UCLouvain

epl

École polytechnique de Louvain

Malware Sandbox Deployment, Analysis and Development

Author: **Luis Tascon Gutierrez**
Supervisor: **Axel Legay**
Readers: **Thomas Given Wilson, Charles Pecheur**
Academic year 2019-2020
Master [120] in Computer Science and Engineering

Abstract

Year 2020, the world is in the midst of an event unprecedented, the coronavirus appeared. A lot of countries reacted by a lockdown which increased the number of people needing internet. However at the same time, malware developers have increased their attacks knowing that a lot of vulnerable people are now surfing on the web.

This thesis talks about the "Malware Sandbox", a technology developed by malware researchers to be able to detect automatically a new malware. *Cuckoo*, the leading open-source malware sandbox, is in the spotlight in this thesis.

It begins by describing how it is possible to deploy *Cuckoo* easier by integrating it into a container. All the difficulties it implied and how they were overcome. The installation process has been automated thanks to *Docker* and some bash scripts.

Then, the second part shows some results collected thanks to this configuration and it shows how the malware evasion is detected by *Cuckoo* and how it gives a score to the malware.

The third part is about developing *Cuckoo 3* which will soon replace *Cuckoo 2* with multiple new features. This part explains what is different in these two versions and describes a contribution to add the machinery module *VirtualBox*.

Keywords: Malware, Sandbox, Cuckoo, Evasion, Docker, VirtualBox

Acknowledgement

Foremost, I would like to express my gratitude to my thesis supervisor, Prof. Axel Legay, for his support and his quick answers in all time.

I would like to thank especially the PhD. Fabien Duchêne, who helped me multiple times when I had practical issues and questions.

Thanks to the readers of my thesis, Prof. Charles Pecheur and Thomas Given Wilson.

A special thanks, to Ricardo van Zutphen, who is in the Cuckoo development team and whom I contacted to make a part of my thesis, for his time and explanations.

Thanks to Mélanie Massoz for her support during the lockdown. Towards my family and my friends, who supported me through the whole year, I express my sincere gratitude.

Thanks again to Mélanie, Charles-Henry and my parents, who have given a feedback to improve this report.

Contents

Chapter 1

Introduction

1.1 Context

During the lockdown, we have seen a significant raise in cyber attacks [1]. However this is not new, there have been many researches in this field and behind the curtains malware developers and malware researchers spent a lot of time trying to beat one another. This war is more like a game of cat and mouse, the mouse being the malware developers who are almost always a step ahead of the malware researchers.

As the number of threats and new malware per day is incredibly high, these malware cannot be all analysed by hand. It would require a gigantic effort that is not available now and that will certainly never be.

In order to respond to these threats, the malware researchers have developed many tools. One of this tool is the "Malware Sandbox" [2]. This term is also known in the field of software programming to have fresh environments to deploy a new software to test. However in this thesis, I talk about this technology in the cybersecurity field.

As *Cuckoo Sandbox* is the most used malware sandbox open-source, this thesis is mostly about *Cuckoo*.

1.2 Goals

The first goal of this thesis is to help to deploy *Cuckoo* which is very difficult to install and configure for the first time. The point would be to help a student who has never done it, to install it easily thanks to the container technology.

The second goal is to analyse multiple malware with this configuration and to retrieve some statistics from these analyses.

The third goal is to help the development of *Cuckoo Sandbox*.

1.3 Structure

Part I This part talks about the theory you need to know in order to understand fully this thesis. Beginning by general knowledge in Chapter 2, then deepening the knowledge about the malware evasion in Chapter 3 and finally a bit of theory to understand specifically *Cuckoo* in Chapter 4.

Part II This part is about my contributions and the fulfilment of the goals previously defined (see Section 1.2). The Chapter 5 explains how I managed to deploy *Cuckoo* thanks to *Docker*. The Chapter 6 describes all the analyses I have done and the Chapter 7 explains my contribution to *Cuckoo Sandbox*.

I wrap up this thesis with a text about the future works in Chapter 8 and then a conclusion in the Chapter 9.

Part I

Theory

Chapter 2

Background

What is a malware? A malware is often called a virus in the common language but we make the difference between these two terms in this paper because they do not have the same technical definition. A malware is a portmanteau term that comes from "malicious software", any bunch of code that has a malicious behaviour once executed on a computer. A virus, on the other hand, is a subset of malware that is defined later. "Malicious" is really vague and it is because a malware can have many different behaviours, it can encrypt your data, steal them or break functionalities in your computer.

Malware developers often have as goal to write a malware to make money but it can be for a lot of other reasons. The malware developers can make a malware for political or religious reason, or even for the challenge.

This chapter reviews the different types of malware in Section 2.1, then explains how a malware can be detected and analysed in Sections 2.2 and 2.3. Finally it explains what is an antivirus in Section 2.4 and a malware sandbox in Section 2.5.

2.1 Types of malware

As the term malware groups all codes that have malicious behaviour, malware researchers came with multiple terms to describe more precisely the kind of malware they are talking about [3, 4].

Adware This is an unwanted software that displays advertisements on your screen. It is generally using a browser to display them and is often installed when you download and install another (wanted) software.

Cryptojacking This malware is newer and aims to mine cryptocurrency on your system while it is running. This can cause some slowdown in the systems but it aims to be undetected so as not to arouse suspicion.

Keylogger It is a code generally quite simple that registers the keys pressed on the keyboard and that sends them to the malware developer. This malware is often used to target a specific computer in order to steal sensible credentials.

Ransomware This is a program that encrypts as fast as possible all the data it can find on the infected computer. It encrypts the data with a public key algorithm so that it is impossible to find the decryption key on the memory at any point of the infection. The private key that is able to decrypt the data is in possession of the malware developer. Typically, there is no other way to recover from a ransomware than to pay the ransom requested in exchange for the key. The only other way to recover from a ransomware is to have made a backup of your data previously and to reset your system entirely.

Rootkit It gives an administrator privilege of the system to the malware developer. In general, a rootkit tries to stay hidden from the user as much as possible to remain on the system as long as possible.

Spyware This is, as the name suggests, a kind of a spy on the system it infected. It sends data and activities of the users to the malware developer.

Trojan This is a particular type of malware, because it does not really describe its behaviour but more its appearance. In fact, the trojan presents itself as a legitimate software with which you want to do something. However, this software is only there to trick you, it may or may not do what you were expecting but it also has an unexpected behaviour which is malicious.

Virus It is a malware that waits for the user to execute it. A malware on its own never does anything. However, once a user runs it, it infects the machine, replicates and tries to send itself to other systems.

Worm It is quite similar to a virus but, unlike the virus, it does not need any user interaction to infect the system and to duplicate. With the network as it is now, a worm can spread exponentially, and rapidly infect a lot of systems if it exploits an issue that is still not patched.

A malware can be a combination of those terms, for example it can be a **virus** and a **trojan** at the same time. This virus waits for you to execute the software that you think may help you in some tasks but once you run it, the virus is executed and your computer infected. It can also be a **keylogger** and an **adware**, that way it can show you unwanted ads and send to the developer your activity on the keyboard.

2.2 Malware Detection

The detection of a malware is not as simple as it may seem [5]. In fact, when you have a **ransomware** on your computer it is obvious that your system is infected because you get a message from the malware asking you to pay the ransom. However, when it is a **rootkit** or a **spyware**, you may not be aware of the presence of such things in your system. Depending on the malware you are facing, the malware developers may have invested considerable efforts trying to be undetected by classical antiviruses that may be installed on a system or even a firewall from a company. Detecting these kinds of malware is a whole different story.

There are at least three moments at which you can detect a malware. The first one is when the malware arrives in your system, when you download it or when you receive a mail. The detection happening at this moment can be done by the system itself or by a "Next Generation Firewall" [6]. The second moment takes place when this malware is in your system but still does not do anything. And the third one is when it executes and you observe a malicious behaviour but it may be too late in some cases.

There are a lot of detection techniques [7, 8, 9] and here is a presentation of the main ones.

The signature-based detection This consists of making a signature of a file, for example with hashing algorithms. You can make a hash of the file that you suspect to be malicious and then compare this hash with a database containing many signatures of other files. If the signature matches with a known malware you know that this file is malicious.

The heuristic detection It is more about reading the code and trying to detect some patterns. If the malware tries to manipulate the camera driver or to access personal files, it may be a spyware. Based on these heuristics, you can build a degree of confidence that the file is a malware.

The behaviour-based detection This is now about executing the malware in a constrained environment and trying to identify if the software is acting normally. For this technique you must make sure that both the environment cannot be detected by the malware and that the malware cannot escape it. Once the analyses are run, you have plenty of data to determine if the behaviour was friendly or harmful.

The machine-learning detection This is about designing a machine learning algorithm and training it to detect malware. Then once the algorithm has been well trained, you give the new file to it and see what is its output.

These methods have each one their pros and cons, but they can be used together. The signature based is really sensible to variations in codes with the same behaviour, this property creates a lot of false negatives. The heuristic depends on what we search but there is a big risk of lots of false positives. The machine learning algorithms have the flaws of any machine learning algorithms, it may overfit the data that was presented to it. Finally, the behaviour-based detection has to be in a controlled environment to allow the testing, but this environment might be detected by the malware.

2.3 Malware Analysis

Malware threat detection is a critical part of the security of a computer system. However, just the detection of a malware is not enough, you have to understand what it does. Malware analysis is defined as determining the origin, the functionality and the impact of a malware on a system. Analysing malware is really important for security analysts, and by analysing a malware, they may discover flaws or weak points in a system. The analysis of malware is done in general after its detection. Even if it can also be a way to detect a malware, you do not always want to do so. An analysis instead of detection is not always a good idea because the process of analysing a malware costs more time and resources than just detection in general. There are a lot of techniques to analyse a malware [10, 11]. However, there are three main categories of analysis (static, dynamic and forensic) [12], in general any way of analysing a malware can fall under one of these categories. A fourth one is also presented here but it is an hybrid approach.

2.3.1 Static analysis

The older one and still much used technique is the static analysis. It consists of gathering information by observing the code and the data of a software that we suspect malicious without executing it. The binary can be disassembled to try to see the instructions and to understand what it does. The information you may be interested in goes from the size of the malware to assembly instructions.

However, there is a lot of limits for this analysis technique. It has been proven that static analysis techniques are not sufficient to identify a malware [13].

2.3.2 Dynamic analysis

The opposite of the static analysis is the dynamic one. This technique consists of executing a (suspected) malware in a controlled environment and observing the behaviour of the program in this environment. It allows us to execute the

(suspected) malware even if its purpose is to crash the environment in which it is executed. With this environment you can then get a lot of information like the system calls the program made, the packets it tried to send to the network, the time taken to execute, the memory dump of the machine after the execution. The other advantage is that you can repeat this execution multiple times on the same machine or even on different machines.

This technique, being not only used for analyses, is also known as a detection technique. That is why the malware authors develop more and more advanced countermeasures to avoid being detected, and therefore being analysed [14].

One other problem with this technique is when a malware only executes itself in specific cases. For example, a malware could be waiting for another process to be launched, it may only run on specific environment as an IOT device like a camera. There exist also techniques to detect an "environment-sensitive" malware [15].

These are the major challenges that this technique is facing.

2.3.3 Forensic analysis

There is also another category for the analysis of a malware, this is the analysis of the memory. The forensic analysis is an analysis of a memory dump of a disk after that the malware was executed. The analysts doing forensic must have a lot of different skills as it is required to understand the architecture and the OS in depth. However, there exist even techniques to reconstruct the chain of events that happened during the execution, thanks to the dumped memory [16].

One disadvantage of this method is that, if the malware is there to crash the system, then you have a crashed system every time you want to analyse the memory.

2.3.4 Concolic analysis

The concolic analysis aims to counter a limitation of the dynamic analysis. This technique is in between the static and the dynamic analysis because it does not execute the binary but it simulates many possible execution paths. The goal is to evaluate all possible paths and to see what would be the behaviour of the malware in those many paths.

This technique is currently facing a lot of challenges, including the fact that it does not scale well [5]. The variables do not have a fixed value but a set of constraints, when there are a lot of variables it becomes difficult to keep track and manage all of them efficiently.

2.4 Antivirus

An antivirus software is a program which aims to detect and remove malware of a system. Originally, the antiviruses were developed to remove viruses, hence the name, but now their goal has been widened to counter all types of malware. The antivirus softwares use mainly the signature-based and the heuristic detection.

2.4.1 Signature

As soon as an antivirus company finds or receives a new malware it registers the signature of the malware in a database. Thereby as soon as a computer sees a new file, the antivirus software computes its signature and sends it to the firm to compare it with the database. If there is a match in the database the antivirus software displays a warning message to the user.

The advantage of this technique is that it is really fast and reliable. In fact, if the malware is known and registered, the signature is in the database so just a request to the database is enough to check and to warn the user.

However, there are multiple problems with this approach. In fact, if the malware is new, or if the database is not up to date, the antivirus identifies it as clean. The second problem is that a hash (the procedure to make a signature) is very sensitive to variants of the code. In fact, if you hash the code "`print("a");print("b");`", it does not end with the same result than the hash of "`print("ab");`". Even if any beginner in computer science could tell you that both give the same behaviour. This was a simple example to illustrate that if there exist multiple variants of the same malicious behaviour, the signatures of all the variants must be registered in the database. Another problem comes from the database of viruses, because you must supply it forever. You have to continuously send the new viruses into the database to maintain it up to date. The last big problem with the antivirus is in the behaviour itself of some viruses. There is a category of viruses, that we call "polymorphic malware" [17], that are composed of 2 parts, the first part is a small code and the other part is just a big part of encrypted text. Once you run the malware, it decrypts the second part and then launch the execution of the part that was hidden with an encryption. As each key results in a different encrypted part, the antiviruses are not able to detect them because each hash is different. Furthermore, you cannot just prevent this behaviour for all softwares because this is a common practice to compress an executable or to obfuscate a proprietary code.

2.4.2 Heuristics

To counter the problem of the variants of the same code with the same behaviour, the antivirus firms try to extract general behaviour from the code statically. As they

are in possession of a lot of malware they can analyse them to extract interesting properties. They generally try to match signatures based on wildcard character (e.g. regular expressions). With this technique they are able to detect if the virus is padded with useless or useful but modified code. It is generally not perfect but with such system you may be able to detect a malware that you have never seen before. And this technique is still very fast because you just have to "scan" the file that is suspicious. However, heuristics are still sensible to obfuscation.

2.4.3 Real-time

One of the features of an antivirus is also to protect your system in real-time. This means that the software monitors your computer and try to search for processes that have strange behaviour. This monitoring is generally invasive for your computer. So invasive that an antivirus considers a monitoring of this kind as malicious. This is one of the reasons why you should not install two antiviruses at the same time on the same system [18].

2.5 Sandbox

A sandbox is a software that implements security mechanisms that separate programs. The purpose is to isolate as much as possible an environment that we allow to be corrupted by a malware, from the working environment. Software developers use sandboxing to test new codes or programs. In cybersecurity, it is often used to execute untrusted or untested programs or code without risking to harm the host. It generally provides a network interface and storage for the code to execute as if it was in a normal environment. The sandbox is the tool by excellence to perform a dynamic analysis.

With the sophistication of malware, it becomes harder to detect them by monitoring the activities and static analysis. Furthermore, malware are obfuscated with many different techniques [19] but this has no impact on dynamic analysis.

There exist a lot of different types of sandbox implementations. There are sandboxes for network access restrictions, some that limit the system call that a program is allowed to do, and some that emulate a complete system with a virtual machine to fake a real system. This is that last category that we will talk about from now on.

To launch an analysis, the sandbox starts an emulated environment, generally with a virtualisation system like *VirtualBox* or *KVM*. Once this environment is set up, it inserts an agent that will monitor the system and the software or document to analyse. Then it launches the virtual environment separately to run the suspicious file. It then waits for the file to finish its execution (or a timeout) and analyses the

system thereafter. It then discovers the processes that have run in the machine, the files that have appeared and possibly the network requests the processes have done. By analysing these data, the sandbox generates a report to the malware analyst, listing all the elements found.

The main weakness of this protocol is that the malware is run in an emulated environment and not in a real one. In fact if the malware can detect by one way or another that it is in a virtual environment, then it can deduce that it is being observed. In such case, the malware has no benefit of executing itself. In fact, if it knows it is not in a real environment, then they are two consequences. First, the infection is of no real use, and second, the security analysts would notice its presence. The malware thus wants to implement the following behaviour:

```
if virtual_environment():
        perform_innocent_actions()
else:
        perform_malicious_actions()
```

This kind of behaviour is called an evasion. Countering an evasion is not an easy task but the sandbox is the best way currently to discover a malicious behaviour from an unknown code without putting its own system in danger. So the security analysts develop constantly new techniques to counter these evasions [20].

Chapter 3

Malware Evasion

Malware developers are constantly trying to develop new techniques to make their malware undetectable. They do not only want to stay hidden on the infected machine, they also want to pass through the automated threat analysis systems and detection. They try to implement the latest evasion techniques to remain hidden from the security analysts. To counter them, the security analysts constantly share the latest evasion techniques, this forces the malware developers to be constantly more creative. This chapter is about known evasion techniques that are used by the malware developers.

The chapter is composed of 4 sections. First, the Section 3.1 gives some implementations of these evasion techniques. The Section 3.2 talks about the different types of evasions possible. Then the Section 3.3 is the longest and describes multiple evasion techniques. Finally, the Section 3.4 follows with evasions specific to *Cuckoo*.

3.1 Evasion implementations

There are multiple communities that want the evasion techniques to be better known. In order to democratize these techniques, they created programs or code samples that implement those ones. I go through the most popular ones of these implementations to explain a bit further their specificities. All these projects are open-source and you can check them on GitHub.

Pafish It is the abbreviation for Paranoid Fish. It is a demonstration tool that use several techniques to detect sandboxes and analysis environments. It is written in C and implements usual tricks seen in malware samples.
https://github.com/a0rtega/pafish

Al-khaser It is a "malware" application developed with good intentions. The purpose of this application is to stress the anti-malware system. It tries multiple common malware tricks already seen in a malware to see if it can evade this system.
https://github.com/LordNoteworthy/al-khaser

InviZzzible It is a tool to evaluate a virtual environment developed by *CheckPoint*. They update regularly this tool with the techniques posted in their repository. Furthermore, more than just the evasion techniques, this tool contains also fixes to counter the evasion of the malware. It supports a lot of different environments, including *Cuckoo Sandbox*. The tool covers a lot of detection techniques and no modification of the source code is needed to add a technique as it can only be made using the JSON configuration files.
https://github.com/CheckPointSW/InviZzzible

3.2 Evasion types

To detect and go unnoticed in the automated tools of the malware analysts, the malware developers have multiple choices. I develop the two main approaches that they can use to spot if they are observed.

Virtualization detection This approach aims to detect the virtualization infrastructure that is used to run the malware. It focuses on the artefacts left by the virtualization machine (*VirtualBox*, *VMWare*, *QEmu*,...) no matter the type of analysis system.

Sandbox detection This approach aims to detect the artefacts left by the different analysis systems (such as *Cuckoo*) in the environment, like the presence of specific files or processes. By adding some of these hints, the malware can build a degree of confidence that it is being observed by an analysis software.

3.3 Evasion Techniques

All the techniques that I explain and more can be found in the Malware Evasion Encyclopedia [21]. This site is a mirror for a Github repository [22] where everyone can make a pull request to add some evasion techniques or to enrich existing techniques with some details.

3.3.1 Time-delay

This evasion is one of the simplest evasion technique to understand. It consists of calling the sleep function of the system to wait a certain amount of time. This

delay would be large enough for the malware to do nothing during the observation time that is reserved in the Sandbox.

In practice, the malware can make a call to the function `NtDelayExecution` or `Sleep` for several minutes. Moreover, there are some variants of this technique and some malware attempts to crash the hooking system of one of these functions by putting an "infinite" delay as argument. The malware can also request the time before and after the sleep, and check that the correct time has elapsed on the system.

The countermeasure for this technique is a bit more difficult than the technique itself. In fact, you have to check the argument of the sleep but also the total number of time this function was called, because you do not want to let the malware sleep a thousand times half a second which would result in a total idle time of more than 8 minutes. However, in addition you have to keep track of all these calls and adapt the response you would give if the malware asks for the current time of the system.

3.3.2 Filesystem

The principle of this evasion is that there are some files or folders that are present in a virtual environment or put there by the sandboxing mechanisms. As these paths are usually not present in a classical environment, the malware can identify the nature of its environment.

In practice the malware tries to search paths such as the network adapter in the drivers (`c:\windows\system32\drivers\prleth.sys`), special files of *VirtualBox* (`c:\windows\system32\vboxdisp.dll`) or files from *VMWare* and others more general. Concerning the folders, it can look for a folder containing the guest additions (for *VirtualBox*: `%PROGRAMFILES%\oracle\virtualbox guest additions`). Some other variants include verifying if the name of the executable itself is specific or if it contains a specific string like `sandbox` or `virus` in its path as we can see in the following code sample from the Pafish project [23].

```
int gensandbox_path() {
        char path[500];
        size_t i;
        DWORD pathsize = sizeof(path);

        GetModuleFileName(NULL, path, pathsize);

        for (i = 0; i < strlen(path); i++) { /* case-insensitive */
                path[i] = toupper(path[i]);
        }
```

```
        // some sample values from the table
        if (strstr(path, "\\SAMPLE") != NULL) {
                return TRUE;
        }
        if (strstr(path, "\\VIRUS") != NULL) {
                return TRUE;
        }
        if (strstr(path, "SANDBOX") != NULL) {
                return TRUE;
        }

        return FALSE;
}
```

For the countermeasure, it is recommended to hook the function in *Windows* `GetFileAttributes(path)` and to return appropriate results if a file specific to a virtual environment is checked.

3.3.3 Registry

In this case, the malware tries to search registry keys that are usually not present in common host but which exist in virtual environments. Some of these methods are not completely reliable. In fact, some keys are present both in the host and in the guest system. Knowing that a system may have virtual machines installed for other purposes, a check of this kind may be positive for legitimate systems. In this case, the malware combines this check with others to confirm that it is executed in the virtual environment.

In practice, the malware requests the keys of the system matching a regex and it tries to see if he receives matching keys. These keys may be from *Hyper-V* (`HKLM\SOFTWARE\Microsoft\VirtualMachine`), *VirtualBox* (`HKLM\HARDWARE\ACPI\DSDT\VBOX__`), *Wine* (`HKCU\SOFTWARE\Wine`) and others. However, it can also request the value of a key and try to match this value with a string corresponding to a virtualization system. For example if the key contains as value `VBox`, `VMWare` or `QEMU`, it is highly probable that you are in a VM.

The countermeasure is quite simple, you have to know what are the sensible keys and return non sensitive values if these keys or their values are requested.

3.3.4 Generic OS queries

This technique consists of possible verifications of the computer configuration in general. Usually hosts have meaningful usernames or computer names. If the

malware detects that these names are generic or default user/computer names given by virtual environments, it knows that it is located in a VM. However, there are other possibilities in the configuration of the computer. You can in fact be almost sure that if the available RAM of your computer is below or equal 1GB it is certainly a VM as all recent computers have more than that nowadays.

In practice, the malware can verify if the username, the computer name or the host name is specific. Then it can inspect if the RAM, the screen resolution, the number of processor, the quantity of monitors or the hard drive is non usual for a modern computer. It can also check if the uptime from the computer is small or if it is up from a long time, like this code sample from the InviZzzible project [24]:

```
bool Generic::CheckSystemUptime() const {
        const DWORD uptime = 1000 * 60 * 12; // 12 minutes
        return GetTickCount() < uptime;
}
```

The countermeasure for these techniques are not really obvious because most of the function calls used to get these features may totally be legitimate. You have to pay particular attention to the return value of all these functions and be sure that you do not return a value that may be interpreted as a fake environment.

3.3.5 Processes

The virtual environments generally launch specific helper processes. These specific processes are loaded into the processes address space but are generally not executed in usual host OS.

In practice, the malware asks a snapshot from the running processes and then checks if some processes as `vboxservice.exe` from *VirtualBox* or `vmware.exe` from *VMWare* are present. It can also search for `WPE Pro.exe` which is a sniffer but is often used in VM. Another technique consists of checking the libraries, there may be `sbiedll.dll` for *Sandboxie* or `vmcheck.dll` for *VirtualPC*. The malware can also check if specific functions are present in specific libraries. For example, a function that it may verify is `wine_get_unix_file_name` for Wine.

The countermeasures are different for the processes, the libraries or the functions. For the processes, you have to exclude the target processes from enumeration or to terminate them. For the libraries, you must exclude them from enumeration. And for the functions in libraries, you must hook appropriate functions and compare their arguments against possible targets.

3.3.6 Global OS objects

The principle in this case is to find some general objects that are not present in a usual host but that are in virtual environments or sandboxes. This is quite general and all artefacts present in the OS that cannot be in another category belong certainly to this one.

In practice, the malware can for example search for global mutexes by trying to create or open a mutex, the malware can see if this one exists in the system. It can look for `MicrosoftVirtualPC7UserServiceMakeSureWe'reTheOnlyOneMutex` in *VirtualPC* or `Frz_State` used with the *DeepFreeze* application. The malware can also look for virtual devices present only in virtual environments like `\\.\VBoxGuest` for *VirtualBox* or `\\.\vmci` for *VMWare*. There are also some pipes that are present for example in *VirtualBox* (`\\.\pipe\VBoxMiniRdDN`). Another technique available, for the malware, is to check for global objects like devices or drivers.

The countermeasure for this kind of attempt is just to look if one known object is requested and to return appropriate results. Sometimes it may be necessary to stop some devices.

3.3.7 OS features - Debug privileges

When we run the malware in a sandbox like *Cuckoo* (or in a debugger) the process may have a debug privilege. The malware inherits this debug privilege from the parent process. The evasion technique consists of trying to detect this privilege and to stop the process if it has this particular privilege.

Concretely, the malware tries to open system processes like `csrss.exe` and tries to terminate them. With the administrator rights these attempts would end in a failure because even an admin cannot terminate those processes. However, if the debug privilege is enabled for the malware process, it can terminate any processes even the system's ones.

To countermeasure this technique, the goal is to hook the `OpenProcess` system call and to track if the requested process ID is a critical system. If it is the case, you should return an error to the malware.

3.3.8 UI Artefacts

These techniques rely on the windows opened in a virtual environment and not in a usual host, or vice versa. We can also say that some host OSs contain a lot of windows while VMs and sandboxes keep a minimum of opened windows.

In practice, the malware can check if windows with some class names are present. For example, there are windows with the name `VBoxTrayToolWnd` in *VirtualBox*. However, the malware can also verify the number of windows in the OS, if a host

have more than 10 top level windows, the malware is pretty sure that it is not in a virtual environment. For example, the following code verify if windows with class names containing `VirtualBox` are present, this sample comes from the Al-khaser project [25]:

```
BOOL vbox_window_class() {
        HWND hClass = FindWindow(_T("VBoxTrayToolWndClass"), NULL);
        HWND hWindow = FindWindow(NULL, _T("VBoxTrayToolWnd"));

        if (hClass || hWindow)
                return TRUE;
        else
                return FALSE;
}
```

The countermeasure for this problem is in two parts. First, you have to exclude (or modify the names of) all windows that have these particular names from the list of windows that the malware can retrieve. Then, you should create fake windows in the system, to have a sufficiently big number of opened windows to fool the malware.

3.3.9 Network

These kinds of evasions all rely on network to identify their environments. The network parameters are often different in a VM than in a usual host but the use of the network is also different. It is kind of obvious not to let a malware have access to the network when we just want to test it and we do not want it to spread.

In practice, a malware can look for unusual properties in the network configuration. For example, it can check the MAC address of the environment and see if it starts with specific combinations corresponding to VMs (like `08:00:27` for *VirtualBox*, `00:05:69` for *VMWare* or `00:1C:42` for *Parallels*). The malware can also look for an adapter name that may be specific like in *VMWare* where the adapter is named `Vmware`. The malware could also inspect the provider name, for *VirtualBox* this name is explicit and contains `VirtualBox` in it. The malware can also verify the network security perimeter. By making a request to some website the malware can identify the IP address of the machine it is running on. And most importantly, there is a website which returns information about this IP address by a little trick [26], this information contains the country, the city and even the organization associated with the address. If there are some specific strings in the response of this request the malware can know that it is in a VM (these strings can be `anonymous`, `hosted`, `VMVault` and lots of others).

The countermeasures are basically to change the network parameters for the virtual environments and to emulate a network that gives appropriate responses to the malware without allowing it to spread.

3.3.10 CPU

There are multiple methods possible using the CPU for a malware to evade. All these techniques are based on getting particular information about the CPU or on executing a sequence of instructions which behave differently in a VM than in usual host OS.

Vendor String In practice, a malware can first inspect the vendor ID string. Via a CPUID instruction, the malware can identify a vendor but can also verify if it is in a virtual environment because they return string like `VBoxVBoxVBox`, `VMwareVMware`, `KVMKVMKVM`, or others of the same kind.

The countermeasure for this technique would simply be to hook the called function and to return an appropriate string.

Hypervisor via CPUID instruction The malware can check if it is being run in Hypervisor thanks to the CPUID instruction. When you set the parameter EAX to 1, the 31st bit in ECX returned by the instruction indicates if the program is being run in Hypervisor.

The countermeasure would simply be to set to 0 the 31st bit in ECX in all cases.

Exotic instructions Sometimes subsets of CPU instructions are not supported by emulators. If you try to use some subsets such as MMX instructions an exception may be thrown when the malware tries to use it.

The countermeasure here is not really simple, you should use a VM that is able to comply with all the instructions possible.

3.3.11 Hardware

This group of techniques implies to gather information about the hardware. The malware can check for the hard drive or the audio device for example. With some info about them, it may be able to tell if the hard drive is real or not.

In practice, the malware can look for the hard drive properties and more specifically for the name and the vendor ID. Some classical names for hard drive in Virtual Environment are `QEMU`, `VBOX` or `VMWare`. And these strings may be also present in the vendor ID. The malware has other possibilities, it can also verify if the audio device is present or not. If it is absent it is highly probable that it is not

in a usual host. The malware could also check if the temperature of the CPU is available [27], if it is not, it may be in a VM.

The countermeasure is to rename hard drives to have names unknown by the malware. You also have to add an audio device and to output some meaningful information about the CPU temperature when it is asked.

3.3.12 Firmware tables

This method consists of verifying specific strings in the (SMBIOS) Firmware Table. Malware can retrieve firmware tables via `SYSTEM_FIRMWARE_TABLE_INFORMATION` object. The memory areas to dump these tables are different depending on the OS version so the technique must be adapted for each OS version.

In practice, the malware can verify if strings like `Oracle`, `VMware` or `S3 Corp.` (for *VirtualPC*) are present in the the raw firmware table. However, the malware can also check for the SMBIOS firmware table, and it can contain the same kind of strings in it. The following code verifies if one of these specific strings is present in Raw Firmware Table for *Windows Vista+*, this comes from the VMDE project [28]:

```
// First, SYSTEM_FIRMWARE_TABLE_INFORMATION object is initialized
↪  in the following way:
SYSTEM_FIRMWARE_TABLE_INFORMATION *sfti =
↪  (PSYSTEM_FIRMWARE_TABLE_INFORMATION)HeapAlloc(GetProcessHeap(),
↪  HEAP_ZERO_MEMORY, Length);
sfti->Action = SystemFirmwareTable_Get;  // 1
sfti->ProviderSignature = "FIRM";
sfti->TableID = 0xC0000;
sfti->TableBufferLength = Length;

// Then initialized SYSTEM_FIRMWARE_TABLE_INFORMATION object is
// used as an argument for the system information call in the
// following way in order to dump raw firmware table:
NtQuerySystemInformation(SystemFirmwareTableInformation,  // 76
        sfti, Length, &Length);
```

The countermeasure for this evasion depends on the OS version. For *Windows* versions older than *Vista*, you should change the memory content of `csrss.exe` at given addresses. On newer versions, you should hook `NtQuerySystemInformation` for retrieving the `SystemFirmwareTableInformation` class and clean the structure of the strings that may be targeted by the malware.

3.3.13 Hooks

The techniques described here have two purposes. The first one is to detect user actions in order to be sure that there is a user which is present on the machine where the malware is executed. The other one is to check if there are unusual hooks installed on the host.

In practice, the malware can use the technique described in the paper "Hot Knives through butter" [29] which consists of setting a hook to detect a mouse click. If there is in fact a click, the malware considers that there is a user on the same machine and that it is not being executed in a VM. However, the malware can also look for the presence of other hooks by reading to specific addresses in memory. In a virtual environment, it is most likely that some functions are hooked to gather data and statistics during emulation, the malware tries to detect the hook on these functions. The functions that are often verified are `ReadFile`, `DeleteFile` or `CreateProcessA/W`. Once the malware has read the memory, it can check if there is a hook with multiple techniques. For example, it can compare the first two bytes with `\x8B\xFF` that is the instruction `mov edi, edi` which usually begins a kernel32 function. An other technique would be to inspect the usual combo `push/ret` for execution redirection at the end of the function. Multiple techniques to detect user-mode hooks are explained in "Defeating Userland Hooks (ft. Bitdefender)" [30].

The countermeasure is different for the two techniques. An implementation of a simple emulation module for the mouse movement/click is enough. As for the other, you may want to set kernel mode hooks or to use stack routing to implement the hooking.

3.4 Cuckoo Evasion

The evasion techniques described in this section are all about evasions that are specific to *Cuckoo Sandbox*.

3.4.1 OS features - Unbalanced stack

This technique was presented at Virus Bulletin 2016 and is described in the paper "Defeating Sandbox Evasion" [31]. This evasion technique was discovered for *Cuckoo* in particular but can be extended to other Sandboxing systems.

The "Cuckoo Monitor" module allows to hooks some functions from libraries or system calls. In this specific architecture, to track processes behaviour, the hook is called before the original function. The hook takes some place on the stack above the called function and the consequence is that the space used by the hooked function may be larger than the original function call. The malware can

thus exploit this information to try to infer if the function call it is doing is hooked or not.

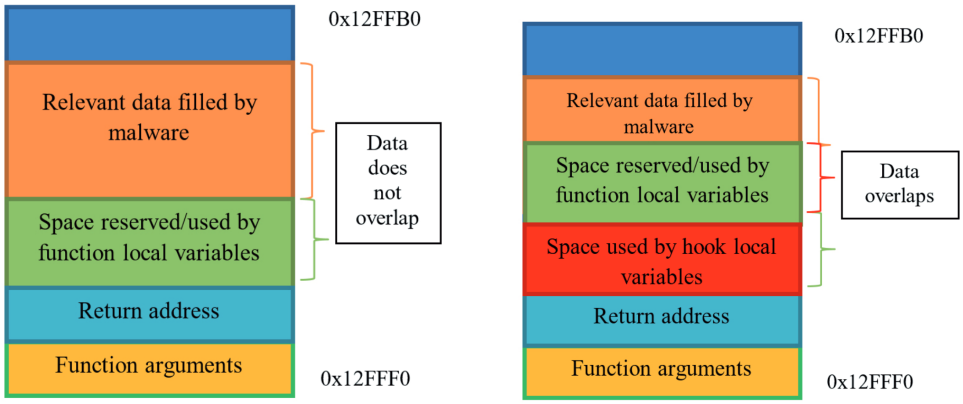

Figure 3.1: Stack seen by the malware without and with hook, from [31]

In practice, the malware moves the stack pointer to a lower address but just the necessary space to store the function arguments, local variables and return address of the target function. Then the malware fills the space below with some data. It then moves the stack pointer to its original location and calls the target function. After the function returns, the malware then looks if the data placed in lower address are still unchanged. If the target function was hooked, the data were certainly overwritten by the function call as the hook has shifted the target function space for local variables. So if the data are untouched the malware continues, but if the data are modified the malware knows it is being observed.

The countermeasure for this evasion is to change the behaviour of the Cuckoo Monitor. It should instead use a two-stage hooking process. In the first stage, it can move the stack pointer towards lower addresses (with an offset that is enough to go over the data of the malware). Then, the function's argument are copied under the new stack pointer. Only after this first step, the "Cuckoo Monitor" can perform the real hooking step. Like that the data filled in by the malware are still on the upper stack addresses.

3.4.2 Network

This technique consists of scanning all outgoing established TCP connections to verify if there is a connection to the TCP port 2042 which is the one used by the "Cuckoo Result Server". To counter this technique, you can change port of the result server in the configuration. The ideal solution would be to randomize in one way or another this port.

Chapter 4

Cuckoo Sandbox

Cuckoo Sandbox is the leading open source automated malware analysis system. It began in 2010 and reached now its 2.0.7 version. This project has now more than fifteen hundred forks and forty-two hundred stars on Github [32]. The version 2 was released in 2016 and *Cuckoo 3* is currently under development, in order to be entirely *Python 3* compatible because *Python 2* is deprecated since the beginning of 2020. *Cuckoo* applies the concept of malware sandboxing and is able to provide a report outlining the behaviour of the submitted file. It can analyse different malicious files like executables but also office documents, pdf's, emails, url's, etc. The analyses can take place under multiple environments from *Windows*, *Linux* or *MacOS*. The environment that *Cuckoo* uses to execute the malicious files has to be made by the user. *Cuckoo* was made quite modular so it is easily customizable. You can modify the environment of analysis, the processing of the results or the reporting stage.

4.1 Installation

The installation of *Cuckoo 2* is quite a pain. All the steps are described in the documentation [33]. The issues begin with the table of contents which is too big to fit on one screen and it may constitute an obstacle for some people. It requires a lot of packages, particularly in *Python 2*. Some of the packages require functionalities that are only available when you install them from source. It is also required to install `apparmor` to allow the capture of the network traffic with `tcpdump`. You also need a virtualization software like *VirtualBox*, *KVM* and *VMWare*, others are also possible but *VirtualBox* is the default one.

The mains steps of the installation of *Cuckoo* consist of:

1. Installing the requirements.

2. Installing *Cuckoo*.
3. Indicating a location for the Cuckoo Working Directory (CWD).
4. Modifying the configuration files.
5. Preparing the guest.

4.2 Guest system

Cuckoo allows to have either a virtual machine or a physical machine as guest. In both cases you must go through multiple steps in order to get your guest system ready:

1. Installing a system on your physical machine or creating your virtual machine.
2. Installing the requirements in your machine.
3. Configuring the network.
4. Installing the "Cuckoo Agent" in the machine.
5. Saving the state of your machine in a clean state in order to return to it after an analysis.

Cuckoo can have one or multiple machine guests indicated in the configuration files. To start an analysis, *Cuckoo* takes each machine mentioned in the configuration files and starts them from the given clean state. If you do not precise a snapshot for a VM, *Cuckoo* simply takes the last one. At least one snapshot is required for each VM that you give, if you do not have a snapshot, *Cuckoo* does not start the analyses on this guest. If you have multiple machines, you can specify the machine on which you want to analyse the malware.

4.3 Architecture

Cuckoo Sandbox consists of a management software which handles sample execution and analyses. The "Cuckoo Host" is responsible for the management of the analyses and the guests. It starts the virtual environments, launches the execution of the malware submitted, intercepts the network traffic going out of the machines and generates the reports. Each analysis is launched in a new environment either on a physical or virtual machine. Thereby the host runs the core component of the sandbox, while the guests are the isolated environments. The Virtual Network

connects the machines with the "Cuckoo Host" and isolate the machines from the internet while giving them the impression to be able to communicate with the real world.

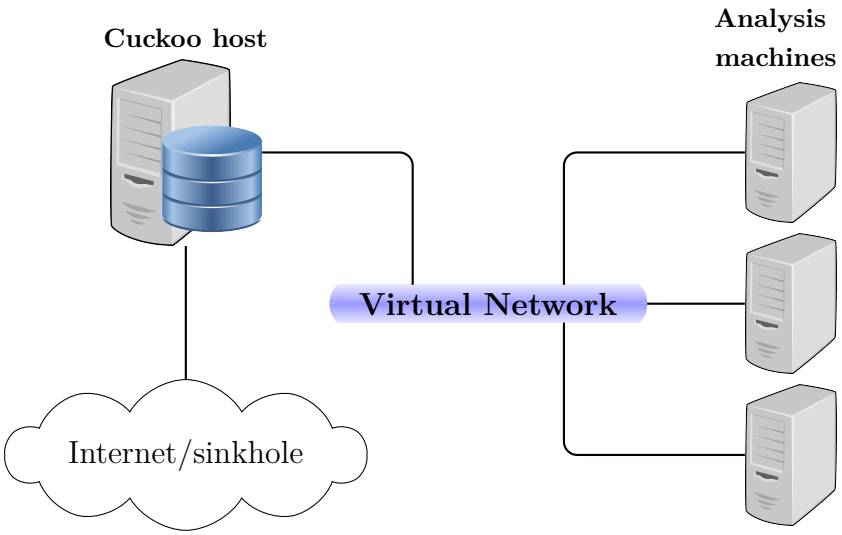

Figure 4.1: Overview of the architecture of *Cuckoo*

4.4 Modules

The great advantage of *Cuckoo* is its modularity and easy customization thanks to its modules. *Cuckoo Sandbox* is composed of a core which starts and handles all the modules. The main modules are the processing module which handles the analyses of the malware, the reporting module which generates the reports, the machinery module which handles the virtual machines and the web module which handles the "Cuckoo Web Interface". There are other modules but they are less important and not needed to understand how *Cuckoo* works.

Cuckoo starts and parses the arguments, if needed it reads the configuration files to configure its modules because some modules (like reporting module) behave differently depending on the configuration. Once the needed modules are up, it waits for submissions if they are none in queue. Once a submission arrives, the core asks the machinery module to start the machines, then asks the processing module to start the analysis of the malware and finally asks the reporting module to generate the reports. These modules can operate in parallel because *Cuckoo* is multithreaded.

Part II

Implementation

Chapter 5

How to deploy Cuckoo?

As already said (see Section 4), *Cuckoo* is the leading open source malware analysis system. Facilitating its deployment can be useful to give a chance to students in security classes to test the program without having to handle the difficult and long journey consisting of installing and configuring *Cuckoo*. So in order to facilitate the deployment, *Docker* was an easy choice, as it is the most used container technology. This section is about all the steps that were necessary to deploy *Cuckoo* with *Docker*.

5.1 Docker

Docker is a product that uses OS-level virtualization to deliver containers. A container is a standard unit of software that packages up code and all its dependencies. As opposed to Virtual Machine, *Docker* does not install any guest OS, it uses the kernel of the host. That makes it more lightweight than any VM with the same libraries and software. Thanks to this lightweight, multiple containers can run on the same server at the same time. Due to their success with Unix, *Docker* has made a partnership with *Windows* and there exists now a "Docker Engine" that allows *Windows* to run containers from *Docker* as well.

In concrete, you must create a "Dockerfile" which contains a starting point, it is generally another docker image from the "Docker Hub", then a list of steps to install or modify your image. You create the image by building the Dockerfile and with this image, you can run one or multiple instances of the image. These instances are called the containers.

The containers do not include any part of the kernel or the modules inside, they interact with the kernel through system calls. Thanks to this property the containers are light and portable.

5.1.1 Dockerfile

As said previously, the Dockerfile contains all the steps necessary to create the image that will be used to create containers. The Dockerfile I have made for Cuckoo begins with the following lines:

```
1 FROM ubuntu:18.04
2 LABEL maintainer="Luis Tascon Gutierrez
  ↪ luis.tascon@student.uclouvain.be"
3 LABEL description="This image aims to have a container to use
  ↪ Cuckoo Sandbox."
4 ENV DEBIAN_FRONTEND=noninteractive
```

The first line indicates that we use the image "ubuntu" version 18.04. In fact, I noticed that if we just use the image ubuntu, *Docker* takes the last version of ubuntu available which is the 20. Moreover, the version 20 of ubuntu has *Python 3* as default version instead of *Python 2*. This is not ideal, because *Cuckoo Sandbox 2.0.7* has still *Python 2* dependencies, and changing the *Python* default version from 3 to 2 requires some more steps to do [34]. The ENV command is used to set an environment variable, in this case it allows the installations of some packages to be made without interactions.

Then different requirements are installed, depending on the final configuration some of them may not be needed. So the Dockerfile installs the requirements for Cuckoo Monitor, the requirements for the documentation and `PostgreSQL` to use it as database. `pydeep` and `guacd` are also installed. After the installation of these packages, a configuration of `tcpdump` is needed then `yara` and `volatility` are installed from source. It configures `tcpdump` to allow *Cuckoo* to capture network traffic. Then it is needed to install `m2crypto`, which is a library used by *Cuckoo*.

Once this is done, the Dockerfile installs *Virtualbox* from source, this is not an easy task because most virtualization software are not aimed to be installed only with the command line. Furthermore, if you try to use `virtualbox` once you have installed it get the following message:

```
WARNING: The vboxdrv kernel module is not loaded. Either there is
↪ no module available for the current kernel (3.11.0-22-generic)
↪ or it failed to load. Please recompile the kernel module and
↪ install it by

        sudo /etc/init.d/vboxdrv setup

You will not be able to start VMs until this problem is fixed.
```

You can in fact try to execute the command `sudo /etc/init.d/vboxdrv setup` but this does not change anything because as already explained in section 5.1, a container cannot contain any kernel module and `vboxdrv` is a kernel module. So this driver should be loaded at creation time of the container.

Then the instructions download the "VirtualBox Extension Pack" but do not install it in the Dockerfile, in fact to install it you need to have the driver `vboxdrv`. So, the extension pack has to be installed at start-up. However, this is not the only problem, if you want to install automatically the extension pack, then you need to accept the license. This licence can only be accepted by command line if you have already accepted it manually, then you can copy the code and use it in your command.

```
License accepted. For batch installation add
↪ --accept-license=dead(...)beef to the VBoxManage command line.
```

Then I installed *MongoDB* to be able to use the web version of *Cuckoo*. This is again a special case, only the `mongodb` shell needs to be installed in the image and the `mongodb server` needs to be installed on the host machine outside of the container. To do so, I must have the same version of `mongodb` in the host and in the container. In order to sucessfully install the same version in the container and in the host, I created a template of the Dockerfile where I put a special string "`__mongodb-version__`". This string is there to be replaced with a `sed` command that automatically generates the final Dockerfile.

After all these steps, *Cuckoo Sandbox* is ready to be installed. The Dockerfile contains the installation of the "Cuckoo Monitor" from source. Then the Dockerfile installs *Cuckoo*. Finally, it downloads the community archive without installing it, because *Cuckoo* cannot start to make the installation if *VirtualBox* does not start.

The Dockerfile can be found by following this link.

5.1.2 Some errors

After struggling to integrate *VirtualBox* in *Docker*, I discovered sometimes directly, sometimes after weeks of usage that there were still some errors in the Dockerfile. Here is a list of the errors I have encountered and how I managed them.

M2Crypto This is a library used by *Cuckoo*. At first I tried to install it with `apt-get` but if we do so, the version 0.27 is installed and *Cuckoo* only supports the version 0.24, if this version is not installed it may cause an error while analysing malware. So I had to use `pip` to install this package to the exact version.

Volatility It is used by *Cuckoo* to analyse the memory dumped. However, to install *Volatility* you need some libraries too, `distorm3` is now at its version 3.5 but volatility does not support this version so I had to install the version 3.4.4 as it is advised on the issue 719 on github [35].

Cuckoo Monitor This is used by Cuckoo to add some signatures of functions to the hooks. When you use the `make` command to build it, there is an error. The correction to this error was suggested in the pull request 71 of the github [36] but is still not merged. So I used the command `sed` to modify in place the involved file and correct this error.

Cuckoo I also corrected some errors I have encountered using *Cuckoo* directly in the Dockerfile in order to have the corrections directly for all the containers.

5.1.3 Creating the container

Once the image is created thanks to the Dockerfile, it is time to create the container itself.

In order to use *VirtualBox* we need to mount the `vboxdrv` as already said. However, we also need some flags to be sure that everything works fine. We need to add the flag `--privileged` to have access to the kernel driver, without this flag, `vboxdrv` is mounted but not accessible. And the flag `--network=host` is also needed to be able to use the `vboxnet0` interface of `vboxmanage` on the host. For the network, there are probably other solutions but they would be much more advanced and error prone.

To easily customize the containers with different configurations of *Cuckoo*, I mount directly the "Cuckoo Working Directory" (CWD) in the container. I also mount different folders like a folder that contains the malware to be analysed, a folder for the `ova` file containing the virtual machine to load (see Subsection 5.1.4), and a last one for the monitors.

The complete command to create the container can be written:

```
docker create --name cuckooContainer -i \
        -v "path/to/CWD":/home/cuckoo/.cuckoo \
        -v "path/to/malwares":/home/cuckoo/malwares \
        -v "path/to/vms":/home/cuckoo/vms \
        -v "path/to/monitors":/home/cuckoo/mymonitors \
        -v "path/to/others":/home/cuckoo/others \
        -v /dev/vboxdrv:/dev/vboxdrv  \
        --network=host \
        --privileged \
        cuckoodocker:latest
```

5.1.4 Guest Machine

In order to have the virtual machines in the containers, I have opted for mounting them via a file. To do so, I simply mount a folder containing one or multiple `ova` files. Once the container is started, I use `VBoxManage` to import the `ova` file.

As already said (see Section 4.2), *Cuckoo* needs at least one snapshot, so once the VM is imported, you also need to take at least one snapshot because it is not possible to include one in the `ova` file.

5.2 Scripts

All these instructions are quite complex but they are all necessary to have a container working well. As the goal of this is to help students using *Cuckoo* for the first time, some scripts could be easier for them to use, so I created them. In this section, I explain which scripts I made and how they work.

5.2.1 Guest Machine

The preparation of the VM is long and requires a good comprehension of Virtual Machines. There are multiple steps needed to complete a VM as it is described in the documentation of *Cuckoo* [37]. You have to deactivate the firewall, the *Windows* update, the UAC and configure the network of the VM to made it reachable from the "Cuckoo Result Server". Thanks to some developers, there exists an open source tool, *VMCloak*, that handles all of this for us [38].

VMCloak is an automated virtual machine generation and cloaking for *Cuckoo Sandbox*. This tool creates and prepares Virtual Machines that can be given to *Cuckoo*. It is quite stable and can create machines from *Windows XP* to *Windows 10*. It requires few dependencies and is really easy to install.

I made a quick script to use *VMCloak*, the principle is that you have to mount an `iso` file on your system. Then you just list the softwares that you want to be present on your guest and it installs them for you on a Virtual Machine. The script using *VMCloak* can take a long time but it does not require any interaction.

```
sudo mount -o loop,ro /path/to/win7ultimate.iso /mnt/win7
vmcloak-vboxnet0
vmcloak init --win7x64 name0
vmcloak install name0 adobe9 (...) ie11
vmcloak snapshot name0 cuckoo1 192.168.56.101
```

However, this script is an exception in this section, because it will normally not be useful for students as an `ova` file can be provided.

This script, `vmcloak_win7.sh`, is available by following this link.

5.2.2 Setup

This is the first script that the student has to use. As already said, in order to have the container working perfectly, some requirements are to be installed on the host machine (such as *Docker*). This script verifies that all requirements are installed and, if it is not the case, it installs them. It also generates the Dockerfile from the template used to modify the `mongodb` version (see Subsection 5.1.1). At the end of the script, it builds the Dockerfile to create the image.

The script, `setup.sh`, can be found by following this link.

5.2.3 Creation of the container

This script verifies if the container does not already exist. If not, it creates it (see Subsection 5.1.3) and performs all actions necessary to start *Cuckoo*. Then it installs the community files of *Cuckoo* and the VirtualBox Extension Pack, imports the VM and takes the snapshot (see Subsections 5.1.1 and 5.1.4).

The script, `create_container.sh`, can be found by following this link.

5.2.4 Start the container

This script begins by verifying if the network interface **vboxnet0** is up, it creates it and puts it up if needed. Then it verifies if the container exists, if not, it calls the script described earlier (see Subsection 5.2.3). Finally, it starts the container.

The script, `start_container.sh`, can be found by following this link.

5.2.5 Web start

This is the second script that a student has to use. It starts by verifying if the port of the "Cuckoo Web Interface" is available and if it is not the case it prints a message to ask the user to handle the problem. Then it starts a container with the script from 5.2.4 then it executes *Cuckoo* and its web interface separately in background and pipes their outputs to a log file. It waits 3 seconds to let the web interface start then it verifies the last line of the log files in order to tell the user if a problem occurred or if everything went well. If everything worked perfectly, the student can now access on his browser the Cuckoo Web Interface at the address `localhost:8000` which gives the screen that we see on Figure 5.1.

The script, `web_start.sh`, can be found by following this link.

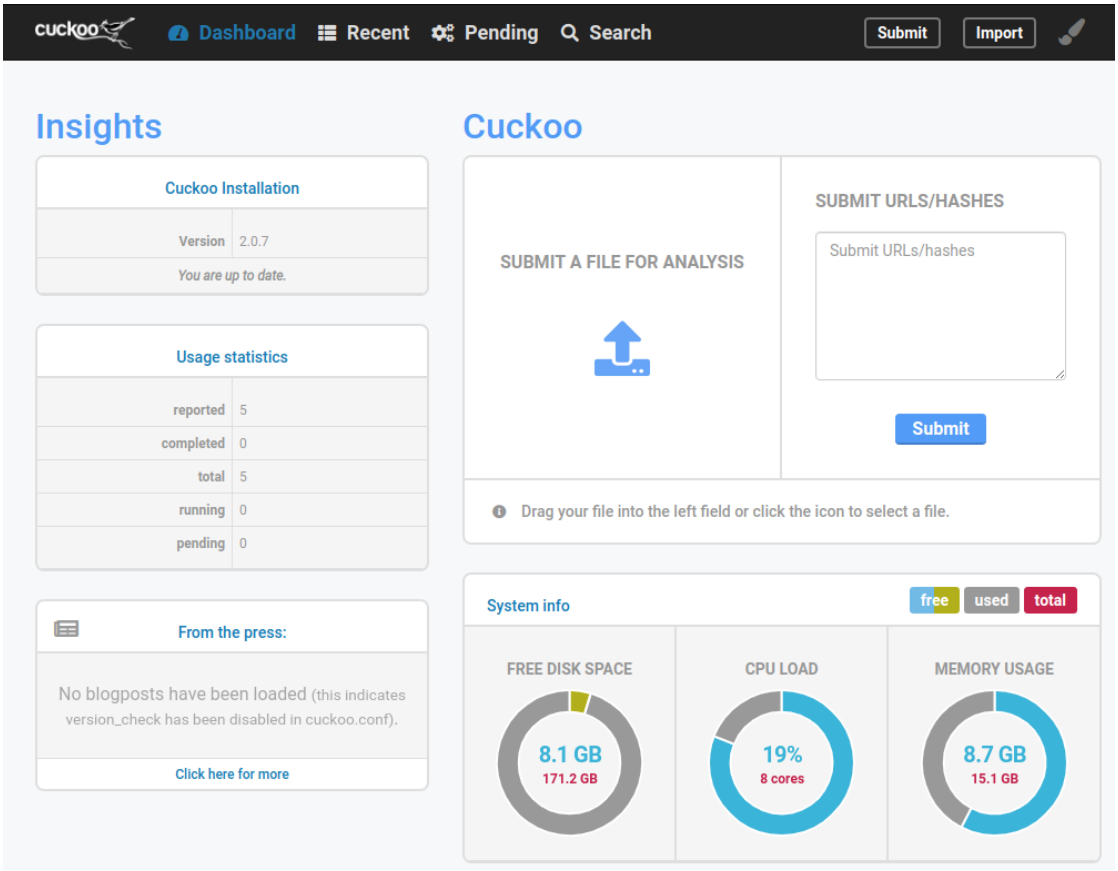

Figure 5.1: Cuckoo Web Interface

5.2.6 Stopping and Deleting the container

I have made the same kind of scripts to stop the containers and to delete them. The student can thus stop *Cuckoo*, its web interface and the container by executing a simple script.

The scripts, `stop_container.sh` and `delete_container.sh`, can be found by following respectively this link and that link.

5.2.7 Clean

This script is for a student who wishes to remove everything that was installed by the setup script (see Subsection 5.2.2). The student is asked for each single package if he wants to uninstall it and the script removes the packages accordingly.

The script, `clean_all.sh`, can be found by following this link.

Chapter 6

Analyses of malware

6.1 Preparation of the analyses

For a part of my thesis, I was asked to analyse a bunch of malware in order to be able to see what *Cuckoo* is capable of. So I contacted the helpdesk of the INGI department to get a machine on which I could run some analyses. They did so, then I gave an SSH private key to connect myself to the server and I received root access to the machine. Unfortunately, the lockdown began at that moment and I noticed thereafter that the machine they gave me was a virtual machine QEMU and that the deployment with *Docker* (see chapter 5) does not work with it. This section is all about how I managed to launch the analysis of thousands of malware and store the results on my computer which has limited amount of free memory.

6.1.1 The problem

As just mentioned, because of the lockdown, it was not possible any more to ask for a physical machine, so I had to find a way to launch the analyses on my computer. The first time I did it, 100 malware were analysed during the night. That was not so much but enough for a first time. When I opened my computer in the morning, I discovered that the memory used to store the results of 100 malware was about 42GB. As I was asked to analyse multiple thousands of malware, I had to find a way to store only the results that I needed.

6.1.2 The configuration of Cuckoo

To take less memory *Cuckoo* has already multiple parameters that can be tuned in the configuration files. I review rapidly what I changed in these files.

In the `cuckoo.conf` file you can modify the configuration to delete the copy of the binary after the analysis has finished. The screenshots can be useful if you

want to see personally what was the behaviour of the malware but I did not need them so I could prevent them in the `processing.conf` file. In the same file you can also specify if you want to delete the memory dumped after the analyses to save disk space.

6.1.3 Concentrate on what is needed

To ease the gathering of the needed information from all the files, I modified also the `reporting.conf` file. First, I modified the parameters to generate only the report in a JSON format (I noticed later that this same file could be directly saved in `mongodb`). However, even this file was too much information than what I needed and furthermore some of these JSON files were almost 1GB of memory and that was not acceptable to keep so much amount of data for one single malware.

In order to register only the needed part of this JSON report, I made a python script to parse the file and to extract only the information that I was interested in. This script is named `store_in_db.py`. It lists all the analyses from a CWD and it parses the generated reports. Once a report is parsed, it verifies if all the necessary keys are present, and it registers in a new dictionary the interesting keys (which are "info", "signatures", "target" and "debug") with their values. Then it connects to `mongodb` and registers the results in a database. This targeted information from the report allowed me to store the results I wanted without taking too much memory.

The script, `store_in_db.py`, can be found by following this link.

6.1.4 How to analyse all of them efficiently

Once I have done this, I faced another problem. Let's say I have a thousand malware on my computer, I have the time to analyse 200 of them per night but I need my computer for other usage during the day.

The first thing that can come to mind is to submit the 1000 malware the first day and to ask cuckoo to analyse 200 of them in per night. This is possible but I need to remove the files that are generated by the analysis (see Subsection 6.1.1). And the amount of available memory is not large enough to have 200 results of analysis in one time, so it must be a smaller number. I can make a script to ask *Cuckoo* to store the results and clean the memory every 20 analyses (see Subsection 6.1.3). *Cuckoo* has already a command that allows us to clean the memory used by the generated analyses reports, this is the command `clean`.

This command is great and easy but the problem is that it also removes all submissions that have been done. So I modified the `store_in_db.py` script to automatically remove all files generated by an analysis if its results have been well registered in the database.

To make all of this easier for me, I have made a bash script that handles all of this and even some corner cases that I still did not talk about. This script takes as parameter the number of analyses it has to perform. Then it performs the following operations:

1. it verifies that it has as argument a number greater than 0,
2. it sets multiple variables and constants,
3. it cleans *Cuckoo* and creates some folders for the logs and results if needed,
4. it lists all malware files available in the specified folder,
5. it searches if the malware are already present in the list containing the malware already analysed (this list is a file filled by the script python that registers the data in the database)
6. if the name of the file is not present in this list, it submits it to *Cuckoo*
7. it loops while there are still analyses to do and run 20 analyses at a time then clean the 20 analyses with a call to the script explained in Subsection 6.1.3,
8. it finishes by telling if the number of analyses performed is the same than the number of analyses requested.

The script, `submit_some_malware.sh`, can be found by following this link.

6.1.5 Corner cases

One of the problems I have encountered with this script is that the `mongod` service has to be restarted sometimes. So just before calling *Cuckoo* I check if the service is up and running.

Another problem, I have faced is that when I restarted my computer, the network interface `vboxnet0` was still up but the machines that were connected to it, did not seem to be connected any more. So if *Cuckoo* says that it is unable to bind the result server, I delete and recreate the interface.

To handle all these cases and even more the script is about 120 lines long, it is quite heavy but it works well and displays its progression along the way.

6.2 Results

During multiple nights, I have executed this script and collected the results. 8931 malware were analysed and I present some interesting aspects of these results in this

section. 8930 malware came from the company *Cisco* and were sent to *UCLouvain*, I had access to it as part of my thesis. I also added to the analyses the executable from the project Pafish [23].

The whole database is available online with instructions on my GitHub repository [39] to download and use it. I made a python "package" to compute all the results presented in this section. This "package" contains multiple functions with the goal to facilitate the gathering of information from the database. It is composed with these elements:

- `commands` This is a dictionary which contains all the commands to execute with the name of the command as key. A command is itself a dictionary which contains among others a function to execute and a type of result.
- `add_command` This function allows to add a command to `commands`.
- `execute_commands` This is a function which goes through all malware registered in the database and executes all the commands for each one of them.
- `get_result` This function gives the results corresponding to the name of the command received in argument.

Thanks to these simple functions, I was able to concentrate uniquely on the informations that I needed and not any more on the ways to get it. To use it, you just have to add this line on the top of your file:

```
from package_db import *
```

This could have been improved but it was enough for my personal usage. You can find these functions on the file `package_db.py` available by following this link.

The graphs and information of this section are available by executing the python script on the same repository and available here. This python script uses the functions described above.

6.2.1 Malware scores

Cuckoo 2 has a scoring system, even though it is a good indicator, it is not a good scoring system. The Cuckoo team members are aware of that and they warn the users that this score is purely informative. The scoring system is based on the count of recognized signatures in the behaviour of the malware, but this scoring system has been proved flawed [40].

Nonetheless, 214 malware received a score of zero. Within these 214 malware, 119 have encountered at least one error during the execution. We can easily find that these errors are correlated with the score that the malware has received.

In fact, all these errors except one (surely uncorrelated) happened in the Static module in the `_get_signature` function. The error is related to the issue 1421 [32] with the `m2crypto` python package, as said previously (see Subsection 5.1.2) if the package is not running the good version there are errors as it is the case here. I have corrected that in the latest version of the Dockerfile but some analyses were done before that.

As we can see on Figure 6.1, the majority of the malware did not get a high score. In fact, more than half of the malware received a score lower than 4. As I just said, the scoring system is not the best one and in the web interface you would see that the score of the malware is of 4 out of 10, but as you can see a bunch of malware has obtained a score of more than 10. Actually there are exactly 1123 malware that have a score of 10 or more which represents a little more than an eighth.

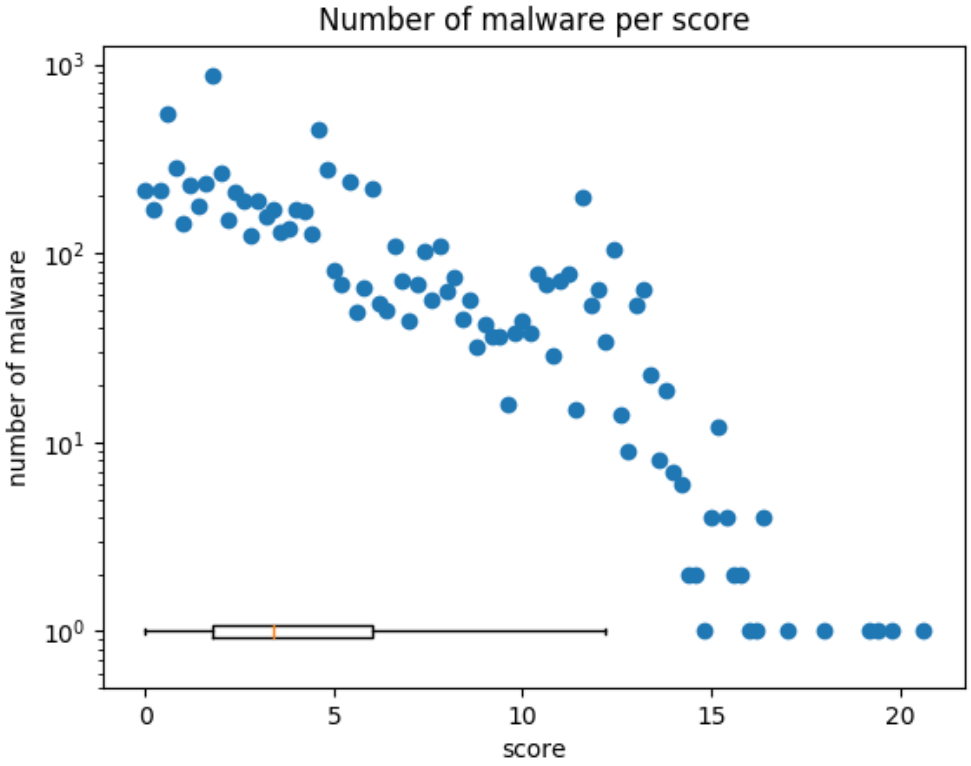

Figure 6.1: Number of malware per score given by *Cuckoo*

6.2.2 Signatures

In *Cuckoo*, each signature has a number which determines its severity. In function of its occurrence and its severity, the signature participates at a higher degree to the score. For example, the signature `antivm_queries_computername` has only a severity of 1 because it may be a query to check if the malware is currently in a VM but it may also be a totally legitimate request coming from a legitimate software. With these signatures, we can also extract some interesting figures.

First, I looked at all malware containing at least 1 signature that matched, and I have found 8727 malware. This corresponds again to what was previously obtained because we noticed that 214 malware received a 0 score which is exactly the missing number to the total.

Then, I noticed that 5836 malware had a signature with the severity of 5 but this severity is only used when the code is performing a very high suspicious activity. For these malware the score should be directly high, without even taking into account other parameters. However, this is not really the case, as we can see on Figure 6.2 in blue, lots of malware with low scores have a signature of severity 5. We can also notice on the same figure that without a signature of severity 5, it is harder to reach a big score.

There is only one signature which has a severity of 6. This level of severity is reached when there is at least one antivirus from *VirusTotal* which found the malware in its database. This is an automated verification that can be specified in the configuration file.

6.2.3 Evasions

Cuckoo has multiple signatures that indicate a tentative of evasion. Fortunately, these signatures are explicit and contain "`antivm`" or "`antisandbox`" in their names, that is why it is easy to verify if the malware has tempted an evasion.

To get the results for this section I made a python code that allows me to retrieve information of all the signatures containing a substring given as argument. This script is available by following this link. You can also go and see this file which was generated by the code and which contains all the signatures and their information.

In this section, I review some of the evasions we have talked about in Section 3.3 and see how malware use them.

As already said, the "malware" from the project Pafish [23] is included in the analyses. This "malware" matches most of the evasion techniques, so I only mention when it is the only one which has a match with a signature.

Time delay (3.3.1) This concerns the signature `antisandbox_sleep` (attempts to delay the analysis task). Of course, the signature concerning the sleep is

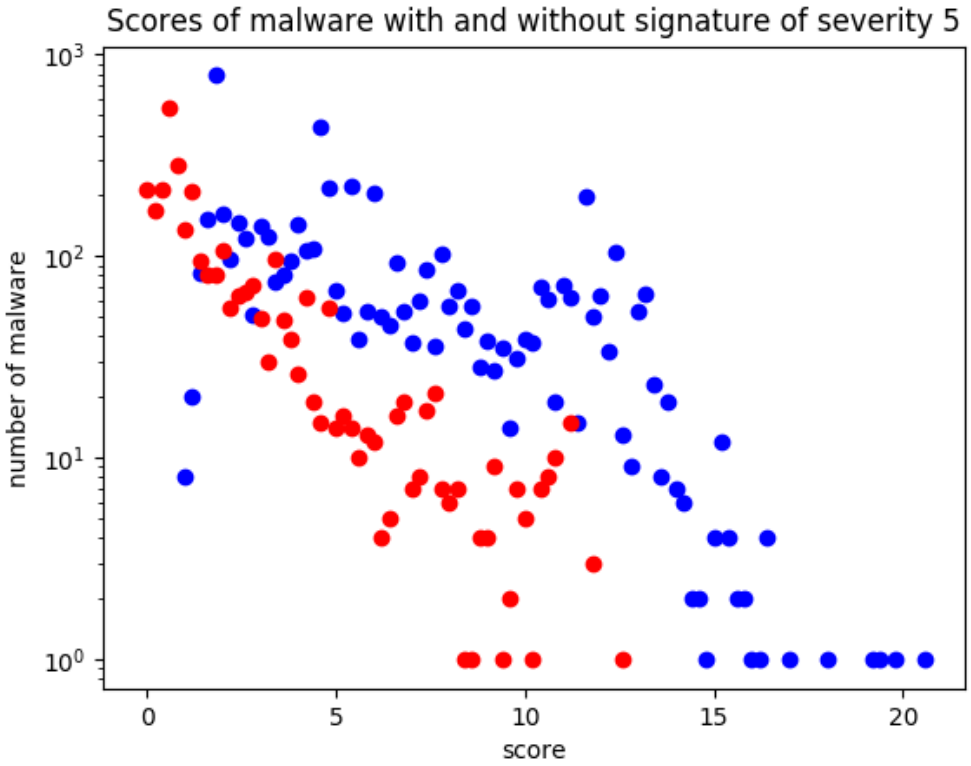

Figure 6.2: Score of the malware with (in blue) and without (in red) a signature of severity 5

very frequent so it has been found in 1202 malware but this may not be for an attempt to evade in all of these cases.

Filesystem (3.3.2) This evasion concerns the signatures `antisandbox_cuckoo_files` (tries to detect cuckoo with a file), `antisandbox_file` (searches paths where sandbox executes files), `antisandbox_joe_anubis_files` (tries to detect Joe or Anubis Sandboxes), `antiav_detectfile` (tries to detect AV directory) and some others. This is a technique that is quite easy and has multiple ways of being used, for every sandboxes and every virtualization softwares you can expect a different realisation. That is why it is quite used. 222 malware for Cuckoo, 104 for *VMWare*, 7 for Joe or Anubis and 7 for general sandbox were detected trying an evasion.

Registry (3.3.3) This concerns again multiple signatures like `av_detect_china_key` (searches for AV software regKey), `antivm_vbox_keys` (detects *VirtualBox* with regKey) or `antivm_vmware_keys` (detects *VMWare* with regKey).

These attacks are not common because these signatures have respectively 50, 24 and 1 occurrences in the malware analysed. The only occurrence of `antivm_vmware_keys` comes from the analysis of **Pafish** so it is not really a match from a malware.

Generic OS queries (3.3.4) This evasion concerns multiple possible checks of the computer configuration. One signature is `antivm_queries_computername` and was found in 2061 malware this is one of the easiest evasion, but has severity of 1 because there may be tons of other reasons to request the computer name. The signatures `antivm_memory_available` and `antivm_disk_size` are of the same kind, really frequent but may be legitimate for a software. Another one, `antisandbox_idletime` (tries to determine the uptime of *Windows*) was found 41 times but has low chances to be there accidentally and it has a severity of 3.

Processes (3.3.5) This is about searching for processes that only run in a sandbox or virtual environment. This one has the signature `process_interest` (interests in a specific process) but like some others, this is very generic and is not sufficient to say that the malware attempted an evasion. This signature was found 937 times.

Global OS objects (3.3.6) This evasion may be detected by *Cuckoo* with the signatures `antivm_shared_device` or `antivm_vbox_devices` (detects presence of VM (or *VirtualBox* specifically) with a device presence). These two signatures have only occurred once for each of them but they were both from the malware `pafish` (see section 3.1), so it should not be counted as present in any malware.

Debug privileges (3.3.7) This concerns only the signature `checks_debugger` (checks if currently being debugged) which occurred 2317 times. The severity of this signature is only 1 but it was present in many samples.

UI artefacts (3.3.8) This one is exactly like "Global OS Objects", the signature `antivm_vbox_window` (detects *VirtualBox* through presence of a window) was only detected with **pafish**. However, there is also another signature which tries to see if the foreground windows have changed, this may be a legitimate request so the severity is only of 2. This signature is `antisandbox_foregroundwindows` and has been seen 937 times.

Network (3.3.9) This kind of evasion includes every trick based on network. The signature concerned is `antivm_network_adapters` (checks adapter addresses to detect virtual network interface) and was found 1292 times. However, the

evasion is not the only possible reason to check the adapter addresses, so the severity is only of 2.

CPU (3.3.10) This evasion can be implemented in multiple ways, but only one signature is corresponding `antivm_generic_cpu` (checks the CPU name) which has a severity of 3 and 61 occurrences. This signature has not a lot of matches but it is very likely that these are not innocent.

Hardware (3.3.11) This concerns one signature, `antivm_generic_ide` (checks the presence of IDE drives) but it is again only observed in the malware **Pafish**.

Firmware tables (3.3.12) This evasion is very specific but has one signature. `antivm_firmware` (detects VM through firmware) was registered 6 times with a severity of 3.

Hooks (3.3.13) As in the theory, we can decompose this part in two. The first one is about detecting a user action thanks to a hook and the signature associated is `antisandbox_mouse_hook` (hooks the mouse events). The other one tries to detect if there are unusual hooks installed on the host, with the signature `antisandbox_unhook` (tries to unhook functions monitored). These signatures have been observed 164 times and 10 times respectively. As we can see monitoring the mouse events seems to be easier to implement and a known technique to try to evade an environment.

The evasions specific to *Cuckoo* (see Section 3.4) have no signature that I am aware of. However, they could be encountered by updating *Cuckoo* with a new version that would take these kinds of evasion into account.

6.2.4 VirusTotal

VirusTotal is a free organization that inspects items with 70 antivirus scanners. There is a configuration parameter in *Cuckoo* to allow it to send a request to *VirusTotal* in order to know what are the results for these 70 scans. However, for this request to happen without trouble, there is in fact a need for an internet connection. As already said, I ran the analyses during the nights on my personal computer and sometimes, unfortunately, the internet connection had some issues. So for the 8941 malware, I miss the results of *VirusTotal* for 80 of them.

For 2170 malware, I received the response "resource has not been scanned yet" from *VirusTotal*. This means that *Cuckoo* has sent the hashes of the malware but *VirusTotal* did not find the malware in the database. If we really want to, we can submit manually the malware to *VirusTotal* but it is not included in *Cuckoo*.

As we can see on Figure 6.3, the vast majority of the malware scanned have a high number of matches. According to the box plot of the same figure, all the malware with 29 matches or less are considered as outliers considering the very high number of malware between 40 and 60.

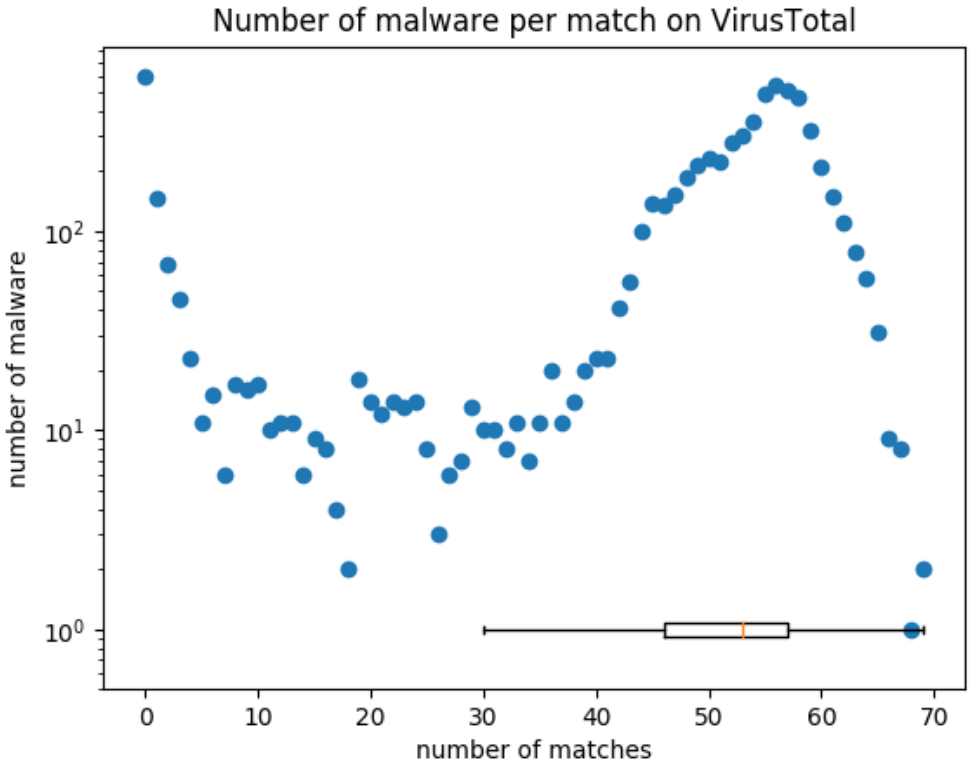

Figure 6.3: Number of malware per match on *VirusTotal*

Chapter 7

Cuckoo 3

Cuckoo 2 is currently at an end concerning its development. There are multiple reasons amongst which some dependencies of the core that are still with *Python 2* even though it is deprecated since 1st January, 2020. This end is easily noticeable when you look at the activity on the *Github* repository [32]. From July 2019 to July 2020 there were only 8 commits merged on the master branch and for the same period one year earlier there were more than 150 commits and it is already less than the previous years. By going into the slack server with the developers of *Cuckoo Sandbox*, I learned that *Cuckoo 3* was in development. This version is currently not available to the public because the developers have not even a stable core. The release of the third version is still not announced for the same reason. They are waiting to have a stable core and architecture in order to publish an official version. Since I wanted to help in the development, I contacted Ricardo van Zutphen on *Slack*, he is one of the developers of the Cuckoo team. There he explained me a bit what was *Cuckoo 3* about and what I could already do even though the core is not released yet.

7.1 Cuckoo 2 vs Cuckoo 3

Why even build a new version of *Cuckoo*? The developers have noticed that multiple functionalities of the core did not correspond any more to their needs.

We already cited the first reason before, it is the dependencies with *Python 2* which is now deprecated. So *Cuckoo 3* will get rid of these dependencies.

As new functionalities, Ricardo v.Z. told me that they wanted each module to be more independent. The goal of this independence is to be able to update each one of the modules without modifying the others. So they want an increase of modularity of the program in order for *Cuckoo 3* to be easier to evolve.

Another targeted functionality, is to be able to use multiple machineries at the

same time. For *Cuckoo 2*, if you want to use at the same time *KVM*, *VirtualBox* and a physical machine you are forced to use 3 instances of *Cuckoo 2*. The goal would be to use all of them in one instance of *Cuckoo 3*.

7.2 My contribution

As the core of *Cuckoo* is still not finished the possible task I could do to help *Cuckoo 3* was not easy to find, but after some chat with Ricardo v.Z. he proposed me to implement other machinery modules than the one that was already done. The only machinery module that was already implemented was for *KVM*, so with the consultation of Axel Legay, we decided for me to implement the module for the machinery *VirtualBox*.

7.2.1 VirtualBox and Cuckoo

VirtualBox is the default machinery used with *Cuckoo 2*. However, this will not be the case any more with *Cuckoo 3*. In fact, the developers want to have a different default machinery depending on your host OS.

VirtualBox in *Cuckoo 2* was not the easiest module to read. The reason is that they did not use any library to interact with *VirtualBox*, so every call to the machines was done via a call to `VBoxManage` with a subprocess and a try except statement to intercept the errors that may arise due to the subprocess call. This is not convenient but it worked well for *Cuckoo 2*.

To develop this machinery module, I learned that *VirtualBox* has an official "Software Developer Kit" (SDK) and that it is possible to interact with it in *Python 3*. To install this SDK you have to go to the official website of *VirtualBox* [41] and to download the SDK corresponding to the version currently installed on your host machine.

7.2.2 VirtualBox SDK

VirtualBox is composed of multiple layers to manage VMs, as shown the Figure 7.1. On the first layer there is the "Hypervisor", the virtualization engine that runs on the kernel of the host machine. Above the hypervisor lies multiple modules like the "Resource monitor". Above these modules, there is the main "VirtualBox API", also known as "Main API". This exposes the entire feature set of the "Virtualization engine". Above this API, we found the "VirtualBox GUI" or the tool "VBoxManage", they both use the Main API to interact with the hypervisor.

If you want to interact with *VirtualBox* with a programming language, there are basically two options. Either you use the web service which interacts for you with

the Main API, or you interact directly with the Main API using the Component Object Model (COM or Cross Platform COM also named XPCOM).

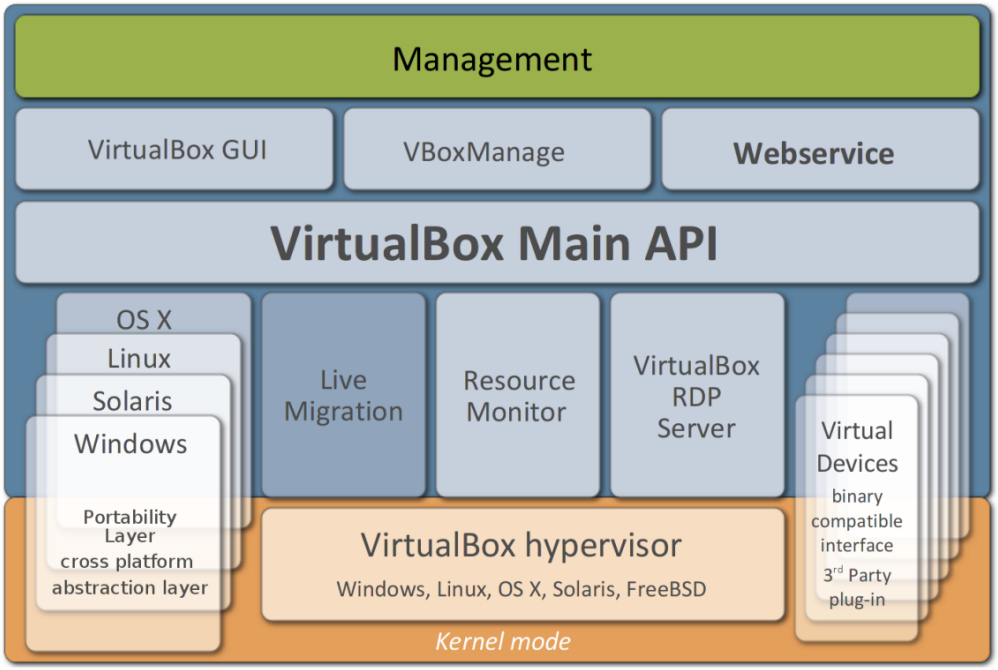

Figure 7.1: The building blocks of *VirtualBox* from the Programming Guide [42]

7.2.3 Python virtualbox

Python proposes a library to have a "pythonic" interface to the VirtualBox's COM API. So this is what I used to write the code of the machinery module for *VirtualBox*. The library, named `virtualbox`, is at its 2.0.0 version [43]. You can install it with `pip`, if the SDK of *VirtualBox* is already installed on your computer. However, this is not the only thing you need to do to use it. You also have to export a modified version of your python path because the library uses some files installed by the SDK in a folder not accessible by default by *Python*.

7.2.4 The code

One of the requirements for this code is to handle all possible errors or divergent behaviours that may happen when executing and if needed throw a "Cuckoo" specific error. The point of this manoeuvre is to never let an abnormal behaviour continue to execute. A good example is the possible machine states, if the machine

state does not correspond to one of the states *Cuckoo* allows, a specific error must be raised.

As previously mentioned (see Section 7.1), one of the functionalities desired for *Cuckoo 3* is to use multiple machineries with one instance. That is why they added a "Machinery Manager" which handles the launching and stopping of machines via their own "Machinery Module". To allow each modules to be called by the machinery manager, there is an abstract class `Machinery` that each module has to inherit. This abstract class has 8 functions that are to be implemented amongst which 6 throw a `NotImplementedError`. I was asked to implement 6 of them because two are not critical at the moment and they could do it by themselves later.

The first function is `restore_start` and must restore the machine to the snapshot specified in the configuration file or the last snapshot if nothing is specified and then start the VM. The function `norestore_start` has to start the machine but is not needed for now even if its implementation can be made starting from the previous function quite easily. The function `stop` should stop the VM gracefully. The function `acpi_stop` has to force the machine to stop in case the graceful stop has not been effective after a timeout. The `state` function should return the state of the VM from one of the possible states of a "Cuckoo Machine" or it should raise an error. A sixth function was there to dump the memory of the VM but Ricardo told me it is not needed for the time being. The goal of the two other functions are not VM management, one is to get the version of the virtualization software as a string and the other to verify if the dependencies of the module are met on the host machine.

All the functions are implemented using entirely the virtualbox library (See subsection 7.2.3) except for the `restore_start` function. When the function `restore_snapshot` is called on a VM, there is an error that is raised from the XPCOM library because this function is apparently not implemented.

```
>>> vm.restore_snapshot(vm.current_snapshot)
Traceback (most recent call last):
  (...)
  File "<XPCOMObject method 'restoreSnapshot'>", line 3, in restoreSnapshot
xpcom.Exception: 0x80004001 (Method restoreSnapshot is not implemented)

During handling of the above exception, another exception occurred:
Traceback (most recent call last):
  (...)
virtualbox.library.OleErrorNotimpl: 0x80004001 (Method restoreSnapshot is not
↪  implemented)
```

In a way the python library was not lying when they said that this is a complete implementation of the VirtualBox's COM API, because it makes a call to the good function, but this function is not even implemented in the XPCOM API. As restoring

a snapshot is quite an essential need for *Cuckoo*, I had to implement it by another way. So I used the same way than *Cuckoo 2* which means using `subprocess` in order to call the tool `VBoxManage`. This solution is not ideal, but the code to use the API is commented in the class with an explanation, and with the hope that the XPCOM API will be completed in the future.

In addition to the `VBox` class that I have implemented, I had to modify the `config.py` file. This file contains the code to fill the `virtualbox.yaml.jinja2` file and to generate the configuration file that is used to specify the *VirtualBox* parameters, like which machines must be used by *Cuckoo*.

7.3 Outcome

Ricardo was happy with the result of my contribution, he said it will be integrated in *Cuckoo 3*. He has already merged my contribution with the version of *Cuckoo 3* that was done by the time I had finished this contribution.

The complete code and my contribution can be found on this private repository [44] because the developers do not want to publish the complete structure of *Cuckoo 3* while the core is not even finished.

Chapter 8

Future works

There are multiple things that could be improved in the aspects I have approached. In this section, I review what work can be built in the continuation of mine for each previous chapter of the implementation part.

Chapter 5

In this chapter, a lot of things may be improved. I thought about making a tutorial to help the students step by step to deploy *Cuckoo* with the scripts.

And a more technical part would be to ensure that everything is present for any use of *Cuckoo* in the container. For example, `elasticsearch` is not installed in the container and we cannot use the research functionality that is present on the Cuckoo Web Interface. *Cuckoo* also proposed to use `uwsgi` and `nginx` to have a stronger web deployment but they are not installed currently in the image.

Another great improvement, would be to start from this Dockerfile to create one for *Cuckoo 3* when it will be released.

An important step for this part would be to understand why it was not possible to deploy this configuration in a virtual machine *Qemu*. I have spent hours with Fabien Duchêne trying to spot the problem but we did not succeed.

Chapter 6

For the chapter about the analyses, it would be great to extend the analyses of the malware I have already done to a much bigger size. In fact, the more data you can have, the better. Maybe also to submit again some of the malware I have already done, as I have explained in the chapter, some of them had errors in their execution caused by a bug in *Cuckoo* that I have fixed later on.

Chapter 7

This chapter is about *Cuckoo 3*, which will need further improvements and collaborations in the months to come, and we can hope that it will be soon released.

As already told in the chapter, Ricardo did not found necessary to do all the functions but the functions `norestore_start` and `dump_memory` will still have to be done.

The `restore_snapshot` is still not implemented. It would certainly be interesting to implement it to improve the XPCOM library.

Chapter 9

Conclusion

I have now been working for a year on this thesis and it is time to review what was done.

In Chapter 2, I explained the basis in cybersecurity and I provided some background to understand the malware. I have also explained what is a "Malware Sandbox" and why it is important to have this technology.

Then in Chapter 3, I detailed the flaws of the sandboxes, and multiple ways to evade them. This proves that the technique is not a perfect solution but a perfectible one.

In Chapter 4, I dive into details what is *Cuckoo Sandbox*. And I showed that the installation was a bit difficult. Even when the installation is done, the sandbox does not run directly but it also needs some configuration.

These difficulties lead to the reason why it is worth simplifying the deployment of *Cuckoo*. I explain what I did and all the ambushes I have met during this process in the Chapter 5.

Once the sandbox is up and running it is time to see what it is capable of. That is why I have launched a lot of analyses. I explained all the work I have done to analyse more than 8 thousands malware and compiled the results of these analyses in the Chapter 6. This chapter allows us to confirm that the sandboxes are a sustainable solution and that the attempts to evade them may be noticed and can even backfire the malware. If the tested "software" tried to evade it is even more suspicious for the malware researcher.

Finally, in Chapter 7, I helped developing the future *Cuckoo Sandbox* that we hope will be better on multiple aspects than the previous version.

I have learned a lot about malware and sandboxes during this thesis. I hope I transmitted it well in this report, and as said in Chapter 8, there is still plenty of work to do. I wish an amazing journey to anyone who would pick up where I stopped.

Bibliography

[1] C. Point, "Cyber attack trends: 2020 mid-year report," https://pages.checkpoint.com/cyber-attack-2020-trends.html, accessed: 2020-08-14.

[2] Wikipedia, "Sandbox (computer security)," https://en.wikipedia.org/wiki/Sandbox_(computer_security), accessed: 2020-08-14.

[3] Cisco, "Types of malware," https://www.cisco.com/c/en/us/products/security/advanced-malware-protection/what-is-malware.html#~types-of-malware, accessed: 2020-08-14.

[4] N. DuPaul, "Common malware types: Cybersecurity 101," https://www.veracode.com/blog/2012/10/common-malware-types-cybersecurity-101, accessed: 2020-08-14.

[5] F. Biondi, T. Given-Wilson, A. Legay, C. Puodzius, and J. Quilbeuf, "Tutorial: An overview of malware detection and evasion techniques," in *International Symposium on Leveraging Applications of Formal Methods*. Springer, 2018, pp. 565–586.

[6] Fortinet, "Next-generation firewall (ngfw)," https://www.fortinet.com/products/next-generation-firewall, accessed: 2020-08-14.

[7] N. Idika and A. P. Mathur, "A survey of malware detection techniques," *Purdue University*, vol. 48, pp. 2007–2, 2007.

[8] Y. Ye, T. Li, D. Adjeroh, and S. S. Iyengar, "A survey on malware detection using data mining techniques," *ACM Computing Surveys (CSUR)*, vol. 50, no. 3, pp. 1–40, 2017.

[9] P. Vinod, R. Jaipur, V. Laxmi, and M. Gaur, "Survey on malware detection methods," in *Proceedings of the 3rd Hackers' Workshop on computer and internet security (IITKHACK'09)*, 2009, pp. 74–79.

[10] E. Gandotra, D. Bansal, and S. Sofat, "Malware analysis and classification: A survey," *Journal of Information Security*, vol. 2014, 2014.

[11] D. Uppal, V. Mehra, and V. Verma, "Basic survey on malware analysis, tools and techniques," *International Journal on Computational Sciences & Applications (IJCSA)*, vol. 4, no. 1, p. 103, 2014.

[12] R. Sihwail, K. Omar, and K. Z. Ariffin, "A survey on malware analysis techniques: Static, dynamic, hybrid and memory analysis," *International Journal on Advanced Science, Engineering and Information Technology*, vol. 8, no. 4-2, p. 1662, 2018.

[13] A. Moser, C. Kruegel, and E. Kirda, "Limits of static analysis for malware detection," in *Twenty-Third Annual Computer Security Applications Conference (ACSAC 2007)*. IEEE, 2007, pp. 421–430.

[14] A. Afianian, S. Niksefat, B. Sadeghiyan, and D. Baptiste, "Malware dynamic analysis evasion techniques: A survey," *arXiv preprint arXiv:1811.01190*, 2018.

[15] M. Lindorfer, C. Kolbitsch, and P. M. Comparetti, "Detecting environment-sensitive malware," in *International Workshop on Recent Advances in Intrusion Detection*. Springer, 2011, pp. 338–357.

[16] J. Li, D. Gu, and Y. Luo, "Android malware forensics: Reconstruction of malicious events," in *2012 32nd International Conference on Distributed Computing Systems Workshops*. IEEE, 2012, pp. 552–558.

[17] Wikipedia, "Polymorphic code," https://en.wikipedia.org/wiki/Polymorphic_code, accessed: 2020-08-14.

[18] S. Batt, "Why you shouldn't install multiple antivirus programs on one pc," https://www.maketecheasier.com/multiple-antivirus-programs-on-one-pc/#:~:text=Theworsteffectofmultiple,assumeit'sactuallyavirus., accessed: 2020-08-14.

[19] I. You and K. Yim, "Malware obfuscation techniques: A brief survey," in *2010 International conference on broadband, wireless computing, communication and applications*. IEEE, 2010, pp. 297–300.

[20] C. S. Veerappan, P. L. K. Keong, Z. Tang, and F. Tan, "Taxonomy on malware evasion countermeasures techniques," in *2018 IEEE 4th World Forum on Internet of Things (WF-IoT)*. IEEE, 2018, pp. 558–563.

[21] C. P. Research. Evasion techniques. https://evasions.checkpoint.com. Accessed: 2020-03-25.

[22] CheckPointSW, "Evasions," https://github.com/CheckPointSW/Evasions, 2020, repository GitHub.

[23] a0rtega, "pafish," https://github.com/a0rtega/pafish, 2020, repository GitHub.

[24] CheckPointSW, "Invizzzible," https://github.com/CheckPointSW/InviZzzible, 2020, repository GitHub.

[25] LordNoteworthy, "al-khaser," https://github.com/LordNoteworthy/al-khaser, 2020, repository GitHub.

[26] C. F. on Sentinel One blog, "Anti vm tricks," https://www.sentinelone.com/blog/anti-vm-tricks/, accessed: 2020-08-07.

[27] W. Mercer and P. Rascagneres. Gravityrat - the two-year evolution of an apt targeting india. https://blog.talosintelligence.com/2018/04/gravityrat-two-year-evolution-of-apt.html. Accessed: 2020-04-30.

[28] hfiref0x, "Vmde," https://github.com/hfiref0x/VMDE, 2020, repository GitHub.

[29] A. Singh and Z. Bu, "Hot knives through butter: Evading file-based sandboxes," *Threat Research Blog*, 2013.

[30] dtm. Defeating userland hooks (ft. bitdefender). https://0x00sec.org/t/defeating-userland-hooks-ft-bitdefender/12496. Accessed: 2020-04-30.

[31] A. Chailytko and S. Skuratovich, "Defeating sandbox evasion: how to increase the successful emulation rate in your virtual environment," in *ShmooCon 2017*, 2017.

[32] cuckoosandbox, "Cuckoo," https://github.com/cuckoosandbox/cuckoo, 2020, repository GitHub.

[33] C. Foundation. Installation. https://cuckoo.sh/docs/installation/index.html. Accessed: 2020-05-06.

[34] L. Rendek, "Ubuntu 20.04 python version switch manager," https://linuxconfig.org/ubuntu-20-04-python-version-switch-manager, accessed: 2020-06-08.

[35] volatilityfoundation, "volatility," https://github.com/volatilityfoundation/volatility, 2020, repository GitHub.

[36] cuckoosandbox, "monitor," https://github.com/cuckoosandbox/monitor, 2020, repository GitHub.

[37] C. Foundation. Cuckoo: Preparing the guest. https://cuckoo.sh/docs/installation/guest/index.html. Accessed: 2020-04-30.

[38] hatching, "Vmcloak," https://github.com/hatching/vmcloak, 2019, repository GitHub.

[39] TGLuis, "Lingi2990-malwaresandbox," https://github.com/TGLuis/LINGI2990-MalwareSandbox, 2020, repository GitHub.

[40] A. Walker, M. F. Amjad, and S. Sengupta, "Cuckoo's malware threat scoring and classification: Friend or foe?" in *2019 IEEE 9th Annual Computing and Communication Workshop and Conference (CCWC)*. IEEE, 2019, pp. 0678–0684.

[41] Oracle, "Virtualbox," https://www.virtualbox.org, accessed: 2020-07-13.

[42] V. Oracle, "Virtualbox programming guide and reference," 2012.

[43] pypi, "virtualbox," https://pypi.org/project/virtualbox/, accessed: 2020-08-12.

[44] TGLuis, "cuckoo3vbox," https://github.com/TGLuis/cuckoo3VBox/, 2020, repository GitHub.

UNIVERSITÉ CATHOLIQUE DE LOUVAIN
École polytechnique de Louvain
Rue Archimède, 1 bte L6.11.01, 1348 Louvain-la-Neuve, Belgique | **www.uclouvain.be/epl**