2 - Mobile Challenge

To do the challenge we need to connect to:
`sas.hackthe.space` and login with TU wien.

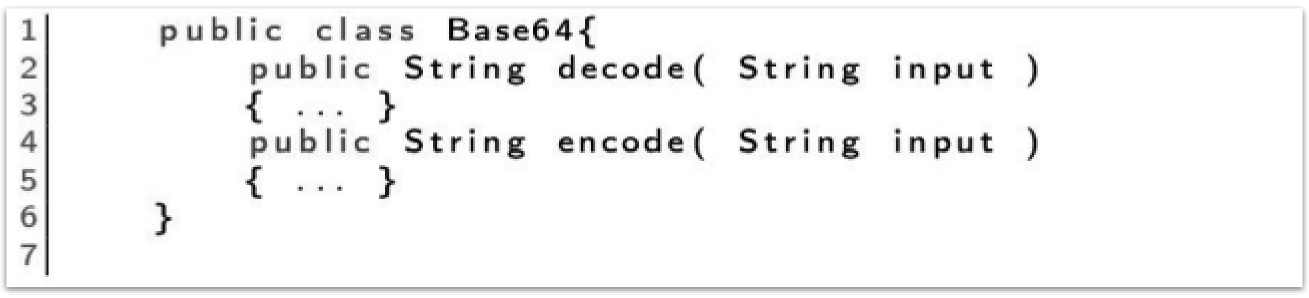
code protection techniques

There are some different techniques:

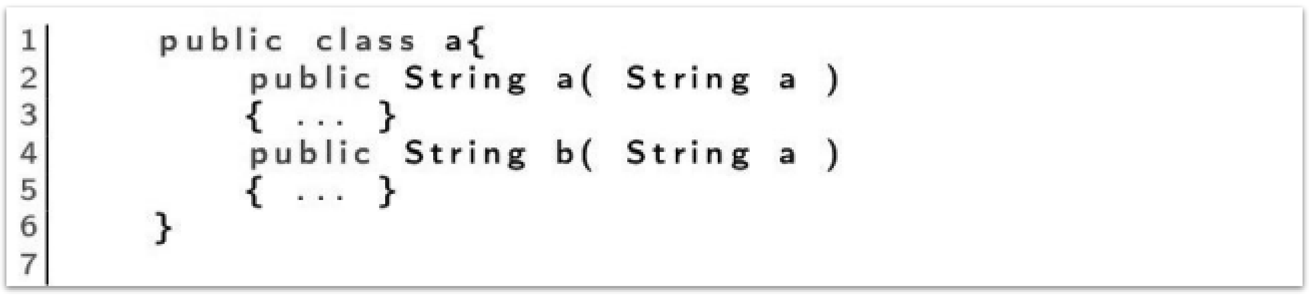
- **Obfuscation**. It involves renaming, string encryption, java reflection, code modification, dynamic code loading ecc
- **Optimization**. It involves removing dead code (also in libraries)
- **Packing**. It involves encrypt, pack classes, native code, resources ecc

They can be combined.

Basic name/identifier mangling



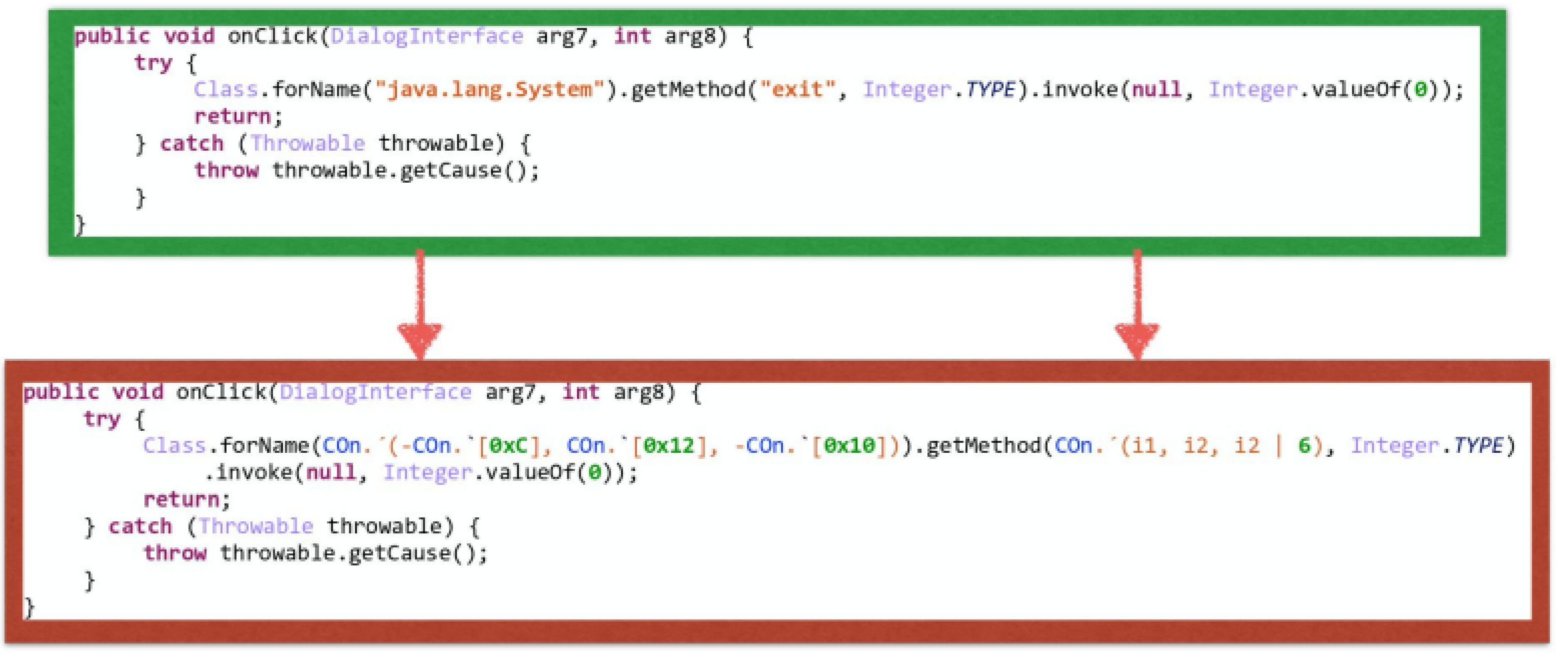
```
public class Base64{
    public String decode( String input )
    { ... }
    public String encode( String input )
    { ... }
}
```



```
public class a{
    public String a( String a )
    { ... }
    public String b( String a )
    { ... }
}
```

It is very common (also for package names), example com.my.example.app -> a.a.a.a

Java reflection



```
public void onClick(DialogInterface arg7, int arg8) {
    try {
        Class.forName("java.lang.System").getMethod("exit", Integer.TYPE).invoke(null, Integer.valueOf(0));
        return;
    } catch (Throwable throwable) {
        throw throwable.getCause();
    }
}
```

```
public void onClick(DialogInterface arg7, int arg8) {
    try {
        Class.forName(COn.´(-COn.`[0xC], COn.`[0x12], -COn.`[0x10])).getMethod(COn.´(i1, i2, i2 | 6), Integer.TYPE)
            .invoke(null, Integer.valueOf(0));
        return;
    } catch (Throwable throwable) {
        throw throwable.getCause();
    }
}
```

Dynamic code Loading

==It is a technique in which we load the code during the execution of the application. ==

This code can be included in the apk or also downloaded at runtime. It is potentially combined with encryption.

Very well supported on Android

- **java.net.url -> to download it**
- **DexClassLoader -> to load it from the apk**

It is also used to avoid the pushing of updates on Google Play (we just download the new code).

R8 (formerly ProGuard)

R8 are compilers that optimize and "obfuscate" Dalvik code in one step (just renaming strings):

- previous ProGuard was run as a plugin after java -> bytecode -> dex compilation

But commercial obfuscation is more popular and powerful.

Static analysis

The static analysis is used to analyze the app without running it and only looking at the code.

We need to disassemble and decompile them.

There are some tools already part of the SDK.

It is also useful to modify apk in order to execute then a dynamic analysis.

- we can add analysis code or also remove anti-analysis code.

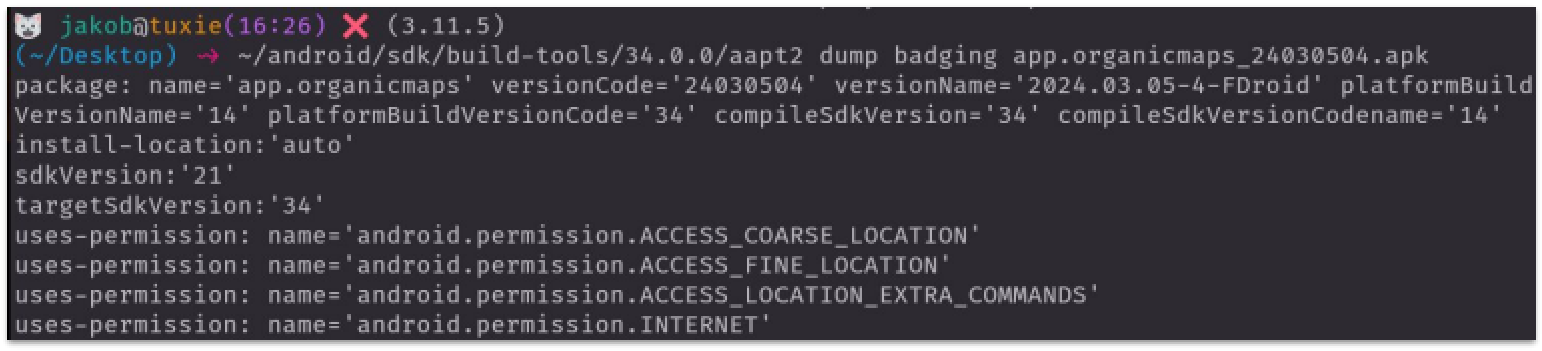
It is limited by:

- **obfuscation**
- **dynamic code loading and modifications**

android asset packaging tool (aapt/aapt2)

It is included into Android SDK. It gives info about APK file:

- file listing, used permissions, AndroidManifest



```
jakob@tuxie(16:26) ✗ (3.11.5)
(~/Desktop) → ~/android/sdk/build-tools/34.0.0/aapt2 dump badging app.organicmaps_24030504.apk
package: name='app.organicmaps' versionCode='24030504' versionName='2024.03.05-4-FDroid' platformBuild
VersionName='14' platformBuildVersionCode='34' compileSdkVersion='34' compileSdkVersionCodename='14'
install-location:'auto'
sdkVersion:'21'
targetSdkVersion:'34'
uses-permission: name='android.permission.ACCESS_COARSE_LOCATION'
uses-permission: name='android.permission.ACCESS_FINE_LOCATION'
uses-permission: name='android.permission.ACCESS_LOCATION_EXTRA_COMMANDS'
uses-permission: name='android.permission.INTERNET'
```

```
aapt2 dump badging file.apk
```

android disassembly

We could use Assembler/Disassembler for the DEX format used by Dalvik:

- https://github.com/JesusFreke/smali

But also to read on Dalvik bytecode format:

- https://source.android.com/devices/tech/dalvik/dalvik-bytecode.html

One of the best tools is JADX:

- https://github.com/skylot/jadx

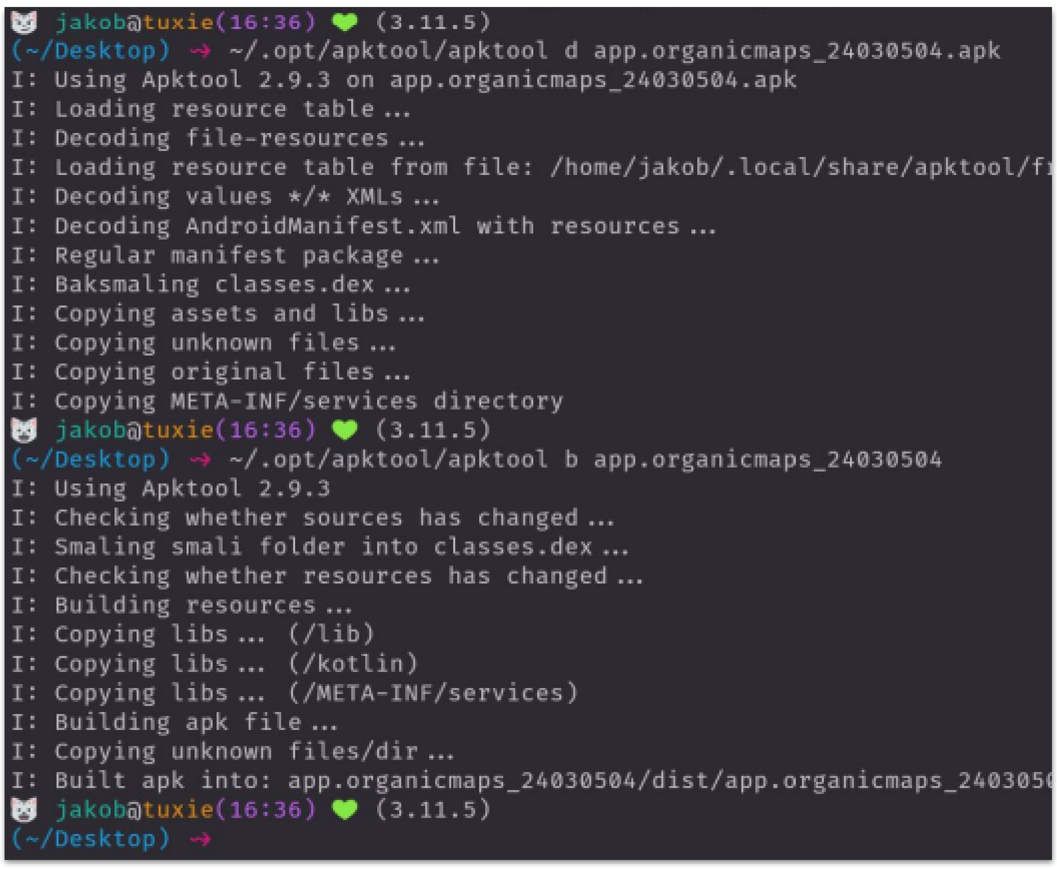
bytecode manipulation

The idea here is to write code in Java/Kotlin:

1. write an app in Android studio
2. compile the apk
3. decomoile it
4. extract the desired function
5. merge it into the apk we want to modify

apktool

It can be used to decompile the entire apk and obtain the source code.



```
jakob@tuxie(16:36) 💚 (3.11.5)
(~/Desktop) ➜ ~/.opt/apktool/apktool d app.organicmaps_24030504.apk
I: Using Apktool 2.9.3 on app.organicmaps_24030504.apk
I: Loading resource table ...
I: Decoding file-resources ...
I: Loading resource table from file: /home/jakob/.local/share/apktool/f
I: Decoding values */* XMLs ...
I: Decoding AndroidManifest.xml with resources ...
I: Regular manifest package ...
I: Baksmaling classes.dex ...
I: Copying assets and libs ...
I: Copying unknown files ...
I: Copying original files ...
I: Copying META-INF/services directory
jakob@tuxie(16:36) 💚 (3.11.5)
(~/Desktop) ➜ ~/.opt/apktool/apktool b app.organicmaps_24030504
I: Using Apktool 2.9.3
I: Checking whether sources has changed ...
I: Smaling smali folder into classes.dex ...
I: Checking whether resources has changed ...
I: Building resources ...
I: Copying libs ... (/lib)
I: Copying libs ... (/kotlin)
I: Copying libs ... (/META-INF/services)
I: Building apk file ...
I: Copying unknown files/dir ...
I: Built apk into: app.organicmaps_24030504/dist/app.organicmaps_240305
jakob@tuxie(16:36) 💚 (3.11.5)
(~/Desktop) ➜
```

```
apktool d app.apk  #to decompile

apktool b app.apk  #to build
```

But it doesn't sign APKs and we need a valid signature to install apps.

Resigning APKs

The solution is to use keytool and apksigner.

Dependig on Android OS version the key signature algorithm needs to fulfill requirements:

- https://source.android.com/docs/security/features/apksigning?hl=it

To create a key:

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

Before to sign we need to align the apk:

```

```

To sign the apk:

```
apksigner sign --ks my-release-key.keystore --ks-key-alias alias_name ./MyApp/dist/MyApp.apk
```

To verify a signature:

```
apksigner verify ./MyApp/dist/MyApp.apk
```

Androguard

Is a python tool for playing with apk files, first major version is released in 2023 and has no up to date documentation.

Many project depend on the older version 3.4.0a1

Android decompilation: Dalvik to Java

dex2jar, sootUP allows to convert dalvik into java or compatible formats.

This allows the using of many tools especially in automated analysis.

However java has a maximum method sizr that is 65535 bytes, so maybe people write method greater than it in order to avoi the conversion from dex to jar.

Dynamic app analysis

The idea here is to install and execute app in the emulator or on a phone to observe its behavior on different levels of granularity.

- in order to monitor system changes (such as file system), API calls, system call, instructions

We can also use APIs and modify the APK/system on the fly.

Or capture and modify HTTP(s) network traffic.

Limitations:

- anti-debugging checks
- anti-emulator checks
- code coverage, we need to intercat a lot

android debug bridge (adb)

It is a powerful device toolkit to debugging and inspecting device state.

It allows privileged access to device's file system and applications and can allow unauthorized access to data.

If it is used with a physical device it needs DeveloperOptions:

- https://developer.android.com/studio/debug/dev-options?hl=it

root detection

Some apps try to detect rooted devices and refuse to run if it is detected.

They use some heuristics like :

- checking some packages (superuser, supersu)
- checking certain apps (like busybox)
- checking BUILD-tags for test-keys of custom ROMs
- checking build.tags=release-keys on must stock ROMs

Android emulators

They are available as part of the SDK in adroid studio, but there are some alternatives:

- cuttlefish
- genymotion
- bluestacks
- waydroid

Many uses adv so are not so useful if we want to intecept traffic.

Instrumentation / Hooking toolkits

They are used to modify APK and system behavior (hooking framework APIs)

We don't need to modify the APK in this case.
In general they are divided into modular systems.

Frida:

- https://frida.re/

For iOS **CydiaSubstrate**:

- http://www.cydiasubstrate.com/

Dynamic Analysis Automation

This can be a huge challenge if we don't have "main" method and by the fact that apps have many entry points (activities, services). In addition sometimes apps need user input or authentication.

However Monkey (Android SDK) generates random clicks and text inputs.

There is a more structured approach that requires writing test cases:

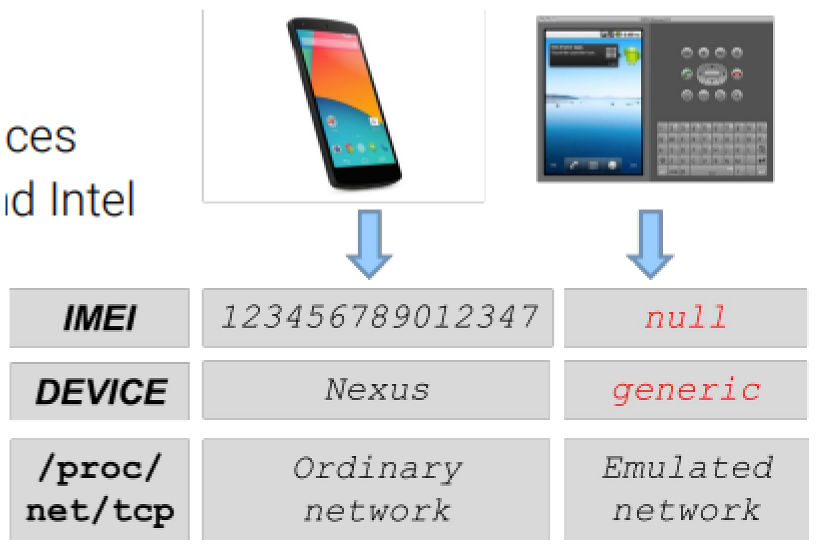
- UI autmoator

- Robotium
- Appium

Dynamic Analysis Evasion on emaulators

Android emulator are easy to noticed:

- check for strings in build properties (product, model, device)
- check for empty/known IMEI/IMSI
- look for emulator process (quemud)
- timing and caching differences caused by translation from ARM and Intel
- battery level always at the same %



IMEI	123456789012347	null
DEVICE	Nexus	generic
/proc/net/tcp	Ordinary network	Emulated network

Dynamic Analysis Evasion on phones

Real users would have usage cases:

- e.g., call log, SMS, contacts, browsing history, installed apps

Real device might not be always charging
Real device might not be always stationary

- e.g., look at sensor data from gyroscope, accelerometer, …

Real users interact differently with the UI than “monkeys”

Applied Security Analysis: Play Protect

Google performs dynamic analysis to check the apps submitted to google play.

what about the privacy?

Play Protect flags apps according to Google’s (internal) policies

Google provides recommendations for how apps should protect user’s privacy:

- Best practices: https://developer.android.com/privacy/best-practices (Does not necessarily enforce them, e.g., check then with Play Protect)
- **Usually apps just justify their data collection in a (very general) privacy policy to justify Play Store policies and legislators**

mobile vulnerabilities and exploits

iOS operation trinagulation

The attack is carried out using an invisible iMessage with a malicious attachment, which, using a number of vulnerabilities in the iOS operating system, is executed on a device and installs spyware.

The spyware then quietly transmits private information to remote servers: microphone recordings, photos from instant messengers, geolocation, and data about a number of other activities of the owner of the infected device.

A WebView to a kill (2012/13)

WebView JavaScript bridge allowed bidirectional calls, access to arbitrary Java objects through reflection

It allows **breaking browser sandbox by providing access to anything the app has permissions for, such as system resources (e.g., files, camera) and sensitive data** (e.g., contacts)

It also involved **instant remote code execution**, especially

- **When included JS is loaded through plain HTTP**
- ==**When app ignores HTTPS certificate errors (empty onReceivedSslError())** ==

Since Android 4.2 apps need to annotate which Java methods to export to JavaScript

- https://labs.f-secure.com/archive/webview-addjavascriptinterface-remote-code-execution/
- https://www.usenix.org/conference/leet13/workshop-program/presentation/neugschwandtner

Masterkey (13/14)

Multiple vulnerabilities in APK signing process allowed attackers to modify (e.g., Trojanize) **apps without breaking signature** and e.g., replace classes.dex, native libraries, AndroidManifest.xml, …

Essentially Time-of-check to time-of-use (TOCTOU) vulnerability

Original: ZIP format allows two files with the same name in the same archive

- Android signature verification (in Java) looked at first matching file
- Android app loader (in C) launched the second matching file
- http://www.saurik.com/id/19
- http://www.saurik.com/id/18
- http://www.saurik.com/id/17

Stagefright (2015)

It was generated by a multimedia parsing library on Android (libstagefright) and it is an integer overflow vulnerability

It allows remote execution Triggered by MMS message, eMail, visiting a website, loading a video, …

- https://www.blackhat.com/docs/us-15/materials/us-15-Drake-Stagefright-Scary-Code-In-The-Heart-Of-Android.pdf

Dirty COW (2016)

Race condition in handling copy-on-write (COW)
Local privilege escalation by gaining write access to read-only memory

- e.g., also modifying on-disk binaries

Inherited vulnerability from Linux kernel

- https://github.com/dirtycow/dirtycow.github.io/wiki/VulnerabilityDetails

Drammer (2016)

Deterministic exploit of the Rowhammer DRAM vulnerability

It allows attacker to change memory they don't have any access to by forcing bit flips

Result: Root an Android phone without any software vulnerabilities

Only superficially fixed by Google by disabling an API, essentially complicating the attack

- https://www.vusec.net/projects/drammer/
- https://rampageattack.com/

StrandHogg (2019)

Confused Deputy Attack in Android's Handling of Multitasking

Allows escalation of permissions and/or phishing attacks

Malicious app can “piggy-back” on other app launch and hijack the displayed activity, i.e., appear on top of legitimate activity as an overlay

Requires attacker to specify targeted victim app in AndroidManifest.xml

StrandHogg 2.0 (2020): the idea here is to target any app by programmatically, i.e., dynamically at runtime

- https://go.promon.co/l/866772/2020-05-24/mg6msk/866772/87590/Promon_POC_Strandhogg_2_0.pdf
- https://promon.co/security-news/strandhogg/
- https://promon.co/strandhogg-2-0/

“Analyzing a Modern In-the-wild Android Exploit” (2023)

It uses two 0-days in dependencies:

- CVE-2023-0266 (ALSA compatibility layer)
- CVE-2023-26083 (Mali GPU driver)

The vulnerabilities that it exploits are:

- change in 32-bit compatibility layer skips setting locks
- exploiting race condition leads to heap spray
- ultimately arbitrary kernel read/write access
- https://googleprojectzero.blogspot.com/2023/09/analyzing-modern-in-wild-android-exploit.html

Further Reading and Resources

- Google Documentation
    - https://source.android.com/reference
    - https://developer.android.com/docs/
    - https://developer.android.com/guide/
    - https://developer.android.com/reference/
    - https://developer.android.com/training/articles/security-tips
- Android Hacker’s Handbook Joshua J. Drake, Zach Lanier, Collin Mulliner, Pau Oliva Fora, Stephen A. Ridley, Georg Wicherski
- Mobile Systems and Smartphone Security Course (Mobisec) by Yanick Fratantonio
    - https://mobisec.reyammer.io
- OWASP Mobile Security Testing Guide (covers both Android & iOS)
    - https://mobile-security.gitbook.io/mobile-security-testing-guide/