Tampere University

Kamu Malinen

STEALTH MALWARE TECHNIQUES

A survey on how stealth malware maintain stealth

Bachelor's thesis
Faculty of Information Technology and Communication Sciences
December 2023

ABSTRACT

Kamu Malinen: Stealth malware techniques
Bachelor's thesis
Tampere University
Bachelor's Programme in Computing and Electrical Engineering
December 2023

This thesis surveys techniques malware use to maintain stealth on the host machine. The main focus is on rootkits and their methods, but both anti-malware methods, malware types and other methods of evading detection are briefly introduced. Some methods used by anti-malware are projected onto stealth techniques to highlight the motives behind different methods. It should be noted that this thesis isn't a comprehensive survey and therefore many malware types and techniques used by both malware and anti-malware aren't noted. Instead, the aim is to introduce the most common and simplest types and techniques.

This thesis is a literature review and the sources for this work were selected by a keyword search on peer-reviewed academic texts that were then checked to be relevant manually. Sources that aren't strictly academic are only used when a further explanation of the subject matter is required and the subject matter is too precise to find trustworthy and or detailed enough information from an academic source. This only concerns definitions of malware types, definitions of parts of a commercial system or examples of malware in the wild and even then critique has been applied so that only relevant actors within the topics' fields were chosen.

The work consists of three parts: an introduction to techniques used by anti-malware, an introduction to types of malicious software and their techniques, and lastly a brief conclusion regarding techniques and general observations.

While some of the types of malware are introduced, it's noted that it is rather difficult to define them accurately due to their abstract nature and vague and differing or even contradicting definitions by notable actors in the field of cybersecurity. Techniques used to evade detection — except for obfuscation — are observed to mostly place the malware outside of the anti-malware scope of operation. Obfuscation techniques in turn try to change the appearance and behavior of the malware to not match those of malware known by the anti-malware and therefore manage to evade methods that use pattern matching for detection.

Keywords: stealth, malware, detection, evasion

TIIVISTELMÄ

Kamu Malinen: Piilohaittaohjelmien tekniikat
Kandidaatintyö
Tampereen yliopisto
Tieto- ja sähkötekniikan kandidaattiohjelma
Joulukuu 2023

Tämä työ tutkii piiloutuvien haittaohjelmien tekniikoita isäntäkoneella havaituksi tulemisen välttämiseksi. Työn pääpainona on rootkit-haittaohjelmat ja niiden käyttämät tekniikat, mutta myös haittaohjelmien torjuntaan käytettyjä tekniikoita, haittaohjelmatyyppejä ja eri tapoja välttää havaituksi tuleminen esitellään lyhyesti. Joitakin haittaohjelmien torjuntaan käytettyjä menetelmiä käytetään korostamaan tarvetta peitetekniikoille. On huomattava, että tämä työ ei ole kattava tutkimus ja siksi useita haittaohjelmatyyppejä, sekä haittaohjelmien käyttämiä ja niiden torjuntaan käytettyjä tekniikoita ei huomioida. Tarkoituksena on esitellä yleisimmät ja yksinkertaisimmat tyypit ja tekniikat.

Työ on kirjallisuuskatsaus, johon lähteet on valittu avainsana haulla vertaisarvioiduista akateemisista teksteistä, joista valittiin merkitykselliset lähteet käsin. Lähteitä, jotka eivät ole akateemisia käytetään vain, kun aihealue on niin tarkka, ettei siitä löydy tarpeeksi luotettavaa tai yksityiskohtaista akateemista lähdettä. Tämä koskee vain haittaohjelmatyyppien määritelmiä, kaupallisten järjestelmien määritelmiä ja asiakirjoja koskien haittaohjelmia luonnossa. Tällöin lähdekriitiikkiä on sovellettu siten, että vain aiheen alalla keskeisten tekijöiden tuottamia tekstejä valittiin lähteiksi.

Työ koostuu kolmesta osasta: johdatus haittaohjelmien torjunnan menetelmiin, johdatus haittaohjelmatyyppeihin, sekä lyhyt yhteenveto tekniikoista ja havainnoista.

Vaikka joitakin haittaohjelmien tyyppejä esitellään, on huomattava, että niiden tarkka määrittely on haasteellista abstraktin luonteen ja kyberturvallisuudessa huomattavien tekijöiden antamien epämääräisten, eroavien tai jopa ristiriitaisten määritelmien takia. Havaituksi tulemisen välttämiseksi käytettyjen tekniikoiden — lukuunottamatta hämärtämistekniikoita — huomataan pääasiassa sijoittavan haittaohjelma antiviruksen toiminnan ulottumattomiin. Hämärtämistekniikat puolestaan pyrkivät muuttamaan haittaohjelman ulkoasua ja käyttäytymistä siten, että ne eivät muistuta tunnettujen haittaohjelmien piirteitä, eivätkä siksi tule haivaituksi kuviosovitus tekniikoiden toimesta.

Avainsanat: piilo, haittaohjelma, havaitseminen, piiloutuminen

CONTENTS

1. Introduction 1
2. Anti-malware techniques 3
2.1 Signature scanning 3
2.2 Behavioral detection 3
2.3 Heuristic detection 4
2.4 Hook detection 4
2.5 Virtualization 4
3. Malicious software 6
3.1 Types of malicious programs 6
3.1.1 Riskware 6
3.1.2 Viruses 6
3.1.3 Worms 7
3.1.4 Trojans 7
3.1.5 Rootkits 7
3.2 Evasion techniques 8
3.2.1 Obfuscation 8
3.2.2 Anti-emulation and targeting mechanisms 9
3.2.3 Virtualization and hardware 10
3.2.4 Process injection and hooking 10
3.2.5 Direct kernel object manipulation 11
3.3 Hooking techniques 11
3.3.1 User and kernel space 11
3.3.2 Import address table hooking 12
3.3.3 Inline function patching 13
3.3.4 SSDT hooking 14
4. Conclusion 15
References 16

1. INTRODUCTION

There's a wide variety of malware — short for malicious software — and understanding the type and functionality of malware helps in both documenting, explaining and repelling them. However, classifying becomes more difficult as modern malware combines methods of multiple malware types. To make things worse, malware is constantly developed to evade existing countermeasures set by cybersecurity professionals. To keep up with this seemingly endless cycle of development, it's important to be up to date on what and how malware does to avoid evasion in the present.

As both malware and their countermeasures are ineffective if they are known by each other's developers, the information available is limited. Therefore, as this thesis is a literature review, the thesis is limited to available information on the most common types of malware and their countermeasures. Since the subject of the thesis is stealth malware, there's an emphasis on stealth capabilities rather than propagation models or malicious acts the malware commits. Additionally, while historical knowledge is important and shouldn't be neglected, this thesis attempts to capture a modern view of the way stealth malware and their counterparts act.

This thesis surveys some of the methods both benevolent and malicious actors use and introduces and defines some malware types in two sections. The first one introduces methods antimalware use to detect malware and the second introduces and defines malware types, their methods of hiding while emphasizing rootkits and the most common method they use to maintain stealth. This order of presentation is chosen to help rationalize the motivation behind some methods, as stealth techniques have been developed only after countermeasures have been put in place. Even though modern malware may act like multiple or different malware types at times, a few types are introduced to give the reader an insight into what different types are used to accomplish.

Table 1.1. *Keywords*

malware	rootkit
stealth	evasion
detection	anti-malware
antivirus	morphism
process hiding	obfuscation

The survey is implemented as a literature review of available literature. Searching for literature is carried out using keyword searches on multiple search engines including `scholar.google.com`, `google.com`, *IEEE Xplore*, *ACM Digital library* and Tampere University's *Andor*. As `google.com` results aren't strictly academic, sources found there are viewed with caution and only used if available academic sources aren't deemed to be precise, reliable or relevant enough. Search queries are formed by using different combinations of keywords in table 1.1 sometimes using the AND operator to signify the requirement of both keywords appearing. Finding complementary information is done using the topic as the query. For example, *PE format* is used to find information regarding Windows' packaged executable.

2. ANTI-MALWARE TECHNIQUES

To understand the purpose of stealth malware techniques, it's beneficial to understand common anti-malware and Intrusion Detection System (IDS) techniques i.e. their methods of detecting malicious software. Understanding these assists in understanding the motivation behind certain stealth malware techniques.

This chapter introduces signature scanning, two less complex methods based on dynamic analysis, hook detection and virtualization which is used as a tool for dynamic analysis to maintain security. While these don't cover nearly all of the existing methods, these give some insight as to what common stealth methods attempt to evade.

2.1 Signature scanning

Signature scanning uses signatures generated from code fragments which are compared to a signature database of known threats. The signature can be either the code fragment or it can be some hash of a code fragment. Hashes are beneficial as different hashing algorithms can be used to maintain secrecy over what piece of the program is used to generate the signature, which stops malware developers from avoiding matching signatures. [1, 22, 23]

Signature scanning is a simple yet efficient way of detecting known malware. An advantage signature scanning has regarding safety is that it's static and doesn't need to execute code to determine whether or not the code is malicious. However, there are disadvantages to this method. Simple signature scanning can't detect new threats, comparing samples to an evergrowing database gets slower over time requiring extended storage and computing solutions, and if implemented poorly so that the attacker can extract the signature used to detect a threat, the anti-malware can be exploited to carry out attacks on legitimate files by implanting the signature onto target files or programs. [23] There are ways to mitigate these issues, for example, machine learning has been deployed for sequence analysis to overcome not detecting new threats [22] and the extraction of signatures can be made more difficult with previously mentioned hashing.

2.2 Behavioral detection

Behavioral detection is based on analyzing the behavior of a program and unlike signature scanning, behavioral detection is a dynamic technique. Behavior can consist of things such as system calls, file changes or network activities, but isn't limited to them and could consist of just analyzing the program in a sandbox, which is further explained in 2.5. [1]

Behavior-based analysis has at least two obvious drawbacks, firstly the program has to be run, which means that if the program is malicious it can cause harm upon running. Secondly, discerning between malicious and legitimate behavior may be difficult as some benevolent applications use similar methods to malicious applications. However, the positive aspect of this method is that it can detect new threats so long as the behavior of the new threat is detectable i.e. similar enough to the method known to be malicious. [1, 23]

2.3 Heuristic detection

To overcome the shortcomings of behavioral detection and signature scanning, heuristic methods of detection were implemented[1, 2, 23]. Heuristic-based detection uses machine learning and data mining to analyze and classify a program based on defined features. API calls, OpCode and n-grams are three features in use for heuristic methods. However, unlike in behavioral detection, API calls and OpCode may be data mined instead of obtained by running the program. [2]

In a sense, heuristic detection resembles both behavior detection and signature scanning, as the behavior and different types of signatures are run through machine learning algorithms to conclude the quality of the program. API calls and OpCode illustrate the behavior of the program, while n-grams are similar to signatures, as they are substrings of parts of the program used to identify it.

2.4 Hook detection

Hooks are explained further in 3.2.4 and 3.3, but for now it suffices to think of it as attaching to a part of a program or the system so that the attached code is run with the code called. It's important as rootkits and other malware with stealth capabilities exploit hooks. Hook detection is, at its easiest, done by hooking attack points first to notice when it's hooked again. While this seems like a relatively easy approach, finding and hooking common attack points requires a wide knowledge of said attack points and case-specific approaches. [22]

Hook detection requires optimizing, as an anti-malware implementation could be made so that it hooks everything, therefore detecting all hooks made, consequently detecting all malware attempting to hook. Even after optimizing, hook detection alone isn't enough, as legitimate uses of hooking are common. After finding a hook, anti-malware must be able to decide whether or not the program behind the hook is malicious or not. Therefore hook detection must use other methods of detection sequentially. [22]

2.5 Virtualization

Sandboxing, emulation and virtualization mean virtually the same thing in the context of malware detection, though virtualization is the underlying technology used for both sandboxing and emulation in modern malware analysis. Their goal is to avoid malware from being executed on a machine and optionally to analyze malware further. Avoiding code execution on a machine is implemented by using virtual environments i.e. virtual machines.

Virtualization is implemented by having a hypervisor manage the physical resources of a machine between virtual environments running on the machine [4, 10]. As the hypervisor manages the resources for the operating system (OS), resource monitoring can be implemented in a way that the OS cannot interfere and therefore malware operating in the OS can't hide its presence in said resources. Therefore, even if the OS is compromised, the malware can be detected and analyzed via its used resources and executed machine instructions. [22, 23]

3. MALICIOUS SOFTWARE

In this chapter, types of malware, including potentially malicious software, and their evasion techniques are defined and introduced. The chapter is divided into three parts, the first part defines the types and the second introduces some of the stealth techniques used by malware. The last part goes through hooking techniques with more detail than other techniques are gone through while maintaining a non-comprehensive level of detail.

3.1 Types of malicious programs

A few types of malicious programs are introduced and explained briefly in this section. It's good to note that malware is a hypernym, which includes viruses, worms, trojans and more. [4] While modern malware may act like multiple types such that it could, for example, act like both a trojan, a virus and a worm at various times of its propagation and operation [22]. Distinguishing malware types is beneficial when malware is the subject, but it should be noted that categorizing malware is difficult due to the ambiguous definitions and complex nature of malware.

3.1.1 Riskware

While riskware or grayware isn't defined by either of the relevant dictionaries [4, 11], it is defined by notable actors in the field of cybersecurity [12, 15, 18]. Riskware is software that may not be inherently harmful but can be depending on the use and context and therefore isn't malware.

In the least threatening case, riskware can be merely annoying or hinder the use of a machine [12], but as stated in 2.1, poorly implemented anti-malware can be exploited to attack a system, which therefore could be qualified as riskware while being capable of causing notable damage. Since riskware isn't inherently malicious, riskware doesn't completely pertain to this chapter, but it is worth noting due to its malicious capabilities.

3.1.2 Viruses

Viruses are unwanted programs that perform malicious tasks. The severity of the task can vary but is unwanted nonetheless making a distinct difference between viruses and riskware. [11] Viruses spread in a host machine by replicating themselves onto other programs or parts of the filesystem. There are many variants of viruses for different purposes with different capabilities. Variants have different methods of infection, evasion and acting maliciously. [4]

The way viruses replicate depends on the virus type. They may infect the file system, boot sector or both and depending on what they infect, they are classified as file infector, boot sector virus or multipartite virus respectively. The types may be divided further based on their propagation model, but they are outside the scope of this paper. Upon appropriate conditions, the virus searches for a target within its scope and infects it. To explain infecting simply, as long as the virus doesn't reside solely in the memory, it infects files by overwriting, appending, prepending or replacing them. Note that overwriting and replacing the original file or program differ so that when replacing, the original file is renamed or the virus uses a file extension that is preferred over the original file's extension in the execution hierarchy. The infection can then be repeated to spread further in the system. [5]

3.1.3 Worms

As pointed out by [14], defining worms proves to be difficult, as different notable sources [13, 16, 17] have differing definitions for worms. These differences are broad, as [16] goes to define worm as a type of Trojan, while other sources define it as malware [13, 17] or a virus-like program [4], but even those defining it as malware state it may have Trojan functionalities [13]. The common features seem to be spreading through a network, requiring little or no human interaction and being a standalone program such that it doesn't need a host file or program. [13, 14, 16, 17]

The actual method of spreading may differ for worms, such that the simplest methods rely on user interaction to infect a machine and the most advanced simply exploit vulnerabilities in a machine on the internet. The former method could be implemented by sending the worm out as an attachment or a downloadable link in a message such as an e-mail or an instant message. [5]

3.1.4 Trojans

Similarly to worms, the definitions for Trojan by different notable sources are ambiguous [5] and what malicious acts, if any, they commit isn't agreed on. However, it is agreed that, as the name suggests, they are programs that act maliciously under a legitimate pretense. [5, 11, 14]

While Remote Access Trojan (RAT) has a similar name, it isn't a Trojan in the same sense as a Trojan. Other names used for remote access trojan include remote administrating trojan, remote administrating tool and remote access tool. [5]

3.1.5 Rootkits

Rootkits are a collection of software components used to create and maintain stealth routes and features such as backdoors, hiding files and processes and masking events [4]. While other components for malicious intent may be included, they are not inherent for rootkits. Rootkits are not necessarily malicious and can be used for benevolent purposes, which makes detecting malicious rootkits harder. Anti-virus implementations are a notable use for benevolent rootkits. [22]

Some of the common methods rootkits can use to hide their presence are hooking and process injection. Hooking is an OS feature, which is used for both benevolent and malicious purposes making it ideal for malware since the anti-malware present on the host machine has to identify the legitimacy of the hook. For example, monitoring, hot patching and debugging are examples of benevolent purposes hooking is used for. [7, 22] Process injection is, similarly to hooking, a legitimate functionality provided by the OS and therefore may be difficult to discern from benign activity. [19] Both will be further discussed in 3.2.4.

Other methods available for rootkits are replacing or modifying system files — known as static patching — and operating in the Basic Input/Output System(BIOS), virtualization layer or hardware. However, the formermost is easily detected by modern anti-malware and even the OS itself and the latter has yet to be found and only exists as a proof of concept for now. [22] Additionally anything beyond the OS, i.e. BIOS, virtualization layer or hardware is difficult to gain access into and control over, as to do so the malware has to either exploit a vulnerability in them or externally gain access to them instead of the OS. Maintaining these parts is also more difficult, as to avoid causing suspicion, the malware must still stay hidden, but in doing so it may cause issues visible to the user or system administrator such as crashing or other erroneous behavior, as the system relies on these lower level parts being intact. [22]

3.2 Evasion techniques

In this section, modern techniques malware use to avoid detection are introduced. These techniques define stealth malware as their presence separates malware and stealth malware. Some techniques are justified by referring to the detection method they aim to elude. It should be noted that these techniques can be used in combination to further fade out traces left by malware.

3.2.1 Obfuscation

The primary goal of obfuscation is to prevent detection by modifying the code or signature of the malware without changing the functionality, which causes pattern-based detection techniques such as signature scanning to fail. Obfuscation may be implemented by adding redundant code, changing the order of execution, code encryption or code mutation. [20, 22] Obfuscation may additionally make reverse engineering more difficult.

Encryption

Encryption is done on the malware payload by running it through an encryption engine. The payload is then ready to be distributed with the decryption engine, or loader, and once it has spread to a machine, the payload is decrypted for execution. In the encrypted state, the payload isn't runnable and therefore not analyzable. However, this doesn't mean it isn't detectable, as both the encrypted payload and the loader still have signatures. [8, 22]

The static payload signature can be avoided by implementing a cryptographic re-randomization

algorithm [8], which allows the malware to appear to be different on different levels of propagation. Encryption without the combination of other stealth techniques, even with re-randomization, leaves the loader vulnerable to detection. [20, 22]

Code mutation

One way the aforementioned loader could be hidden is by adding code mutation to it [22]. Code mutation is achieved with polymorphic or metamorphic code, the former of which is not to be confused with code that is polymorphic in programming language theory. Polymorphic code in malware means that the program has a polymorphic engine that mutates the decryptor without changing the functionality. In practice, this could be changing the order of execution and variable values when possible. With only the signature and order of execution changing, polymorphic malware is still susceptible to being detected by behavioral and heuristic analysis, and due to the fact the size, location of the code and the underlying code don't change allowing some forms of signatures to be generated. [20, 22, 23]

To resolve the shortcomings of polymorphic code, metamorphic code improves further on the idea of code mutation by not only changing the decryptor but the entire malware. In addition to changing the order of execution, metamorphic code also changes the size of the code by either changing the way some functionalities are implemented or adding redundant code. [22] Unlike polymorphic code, metamorphic code changes the code completely by disassembling, transforming and reassembling the code. This results in well-implemented metamorphic malware having similar behavior but no common patterns between generations. [20] However, this mutation would likely have to happen outside of the scope of an anti-malware program, as the disassembled and analyzed versions of code, which could be relatively similar, must reside somewhere on a machine during mutation and therefore could be detected if it was to be scanned by anti-malware.

3.2.2 Anti-emulation and targeting mechanisms

Anti-emulation and targeting mechanisms both change the way malware acts in specific environments. Anti-emulation is developed to withstand sandboxing, which was introduced in 2.5, and it tries to figure out whether or not the malware is in an emulated environment. When emulation is detected, the malicious code isn't executed to avoid detection. Targeting mechanics on the other hand restrict the execution of malicious code and spreading to certain environments, which may be determined by the machine, features, files or other variables.

In a sense anti-emulation is a targeting mechanism, but targeting mechanisms can reduce the number of environments where the malware starts acting even further. [22] A notable implementation of targeting mechanisms in combination with encryption is Gauss [21], which uses encryption keys derived from environmental variables to decrypt modules on only certain machines.

3.2.3 Virtualization and hardware

While the other methods discussed in this thesis are applicable and desirable when the malware is operating in the OS, rootkits can theoretically operate on a lower level such as the virtualization layer, BIOS or hardware. These options require customized software and or hardware components, which makes them difficult to produce. However, operating on a lower level than the OS has the benefit of not being OS-dependent as the OS runs on top of the malicious part. [22]

Virtualization-based rootkits work by having the rootkit become the host OS in a virtualized environment and making the OS in use the guest OS. Since the guest OS cannot access the memory or the processes of the host OS, malicious programs running on it cannot be detected. The change from host to guest OS has to be hidden from the OS in use, as this would cause a detection. Depending on whether the rootkit is software- or hardware-based, the way the rootkit can be detected changes so that the latter can be detected by the non-virtualizable instructions and the former requires either access to the physical memory or that the timing discrepancies they introduce are noticed. [7]

3.2.4 Process injection and hooking

Hooking is introduced here but different hooking techniques will be further explained in 3.3. Hooking means intercepting a function call — usually a system API call — and redirecting it so that the target code is run along with the regular code. This means to somehow trick the system at runtime to either call the malicious function or insert the call of the malicious function into the called function. Hooking is a convenient approach for malware, as hooking is also used for benevolent purposes, as mentioned in 3.1.5 and it allows operating within other processes instead of having to create a separate process for the malware that would be trivial to discover. [6, 22]

Process injection as the name suggests means to inject something, into a process to either gain access to the process or exploit its privileges. [19] Process injection is related to hooking in a sense as most hooking techniques require a part of a process to be altered which can be achieved using process injection. [22]

Although process injection encompasses much more than just dynamic link library (DLL) injection, DLL injection is highlighted here as some hooking methods need it to alter the target of the hook. DLL injection is specific to the Windows OS as DLLs are exclusively Windows' way of sharing code and data between programs [9]. As the intended function of DLLs is to have a process load them, discerning whether or not an injection is benign may prove to be difficult. The actual injection is done by forcing the process to load the wanted DLL which can be achieved in several ways. Once injected the DLL code is run in the context of the host process giving it the privileges of the host process. Similarly to hooking, this method doesn't require the malware to operate with a process of its own but requires the malicious DLL to exist within the system to be injected into some other process. [22]

3.2.5 Direct kernel object manipulation

Direct kernel object manipulation (DKOM) is another Windows-specific method stealth malware uses where instead of intercepting program execution by running malicious code, dynamic kernel data structures are modified maliciously. This technique requires the developer to have a good understanding of the kernel as corrupting data in the kernel may lead to unexpected behavior and crashes. [7, 22]

As DKOM attacks are limited to modifying data structures the applications are few but because when properly implemented they are difficult to detect their effectiveness shouldn't be underestimated. Even though applications are limited, DKOM attacks are capable of process hiding, altering pseudorandom number generators and firewalls, and spoofing memory views. [22]

3.3 Hooking techniques

In this section, user and kernel spaces are introduced and then hooking methods to both spaces. As the majority of desktop computers in use are running on the Windows OS [24], all methods under this section are only applicable on the Windows OS.

There are hooking methods that aren't mentioned here due to limited information from sources used and or limited usability. Some examples of hooking methods not covered here are hybrid hooking, SYSENTER hooking, IDT hooking and IRP hooking.

3.3.1 User and kernel space

In the Windows OS, there are two processor modes to separate applications and the OS, user- and kernel-mode which are run in ring 3 and ring 0 modes respectively. Rings refer to processor protection rings which range from 0 through 3 for x86 processors, where 0 is the highest privilege and 3 the lowest. However, for this thesis, understanding what user- and kernel-mode are suffices. [6, 25] Figure 3.1 is a simplified representation of the entire OS architecture with protection rings on the left-hand side of each level. However, it should be noted that drivers in the Windows OS do not operate on levels 1 and 2 but rather on level 0 alongside the kernel [25].

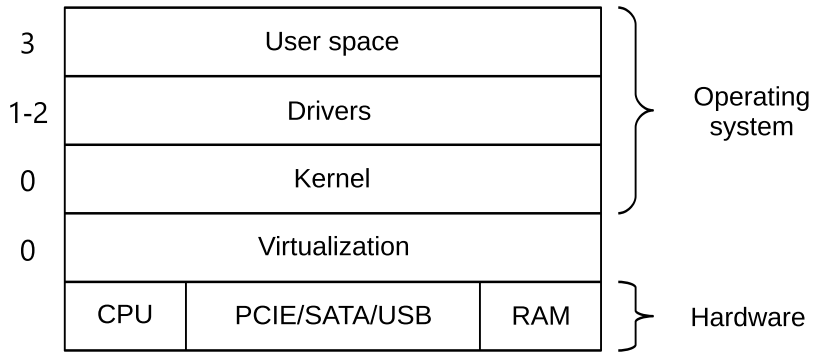

Figure 3.1. *Simplified operating system architecture*

As the idea of separate spaces suggests, user-mode applications cannot access kernel-mode services or memory directly. Instead, access is provided via Windows' DLLs. [25] This separation makes hooking into kernel-mode processes desirable for malware, as anti-malware operating in user mode cannot scan the kernel-mode memory. However, hooking to the kernel mode and maintaining the malware in it introduces difficulties as the system relies on the kernel being intact and even slight errors may cause system instability which in turn may raise suspicion from the user leading to detection. [22]

3.3.2 Import address table hooking

Import address table (IAT) hooking is a user-mode hooking technique where the IAT address is overwritten to point to a malicious function in the memory. [22] The IAT alongside the export address table (EAT) are parts of the PE Format supported by Windows that are populated with the memory addresses of exported and imported functions respectively. The latter however is an extended DLL characteristic and is thus only found in DLLs and the former can be found in all PE format files when external functions are imported. [3]

In practice, IAT hooking would be implemented so that the malicious code is in a malicious DLL which has an IAT for at least the actual function(s), EAT for the malicious function(s) and is then hooked onto a user executable by exploiting the Windows API. [7] Figures 3.2 and 3.3 represent program flows without and with an IAT hook in place respectively. In 3.3 the malicious code has been hooked to the first function in the executables IAT and the new pointer in the IAT points to the malicious code. The final call in the malicious code is the actual function which is called to return the requested value to avoid causing suspicion but this is by no means required for a successful hook.

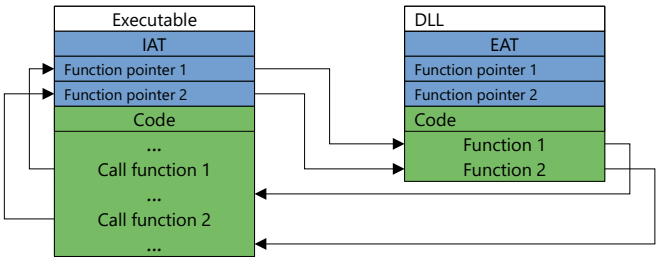

Figure 3.2. *Program flow without IAT hooking, adapted from figure in [22].*

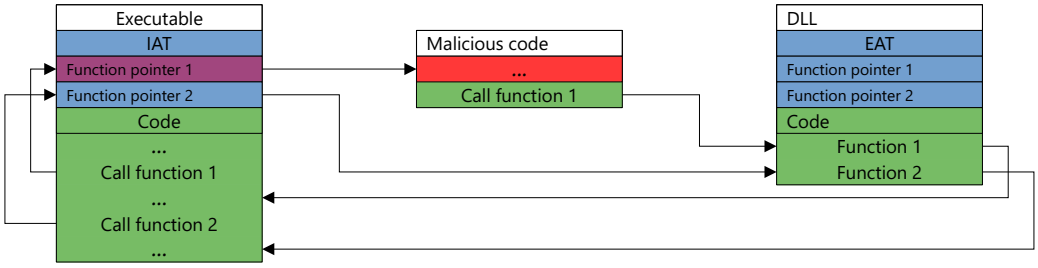

Figure 3.3. *Program flow with IAT hook in place, adapted from figures in [22] and [7].*

The EAT is visible in 3.2 and 3.3, but isn't connected anywhere because it isn't actively used but the addresses in it are copied to the executables IAT when the DLL is loaded. The DLL could however be loaded during load time or when the DLL is needed at runtime of the executable which makes IAT hooking more difficult as the DLL has to be loaded for IAT hooking to be successful. IAT hooking also has to be engineered for a specific call which in most cases leads to OS API calls being the only practical target of IAT hooks. [22]

3.3.3 Inline function patching

Inline function patching — also known as detouring — is another common hooking technique which instead of changing the address of a function, like IAT hooking, modifies the underlying code of a function in memory with an unconditional jump instruction to the malicious function. Unlike IAT hooking, detouring isn't limited to the user mode and can be implemented in kernel space. Whereas to detect an IAT hook, checking if all IAT addresses reside in the corresponding DLLs' memory areas sufficed, detouring requires a more comprehensive analysis of the code. [7, 22]

Though detouring requires more to be detected than IAT hooking, there are more subtle difficulties as erroneous hooking may lead to unexpected behavior in the original code due to missing instructions. Additionally, an arbitrary placement of the unconditional jump instruction could lead to it being at a part of the code where it's executed at unwanted frequencies. [22] However, detouring, like many other methods exploited by malware, isn't inherently malicious but has legitimate uses such as hot patching where the execution of a function is diverted to an updated version of itself. Thanks to this, compilers might leave redundant bytes at the beginning of a function, and distinguishing benevolent and malevolent detours is more difficult. [22]

In practice, malicious detouring is done by editing some — usually the first few — bytes of a function in memory to unconditionally jump to the malicious function which in turn performs its operations, the possibly replaced instructions, and then jumps back to the original function but after the replaced bytes. By jumping back, the execution seems untouched removing one reason for suspicion. [7, 22]

3.3.4 SSDT hooking

System service descriptor table (SSDT) is a table similar to IAT and contains function addresses that the system service dispatcher relies on, but it's located in Windows' kernel memory making SSDT hooking strictly a kernel-mode hooking technique. Due to being a kernel-mode hooking technique, the malware has essentially no restrictions priviledgewise but careless operation will still result in unexpected or erroneous behavior. [7, 22]

Besides the positive aspects of SSDT hooking, residing in the kernel memory adds the requirement of being able to write into kernel memory making the attack more difficult, and to further complicate writing, the SSDT memory area is set to be read-only by Windows. The read-only restriction can however be bypassed. [22]

SSDT hooking, unlike IAT hooking, provides a system-wide hook as all system calls that require anything from the kernel go through the system service dispatcher therefore accessing the SSDT [6, 7]. As a result of this, if the malware causes errors or unexpected behavior it will be visible on a wider scale as well, requiring the malware developer to be cautious when using an SSDT hook. [22]

4. CONCLUSION

This thesis overviewed the methods used by both malware and anti-malware, with emphasis on rootkit and stealth malware methods of operating within the OS. Even with the limited scope available and only addressing the simplest anti-malware techniques it is obvious that current techniques do not rely on detecting new variants but instead rely on recognizing previously known malware. In addition to techniques used being slightly lacking, it is discovered that malware definitions are often ambiguous and not agreed upon among notable actors within the field even though they are used by each to describe malware. However, manual analysis with tools like virtualization does allow the detection of new malware variants and machine learning is being utilized to attempt to do so without manual analysis.

The term stealth malware mainly implies the use of stealth techniques and while only some techniques have been introduced in this thesis, it is obvious that primarily they aim to either obfuscate the malware or operate outside of the anti-malware scope of operation. The former can be implemented in many ways with the simplest — and least effective — just changing variable and function names and the most difficult creating a metamorphic program where a perfect implementation would have no common common patterns while retaining functionality. The latter is achieved by hiding the malware in a benign process, operating in a part of the OS the anti-malware has no access to or outside of the OS in the virtualization or hardware layers.

While currently methods that apply within the OS are essential, the future might be different and virtualization- or hardware-based malware may become more frequent. The danger they pose is great as, if they can be deployed before the initialization of a system and its anti-malware components, and they are implemented correctly there's practically no reason to suspect their presence and therefore no reason to scan for them.

As malware and their counterparts attempt to operate discreetly there's no guarantee that the sources used or even the most recent sources would include the most recent methods each party uses. Additionally, the sources used do not mention recent developments in computer learning due to being released before them. Lastly, this thesis contains only a portion of techniques used by both anti-malware and malware and most techniques are only applicable on the Windows OS. Taking these factors into account, there are many perspectives on stealth malware and its techniques this thesis doesn't gloss over nor could it.

REFERENCES

[1] S. R. Aslan Omer. A Comprehensive Review on Malware Detection Approaches. eng. *IEEE access* 8 (2020), 6249–6271. ISSN: 2169-3536.

[2] Z. Bazrafshan, H. Hashemi, S. M. H. Fard and A. Hamzeh. A survey on heuristic malware detection techniques. eng. *The 5th Conference on Information and Knowledge Technology*. IEEE, 2013, 113–120. ISBN: 1467364908.

[3] K. Bridge. *PE format - win32 apps*. Accessed: 13.12.2023. Mar. 2023. URL: https://learn.microsoft.com/en-us/windows/win32/debug/pe-format#import-address-table.

[4] A. Butterfield, G. E. Ngondi and A. Kerr. *A Dictionary of Computer Science*. Oxford University Press, 2016. ISBN: 9780191768125. DOI: 10.1093/acref/9780199688975.001.0001. URL: https://www.oxfordreference.com/view/10.1093/acref/9780199688975.001.0001/acref-9780199688975.

[5] C. Elisan. Malware, rootkits & botnets : a beginner's guide. eng. 1st edition. New York, 2013. Chap. 2. ISBN: 1-283-57893-X.

[6] C. Elisan. Malware, rootkits & botnets : a beginner's guide. eng. 1st edition. New York, 2013. Chap. 3. ISBN: 1-283-57893-X.

[7] S. Eresheim, R. Luh and S. Schrittwieser. The evolution of process hiding techniques in malware - Current threats and possible countermeasures. eng. *Journal of information processing (Tokyo)* 25 (2017), 866–874. ISSN: 0387-5806.

[8] H. Galteland and K. Gjøsteen. Malware, Encryption, and Rerandomization - Everything is Under Attack. eng. (2017). ISSN: 0302-9743.

[9] D. Han. *Dynamic Link Library (DLL) - windows client*. Accessed: 13.12.2023. Apr. 2023. URL: https://learn.microsoft.com/en-us/troubleshoot/windows-client/deployment/dynamic-link-library.

[10] IBM. *What is Virtualization? | IBM — ibm.com*. Accessed: 05.11.2023. URL: https://www.ibm.com/topics/virtualization (visited on 11/05/2023).

[11] D. Ince. *A Dictionary of the Internet*. Oxford University Press, 2019. ISBN: 9780191884276. URL: https://www.oxfordreference.com/view/10.1093/acref/9780191884276.001.0001/acref-9780191884276.

[12] Kaspersky. *What is Riskware?* Accessed: 05.11.2023. URL: https://usa.kaspersky.com/resource-center/threats/riskware (visited on 11/05/2023).

[13] Kaspersky. *What's the Difference between a Virus and a Worm?* Accessed: 26.11.2023. URL: https://www.kaspersky.com/resource-center/threats/computer-viruses-vs-worms (visited on 11/26/2023).

[14] D. Kienzle and M. Elder. Recent worms: a survey and trends. eng. *Proceedings of the 2003 ACM workshop on rapid malcode*. ACM, 2003, 1–10. ISBN: 1581137850.

[15] Malwarebytes. *Greyware*. Accessed: 05.11.2023. URL: https://www.malwarebytes.com/glossary/greyware (visited on 11/05/2023).

[16] Malwarebytes. *Worm*. Accessed: 26.11.2023. URL: https://www.malwarebytes.com/blog/threats/worm (visited on 11/26/2023).

[17] Norton. *What is a computer worm, and how does it work?* Accessed: 26.11.2023. URL: https://us.norton.com/blog/malware/what-is-a-computer-worm (visited on 11/26/2023).

[18] Norton. *What is Grayware?* Accessed: 05.11.2023. URL: https://in.norton.com/blog/malware/what-is-grayware (visited on 11/05/2023).

[19] A. Pingios, C. Beek and R. Becwar. *Process injection*. Accessed: 14.12.2023. Mar. 2023. URL: https://attack.mitre.org/versions/v14/techniques/T1055/.

[20] B. B. Rad, M. Masrom and S. Ibrahim. Camouflage in malware: from encryption to metamorphism. *International Journal of Computer Science and Network Security* 12.8 (2012), 74–83.

[21] K. L. G. Research and A. Team. *Gauss: Abnormal Distribution*. Accessed: 05.11.2023. URL: https://securelist.com/gauss-abnormal-distribution/36620/ (visited on 11/05/2023).

[22] E. M. Rudd, A. Rozsa, M. Gunther and T. E. Boult. A Survey of Stealth Malware Attacks, Mitigation Measures, and Steps Toward Autonomous Open World Solutions. eng. *IEEE Communications surveys and tutorials* 19.2 (2017), 1145–1172. ISSN: 1553-877X.

[23] D. Samociuk. Antivirus Evasion Methods in Modern Operating Systems. eng. *Applied sciences* 13.8 (2023), 5083–. ISSN: 2076-3417.

[24] StCo. *Desktop Operating System Market Share Worldwide*. Accessed: 02.10.2023. URL: https://gs.statcounter.com/os-market-share/desktop/worldwide/ (visited on 10/02/2023).

[25] P. Yosifovich, A. Ionescu and D. A. Solomon. *Windows Internals, Part 1: System architecture, processes, threads, memory management, and more*. eng. Seventh edition. Vol. Book 1. Windows internals ; Part 1. Pearson Education, 2017. ISBN: 9780735684188.