UCLouvain

École polytechnique de Louvain

Building a mutation tool for malware

Author: **Dimitri Wauters**
Supervisor: **Axel Legay**
Readers: **Charles-Henry Bertrand Van Ouytsel, François Michel**
Academic year 2022–2023
Master [120] in Computer Science

Acknowledgment

Foremost, I would like to express my gratitude to my supervisor, Pr Axel Legay, for giving me the opportunity to work on this subject and for the time he has devoted to answering my questions.

I would also like to thank the team of Pr Axel Legay who followed and helped us during the time of this Master Thesis. For that, I will thank Christophe Crochet, Serena Lucca and particularly Charles-Henry Bertrand Van Ouytsel, that took the time to explain the side of this project that I didn't understand directly and for the time he took to answer my numerous questions.

I would like to thank my readers, François Michel and Charles-Henry Bertrand Van Ouytsel for taking the time to read and to evaluate this Master Thesis.

Finally, I would like to thank Sophie for her continuous support during the writing of this Master Thesis.

Abstract

The number of malware spreading on Internet grows each year, but the majority of these malware are variation of existing ones. These variations help the malware to escape the detection of antivirus solutions and be able to continue to spread while being undetected thanks to a minimum of modification.

This Master Thesis aims to develop two tools: a first tool that will be a mutation tool for malware by encapsulating them into another software that will protect them against these antivirus and a second tool for detecting these kinds of "malware enveloping" software (called packers). The proposed solutions are intended to be added to the existing work of Pr Axel Legay team: SEMA, an open-source symbolic execution toolchain for malware analysis.

To achieve that, we will build our solutions on top of well-established open source python software: Inceptor and PANDA.

Contents

1 Introduction **7**
- 1.1 Objectives . . . 10
- 1.2 Structure . . . 11

2 Problem Statement **12**

3 Background Information **14**
- 3.1 Malware . . . 14
  - 3.1.1 Malware Classification . . . 14
  - 3.1.2 Malware Detection . . . 16
    - 3.1.2.1 Static Analysis . . . 16
    - 3.1.2.2 Dynamic Analysis . . . 16
    - 3.1.2.3 Symbolic Execution . . . 17
  - 3.1.3 Malware Mutation . . . 17
- 3.2 SEMA Toolchain . . . 17
  - 3.2.1 Architecture . . . 17
    - 3.2.1.1 SEMA-SCDGs . . . 18
    - 3.2.1.2 SEMA-Classifier . . . 18
    - 3.2.1.3 SEMA-Federated Learning . . . 18
  - 3.2.2 Purpose . . . 18
- 3.3 Packer . . . 19
- 3.4 Sandbox . . . 19
- 3.5 Debugger . . . 20
- 3.6 Sandbox/Debug Evasion . . . 20
- 3.7 Antivirus (AV) . . . 20
- 3.8 Endpoint Detection and Response (EDR) . . . 21

4 Design and Implementation **22**
- 4.1 Packer (Protector) with Inceptor . . . 22
  - 4.1.1 Introduction to Inceptor . . . 22
  - 4.1.2 Installation . . . 23

4.1.2.1 Python Virtual Environment . . . 25
4.1.2.2 Visual Studio and Installation Script . . . 25
4.1.3 Command Line Usage . . . 25
4.1.4 Inceptor Modifications . . . 27
4.1.5 Evasion Techniques Source . . . 27
4.1.5.1 LordNoteworthy/al-khaser . . . 27
4.1.5.2 a0rtega/pafish . . . 28
4.1.5.3 CheckPointSW for Sandbox/Anti-Debug . . . 28
4.1.5.4 CheckPointSW/InviZzzible . . . 28
4.1.6 Debug Evasion Techniques . . . 28
4.1.6.1 Exceptions . . . 28
4.1.6.2 Flags . . . 29
4.1.6.3 Memory . . . 29
4.1.6.4 Object . . . 29
4.1.6.5 Timing . . . 30
4.1.7 Sandbox Evasion Techniques . . . 30
4.1.7.1 CPU . . . 30
4.1.7.2 OS Features . . . 30
4.1.7.3 Filesystem . . . 31
4.1.7.4 Generic OS Queries . . . 32
4.1.7.5 Human-like Behavior . . . 32
4.1.7.6 Network . . . 32
4.1.7.7 Global OS Objects . . . 33
4.1.7.8 Processes . . . 33
4.1.7.9 Registers . . . 33
4.1.7.10 Timing . . . 33
4.1.7.11 User Interface Artifacts . . . 34
4.1.8 Evaluation . . . 34
4.1.9 Role in the Toolchain . . . 35
4.1.10 Future Improvements . . . 35
4.2 Detect Packers with PANDA . . . 36
4.2.1 Introduction to PANDA . . . 36
4.2.2 Installation . . . 36
4.2.3 Use PANDA to Detect Packers . . . 37
4.2.4 Script Usage . . . 38
4.2.5 Create a Windows Image . . . 38
4.2.6 Other PANDA Capabilities . . . 39
4.2.6.1 Record / Replay . . . 39
4.2.6.2 Callbacks . . . 39
4.2.6.3 Plugins . . . 40

4.2.7 Role in the Toolchain 40
4.2.8 Future Improvements 40

5 Related Work **41**
5.1 PEZor: Open-Source Shellcode & PE Packer 41
5.1.1 Explanation 41
5.1.2 Differences with Inceptor 41
5.2 Binary Rewriting by Modifying the Intermediate Representation (IR) 42
5.3 Self-Mutating Malware Detection with Code Normalizer 42
5.3.1 Polymorphic 43
5.3.2 Metamorphic 43

6 Conclusion **44**

Appendices **48**

A PANDA: Docker-compose File **48**

B Inceptor VirusTotal Evaluation **49**

Chapter 1

Introduction

Malware, short for malicious software, is any software designed by attackers to cause harm to a computer, a server or a computer network. This is an umbrella word used for referring to a variety of forms of hostile or intrusive software, such as viruses, ransomware, trojan horses, worms or keyloggers. Most computer historians say that the beginnings of malware are around 1970s, with the proof-of-concept The Creeper worm (1971). Its intent was to be spread through a computer network to infect connected computers, by replicating itself across the early version of the modern Internet, ARPANET[1]. Once activated, The Creeper displayed the message `"I'm the creeper, catch me if you can!"` on the infected computer, without other malicious action: the intent was to see whether the message could propagate to other computers via the network [24]. The first antivirus Reaper was then created in 1972 to prevent the propagation of such software across ARPANET.

As computers became more common, so became malicious programs: in 1982 appeared the first virus targeting Apple II computers, called Elk Cloner. 4 years later, the first PC-based virus, known as Brain, was developed by 2 Pakistani brothers (Farooq Alvi brothers) either to prove that IBM PC was not secure, according to some authors [18], or to track illegal copies of their medical software, according to others [24], [28]. Brain affects the PC by replacing the boot sector of a floppy disk with a copy of the virus[2], displaying a notification with Farooq Alvi brothers' contact information for requesting assistance.

[1] ARPANET was a US military computer network, standing for "Advanced Research Projects Agency Network". This was the first packed-switched network and one of the first to implement TCP/IP protocols.

[2] As Internet did not exist, the virus was propagated through the copy of the infected floppy disks.

The first malware that was (accidentally) distributed over the Internet was The Morris worm, called after its author, Robert Morris. As for The Creeper, The Morris worm did not require any human interaction to be propagated. It was aimed to count the number of computers connected to the Internet, by replicating itself from a connected computer to another one (and an incremental counter). However, there was a bug in the program and the worm revisited computers that it had previously visited, generating a significant amount of network traffic that make devices inoperable, causing then a denial-of-service (DoS).

In 1991 the malware evolution moved forward with the introduction of the Dark Avenger Mutation Engine (DAME). This tool adds mutation functionality (or polymorphism) to viruses, making them more difficult to be detected by antivirus software, that were relying so far on file signatures and changes in file signatures. In 1992 appeared Virus Creation Laboratory, one of the earliest attempts to provide a virus creation tool with a user interface (UI), so that individuals with little or no programming skills can also create computer viruses.

Other viruses followed as well in the 1990s, such as Michelangelo (1992) and The Walker (1992). The first one was programmed to rewrite the first 100 sectors of a hard disk on March 6, Michelangelo's birthday, leading to the destruction of the file allocation table (FAT). There was a lot of media coverage recommending users to turn off their computers on that day. Before Michelangelo, no other malware (virus or worm) gained so much attention from the mainstream media. The second one produced an animated walker, going from one side of the screen to the other one. In that sense, The Walker was the first visual virus, followed by variants (e.g. The Ambulance).

Once it has been released, Windows has not been spared either by malware: Winvir was the first (non-harmful) virus to infect any .exe files in the current working directory of the executed Winvir-infected program, before rolling back to the original state of the file Winvir was executed from, once other files have been infected.

The first macro viruses appeared in the 1990s as well: Concept for Microsoft Word (1995) and Laroux for Microsoft Excel (1996). These viruses were written respectively in Microsoft Word macro language and in Visual Basic for Application (VBA) macro language. Their code were automatically executed when the document or the spreadsheet in which they were embedded has been opened by the user, as Microsoft did not disable macros by default in Office (for Office versions prior to 2000).

Happy99 (1999) and ILOVEYOU (2000) are the first examples of mail viruses in the malware history. Happy99 was spread as an executable attachment file[3] that was transferred to the user's contacts once the executable was run. ILOVEYOU was an e-mail message relying on social engineering to encourage the user to click on the attachment "LOVE-LETTER-FOR-YOU.TXT.vbs" for reading a love letter presumably addressed to him/her. Opening the attachment activates the Visual Basic script, that rewrites some important files on the victim's system.

In 2003 appeared the first malware designed to generate revenue: Fizzer, a mail worm that turned the infected machine into a spam sender. Before Fizzer, malware were written by enthusiasts without malicious intent for demonstrating a technique. From Fizzer, the main focus for malware programmers was profit.

A huge step in malware evolution has occurred around 2010, to include nation-state attacks. From this moment, malware is considered just like any other weapon and is created by intelligence agencies, military or police forces of several countries. Stuxnet (2010), Petya/NotPetya (2016) or Wannacry (2017) are the most known examples of malware that are believed to be developed by the intelligence services of one or several countries for sabotage and espionage. The first one physically sabotaged Iranian turbines for uranium enrichment by changing their rotation frequencies, while the other ones are ransomware leveraging a zero-day exploit kept by the NSA targeting Microsoft's SMB protocol (EternalBlue).

As the number of Internet-of-Things (IoT) devices increases, so increase malware. According to some authors [10], the number of attacks on IoT devices are growing over 700% since 2019. They even state that only 29% of IoT devices uses some sort of encryption for communicating over a network [10]. Mirai (2016) was the first DDoS botnet to target unmaintained IoT devices and/or IoT devices with default configuration[4] to redirect their internet traffic to specific websites, making them unavailable. Nowadays, the evolution of malware is moving towards a lucrative business model known as Ransomware as a Service, where malware creators develop ransomware that can be used by affiliates (in exchange for a percentage of the actual ransom) to find and infect victims.

To conclude, as shown here, in the early years of malware evolution virus authors did not have malicious intent. They wanted to check if the technique implemented in their programs would work, while the main goal of writing such programs after 2000 was generating profit and performing nation-state attacks (sabotage, espionage),

[3]At that time, spam filters barely existed and were not widely used.

[4]Although Mirai primarily targeted network routers, it also included other IoT devices.

with the diversification of malware authors profiles (researchers, students, criminals, military, etc.). This evolution of threats is also reflected in the evolution of the initial term "virus" to today's all-encompassing "malware" term. Similarly, as malware evolved, anti-malware software (or antivirus) evolved as well, leading to more complex malware trying to escape antivirus defenses based on signature-based detection, integrity checks and machine learning, as well as execution of the malware in an isolated environment (sandbox) to study how it behaves. This is what this master thesis is about: understanding how malware can mutate to bypass detection methods used by antivirus.

1.1 Objectives

In this Master Thesis, we will focus on how malware can mutate to evade anti-viruses defense. There are many ways available to malware creator to perform this evasion, but we will focus here on one aspect: how packers (we should speak about protectors to be exacts) can be used to evade anti-viruses. This means that we will not perform any binary rewriting (more to see in the related work, section 5.2) but focus on the means that can be put in place around the malware to allow it to escape the vigilance of anti-viruses/sandboxes.

The first tool that we will present is a packer protection-oriented, that means that it will protect its payload (in that case, the malware) and deploys it only when it seems safe for the malware.

The second tool is responsible to analyze the memory writes that a software makes to deduce whether it is packed or not.

The final purpose of this thesis is to include the mutation tool that we will describe in this document into a toolchain developed by Axel Legay and his team. We will describe later what is the purpose of this toolchain in the section 3.2. The tools presented in this Master Thesis will be added to this toolchain to expand its capabilities, the exact purpose of each tool are presented in their corresponding section: 4.1.9 for the packer creator and 4.2.7 for the packer detection.

1.2 Structure

This thesis is structured as follows: Chapter 2 exposes the current issue of malware mutation and how a mutation tool can improve the detection of a malware or one of its variants by an antivirus. Chapter 3 provides background information about the essential concepts that will be discussed throughout this work. Chapter 4 describes more precisely our solution and its implementation. Chapter 5 will discuss some related works, before concluding this master thesis with Chapter 6.

Chapter 2

Problem Statement

As the number of malware detected increase every year, one can think that each of these malware is a new one. However, most of these detected malware are in fact variants of existing ones. Malware authors generally take an existing malware already detected by an antivirus and modify it just enough to not be detected anymore. Antivirus have multiple ways of detecting a malware, the more common are signature-based or heuristic-based methods. The first class of methods is based on a specificity that will define the malware, generally the hash of the file. The second is based on the attributes that might indicate a sign of malware. The malware authors will always try to evade these detection mechanisms by creating new ways of obfuscation and new evasion techniques. This is a big issue for antivirus vendors as it is mandatory for them to always researching these evasion techniques to be able to counter them.

A malware might also detect the sandbox/antivirus and might decide to hide its malicious code by not executing it, so that it will not be classified as a malware from the observer point of view. This is exactly what we aim to develop in this master thesis with our two solutions. This is this second point that we will worry about during this Master Thesis, the purpose of our evasions will be to prevent the debugger to attach to our malware or, if we cannot, do not give hint that might lead the analyst to think that the software is a malware. The same goes for the sandbox, if we detect it, we will try to not execute any malicious code that might give a hint.

Axel Legay's team has developed in collaboration with Avast a toolchain for malware analysis, based on state-of-the-art symbolic execution techniques. The solutions proposed in this Master Thesis will be added to the toolchain at different locations, depending on its purpose.

The addition of a malware mutation generator into the toolchain will help it to expand its initial database of malware and allow it to train on a bigger dataset. The ultimate purpose is to recognize the pattern of these mutations before they are made on actual malware.

Chapter 3

Background Information

In this section, we will describe the terms that we will use during this Master Thesis. We will start by explaining the basics: what is a malware. We will discuss how to classify them (3.1.1) and to detect them (3.1.2). We will then explain the toolchain that we speak about earlier (3.2), what is a packer (3.3), a sandbox (3.4) or a debugger (3.5). We finish by explaining what is an evasion technique (3.6) and how antivirus (3.7) can detect a malware.

3.1 Malware

Malware, short for malicious software, is any software designed by attackers to cause harm to a computer, a server or a computer network, although initially malware authors did not have malicious intent. There exist multiple categories of malware, as shown below. In the scope of this thesis, we will not focus on a specific category of malware, and we will not differentiate them based on their purpose. Indeed, we will mostly discuss packers (see section 3.3) and use any malware as its payload.

3.1.1 Malware Classification

In this section, we will define briefly the most notable types of malware and their particularities.

- **Virus**: A virus is a malicious piece of software that does not even present itself as legitimate. It usually requires human interaction to propagate.

- **Worm**: A worm is a self-replicating piece of code that uses the network to propagate itself to multiple hosts, via e-mails, network shares or software bugs/backdoors. A worm usually does not require human interaction to propagate.
- **Backdoor**: A backdoor refers to any method by which normal security measures of a system can be bypassed.
- **Ransomware**: Malware that encrypts victims' hard disks and demands a ransom for the decryption key (without guarantee that the victim will have access to his/her original data after paying the ransom).
- **Spyware**: A spyware is a software installed on a computer device gathering (silently) information about the user activity without his/her knowledge. Spyware can steal sensitive information, such as bank details or passwords, and forward them to a third party.
- **Keylogger**: A keylogger is a type of spyware that collects keystrokes typed by the user (including passwords or credit card numbers).
- **Adware**: An adware is any application that displays unwanted advertisements on the computer (either as pop-up on the Internet browser or as banner in a software application) without the user consent.
- **Trojan**: A trojan is a type of malicious software either embedded in a legitimate software or that presents itself as a legitimate software with some useful or benign purpose. However, it masks some hidden malicious functionality.
- **Rootkit**: Once installed on the victim's system, a rootkit modifies the existing operating system so that an attacker can keep access to a machine and hide on it, with administrator privilege. This can be used to inject other malware into the target system.
- **Bot**: A bot is a piece of malware that will infect a user system to be able to perform commands remotely on the system. When an attacker has a collection of bot malware installed on multiple systems, we call it a botnet.
- **Wiper**: The only purpose of this type of malware is to wipe all the user data without any chance of recovery. This can be used to delete all traces of an intrusion on a system, for example.

A malware can also be a combination of multiple, two or more from above. For example, `Stuxnet` is a malware that can fall into the worm, virus and rootkit categories.

3.1.2 Malware Detection

Malware detection is a key component that we must consider if we want to improve general computer security. To be able to counter malware, we must know what they are doing. This is what malware detection is about: by analyzing how a malware behaves, patterns can be identified and used for preventing either the infection of a computer system by a malware or the propagation of this latter one.

There are multiple ways of analyzing a software and classifying it as a legitimate/malicious software. We will explain here the 3 main methods of analyzing a software behavior: static analysis, dynamic analysis and symbolic execution.

3.1.2.1 Static Analysis

The purpose of the static analysis is to analyze the file without ever executing it. This method of analysis is relatively safe as the potential malware cannot infect the system. The (reverse) engineer will look into the file source-code (with disassemblers or other means[1]), the header of the file, the strings stored in the files, the system calls and much more.

There are multiple tools available to the reverse engineer during the static analysis. For instance, to analyze the strings contained in an executable, the tool `YARA` can help by defining some rules that will trigger a warning.

3.1.2.2 Dynamic Analysis

In opposite to the static analysis, the dynamic analysis will run the file to analyze it. As running an unknown (and suspicious) piece of code presents some risks, these are run inside an isolated environment called sandbox (defined later on in the section 3.4). This can prevent the engineer system to be infected by the potential malware.

However, a malware might detect the sandbox and could not run any harmful code, so that it will not be classified as a malware from the engineer point of view. This is exactly what we aim to develop in this master thesis.

[1] A disassembler is a software that translates machine language into assembly language to make the code human-readable and therefore understanding the original source code behind it.

3.1.2.3 Symbolic Execution

The symbolic execution of a program is a way of analyzing it without the need to run it. The program will be analyzed by determining what input gives a specific part of the program to execute. This means that some parts of the software will be explored but not all of them, this is what an abstract interpretation is doing. The advantage of this approach is that it never gives false positive in opposite to an AI. Each result given by the symbolic execution represent a real and true possible path of the software.

Symbolic execution is not exclusively a malware detection technique in opposite to static/dynamic analysis. But we have chosen to introduce the symbolic execution in this section because it can be used for malware analysis, like we will see when we will speak about the SEMA toolchain (3.2).

3.1.3 Malware Mutation

A malware mutation is a modification of an original malware to allow it to bypass a previously established antivirus/sandbox detection. This mutation can be performed by anyone, not necessarily the malware original author. Moreover, the source code is not always needed to perform a malware mutation, as we can also encompass it into a software called packer (that will be described in section 3.3) and run the malware only when the detection technique is evaded. In this case, the evasion is performed by the packer itself and not the malware that is unchanged from the previously detected one.

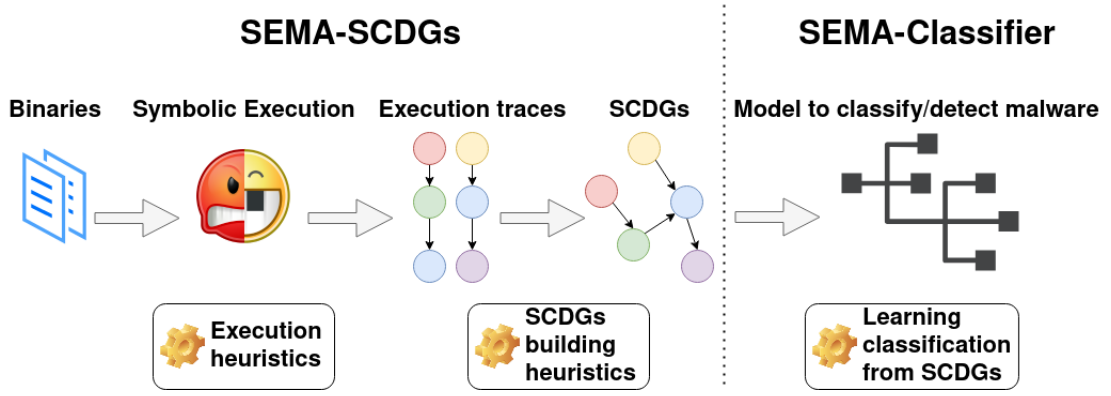
3.2 SEMA Toolchain

We will describe here the toolchain that will be mentioned in this master thesis. As previously said, this toolchain is developed by the Pr Axel Legay and his team. This toolchain is called `SEMA`[1], a Symbolic Execution open-source toolchain for Malware Analysis, which is briefly described below.

3.2.1 Architecture

The toolchain is divided into three main components: `SEMA-SCDGs`, `SEMA-Classifier` and `SEMA-Federated Learning`. We will rapidly describe the role of each of these components [3].

Figure 3.1: Workflow of SEMA | Source: https://github.com/AnonymousSEMA/SEMA-ToolChain



3.2.1.1 SEMA-SCDGs

The first part, the System Call Dependency Graphs extractor, is an extension of Angr, a symbolic execution framework. This module is capable of handling ELF and PE files, and it is dedicated to the efficient computation of SCDGs for malware analysis. It output the SCDGs as JSON files.

3.2.1.2 SEMA-Classifier

This part is in charge of the malware detection/classification. It works with SCDGs as input (in JSON format) and allows training them on classifiers and save the result. We can easily add new classifiers as this module is modular.

3.2.1.3 SEMA-Federated Learning

In this section, there are n devices (each of them trains its own model with its own dataset) that communicate with each other via a server in order to train an accurate classifier for malware detection. They used a deep neural approach where each trained model can be characterized by a set of parameters. These parameters are shared with the server.

3.2.2 Purpose

Most antivirus vendors use signature-based detection techniques (as we will explain in section 3.7) for classifying a software. These are static malware detection techniques. However, signature are very sensitive to any minor change in the malware, meaning that the malware will generally not be recognized anymore by the antivirus if it had been changed.

There are also dynamic detection techniques, but these can also be defeated with evasion techniques, like the ones described in sections 4.1.6 and 4.1.7.

The authors of anti-malware solutions have then adopted a new approach: symbolic execution, on which relies the SEMA toolchain. As explained before, it uses symbolic analysis-based malware classification with System Call Dependency Graphs (SCDG)[2].

3.3 Packer

A packer is a piece of software that will encapsulate another software (in this case a malware) and will deploy it once executed depending on some predefined conditions (stated by the user who has generated this packer). Multiple types of packers can be distinguished, depending on their purposes:

- **Packer**: The general term "packer" refers to an archive containing a piece of software. This software will be copied into the memory and executed when the packer is run. This is not exclusively used to perform malicious activities: for example, a packer such as UPX[3] can be used to reduce the size of an executable.
- **Crypter**: A crypter is a type of packer where the payload will be encrypted and/or obfuscated. A simpler crypter will only obfuscate its payload, but this can be easily de-obfuscated. A more complex crypter will perform an actual encryption of the payload.
- **Protector**: A protector will try to prevent reverse engineering of the executable. To do that it will combine the methods of the packer and the crypter, it will hide the obfuscated/encrypted payload in its executable and launch it only if certain conditions are met. It can also implement some techniques to evade a potential sandbox or a debugger.

3.4 Sandbox

The term "sandbox" is not specific to malware analysis. It can be used for software developers to have a reproducible/trustable environment to work with.

[2]The reader that might want to know more about this SEMA toolchain can read the papers called `Malware Analysis with Symbolic Execution and Graph Kernel` [27] from C-H. Bertrand Van Ouytsel and A. Legay, and `Tool paper - SEMA: Symbolic Execution toolchain for Malware Analysis` [3] from C-H. Bertrand Van Ouytsel et al.

[3]Free and open source executable packer, see https://en.wikipedia.org/wiki/UPX.

The main use of a sandbox is to isolate untrusted code in a virtual machine that cannot communicate with the host machine (or more generally the outside world).

In the cybersecurity field, a sandbox is a virtual machine dedicated to malware analysis. The purpose of this machine is to make the sample (i.e. the software being analyzed) trust that it was executed in a legitimate environment (i.e. not in a sandbox). For example, there is an internal network emulator to make the sample believe it is connected to the Internet.

3.5 Debugger

A debugger is a piece of software that helps a developer to find bugs in its program. It is usually used by developers when they are writing their software, but this is not the only usage of the debugger. It can also be used in cybersecurity, when an engineer wants to analyze the code path of a software to determine if it contains malicious intents. This is also one of the objectives of this master thesis: the purpose of our evasions techniques is to prevent the debugger to attach to our malware or, if it is not possible, it should not give hints about the malicious nature of the software being analyzed.

3.6 Sandbox/Debug Evasion

Some malware will try to escape the detection of the sandbox or avoid being debugged. These malwares implement some evasion techniques that will allow them to escape vigilance of the sandbox/debug (these techniques are covered in the sections 4.1.7 and 4.1.6). Obviously, the vendors of these sandbox/debug are aware of these evasions technique and are trying to implement some countermeasures to make it more difficult for the malware to escape them. This is a perpetual cat-and-mouse game between the sandbox/debug vendors and the malware creators.

3.7 Antivirus (AV)

An antivirus is a software built exclusively to protect a system against malware. It can monitor in real-time the operating system to try to detect malware before they can harm the system. To perform this difficult task, the antivirus use mainly four types of methods to detect malware: signature-based, heuristic-based, behavioral and cloud.

- **Signature-based method**: the antivirus can use a signature based on the software itself, like a cryptographic hash of the file or its sections. A database with all known viruses and malicious pieces of software is kept by the antivirus, which compares database signatures with the files on a system. It is quite efficient for malware that have been already detected before, but not for the new ones. A malware mutation can also affect the capabilities of this signature-based detection, as the signature will likely change with the mutation.

- **Heuristic-based method**: this method is useful to detect new malware as it does not use a signature but will look for characteristics that might be the ones of a malware. This method is not perfect, as one characteristic that the algorithm detects as potentially malicious can be enough to flag the whole executable as a malware. This might increase the number of false positive, but it is efficient to detect numerous true malware as it is quite sensitive. This type of analysis is made statically, the program is never executed.

- **Behavioral method**: In opposite to the heuristic-based method that does not execute the software, this method does execute it. It does the same as the previous one, i.e. it looks for behavior that might indicate a malware.

- **Cloud-based method**: This method does not analyze the software on the user system directly, but uploads it to a cloud-based infrastructure to analyze it there. This can help the antivirus vendors to extract patterns that might certainly indicate a malware and use it to improve its own defense techniques. As this method is community-based, all the clients of the antivirus might benefit from the result of the upload of the others.

3.8 Endpoint Detection and Response (EDR)

The Endpoint Detection and Response (EDR) implements an antivirus system in addition to some tools that help to detect malware more easily. An Endpoint is any terminal that the user might use. The EDR will not simply analyze the files executed on the system but will actively monitor all the known vulnerabilities of the system and the scripts executed (e.g. PowerShell scripts). The EDR can also report anomalies to a professional investigator (this person likely works in a Security Operations Center (SOC) and investigates incidents to decide whether it should be escalated or not).

Chapter 4

Design and Implementation

The following chapter will describe in details the implementation made in the scope of this Master Thesis. The same hardware configuration will be used for the two tools. The operating system that will be used to run the tools is Windows 7 as, we will see in the corresponding section, PANDA is currently only compatible with Windows up to version 7 (more specifically, its OSI plugins). This Windows OS will be run into a virtual machine with the following host configuration: Ubuntu 20.04.5 (kernel 5.4.0-132-generic) on VivoBook S14 X430UA with an Intel i3-8130U (4 cores) at 3.400GHz and 11859 MiB of RAM.

4.1 Packer (Protector) with Inceptor

4.1.1 Introduction to Inceptor

Inceptor [12] is a template-based packer for Windows designed to bypass AntiVirus (AV) and Endpoint Detection and Response (EDR). This tool automates the process of transforming a binary into a packed binary. This tool has been created by klezVirus, a security researcher [13].

The packer follows the workflow shown in figure 4.1. It first transforms the binary given in input into a shellcode that will be embedded into the packer. The binary is then converted via open-source converters. Inceptor is relying on 3 converters: `Donut`[26], `sRDI`[20] and `Pe2Sh`[9], each of them having its specific purpose: Donut can be used to transform native binaries, DLL (Dynamic Link Library) and .Net binaries into position-independent shellcode. This position-independent shellcode is a piece of code that can be executed regardless of its absolute address, meaning that none of the used address is hardcoded as it cannot assume where it will be placed in memory when executed. sRDI is an alternative to Donut and can convert DLL

into position-independent shellcode as well. Pe2Sh converts EXE into shellcode that can be run as a normal executable.

The shellcode is then given to a loader-independent encoder or a loader-dependent encoder (according to Inceptor's author terminology), depending on the initial configuration given when starting Inceptor. The difference between loader-dependent and loader-independent is that the loader-independent is not managed by the template chosen when starting Inceptor. It usually means that the decoding function of the shellcode is not given by Inceptor, but is embedded into the shellcode itself. This is possible thanks to `sgn`[8] or `Shikata ga nai`, an open-source tool designed to make the shellcode polymorphic and make it undetectable to a common signature-based malware detection (more on that in the related work section, at 5.3.1).

This type of encoder is not suitable for every template that the user may give to Inceptor. This is the reason why loader-dependent encoders exist as well, this type of encoder let the loader that care of the decoding, meaning that the decoder has to be implemented directly by the user in the template. These encoders have the capability of being chained, combining multiple of them to encode the payload. This chaining capability can help to better obfuscate the shellcode, but might expose the decoding routing to reverse engineers. They can use these to create a signature that will be used later by antivirus to help them to detect the malware.

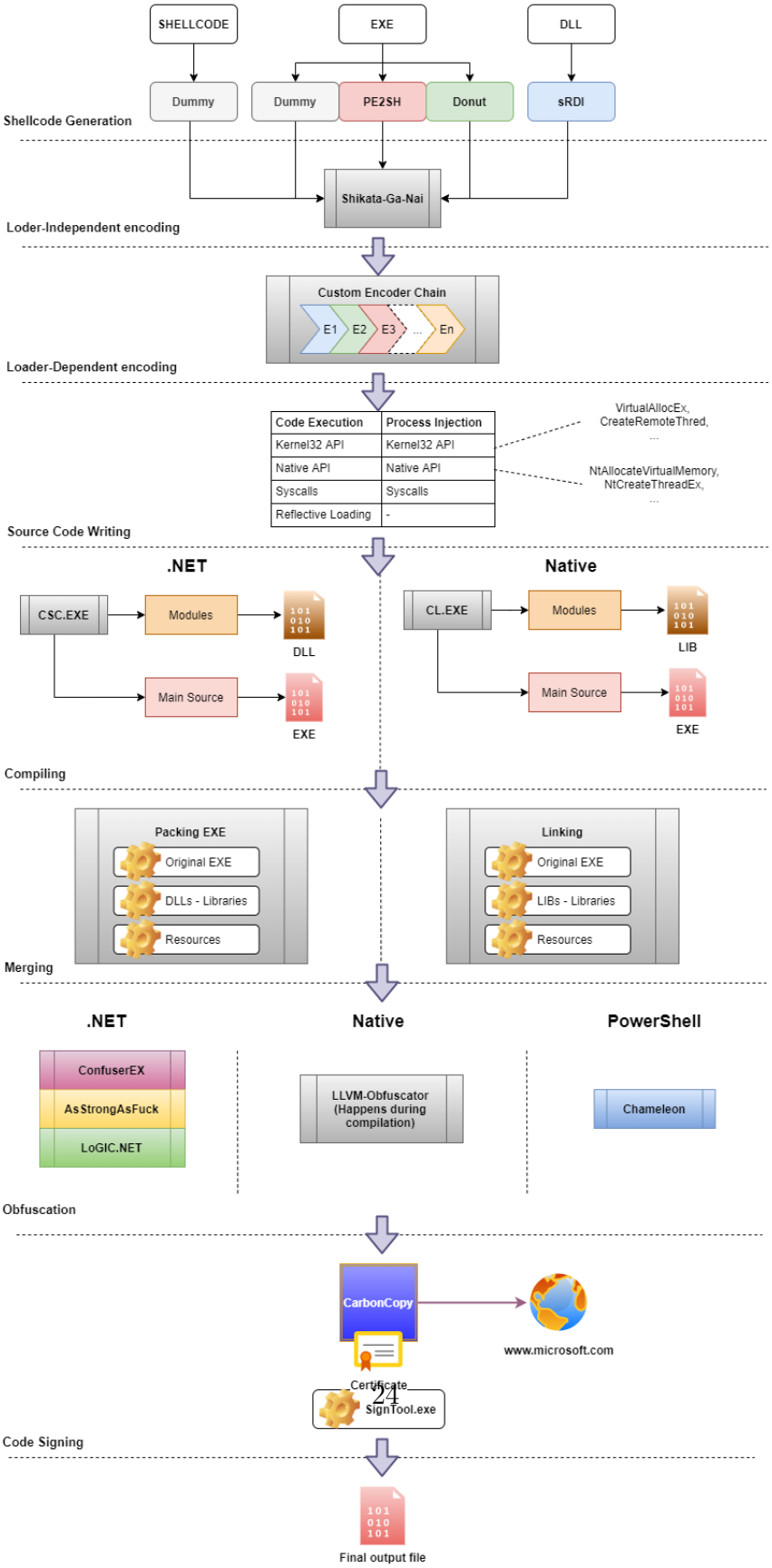
For this reason, Inceptor also implements some mechanisms to obfuscate the final artifact itself to harden the reversing. The tools implemented are `ConfuserEx`[19], `Chameleon`[11] and `LLVM-Obfuscator`[14]. Inceptor selects one of them depending on the language used (.Net, PowerShell or C/C++).

The last step of the workflow is to sign the generated artifact. Using `CarbonCopy`[22], Inceptor can sign the binary, as some antivirus check less strictly binaries with signature. These signatures are generally not even verified by antivirus.

4.1.2 Installation

As Inceptor is only compatible with Windows systems (in opposite to PeZor as we will see in 5.1), we have to set up a Windows machine before installing Inceptor. There is a Python script to facilitate its installation. This script is available in our GitHub repository (`https://github.com/dimitriwauters/inceptor`) or directly in the original one (`https://github.com/klezVirus/inceptor`) under

Figure 4.1: Workflow of Inceptor | Source: `https://github.com/klezVirus/inceptor`



the directory `inceptor/update_config.py`. We will first explain how to setup the needed virtual environment, then how to execute the script.

4.1.2.1 Python Virtual Environment

The virtual environment is quite simple to setup. First, the project needs to be cloned with the recursive flag. Then we have to go into the (cloned) Inceptor directory, use the "virtualenv" command from Python to create a new virtual environment and activate it with the `activate.bat` script generated automatically by Python. The last step is to install the required packages needed by Inceptor to run correctly.

```
git clone --recursive https://github.com/dimitriwauters/inceptor.git
cd inceptor
virtualenv venv
venv\Scripts\activate.bat
pip install -r requirements.txt
```

4.1.2.2 Visual Studio and Installation Script

Once the virtual environment is configured, we can execute inside it the following Python script: `update-config.py`. It will analyze our system to detect if all the prerequisites and dependencies are satisfied for running Inceptor.

This script will try to fetch `LLVM-Obfuscator` and detect if Visual Studio is installed. It will do this by looking for the existence of some specific path in the system. After making sure that Visual Studio is installed, it will check if the required compilers are also correctly installed. It will check for `Windows Clang`, `Windows BuildTools`, `Dumpbin` and for `Windows Code Signing Tools`.

When all the prerequisite are checked, Inceptor is ready to be used inside the virtual environment.

4.1.3 Command Line Usage

Inceptor can directly be used in the terminal of the virtual environment as follows:

```
$ usage: inceptor.py {native,dotnet,powershell} [-h] [-t {loader,
    pe2sh,donut,srdi}] [-C {cl,clang,llvm}] [-a COMPILER_ARGS] [--
    classname CLASSNAME] [--function FUNCTION] [--exports EXPORTS]
    [-e ENCODER] -o OUTFILE [-m MODULES] [-P] [-PO PROCESS] [--arch
```

```
    {x86,x64}] [--sgn] [-s] [-so] [-ss SIGN_STEAL] [-sd
   SIGN_DOMAIN] [-O] [--dll] [--clone CLONE] [--delay DELAY] [-hw]
    binary

inceptor: A Windows-based PE Packing framework designed to help
          Red Team Operators to bypass common AV and EDR solutions

positional arguments:
  {native,dotnet,powershell}
    native              Native Binaries Generator
    dotnet              .NET Binaries Generator
    powershell          PowerShell Wrapper Scripts Generator

```

There is only one mandatory parameter as described above: the type of the sample we want to pack (native, dotnet or powershell). The other parameters are optional.

The one that will activate the modifications made in the scope of this thesis is the one related to modules. A module is a plugin that will modify the behavior of Inceptor. The modules used for this master thesis are `AntiDebugModule` and `AntiSandboxModule`. These modules can be activated with the `-m MODULE_NAME` parameter, where `MODULE_NAME` is `anti_debug` or `anti_sandbox`.

When using these modules, we can add an option to choose the file(s) we want to use in the module: all evasion techniques are indeed not needed when activating a module and therefore these can be chosen via the following command: `-m MODULE_NAME[FILE(S) ID]`, where the file ID corresponds to the position or index of the file in the folder when we open it with a file explorer.

This capability was not originally present in the original version of Inceptor, we added it to make the automatization process of malware mutation easier.

For example, if we want to activate only the timing and flags debug evasion techniques, we would type the following command:

```
python .\inceptor.py native test.exe -o artifacts\test.exe -m anti_debug[1,4]
```

Considering the following folder ordering (and the position or index of the file starting from 0):

```
exception.nodebug.classic.cpp
flag.nodebug.classic.cpp
memory.nodebug.classic.cpp
```

```
object.nodebug.classic.cpp
timing.nodebug.classic.cpp
...
```

4.1.4 Inceptor Modifications

The main modification that we made to Inceptor is expending its capability of evasion. We added some template responsible to check if the binary is executed with a debugger attached or in a sandbox, the shellcode is executed only if all the checks are negatives. Indeed, we have modified Inceptor in two aspects: with debug evasion technique and also with sandbox evasion technique. Some technique already existing within Inceptor before our contribution, but the methods already present were quite simple. We will detail in the next sections 4.1.6 and 4.1.7 our implementation, separated into two part, one for each aspect.

Another modification made to Inceptor, already described in the previous section 4.1.3, is the addition of the possibility to choose the evasion technique we want to use. When running the command line, we have the possibility to indicate the file that will be used by Inceptor. When activating the module, we can pick the evasion technique that we want based on the file position in the folder. This feature was not present in the original unmodified version of Inceptor.

When using the Inceptor project without any modification, a simple Hello World packed will be flagged by 27 security vendors on VirusTotal. (py inceptor.py native -o test.exe main.exe). When we make the same verification on VirusTotal with some of the modifications that we added, only 5 security vendors flagged the executable. We will see some more examples during the evaluation of our modifications, at section 4.1.8.

4.1.5 Evasion Techniques Source

We implemented some evasion techniques found from different sources. Indeed, there are some resources that expose evasion technique on the Internet. Like on GitHub repositories, for example. We will cite them in this section, each of the presented tool has it own particularities.

4.1.5.1 LordNoteworthy/al-khaser

al-khaser[16] is a GitHub repository containing public known malware evasion techniques. It is a software that will check if an evasion technique works on a

machine and if so, warns the user through an interface.

4.1.5.2 a0rtega/pafish

pafish[2] is a GitHub repository doing the same as the previous shown al-khaser. This is also a tool that checks the feasibility of evasion techniques and alerts the user about the ones that work.

4.1.5.3 CheckPointSW for Sandbox/Anti-Debug

This is a website that regroups some evasion technique from different sources. Some of these sources are the GitHub repositories cited above. There are two versions of the website: one that regroups the sandbox evasion techniques and another one that does the same but for debug evasion techniques. These websites can be accessed at `https://evasions.checkpoint.com/` for the sandbox evasions and at `https://anti-debug.checkpoint.com/` for the anti-debug.

4.1.5.4 CheckPointSW/InviZzzible

InviZzzible[6] is a GitHub repository made by the same team from the two previously presented websites. It does the same as the other tools, i.e. it checks if a machine is vulnerable to some known evasion techniques.

The sandbox/debug evasion techniques that will be presented next will be all implemented into Inceptor. Obviously, there exist more evasion techniques as this is still an open question, and we will not cite them all here (you will likely find some more evasion techniques in the literature). This will be even more likely if the evasion techniques are kept private (or made public after the publication of this Master Thesis) or not well-known.

4.1.6 Debug Evasion Techniques

4.1.6.1 Exceptions

The first debug technique that we will present involves exceptions. This technique throws exceptions on purpose to check the behavior of the system. This can detect a behavior that will indicate the presence of a debugger attached to the analyzed process.

For example, when throwing a new/modified exception, the normal behavior of the system will be that the executed software handle the exception itself. But in presence of a debugger, that will be the debugger that will catch the exception

and not the process. To detect that, we will add a custom exception filter called from `UnhandledExceptionFilter` that will be called only if there is no debugger attached to the analyzed process. If the custom filter is executed, there is no debugger attached.

4.1.6.2 Flags

There are some special flags in the system that can indicate that the current process is being debugged. We can get the value of these flags at any time using Windows API calls or by directly accessing to the system tables in memory.

For example, we can call a Windows API function called `IsDebuggerPresent()` (from `kernel32`) that will check the `BeingDebugged` flag of the system. This is one of the most simple and most used ways of knowing if a debugger is attached to the process. As it is widely used, it is not very efficient to detect debugger that will try to hide themselves as it is generally one of the first technique that will be patched.

There are some other flags that can be checked to detect the presence of a debugger, with techniques that are a little more advanced. For example, with the `NtQueryInformationProcess()` (from `ntdll`) that can get some information about a specific process.

4.1.6.3 Memory

The process can look into its own memory region to check the presence of a debugger and if its presence is confirmed, try to interfere with the debugger. This can be done through multiple ways: searching in the process memory, thread context, making anti-attaching methods or by detecting breakpoints.

For example, by verifying code checksum, we can be able to detect software breakpoints. We will compute the checksum of a function beforehand and re-calculate it when executing the process to detect any changes that will be made to it, like for example the addition of software breakpoints.

4.1.6.4 Object

This technique uses kernel objects handles to detect the presence of a debugger. There exist some Windows API functions that get a kernel object in parameter that behave differently when a debugger is attached to the process.

For example, the `OpenProcess` API function (from `kernel32`) has this kind of behavior. Indeed, some files (like `csrss.exe`) will be accessible only if the current process is being debugged as, when being debugger, the process will be granted some new privileges like `SeDebugPrivilege`. With this privilege, we will be able to open files that will have not been accessible otherwise.

4.1.6.5 Timing

Obviously, when a debugger is attached to a process, the debugger slows it. This is not usual, delay can be measured to know if the process is being debugged. We will measure the delay between the instruction and its execution. To do that, we can use Windows API functions or use special functions if some flags are set (in kernel mode only).

For example, we can calculate the time that a function took to be executed (with or without a margin, depending on the sensibility we want to have) beforehand and compare this value at runtime with the actual time that the function took. If this value is greater than the pre-computed one, we can consider that a debugger is attached to the process. The Windows API function `GetLocalTime()` can be used to do just that.

4.1.7 Sandbox Evasion Techniques

4.1.7.1 CPU

The first sandbox evasion technique inspects directly the `Central Processing Unit` (`CPU`) by executing specific instructions to get information about the CPU or to detect behavior that will not occur on usual host OS.

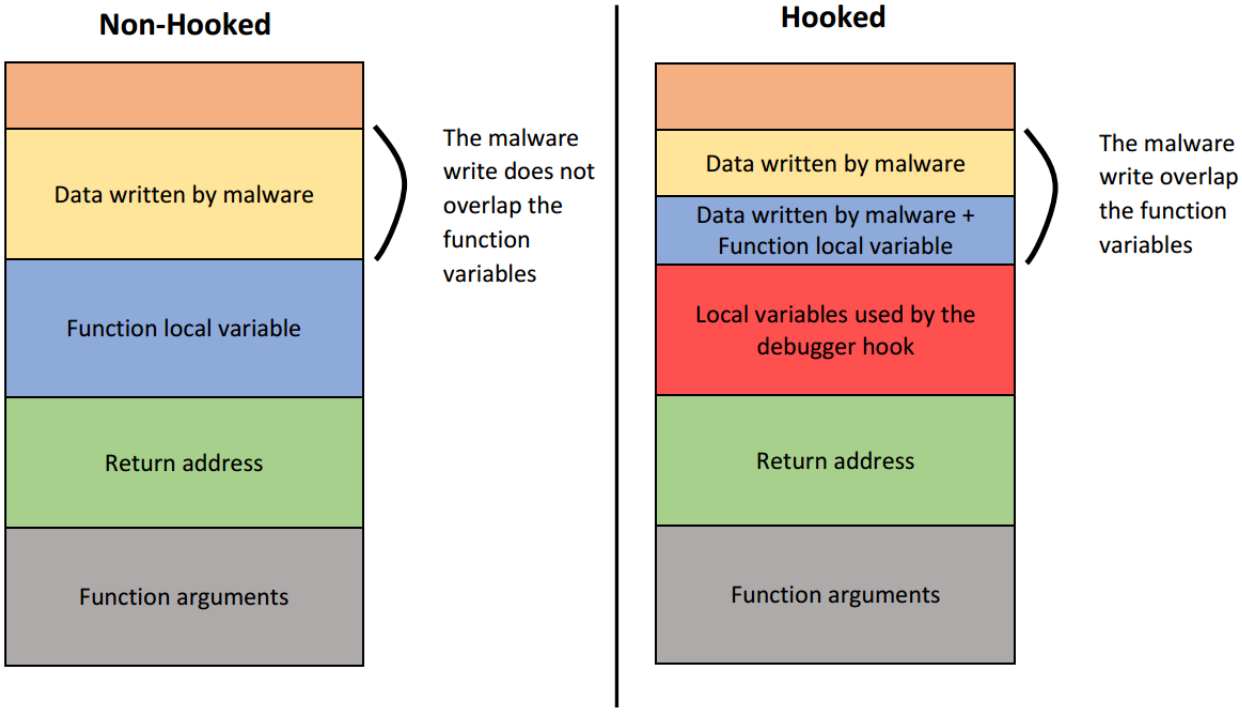
For example, we could fetch the CPU Vendor ID and compare it with known ones. As virtual machines often use a virtual CPU or hypervisor between the real hardware and the virtual machine, they have a specific known identifier. If the CPU Vendor Identifier matches one of the known ones, we can consider that we are in a sandbox environment.

4.1.7.2 OS Features

This uses some specific OS features of how the OS works, like having debug privileges right away when running the process (meaning that it is launched by a parent process having debug privilege itself, indicating a potential sandbox).

For example, we can check for unbalanced stack, that can be used for Cuckoo sandbox evasion. This technique, described by Alexander Chailytko and al. [5], consists in the following: a malware can get information about the space used by the called function and can therefore move the stack pointer toward the lower address with an offset sufficient to store the function arguments, return address and local variables. If the function is not hooked by a sandbox/anti-debug, the malware will fill the space up to the needed function data (function arguments, return address and local variables) without affecting them, the malware run without any perturbation. If the function is hooked, the writes made by the malware will overlap the space needed for the function to run correctly and might lead to the crash of the application if the corrupted data contains a pointer to some other function in the code, leading to arbitrary code execution.

Figure 4.2: Stack on non-hooked and on hooked function call[5]



4.1.7.3 Filesystem

This method will search for specific files or directories inside the host. If some of these files or directories are found, that certainly means that we are in a sandbox environment.

For example, we will check if the OS contains some files that are normally specific to virtual machines, like the following path that is specific to a driver of VirtualBox.

```
C:\WINDOWS\System32\drivers\VBoxVideo.sys
```

4.1.7.4 Generic OS Queries

Here, we will perform some queries into the OS to search for values that can indicate the presence of a virtual machine.

For example, we can fetch the current username to compare it with a list of known sandbox account username.

4.1.7.5 Human-like Behavior

This technique will search for actions that may differ from a normal human interaction with a computer. Some of these interactions can be checked via registers or other means.

By checking the registers, we could check the number of previously opened documents or check if the browser history contains more than a fixed number of visited URLs.

We could also check the user interactions, like monitoring the mouse movements or check the last input.

4.1.7.6 Network

The process can also look for a network connectivity, if it has one, to know if it is executed into a sandbox.

If there is no network connectivity, but a network card attached, we can compare the MAC address of the network card with a list of known virtual machine network card MAC address. We could do the same for the adapter name.

If there is a network connectivity, we could simply check the ISP name or location to have a clue.

4.1.7.7 Global OS Objects

We can check OS objects, like for the anti-debug from 4.1.7.4, to detect if the process is not in a virtual environment.

We could check the mutexes thanks to the Windows API function call `CreateMutex` and `OpenMutex`, or even check for virtual devices names specific to virtual machines with `NtCreateFile`. For example, when checking for a virtual device's name, we could check for the `VBoxVideo` virtual device in VirtualBox.

4.1.7.8 Processes

When running a virtual environment, some additional processes can be run in the background. These processes are usually not run in "normal" OS. We can search for theses specific processes to detect virtual machines.

For example, thanks to `EnumProcesses` from the Windows API (in `kernel32`), we can get the processes running on the system and compare them with a known list. Finding `vboxtray.exe` would mean that VirtualBox is running the VM.

4.1.7.9 Registers

Like the processes, some registers can be specific to an virtualized environment and can be caught in the same ways as the processes are. By using the Windows API function `RegOpenKey` we can read the content of a specific register or even list the differents keys with `RegEnumKeyEx` (all from `kernel32`).

4.1.7.10 Timing

As shown in the anti-debug section 4.1.6.5, timing measurement can be used to detect debuggers or sandbox in this case. Here we will base ourselves on the fact that sandboxes cannot run our process for a long time and will likely timeout after a fixed amount of time. For this method, the process can wait long enough to trigger that sandbox timeout. However, the sandbox can manipulate the clock to make the process run faster and prevent it to wait long enough to access the timeout threshold. As the clock will run faster for the whole process (and the Windows API too), we can measure the time that a function will take to be executed and compare it with a pre-computed measure. If the two are not equal (with a margin of error), we can consider that we are in a sandbox environment.

4.1.7.11 User Interface Artifacts

This method leverages the difference in terms of user interface between a legitimate host and a sandbox. Legitimately, a sandbox will have less opened windows than a regular user. In top of that, some sandbox environment uses specific windows names and therefore these can be detected as a presence of a sandbox environment.

The Windows API function `FindWindow` will return true if a window has the searched name. For counting the number of opened windows, `EnumWindows` in the Windows API exists.

4.1.8 Evaluation

We will now present the result of some software after passing them through the modified Inceptor. We will first describe the content of the software with a short sentence, then present the number of antivirus vendors that flagged the executable on VirusTotal.

- **Hello-world**: Small piece of code that displays "Hello world!" then exits immediately, does nothing malicious.
- **DarkTequila**: Spyware and worm[1] that targeted Mexican people. Its purpose was to steal bank credentials, passwords or any related information.
- **AgentTesla**: Trojan and spyware that registers user input (keyboard and clipboard) and sent them to the attacker.
- **Petya**: Ransomware known globally in the world, which costs some thousand million dollars to companies.
- **Stuxnet**: Like presented in the introduction and in the malware classification description, this is a worm, virus and rootkit that targeted Iranian nuclear installation.

We can see that even a legitimate software, once packed, is detected by some antivirus vendors as malware. This is due to the fact that Inceptor puts the shellcode of the software, in this case a Hello-World, into a protector. This is the protector that has been flagged, but not the content of the payload itself.

Even the well known malware can still hide themselves from antivirus vendors with these evasion techniques. This will not always be the case in real-world scenarios, as the antivirus available on VirusTotal are not the latest and the most complete versions.

[1]The reader can learn more about these types of malware in the section 3.1.1.

Name	Before	Applied techniques (Debug)	Applied techniques (Sandbox)	After
Hello-world	27/71	Flag, OS Object	OS Object, Network, Timing	5/71
DarkTequila	60/71	Flag, OS Object	OS Object, Network, Timing	4/69
AgentTesla	56/71	Flag, OS Object, Timing	OS Object, Network, Timing, UI	4/71
Petya	65/68	Flag, Timing	Timing, UI	5/71
Stuxnet	61/72	Flag, Timing	Timing, UI	5/72

4.1.9 Role in the Toolchain

This tool will be used in the toolchain to expand the initial database of malware. It will be used to automatically generate multiple mutations of the same malware for the toolchain to train on. The packer will try to escape the detection of the toolchain and the role of the toolchain will be to adapt to these evasion techniques to finally be able to detect all of them.

4.1.10 Future Improvements

After our modifications, we have not been able to go below 4 security vendor flags on VirusTotal. The first improvement that we can think of is to continue to implement checks to finally have a perfect score, meaning no detection at all.

Another modification is to improve the evasion methods implemented in the modified Inceptor. There is always room for improvement and there are always new methods that can be added.

The last improvement we will speak about here is to facilitate the choice of the evasion technique we want to implement to our payload. At this time, we have to know how files are arranged in the evasion technique directory to be able to choose them, as we will have to add their position in the command-line to activate them. Indeed, as said in the section 4.1.3, an evasion technique must be activated by the `-m MODULE_NAME` in the command-line, and it requires knowing exactly the position and the content of each file.

4.2 Detect Packers with PANDA

4.2.1 Introduction to PANDA

PANDA[21] is an open-source platform for Architecture-Neutral Dynamic Analysis. It means that this platform can perform dynamic analysis on software, independently of the hardware architecture and the operating system that it is run on. It is developed by the Panda-RE team. It is built on top of `QEMU`[23], an open-source machine and user-space emulator and visualizer. `QEMU` is capable of emulating a whole machine in software without the need of hardware virtualization. It can be combined with `KVM` or `Xen` to add a hardware virtualization to get an almost native CPU performance.

PANDA uses `QEMU` to get access to the guest operating system to have the possibility to analyze it. Thanks to callback made available to the user by PANDA, we will be able to catch when an application performs an action on the system. For example, when the software performs write on the memory.

Figure 4.3: Workflow of PANDA | Source: `https://github.com/panda-re/panda`

4.2.2 Installation

The installation of PANDA is quite straightforward. They have multiple ways of installing PANDA: Docker, Python PIP or from sources.

The installation with Docker is simple, we have only to pull the image `pandare/panda` from Dockerhub and run
`docker run -rm pandare/pandadev /panda/build/panda-system-i386 [CMD]`.
This docker has pyPANDA installed directly. If we want to run our modifications, the only command to run is `docker-compose up -d` as everything is packed inside a `docker-compose.yml` file. Once launched, the script will directly start to analyze the "payload" folder.

pyPANDA can also be installed directly on the host with PIP by running `pip3 install pandare`. This installs all the dependencies we need to run pyPANDA, but this does not provide PANDA binaries.

The last installation method depends on the operating system. There is an installation script provided for Ubuntu, Arch and macOS.

4.2.3 Use PANDA to Detect Packers

We use a Python interface of PANDA called `pyPANDA`[7] that will allow us to control PANDA with a simple Python script. We will use one of the callbacks provided by PANDA, this callback is `PANDA_CB_VIRT_MEM_AFTER_WRITE`. This callback will allow us to gather information when a program running in the guest OS performs a memory write. This memory write can reveal whether a software is packed or not. Indeed, we can deduce this by looking at the program counter and the written address, as we will explain in the next paragraphs.

This analysis is made inside a virtual machine handled by PANDA. As the plugins that PANDA uses to analyze the Windows operating system are only compatible to Windows 7 at most, we used Windows 7 for this thesis. The guest OS is stored inside a .QCOW2 file and will be load in a certain save-state for each sample. The plugins used here are OSI[2], Hooks[3] and Wintrospection[4].

We used the technique presented in the paper by Alessandro Mantovani and al. [17], which allows us to detect when a program tries to execute a memory area where it has previously written to. This detector will consist of two lists: one list for the memory region written by the program (later called `[WL]`) and a list for the memory region that has been executed and was already in the first list (later called `[WXL]`). We will gather the needed information through the `PANDA_CB_VIRT_MEM_AFTER_WRITE` PANDA callback that gives us the state of the CPU at the time of the write, the address pointed by the program counter (PC), the address where the write was made, the size of the write and finally the address of the buffer used for the write.

The information we want to extract is the address pointed by the program counter (PC) and the address where the write was made. With these we will be

[2]Availaible at `https://github.com/panda-re/panda/tree/dev/panda/plugins/osi`

[3]Availaible at `https://github.com/panda-re/panda/tree/dev/panda/plugins/hooks2`

[4]Availaible at `https://github.com/panda-re/panda/tree/dev/panda/plugins/wintrospection`

able to populate the first list [WL] with the tuple (PC, ADDR). As said before, this list contains the instruction that provoked the write (pointed by the program counter) and the address written to. When another write comes, we will compare the address pointer by the program counter and the addresses ADDR already present in the [WL] list. If there is a match, this means that the previously written memory region is now executed by the program. We can now copy the tuple from the [WL] list to the [WXL] list. Again, as said before, this list contains the written and executed memory regions.

When the program finished or when a timeout of 40 minutes occurs, the list [WXL] is returned and analyzed. If it is not empty and contains some consecutive memory regions, we can consider the analyzed software as packed.

We made a layer above the detector that will be responsible for the analysis and for the software handling. This consists of a Python script that will first explore the content of a defined directory and will treat the files present in this directory one by one. One by one, the files are packed into ISO files that are attached to the Windows 7 virtual machine launched by PANDA. The guest OS will load a save-state with a prompt open for each file and mount the ISO as a CD-ROM input. Then the software is executed, and the output is analyzed as described above. This loops for every file in the directory.

4.2.4 Script Usage

To run our script, a folder called `payload` must be created just next to the script itself. This folder will then be populated with the sample we want to analyze. The script must be run inside the PANDA Docker container, the easiest way to do that is to create a `docker-compose.yml` file and run the PANDA image with the script as the command and the `payload` folder as a volume (it is important that the folder is called `payload` inside the Docker container, but the name can be changed on the host). The complete `docker-compose.yml` file is available in the appendix.

4.2.5 Create a Windows Image

PANDA provides a list of ready-to-use pre-built `QEMU` virtual machines that can be automatically downloaded and can be controlled through a prompt. Unfortunately for us, these VMs are only based on `Linux` and `FreeBSD`, not `Windows`. We will have to build ourselves a Windows guest that will be used for our implementation of PANDA.

To do that, we have downloaded a Windows VM pre-built by Microsoft (IEUser) on top of which we can build our machine by adding it the required environment.

As we cannot use a prompt directly with a Windows VM in opposite to a Linux/FreeBSD VM, we will have to use some tricks. To bypass this issue, we will use the interface given by `QEMU`. This interface will allow us to save the VM state when we open the command prompt and load it later when needed.

We will apply this to our freshly downloaded Windows VM. We will first start it and then directly open a prompt (cmd.exe). Once it is done, we will immediately save the VM state in the first slot of `QEMU`. This state will be load by our script for each sample that we want to analyze.

4.2.6 Other PANDA Capabilities

In this section, we will address briefly the other capabilities of PANDA that we have not used in the scope of this Master Thesis. We have chosen to speak about these capabilities as they can eventually be leveraged for future improvements.

4.2.6.1 Record / Replay

PANDA can record the execution of the virtual machine into a file and replay it inside PANDA afterward without having to rerun the whole virtual machine. This allows the user to repeat exactly the same action every time, this is not possible if the system is re-run because the system is non-deterministic. Indeed, there are many factors that could cause the system to behave differently like network traffic, hard-drive read/write, etc. The record system does not allow to "go live" at the middle of the recording because it does not record the inputs. The recording system can be useful when we want to analyze an aspect of the system that would take too much time to analyze in real-time.

4.2.6.2 Callbacks

As previously said, PANDA puts at our disposal 48 callbacks that will be called when a certain event occurs. These callbacks can be used to perform analysis during the execution of the sample. For example, for our usage of PANDA, we used a callback `PANDA_CB_VIRT_MEM_AFTER_WRITE` that will be executed each time a process perform a memory write. This callback is called during real-time execution and during a replay. The complete documentation is available on the PANDA GitHub[5].

[5] `https://github.com/panda-re/panda/blob/dev/panda/docs/manual.md#appendix-a-callback-list`

4.2.6.3 Plugins

A lot of plugins are available to extend PANDA. This allows a lot of new dynamic analysis techniques to be added to PANDA without the need to modify `QEMU`. We can use plugins written by the community, some of them are directly packed with PANDA and can be used directly. We can also build our own plugins that will perfectly fit our needs, a complete documentation is available on the PANDA GitHub[6].

4.2.7 Role in the Toolchain

The role of this PANDA implementation in the toolchain will firstly be experimental. It will be used as a basis for a new packing detection module that will be added to the toolchain. Another implementation of a detection mechanism will be implemented to the toolchain and PANDA will be there as a sort of backup. When the detection mechanism will be not certain of the result or cannot analyze the sample, the toolchain will use PANDA to determine if the sample is packed or not.

4.2.8 Future Improvements

The first improvement is to include an entropy check of each section before and/or during the analysis of the sample. This entropy check can give a hint on the fact that a software might be packed. Indeed, the packing process introduces an increased entropy in one of the sections of the packer (ideally the .data) due to the fact that its payload is probably obfuscated or encrypted.

[6] `https://github.com/panda-re/panda/blob/dev/panda/docs/manual.md#writing-a-plugin`

Chapter 5

Related Work

In this section, we will discuss the state-of-the-art. We will discuss other work that could serve the same purpose as the one discussed in this Master Thesis: building a mutation tool for malware. These represent the other directions that could have been taken to respond to the presented problem.

5.1 PEZor: Open-Source Shellcode & PE Packer

5.1.1 Explanation

PEZor[25] is Open-Source Shellcode & PE Packer created by Francesco Soncina. This is a packer made to run exclusively on GNU/Linux distributions because the author wanted a solution that does not require installing the whole Visual Studio Windows suite. As the author wanted that the tool still works on Windows, he had to find another compiler that work on both Windows and Linux distributions, `Mingw-w64`.

5.1.2 Differences with Inceptor

PEZor is an alternative packer to Inceptor. The two software are quite similar in the way they work, but they have each some particularities. Inceptor has some specificity that makes it more interesting for this Master Thesis, it has a template capability that PEZor does not have. We could have made the same modifications on PEZor, but they would not have been so easy to implement, as we would have to add an evasion selection at the command-line level to have the same functionalities.

Inceptor has also some more advanced capabilities on certain features (.NET obfuscation, .NET anti-debug, loader signing, etc.) [15] that makes it more fitted to make this Master Thesis project more future-proof.

5.2 Binary Rewriting by Modifying the Intermediate Representation (IR)

Another approach that we could have made for the scope of this Master Thesis is to use the Intermediate Representation (IR) of the binary and manipulate it to introduce the mutation into the executable. We can use, for example, Low Level Virtual Machine (LLVM) that is an Intermediate Representation between a high-level language (C/C++/...) and assembly code.

This aspect is not at all discussed in this Master Thesis, as it is the subject of some students that will finish their Master Thesis in the same year. If the reader wants more information about this subject, he/she can read their paper. This paper is called "Building a mutation tool for malware", made by Arnold Gauthier and Bastien Wiaux under the supervision of Pr Axel Legay during the 2022-2023 academical year.

5.3 Self-Mutating Malware Detection with Code Normalizer

A malware can self-mutate itself, without any exterior interaction. This type of malware can modify its malicious code by itself.

However, we can prevent these self-mutating malware to mutate by detecting them beforehand by using code normalization. This can be done by firstly normalize the code of the malware, meaning that we will transform it into a canonical form. This is a simpler version of the software to analyze while maintaining the original semantic. The normalization process will remove all the noise that the malware generated during its mutation. When self-mutating, a malware will generally generate an unoptimized version of its original malware code, that is what we call noise here. Removing this noise also help to remove dead code, code that will never be called. This type of analysis can be categorized into the static analysis category.

Some papers [4] use this method to detect self-mutating malware thanks to the addition of a control-flow graph matching. It is done in three steps: first, transforming the executable machine code into a new machine code that allows to evaluate each instruction in terms of operation made by the CPU. Second, performing a control-flow analysis and a data-flow analysis to detect dependencies and relationship among the instructions. Third, performing the code transformation on the code gathered on the first step with the data collected on the second step.

This transformation is used to normalize the software and therefore detect malicious code if any.

There are generally two types of self-mutating malware. We will describe them now.

5.3.1 Polymorphic

A polymorphic self-mutating malware is capable of modifying itself to stay undercover for the anti-viruses. For example, it can be capable to change its encryption key and modifying accordingly its decryption routine with obfuscation methods. As the signature of the malware is modified at each encryption key modification, the antivirus that uses signature-based detection can never recognize this type of malware. Its code stays the same between each decryption, this is why it is called poly-morphic, multiple appearances.

5.3.2 Metamorphic

A metamorphic self-mutating malware has no encryption capabilities but can modify its obfuscation technique itself. By changing its obfuscation technique, the malware does not look the same for the antivirus and can escape it. In opposite to the polymorphic malware, the metamorphic malware change its code at each time it is propagated to a new host. This can be done simply by varying the number of NOP instructions at runtime, for example.

The downside of the metamorphic malware is that the metamorphic engine stays the same between each distribution, there is only the malware code itself that changes for each infected user. This might facilitate its recognition by antivirus.

Chapter 6

Conclusion

The race between malware authors and antivirus vendors is, and will always be, an important point of cybersecurity. Malware will always mutate to escape the vigilance of antivirus, and the antivirus vendors will always have to adapt to be able to counter these mutations quickly as possible. This is a real challenge that must be faced every day to be effective.

This Master Thesis addresses this challenge by creating a semi-automated mutation system that can be used to generate protectors that will include mutations of a payload (in the scope of this paper, these payloads will be malware) to allow a toolchain called SEMA and developed by Pr Axel Legay and his team to pre-generate mutations for their initial malware database that the toolchain train on. This protector generator is built on top of a template-based packer generator for Windows called Inceptor. The other contribution to this challenge made by this Master Thesis is a packer detector made with PANDA, an open-source platform for Architecture-Neutral Dynamic Analysis. This will also be implemented into the toolchain SEMA to detect packer when their initial implemented method fails or does not return a convincing result.

However, as all tool that we can find in the wild, these are not perfect and must be adapted continuously to implement the latest evasion techniques known publicly (or not). This can be a part of the future work for the scope of this Master Thesis, update/upgrade the templates put on top of Inceptor to adapt them to be up-to-date with the new evasion techniques that will certainly come out after the publication of this Master Thesis.

Bibliography

[1] SEMA. URL: https://github.com/AnonymousSEMA/SEMA-ToolChain.

[2] a0rtega. pafish. URL: https://github.com/a0rtega/pafish.

[3] Charles-Henry Bertrand Van Ouytsel, Christophe Crochet, and Axel Legay. Tool paper - SEMA: Symbolic Execution toolchain for Malware Analysis. CRiSIS '22: Proceedings of the 17th International Conference on Risks and Security of Internet and Systems, 2022. URL: http://hdl.handle.net/2078.1/267919.

[4] Danilo Bruschi, Lorenzo Martignoni, and Mattia Monga. Detecting Self-mutating Malware Using Control-Flow Graph Matching. In David Hutchison, Takeo Kanade, Josef Kittler, Jon M. Kleinberg, Friedemann Mattern, John C. Mitchell, Moni Naor, Oscar Nierstrasz, C. Pandu Rangan, Bernhard Steffen, Madhu Sudan, Demetri Terzopoulos, Dough Tygar, Moshe Y. Vardi, Gerhard Weikum, Roland Büschkes, and Pavel Laskov, editors, *Detection of Intrusions and Malware & Vulnerability Assessment*, volume 4064, pages 129–143. Springer Berlin Heidelberg, Berlin, Heidelberg, 2006. Series Title: Lecture Notes in Computer Science. URL: http://link.springer.com/10.1007/11790754_8, https://doi.org/10.1007/11790754_8 doi:10.1007/11790754_8.

[5] Alexander Chailytko and Stanislav Skuratovich. Defeating Sandbox Evasion: How To Increase The Successful Emulation Rate In Your Virtual Environment. October 2016. URL: https://blog.checkpoint.com/wp-content/uploads/2016/10/DefeatingSandBoxEvasion-VB2016_CheckPoint.pdf.

[6] CheckPointSW. InviZzzible. URL: https://github.com/CheckPointSW/InviZzzible.

[7] Luke Craig, Andrew Fasano, Tiemoko Ballo, Tim Leek, Brendan Dolan-Gavitt, and William Robertson. PyPANDA: Taming the PANDA-monium of Whole System Dynamic Analysis. In *Proceedings 2021 Workshop on Binary Analysis Research*, Virtual, 2021. Internet Society.

URL: https://www.ndss-symposium.org/wp-content/uploads/bar2021_23001_paper.pdf, https://doi.org/10.14722/bar.2021.23001 doi:10.14722/bar.2021.23001.

[8] EgeBalci. sgn. URL: https://github.com/EgeBalci/sgn.

[9] hasherezade. Pe2Sh. URL: https://github.com/hasherezade/pe_to_shellcode.

[10] Alexa Hernandez. Cyber Attacks on IoT Devices Are Growing at Alarming Rates. August 2021. URL: https://www.venafi.com/blog/cyber-attacks-iot-devices-are-growing-alarming-rates-encryption-digest-64.

[11] klezVirus. chameleon. URL: https://github.com/klezVirus/chameleon.

[12] klezVirus. Inceptor. URL: https://github.com/klezVirus/inceptor.

[13] Klezvirus. klezvirus.github.io - Random Stuff about CyberSecurity. URL: https://klezvirus.github.io/.

[14] klezVirus. obfuscator. URL: https://github.com/klezVirus/obfuscator.

[15] klezvirus. The path to code execution in the era of EDR, Next-Gen AVs, and AMSI. URL: https://klezvirus.github.io/RedTeaming/AV_Evasion/CodeExeNewDotNet/.

[16] LordNoteworthy. al-khaser. URL: https://github.com/LordNoteworthy/al-khaser.

[17] Alessandro Mantovani, Simone Aonzo, Xabier Ugarte-Pedrero, Alessio Merlo, and Davide Balzarotti. Prevalence and Impact of Low-Entropy Packing Schemes in the Malware Ecosystem. In *Proceedings 2020 Network and Distributed System Security Symposium*, San Diego, CA, 2020. Internet Society. URL: https://www.ndss-symposium.org/wp-content/uploads/2020/02/24297.pdf, https://doi.org/10.14722/ndss.2020.24297 doi:10.14722/ndss.2020.24297.

[18] N. Milosevic. History of malware. *Digital forensics magazine*, (1(16)):58–66, 2013. URL: https://pure.manchester.ac.uk/ws/portalfiles/portal/32297162/FULL/_TEXT.PDF.

[19] mkaring. ConfuserEx. URL: https://github.com/mkaring/ConfuserEx.

[20] monoxgas. sRDI. URL: https://github.com/monoxgas/sRDI.

[21] panda re. PANDA. URL: https://github.com/panda-re/panda.

[22] paranoidninja. CarbonCopy. URL: https://github.com/paranoidninja/CarbonCopy.

[23] qemu. qemu. URL: https://github.com/qemu/qemu.

[24] Val Saengphaibul. A Brief History of The Evolution of Malware. March 2022. URL: https://www.fortinet.com/blog/threat-research/evolution-of-malware.

[25] Francesco Soncina. PEzor. URL: https://github.com/phra/PEzor.

[26] TheWover. Donut. URL: https://github.com/TheWover/donut.

[27] Charles-Henry Bertrand Van Ouytsel and Axel Legay. Malware Analysis with Symbolic Execution and Graph Kernel. 2022. Publisher: arXiv Version Number: 1. URL: https://arxiv.org/abs/2204.05632, https://doi.org/10.48550/ARXIV.2204.05632 doi:10.48550/ARXIV.2204.05632.

[28] Wikipedia contributors. Brain (computer virus) — Wikipedia, the free encyclopedia. https://en.wikipedia.org/w/index.php?title=Brain_(computer_virus)&oldid=1116591763, 2022. [Online; accessed 27-December-2022].

Appendix A

PANDA: Docker-compose File

In this docker-compose file, the `dev` folder contains the script `run_panda.py`, this is not mandatory to have that but the docker-compose file must be changed accordingly.

The `.root` folder is mandatory as it must contain the `QEMU` file for the Windows 7 virtual machine.

```
version: "3.3"
services:
  pandare:
    image: pandare/panda
    command: /bin/sh -c "python3 /addon/run_panda.py"
    volumes:
      - "./dev:/addon"
      - "./payload:/payload"
      - "./.panda:/root/.panda"

```

Appendix B

Inceptor VirusTotal Evaluation

You will find in the list below the ID generated on VirusTotal. You can use these IDs to recover the whole VirusTotal report. You will have to replace `[ID]` from the url `https://www.virustotal.com/gui/file/[ID]` to see the report.

IDs BEFORE addition of the evasion techniques:

- **Hello-world**: 5356eb610c6da4df351ea370e03223b59daee0ad9f1e6a8578ad50f35bd5e35f
- **DarkTequila**: dce2d575bef073079c658edfa872a15546b422ad2b74267d33b386dc7cc85b47
- **AgentTesla**: 699ec052ecc898bdbdafea0027c4ab44c3d01ae011c17745dd2b7fbddaa077f3
- **Petya**: 027cc450ef5f8c5f653329641ec1fed91f694e0d229928963b30f6b0d7d3a745
- **Stuxnet**: 9d88425e266b3a74045186837fbd71de657b47d11efefcf8b3cd185a884b5306

IDs AFTER addition of the evasion techniques:

- **Hello-world**: 9ba961f7b61682d88ce5a3397eaf9d0d300a6da6617cbb429077c90947b942a4
- **DarkTequila**: 4dcc4f5b71eae3394fbcbd53511192c796b2553e50ab5dae3b6a2bbbad6ae988
- **AgentTesla**: d2453597d7d7c1cefb196f28ab2a3f698de59a60d49d19dc0a02db004d96d658
- **Petya**: 6393014f0cfe08b40f51fd5b5e3ac6167877705d7ab35f06a01a342c0775a3bd
- **Stuxnet**: 795594308f6f9c7b9b3a5dc59c7b30c9d40cdb095cf54598cb90df6c21d0a99e

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