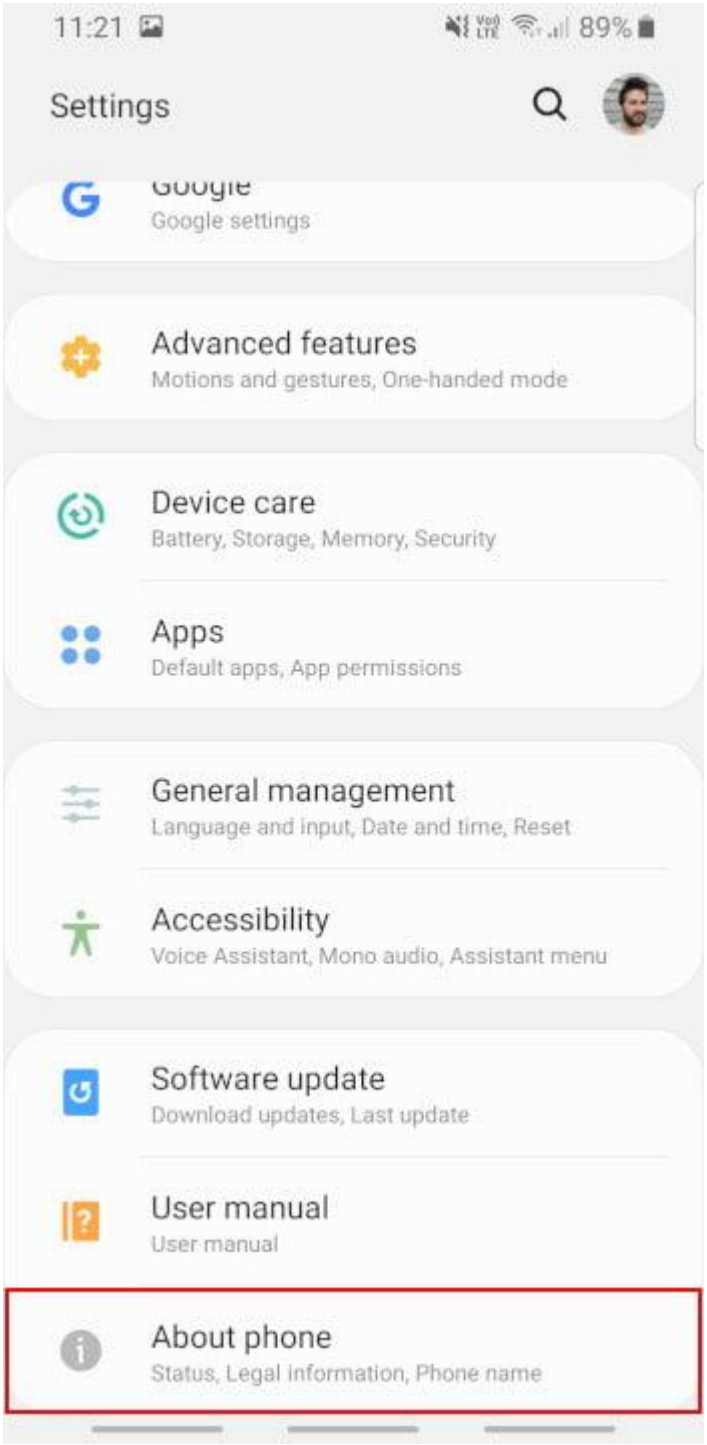
One of the easiest ways to root an Android device is by using an app, and several rooting apps have garnered attention over the years — Framaroot, Firmware.mobi, Kingo Root, BaiduRoot, One Click Root, SuperSU, and Root Master are among the most reliable. These services will usually root your device in the time you take to brush your teeth. But some of them only support devices running older versions of Android, so you may need to do some shopping around to find one that works for your device. If you're looking to root an even older device, you may need to check Firmware.mobi.

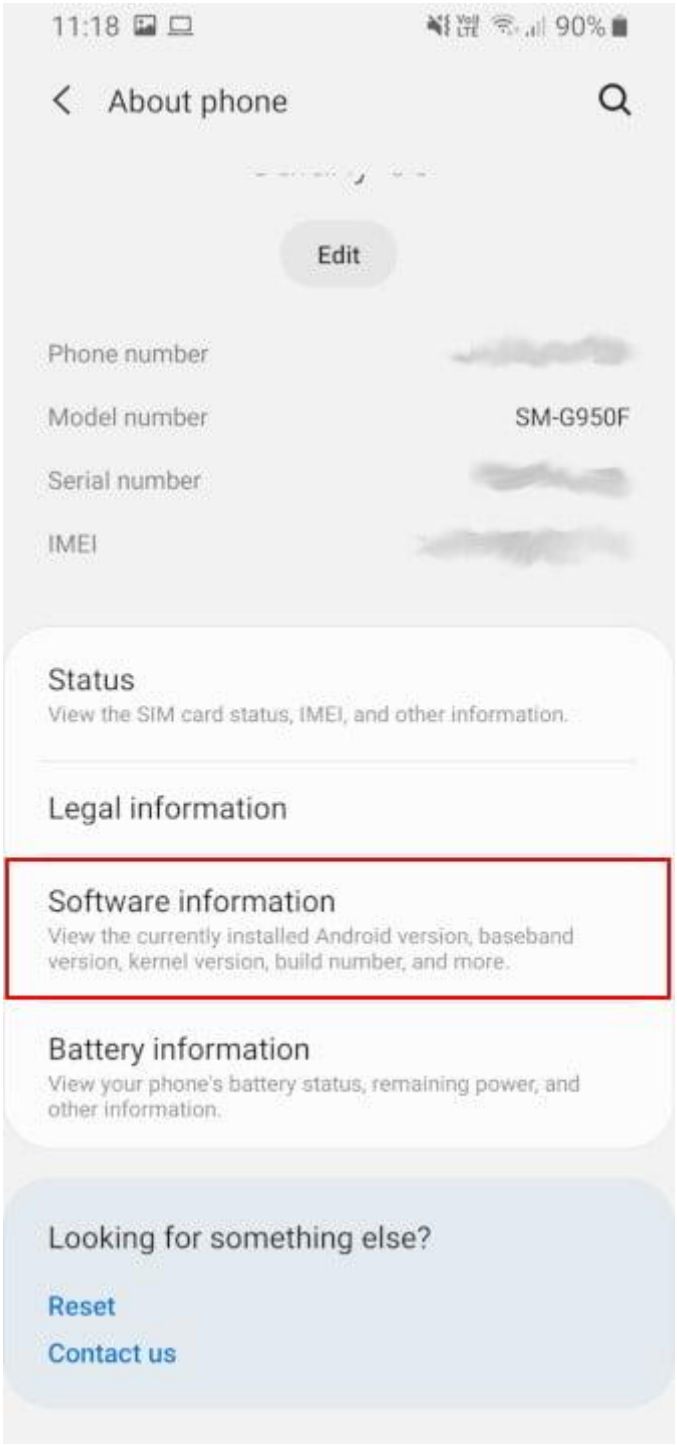
It used to be that rooting Android versions from Android 7.0 Nougat upwards was more difficult. The verified boot service will check the device's cryptographic integrity to detect if your device's system files are compromised, inhibiting legitimate rooting apps. Thankfully, rooting apps have caught up with the curve, and rooting newer Android versions is much easier than it used to be.

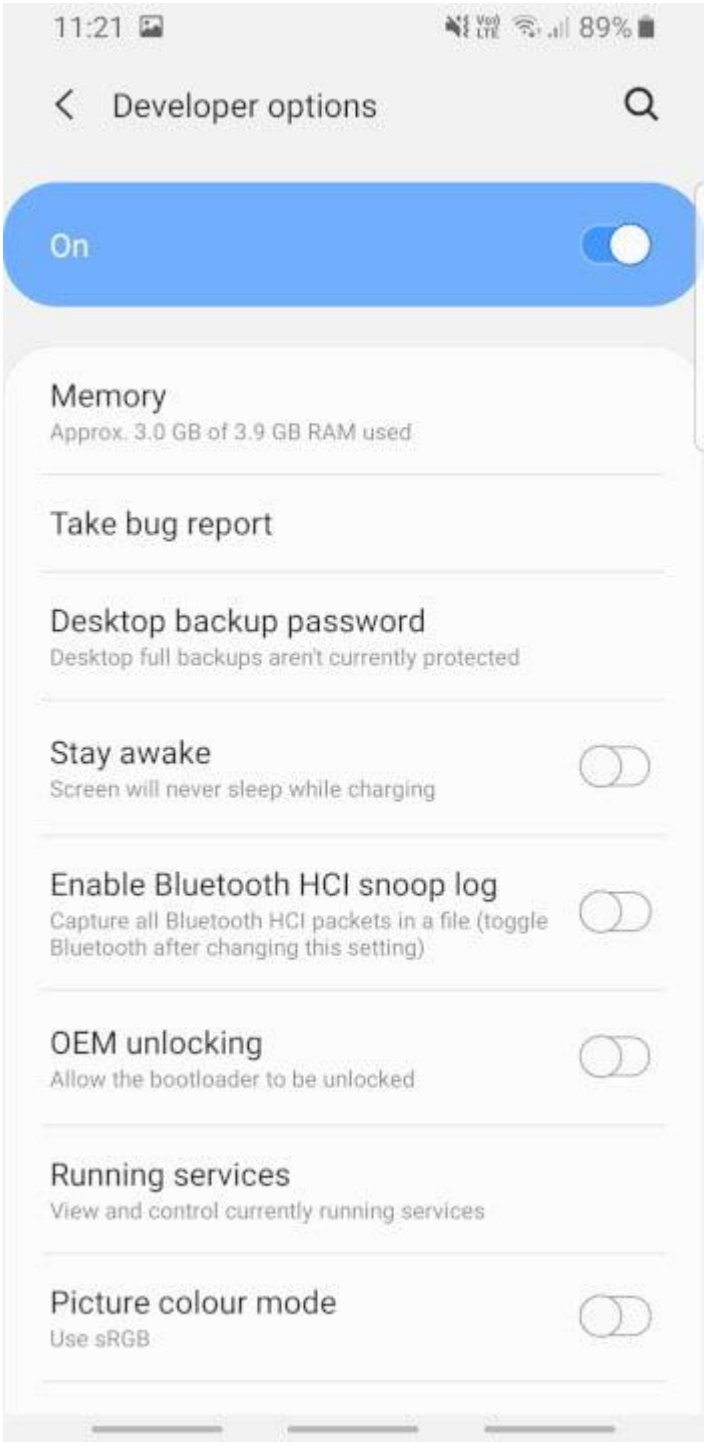
If your phone isn't compatible with a one-click rooting app, you must spend a little time researching alternatives on Android forums. The best place to start is XDA Developers Forum — look for a thread about your phone or tablet, and you're likely to find a method.

Preparing for rooting

Back up everything you cannot live without before you start. You should also always back up your phone's current ROM before you flash a new one. You will also want to ensure that your device has a full charge before you begin.







You will need to turn on *USB Debugging* and *OEM Unlocking*. Do this by opening *Settings* on your device. If you do not see *Developer Options* toward the bottom of the *Settings* screen, follow these steps to activate it.

1. Tap on *About Phone* and find the *Build Number*. The exact path depends on your phone, but it'll usually be found with other software information.
2. Tap on the *Build Number* seven times, and the *Developer Options* will appear on the Settings main page. You may need to confirm your security passcode to enable this.
3. Tap on the *Back* key to see your new developer options.

4. Tap *Developer Options*.
5. Check to enable *USB Debugging.*
6. Check to enable *OEM Unlocking*.

Installing the Android SDK Platform Tools

Rooting used to involve downloading Google's entire Android development kit. Thankfully, that's not the case anymore, and all you need is the Android SDK Platform Tools.

Download and install the Android SDK Platform Tools from Google's developer site. There are choices for Windows, Mac, and Linux. These instructions are for Windows machines. Extract the zipped files. When asked what directory to install the software to, we recommend setting it to *C:android-sdk.* If you choose a different location, make sure you remember it.

Installing device drivers

To ensure your computer can properly communicate with your smartphone or tablet, you will need to install the appropriate USB driver.

Devices from some manufacturers come with the drivers included in the phone's software, so all you need to do to install the appropriate USB driver is attach your phone to your PC by USB cable. OnePlus is an example of this, but it's worth connecting your phone first to see whether USB drivers will automatically install.

Otherwise, here is a list of drivers from the most popular manufacturers:

- Asus
- Acer
- Alcatel
- Coolpad
- Google/Nexus/Pixel
- HTC
- Huawei/Honor
- Lenovo/Motorola
- LG
- Samsung
- Sony
- Xiaomi

Follow the installer's instructions. Once the drivers are installed, proceed to the next step.

Unlock your bootloader

Before you get started, you need to unlock your device's bootloader. The bootloader, simply put, is the program that loads the device's operating system. It determines which applications run during your phone or tablet's startup process.

Some manufacturers require you to get a key to unlock the bootloader. Motorola, HTC, LG, and Sony provide step-by-step instructions on how to do so, but a word of warning: They require you to register for a developer account.

Unfortunately for users of Huawei and Honor devices, those phones' bootloaders can no longer be unlocked. Huawei rescinded the ability to request unlock codes in July 2018. If you still want to root your Huawei or Honor device, you must use a third-party service like DC-Unlocker.

Once you have taken those steps, you can embark on the unlocking process. You will need to put your device in *fastboot* mode. It's different for every phone, but on most devices, rebooting the device and holding down the *Power* and *Volume Down* buttons for 10 seconds does the trick (HTC phones require that you hit the *Volume Down* key and press the *Power* button to select it).

Once you have booted into *fastboot*, head to the folder you previously unzipped your Android SDK files to. Then, open your computer's command prompt by holding down *Shift + Right Click* and choosing *Open a Command Prompt Here*. If your device requires a code, you will get a long string of characters. Paste it into the box on your device manufacturer's website, submit the form, and await an email with a key, a file, and further instructions.

Unlock your device's bootloader by connecting it to your computer and placing it in *fastboot* mode again. Pull up the command prompt by typing *cmd* into your *Start* menu.

For Google Nexus and Pixel devices, the commands are easy:

- Nexus phones: Type "fastboot oem unlock" (without quotes) and hit *Enter.*
- Pixel phones: Type "fastboot flashing unlock" (without quotes) and hit *Enter.*

It's the same for Samsung devices:

- Samsung phones: Type "fastboot flashing unlock" (without quotes) and hit *Enter*.

Motorola's command is a little different:

- Type "oem unlock UNIQUE_KEY" (without quotes), replacing "UNIQUE KEY" with the code you received

So is HTC's:

- Type "unlocktoken Unlock_code.bin" (without quotes), replacing "Unlock_code.bin" with the file you received.

Confirm the unlock, and you're one step closer to rooting your Android device.

Some manufacturers and carriers don't sanction bootloader unlocking, but that doesn't mean it can't be done. Try searching the XDA Developers forum for workarounds and unofficial solutions.

How to root your Android device with multiple programs

There are a lot of different ways to root your phone or tablet. Here are a few of our favorites.

Rooting with Root Master

The XDA Developers forum mentions Root Master as one of the best one-click root methods — and it's simple for beginners to use. It's worth noting, though, that Root Master hasn't updated since 2017. If you decide to use this app, here are the steps to follow:

1. Download and install the APK. You may need to tap the *Unknown Sources* button in your Android *Security* settings to complete the installation.
2. Launch the app, then tap *Start.*
3. The app will let you know if your device is compatible. If it isn't, you must try one of the other apps on our list.
4. If you can root your device, proceed to the next step, and the app will begin rooting. This may take some time, and it's a good idea to try not to use your phone for anything else while it's running.
5. Once you see the *Success* screen, restart your device, and you're done!
6. Once it's finished, you can download and run Magisk to manage your root access.

Rooting with Framaroot

Framaroot is a newer one-click rooting service, and it aims to make the process of rooting easy for everyone with a simple one-click “root” button. However, you might have to jump through a couple of hoops to get it started.

1. Download the APK.
2. Install it — you may need to tap the *Unknown Sources* button in your Android *Security* settings to complete the installation.
3. Open the app, and tap *Root*.
4. If it can root your device, you can root your device.
5. You then must download and run Magisk to manage your root access.

Then that’s it — you’re ready!

Rooting your Android device with Firmware.mobi

Firmware.mobi, an unlocking utility by developer Chainfire, isn't the easiest way to root your Android smartphone, but it is one of the most stable. It works on over 300 devices and provides step-by-step instructions that make the rooting process as seamless as it could be.

You will need to download the ZIP file intended for your device.

Once you have done that, follow these steps:

1. Extract the folder.
2. Navigate to it and find the *root-windows.bat* file. Double-click it.
3. Wait for the script to execute, and press any key.
4. When the process is complete, your phone will automatically reboot, and it will root you.

Rooting your Android device with BaiduRoot

BaiduRoot, a software utility by Beijing-based Baidu Inc., supports over 6,000 Android devices. Still, since those only include devices running Android 2.2 up to Android 4.4, it's going to have limited use for most. However, if you've got an ancient phone lying around, this is a great tool for rooting and repurposing that. It's coded in Chinese, but a crafty translator has released an English version.

BaiduRoot is one of the more straightforward rooting applications. Once you've downloaded it on your computer, it's a step-by-step affair.

First, you must unzip the file. Find *Baidu_Root.RAR* and extract its contents (if you're using Windows, you might need a third-party application like 7-Zip).

Next, attach the device you want to root to your computer via USB and transfer the files. Once that's done, unplug your phone.

You must install the BaiduRoot application manually. Follow these steps:

1. On your smartphone or tablet, head to *Settings > Security* (or *Lock Screen and Security*).
2. Toggle *Unknown Sources*, and press *OK* on the pop-up.
3. Find the folder containing the BaiduRoot app and tap the *APK* file. Follow the instructions to complete the installation.

Now, switch to BaiduRoot:

1. Open BaiduRoot and accept the license agreement.
2. Tap the *Root* button in the center of the screen.
3. After a few seconds, you'll get a message showing that the device successfully rooted.

Here's a video showing the installation process.

Rooting with One Click Root

One Click Root is a new rooting tool that aims to reduce the complicated nature out of rooting. The idea of One Click Root is right there in the name: One click, and you're done. It charges $40 to root your phone and promises that the program won't be able to brick your phone,

except with user negligence. We can't back up those claims, so we recommend taking all the same precautions you would take with any other rooting app.

The One Click Root procedure is simple:

1. Check that your device is supported by the Root Availability Tool.
2. Download the Windows/Mac One Click Root program.
3. Connect your device via USB cable.
4. Enable USB debugging on your device.
5. Run One Click Root and let the software handle the tricky bit.

How to use Kingo Android Root

Kingo Root can install on a Windows-based computer or directly to the device you want to root. First, check to see if your device is compatible with Kingo by checking the official list. Then, grab the Kingo Android Root for Windows program, and install it. Alternatively, download the Kingo Android Root APK to your device, check the *Unknown Sources* box (see above), and install it.

If you've opted to use the Windows client, ensure to enable *USB Debugging* mode on your phone.

From there, usage is pretty simple:

1. Launch Kingo Root on your computer and connect your device via USB.
2. Kingo Root should detect your device automatically and prompt you to root it. Click *Root*, and then hang tight — Kingo will only take a few minutes to grant root privileges.

If you would rather root without a computer, follow these instructions:

1. Install the Kingo Root APK.

2. Open the Kingo Root app.
3. If your device is compatible, you will see a *One Click Root* button. Tap it and be patient — it can take a while.
4. If the root is successful, you will see a large checkmark.

Resources you will need after you root and how to unroot

Arguably no other mobile operating system parallels the diversity of Android OS. For this reason, there is no universal way to root your device. If the above options fail, do not fret. There is likely a guide on how to root your specific device available somewhere online — a few reliable resources include XDA Developers' forum and the Android Forums.

Once you have found the right guide for your phone or tablet, it's simply a matter of working through the listed steps methodically. It can get complicated, and it might take a while. But provided you follow the guide step by step, it should be a relatively pain-free process.

Download Root Checker

You will need to download an app to make sure your device has successfully rooted. There are several apps available on the Google Play store that, when downloaded, will tell you if you have super-user permission — a telltale sign you have succeeded. Root Checker is popular — simply installing and running it will tell you if your phone has super-user permissions.

Install a root management app

Rooting will make your phone more vulnerable to security threats. Installing a root management app will give you more peace of mind. Normally, every app that requires rooted privileges will ask for your approval. This is where root management apps, such as Magisk Manager, come in. Magisk Manager is open-source software that allows you to manage your phone's root permissions, granting or denying individual apps' permission.

How to unroot your Android device

For all the benefits you can gain from rooting, you can go back to the way things were. Whichever method you attempt, always make sure you back up your data before changing your phone.

Unroot with Universal Unroot

You can easily unroot your phone with Universal Unroot. It removes root privileges in most Android devices, but it's not perfect. Most Samsung devices are not compatible, and LG devices will be unrooted but still show as rooted after the app has worked its magic. It used to be a dollar, but it's now free since the developers are no longer updating it. But if you've got an older device that's supported, it's a good way to be sure.

Unroot by flashing stock firmware

One of the most thorough ways to remove root access is by flashing your device with factory firmware. This method will completely wipe your phone or tablet clean of any root traces, but a word of warning: It's not for the faint of heart.

First, download the factory image for your device to your computer. Once again, XDA is a great resource.

Next, unzip the file. You will see another zipped file — unzip that one, too. The unzipped folder should contain a bootloader image, radio, various scripts, and one more zipped file. Again, unzip that.

The next step involves installing ADB and Fastboot on your computer.

Download and install the Android SDK Platform Tools from Google's developer site. There are choices for Windows, Mac, and Linux. These instructions are for Windows machines. Extract the zipped files. When asked what directory to install the software to, we recommend setting it to *C:android-sdk.* If you choose a different location, make sure you remember it.

Make sure *OEM Unlocking* is enabled on your device. Open *Settings*. If you do not see *Developer Options* towards the bottom of the *Settings* screen on your device, follow these steps to activate them.

1. Tap on *About Phone* and find the *Build Number*.
2. Tap on the *Build Number* seven times, and the Developer Options will appear on the Settings main page.
3. Tap on the *Back* key to see the *Developer Options*.
4. Tap on *Developer Options*.
5. Check to enable *OEM Unlocking.*

Switch back to your computer. Copy *boot.img* in the folder you unzipped and place it in your *ADB* folder, *C:android-sdk*.

Connect your phone to your computer via USB. Open your computer's command prompt by holding down *Shift + Right Click* and choosing *Open a Command Prompt Here*. Then, enter these commands:

1. *adb reboot bootloader*
2. *fastboot flash boot boot.img*
3. *fastboot reboot*

Unroot with file explorer

If your phone is running Android Lollipop or older, you can also unroot by deleting the files that granted the root. We recommend using a file explorer app such as File Manager or Cx File Explorer. Once downloaded, you'll then need to turn on *Root Explorer* (or something similar) in the menu or settings of your file explorer app and grant root privileges if asked. Next, take the following steps, which may vary slightly (in terms of names used) depending on your file explorer:

1. Find your device's main drive under /.
2. Go to *System* > *Bin*, then tap and hold on *busybox* and *su* and delete them.
3. Now go to S*ystem* > X*bin*, then tap and hold on *busybox* and *su* and delete them.
4. Finally, go to *System* > *App* and delete *supeuser.apk*.
5. Restart the device, and you should be unrooted.

Unroot with OTA update

Sometimes just installing an OTA update will break root. Look for a software update under *Settings* > *About Device*. Just be careful — it might prove impossible to recover from. In that case, you may need to flash the original firmware first.

None of the root methods or unrooting methods are without risk, so always back up your data, make sure your device is fully charged, read the instructions carefully, and take your time. Again, if you need additional support, we recommend reaching out to the XDA community for more help. There, you will find an active community looking to help.

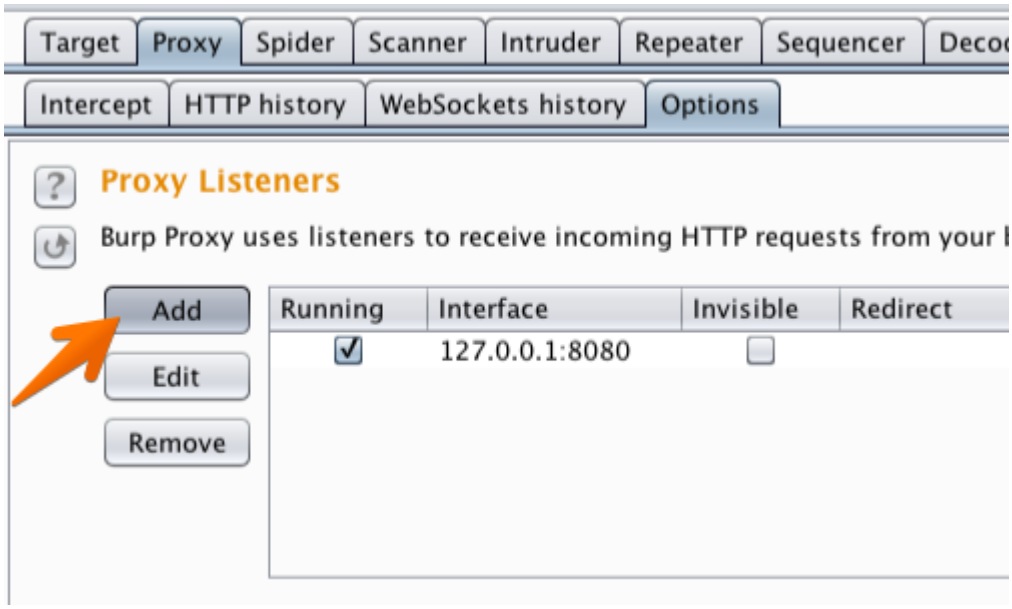
https://www.digitaltrends.com/mobile/how-to-root-android/

Burp Suite

Configuring an Android Device to Work With Burp

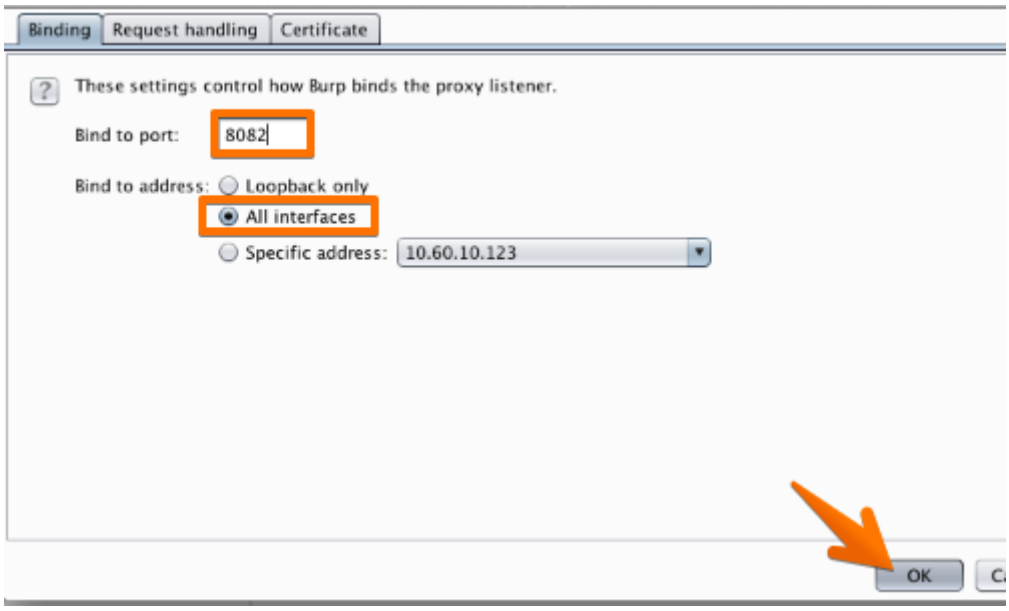
To test web applications using an Android device you need to configure your Burp Proxy listener to accept connections on all network interfaces, and then connect both your device and your computer to the same wireless network. If you do not have an existing wireless network that is suitable, you can set up an ad-hoc wireless network.

Configure the Burp Proxy listener



In Burp, go to the “Proxy” tab and then the “Options” tab.

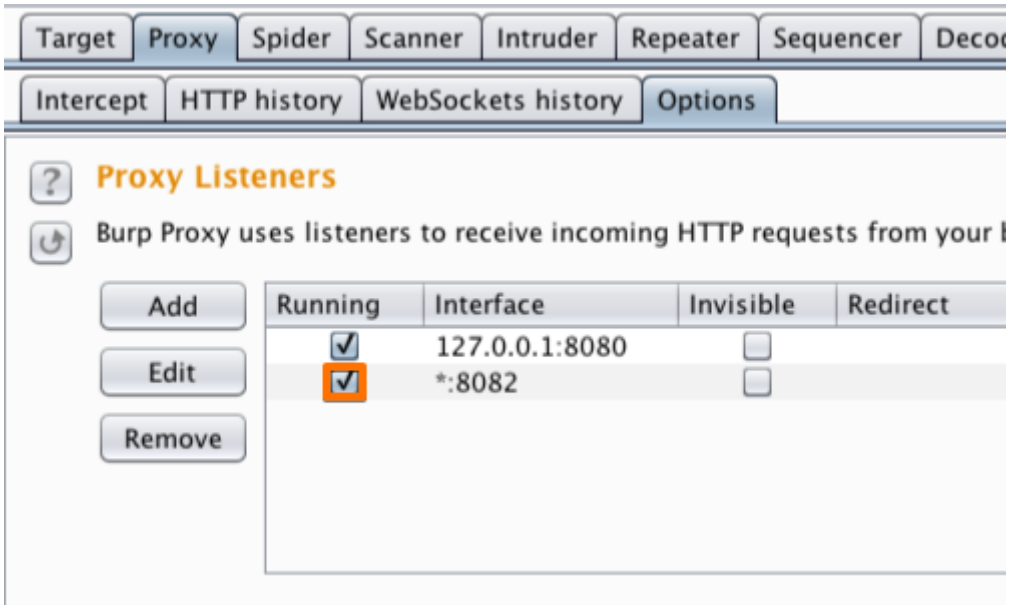
In the “Proxy Listeners" section, click the “Add” button.



In the "Binding" tab, in the “Bind to port:” box, enter a port number that is not currently in use, e.g. “8082”.

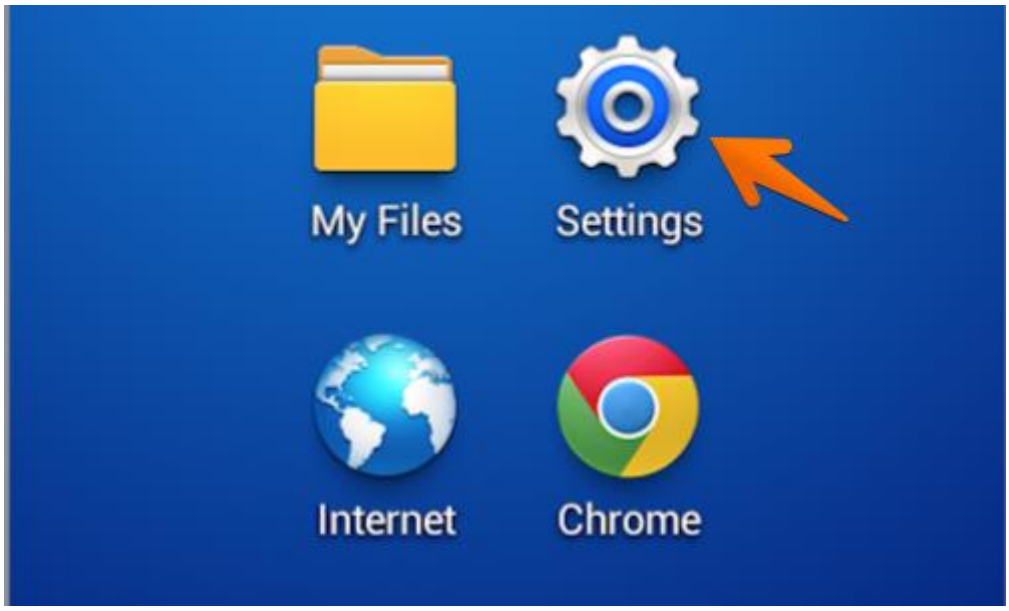
Then select the “All interfaces” option, and click "OK".

Note: You could alternatively edit the existing default proxy listener to listen on all interfaces. However, using different listeners for desktop and mobile devices enables you to filter these in the Proxy history view.

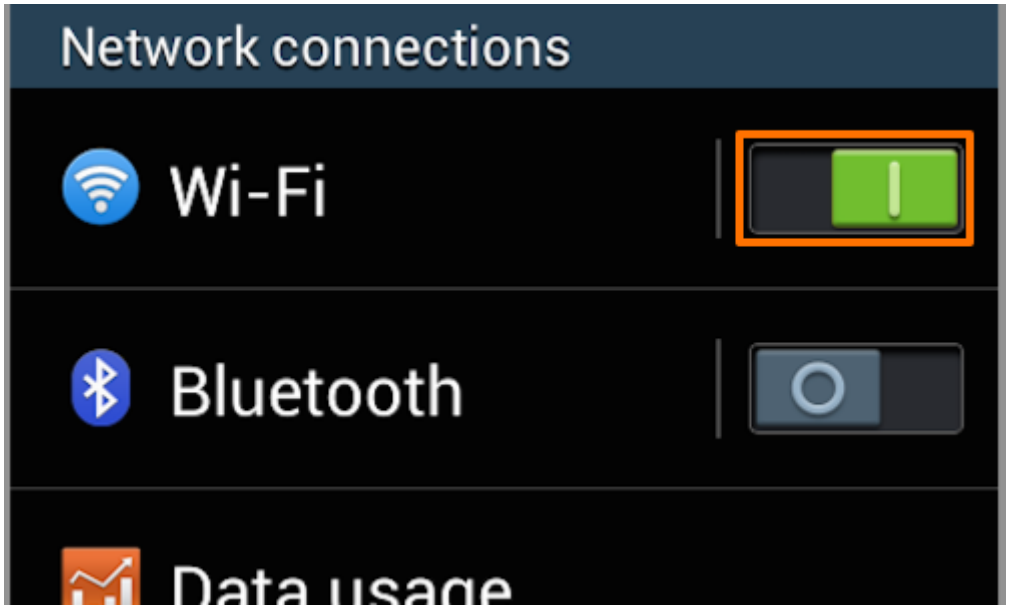


The Proxy listener should now be configured and running.

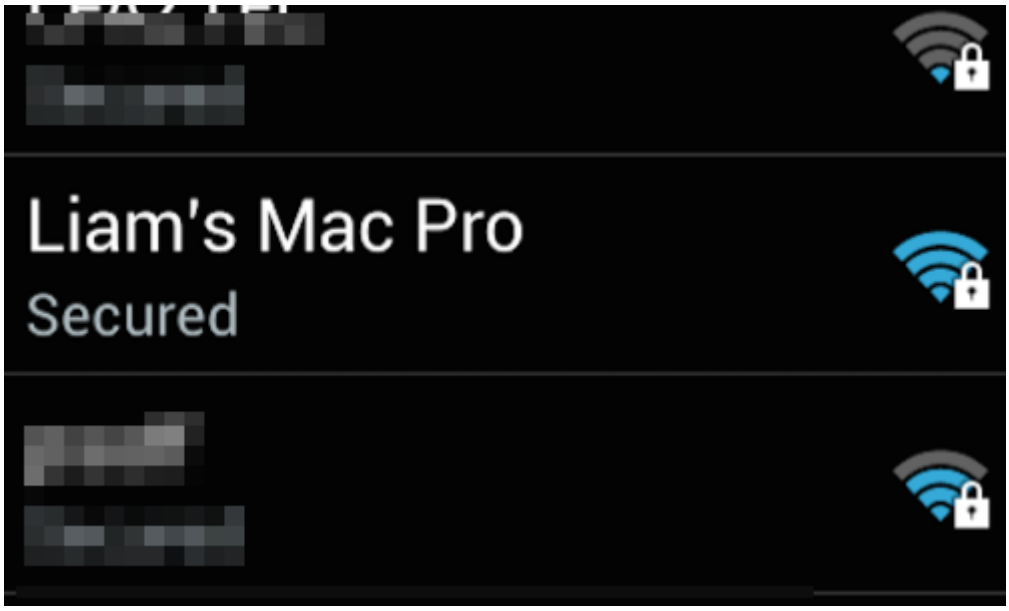
Configure your device to use the proxy



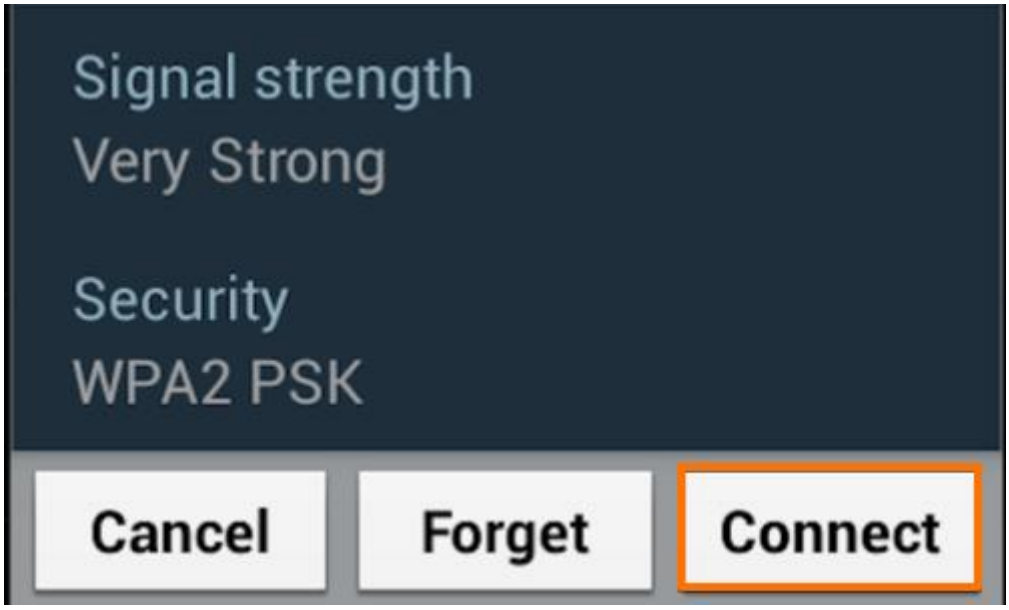
In your Android device, go to the“Settings” menu.



If your device is not already connected to the wireless network you are using, then switch the "Wi-Fi" button on, and tap the “Wi-Fi” button to access the "Wi-Fi" menu.

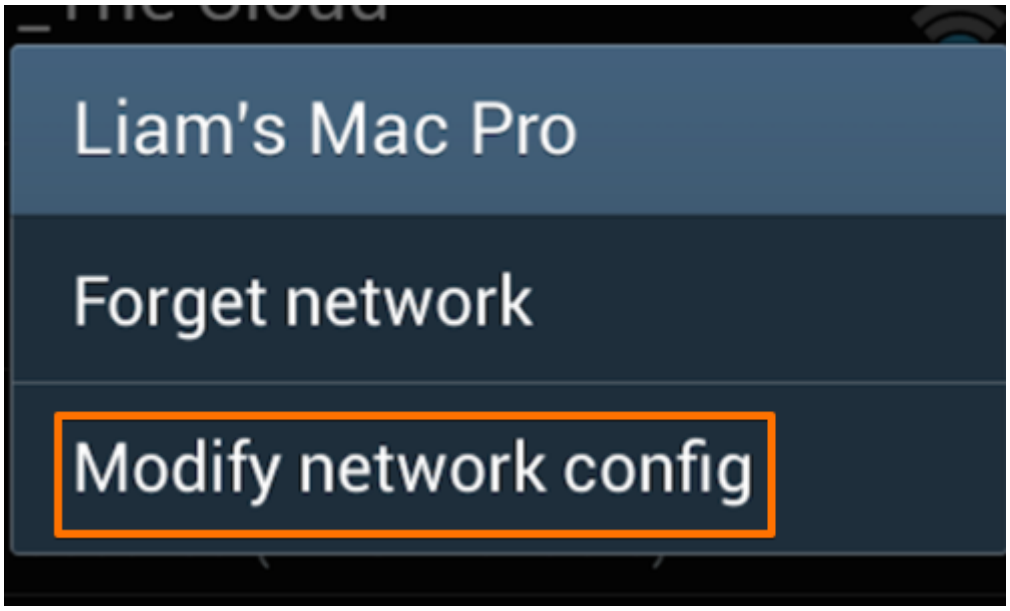


In the "Wi-Fi networks" table, find your network and tap it to bring up the connection menu.



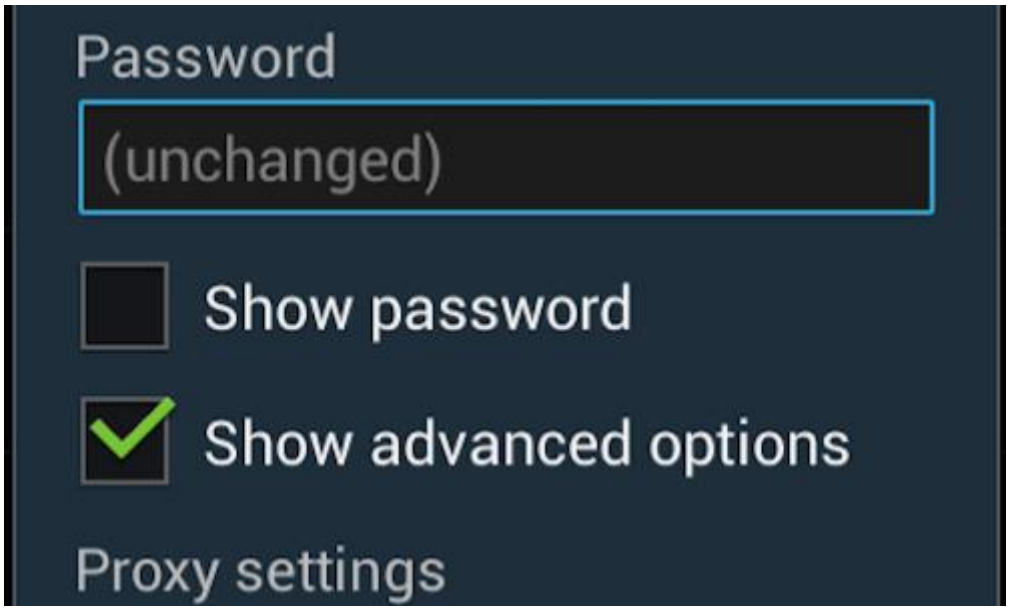
Tap "Connect".

If you have configured a password, enter it and continue.

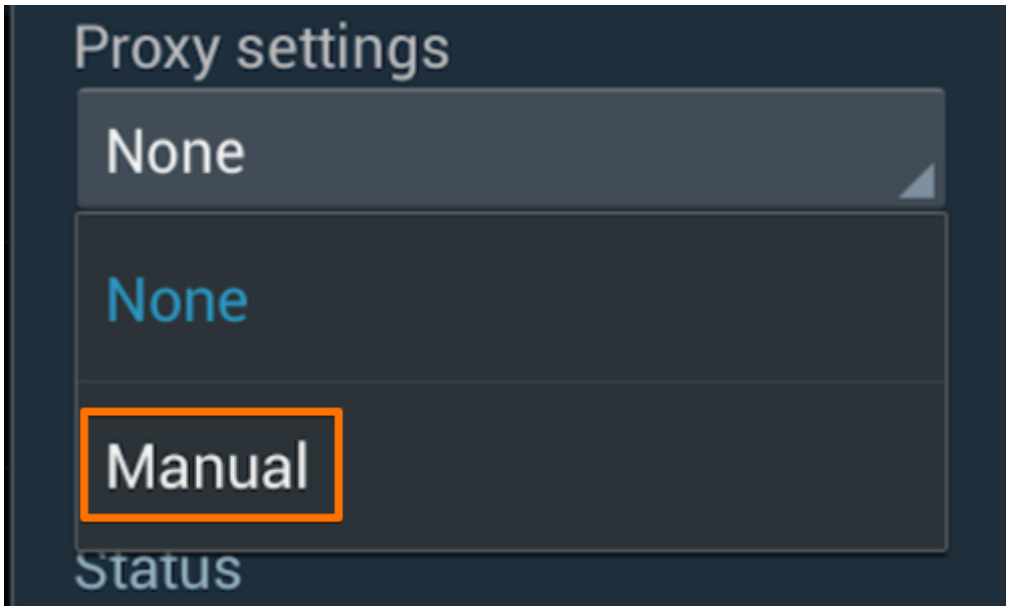


Once you are connected hold down on the network button to bring up the context menu.

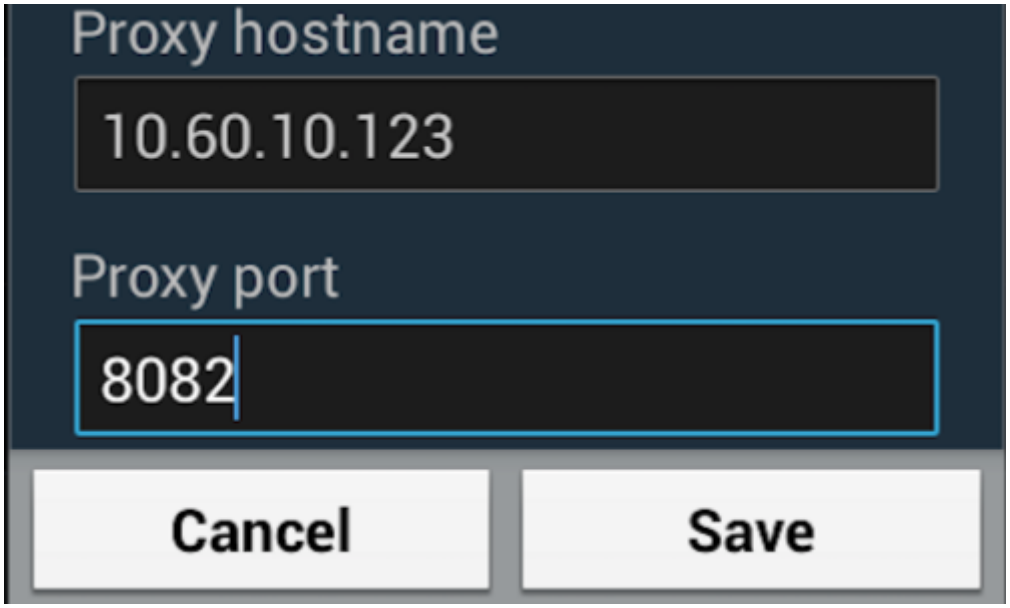
Tap “Modify network config”.



Ensure that the “Show advanced options” box is ticked.



Change the “Proxy settings” to “Manual” by tapping the button.

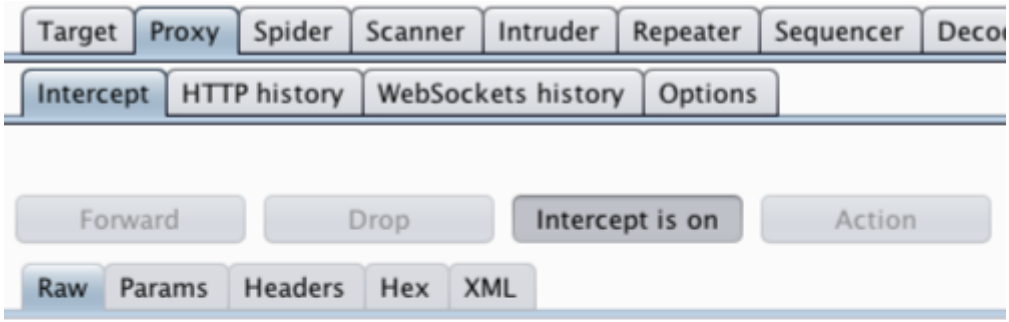


Then enter the IP of the computer running Burp into the “Proxy hostname”.

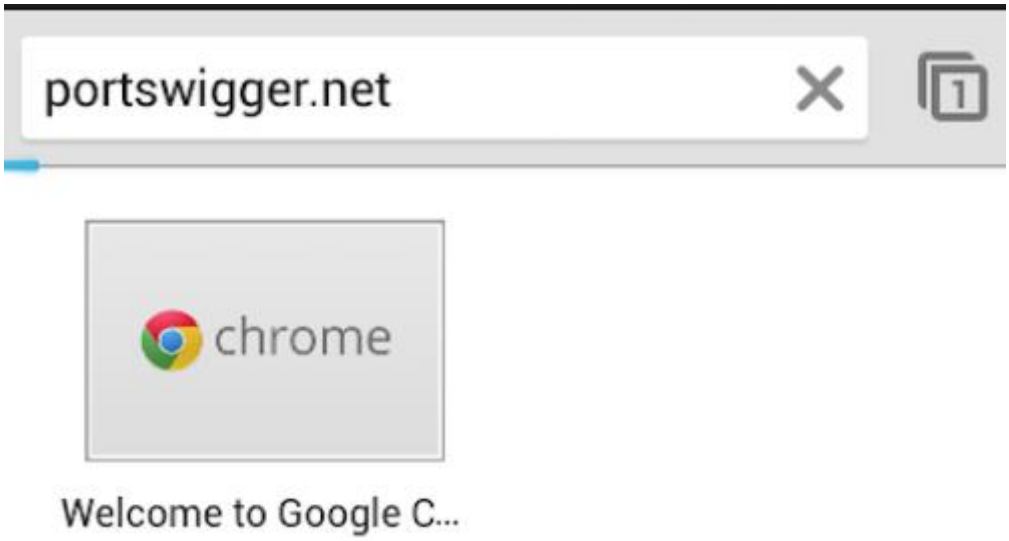
Enter the port number configured in the “Proxy Listeners” section earlier, in this example “8082”.

Tap "Save".

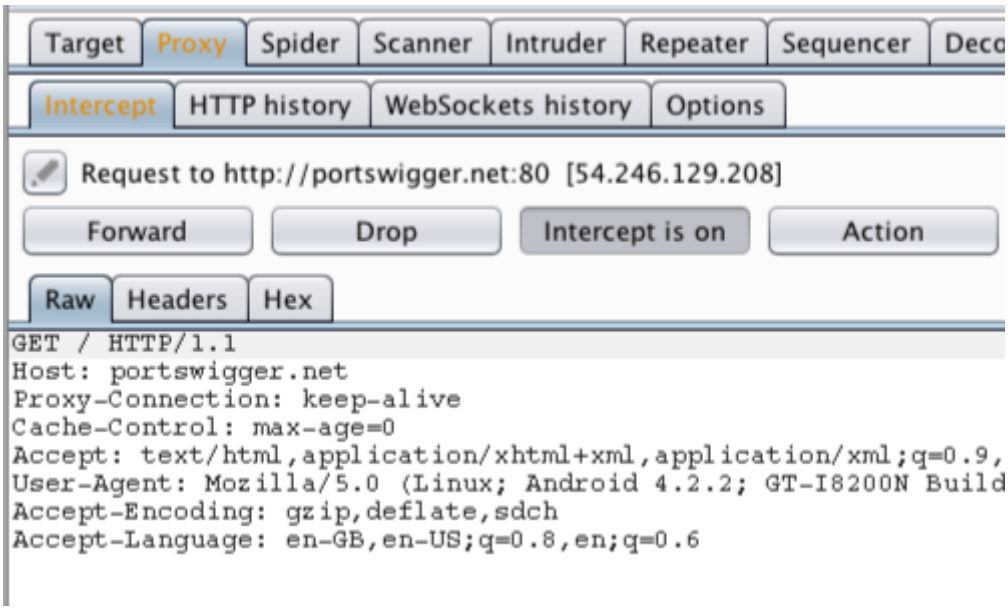
Test the configuration



In Burp, go to the "Proxy Intercept" tab, and ensure that intercept is “on” (if the button says “Intercept is off" then click it to toggle the interception status).



Open the browser on your Android device and go to an HTTP web page (you can visit an HTTPS web page when you have installed Burp's CA Certificate in your Android device.)



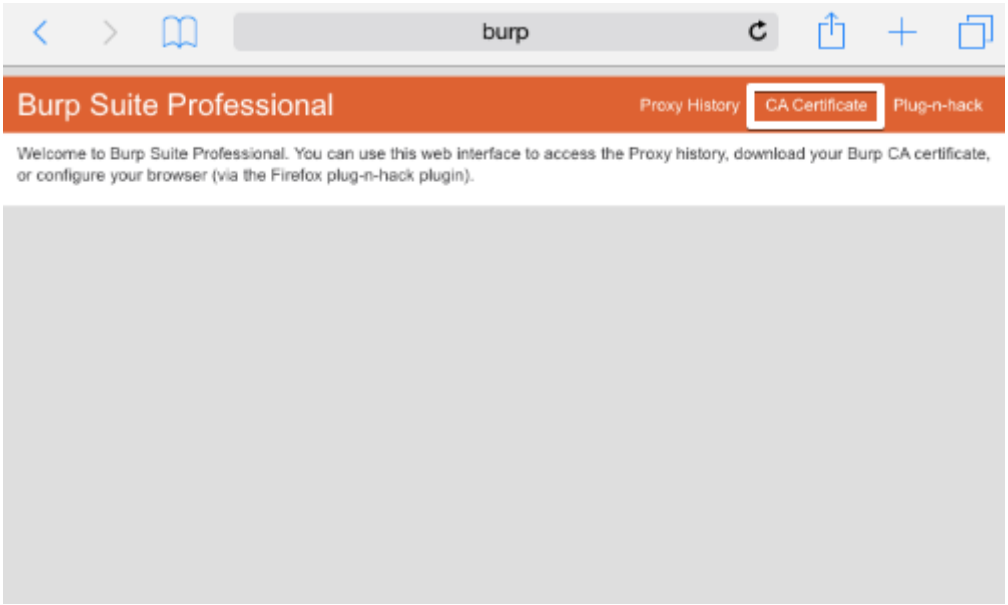
The request should be intercepted in Burp.

Installing Burp's CA Certificate in an Android Device

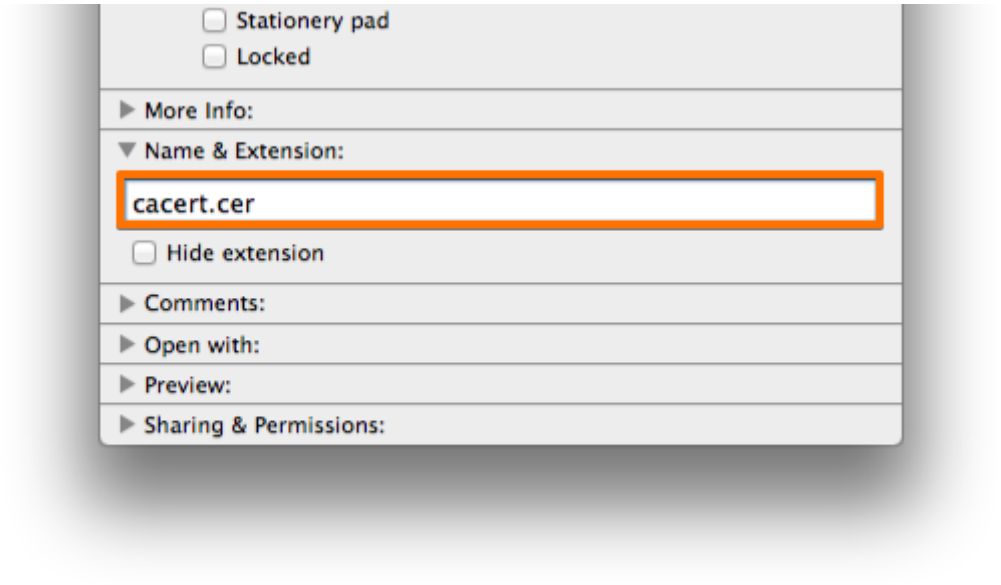
Before you start:

- Ensure you have configured your Android device to work with Burp.
- Ensure your Android device is able to receive email, and that your email filter does not block .cer files.

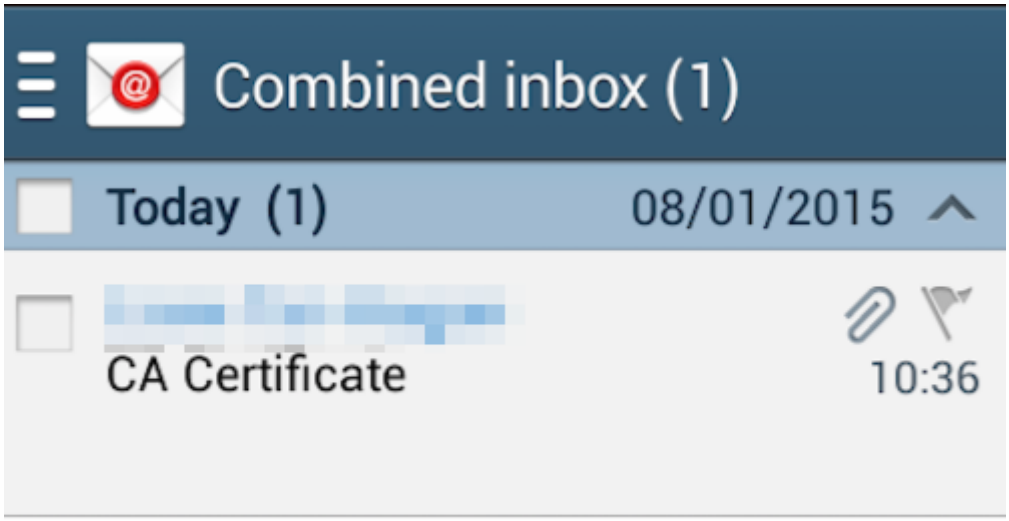
Note: Android Nougat no longer trusts user or admin supplied CA certificates. We recommend that you use an older version of Android for your testing. If you must use Android Nougat then you will need to install a trusted CA at the Android OS level on a rooted device or emulator.



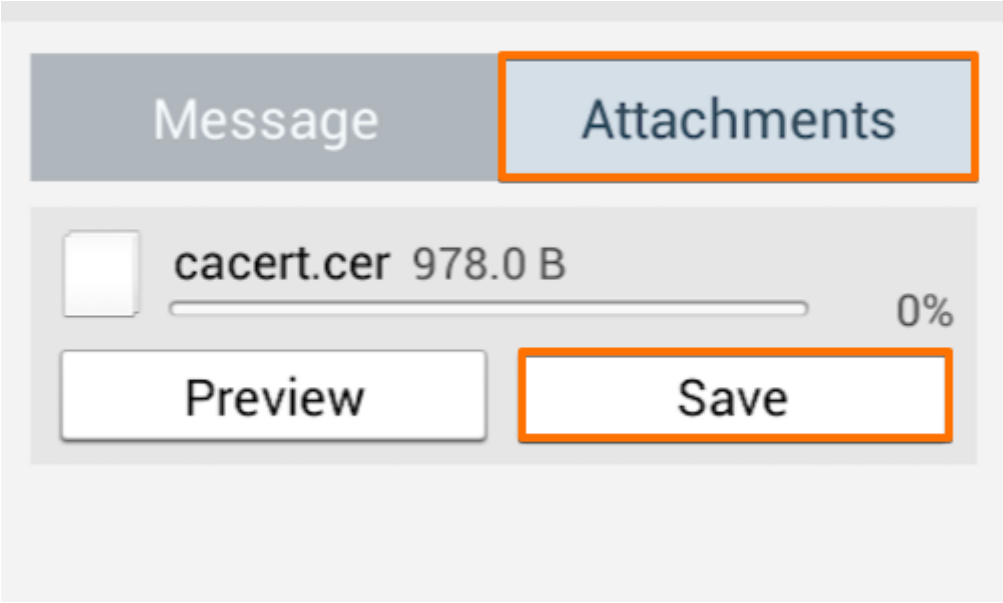
On your computer with Burp running, visit **http://burpsuite** and click the "CA Certificate" link. Save the certificate file on your computer.



On your computer, rename the file with the .cer file extension, and send the file as an email attachment to an account that you can access from your Android device.

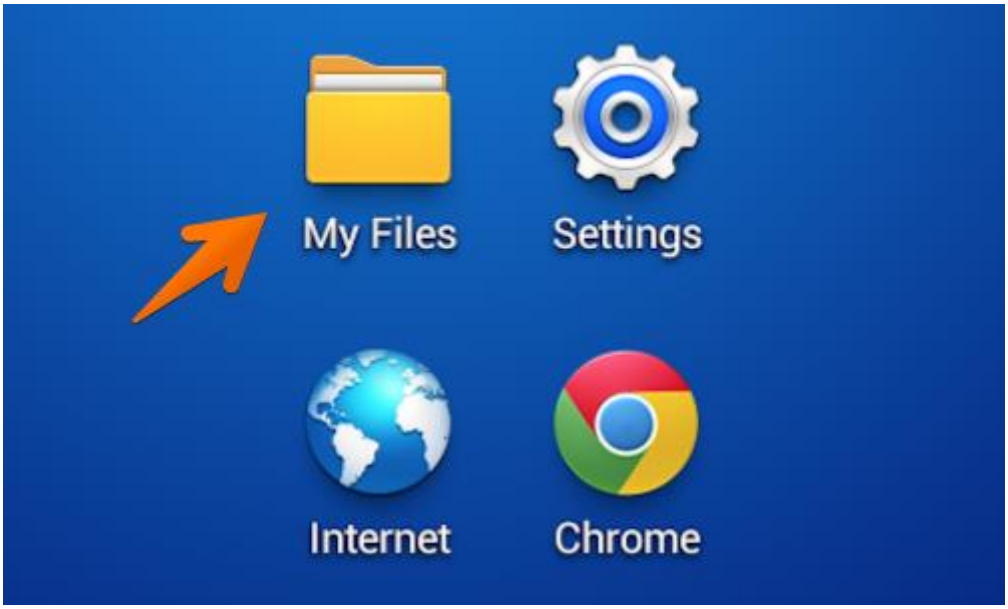


Check your email on the Android device.

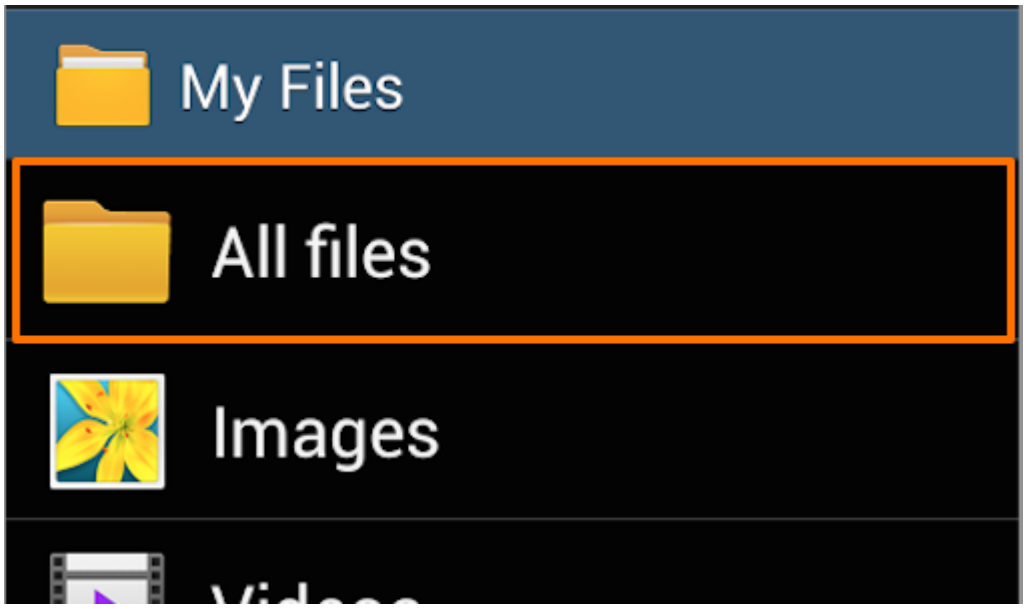


Open the email and tap the attachments button.

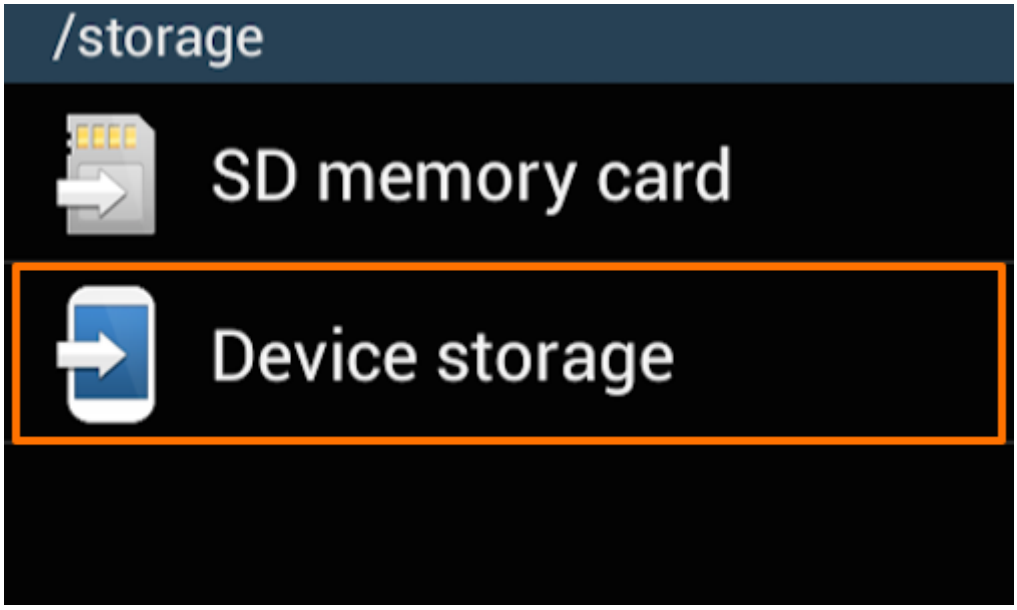
Then tap the save button. This should save the certificate file to your Android device's "Download" folder.



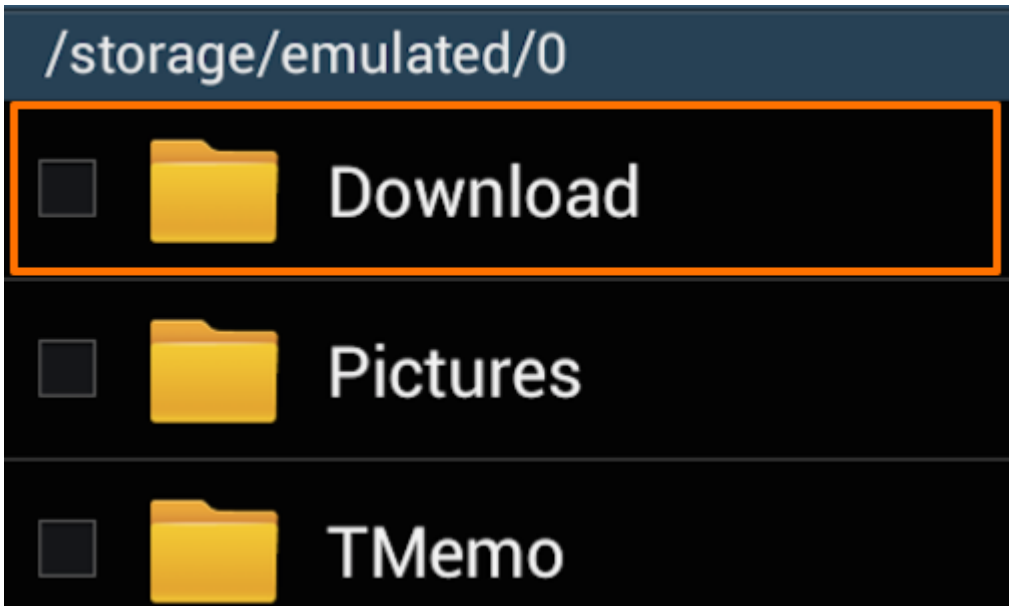
Find your "My Files" folder. This may be located in the "Apps" menu or on one of the device's home screens.



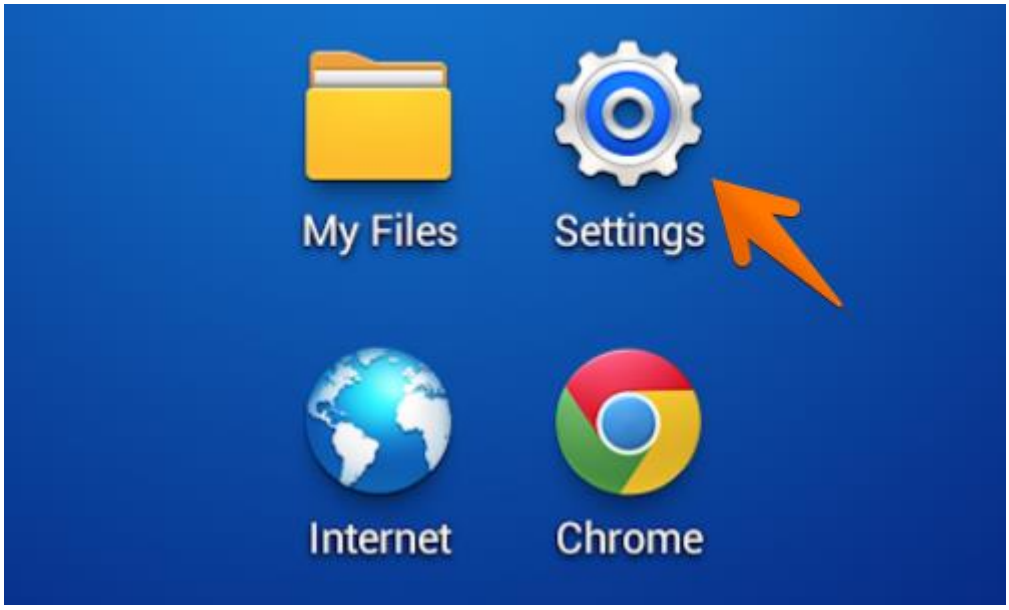
In “My Files” tap the “All Files” folder.



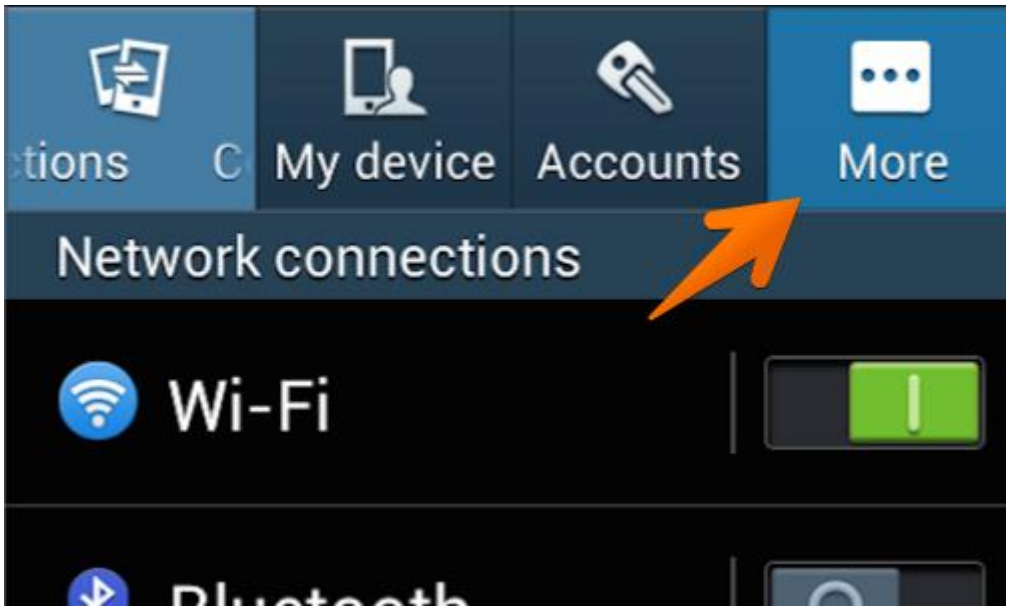
In the “All Files” folder tap “Device storage”.



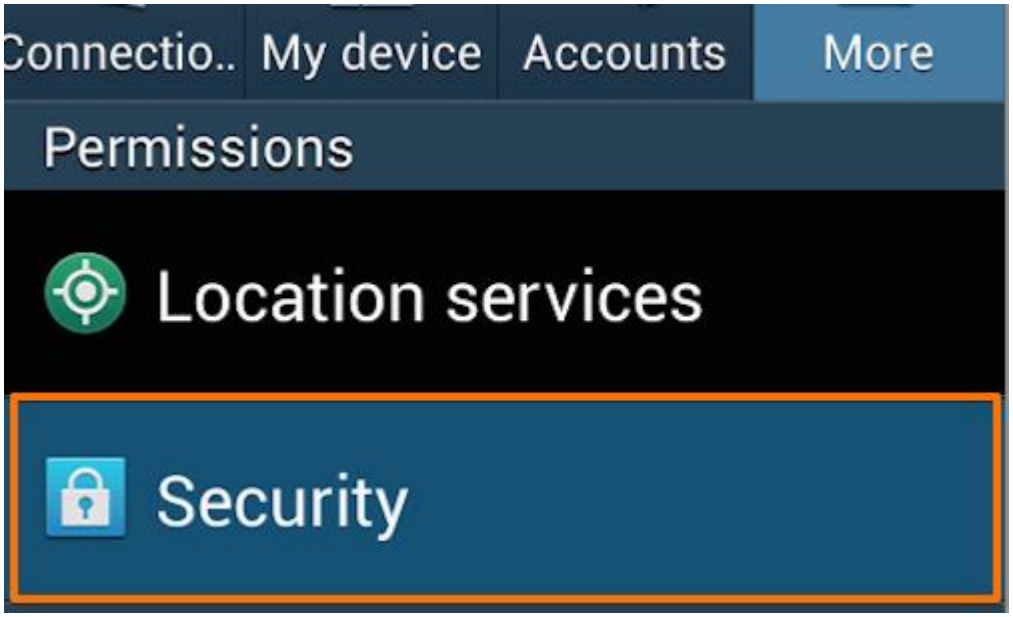
Open the “Download” folder and check that your certificate is correctly located in this folder.



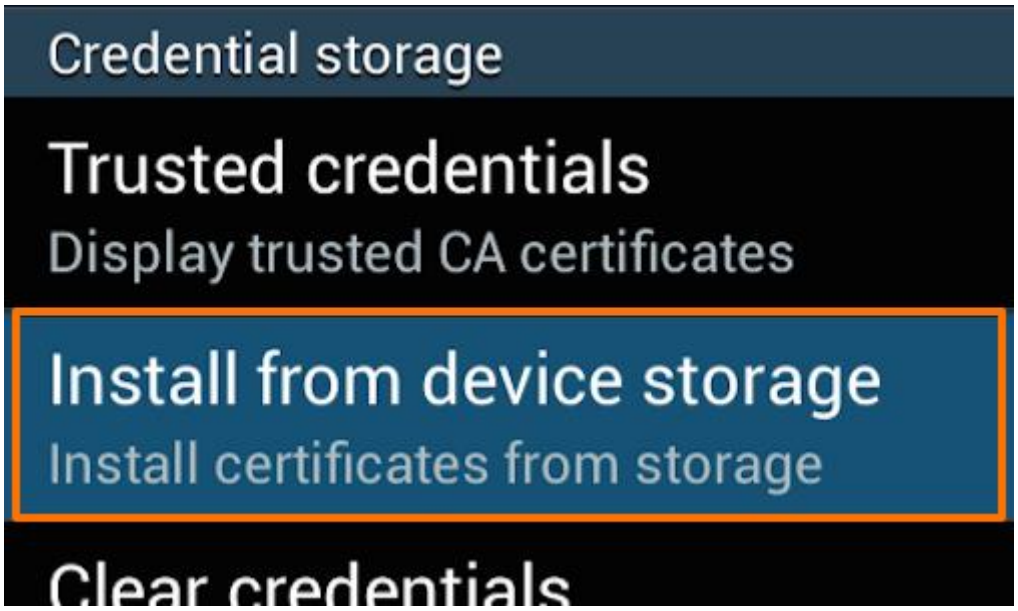
Next locate and tap the "Settings” icon. This may be located in the “Apps” menu or on one of the device's home screens.



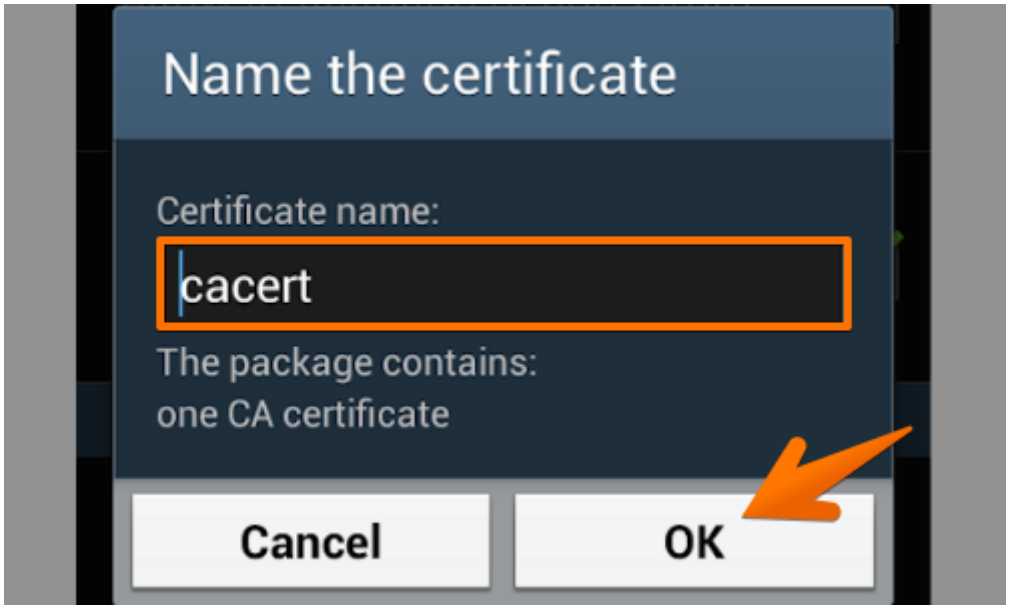
Tap the “More” button.



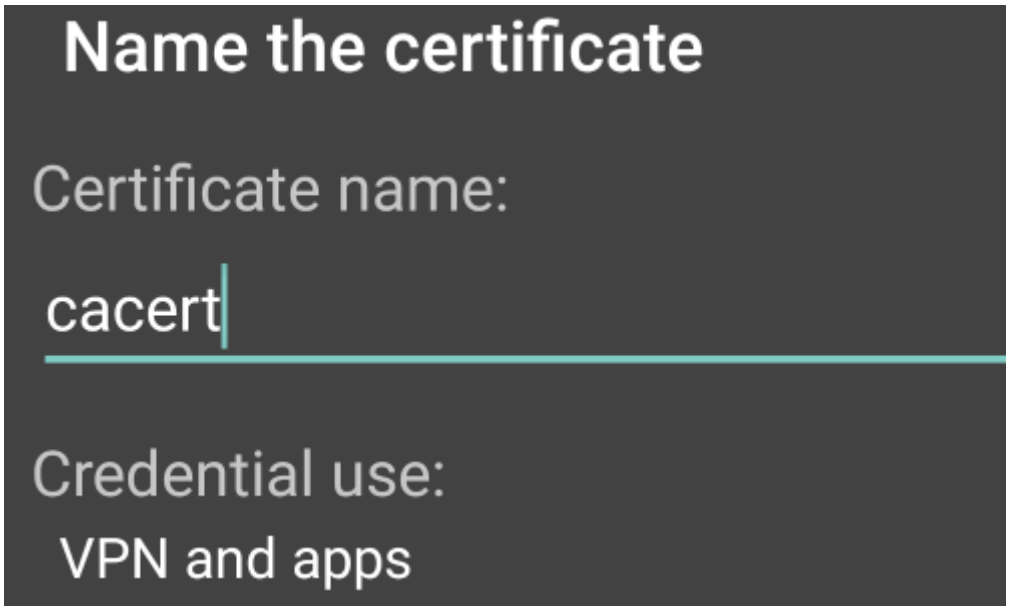
Beneath the “Permissions” header tap the “Security” button.



In the “Security” menu select the “Install from device storage” from beneath the "Credential storage" header.

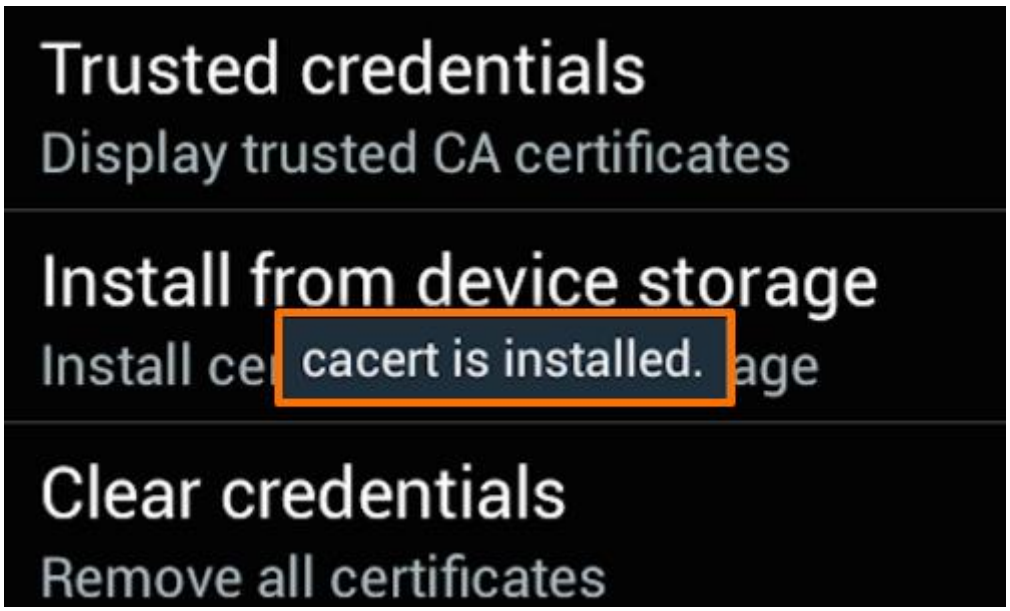


You will now be asked to “Name the certificate”, leave the certificate name as it is and tap “OK”.



In some versions of Android, your device will ask if you want to use the certificate for "VPN and apps" or "WiFi".

In the "Credential use:" options, you should select "VPN and apps".



The phone will revert to the security menu and will inform you via a small pop up that the certificate is installed.

You can check the Certificate is installed by tapping the “Trusted credentials" button.

Tap the "User" tab in the “Trusted credentials” window to show the PortSwigger CA certificate.

You should now be able to visit any HTTPS URL via Burp without any security warnings.

TapJacking

What is Tapjacking in Android and How to Prevent It

Android was built with high expectations, what with being based on the Linux OS model and all that. One would normally expect Android to be at least as secure as a typical Linux box, if not more. Alas, a secure Android remains a distant dream, even after the Marshmallow release. A case in point is an Android attack known as Tapjacking. A combination of “tap” and “jacking”, Tapjacking literally means someone hijacking what a user taps on his smartphone. It is one of the most vicious Android hacks known, as it doesn’t rely on any external tools or libraries, or even special permissions! To understand what Tapjacking is and how it works, let us begin with the basics.

Raise a toast

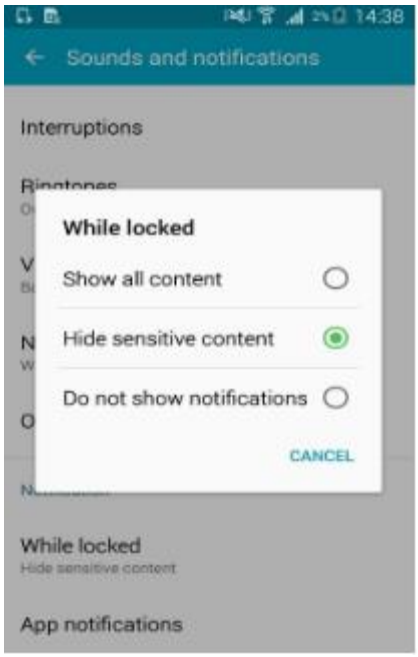
It all begins with the humble toast message. A cute little ephemeral thing that's gone from your screen by the time you notice it. Its typical use is providing non-critical notifications to the user. The user isn't expected to interact with it (because there's no way he can) and there's no way to make the toast message stay indefinitely.

Now let us put down the toast (no pun intended!) for a while and learn about another related concept: screen overlays.

The world of screen overlays

Screen overlays are those translucent layers of UI that you get sometimes. For instance, if you're using Android 6 and are launching an app for the first time, you will be asked to confirm-grant all its critical permissions. The dialog box that opens at that time, the one that causes the rest of the screen to gray out while still allowing you to see what's underneath, is a screen overlay.

Now, screen overlays are actually an incredibly cool feature. Remember the floating chat bubbles used by Facebook Messenger? Can you guess what makes them possible? Yes, it's screen overlays!

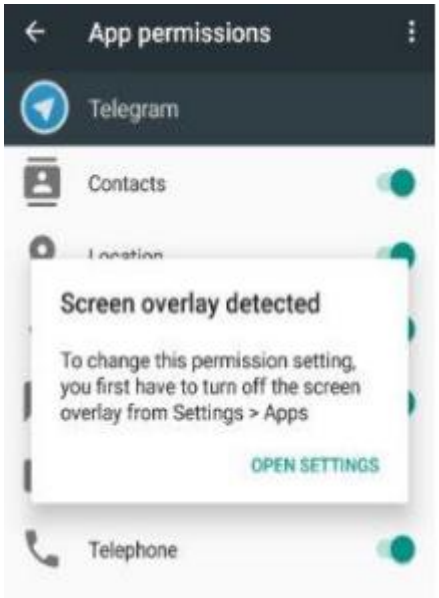


So, what's all this got to do with Tapjacking? Let's dive in.

How screen overlays result into Tapjacking

Now, here's the crux of the whole idea: when you're in the process of granting a critical permission to an app, there *should not* be any screen overlay active. We say "should not" because the actual implementation of this security idea is messed up pretty badly.

When active, this security feature will not allow you to interact with the underlying UI if there's an overlay active at that time.



Why is that? That's because an active screen overlay can listen for taps and intercept any information being passed to the underlying activity, right from passwords to credit card information! A scary prospect, to say the least.

And how would an attacker create a sneaky overlay? This is where the concept of a toast returns. While we as developers have the image of a tiny, short-lived rectangle etched in our brains when toasts are mentioned, there's nothing stopping a toast from being larger and include different forms of content like an image. And what about the lifetime of a toast? Here the clever attacker can make use of the inbuilt Android Timer; as soon as the timer runs out, the toast is redrawn on screen, giving the illusion of permanence. Done cleverly, a toast can be used for anything from listening for taps to presenting false password inputs to users.

It should be obvious by now why Tapjacking is a near-impossible exploit to stop: it just doesn't do anything intrusive.

How to prevent Android Tapjacking

If you look closely, Tapjacking is really simple to prevent. As long as your Android doesn't allow activities to gather input while an overlay is active, all is fine in app land. The reality, unfortunately, is grim. This security setting was disabled by default in Android 4.0.3 and before, making those versions the most infamous in Android history.

The gap was subsequently plugged and everyone was happy with the security model of Android 6. However, for reasons unknown, the Google developers again decided to turn off this setting in version 6.0.1, resulting in several cases of compromised user data. One reason seems to be that Google thought users wanted convenience more than the annoyance of setting permissions all the time, but the price for negligence has been too high.

So what can you do?

If you're a user, simply hop over to your Settings area and set the section deals with overlay screens. It should be called either "Apps that can appear on top" or apps that "Draw over other apps". If you're still not sure, a simple Google search for your phone make will reveal the setting.

If you're a developer, please relate with the plight of users and add the following to your checklist of pre-release: ensure that the setting *filterTouchesWhenObscured* is set to *true*, or that the method *onFilterTouchEventForSecurity()* is implemented in your app.

Tapjacking is not rocket science

To conclude, we'd like to say that understanding and preventing Tapjacking is not rocket science at all. It's a very simple exploit that relies on laziness on part of Google and/or app developers, and lack of awareness among users. However, now you know better.

https://medium.com/devknoxio/what-is-tapjacking-in-android-and-how-to-prevent-it-50140e57bf44

How to Check Tapjacking Vulnerability on Android Marshmallow and Nougat devices

1. Install both ***marshmallow-tapjacking.apk*** and ***marshmallow-tapjacking-service.apk*** files on your device.
2. Open **Tapjacking** app from your app drawer.
3. Tap on **TEST** button.

4. If you see a text box float on top of the permission window that reads **“Some message covering the permission message”**, then **your device is vulnerable to Tapjacking**. See screenshot below: Left: Vulnerable | Right: Not vulnerable

5. Clicking **Allow** will show all your contacts like it should. But if your device is vulnerable, not only you have given access to contacts permission but some other unknown permissions as well to the malicious app.

If your device is vulnerable, be sure to ask your manufacturer to release a security patch to fix the Tapjacking vulnerability on your device.

How to Safeguard yourself from Tapjacking Vulnerability

If your device has tested positive for the Tapjacking vulnerability, we would advise you to not give **Permit drawing over other apps** permission to apps that you do not fully trust. This permission is the only gateway for malicious apps to take advantage of this exploit.

Also, always ensure that the apps you install on your device come from a trusted developer and source.

https://nerdschalk.com/check-tapjacking-vulnerability-marshmallow/

Root detection

The Android operating system that we have on our phones is a commercial version of the OS provided by the manufacturer. This means that the end-user doesn’t have full control over their device due to system-level restrictions and safeguards.

To bypass these, it’s possible to perform a process called rooting, which grants the user root access. With a rooted device, the access control imposed by the operating system is compromised and we can’t guarantee the application sandbox features will securely protect our app’s private data.

In the worst-case scenario, the data can become exposed to malicious software that manages to elevate its privileges to root access.

Preparation phase – What you’ll need to know

To protect apps on rooted devices, we first need to detect whether a device is actually rooted. We do this by performing root detection.

It is important to know that no solution will give you a 100% accurate result. If the result indicates that the device is in fact rooted, we should follow the best practice recommended by OWASP in their Mobile Security Testing Guide book (page 84). We should notify the user that the app runs on a rooted device and that certain high-risk actions will carry additional risk due to its status, or just completely block our app’s usage.

For detecting rooted devices we recommend using an existing implementation rather than doing it from scratch. Google provides SafetyNet which has a set of API-s that help protect your app against security threats altogether.

SafetyNet has no specific API that can tell us if a device is rooted, it’s not a part of its design. However, we can use it to check the flags that we receive from the SafetyNet backend. The flags that we are looking for are ctsProfileMatch and basicIntegrity. If these two flags turn out

false, it implies that the system integrity has been compromised, and rooting is a potential cause.

There is no need to get into implementation details because the official documentation site offers good descriptions. You can also find some code examples in this repo.

However, you might have noticed that SafetyNet has a limit of 10,000 requests per day across your user base. This is a problem for most apps that are actually interested in that kind of a service but fortunately, there are ways to handle this limitation. The official way to handle it is described in the official documentation.

If this option doesn't work for you, you can implement a workaround that could cover most of your cases. It includes using the RECEIVE_BOOT_COMPLETED permission and running the SafetyNet request only when your application receives the mentioned system reboot event. This will keep your API quota limit under control, but it's not a future-proof solution. The official way is always the preferred one.

A good alternative to SafetyNet is RootBeer written by Scott Alexander-Bown who also wrote the Android Security Cookbook. You can also use it in combination with safetyNet to control your API quota limit.

Logs

Logging is very common in every application. We usually use it to pinpoint an undesired behavior, but we can also use it to track fatal and non-fatal crashes on our crashlytics tools.

Logs can be valuable during the development process. In most cases, we treat them as harmless pieces of characters. Unfortunately, that approach can sometimes put us in an undesirable situation where we can leak sensitive data through our logs.

As Android developers, we use Logcat for inspecting our application's logs. We only need a couple of command lines and we get a live dump of all the logs, not just from an individual app, but the entire system. A potential attacker could easily do the same and if they found some useful info, they could use it to exploit our application or the entire system.

Preparation phase – What you'll need to know

To prevent data leakage through logs, it's best to remove logs from production builds. Luckily, there are some handy tools to handle this task for us.

For example, we can use Timber for logging information and use the Tree feature to separate logging implementations for different application variants. We can use the default DebugTree from Timber to log information through Logcat in debug builds. For release builds we can create our own implementation that we will use to send information to our analytics or crashlytics tools. To implement your own tree, you can extend the Timber.Tree class and add your logic inside the log method:

```
class ReleaseTree() : Timber.Tree() {


  override fun log(priority: Int, tag: String?, message: String, t: Throwable?) {

  // handle received logs
```

```
    }
}
```

After you have your custom tree implementation you can set it inside the application class depending on the desired buildconfig, something like this:

```
Timber.plant(if (BuildConfig.DEBUG) Timber.DebugTree else ReleaseTree())
```

In case you don't use Timber you might stumble upon a ProGuard option to remove all log messages using this proguard rule:

```
-assumenosideeffects class android.util.Log {
public static boolean isLoggable(java.lang.String, int);
public static int v(...);
public static int i(...);
public static int w(...);
public static int d(...);
public static int e(...);
public static int wtf(...);
}
```

Keep in mind that the rule above will not help you with dynamically constructed strings used to log the data using the Log methods later on. This is due to the fact that dynamically constructed strings (i.e using StringBuilder) can still be seen in the bytecode. For more details about this particular issue, you can check the MSTG source.

Also, be careful about the above proguard rule because you can find a lot of similar rules on the Internet that can break your application. Check this article for more information. Always double-check your source when it comes to security!

Tapjack

Tapjacking is an old security issue that was well known on devices running Android versions 4.0.3. However, it can still be dangerous today if a user is unaware of it.

The exploit is based on the Android permission SYSTEM_ALERT_WINDOW which allows an app to draw an overlay over other apps. Attackers can use this option to create an overlay that would essentially hijack user taps and use it to obtain sensitive user information, hence the name tapjack.

The Android runtime permission introduced in Android API level 23 is a good measure of protection against tapjacking attacks because the users have to manually grant the permission to draw over other apps. Before API level 23 any developer could just add the permission in the manifest file and they would be able to use the feature immediately when the app is installed.

On some devices that run Android API level 23, there is a system-level security issue with the SYSTEM_ALERT_WINDOW. An app could use the exploit to draw an overlay over the system permission dialog. That would make it possible for the attacker to change the text of the permission dialog to make the user think the permission they are about to grant is not dangerous, while in reality, it could be the opposite.

In the worst-case scenario, this would allow the attacker to obtain user information from the device. Here you can find more information about that specific case.

Another very popular attack class involving the SYSTEM_ALERT_WINDOW permission is the cloak-and-dagger which also enables some advanced tapjacking attacks.

Preparation phase – What you'll need to know

Let's imagine a situation where your app has a login screen and the user has to enter their login credentials. Let's say that the user also has an app on their device that they trust, but is unaware that the app has malicious intent. That kind of an app can use the SYSTEM_ALERT_WINDOW to draw a transparent overlay over any other app and is specifically designed to track keyboard inputs.

With some additional effort, the attacker could obtain our user's login credentials for our app. To take it another step further, the attacker could literally obtain anything that the user enters on the keyboard.

To protect your app from these kinds of attacks, Android offers a built-in solution to detect overlays over specific UI elements that we think are exploitable. You can do this by using the following XML tag:

android:filterTouchesWhenObscured="[true|false]"

Keep in mind that by setting this flag to true, the view will not receive touches whenever a toast, dialog, or other window appears above the view's window. This will get the job done, but it is not recommended to leave it that way because the users will have a terrible time using your app. Imagine you want to press a button and nothing happens, that's just bad user experience.

Luckily, we have a way of detecting when exactly this happens. We can override the onFilterTouchEventForSecurity method in a compound view used to place it above other views. That way it can intercept obscured touch events and react to them.

```
override fun onFilterTouchEventForSecurity(event: MotionEvent): Boolean {
    if (event.flags and MotionEvent.FLAG_WINDOW_IS_OBSCURED ==
MotionEvent.FLAG_WINDOW_IS_OBSCURED) {
    // React to obscured events
    return false
  }
  return super.onFilterTouchEventForSecurity(event)
}
```

Fortunately, tapjacking, even though potentially very dangerous, is easy to handle. You should make sure your users are aware of the problem because the lack of awareness itself might be the biggest problem with this exploit.

There are signs that Google might deprecate the permission entirely. Android Go versions are already not allowed to use that feature. Also, features like Facebook Chat bubbles, which rely on the SYSTEM_ALERT_WINDOW permission , are receiving alternate API solutions to cover their use cases. It could be taken as an indication that the long-term plan is permission removal. We will have to wait and see.

Clipboard manager

Clipboard is one of Android's system components. It is exposed to developers through a class called ClipboardManager which grants copying, monitoring, and paste operations for specified data.

One of the clipboard component's main characteristics is globally accessible to any app running on the device and no additional permission is needed. This leaves users vulnerable in case there is a malicious app on their device which could sniff out its clipboard and obtain passwords, credit card details, and other sensitive user information. This article shows how Android's clipboard could be exploited in a password manager app.

Preparation phase – What you'll need to know

Sadly, there is no good way to mitigate attacks on data contained in the clipboard from within our app, so it should be handled on a system level. There are some mitigation tactics but they strongly depend on the use case you want to achieve. One of the most common and safest approaches is to disable the copying of sensitive data altogether. This better-safe-than-sorry scenario can leave your users with a bad UX, but it protects those less technically savvy.

Screenshots

When you think about it, taking screenshots is very similar to copying plain text. The vulnerability area is very similar to the one from the previous chapter. The only difference, except the data type, is the location of the saved data. Screenshots are usually saved in the device's internal storage in a folder named Screenshots, but this may slightly vary depending on the manufacturer.

Preparation phase – What you'll need to know

The mitigation process is also very similar to the one for Clipboard manager. Luckily, we can use the Android built-in API to determine which screens can be screenshotted and which are prohibited. We do this by using the window flag FLAG_SECURED. The snippet below shows how to set the flag in a fragment.

```
requireActivity().window.setFlags(WindowManager.LayoutParams.FLAG_SECURE,
WindowManager.LayoutParams.FLAG_SECURE)
```

This will also protect your app from screen-recording apps.

Inter-process communication (IPC)

When we install our apps on the Android OS they run in their own secure sandbox. However, like any other system, Android also gives us IPC capabilities so we can communicate with the rest of the system or with other apps.

We can use more traditional ways like using shared files to communicate, but this is not recommended since we have more evolved solutions provided by the Android OS. One of those solutions is Services. They are components used for long-running operations in the background, but they can also be used for IPC with bound services where other components can bind to a service and interact with it.

Further, there is ContentProviders or a more advanced mechanism like AIDL.

Probably the most common way we use for communication between different Android components is Intents. An Intent is an abstract description of an operation that we can use to communicate with Activities, BroadcastReceivers, and Services. This is a broad subject so we will try to focus on the main techniques for protecting your application components from unwanted interactions.

Preparation phase – What you'll need to know

There are several tools at your disposal for making your app safe for IPC. If your app does not handle IPC, make sure that your main components don't have the exported flag set to true in the manifest. It is false by default, but sometimes we set it to true in case we want to start a specific activity from Android Studio for faster development. Just make sure you don't forget to check that flag before release or create a custom lint rule if you already don't have it.

If your components need to be exported for IPC, you should definitely make it restrictive because sniffing broadcasted Intents is as simple as writing a terminal command using tools like Drozer.

What you want to do is set the corresponding permissions for your components so that you indicate to targets that want to communicate with your application that they need to follow the rules you set. Together with that tag and the protection level you can control the restriction level for your components.

For example, if your company has multiple applications in production and you want to make sure that only your applications can communicate with each other in order to avoid impersonated intents from potential attackers, we can define a custom permission inside our manifest like this:

```
<manifest  package="com.example.myapp" >


  <permission

    android:name="com.example.myapp.permission.READ_USER_DATA"

    android:label="Read user data"

    android:description="Can access user data like email, username, password, ..."

    android:protectionLevel="signature" />
```

</manifest>

You'll notice that protectionLevel is set to signature mode, which gives us the desired effect described above. In other words, this is automatically granted to a requesting app if that application is signed by the same certificate. For more information about other protection levels, check the protectionLevel documentation. With our custom permission defined, we can use it in our components. For example, if we want to protect our service with this permission, we would write the following code in our manifest:

<service

android:name=".data.session.SessionService"

android:permission="com.example.myapp.permission.READ_USER_DATA"/>

To protect our entire application with this permission, we can add this to the application tag in our manifest.

Alongside the exported flag and custom permissions we also have intent filters in our arsenal to protect our applications from malicious and undesired IPC.

Intent filters can be used to specify the types of intents that an activity, service, or broadcast receiver can respond to. That way we can anticipate the input in our components. Whatever you decide to use, it is always good practice to check the data you receive programmatically to see if it is something that you expect.

Static Code Analysis

How does linting works?

Linting follows the rules defined in a configuration file like lint.xml. Lint tool then runs those rules against the source code files. See the below image for better understanding.

Using lint in project

There are two ways to do it, using Android Studio and other using terminal and gradle.

Using Android Studio

There are two ways to run the lint tool on your source code. You can just do it from the toolbar Analyze > Inspect Code.. It will then open a dialog where you can specify the scope of the source code on which the lint tool run. Below is an image demonstrating that.After a while, Android studio will present you the results in the Inspection results window as shown in the below image.

Using Gradle

To run lint from Gradle, you can use following commands.

On Windows: gradlew lint

On Linux or Mac: ./gradlew lint

Note that when you run the above commands, gradle by default runs lint on the release build. In order to run it on a different build eg. debug, you need to add the build name as gradlew lintDebug. After the linting is complete, the results are generate in html and xml format.

Note that if you have any lint error, Android studio may not generate the results so you need to ad below lines in your app level gradle file:

```
lintOptions {
        abortOnError false
```

Customizing linting rules.

Perhaps your needs or team's coding convention is different from the default configuration , so you can change settings from gradle files as below:

```
lintOptions {
        abortOnError false
        disable 'ContentDescription'
   }
```

In the above example, we've disable lint check for ContentDescription warning in the entire project. If you don't want to apply this to the entire project but just to some files, you can use @SupressLint annotation on Kotlin or Java files and tools:ignore on xml files. Checkout the example below:

```
@SuppressLint("NewApi")
override fun onCreate(savedInstanceState: Bundle?) {
   super.onCreate(savedInstanceState)
   setContentView(R.layout.main)
```

In case of XML files:

```
<ImageView
        android:id="@+id/avatar_url"
        tools:ignore="HardcodedText"
```

Remember a few minutes ago we said that lint tool makes use of lint.xml configuration file? You can create your own lint.xml file and set rules according to your needs. In the below example, we have created our lint.xml file and set a rule to ignore missing contentDescription warning.

```
 <?xml version="1.0" encoding="UTF-8"?>
<lint>
   <!-- Disable the given check in this project -->
   <issue id="ContentDescription" severity="ignore" />
</lint>
```

After that, you need to put the reference in the gradle file as:

```
lintOptions {
      lintConfig file("lint.xml")
   }
```

Note that in the above case, lint rules you specify in the lint.xml applies to the entire module. If you want to specify it for specific path, you can do so as:

```
<issue id="ContentDescription" severity="ignore">
  <ignore path="src/main/res/values/list_item.xml" />
</issue>
```

So that's it! Go ahead and use this awesome tool in your next project and make a step towards quality code. For more information, feel free to visit this link Lint

https://www.aanandshekharroy.com/articles/2019-01/static-code-analysis-using-android-studio#:~:text=Static%20analysis%20(or%20static%20code,step%20taken%20towards%20code%2Danalysis.

What is the definition of Static Code Analysis?

Basically, Static Code Analysis is the analysis of computer software, which is performed without actually executing programs, in contrast with dynamic analysis that is analysis performed on programs while they are executing. It involves the detection of vulnerabilities and functional errors in deployed or soon-to-be deployed software. Furthermore, Static Code Analysis is performed before the beginning of software testing phase.

In short, this method of analysis addresses weaknesses in source code, which might lead to security flaws and bugs. Besides, this could be achieved through manual code reviews; however, using automated tools is much more effective. Recently, many software companies are requiring projects to pass Static Code Analysis, in addition to doing code reviews and unit testing in the build process. Therefore, learning about Static Code Analysis is a key factor in having an Android application with high quality codes.

Advantages of using Static Code Analysis in Android apps

- As we know, it takes time for developers to do manual code reviews. Automated tools are much faster indeed because it points out exactly where the bugs are in the code.
- Helping detect potential bugs and errors that even unit or manual testing might have missed.
- Defining project-specific rules, for instance: it helps you define a new project structure. Thus, they are configurable and customizable for your specific needs in your Android apps.
- Helping you enhance your knowledge in Android programming.
- Tracing and scanning all files and codes that you might not have ever read accurately. They scan every line of code to identify potential problems. So, this helps you achieve the highest possible quality of codes.

Introducing Static Code Analysis tools in Android

Basically, there are a number of tools and platforms for Static Code Analysis. However, this article aims to introduce some of them, which are used in Android programming as follows:

Programming Mistake Detector (PMD)

PMD is a open-source code analysis tool. It finds common programming flaws such as unused variables, empty catch blocks, unnecessary object creation, and so on. PMD includes built-in rules, and supports the ability to write custom rules as well. It supports a number of languages particularly Java. Also, it includes CPD, the copy-paste-detector. In a word, CPD finds duplicated code in some languages like Java. The complete features of PMD tool are mentioned as below:

- Possible bugs: Empty try, catch, finally, and switch blocks.
- Dead code: Unused local variables, parameters and private methods.
- Empty if and While statements
- Over-complicated expressions: Unnecessary if statements for loops that could be while loops.
- Sub-optimal code: Wasteful String and StringBuffer usage.
- Classes with high cyclomatic Complexity measurements.
- Duplicate code: Copy-paste code can mean copied-pasted bugs, and decreases maintainability.

FindBugs

FindBugs is an open source Static Code Analysis tool that analyses Java byte-code, and it detects a wide range of bugs and problems.

Some of them are indicated as below:

1. Empty finalizer should be deleted.
2. Class defines equals(), but not hashCode().
3. Confusing method names.
4. Impossible downcast.
5. Synchronize and null check on the same field.
6. TestCase has no tests.
7. Field should be package protected.
8. Repeated conditional tests.
9. Useless object created.

For more bugs and problems that are addressed by FindBugs, you should check this document that is called standard bug patterns.

Checkstyle

Checkstyle as an open source tool can check many aspects of your source code. It can find class design and method design problems. Furthermore, it has the capability to check code layouts and formatting issues.

Checkstyle is highly customizable, and it can be used to support some popular coding standards such as Sun Code Conventions and Google Java Style. In other words, you can mention some rules in an XML file to enforce a standard for your project in implementing. In fact, Checkstyle enforces those rules by analyzing your source code, and compares them with accredited standards or conventions.

Lint

The Lint tool checks your Android project source files for potential bugs and optimization improvements for correctness, security, performance, usability, accessibility, and internationalization. Besides, you can configure lint checking for different levels of your project. For instance: entire project and project module.

The whole process of lint can be classified in three steps:

1. *Creating lint.xml file*
2. *Selecting the source code for performing analysis by Lint (.java/.kt/XML file)*
3. *Checking for bugs and suggesting some improvements*

You can customize the Lint checks in the *lint.xml* file. It means you can indicate the rules that you want to include, and ignore the checks that you do not want to include in this file. For instance, if you want to check the unused variables and also do not want to check for naming

conventions, you can accomplish these tasks in lint.xml file (you should make this new file in the root directory of your Android project).

To configure Lint, you have to include the lintOption block in your module-level build.gradle file:

```
android {    lintOptions {       abortOnError false
      quiet true
      lintConfig file("$project.rootDir/tools/rules-lint.xml")
      htmlOutput file("$project.buildDir/outputs/lint/lint.html")
      warningsAsErrors true
      xmlReport false
   }
}
```

- *lintConfig:* the path to lint rule sets file where you can check issues.
- *htmlOutput:* the path where html report will be generated.
- *abortOnError*: If errors find, Lint should exit the process.
- *quiet*: whether to turn off analysis progress reporting

Your *lint.xml* file can include issues for Lint to ignore or modify. The below example is mentioned that Lint must ignore Icon Colors Check for whole project and ignore the path that is mentioned in the below code:

```
<lint>

   <issue id="IconColors"  severity="ignore" />    <issue id="Overdraw">         <ignore
path=".../navigation.xml" />     </issue></lint>
```

A more complete example can be defined like this:

```
<?xml version="1.0" encoding="UTF-8"?><lint>   <!-- Disable the given check in this project -->
   <issue
id="IconMissingDensityFolder" severity="ignore" /><!-- Ignore the ObsoleteLayoutParam issue
in the specified files -->
   <issue id="ObsoleteLayoutParam">
      <ignore path="res/layout/activation.xml" />
      <ignore path="res/layout-xlarge/activation.xml" />
   </issue>
<!-- Ignore the UselessLeaf issue in the specified file -->
   <issue id="UselessLeaf">
      <ignore path="res/layout/main.xml" />
   </issue>
<!-- Change the severity of hardcoded strings to "error" -->
   <issue id="HardcodedText" severity="error" /></lint>
```

In addition to this way, you can be able to customize the Lint checks manually in this path in Android Studio: *Files > Settings > Editor > Inspections*

A significant note can be added to this section about helping Lint for understanding the source code. In other words, you can use some annotations in your code for finding bugs more efficiently by Lint.

In conclusion

Recently, some accredited companies use Static Code Analysis tools in addition to Test and Code Review processes specially in designing and implementing an Android app because of the importance of quality in this major. In this article the importance of this issue and also some tools for Static Code Analysis in Android were considered. Basically, the aim is to find potential vulnerabilities such as bugs and security flaws in a source code in Android development.

https://medium.com/kayvan-kaseb/static-code-analysis-in-android-10c3ef83a29a

Android App Development Companies use tools that parse and analyze your source code without actually executing it. The goal is to find potential vulnerabilities such as bugs and security flaws and ensure conformance to coding guidelines. **Mobile app development services** from **Perfomatix** using these tools helps to keep your code healthy and maintain code quality.

Think like static code analysis tools as an additional compiler that will run before the final conversion into the system language.

Benefits

1. It helps to detect potential bugs that even manual or unit testing might have missed.
2. Defines project-specific rules. For example, it helps to define a new project structure
3. Helps you improve your knowledge of a new language.
4. Scans your whole project, including files that you might not have ever read.

On Android, the most popular code analysis tools are:

1. Android Lint
2. Checkstyle
3. Findbugs

Before starting the android integration, it would be great if a gist of what every tool does, should be provided. So, here it goes.

Android Lint

1. Android lint is the one comes with Android Studio by default.
2. It will check your project source files for identifying potential bugs and optimizations for usability, correctness, performance, security, accessibility, and internationalization.

FindBugs

1. It analyses Java bytecode mainly the .classes to find any design flaw and potential bugs.

2. It needs compiled code to work around and will eventually be fast since it works on bytecode level.
3. The major sections in this tool are: Bad practice, Correctness, Multithreaded Correctness, Dodgy code, Performance Malicious, Code Vulnerability, Security Experimental and Internationalization

Checkstyle

1. It basically analyses the source code of the project and looks to improve the coding standard by traversing over simple AST generated by Checkstyle.
2. It verifies the source code for coding conventions like headers, whitespaces, formatting, imports etc.

How to set up?

All code analysis tools we'll going to learn about in this tutorial are available as Gradle plugins, so we can create individual Gradle tasks for each of them. So we are going use a single Gradle file that will include them all. But before going to that, let's create a folder that will contain all of our files for the static code analysis.

Open Android Studio and inside the app module, create a new directory named **code_quality_tools**. This folder will contain the XML files for the code analysis tools, and it will also have a Gradle file, **quality.gradle**, which will run our static analysis tasks for each tool.

Finally, visit **build.gradle** in the app module folder and include the below line at the end of the file:

```
apply from: '/code_quality_tools/quality.gradle'
```

Here, our **quality.gradle** Gradle script is being applied with a reference to its local file location.

Android Lint

To configure Lint, you have to include the lintOptions {} block in your module-level **build.gradle** file:

```
lintOptions {
  abortOnError false
 quiet true
 lintConfig file('./code_quality_tools/lint.xml')
}
```

The key Lint options we are concerned with are:

1. **abortOnError**: whether lint should set the exit code of the process if errors are found.
2. **quiet**: whether to turn off analysis progress reporting.
3. **lintConfig**: the default configuration file to use.

Your **lint.xml** file can include issues you want Lint to ignore or modify, such as the example below:

```
<?xml version="1.0" encoding="UTF-8"?>

<lint>

  <!-- Disable the given check in this project -->

  <issue id="IconMissingDensityFolder" severity="ignore" />

  <!-- Change the severity of hardcoded strings to "error" -->

 <issue id="HardcodedText" severity="error" />

</lint>
```

You can also run Lint by using Gradle tool window, opening the **verification** group in that, and then clicking on **lint**. Finally, you can run it via the command line as.

On Windows:gradlew lint
On Linux or Mac:./gradlew lint

Finally, a report will be generated after the task has finished executing, and is available at **app module > build > outputs > lint-results.html**.

Checkstyle

Checkstyle enforces the given rules you specify in an XML file by analyzing your project source code and compares them against known conventions or coding standards.

Checkstyle is an open-source tool and is well maintained by the community. This means you can create your own custom checks or modify existing ones to suit to achieve your need. For example, Checkstyle can perform a check on the constant names like final, static, or both in your classes. If the constant names do not obey a rule of being in uppercase with words separated by an underscore, the problem will be displayed in the final report.

```
// incorrect

private final static String myConst = "myConst";

// correct
```

Integrating Checkstyle

I will going to show you how to add Checkstyle into our Android Studio project and will demonstrate a practical example.

First, we need to create our coding rules. Inside **checkstyle.xml (inside code_quality_tools directory)**, we create some Checkstyle configuration rules that will be run against our source code.

```
<?xml version="1.0"?>
```

```
<module name="Checker">


<!-- Checks for Naming Conventions-->

<!-- See https://checkstyle.sourceforge.net/config_naming.html -->

<module name="MethodName"/>

<module name="ConstantName"/>

<!-- Checks for Imports -->

<!-- See https://checkstyle.sourceforge.net/config_imports.html-->

<module name="AvoidStarImport"/>

<module name="UnusedImports"/>

<!-- Checks for Size -->

<!-- Seehttps://checkstyle.sourceforge.net/config_sizes -->

<module name="ParameterNumber">

<property name="max" value="6"/>

</module>

<!-- other rules ignored for brevity -->

</module>
```

In the above section, we include the rules or checks we want Checkstyle to validate in our source code. One rule is AvoidStarImport which, as the name says, it checks if your source code included an import statement like java.Utils.*. (Instead, you should explicitly specify the package to import, e.g. java.util.ArrayList.)

Take a look at other checks on the Checkstyle website.

To run this check over source code, we need to create a Gradle task. So visit the **quality.gradle** file and create a task called checkstyle like below:

```
apply plugin: ‘checkstyle’

task checkstyletask(type: Checkstyle) {

description 'Check code standard'

group 'verification'

configFile file('./code_quality_tools/checkstyle.xml')

source 'src'

include '**/*.java'

exclude '**/gen/**'
```

classpath = files()

ignoreFailures = false

}

Note that in the code above, we first applied the Checkstyle Gradle plugin. We gave it a description and added it to an already predefined Gradle group called verification.

The key properties of the Checkstyle Gradle task in the above task are:

1. **configFile**: the Checkstyle configuration file to use.
2. **IgnoreFailures**: whether or not to allow the build to continue if there are warnings.
3. **include**: the set of include patterns.
4. **exclude**: the set of exclude patterns. In this case, we don't scan generated classes.

Finally, you can run the Gradle script by using the Gradle tool window on Android Studio, opening the **verification** group, and then clicking on **checkstyletask** to run the task.

-

You can use command line to execute checkstyle task

gradle checkstyletaskAfter finished running, a report will be created and is available at **app module > build > reports > checkstyle**. You can open **checkstyle.html** to view the report.

A Checkstyle plugin is available for Android Studio or IntelliJ IDEA. It offers real-time scanning of your Java files and is completely free.

FindBugs

FindBugs is also a free static analysis tool which analyses your class and looking for potential problems by checking your bytecodes against a known list of bug patterns. Some of them are:

- **Class defines hashCode() but not equals()**: A class implements the hashCode() but not equals()—therefore both instances might be equal but not have the same hash codes. This falls under bad practice category.
- **Bad comparison of int value with long constant**: The code is comparing an int value with a long constant that is outside the range of values that can be represented as an int value. This comparison will yield an unexpected result. This falls under the correctness category.

FindBugs is open-source, so you can contribute or monitor the progress of the source code on GitHub. In the **findbugs-exclude.xml** file, if we want to prevent FindBugs from scanning some classes such as auto-generated resource classes and auto-generated manifest classes, they will do it by using regular expressions in our projects.

<FindBugsFilter>

<!-- Do not check auto-generated resources classes -->

<Match>

<Class name="~.*R\$.*"/>

</Match>

<!-- Do not check auto-generated manifest classes -->

<Match>

<Class name="~.*Manifest\$.*"/>

</Match>

<!-- Do not check auto-generated Dagger classes-->

<Match>

<Class name="~.*Dagger*.*"/>

</Match>

<!--

https://findbugs.sourceforge.net/bugDescriptions.html#ST_WRITE_TO_STATIC_FROM_INSTANCE_METHOD–
>
<Match>
<Bug pattern=”ST_WRITE_TO_STATIC_FROM_INSTANCE_METHOD” />
</Match>
</FindBugsFilter>

And finally, we’ll include the **findbugs** task in **quality.gradle**:apply plugin: ‘findbugs’

task findbugs(type: FindBugs) {

description 'Run findbugs'

group 'verification'

classes = files("$project.buildDir/intermediates/classes")

source 'src'

classpath = files()

effort 'max'

reportLevel = "high"

excludeFilter file('./code_quality_tools/findbugs-exclude.xml')

reports {

xml.enabled = false

html.enabled = true

}

ignoreFailures = false

}

In the first line above, we applied FindBugs as a Gradle Plugin and then created a task called findbugs. The key properties of the findbugs task we are really concerned with are:

1. **classes**: the classes to be analyzed.
2. **effort**: the analysis effort level. The value specified should be one of min, default, or max. Be aware that higher levels increase precision and find more bugs at the cost of running time and memory consumption.
3. **reportLevel**: the priority threshold for reporting bugs. If set to low, all bugs are reported. If set to medium (the default), medium and high priority bugs are reported. If set to high, only high priority bugs are reported.
4. **excludeFilter**: the filename of a filter specifying bugs to exclude from being reported, which we have created already.

You can then run the Gradle script by visiting the Gradle tool window, opening the verification group folder, and then clicking on **findbugs** to run the task. Or launch it from the command line:

gradle findbugs

A report will also be generated when the task has finished executing. This will be available at **app module > build > reports > findbugs**. The FindBugs plugin is another freely available plugin for download and integration with either IntelliJ IDEA or Android Studio.

```
apply plugin: 'findbugs'

task findbugs(type: FindBugs) {

description 'Run findbugs'

 group 'verification'

 classes = files("$project.buildDir/intermediates/classes")

source 'src'

classpath = files()

effort 'max'

reportLevel = "high"

excludeFilter file('./code_quality_tools/findbugs-exclude.xml')

reports {

xml.enabled = false

html.enabled = true

}

ignoreFailures = false
```

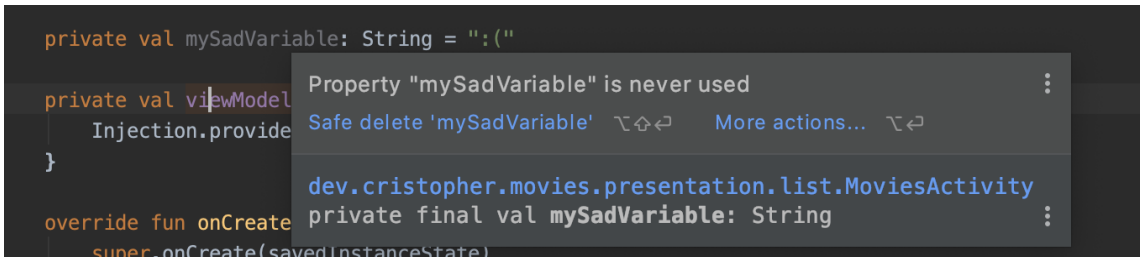
}

Static analysis tools for Android

Photo by Zach Vessels on Unsplash

Let's take a look into the most popular static code analysis tools that you can use to implement and enforce custom rules in your codebase. Some of the benefit from using a linter. These benefits are: enforce standards programmatically, automate code quality and code maintenance.

In Android Studio you're probably familiar with these kind of messages.



You can write your own rules by using these tools:

- Android Lint API
- ktlint
- detekt

We'll describe the step by step process to write some rules on a demo project that you can find here.

Custom rules with Android Lint API

To start with, we're going to write rules by using the Android Lint API. Some of the advantages are:

- You can write rules for Java, Kotlin, Gradle, XML and some other file types.
- No need to add plugins to make the warnings/errors visible on Android Studio

- Simpler integration with your project.

One of the disadvantages is this footnote on their repository https://github.com/googlesamples/android-custom-lint-rules

The lint API is not a final API; if you rely on this be prepared to adjust your code for the next tools release.

So, these are the steps to create our first rule:

1. Create a new module in your project where your custom rules will live in. We'll call this module **android-lint-rules.**
2. Modify the **build.gradle** file on that module to something like this.

Here we're importing as a **compileOnly** the dependency that will allow us to write our custom rules **com.android.tools.lint:lint-api**. You should also beware that here I'm using the **lint-api:27.2.0**, which is still on **beta**.

Here we also specify the **Lint-Registry-v2** which will point to the class that will contain the list of rules.

3. Write the first rule to avoid hardcoded colors on our layouts.

Depending on the rule that we want to implement, we'll extend from a different **Detector** class. A detector is able to find a particular problem. Each problem type is uniquely identified as an **Issue**. In this case we'll use a **ResourceXmlDetector** since we want to check for hardcoded hexadecimal colors in each xml resource.

After the class declaration we create all the information needed to define an **Issue**. Here we can specify the category and severity, along with the explanation that will be display in the IDE if the rule is triggered.

Then we need to specify the attributes that are going to be scanned. We can return a specific list of attributes like this **mutableListOf("textColor", "background")** or we can return **XmlScannerConstants.ALL** to scan all the attributes on each layout. That'll depend on your use case.

Finally we have to add the logic needed to decide if that attribute is an hexadecimal color, so we can raise a report.

4. Create a class called **DefaultIssueRegistry** that extends **IssueRegistry.** Then you need to override the **issues** variable and list all of them.

If you are going to create more rules, you need to add all of them here.

5. To check that the rule is doing their job correctly we're going to implement some tests. We need to have on our **build.gradle** these two dependencies as **testImplementation**: **com.android.tools.lint:lint-tests** and **com.android.tools.lint:lint.** Those will allow us to define a xml file right in the code and scan their content to see if the rule is working fine.

1. The first test check if our rule still works if we're using a custom property. So the TextView will contain a property called **someCustomColor** with the color **#fff**. Then, we can add several issues to scan the mock file, in our case we just specify our only

written rule. Finally we say that the expected result should be 1 issue with an error severity.

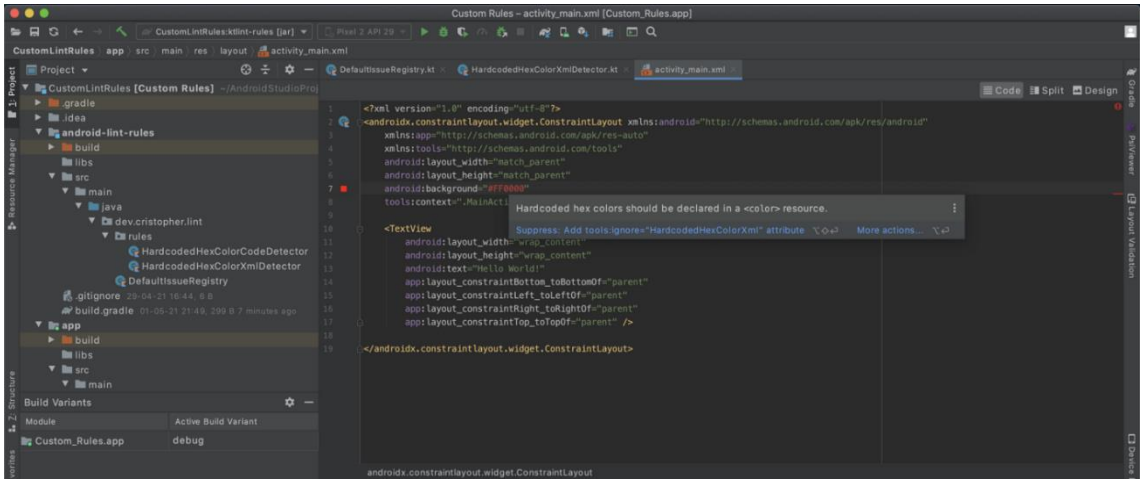
2. In the second test the behavior is really similar. The only change is that we're testing our rule with a normal property and the hexadecimal color is including the alpha transparency.

3. In the last test we check that our rule doesn't raise any error if we specify a color by using our resources. In that case we set a text color with **@color/primaryColor** and the expected result is a clean execution.

6. Now in the **app module**, where we want to apply all these rules, we're going to add this line to the **build.gradle** file:

```
dependencies {
    lintChecks project(':android-lint-rules')
  ....
}
```

And that's it! If we try to set a hardcoded color in any layout an error will be prompt 🎉



This repository can be a good source if you need more ideas to add some custom rules https://github.com/vanniktech/lint-rules

Custom rules with ktlint

ktlint define itself as an anti-bikeshedding Kotlin linter with built-in formatter. One of the coolest things is that you can write your rules along with a way to autocorrect the issue, so the user can easily fix the problem. One of the disadvantages is that it's specifically for Kotlin, so you can't write rules for XML files, as we previously did. Also if you want to visualize the issues on Android Studio, you need to install a plugin. I'm using this one https://plugins.jetbrains.com/plugin/15057-ktlint-unofficial-

So, in this case we're going to enforce a rule about Clean Architecture. Probably, you have heard that we shouldn't expose our models from the data layer in our domain or presentation layers. Some people add a prefix on each model from the data layer to make them easy to identify. In this case we want to check that every model which is part of a package ended on **data.dto** should have a prefix **Data** in their name.

These are the steps to write a rule using ktlint:

1. Create a new module where your custom rules will live in. We'll call this module **ktlint-rules**
2. Modify the **build.gradle** file on that module:

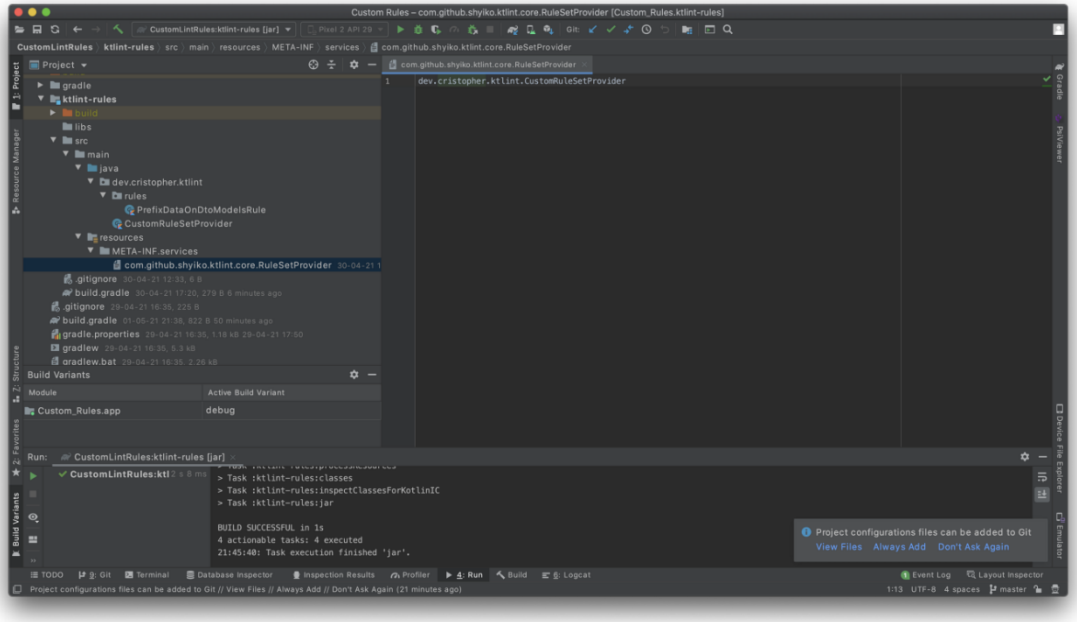
3. Write a rule to enforce the use of a prefix (**Data**) in all the models inside a package name ending on **data.dto.**

First we need to extend the **Rule** class that ktlint provide for us and specify an id for your rule.

Then we override the **visit** function. Here we're going to set some conditions to detect that the package ends with **data.dto** and verify if the classes inside that file has the prefix **Data**. If the classes doesn't have that prefix, then we're going to use the emit lambda to trigger the report and we'll also offer a way to fix the problem.

4. Create a class called **CustomRuleSetProvider** that extends **RuleSetProvider.** Then you need to override the **get()** function and list all your rules there.

5. Create a file in the folder **resources/META-INF/services**. This file must contain the path to the class created on the step 4.

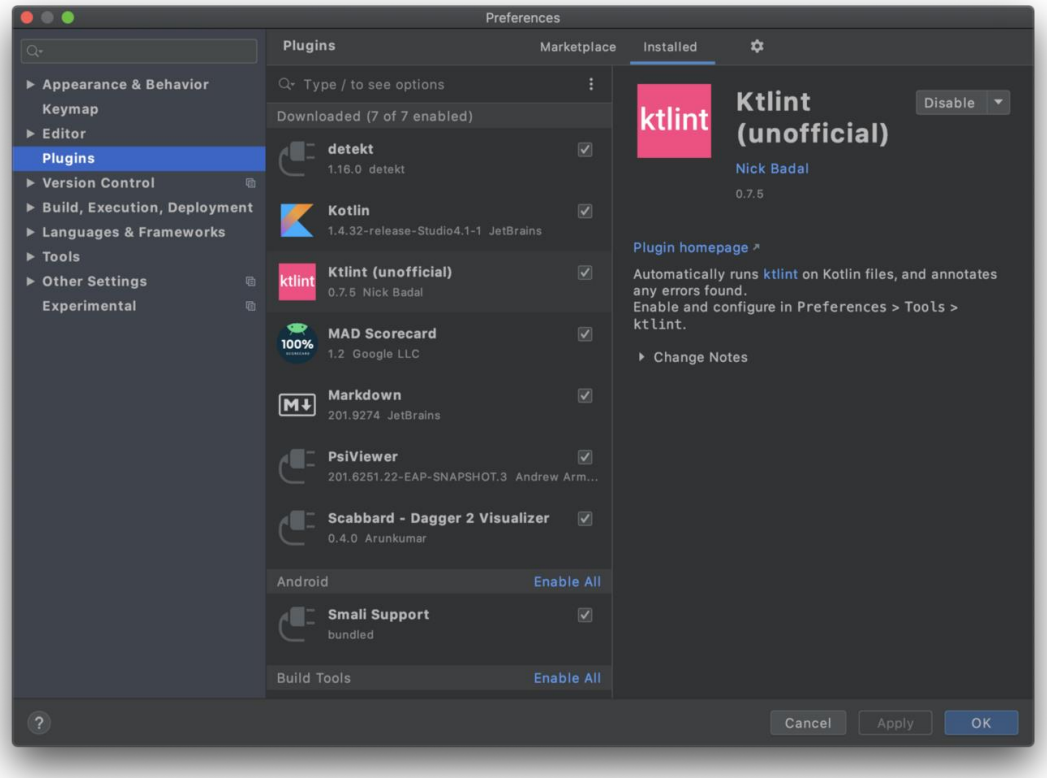


6. Now in our project we're going to add this module, so the rules can be applied. Also we created a task to execute ktlint and generate a report:

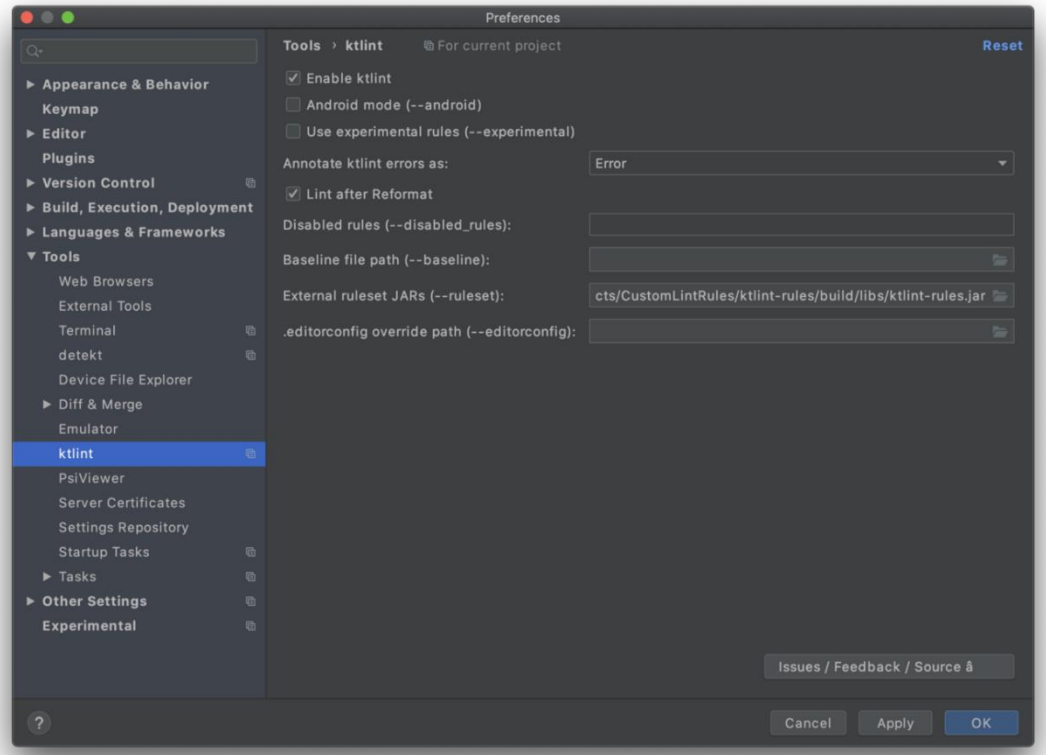
```
configurations {
   ktlint
}

dependencies {
   ktlint "com.github.shyiko:ktlint:$ktlintVersion"
   ktlint project(":ktlint-rules")
   ...
```

```
}task ktlint(type: JavaExec, group: "verification", description: "Runs ktlint.") {
    def outputDir = "${project.buildDir}/reports/ktlint/"

    main = "com.github.shyiko.ktlint.Main"
    classpath = configurations.ktlint
    args = [
        "--reporter=plain",
        "--reporter=checkstyle,output=${outputDir}ktlint-checkstyle-report.xml",
        "src/**/*.kt"
    ]

    inputs.files(
        fileTree(dir: "src", include: "**/*.kt"),
        fileTree(dir: ".", include: "**/.editorconfig")
    )
    outputs.dir(outputDir)
}
```

7. I also highly recommend to install this plugin so you can be notified in the same Android Studio about any errors found.



To see your custom rules in Android Studio you need to generate a jar from your module and add that path in the external rulset JARs like this:



Custom rules with detekt

detekt is a static code analysis tool for the *Kotlin* programming language. It operates on the abstract syntax tree provided by the Kotlin compiler. Their focus is find code smells, although you can also use it as a formatting tool.

If you want to visualize the issues on Android Studio, you need to install a plugin. I'm using this one https://plugins.jetbrains.com/plugin/10761-detekt

The rule that we're going to implement will enforce the use of a specific prefix for the Repository implementations. It's just to show that we can create a custom standard in our project. In this case if we have a **ProductRepository** interface, we want that the implementation use the prefix **Default** instead of the suffix **Impl.**

The steps to write a rule using detekt are:

1. Create a new module where your custom rules will live in. We'll call this module **detekt-rules**
2. Modify the **build.gradle** file on that module:

3. Write a rule to enforce the use of a prefix (**Default**) in all the repository implementations.

First we need to extend the **Rule** class that detekt provide for us. Also we need to override the issue class member and specify name, type of issue, description and how much time it requires to solve the problem.

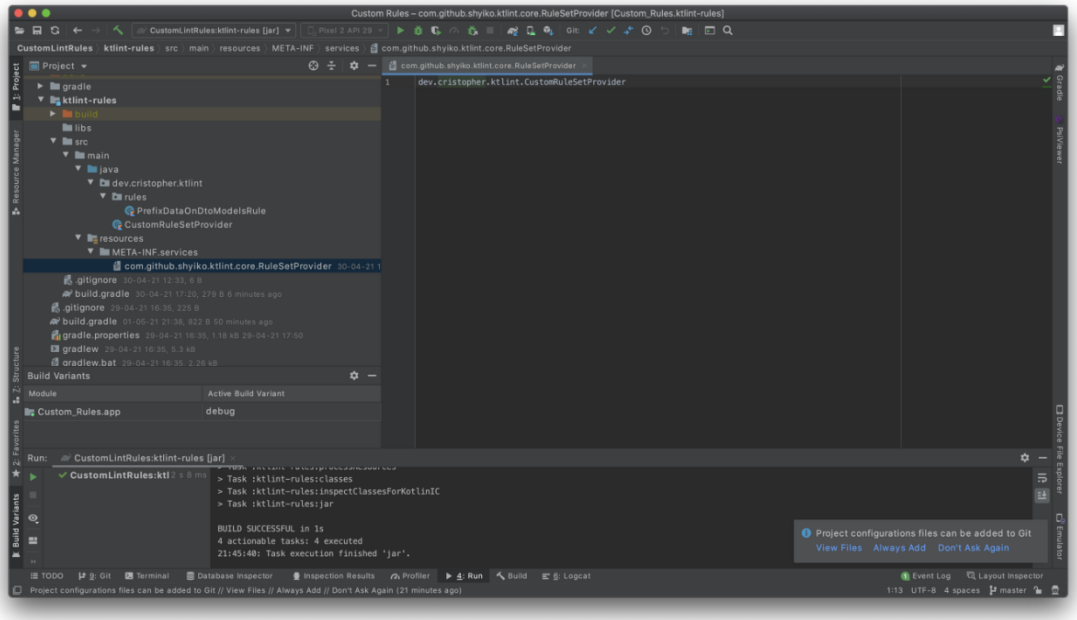
Then we override the **visitClassOrObject** function. Here we check for each implementation of each class. If some of these ends in the keyword **Repository**, then we're going to verify if the

class name doesn't start with our prefix. Inside that condition we will report the problem as a **CodeSmell.**

The next steps are pretty similar to the ones on ktlint.

4. Create a class called **CustomRuleSetProvider** that extends **RuleSetProvider.** Then you need to override the **ruleSetId()** and the **instance(config: Config)** functions and list all your rules there.

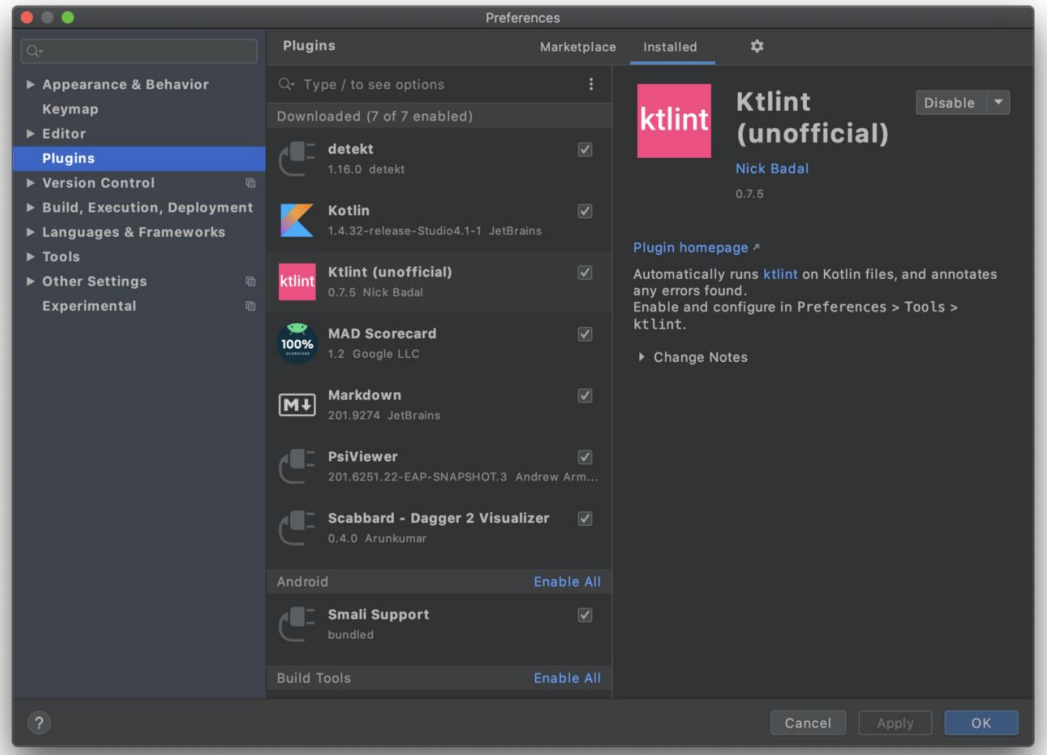
5. Create a file in the folder **resources/META-INF/services**. This file must contain the path to the class created on the step 4.



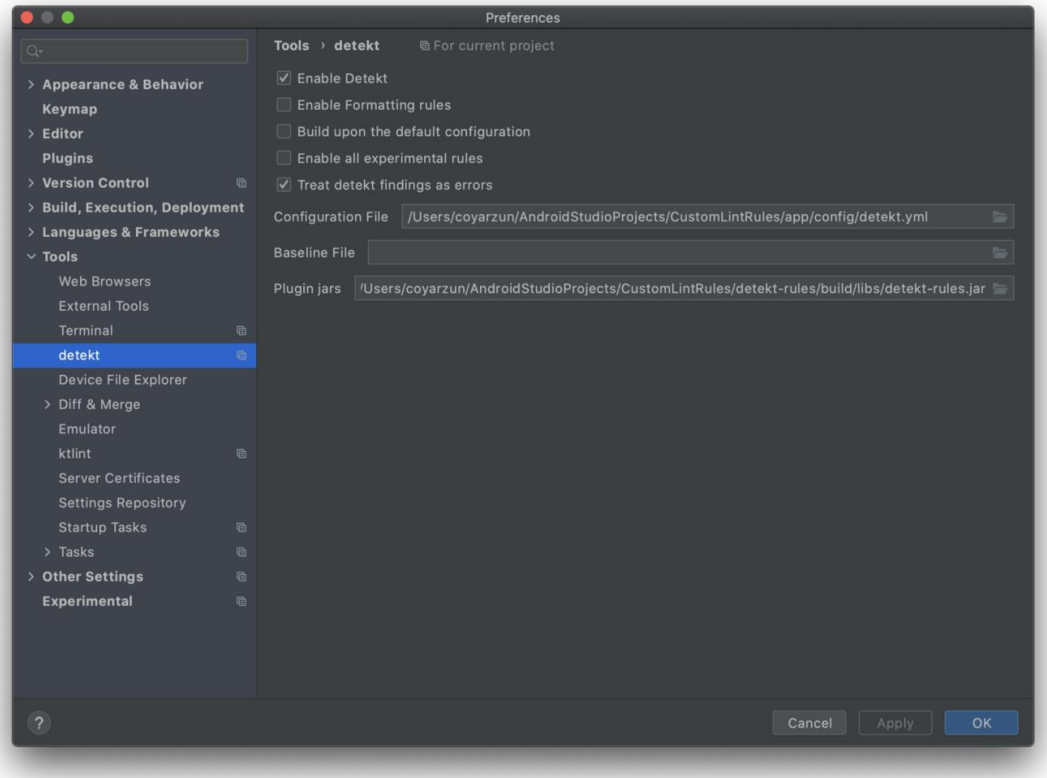
6. Now in our project we're going to add this module, so the rules can be applied. To use detekt in your project you also need to a yaml style configuration file. You can get the default configuration from the same detekt repository here.

```
detekt {
   input = files("$rootDir/app/src")
   config = files("$rootDir/app/config/detekt.yml")
}dependencies {
   detektPlugins "io.gitlab.arturbosch.detekt:detekt-cli:$detektVersion"

   detektPlugins project(path: ':detekt-rules')
   ...
}
```

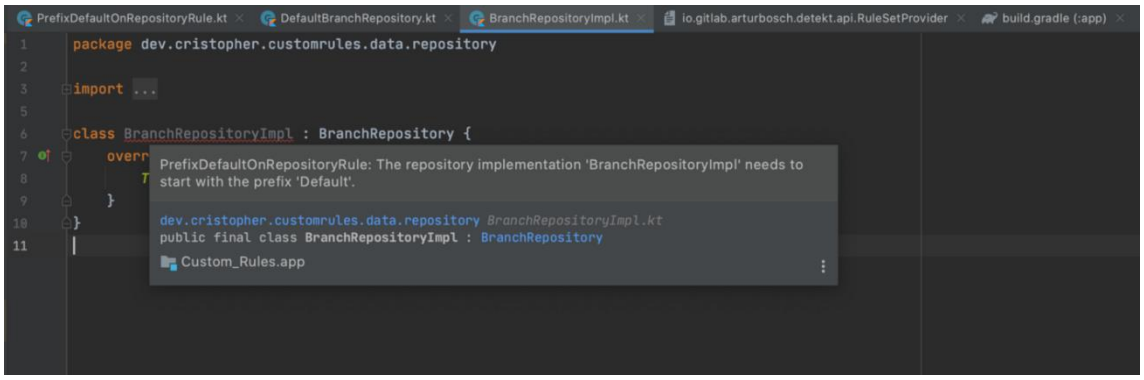
7. I also highly recommend to install this plugin so you can be notified in the same Android Studio about any errors found.



To see your custom rules in Android Studio you need to generate a jar from your module and add that path in the external rulset JARs like this:



And that's it! Now you can see your custom rule applied 🎉



https://proandroiddev.com/static-analysis-tools-for-android-9531334954f6

Dynamic Code Analysis

- Android Hooker Hooker is an opensource project for dynamic analyses of Android applications. This project provides various tools and applications that can be use to automatically intercept and modify any API calls made by a targeted application.It leverages Android Substrate framework to intercept these calls and aggregate all their contextual information (parameters, returned values, …). Collected information can either be stored in a ElasticSearch or in JSON files.

A set of python scripts is also provided to automatize the execution of an analysis to collect any API calls made by a set of applications.

- AppAuditOnline tool (including an API) uses dynamic and static analysis to detect hidden data leaks in an application .
- BareDroidBareDroid allows for bare-metal analysis on Android devices. See the paper here
- CuckooDroidCuckooDroid is an extension of Cuckoo Sandbox the Open Source software for automating analysis of suspicious files, CuckooDroid brigs to cuckoo the capabilities of execution and analysis of android application.
- DroidboxDroidBox is developed to offer dynamic analysis of Android applications. The following information is described in the results, generated when analysis is complete:
    - Hashes for the analyzed package
    - Incoming/outgoing network data
    - File read and write operations
    - Started services and loaded classes through DexClassLoader
    - Information leaks via the network, file and SMS
    - Circumvented permissions
    - Cryptographic operations performed using Android API

- Listing broadcast receivers
- Sent SMS and phone calls

Additionally, two graphs are generated visualizing the behavior of the package. One showing the temporal order of the operations and the other one being a treemap that can be used to check similarity between analyzed packages.

- Droid-FFDroid-FF is an Android File Fuzzing Framework
- Drozerdrozer helps to provide confidence that Android apps and devices being developed by, or deployed across, your organisation do not pose an unacceptable level of risk. By allowing you to interact with the Dalvik VM, other apps' IPC endpoints and the underlying OS.

drozer provides tools to help you use and share public exploits for Android. For remote exploits, it can generate shellcode to help you to deploy the drozer Agent as a remote administrator tool, with maximum leverage on the device.

- MarvinMarvin is a system that analyzes Android applications in search of vulnerabilities and allows tracking of an app through its version history.It is composed of 4 subsystems:
  - Marvin-django: The web application frontend for use and adminsitration of Marvin (this repostory). It includes a bayesian classifier that provides a probability estimation of a given Android app being malware.
  - Marvin-static-Analyzer: A static code analysis system that uses Androguard and Static Android Analysis Framework (SAAF).
  - Marvin-dynamic-Analyzer: A dynamic code analysis system that uses Android x86-emulators and Open Nebula virtualization to find vulnerabilities automatically
  - Marvin-toqueton: An automated GUI testing tool developed to assist Marvin's dynamic code analysis.

A Marvin user's guide is provided in the docs folder of this repository.

- InspeckageInspeckage is a tool developed to offer dynamic analysis of Android applications. By applying hooks to functions of the Android API, Inspeckage will help you understand what an Android application is doing at runtime.
- PATDroidPATDroid is a collection of tools and data structures for analyzing Android applications and the system itself. We intend to build it as a common base for developing novel mobile software debugging, refactoring, reverse engineering tools.

https://securityonline.info/android-arsenal-dynamic-analysis-tools/

A main topic of my work is Android security, mostly from an attacker's view. Android devices are ubiquitous and the operating system has a big market share, so it is one of the most interesting systems to research. Since I also do pentesting, I often look at Android applications from a security standpoint, where developers often do serious mistakes, which can lead to data breaches, misuse and more.
To inspect an app, you often take two approaches: **static** and **dynamic** analysis. Static means

that you investigate the binary itself, the program code and the metadata that comes with it, while dynamic means investigating the execution of the code. In order to get a holistic view of the app, you can not limit yourself to one of these approaches, since each yields different insights.

As an example, static analysis can tell much about the circumstances an app will run:

- Which devices are targeted?
- What are the software dependencies?
- Does it contain suspicious code?

Meanwhile, a dynamic analysis might answer the questions:

- What does the app actually do?
- Is suspicious code executed?
- What data is sent and processed?

In this post, I will be looking at static approaches specific to Android applications, but similar methods are used for other systems.

The Android Package Kit APK

An Android binary, the actual file that is installed on a device, is a ZIP archive that was renamed to the extension apk. Therefore, if you talk about an app, you actually mean a whole bunch of files packed into one single executable. This APK has a number of properties:

- It contains all the **application code** that is loaded from .dex files (more on that later)
- It contains all the **resources** (images, icons, layouts) that are used inside the app
- It contains the app's **configuration**, which includes permissions
- It is **cryptographically signed** such that modifications to the package are detected by the OS
- Often it contains information about the **build process**

In most cases, whatever happens in an app is somehow included in the APK file. Now, you can start analyzing the package by extracting all files, but you will notice that some files are compressed or otherwise obfuscated. For example, the AndroidManifest.xml file, which contains essential information about an app in XML notation, is in a binary format and unreadable. A similar problem is encountered with classes.dex files, which contain the actual bytecode to be executed by the operating system (more precisely: the **Dalvik** Virtual Machine) on a device.

A short note on the DEX format: dex files contain the binary code to be run by Dalvik, Android's implementation of the Java Virtual Machine. So they are very similar to .class files in Java (and can also be converted to actual Java bytecode).

So the first step of static analysis is to make these files readable, and we are going to use Apktool for this.

Where to get the APK

You might ask how to even get the APK of an app. First of all, if you use the Google Play Store, there is no option to download an APK file. If you are interested in downloading them directly from Google, search for some “Play Store crawler” software out there, there are some on GitHub.
If you want to pull the APK from an installed app, check this Stack Overflow answer.

If you are targeting a specific app, check alternative app stores for it. I do not have any suggestions here, simply turn on your favourite search engine and type “APK” to find downloads. Note that there are not many trustworthy **app stores** out there, so **never install** an APK on your real device, or you **might infect** your phone.
A little more trustworthy is F-Droid, which also allows to download APK files and the apps on there are Free and Open-Source, so there is some way to check the source code if you are unsure.

Apktool

Apktool is very simple to use, but also very powerful. It has two main functionalities: **decode** and **build**.
Decoding is for making APK data readable (and modifiable), while building is for transforming the decoded files into an APK file again. For this, *Apktool* makes use of *smali*, a disassembler for DEX files. This turns the actual Dalvik bytecode into an intermediate representation (smali code) that can be modified. With *baksmali* you can reassemble smali code into DEX files, effectively allowing modifications to app code.

Disassembling an APK s pretty simple:

apktool decode -o output_dir application.apk

Here is an example output of smali code of a MainActivity class, as generated by *Apktool*:

.class public Lcom/example/ui/MainActivity;

.super Lcom/example/ui/BaseActivity;

.source "Application"

interfaces

.implements Lx/lz;

.implements Lx/ma;

.implements Lx/mo;

.implements Lx/mp;

instance fields

.field private m:Ljava/util/HashMap;

.annotation system Ldalvik/annotation/Signature;

value = {

```
        "Ljava/util/HashMap<",
        "Ljava/lang/Integer;",
        "Lorg/apache/cordova/CordovaPlugin;",
        ">;"
    }
  .end annotation
.end field
```

Since this notation is easier to read and modify for humans compared to bytecode, it is the preferred format for app analysis. You could do more, like using a **decompiler** to generate actual Java code, but decompilation is often **considered unethical** and some countries even outlaw it. More often than not, you have license agreements that disallow decompilation, so doing it nonetheless is a **breach of contract** and an analyst might get into legal trouble for that. If you are pentesting a mobile application, you should **get explicit permission** for this or even ask for the actual source code, depending on the situation.

So, once *Apktool* has done its job, you get a file structure like this:

```
output_dir/
├── AndroidManifest.xml
├── apktool.yml
├── assets
├── lib
├── original
├── res
├── smali
├── smali_classes2
└── unknown
```

https://davidebove.com/blog/2019/10/10/a-beginners-guide-to-static-analysis-of-android-apps/

Android PenTest Tricks (eMAPT Exam)

https://book.hacktricks.xyz/mobile-apps-pentesting/android-app-pentesting

Mobile Application Penetration Testing Cheat Sheet

The Mobile App Pentest cheat sheet was created to provide concise collection of high value information on specific mobile application penetration testing topics and checklist, which is mapped OWASP Mobile Risk Top 10 for conducting pentest.

- Mobile Application Security Testing Distributions

Mobile Application Security Testing Distributions

- Appie - A portable software package for Android Pentesting and an awesome alternative to existing Virtual machines.
- Android Tamer - Android Tamer is a Virtual / Live Platform for Android Security professionals.
- Androl4b - A Virtual Machine For Assessing Android applications, Reverse Engineering and Malware Analysis
- Vezir Project - Mobile Application Pentesting and Malware Analysis Environment.
- Mobexler - Mobexler is a customised virtual machine, designed to help in penetration testing of Android & iOS applications.

All-in-One Mobile Security Frameworks

- Mobile Security Framework - MobSF - Mobile Security Framework is an intelligent, all-in-one open source mobile application (Android/iOS) automated pen-testing framework capable of performing static and dynamic analysis.
    - python manage.py runserver 127.0.0.1:1337

- Needle - Needle is an open source, modular framework to streamline the process of conducting security assessments of iOS apps including Binary Analysis, Static Code Analysis, Runtime Manipulation using Cycript and Frida hooking, and so on.
- Objection - Objection is a runtime mobile exploration toolkit, powered by Frida. It was built with the aim of helping assess mobile applications and their security posture without the need for a jailbroken or rooted mobile device.
- RMS-Runtime-Mobile-Security - Runtime Mobile Security (RMS), powered by FRIDA, is a powerful web interface that helps you to manipulate Android and iOS Apps at Runtime.

Android Application Penetration Testing

Reverse Engineering and Static Analysis

- APKTool - A tool for reverse engineering 3rd party, closed, binary Android apps. It can decode resources to nearly original form and rebuild them after making some modifications.
    - Disassembling Android apk file
        - apktool d <apk file>
    - Rebuilding decoded resources back to binary APK/JAR with certificate signing
        - apktool b <modified folder>
        - keytool -genkey -v -keystore keys/test.keystore -alias Test -keyalg RSA -keysize 1024 -sigalg SHA1withRSA -validity 10000
        - jarsigner -keystore keys/test.keystore dist/test.apk -sigalg SHA1withRSA -digestalg SHA1 Test
- Bytecode Viewer - Bytecode Viewer is an Advanced Lightweight Java Bytecode Viewer, It's written completely in Java, and it's open sourced.
- Jadx - Dex to Java decompiler: Command line and GUI tools for produce Java source code from Android Dex and Apk files.
- APK Studio - Open-source, cross platform Qt based IDE for reverse-engineering Android application packages.
- Oat2dex - A tool for converting .oat file to .dex files.
    - Deoptimize boot classes (The output will be in "odex" and "dex" folders)
        - java -jar oat2dex.jar boot <boot.oat file>
    - Deoptimize application
        - java -jar oat2dex.jar <app.odex> <boot-class-folder output from above>
    - Get odex from oat
        - java -jar oat2dex.jar odex <oat file>

    - Get odex smali (with optimized opcode) from oat/odex
        - java -jar oat2dex.jar smali <oat/odex file>

- FindBugs + FindSecurityBugs - FindSecurityBugs is a extension for FindBugs which include security rules for Java applications.
- Qark - This tool is designed to look for several security related Android application vulnerabilities, either in source code or packaged APKs.
- SUPER - SUPER is a command-line application that can be used in Windows, MacOS X and Linux, that analyzes .apk files in search for vulnerabilities. It does this by decompressing APKs and applying a series of rules to detect those vulnerabilities.
- AndroBugs - AndroBugs Framework is an efficient Android vulnerability scanner that helps developers or hackers find potential security vulnerabilities in Android applications. No need to install on Windows.
- Simplify - A tool for de-obfuscating android package into Classes.dex which can be use Dex2jar and JD-GUI to extract contents of dex file.
    - simplify.jar -i "input smali files or folder" -o <output dex file>
- ClassNameDeobfuscator - Simple script to parse through the .smali files produced by apktool and extract the .source annotation lines.
- Android backup extractor - Utility to extract and repack Android backups created with adb backup (ICS+). Largely based on BackupManagerService.java from AOSP. Tip !! "adb backup" command can also be used for extracting application package with the following command:
    - adb backup <package name>
    - dd if=backup.ab bs=1 skip=24 | python -c "import zlib,sys;sys.stdout.write(zlib.decompress(sys.stdin.read()))" > backup.tar
- GDA(GJoy Dex Analysizer) - GDA, a new Dalvik bytecode decompiler, is implemented in C++, which has the advantages of faster analysis and lower memory&disk consumption and an stronger ability to decompiling the APK, DEX, ODEX, OAT files(supports JAR, CLASS and AAR files since 3.79)

Dynamic and Runtime Analysis

- Cydia Substrate - Cydia Substrate for Android enables developers to make changes to existing software with Substrate extensions that are injected in to the target process's memory.
- Xposed Framework - Xposed framework enables you to modify the system or application aspect and behaviour at runtime, without modifying any Android application package(APK) or re-flashing.
- PID Cat - An update to Jeff Sharkey's excellent logcat color script which only shows log entries for processes from a specific application package.

- Inspeckage - Inspeckage is a tool developed to offer dynamic analysis of Android applications. By applying hooks to functions of the Android API, Inspeckage will help you understand what an Android application is doing at runtime.
- Frida - The toolkit works using a client-server model and lets you inject in to running processes not just on Android, but also on iOS, Windows and Mac.
- Diff-GUI - A Web framework to start instrumenting with the avaliable modules, hooking on native, inject JavaScript using Frida.
- Fridump - Fridump is using the Frida framework to dump accessible memory addresses from any platform supported. It can be used from a Windows, Linux or Mac OS X system to dump the memory of an iOS, Android or Windows application.
- House - A runtime mobile application analysis toolkit with a Web GUI, powered by Frida, is designed for helping assess mobile applications by implementing dynamic function hooking and intercepting and intended to make Frida script writing as simple as possible.
- AndBug - AndBug is a debugger targeting the Android platform's Dalvik virtual machine intended for reverse engineers and developers.
    - Identifying application process using adb shell
        - adb shell ps | grep -i "App keyword"
    - Accessing the application using AndBug in order to identify loaded classes
        - andbug shell -p <process number>
    - Tracing specific class
        - ct <package name>
    - Debugging with jdb
        - adb forward tcp:<port> jdwp:<port>
        - jdb -attach localhost:<port>
- Cydia Substrate: Introspy-Android - Blackbox tool to help understand what an Android application is doing at runtime and assist in the identification of potential security issues.
- Drozer - Drozer allows you to search for security vulnerabilities in apps and devices by assuming the role of an app and interacting with the Dalvik VM, other apps' IPC endpoints and the underlying OS.
    - Starting a session
        - adb forward tcp:31415 tcp:31415
        - drozer console connect
    - Retrieving package information
        - run app.package.list -f <app name>

        - run app.package.info -a <package name>
    - Identifying the attack surface
        - run app.package.attacksurface <package name>
    - Exploiting Activities
        - run app.activity.info -a <package name> -u
        - run app.activity.start --component <package name> <component name>
    - Exploiting Content Provider
        - run app.provider.info -a <package name>
        - run scanner.provider.finduris -a <package name>
        - run app.provider.query <uri>
        - run app.provider.update <uri> --selection <conditions> <selection arg> <column> <data>
        - run scanner.provider.sqltables -a <package name>
        - run scanner.provider.injection -a <package name>
        - run scanner.provider.traversal -a <package name>
    - Exploiting Broadcast Receivers
        - run app.broadcast.info -a <package name>
        - run app.broadcast.send --component <package name> <component name> --extra <type> <key> <value>
        - run app.broadcast.sniff --action <action>
    - Exploiting Service
        - run app.service.info -a <package name>
        - run app.service.start --action <action> --component <package name> <component name>
        - run app.service.send <package name> <component name> --msg <what> <arg1> <arg2> --extra <type> <key> <value> --bundle-as-obj

Network Analysis and Server Side Testing

- Tcpdump - A command line packet capture utility.
- Wireshark - An open-source packet analyzer.
    - Live packet captures in real time
        - adb shell "tcpdump -s 0 -w - | nc -l -p 4444"
        - adb forward tcp:4444 tcp:4444

        - nc localhost 4444 | sudo wireshark -k -S -i –
- Canape - A network testing tool for arbitrary protocols.
- Mallory - A Man in The Middle Tool (MiTM) that use to monitor and manipulate traffic on mobile devices and applications.
- Burp Suite - Burp Suite is an integrated platform for performing security testing of applications.
    - Installing trusted CA at the Android OS level (Root device/Emulator) for Android N+ as the following:
        - openssl x509 -inform PEM -subject_hash -in BurpCA.pem | head -1
        - cat BurpCA.pem > 9a5ba580.0
        - openssl x509 -inform PEM -text -in BurpCA.pem -out /dev/null >> 9a5ba580.0
        - adb root
        - abd remount
        - adb push 9a5ba580.0 /system/etc/security/cacerts/
        - adb shell “chmod 644 /system/etc/security/cacerts/9a5ba580.0”
        - adb shell “reboot”
        - Check Settings > Security > Trusted Credentials > SYSTEM to confirm your newly added CA is listed.
- Burp Suite Mobile Assistant - Burp Suite Mobile Assistant is a tool to facilitate testing of iOS apps with Burp Suite; It can modify the system-wide proxy settings of iOS devices so that HTTP(S) traffic can be easily redirected to a running instance of Burp, It can attempt to circumvent SSL certificate pinning in selected apps, allowing Burp Suite to break their HTTPS connections and intercept, inspect and modify all traffic.
- OWASP ZAP - OWASP Zed Attack Proxy Project is an open-source web application security scanner. It is intended to be used by both those new to application security as well as professional penetration testers.
- Proxydroid - Global Proxy App for Android System.
- mitmproxy - is an interactive, SSL/TLS-capable intercepting proxy with a console interface for HTTP/1, HTTP/2, and WebSockets.

Bypassing Root Detection and SSL Pinning

- Magisk - Magisk suites provide root access to your device, capability to modify read-only partitions by installing modules and hide Magisk from root detections/system integrity checks.
- Xposed Module: Just Trust Me - Xposed Module to bypass SSL certificate pinning.

- Xposed Module: SSLUnpinning - Android Xposed Module to bypass SSL certificate validation (Certificate Pinning).
- Cydia Substrate Module: Android SSL Trust Killer - Blackbox tool to bypass SSL certificate pinning for most applications running on a device.
- Cydia Substrate Module: RootCoak Plus - Patch root checking for commonly known indications of root.
- Android-ssl-bypass - an Android debugging tool that can be used for bypassing SSL, even when certificate pinning is implemented, as well as other debugging tasks. The tool runs as an interactive console.
- Apk-mitm - A CLI application that automatically prepares Android APK files for HTTPS inspection
- 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.
    - Bypassing Root Detection
        - frida --codeshare dzonerzy/fridantiroot -f YOUR_BINARY
    - Bypassing SSL Pinning
        - frida --codeshare pcipolloni/universal-android-ssl-pinning-bypass-with-frida -f YOUR_BINARY

Security Libraries

- PublicKey Pinning - Pinning in Android can be accomplished through a custom X509TrustManager. X509TrustManager should perform the customary X509 checks in addition to performing the pinning configuration.
- Android Pinning - A standalone library project for certificate pinning on Android.
- Java AES Crypto - A simple Android class for encrypting & decrypting strings, aiming to avoid the classic mistakes that most such classes suffer from.
- Proguard - ProGuard is a free Java class file shrinker, optimizer, obfuscator, and preverifier. It detects and removes unused classes, fields, methods, and attributes.
- SQL Cipher - SQLCipher is an open source extension to SQLite that provides transparent 256-bit AES encryption of database files.
- Secure Preferences - Android Shared preference wrapper than encrypts the keys and values of Shared Preferences.
- Trusted Intents - Library for flexible trusted interactions between Android apps.
- RootBeer - A tasty root checker library and sample app.
- End-to-end encryption - Capillary is a library to simplify the sending of end-to-end encrypted push messages from Java-based application servers to Android clients.

iOS Application Penetration Testing

Access Filesystem on iDevice

- FileZilla - It supports FTP, SFTP, and FTPS (FTP over SSL/TLS).
- Cyberduck - Libre FTP, SFTP, WebDAV, S3, Azure & OpenStack Swift browser for Mac and Windows.
- itunnel - Use to forward SSH via USB.
- iProxy - Let's you connect your laptop to the iPhone to surf the web.
- iFunbox - The File and App Management Tool for iPhone, iPad & iPod Touch.

Reverse Engineering and Static Analysis

- otool - The otool command displays specified parts of object files or libraries.
- Clutch - Decrypted the application and dump specified bundleID into binary or .ipa file.
- Dumpdecrypted - Dumps decrypted mach-o files from encrypted iPhone applications from memory to disk. This tool is necessary for security researchers to be able to look under the hood of encryption.
    - iPod:~ root# DYLD_INSERT_LIBRARIES=dumpdecrypted.dylib /var/mobile/Applications/xxxxxxxx-xxxx-xxxx-xxxx-xxxxxxxxxxxx/Scan.app/Scan
- class-dump - A command-line utility for examining the Objective-C runtime information stored in Mach-O files.
- dsdump - An improved nm + objc/swift class-dump.
- Weak Classdump - A Cycript script that generates a header file for the class passed to the function. Most useful when you cannot classdump or dumpdecrypted , when binaries are encrypted etc.
    - iPod:~ root# cycript -p Skype weak_classdump.cy; cycript -p Skype
    - #cy weak_classdump_bundle([NSBundle mainBundle],"/tmp/Skype")
- Fridpa - An automated wrapper script for patching iOS applications (IPA files) and work on non-jailbroken device.
- Frida-iOS-Dump - Pull a decrypted IPA from a jailbroken device.
- bagbak - Yet another frida based iOS dumpdecrypted, supports decrypting app extensions and no SSH required.
- bfinject - bfinject loads arbitrary dylibs into running App Store apps. It has built-in support for decrypting App Store apps, and comes bundled with iSpy and Cycript.
    - A Simple Test
        - bash bfinject -P Reddit -L test
    - Decrypt App Store apps
        - bash bfinject -P Reddit -L decrypt
    - Cycript

        - bash bfinject -P Reddit -L cycript

- HopperApp - Hopper is a reverse engineering tool for OS X and Linux, that lets you disassemble, decompile and debug your 32/64bits Intel Mac, Linux, Windows and iOS executables.
- hopperscripts - Hopperscripts can be used to demangle the Swift function name in HopperApp.
- Radare2 - Radare2 is a unix-like reverse engineering framework and commandline tools.
- XReSign - XReSign allows you to sign or resign unencrypted ipa-files with certificate for which you hold the corresponding private key. Checked for developer, ad-hoc and enterprise distribution.

Dynamic and Runtime Analysis

- cycript - Cycript allows developers to explore and modify running applications on either iOS or Mac OS X using a hybrid of Objective-C++ and JavaScript syntax through an interactive console that features syntax highlighting and tab completion.
    - Show currently visible view controller
        - cy# UIApp.keyWindow.rootViewController.visibleViewController
    - Show view controller at the top of the navigation stack
        - cy# UIApp.keyWindow.rootViewController.topViewController
    - Get an array of existing objects of a certain class
        - cy# choose(UIViewController)
    - UI Dump,cuts off lots of descriptions of UIViews
        - cy# [[UIApp keyWindow] _autolayoutTrace].toString()
    - Skip UIViews and nextResponders to get ViewControllers directly
        - cy# [[[UIApp keyWindow] rootViewController] _printHierarchy].toString()
    - List method at runtime
        - cy# classname.messages or cy# function printMethods(className, isa) { var count = new new Type("I"); var classObj = (isa != undefined) ? objc_getClass(className)->isa : objc_getClass(className); var methods = class_copyMethodList(classObj, count); var methodsArray = []; for(var i = 0; i < *count; i++) { var method = methods[i]; methodsArray.push({selector:method_getName(method), implementation:method_getImplementation(method)}); } free(methods); return methodsArray; }
    -
        - cy# printMethods("<classname>")

    - Prints out all the instance variables
        - cy# a=#0x15d0db80
        - cy# *a or
        - cy# function tryPrintIvars(a){ var x={}; for(i in *a){ try{ x[i] = (*a)[i]; } catch(e){} } return x; }
        - cy# a=#0x15d0db80
        - cy# tryPrintIvars(a)
    - Manipulating through property
        - cy# [a pinCode]
        - cy# [a setPinCode: @"1234"] or cy# a.setPinCode= @"1234"
    - Method Swizzling for Instance Method
        - cy# [a isValidPin]
        - cy# <classname>.prototype.isValidPin = function(){return 1;}
    - Method Swizzling for Class Method
        - cy# [Pin isValidPin]
        - cy# Pin.contructor.prototype.['isValidPin'] = function(){return 1;}

- iNalyzer - AppSec Labs iNalyzer is a framework for manipulating iOS applications, tampering with parameters and method.
- Passionfruit - Simple iOS app blackbox assessment tool with Fully web based GUI. Powered by frida.re and vuejs.
- Introspy-iOS - Blackbox tool to help understand what an iOS application is doing at runtime and assist in the identification of potential security issues.
- Apple configurator 2 - A utility which can be used to view live system log on iDevice.
- keychaindumper - A tool to check which keychain items are available to an attacker once an iOS device has been jailbroken.
- BinaryCookieReader - A tool to dump all the cookies from the binary Cookies.binarycookies file.

Network Analysis and Server Side Testing

- Canape - A network testing tool for arbitrary protocols.
- Mallory - A Man in The Middle Tool (MiTM) that use to monitor and manipulate traffic on mobile devices and applications.
- Burp Suite - Burp Suite is an integrated platform for performing security testing of applications.

- OWASP ZAP - OWASP Zed Attack Proxy Project is an open-source web application security scanner. It is intended to be used by both those new to application security as well as professional penetration testers.
- Charles Proxy - HTTP proxy / HTTP monitor / Reverse Proxy that enables a developer to view all of the HTTP and SSL / HTTPS traffic between their machine and the Internet.

Bypassing Root Detection and SSL Pinning

- SSL Kill Switch 2 - Blackbox tool to disable SSL certificate validation - including certificate pinning - within iOS and OS X Apps.
- iOS TrustMe - Disable certificate trust checks on iOS devices.
- tsProtector - Another tool for bypassing Jailbreak detection.
- JailProtect - Apart from bypassing jailbreak detection, it also allows you to spoof your iOS firmware version easily.
- Shadow - Shadow is a tweak to bypass jailbreak detection that defeats basic detection methods used by many App Store apps.
- 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.
    - Bypassing SSL Pinning
        - frida --codeshare lichao890427/ios-ssl-bypass -f YOUR_BINARY
        - frida --codeshare dki/ios10-ssl-bypass -f YOUR_BINARY

Security Libraries

- PublicKey Pinning - iOS pinning is performed through a NSURLConnectionDelegate. The delegate must implement connection:canAuthenticateAgainstProtectionSpace: and connection:didReceiveAuthenticationChallenge:. Within connection:didReceiveAuthenticationChallenge:, the delegate must call SecTrustEvaluate to perform customary X509 checks.
- Swiftshield - SwiftShield is a tool that generates irreversible, encrypted names for your iOS project's objects (including your Pods and Storyboards) in order to protect your app from tools that reverse engineer iOS apps, like class-dump and Cycript.
- IOSSecuritySuite - iOS Security Suite is an advanced and easy-to-use platform security & anti-tampering library written in pure Swift! If you are developing for iOS and you want to protect your app according to the OWASP MASVS standard, chapter v8, then this library could save you a lot of time.
- OWASP iMAS - iMAS is a collaborative research project from the MITRE Corporation focused on open source iOS security controls.

Mobile Penetration Testing Lab

- WaTF Bank - What-a-Terrible-Failure Mobile Banking Application (WaTF-Bank), written in Java, Swift 4, Objective-C and Python (Flask framework) as a backend server, is

designed to simulate a "real-world" web services-enabled mobile banking application that contains over 30 vulnerabilities based on OWASP Mobile Top 10 Risks.

- InsecureBankv2 - WThis vulnerable Android application is named "InsecureBankv2" and is made for security enthusiasts and developers to learn the Android insecurities by testing this vulnerable application. Its back-end server component is written in python.
- DVIA-v2 - Damn Vulnerable iOS App (DVIA) is an iOS application that is damn vulnerable. Its main goal is to provide a platform to mobile security enthusiasts/professionals or students to test their iOS penetration testing skills in a legal environment.
- DIVA Android - DIVA (Damn insecure and vulnerable App) is an App intentionally designed to be insecure.The aim of the App is to teach developers/QA/security professionals, flaws that are generally present in the Apps due poor or insecure coding practices.
- DVHMA - Damn Vulnerable Hybrid Mobile App (DVHMA) is an hybrid mobile app (for Android) that intentionally contains vulnerabilities. Its purpose is to enable security professionals to test their tools and techniques legally, help developers better understand the common pitfalls in developing hybrid mobile apps securely.
- MSTG Hacking Playground - This is a collection of iOS and Android mobile apps, that are intentionally build insecure. These apps are used as examples to demonstrate different vulnerabilities explained in the the OWASP Mobile Security Testing Guide.
- UnCrackable Mobile Apps - UnCrackable Apps for Android and iOS, a collection of mobile reverse engineering challenges. These challenges are used as examples throughout the Mobile Security Testing Guide.
- OWASP iGoat - Goat is a learning tool for iOS developers (iPhone, iPad, etc.) and mobile app pentesters. It was inspired by the WebGoat project, and has a similar conceptual flow to it.

https://github.com/tanprathan/MobileApp-Pentest-Cheatsheet/blob/master/README.md

Android

General - Blogs, Papers, How To's

- Android: Gaining access to arbitrary* Content Providers
- Evernote: Universal-XSS, theft of all cookies from all sites, and more
- Interception of Android implicit intents
- TikTok: three persistent arbitrary code executions and one theft of arbitrary files
- Persistent arbitrary code execution in Android's Google Play Core Library: details, explanation and the PoC - CVE-2020-8913
- Android: Access to app protected components
- Android: arbitrary code execution via third-party package contexts

- Android Pentesting Labs - Step by Step guide for beginners
- An Android Hacking Primer
- Secure an Android Device
- Security tips
- OWASP Mobile Security Testing Guide
- Security Testing for Android Cross Platform Application
- Dive deep into Android Application Security
- Pentesting Android Apps Using Frida
- Mobile Security Testing Guide
- Mobile Application Penetration Testing Cheat Sheet
- Android Applications Reversing 101
- Android Security Guidelines
- Android WebView Vulnerabilities
- OWASP Mobile Top 10
- Practical Android Phone Forensics
- Mobile Reverse Engineering Unleashed
- Android Root Detection Bypass Using Objection and Frida Scripts
- quark-engine - An Obfuscation-Neglect Android Malware Scoring System
- Root Detection Bypass By Manual Code Manipulation.
- Application and Network Usage in Android
- GEOST BOTNET - the discovery story of a new Android banking trojan
- Mobile Pentesting With Frida
- Magisk Systemless Root - Detection and Remediation
- AndrODet: An adaptive Android obfuscation detector
- Hands On Mobile API Security
- Zero to Hero - Mobile Application Testing - Android Platform
- How to use FRIDA to bruteforce Secure Startup with FDE-encryption on a Samsung G935F running Android 8
- Android Malware Adventures
- AAPG - Android application penetration testing guide
- Bypassing Android Anti-Emulation

- Bypassing Xamarin Certificate Pinning
- Configuring Burp Suite With Android Nougat

Books

- SEI CERT Android Secure Coding Standard
- Android Security Internals
- Android Cookbook
- Android Hacker's Handbook
- Android Security Cookbook
- The Mobile Application Hacker's Handbook
- Android Malware and Analysis
- Android Security: Attacks and Defenses

Courses

- Learning-Android-Security
- Mobile Application Security and Penetration Testing
- Advanced Android Development
- Learn the art of mobile app development
- Learning Android Malware Analysis
- Android App Reverse Engineering 101
- Android Pentesting for Beginners

Tools

Static Analysis

- Amandroid – A Static Analysis Framework
- Androwarn – Yet Another Static Code Analyzer
- APK Analyzer – Static and Virtual Analysis Tool
- APK Inspector – A Powerful GUI Tool
- Droid Hunter – Android application vulnerability analysis and Android pentest tool
- Error Prone – Static Analysis Tool
- Findbugs – Find Bugs in Java Programs
- Find Security Bugs – A SpotBugs plugin for security audits of Java web applications.
- Flow Droid – Static Data Flow Tracker
- Smali/Baksmali – Assembler/Disassembler for the dex format

- Smali-CFGs – Smali Control Flow Graph's
- SPARTA – Static Program Analysis for Reliable Trusted Apps
- Thresher – To check heap reachability properties
- Vector Attack Scanner – To search vulnerable points to attack
- Gradle Static Analysis Plugin
- Checkstyle – A tool for checking Java source code
- PMD – An extensible multilanguage static code analyzer
- Soot – A Java Optimization Framework
- Android Quality Starter
- QARK – Quick Android Review Kit
- Infer – A Static Analysis tool for Java, C, C++ and Objective-C
- Android Check – Static Code analysis plugin for Android Project
- FindBugs-IDEA Static byte code analysis to look for bugs in Java code
- APK Leaks – Scanning APK file for URIs, endpoints & secrets

Dynamic Analysis

- Adhrit - Android Security Suite for in-depth reconnaissance and static bytecode analysis based on Ghera benchmarks
- Android Hooker - Opensource project for dynamic analyses of Android applications
- AppAudit - Online tool (including an API) uses dynamic and static analysis
- AppAudit - A bare-metal analysis tool on Android devices
- CuckooDroid - Extension of Cuckoo Sandbox the Open Source software
- DroidBox - Dynamic analysis of Android applications
- Droid-FF - Android File Fuzzing Framework
- Drozer
- Marvin - Analyzes Android applications and allows tracking of an app
- Inspeckage
- PATDroid - Collection of tools and data structures for analyzing Android applications
- AndroL4b - Android security virtual machine based on ubuntu-mate
- Radare2 - Unix-like reverse engineering framework and commandline tools
- Cutter - Free and Open Source RE Platform powered by radare2
- ByteCodeViewer - Android APK Reverse Engineering Suite (Decompiler, Editor, Debugger)

- Mobile-Security-Framework MobSF
- CobraDroid - Custom build of the Android operating system geared specifically for application security
- Magisk v20.2 - Root & Universal Systemless Interface
- Runtime Mobile Security (RMS) - is a powerful web interface that helps you to manipulate Android and iOS Apps at Runtime
- MOBEXLER - A Mobile Application Penetration Testing Platform

Android Online APK Analyzers

- Oversecured - A static vulnerability scanner for Android apps (APK files) containing 90+ vulnerability categories
- Android Observatory APK Scan
- Android APK Decompiler
- AndroTotal
- NVISO ApkScan
- VirusTotal
- Scan Your APK
- AVC Undroid
- OPSWAT
- ImmuniWeb Mobile App Scanner
- Ostor Lab
- Quixxi
- TraceDroid
- Visual Threat
- App Critique

Labs

- OVAA (Oversecured Vulnerable Android App)
- DIVA (Damn insecure and vulnerable App)
- SecurityShepherd
- Damn Vulnerable Hybrid Mobile App (DVHMA)
- OWASP-mstg
- VulnerableAndroidAppOracle
- Android InsecureBankv2

- Purposefully Insecure and Vulnerable Android Application (PIIVA)
- Sieve app
- DodoVulnerableBank
- Digitalbank
- OWASP GoatDroid
- AppKnox Vulnerable Application
- Vulnerable Android Application
- MoshZuk
- Hackme Bank
- Android Security Labs
- Android-InsecureBankv2
- Android-security
- VulnDroid
- FridaLab
- Santoku Linux - Mobile Security VM
- Vuldroid

Talks

- Blowing the Cover of Android Binary Fuzzing (Slides)
- One Step Ahead of Cheaters -- Instrumenting Android Emulators
- Vulnerable Out of the Box: An Evaluation of Android Carrier Devices
- Rock appround the clock: Tracking malware developers by Android
- Chaosdata - Ghost in the Droid: Possessing Android Applications with ParaSpectre
- Remotely Compromising Android and iOS via a Bug in Broadcom's Wi-Fi Chipsets
- Honey, I Shrunk the Attack Surface – Adventures in Android Security Hardening
- Hide Android Applications in Images
- Scary Code in the Heart of Android
- Fuzzing Android: A Recipe For Uncovering Vulnerabilities Inside System Components In Android
- Unpacking the Packed Unpacker: Reverse Engineering an Android Anti-Analysis Native Library
- Android FakeID Vulnerability Walkthrough
- Unleashing D* on Android Kernel Drivers

- The Smarts Behind Hacking Dumb Devices
- Overview of common Android app vulnerabilities
- Android Dev Summit 2019
- Android security architecture
- Get the Ultimate Privilege of Android Phone

Misc.

- Android-Reports-and-Resources
- android-security-awesome
- Android Penetration Testing Courses
- Lesser-known Tools for Android Application PenTesting
- android-device-check - a set of scripts to check Android device security configuration
- apk-mitm - a CLI application that prepares Android APK files for HTTPS inspection
- Andriller - is software utility with a collection of forensic tools for smartphones
- Dexofuzzy: Android malware similarity clustering method using opcode sequence-Paper
- Chasing the Joker
- Side Channel Attacks in 4G and 5G Cellular Networks-Slides
- Shodan.io-mobile-app for Android
- Popular Android Malware 2018
- Popular Android Malware 2019
- Popular Android Malware 2020

iOS

General - Blogs, Papers, How to's

- iOS Security
- Basic iOS Apps Security Testing lab
- IOS Application security – Setting up a mobile pentesting platform
- Collection of the most common vulnerabilities found in iOS applications
- IOS_Application_Security_Testing_Cheat_Sheet
- OWASP iOS Basic Security Testing
- Dynamic analysis of iOS apps w/o Jailbreak
- iOS Application Injection

- Low-Hanging Apples: Hunting Credentials and Secrets in iOS Apps
- Checkra1n Era - series
- BFU Extraction: Forensic Analysis of Locked and Disabled iPhones
- HowTo-decrypt-Signal.sqlite-for-IOS
- Can I Jailbreak?
- How to Extract Screen Time Passcodes and Voice Memos from iCloud
- Reverse Engineering Swift Apps
- Mettle your iOS with FRIDA
- A run-time approach for pentesting iOS applications
- iOS Internals vol 2
- Understanding usbmux and the iOS lockdown service
- A Deep Dive into iOS Code Signing
- AirDoS: remotely render any nearby iPhone or iPad unusable
- How to access and traverse a #checkra1n jailbroken iPhone File system using SSH
- Deep dive into iOS Exploit chains found in the wild - Project Zero
- The Fully Remote Attack Surface of the iPhone - Project Zero

Books

- Hacking and Securing iOS Applications: Stealing Data, Hijacking Software, and How to Prevent It
- iOS Penetration Testing
- iOS App Security, Penetration Testing, and Development
- IOS Hacker's Handbook
- Hacking iOS Applications a detailed testing guide
- Develop iOS Apps (Swift)
- iOS Programming Cookbook

Courses

- Pentesting iOS Applications
- Reverse Engineering iOS Applications
- App Design and Development for iOS

Tools

- Cydia Impactor

- checkra1n jailbreak
- idb - iOS App Security Assessment Tool
- Frida
- Objection - mobile exploration toolkit by Frida
- Bfinject
- iFunbox
- Libimobiledevice - library to communicate with the services of the Apple ios devices
- iRET (iOS Reverse Engineering Toolkit) - includes oTool, dumpDecrypted, SQLite, Theos, Keychain_dumper, Plutil
- Myriam iOS
- iWep Pro - wireless suite of useful applications used to turn your iOS device into a wireless network diagnostic tool
- Burp Suite
- Cycript
- needle - The iOS Security Testing Framework
- iLEAPP - iOS Logs, Events, And Preferences Parser
- Cutter - Free and Open Source RE Platform powered by radare2
- decrypt0r - automatically download and decrypt SecureRom stuff
- iOS Security Suite - an advanced and easy-to-use platform security & anti-tampering library

Labs

- OWASP iGoat
- Damn Vulnerable iOS App (DVIA) v2
- Damn Vulnerable iOS App (DVIA) v1
- iPhoneLabs
- iOS-Attack-Defense

Talks

- Behind the Scenes of iOS Security
- Modern iOS Application Security
- Demystifying the Secure Enclave Processor
- HackPac Hacking Pointer Authentication in iOS User Space
- Analyzing and Attacking Apple Kernel Drivers

- Remotely Compromising iOS via Wi-Fi and Escaping the Sandbox
- Reverse Engineering iOS Mobile Apps
- iOS 10 Kernel Heap Revisited
- KTRW: The journey to build a debuggable iPhone
- The One Weird Trick SecureROM Hates
- Tales of old: untethering iOS 11-Spoiler: Apple is bad at patching
- Messenger Hacking: Remotely Compromising an iPhone through iMessage
- Recreating An iOS 0-Day Jailbreak Out Of Apple's Security Updates
- Reverse Engineering the iOS Simulator's SpringBoard
- Attacking iPhone XS Max

Misc.

- Most usable tools for iOS penetration testing
- iOS-Security-Guides
- osx-security-awesome - OSX and iOS related security tools
- Trust in Apple's Secret Garden: Exploring & Reversing Apple's Continuity Protocol-Slides
- Apple Platform Security
- Mobile security, forensics & malware analysis with Santoku Linux

https://book.hacktricks.xyz/mobile-apps-pentesting/android-checklist

Exam Review

https://www.doyler.net/security-not-included/emapt-review#:~:text=While%20the%20eMAPT%20has%20definitely,vulnerability%20on%20a%20mobile%20device.

https://www.linkedin.com/pulse/emaptv2-exam-elearnsecurity-overview-joas-antonio/

https://brcyrr.medium.com/recommendations-review-of-emapt-819e72a27f06

https://www.youtube.com/watch?v=e7PcuZOD8bs

https://book.hacktricks.xyz/courses-and-certifications-reviews/ine-courses-and-elearnsecurity-certifications-reviews

Tips

1 - Shared Preferences

2 - SQL Injection

3 - Content Ptovider

4 - Crypto