Android APK Checklist

Learn Android fundamentals

- Basics
- Dalvik & Smali
- Entry points
    - Activities
    - URL Schemes
    - Content Providers
    - Services
    - Broadcast Receivers
    - Intents
    - Intent Filter
- Other components
- How to use ADB
- How to modify Smali

Static Analysis

- Check for the use of obfuscation, checks for noting if the mobile was rooted, if an emulator is being used and anti-tampering checks. Read this for more info.
- Sensitive applications (like bank apps) should check if the mobile is rooted and should actuate in consequence.
- Search for interesting strings (passwords, URLs, API, encryption, backdoors, tokens, Bluetooth uuids...).
    - Special attention to firebase APIs.
- Read the manifest:
    - Check if the application is in debug mode and try to "exploit" it
    - Check if the APK allows backups
    - Exported Activities
    - Content Providers
    - Exposed services
    - Broadcast Receivers
    - URL Schemes
- Is the application saving data insecurely internally or externally?
- Is there any password hard coded or saved in disk? Is the app using insecurely crypto algorithms?
- All the libraries compiled using the PIE flag?
- Don't forget that there is a bunch of static Android Analyzers that can help you a lot during this phase.

Dynamic Analysis

- Prepare the environment (online, local VM or physical)
- Is there any unintended data leakage (logging, /paste, crash logs)?
- Confidential information being saved in SQLite dbs?
- Exploitable exposed Activities?
- Exploitable Content Providers?
- Exploitable exposed Services?
- Exploitable Broadcast Receivers?

- Is the application transmitting information in clear text/using weak algorithms? is a MitM possible?
- Inspect HTTP/HTTPS traffic
    - This one is really important, because if you can capture the HTTP traffic you can search for common Web vulnerabilities (Hacktricks has a lot of information about Web vulns).
- Check for possible Android Client Side Injections (probably some static code analysis will help here)
- Frida: Just Frida, use it to obtain interesting dynamic data from the application (maybe some passwords...)

Some obfuscation/Deobfuscation information

- Read here

Android Applications Pentesting

Android Applications Basics

It's highly recommended to start reading this page to know about the **most important parts related to Android security and the most dangerous components in an Android application**:
PAGEAndroid Applications Basics

ADB (Android Debug Bridge)

This is the main tool you need to connect to an android device (emulated or physical). **ADB** allows to control devices either over **USB** or **Network** from a computer. This utility enables the **ing** of files in both directions, **installation** and **uninstallation** of apps, **execution** of shell commands, **backing up** of data, **reading** of logs, among other functions.

Take a look to the following list of **ADB Commands** to learn how to use adb.

Smali

Sometimes it is interesting to **modify the application code** to access **hidden information** (maybe well obfuscated passwords or flags). Then, it could be interesting to decompile the apk, modify the code and recompile it. **In this tutorial** you can **learn how to decompile and APK, modify Smali code and recompile the APK** with the new functionality. This could be very useful as an **alternative for several tests during the dynamic analysis** that are going to presented. Then, **keep always in mid this possibility**.

Other interesting tricks

- Spoofing your location in Play Store
- **Download APKs**: https://apps.evozi.com/apk-downloader/, https://apkpure.com/es/, https://www.apkmirror.com/, https://apkcombo.com/es-es/apk-downloader/, https://github.com/kiber-io/apkd
- Extract APK from device:

```
adb shell pm list packages
com.android.insecurebankv2

adb shell pm path com.android.insecurebankv2
package:/data/app/com.android.insecurebankv2-Jnf8pNgwy3QA_U5f-n_4jQ==/base.apk

adb pull /data/app/com.android.insecurebankv2-Jnf8pNgwy3QA_U5f-n_4jQ==/base.apk
```

- Merge all splits and base apks with APKEditor:

```
mkdir splits
adb shell pm path com.android.insecurebankv2 | cut -d ':' -f 1 | xargs -n1 -i adb pull {} splits
java -jar ../APKEditor.jar m -i splits/ -o merged.apk

# after merging, you will need to align and sign the apk, personally, I like to use the uberapksigner
java -jar uber-apk-signer.jar -a merged.apk --allowResign -o merged_signed
```

Static Analysis

First of all, for analysing an APK you should **take a look to the to the Java code** using a decompiler. Please, **read here to find information about different available decompilers**.

Looking for interesting Info

Just taking a look to the **strings** of the APK you can search for **passwords**, **URLs** (https://github.com/ndelphit/apkurlgrep), **api** keys, **encryption**, **bluetooth uuids**, **tokens** and anything interesting... look even for code execution **backdoors** or authentication backdoors (hardcoded admin credentials to the app).

Firebase

Pay special attention to **firebase URLs** and check if it is bad configured. More information about whats is FIrebase and how to exploit it here.

Basic understanding of the application - Manifest.xml, strings.xml

The **examination of an application's _Manifest.xml**** *and* ****strings.xml_**** files can reveal potential security vulnerabilities**. These files can be accessed using decompilers or by renaming the APK file extension to .zip and then unzipping it.

Vulnerabilities identified from the **Manifest.xml** include:

- **Debuggable Applications**: Applications set as debuggable (`debuggable="true"`) in the *Manifest.xml* file pose a risk as they allow connections that can lead to exploitation. For further understanding on how to exploit debuggable applications, refer to a tutorial on finding and exploiting debuggable applications on a device.
- **Backup Settings**: The `android:allowBackup="false"` attribute should be explicitly set for applications dealing with sensitive information to prevent unauthorized data backups via adb, especially when usb debugging is enabled.
- **Network Security**: Custom network security configurations (`android:networkSecurityConfig="@xml/network_security_config"`) in *res/xml/* can specify security details like certificate pins and HTTP traffic settings. An example is allowing HTTP traffic for specific domains.
- **Exported Activities and Services**: Identifying exported activities and services in the manifest can highlight components that might be misused. Further analysis during dynamic testing can reveal how to exploit these components.
- **Content Providers and FileProviders**: Exposed content providers could allow unauthorized access or modification of data. The configuration of FileProviders should also be scrutinized.
- **Broadcast Receivers and URL Schemes**: These components could be leveraged for exploitation, with particular attention to how URL schemes are managed for input vulnerabilities.
- **SDK Versions**: The `minSdkVersion`, `targetSDKVersion`, and `maxSdkVersion` attributes indicate the supported Android versions, highlighting the importance of not supporting outdated, vulnerable Android versions for security reasons.

From the **strings.xml** file, sensitive information such as API keys, custom schemas, and other developer notes can be discovered, underscoring the need for careful review of these resources.

Tapjacking

Tapjacking is an attack where a **malicious application** is launched and **positions itself on top of a victim application**. Once it visibly obscures the victim app, its user interface is designed in such a way as to trick the user to interact with it, while it is passing the interaction along to the victim app. In effect, it is **blinding the user from knowing they are actually performing actions on the victim app**.

Find more information in: PAGETapjacking

Task Hijacking

An **activity** with the **launchMode** set to **singleTask without any taskAffinity** defined is vulnerable to task Hijacking. This means, that an **application** can be installed and if launched before the real application it could **hijack the task of the real application** (so the user will be interacting with the **malicious application thinking he is using the real one**).

More info in: PAGEAndroid Task Hijacking

Insecure data storage

Internal Storage

In Android, files **stored** in **internal** storage are **designed** to be **accessible** exclusively by the **app** that **created** them. This security measure is **enforced** by the Android operating system and is generally adequate for the security needs of most applications. However, developers sometimes utilize modes such as `MODE_WORLD_READABLE` and `MODE_WORLD_WRITABLE` to **allow** files to be **shared** between different applications. Yet, these modes **do not restrict access** to these files by other applications, including potentially malicious ones.

1. **Static Analysis:**
   - **Ensure** that the use of `MODE_WORLD_READABLE` and `MODE_WORLD_WRITABLE` is **carefully scrutinized**. These modes **can potentially expose** files to **unintended or unauthorized access**.
2. **Dynamic Analysis:**
   - **Verify** the **permissions** set on files created by the app. Specifically, **check** if any files are **set to be readable or writable worldwide**. This can pose a significant security risk, as it would allow **any application** installed on the device, regardless of its origin or intent, to **read or modify** these files.

External Storage

When dealing with files on **external storage**, such as SD Cards, certain precautions should be taken:

1. **Accessibility**:
   - Files on external storage are **globally readable and writable**. This means any application or user can access these files.
2. **Security Concerns**:
   - Given the ease of access, it's advised **not to store sensitive information** on external storage.
   - External storage can be removed or accessed by any application, making it less secure.
3. **Handling Data from External Storage**:
   - Always **perform input validation** on data retrieved from external storage. This is crucial because the data is from an untrusted source.
   - Storing executables or class files on external storage for dynamic loading is strongly discouraged.

- If your application must retrieve executable files from external storage, ensure these files are **signed and cryptographically verified** before they are dynamically loaded. This step is vital for maintaining the security integrity of your application.

External storage can be **accessed** in `/storage/emulated/0` , `/sdcard` , `/mnt/sdcard`

Starting with Android 4.4 (**API 17**), the SD card has a directory structure which **limits access from an app to the directory which is specifically for that app**. This prevents malicious application from gaining read or write access to another app's files.

Sensitive data stored in clear-text

- **Shared preferences**: Android allow to each application to easily save xml files in the path `/data/data/<packagename>/shared_prefs/` and sometimes it's possible to find sensitive information in clear-text in that folder.
- **Databases**: Android allow to each application to easily save sqlite databases in the path `/data/data/<packagename>/databases/` and sometimes it's possible to find sensitive information in clear-text in that folder.

Broken TLS

Accept All Certificates

For some reason sometimes developers accept all the certificates even if for example the hostname does not match with lines of code like the following one:

```
SSLSocketFactory sf = new cc(trustStore);
sf.setHostnameVerifier(SSLSocketFactory.ALLOW_ALL_HOSTNAME_VERIFIER);
```

A good way to test this is to try to capture the traffic using some proxy like Burp without authorising Burp CA inside the device. Also, you can generate with Burp a certificate for a different hostname and use it.

Broken Cryptography

Poor Key Management Processes

Some developers save sensitive data in the local storage and encrypt it with a key hardcoded/predictable in the code. This shouldn't be done as some reversing could allow attackers to extract the confidential information.

Use of Insecure and/or Deprecated Algorithms

Developers shouldn't use **deprecated algorithms** to perform authorisation **checks**, **store** or **send** data. Some of these algorithms are: RC4, MD4, MD5, SHA1... If **hashes** are used to store passwords for example, hashes brute-force **resistant** should be used with salt.

Other checks

- It's recommended to **obfuscate the APK** to difficult the reverse engineer labour to attackers.
- If the app is sensitive (like bank apps), it should perform it's **own checks to see if the mobile is rooted** and act in consequence.
- If the app is sensitive (like bank apps), it should check if an **emulator** is being used.
- If the app is sensitive (like bank apps), it should **check it's own integrity before executing** it to check if it was modified.
- Use **APKiD** to check which compiler/packer/obfuscator was used to build the APK

React Native Application

Read the following page to learn how to easily access javascript code of React applications:

PAGEReact Native Application

Xamarin Applications

Read the following page to learn how to easily access C# code of a xamarin applications:

PAGEXamarin Apps

Superpacked Applications

According to this **blog post** superpacked is a Meta algorithm that compress the content of an application into a single file. The blog talks about the possibility of creating an app that decompress these kind of apps... and a faster way which involves to **execute the application and gather the decompressed files from the filesystem.**

Automated Static Code Analysis

The tool **mariana-trench** is capable of finding **vulnerabilities** by **scanning** the **code** of the application. This tool contains a series of **known sources** (that indicates to the tool the **places** where the **input** is **controlled by the user**), **sinks** (which indicates to the tool **dangerous places** where malicious user input could cause damages) and **rules**. These rules indicates the **combination** of **sources-sinks** that indicates a vulnerability.

With this knowledge, **mariana-trench will review the code and find possible vulnerabilities on it**.

Secrets leaked

An application may contain secrets (API keys, passwords, hidden urls, subdomains...) inside of it that you might be able to discover. You could us a tool such as https://github.com/dwisiswant0/apkleaks

Bypass Biometric Authentication

PAGEBypass Biometric Authentication (Android)

Other interesting functions

- **Code execution**: `Runtime.exec(), ProcessBuilder(), native code:system()`
- **Send SMSs**: `sendTextMessage, sendMultipartTestMessage`
- **Native functions** declared as `native`: `public native, System.loadLibrary, System.load`
    - Read this to learn **how to reverse native functions**

Other tricks

PAGEcontent:// protocol

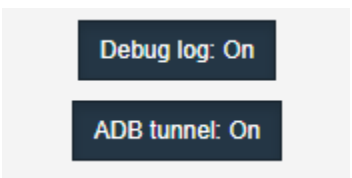
Dynamic Analysis

First of all, you need an environment where you can install the application and all the environment (Burp CA cert, Drozer and Frida mainly). Therefore, a rooted device (emulated or not) is extremely recommended.

Online Dynamic analysis

You can create a **free account** in: https://appetize.io/. This platform allows you to **upload** and **execute** APKs, so it is useful to see how an apk is behaving.

You can even **see the logs of your application** in the web and connect through **adb**.



Thanks to the ADB connection you can use **Drozer** and **Frida** inside the emulators.

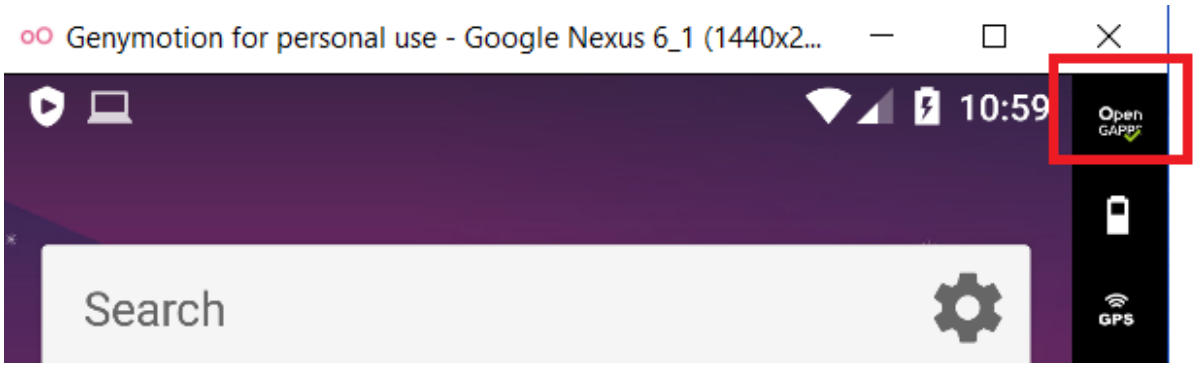
Local Dynamic Analysis

Using an emulator

- **Android Studio** (You can create **x86** and **arm** devices, and according to **this** **latest x86** versions **support ARM libraries** without needing an slow arm emulator).
    - Learn to set it up in this page: PAGEAVD - Android Virtual Device
- **Genymotion** **(Free version:** Personal Edition, you need to create an account. *It's recommend to **download** the version **WITH** **VirtualBox** to avoid potential errors.*)
- **Nox** (Free, but it doesn't support Frida or Drozer).

When creating a new emulator on any platform remember that the bigger the screen is, the slower the emulator will run. So select small screens if possible.

To **install google services** (like AppStore) in Genymotion you need to click on the red marked button of the following image:



Also, notice that in the **configuration of the Android VM in Genymotion** you can select **Bridge Network mode** (this will be useful if you will be connecting to the Android VM from a different VM with the tools).

Use a physical device

You need to activate the **debugging** options and it will be cool if you can **root** it:

1. **Settings**.
2. (FromAndroid 8.0) Select **System**.
3. Select **About phone**.
4. Press **Build number** 7 times.
5. Go back and you will find the **Developer options**.

Once you have installed the application, the first thing you should do is to try it and investigate what does it do, how does it work and get comfortable with it. I will suggest to **perform this initial dynamic analysis using MobSF dynamic analysis + pidcat**, so we will be able to **learn how the application works** while MobSF **captures** a lot of **interesting** **data** you can review later on.

Unintended Data Leakage

Logging

Developers should be cautious of exposing **debugging information** publicly, as it can lead to sensitive data leaks. The tools **pidcat** and `adb logcat` are recommended for monitoring application logs to identify and protect sensitive information. **Pidcat** is favored for its ease of use and readability.

Note that from **later newer than Android 4.0**, **applications are only able to access their own logs**. So applications cannot access other apps logs. Anyway, it's still recommended to **not log sensitive information**.

/Paste Buffer Caching

Android's **clipboard-based** framework enables -paste functionality in apps, yet poses a risk as **other applications** can **access** the clipboard, potentially exposing sensitive data. It's crucial to **disable /paste** functions for sensitive sections of an application, like credit card details, to prevent data leaks.

Crash Logs

If an application **crashes** and **saves logs**, these logs can assist attackers, particularly when the application cannot be reverse-engineered. To mitigate this risk, avoid logging on crashes, and if logs must be transmitted over the network, ensure they are sent via an SSL channel for security.

As pentester, **try to take a look to these logs**.

Analytics Data Sent To 3rd Parties

Applications often integrate services like Google Adsense, which can inadvertently **leak sensitive data** due to improper implementation by developers. To identify potential data leaks, it's advisable to **intercept the application's traffic** and check for any sensitive information being sent to third-party services.

SQLite DBs

Most of the applications will use **internal SQLite databases** to save information. During the pentest take a **look** to the **databases** created, the names of **tables** and **columns** and all the **data** saved because you could find **sensitive information** (which would be a vulnerability). Databases should be located in `/data/data/the.package.name/databases` like `/data/data/com.mwr.example.sieve/databases`

If the database is saving confidential information and is **encrypted b**ut you can **find** the **password** inside the application it's still a **vulnerability**.

Enumerate the tables using `.tables` and enumerate the columns of the tables doing `.schema <table_name>`

Drozer (Exploit Activities, Content Providers and Services)

From Drozer Docs: **Drozer** allows you to **assume the role of an Android app** and interact with other apps. It can do **anything that an installed application can do**, such as make use of Android's Inter-Process Communication (IPC) mechanism and interact with the underlying operating system. . Drozer is s useful tool to **exploit exported activities, exported services and Content Providers** as you will learn in the following sections.

Exploiting exported Activities

Read this if you want to refresh what is an Android Activity. Also remember that the code of an activity starts in the **`onCreate`** method.

Authorisation bypass

When an Activity is exported you can invoke its screen from an external app. Therefore, if an activity with **sensitive information** is **exported** you could **bypass** the **authentication** mechanisms **to access it.**

Learn how to exploit exported activities with Drozer.

You can also start an exported activity from adb:

- PackageName is com.example.demo
- Exported ActivityName is com.example.test.MainActivity

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

NOTE: MobSF will detect as malicious the use of ***singleTask/singleInstance*** as `android:launchMode` in an activity, but due to this, apparently this is only dangerous on old versions (API versions < 21).

Note that an authorisation bypass is not always a vulnerability, it would depend on how the bypass works and which information is exposed.

Sensitive information leakage

Activities can also return results. If you manage to find an exported and unprotected activity calling the **`setResult`** method and **returning sensitive information**, there is a sensitive information leakage.

Tapjacking

If tapjacking isn't prevented, you could abuse the exported activity to make the **user perform unexpected actions**. For more info about **what is Tapjacking follow the link**.

Exploiting Content Providers - Accessing and manipulating sensitive information

Read this if you want to refresh what is a Content Provider. Content providers are basically used to **share data**. If an app has available content providers you may be able to **extract sensitive** data from them. It also interesting to test possible **SQL injections** and **Path Traversals** as they could be vulnerable.

Learn how to exploit Content Providers with Drozer.

Exploiting Services

Read this if you want to refresh what is a Service. Remember that a the actions of a Service start in the method `onStartCommand`.

As service is basically something that **can receive data**, **process** it and **returns** (or not) a response. Then, if an application is exporting some services you should **check** the **code** to understand what is it doing and **test** it **dynamically** for extracting confidential info, bypassing authentication measures...
Learn how to exploit Services with Drozer.

Exploiting Broadcast Receivers

Read this if you want to refresh what is a Broadcast Receiver. Remember that a the actions of a Broadcast Receiver start in the method `onReceive`.

A broadcast receiver will be waiting for a type of message. Depending on ho the receiver handles the message it could be vulnerable. **Learn how to exploit Broadcast Receivers with Drozer.**

Exploiting Schemes / Deep links

You can look for deep links manually, using tools like MobSF or scripts like this one. You can **open** a declared **scheme** using **adb** or a **browser**:

```
adb shell am start -a android.intent.action.VIEW -d "scheme://hostname/path?param=value" [your.package.name]
```

*Note that you can **omit the package name** and the mobile will automatically call the app that should open that link.*

```
<!-- Browser regular link -->
<a href="scheme://hostname/path?param=value">Click me</a>
<!-- fallback in your url you could try the intent url -->
<a href="intent://hostname#Intent;scheme=scheme;package=your.package.name;S.browser_fallback_url=http%3A%2F%2Fwww.example.com;end">with alternative</a>
```

Code executed

In order to find the **code that will be executed in the App**, go to the activity called by the deeplink and search the function **onNewIntent**.

```
public void onNewIntent(Intent intent) {
    super.onNewIntent(intent);
    n().a(intent);
}
```

Sensitive info

Every time you find a deep link check that i**t's not receiving sensitive data (like passwords) via URL parameters**, because any other application could **impersonate the deep link and steal that data!**

Parameters in path

You **must check also if any deep link is using a parameter inside the path** of the URL like: `https://api.example.com/v1/users/{username}` , in that case you can force a path traversal accessing something like: `example://app/users?username=../../unwanted-endpoint%3fparam=value` . Note that if you find the correct endpoints inside the application you may be able to cause a **Open Redirect** (if part of the path is used as domain name), **account takeover** (if you can modify users details without CSRF token and the vuln endpoint used the correct method) and any other vuln. More info about this here.

More examples

An interesting bug bounty report about links (*/.well-known/assetlinks.json*).

Transport Layer Inspection and Verification Failures

- **Certificates are not always inspected properly** by Android applications. It's common for these applications to overlook warnings and accept self-signed certificates or, in some instances, revert to using HTTP connections.
- **Negotiations during the SSL/TLS handshake are sometimes weak**, employing insecure cipher suites. This vulnerability makes the connection susceptible to man-in-the-middle (MITM) attacks, allowing attackers to decrypt the data.
- **Leakage of private information** is a risk when applications authenticate using secure channels but then communicate over non-secure channels for other transactions. This approach fails to protect sensitive data, such as session cookies or user details, from interception by malicious entities.

Certificate Verification

We will focus on **certificate verification**. The integrity of the server's certificate must be verified to enhance security. This is crucial because insecure TLS configurations and the transmission of sensitive data over unencrypted channels can pose significant risks. For detailed steps on verifying server certificates and addressing vulnerabilities, **this resource** provides comprehensive guidance.

SSL Pinning

SSL Pinning is a security measure where the application verifies the server's certificate against a known stored within the application itself. This method is essential for preventing MITM attacks. Implementing SSL Pinning is strongly recommended for applications handling sensitive information.

Traffic Inspection

To inspect HTTP traffic, it's necessary to **install the proxy tool's certificate** (e.g., Burp). Without installing this certificate, encrypted traffic might not be visible through the proxy. For a guide on installing a custom CA certificate, **click here**.

Applications targeting **API Level 24 and above** require modifications to the Network Security Config to accept the proxy's CA certificate. This step is critical for inspecting encrypted traffic. For instructions on modifying the Network Security Config, **refer to this tutorial**.

Bypassing SSL Pinning

When SSL Pinning is implemented, bypassing it becomes necessary to inspect HTTPS traffic. Various methods are available for this purpose:

- Automatically **modify** the **apk** to **bypass** SSLPinning with **apk-mitm**. The best pro of this option, is that you won't need root to bypass the SSL Pinning, but you will need to delete the application and reinstall the new one, and this won't always work.
- You could use **Frida** (discussed below) to bypass this protection. Here you have a guide to use Burp+Frida+Genymotion: https://spenkk.github.io/bugbounty/Configuring-Frida-with-Burp-and-GenyMotion-to-bypass-SSL-Pinning/
- You can also try to **automatically bypass SSL Pinning** using **objection:** `objection --gadget com.package.app explore --startup-command "android sslpinning disable"`

- You can also try to **automatically bypass SSL Pinning** using **MobSF dynamic analysis** (explained below)
- If you still think that there is some traffic that you aren't capturing you can try to **forward the traffic to burp using iptables**. Read this blog: https://infosecwriteups.com/bypass-ssl-pinning-with-ip-forwarding-iptables-568171b52b62

Looking for Common Web Vulnerabilities

It's important to also search for common web vulnerabilities within the application. Detailed information on identifying and mitigating these vulnerabilities is beyond the scope of this summary but is extensively covered elsewhere.

Frida

Frida is a dynamic instrumentation toolkit for developers, reverse-engineers, and security researchers. **You can access running application and hook methods on run time to change the behaviour, change values, extract values, run different code...** If you want to pentest Android applications you need to know how to use Frida.

- Learn how to use Frida: **Frida tutorial**
- Some "GUI" for actions with Frida: **https://github.com/m0bilesecurity/RMS-Runtime-Mobile-Security**
- Ojection is great to automate the use of Frida: **https://github.com/sensepost/objection** , **https://github.com/dpnishant/appmon**
- You can find some Awesome Frida scripts here: **https://codeshare.frida.re/**
- Try to bypass anti-debugging / anti-frida mechanisms loading Frida as in indicated in https://erfur.github.io/blog/dev/code-injection-without-ptrace (tool linjector)

Dump Memory - Fridump

Check if the application is storing sensitive information inside the memory that it shouldn't be storing like passwords or mnemonics.

Using **Fridump3** you can dump the memory of the app with:

```
# With PID
python3 fridump3.py -u <PID>

# With name
frida-ps -Uai
python3 fridump3.py -u "<Name>"
```

This will dump the memory in ./dump folder, and in there you could grep with something like:

```
strings * | grep -E "^[a-z]+ [a-z]+ [a-z]+ [a-z]+ [a-z]+ [a-z]+ [a-z]+ [a-z]+ [a-z]+ [a-z]+ [a-z]+$"
```

Sensitive data in Keystore

In Android the Keystore is the best place to store sensitive data, however, with enough privileges it's still **possible to access it**. As applications tends to store here **sensitive data in clear text** the pentests should check for it as root user or someones with physical access to the device could be able to steal this data.

Even if an app stored date in the keystore, the data should be encrypted.

To access the data inside the keystore you could use this Frida script: https://github.com/WithSecureLabs/android-keystore-audit/blob/master/frida-scripts/tracer-cipher.js

```
frida -U -f com.example.app -l frida-scripts/tracer-cipher.js
```

Fingerprint/Biometrics Bypass

Using the following Frida script it could be possible to **bypass fingerprint authentication** Android applications might be performing in order to **protect certain sensitive areas:**

```
frida --codeshare krapgras/android-biometric-bypass-update-android-11 -U -f <app.package>
```

Background Images

When you put an application in background, Android stores a **snapshot of the application** so when it's recovered to foreground it starts loading the image before the app so ot looks like the app was loaded faster.

However, if this snapshot contains **sensitive information**, someone with access to the snapshot might **steal that info** (note that you need root to access it).

The snapshots are usually stored around: **`/data/system_ce/0/snapshots`**

Android provides a way to **prevent the screenshot capture by setting the FLAG_SECURE** layout parameter. By using this flag, the window contents are treated as secure, preventing it from appearing in screenshots or from being viewed on non-secure displays.

```
getWindow().setFlags(LayoutParams.FLAG_SECURE, LayoutParams.FLAG_SECURE);
```

Android Application Analyzer

This tool could help you managing different tools during the dynamic analysis: https://github.com/NotSoSecure/android_application_analyzer

Intent Injection

Developers often create proxy components like activities, services, and broadcast receivers that handle these Intents and pass them to methods such as `startActivity(...)` or `sendBroadcast(...)`, which can be risky.

The danger lies in allowing attackers to trigger non-exported app components or access sensitive content providers by misdirecting these Intents. A notable example is the `WebView` component converting URLs to `Intent` objects via `Intent.parseUri(...)` and then executing them, potentially leading to malicious Intent injections.

Essential Takeaways

- **Intent Injection** is similar to web's Open Redirect issue.
- Exploits involve passing `Intent` objects as extras, which can be redirected to execute unsafe operations.
- It can expose non-exported components and content providers to attackers.
- `WebView`'s URL to `Intent` conversion can facilitate unintended actions.

Android Client Side Injections and others

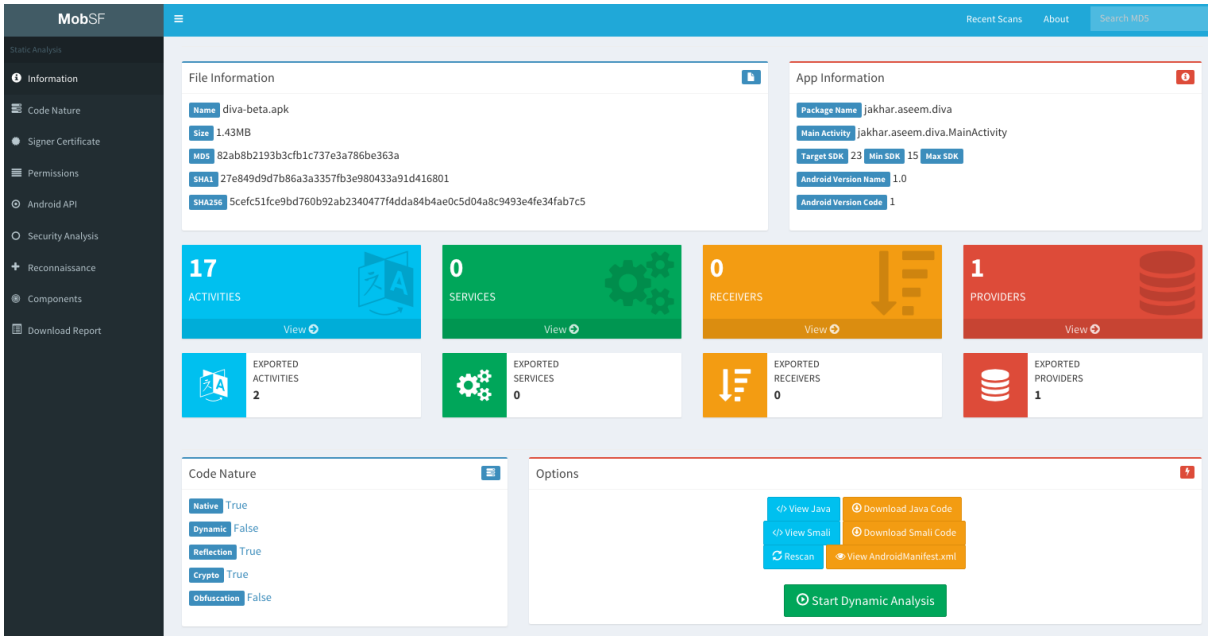
Probably you know about this kind of vulnerabilities from the Web. You have to be specially careful with this vulnerabilities in an Android application:

- **SQL Injection:** When dealing with dynamic queries or Content-Providers ensure you are using parameterized queries.
- **JavaScript Injection (XSS):** Verify that JavaScript and Plugin support is disabled for any WebViews (disabled by default). More info here.
- **Local File Inclusion:** WebViews should have access to the file system disabled (enabled by default) - `(webview.getSettings().setAllowFileAccess(false);)`. More info here.
- **Eternal cookies**: In several cases when the android application finish the session the cookie isn't revoked or it could be even saved to disk
- **Secure Flag** in cookies

Automatic Analysis

MobSF

Static analysis



Vulnerability assessment of the application using a nice web-based frontend. You can also perform dynamic analysis (but you need to prepare the environment).

```
docker pull opensecurity/mobile-security-framework-mobsf
docker run -it -p 8000:8000 opensecurity/mobile-security-framework-mobsf:latest
```

Notice that MobSF can analyse **Android**(apk)**, IOS**(ipa) **and Windows**(apx) applications (*Windows applications must be analyzed from a MobSF installed in a Windows host*). Also, if you create a **ZIP** file with the source code if an **Android** or an **IOS** app (go to the root folder of the application, select everything and create a ZIPfile), it will be able to analyse it also.

MobSF also allows you to **diff/Compare** analysis and to integrate **VirusTotal** (you will need to set your API key in *MobSF/settings.py* and enable it: `VT_ENABLED = TRUE` `VT_API_KEY = <Your API key>` `VT_UPLOAD = TRUE`). You can also set `VT_UPLOAD` to `False`, then the **hash** will be **upload** instead of the file.

Assisted Dynamic analysis with MobSF

MobSF can also be very helpful for **dynamic analysis** in **Android**, but in that case you will need to install MobSF and **genymotion** in your host (a VM or Docker won't work). *Note: You need to **start first a VM in genymotion** and **then MobSF.*** The **MobSF dynamic analyser** can:

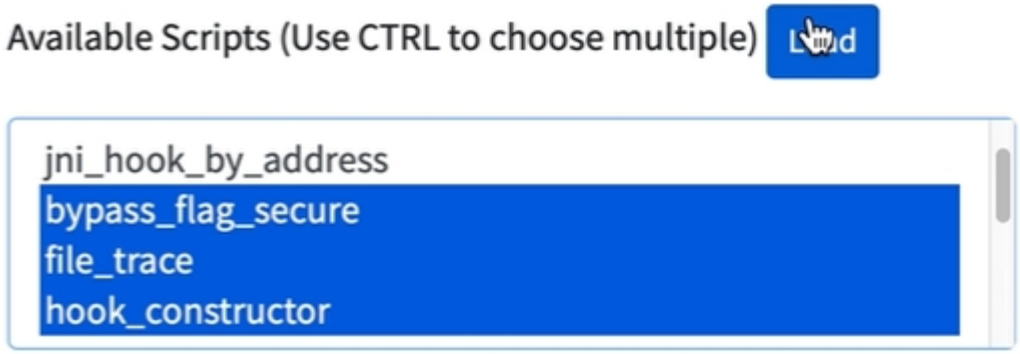
- **Dump application data** (URLs, logs, clipboard, screenshots made by you, screenshots made by "**Exported Activity Tester**", emails, SQLite databases, XML files, and other created files). All of this is done automatically except for the screenshots, you need to press when you want a screenshot or you need to press "**Exported Activity Tester**" to obtain screenshots of all the exported activities.
- Capture **HTTPS traffic**
- Use **Frida** to obtain **runtime information**

From android **versions > 5**, it will **automatically start Frida** and will set global **proxy** settings to **capture** traffic. It will only capture traffic from the tested application.

Frida

By default, it will also use some Frida Scripts to **bypass SSL pinning**, **root detection** and **debugger detection** and to **monitor interesting APIs**. MobSF can also **invoke exported activities**, grab **screenshots** of them and **save** them for the report.

To **start** the dynamic testing press the green bottom: "**Start Instrumentation**". Press the "**Frida Live Logs**" to see the logs generated by the Frida scripts and "**Live API Monitor**" to see all the invocation to hooked methods, arguments passed and returned values (this will appear after pressing "Start Instrumentation"). MobSF also allows you to load your own **Frida scripts** (to send the results of your Friday scripts to MobSF use the function `send()`). It also has **several pre-written scripts** you can load (you can add more in `MobSF/DynamicAnalyzer/tools/frida_scripts/others/`), just **select them**, press "**Load**" and press "**Start Instrumentation**" (you will be able to see the logs of that scripts inside "**Frida Live Logs**").



Moreover, you have some Auxiliary Frida functionalities:

- **Enumerate Loaded Classes**: It will print all the loaded classes
- **Capture Strings**: It will print all the capture strings while using the application (super noisy)
- **Capture String Comparisons**: Could be very useful. It will **show the 2 strings being compared** and if the result was True or False.
- **Enumerate Class Methods**: Put the class name (like "java.io.File") and it will print all the methods of the class.
- **Search Class Pattern**: Search classes by pattern
- **Trace Class Methods**: **Trace** a **whole class** (see inputs and outputs of all methods of th class). Remember that by default MobSF traces several interesting Android Api methods.

Once you have selected the auxiliary module you want to use you need to press "**Start Intrumentation**" and you will see all the outputs in "**Frida Live Logs**".

Shell

Mobsf also brings you a shell with some **adb** commands, **MobSF commands**, and common **shell commands** at the bottom of the dynamic analysis page. Some interesting commands:

```
help
shell ls
activities
exported_activities
services
receivers
```

HTTP tools

When http traffic is capture you can see an ugly view of the captured traffic on "**HTTP(S) Traffic**" bottom or a nicer view in "**Start HTTPTools**" green bottom. From the second option, you can **send** the **captured requests** to **proxies** like Burp or Owasp ZAP. To do so, *power on Burp --> turn off Intercept --> in MobSB HTTPTools select the request* --> press "**Send to Fuzzer**" --> *select the proxy address* (http://127.0.0.1:8080\).

Once you finish the dynamic analysis with MobSF you can press on "**Start Web API Fuzzer**" to **fuzz http requests** an look for vulnerabilities.

After performing a dynamic analysis with MobSF the proxy settings me be misconfigured and you won't be able to fix them from the GUI. You can fix the proxy settings by doing:

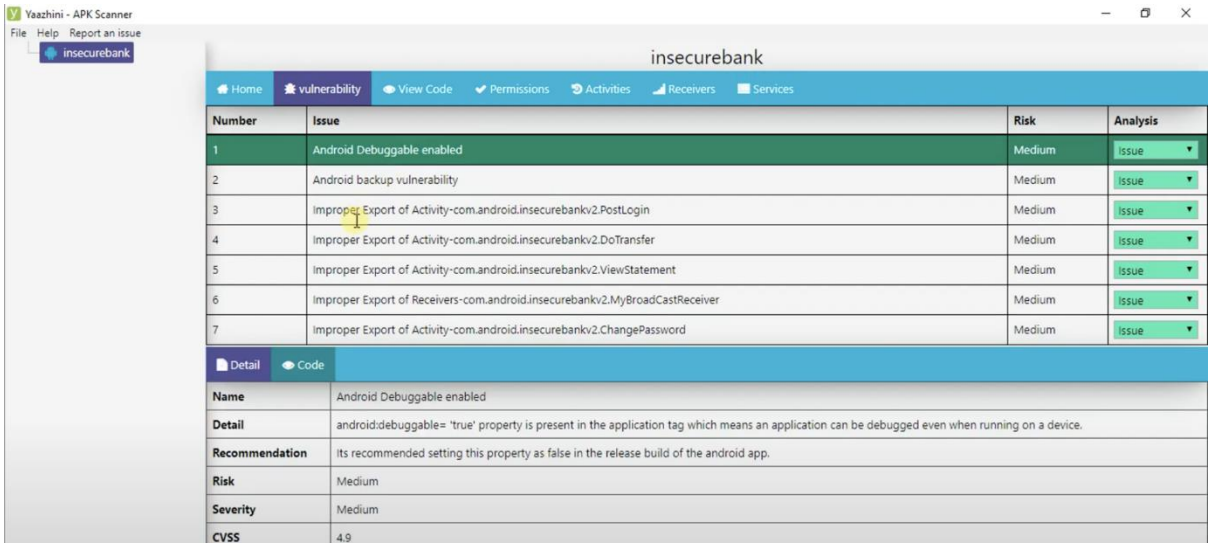
```
adb shell settings put global http_proxy :0
```

Assisted Dynamic Analysis with Inspeckage

You can get the tool from **Inspeckage**. This tool with use some **Hooks** to let you know **what is happening in the application** while you perform a **dynamic analysis**.

Yaazhini

This is a **great tool to perform static analysis with a GUI**



Qark

This tool is designed to look for several **security related Android application vulnerabilities**, either in **source code** or **packaged APKs**. The tool is also **capable of creating a "Proof-of-Concept" deployable APK** and **ADB commands**, to exploit some of the found vulnerabilities (Exposed activities, intents, tapjacking...). As with Drozer, there is no need to root the test device.

```
pip3 install --user qark  # --user is only needed if not using a virtualenv
qark --apk path/to/my.apk
qark --java path/to/parent/java/folder
qark --java path/to/specific/java/file.java
```

ReverseAPK

- Displays all extracted files for easy reference
- Automatically decompile APK files to Java and Smali format
- Analyze AndroidManifest.xml for common vulnerabilities and behavior
- Static source code analysis for common vulnerabilities and behavior
    - Device info
    - and more

```
reverse-apk relative/path/to/APP.apk
```

SUPER Android Analyzer

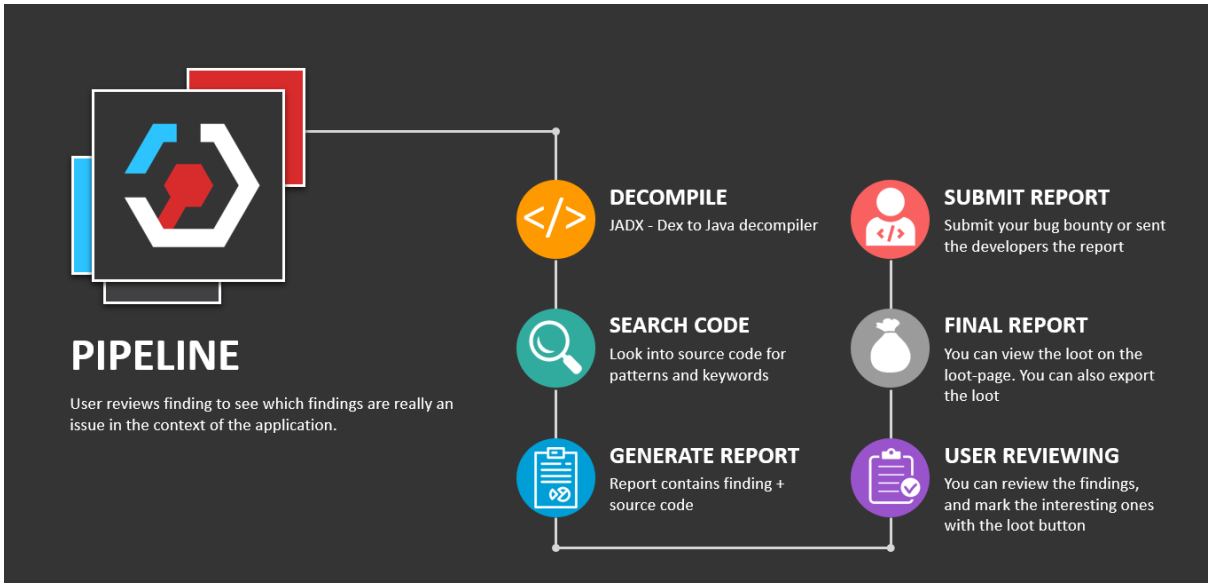
SUPER is a command-line application that can be used in Windows, MacOS X and Linux, that analyzes *.apk* files in search for vulnerabilities. It does this by decompressing APKs and applying a series of rules to detect those vulnerabilities.

All rules are centered in a `rules.json` file, and each company or tester could create its own rules to analyze what they need.

Download the latest binaries from in the download page

```
super-analyzer {apk_file}
```

StaCoAn



StaCoAn is a **crossplatform** tool which aids developers, bugbounty hunters and ethical hackers performing static code analysis on mobile applications.

The concept is that you drag and drop your mobile application file (an .apk or .ipa file) on the StaCoAn application and it will generate a visual and portable report for you. You can tweak the settings and wordlists to get a customized experience.

Download latest release:

```
./stacoan
```

AndroBugs

AndroBugs Framework is an Android vulnerability analysis system that helps developers or hackers find potential security vulnerabilities in Android applications. Windows releases

```
python androbugs.py -f [APK file]
androbugs.exe -f [APK file]
```

Androwarn

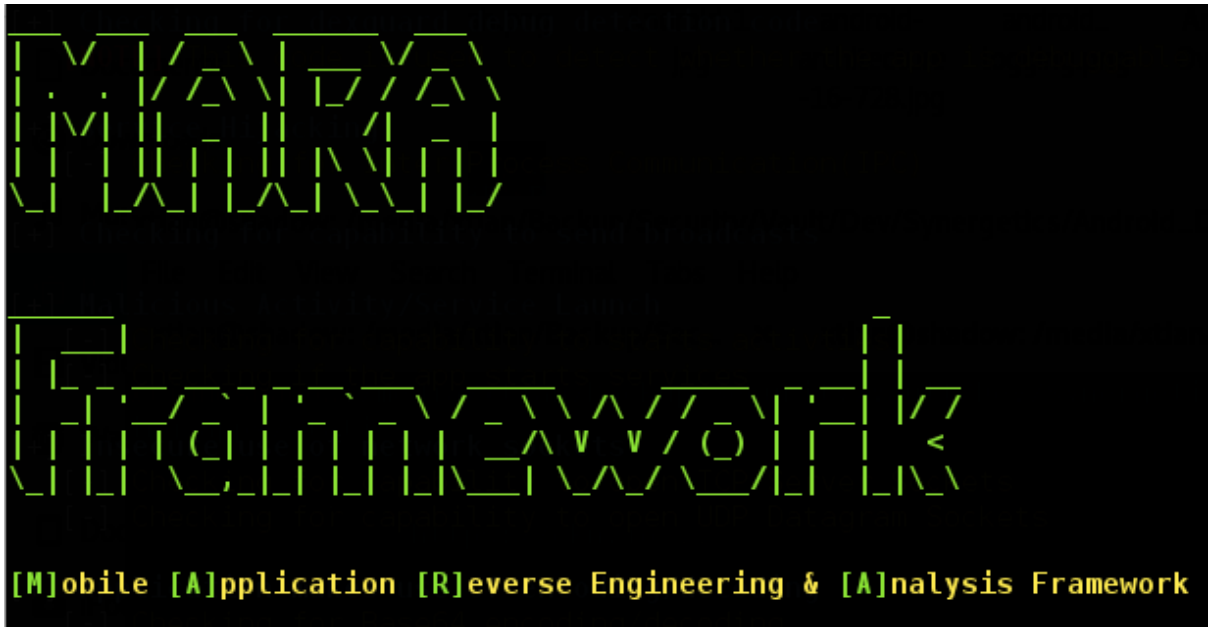
Androwarn is a tool whose main aim is to detect and warn the user about potential malicious behaviours developped by an Android application.

The detection is performed with the **static analysis** of the application's Dalvik bytecode, represented as **Smali**, with the androguard library.

This tool looks for **common behavior of "bad" applications** like: Telephony identifiers exfiltration, Audio/video flow interception, PIM data modification, Arbitrary code execution...

```
python androwarn.py -i my_application_to_be_analyzed.apk -r html -v 3
```

MARA Framework



MARA is a **M**obile **A**pplication **R**everse engineering and **A**nalysis Framework. It is a tool that puts together commonly used mobile application reverse engineering and analysis tools, to assist in testing mobile applications against the OWASP mobile security threats. Its objective is to make this task easier and friendlier to mobile application developers and security professionals.

It is able to:

- Extract Java and Smali code using different tools
- Analyze APKs using: smalisca, ClassyShark, androbugs, androwarn, APKiD
- Extract private information from the APK using regexps.
- Analyze the Manifest.
- Analyze found domains using: pyssltest, testssl and whatweb
- Deobfuscate APK via apk-deguard.com

Koodous

Useful to detect malware: https://koodous.com/

Obfuscating/Deobfuscating code

Note that depending the service and configuration you use to obfuscate the code. Secrets may or may not ended obfuscated.

ProGuard

From Wikipedia: **ProGuard** is an open source command-line tool that shrinks, optimizes and obfuscates Java code. It is able to optimize bytecode as well as detect and remove unused instructions. ProGuard is free software and is distributed under the GNU General Public License, version 2.

ProGuard is distributed as part of the Android SDK and runs when building the application in release mode.

DexGuard

Find a step-by-step guide to deobfuscate the apk in https://blog.lexfo.fr/dexguard.html

(From that guide) Last time we checked, the Dexguard mode of operation was:

- load a resource as an InputStream;
- feed the result to a class inheriting from FilterInputStream to decrypt it;
- do some useless obfuscation to waste a few minutes of time from a reverser;
- feed the decrypted result to a ZipInputStream to get a DEX file;
- finally load the resulting DEX as a Resource using the `loadDex` method.

DeGuard

DeGuard reverses the process of obfuscation performed by Android obfuscation tools. This enables numerous security analyses, including code inspection and predicting libraries.

You can upload an obfuscated APK to their platform.

Simplify

It is a **generic android deobfuscator.** Simplify **virtually executes an app** to understand its behavior and then **tries to optimize the code** so it behaves identically but is easier for a human to understand. Each optimization type is simple and generic, so it doesn't matter what the specific type of obfuscation is used.

APKiD

APKiD gives you information about **how an APK was made**. It identifies many **compilers**, **packers**, **obfuscators**, and other weird stuff. It's *PEiD* for Android.

Manual

Read this tutorial to learn some tricks on **how to reverse custom obfuscation**

Labs

Androl4b

AndroL4b is an Android security virtual machine based on ubuntu-mate includes the collection of latest framework, tutorials and labs from different security geeks and researchers for reverse engineering and malware analysis.

References

- https://owasp.org/www-project-mobile-app-security/
- https://appsecwiki.com/#/ It is a great list of resources
- https://maddiestone.github.io/AndroidAppRE/ Android quick course
- https://manifestsecurity.com/android-application-security/
- https://github.com/Ralireza/Android-Security-Teryaagh
- https://www.youtube.com/watch?v=PMKnPaGWxtg&feature=youtu.be&ab_channel=B3nacSec

Yet to try

- https://www.vegabird.com/yaazhini/
- https://github.com/abhi-r3v0/Adhrit