On the (In)security of Behavioral-based Dynamic Anti-Malware Techniques

by

Erkan Ersan
B.Sc., University of Kocaeli, 1998

A Thesis Submitted in Partial Fulfillment of the
Requirements for the Degree of

MASTER OF SCIENCE

in the Department of Computer Science

On the (In)security of Behavioral-based Dynamic Anti-Malware Techniques

by

Erkan Ersan
B.Sc., University of Kocaeli, 1998

Supervisory Committee

---

Prof. Bruce Kapron, Co-Supervisor
(Department of Computer Science, University of Victoria)

---

Dr. Lior Malka, Co-Supervisor
(Faculty of Graduate Studies, University of Victoria)

**Supervisory Committee**

---

Prof. Bruce Kapron, Co-Supervisor
(Department of Computer Science, University of Victoria)

---

Dr. Lior Malka, Co-Supervisor
(Faculty of Graduate Studies, University of Victoria)

## ABSTRACT

The Internet has become the primary vector for the delivery of malicious code in cyber attacks, and malware has rapidly become a pervasive critical threat. Anti-malware products offer effective protection from malware threats for servers and end-point devices using a variety of techniques. Advanced enterprise-level anti-malware products rely on state-of-art behavioral-based detection algorithms, in addition to traditional signature-based mechanisms. These dynamic detection techniques have been around for more than a decade and in response hackers have developed methods to evade them. However, currently known bypass methods require intensive manual labor. Moreover, this manual work has to be repeated whenever a parameter of the environment (such as the payload, operating system, Antivirus version, etc) changes, making these methods impractical. This may lead to the belief that dynamic techniques provide a good deterrence, and hence good protection.

In this thesis we evaluate dynamic techniques. Specifically, we build tools to implement generic *unhooking* and *funneling*, and using these tools we show how dynamic techniques can be bypassed with considerably less effort than by fully manual methods. We also extend the repertoire of existing bypass methods and introduce a new malicious function call technique which exploits detection techniques that monitor a limited collection of critical system functions, as well as a method for bypassing guard-page protections. We demonstrate the effectiveness of all our techniques by conducting attacks against two enterprise antivirus products. Our results lead us to conclude that that dynamic techniques do not provide sufficient protection.

# Contents

# List of Tables

# List of Figures

# ACKNOWLEDGEMENTS

I would like to thank:

- my supervisors Prof. Bruce M. Kapron and Dr. Lior Malka for their endless support, inspiring motivation, and flawless mentoring throughout this degree;
- the Hawkins family, Gill and Gordon Hawkins as well as Jillian Zaruk for their big, warm hearts and being a wonderful family for me in Canada;
- Muharrem Şar, Yağmur Akbulut, and Dr. Yagız Onat Yazır for their continuous encouragement, trust and friendly advice;

My special thanks go to my parents, my family, and my girlfriend for all their love and limitless trust.

The work in this thesis was completed as part of a Collaborative Research Agreement between Intel Corporation and the University of Victoria titled *"Automated Antivirus Evaluations via Malware Mutations"*, and was fully funded by Intel. I would like to thank Intel for their generous support.

*Life is a state of mind.*
Being There (1979)

# DEDICATION

*To my lovely grandmother.*

# Chapter 1

# Introduction

The Internet has reshaped people's lifestyles worldwide since the 1980s and boosted the digitization of everyday life. Although started by a small number of academics for research purposes, today the Internet widely connects approximately half of the world's population to each other without any geographical and cultural boundaries [51]. Almost all countries from every world region have been integrated into this tremendous cyber world, which continues its rapid growth and evolution. While computers are still the dominant means for connecting to the Internet, the popularity of smartphone and tablet usage continues to increase globally [175]. People's daily lives have become closely intertwined with this technology. Technological innovations in computers and especially mobile devices have lead to a dramatic increase of people's online time. The cutting edge of technologies, including smart wearable devices and the Internet of Things (IoT), engage innumerable promising capabilities for pervasive connectivity. It is estimated that every day, over 3.6 billion people all around the world employ the Internet for a very broad range of purposes, including communication, education, entertainment, and commerce.

The Internet has developed into a big "cyber-metropolis"; therefore, while it provides many conveniences, it has developed problems, including security issues, similarly to any large and densely populated area. The Internet has become a focal point for individuals, commercial enterprises, governments, as well as international criminal enterprises and terrorist groups. Regrettably, it is now the case that citizens of the Internet could potentially encounter innumerable serious cyber crimes from simple theft to organizational fraud [46, 52, 37]. Cyber-thieves are able to steal sensitive private and corporate information, including identities, health histories, bank accounts, trade secrets, and cutting-edge research [47]. By exploiting vulnerable servers and

computers on the Internet, cyber-lawbreakers are able to marshall malicious botnets in order to anonymously conduct massive attacks on sensitive targets and to distribute malicious software. Services running on critical corporate and government infrastructure are targeted and can be disrupted and disabled by cyber-saboteurs, resulting in losses of productivity, reputation and profits. Cyber-threats usually have no nationality because criminal individuals and illegal organizations around the world often work together in order to boost their effectiveness as well as their revenue. Therefore, most countries cooperate with other countries, providing worldwide intelligence to each other in order to efficiently tackle cybercrime, such as financial crime and terrorism. However, in most cyber spying cases targeting a particular country [169], the aggressors are likely supported or hired by agencies or actors from other countries. Cyber-espionage and cyber-sabotage activities are conducted by personal, economical, political, and military opponents in an ongoing cyberwar [97, 166]. Thus, the threat of malignant attacks in the cyberworld impacts society at every level: individual, corporate, and national.

## 1.1 Motivation and Problem Description

Most attacks in the cyberworld involve the installation of malicious code on victim computers which allow attackers to control these target systems for various ends. Malware has rapidly become a pervasive critical threat on the Internet. Every year, it is estimated that five hundred million new unique pieces of malware are developed by cyber criminals. In 2015, the number of newly discovered malware variants increased by 36 percent over the year before. Similarly, the web-based attacks per day doubled in 2015 and reached 1.1 million, up 117 percent from the previous year, 2014 [156]. The spending on information security was estimated 81.6 billion U.S. dollars globally in 2016 [45]. Against evolving cyber threats, modern anti-malware products powered with intrusion prevention and detection techniques are widely used on computer system. They provide efficient and effective endpoint protection to stop malware-initiated attacks.

Anti-malware products most often use signature-based and behavioral-based detection techniques to identify malicious threats. In signature-based detection, the patterns of known malware are searched in suspicious files and network packets, using signatures stored in predefined databases. Even though signature-based anti-malware methods provide a fast detection ability with lower false positive rate and less compute

resources usage, they are only able to detect previously known malware and usually ineffective against new unknown malware. Due to the signature creation process, there is an unsafe time lag between discovery of a new piece of malware and availability of protection against it. However, behavioral-based detection identifies both known and unknown threats using behavioral patterns. It offers the best protection against attacks exploiting known and zero-day vulnerabilities. Thus, the-state-of-art behavioral-based detection algorithms are implemented in advanced enterprise-level anti-malware products, in addition to signature-based techniques. Since a significant number of individuals, organizations, and governments relies on anti-malware products for protection, the effectiveness of their prevention technologies is highly critical.

## 1.2 Scope and Contribution

The goal of this thesis is to assess the efficacy of behavioral-based dynamic malware mitigation and the potential pitfalls of user-level anti-malware techniques. In this context, we have implemented two evasion methods, *unhooking* and *funneling*, and evaluated the performance of two enterprise level anti-malware products in the presence of these methods.

Our methods directly target anti-malware components in the virtual memory address spaces of processes in order to eliminate anti-malware protections and prevent them from detecting malicious shellcode. The principal difference between the *unhooking* and *funneling* methods lies in the procedure for disabling the anti-malware functionalities. In order to deactivate anti-malware, the unhooking method patches the code sections of modules belonging to the operating system, whereas the funneling method only alters the anti-malware's main control code in memory. By disabling all protections before executing a payload, both evasion methods allow malware to use any payload without any further code modifications.

Due to recent improvements in computer protections against malicious cyber attacks, there has been a dramatically increase in interest in behavioral-based anti-malware techniques in academia as well as industry. Our research differs from previously proposed research as, instead of considering methods for bypassing detection algorithms, we concentrate on deactivating anti-malware products by disabling protection mechanisms in user space, thereby enabling the use of any regular payload.

To the best of our knowledge, the automated approaches of *unhooking* and *funneling* methods, the new evasion technique against guard-page-based protections and, the new malicious function call emulation technique provided in this thesis have not been publicly employed in academia and the exploitation domain. In this research, we focus on behavioral-based dynamic anti-malware techniques; the domain of signature-based techniques is beyond the scope of this study.

### 1.2.1 Summary of Contribution

In summary, this research makes the following contributions:

- We have implemented two different evasion methods, *unhooking* and *funneling*, in order to evaluate the performance of behavioral-based dynamic anti-malware techniques in user space, and have tested the enhanced malware variants against two enterprise-level anti-malware products.
- We provide an in-depth analysis of two enterprise level anti-malware products which dynamically detect malicious threats using behavioral-based mitigation techniques.
- We introduce a new evasion technique used for invoking critical functions monitored by anti-malware using inline hooking, which also presents the weakness of monitoring a limited number of system functions that are considered critical.
- We introduce a new evasion technique against guard-page-based protections used for monitoring critical regions in memory.
- We have implemented a new just-in-time debugging tool using Intel's Pin framework, which automatically discovers critical functions hooked by anti-malware in memory, and provides their offsets as well as the original and altered opcodes in hex and Assembly.
- We have demonstrated that it is possible to alter publicly known detectable malware and improve its functionality so that it completely bypasses the behavioral detection techniques employed by two well-known enterprise-level anti-malware products.

## 1.3 Thesis Organization

Following this introduction, the rest of this thesis proceeds as follows. In Chapter 2, we provide background information about related work, cyber-security, malware which is the most serious threat in cyber-life, as well as the common vulnerabilities used by malware and protection techniques against these security holes. Chapter 3 introduces the main working principals of anti-malware evasion methods and describes the development framework used in the thesis to implement new malicious variants. It also covers the behavioral-based protection methods implemented in the two tested anti-malware products, and discusses several evasion techniques against them as well as explaining the *unhooking* and *funneling* methods. In Chapter 4, the implementation details of these two evasion methods are carefully discussed. This chapter aims to cover possible bypassing techniques against behavioral-based protection used by enterprise level products, as well as providing comprehensive information about the malware variants developed in this study, and the vulnerabilities exploited by these variants to conduct a malicious attack. In Chapter 5, we outline the results and practical findings of this thesis, and future work planned to build on the research.

# Chapter 2

# Background

## 2.1 Related Work

Malware is malicious software that is able to compromise computer systems, by exploiting several software vulnerabilities, including stack- and heap-based buffer overflows, integer overflow, and format string vulnerabilities [100, 105, 13, 118].

The problem domain of malware detection has been investigated under two main categories where the method of malware identification relies on either a distinctive signature, or a behavioral pattern-matching. In short these methods are termed signature-based and behavioral-based. In malware detection, static and dynamic analysis are employed to gather information about an unknown application in order to determine whether it is malicious.

### 2.1.1 Signature-based and Behavioral-based Detection

Signature-based algorithms offer efficient detection as well as rare false positive detection and as a result they have been widely implemented in most consumer and enterprise-level anti-malware products. However, due to the dependence of detection on a distinctive sequence of bytes, called a signature, this method is sensitive to differences in code appearance. Naturally, zero-day (never seen before) malware is also undetectable by signature-based methods. Several attack techniques have been invented by malware developers to successfully bypass signature-based detection, such as packing, encryption, morphing, and code obfuscation [158]. Signature-based detection is susceptible to morphing and obfuscation of code [87] so self-modifying malware [173] produces new malicious variants altering appearances with every infection yet

while preserving semantic to evade detection in signature-based attacks. Signature-based anti-malware products may only identify self-modifying malware using constant parts in its code, such as a constant morphing engine in polymorphic malware [62, 23].

In response to zero-day and signature-evading attack techniques, various [22, 59, 64] countermeasures based on behavior characterization have been developed by security researchers. Using behavioral-patterns, behavioral-based detection identifies malicious code, while not requiring static malware signatures.

In the both categories, malware detection can be performed using static and dynamic features in order to determine whether an application is malicious or benign. The fundamental difference between static and dynamic features depends on where features are collected: offline or at runtime. The possible features used for behavioral-based malware detection include:

- program control flow integrity [20, 101, 43];
- byte sequences (n-grams);
- printable and non-printable string information (PSI) embedded in software;
- dynamic module information (DMI) in executables;
- opcode sequences (OS);
- function features (FF) in software, such as name and length;
- runtime system API calls (SAC) sequences and graphs.

### 2.1.2 Behavioral-based Detection based on Control Flow Integrity (CFI)

One of the most effective methods in behavioral-based detection is based on Control Flow Integrity (CFI) [2, 3]. CFI dynamically detects malicious activities in systems by monitoring the behaviors of applications at runtime, and so it does not require any access to software source code for detection. A method proposed in [2], called full-grained CFI, uses control-flow graphs (CFG). A CFG includes every legitimate execution path of an application, similar to whilelists. In order to track every branching event during execution, CFI labels each building code block. Then, CFI checks whether its label is valid when a branching event occurs, such as with call, jump, and return instructions. A control flow path is only permitted if it is on the CFG. Otherwise, it will be a security violation for CFI detection. Thus, it is not only effective against binary malware but also runtime attacks implementing code-reuse techniques [34, 138], such as web-based malware.

Specifically, in web-based attacks, malware gains control of vulnerable applications such as browsers, and it manipulates control flow in order to execute its malicious payload injected in memory. This will change the usual behavior of the vulnerable application, which may be detected by CFI-based behavioral detection.

Due to high performance overheads, averaging 21% [2], the implementation of fine-grained CFI may be impractical for most real-world scenarios. Therefore, a number of CFI approaches have applied several relaxations in order to solve performance issues, resulting in a coarse-grained CFI approach. An example is that in fine-grained CFI, a function return is checked if it returns to its original caller functions [2], whereas in coarse-grained CFI, a function return is checked if it returns to an address storing a call-preceded instruction. Likewise, [2] has suggested that CFI detection control code should be executed at every indirect branch, such as return, jump, and call instructions, in a fine-grained CFI approach [31]. This causes a significant number of executions of CFI code leading to a performance overhead. Thus, in coarse-grained CFI approaches, this condition is relaxed to so that CFI checks are performed only when a system call is invoked.

An example of an alternative to coarse-grained CFI detection for ROP defence is based on stack integrity checks [32], [170], [21]. For example, ROPdefender [32] is a pintool that performs a return address check using a shadow-copy of the stack at runtime. Therefore, it dynamically stores return address for every call instruction in the shadow-copy of stack. In [32], the CFI detection algorithm accomplishes a return address check by comparing values on the top of the original stack and its shadow copy, before a return instruction is executed. The new malicious function call emulation introduced in this thesis will not be successful against [32] since the CFI detection code is executed for each return instruction, instead of at the beginning of a function using inline hooking. However, performing checks at every execution of a return instruction causes a high performance overhead.

The unhooking and funnelling malware variants developed in this thesis include an antivirus-aware payload combining ROP-gadgets and evasion code in hex in order to bypass DEP and ASLR, as well as to deactivate anti-malware protections by disabling them, such as inline hooking and guard-page-based protections.

### 2.1.3 Coarse-grained CFI Detection against ROP Attacks

There are several academic and industrial studies based on coarse-grained CFI detection for ROP defence, such as kBouncer [101], ROPecker [20], CTO for COTS [176], ROPGuard [43], Microsoft EMET [80], McAfee HIPS [131]. They are able to detect malicious activities with a lower overhead using the behavioral patterns of ROP-based attacks.

ROP-based attacks have been introduced for various computer architectures, such as Intel x86 [138], SPARC [15], ARM [61], Atmel AVR [42], PowerPC [70], and Z80 [19]. Similar to return-to-libc [34] attack, Return Oriented Programing (ROP) reuses instruction sequences in memory in order to avoid DEP protection. It offers Turing-complete [167], arbitrary code execution without injecting any code onto the stack, by reusing code present in memory [138]. Instead of performing a whole function call line by line, in ROP attacks, malware jumps into the middle of a legitimate function, and hence it only executes a limited number of instructions, called a gadget. Gadgets are typically located at the end of functions. A ROP-gadget is very small and usually consists of between two or five instructions. Typically, each gadget performs a specific task. Therefore, in ROP attacks, a group of ROP gadgets are inserted onto the stack to invoke them in a malicious order. ROP attacks can be detected by behavioral-based anti-malware, due to the special behavioral characteristics of a ROP attack, such as unusual control flow, malicious stack usage, and the small number of instructions included in a ROP-gadget.

In modern processors [49], Last Branch Recording (LBR) is a kernel mode feature that enables tracking of branches by recording them onto an LBR stack limited to 16 entries. kBouncer [101] and ROPecker [20] employ LBR to examine the last indirect branch instructions executed by a processor and perform several anti-malware controls on past execution.

Against ROP attacks, kBouncer [101] is coarse-grained CFI approach using runtime monitoring of critical functions by adding inline hooks. For every function call, it performs two main checks: (1) return addresses if they return to an instruction that follows a call instruction, and (2) the number of instructions executed between two sequential branches. kBouncer protects a limited number of critical functions in Windows modules using the hooking method. Its detection code is triggered at every call to an API function that is protected by [101], and it write a checkpoint in kernel space to prevent malware from simply jumping over the hook in user space.

ROPecker [20] proposes runtime protection based on LBR to identify malicious indirect branches in ROP attacks, similarly to kBouncer [101]. In [20], a database including all possible ROP-gadgets is statically generated offline, and then its detection algorithm determines that an indirect branch is malicious if control flow is directed to a ROP gadget in the gadget database. It also investigates the stack for future ROP gadgets, and these are checked based on the gadget database. ROPecker does not inject inline hooks in user space for monitoring. It implements its own mechanism for performing inspections, called a "sliding windows mechanism". The detection code of ROPecker is triggered whenever execution reaches an instruction on a new code page that is not included in a special fixed-size set maintained by ROPecker for tracking code pages in memory. The recently reached page is always replaced with the oldest executed page entry in the tracking set.

ROPGuard [43] is another defensive approach against ROP attacks. In [43], stacks are inspected by detection code to identify ROP-gadgets waiting for future execution. Similarly to [101], it also offers protection by checking return addresses if they return to an instruction following a call instruction. Its detection code is executed when a critical function is invoked. In this thesis, unhooking and funneling malware variants employ the invocation of non-critical NTDLL functions to make system calls, which enable them to evade this protection.

Similarly to ROPGuard [43], Microsoft EMET is enterprise-level anti-malware that identifies unknown malicious attacks using behavioral-based detection techniques at runtime. According to several Microsoft's Security Bulletins, Microsoft EMET is able to dynamically detect and terminate attacks that exploit various vulnerabilities [163, 164] in different types of applications, such as Internet Explorer and Microsoft Office, on computers not protected by an installed antivirus program. In 2012, Microsoft (MS) organized a defensive security contest for security researchers, called BlueHat Prize Contest [79]. After the contest, Microsoft implemented four new ROP mitigation techniques in MS EMET v3.5 based on the BlueHat Prize submissions, including caller checks, execution flow simulation, stack pivot mitigation, and special function checks that split into load library checks and memory protection checks [152, 10, 101, 43]. EMET includes protection techniques almost identical to those in ROPGuard [43] as a result of new feature addition after BlueHat. Similarly to kBouncer and ROPGuard, EMET employs an inline hooking method to monitor a set of Windows system functions considered as critical. Additionally, EMET tracks memory access operations to critical data structures using a guard-page-based pro-

tection. Its detection code is executed when a critical function is invoked or a critical data structure is accessed. Thus, in this thesis, in order to evade EMET's detection code, unhooking and funneling malware variants do not call critical functions directly, and do not access protected data structures, before removing all anti-malware protections.

### 2.1.4 Effectiveness of Coarse-grained CFI Detection

In recent years, several published security research studies [17, 117, 31] have focused on the effectiveness of defense techniques based on coarse-grained CFI detection leveraging behavioral patterns.

In order to efficiently identify ROP attacks, most defensive research studies have implemented several dynamic detection algorithms employing the following generic ROP behaviours for control flow integrity analysis.

- Stack manipulation by inserting consecutive return addresses [101, 20];
- Call-Return mismatches: return addresses usually do not point to an instruction that follows a call instruction [101, 43, 176, 80];
- Short sequences: few instructions executed between two sequential branches [101, 20];
- A long chain of short sequential branches in a row [101, 20].

Due to performance concerns, the proposed detections are only triggered by the following events:

- Any critical function call [101, 20, 43, 80, 131];
- Any critical data structure access [80];
- And indirect branch, such as call, jmp, and ret [176];
- And "sliding code page window" alteration [20].

Additionally, several detection algorithms [20, 43] analyze entries on the stack to perform anti-malware checks on future execution of possible ROP-gadgets. Likewise, several detection algorithms [101, 20] also employ the LBR feature to execute checks on previously executed instructions in recent indirect branches. The execution history provided by LBR is used to control two behavioral patterns of ROP attacks by checking "sequence length" and "the number of short sequence in a row". For example, kBouncer analyzes each LBR entry and marks an instruction sequence between

two branches as a "short sequence" if the sequence consists of less than twenty (20) instructions, which indicates that the sequence is possibly a ROP-gadget. Therefore, if kBouncer detects eight (8) and more short sequences in a row, the detection algorithm decides it as a malicious attack based on ROP-gadgets.

Research outlined in [17] provides three evasion techniques against the defences offered by [101, 20] to demonstrate how to evade behavioral checks on call-return mismatches, short sequences, as well as execution history checks employing the LBR feature. In [117], evasion techniques against [101, 20, 43] demonstrate that detection based on call-return mismatches and history information with a limited number of recent branches can be evaded by flushing the LBR stack. The LBR flushing techniques in [117] differ from those implemented in [17, 31]. The attack in [117] against [101] calls a kernel32 function that executes more than twenty (20) indirect branches, namely kernel32.lstrcmpiW, to flush the LBR stack used for storing recent branches in [101]. The study outlined in [31] implements successful attacks against a diverse collection of defence detection algorithms developed in [101, 20, 176, 43, 80] using call-ret-pair and long-nop gadgets. [31] evades all detection algorithms, including anti-malware checks for call-return mismatches, short sequences, as well as historical checks based on the LBR feature.

These research studies have shown that detection based on return address validation for call-return mismatches can be easily bypassed using a call-ret-pair gadget [31, 117] or a call-preceded [17] gadget. Using several evasion techniques, research outlined in [17, 117, 31] demonstrates that detection algorithms leveraging the LBR feature can be bypassed by enhanced malware. Due to the capacity limitation of 16 entries, the branching history stored on the LBR stack can be manipulated by malware at any given time, in a technique called history flushing. In the attacks [17, 31], the use of long-gadgets have also proved that the behavioral assumption of that "only short gadgets are employed in a ROP attack" is not secure. In contrast to [17, 31], a system function that executes several indirect branches has been employed by malware in [117] to flush history before triggering any detection algorithm.

The evasion attacks outlined in [17, 117, 31] have successfully evaded detection by history flushing and the use of call-preceded and non-operational-long gadgets. However, instead of focusing detection algorithms, the unhooking and funneling techniques implemented in this thesis do not trigger any detection mechanism, by leveraging an advanced critical function call emulation algorithm, and do not call any critical function monitored by anti-malware until removing all protections in memory, such as

inline-hooks and guard pages.

In addition to [17, 117, 31], several security reports and articles that concentrate on bypassing EMET's protections have been published on the Internet [5, 53, 134, 133, 132, 11, 33, 106, 58, 9, 1].

- One of the most recent studies of EMET v5.2 [5] has simply disabled EMET's protections by executing the unloading code portion existing in EMET, which is located at offset 0x65813 in EMET.dll v5.2.0.1. The vulnerability has been patched in EMET v5.5.

- The work outlined in [53] against EMET v3.5 has introduced two attack techniques. Due to the fact that kernelbase was unprotected by EMET v3.5, the first method employed the kernelbase.VirtualProtect() function to manipulate memory access protections, until the implementation of the deep hooking protection within EMET v4.0. The second method in [53] has discovered the address of KiFastSystemCall using the SHARED_USER_DATA structure located in 0x7ffe0000 in order to perform any system call and alter the access protections of memory pages. Similar to the funneling method implemented in this thesis, the attack in [53] against an early version (v3.5) changes EMET's code in memory to disable protection. In contrast to our approach, it does not disable any critical data structure (EAF) protection by manipulating debug registers or guard pages.

- The attack technique discussed in [134, 133, 132] disables ROP protections by exploiting a vulnerable global variable that controls EMET's ROP protections in several EMET versions, including v4.1, v5.0, and v5.1. The key vulnerability was that the critical global variable was located in a writable memory address at a fixed offset, such as the 0x0007E220 offset in EMET v4.1 [134].

- The attack implemented in [11] disables EMET's protection that monitors all accesses to the Export Address Table by manipulating the debug registers using the NtSetContextThread and NtContinue functions.

- The report outline in [58] introduces a technique to bypass EMET by abusing the WoW64 subsystem of 64-bit Windows, which is a compatibility layer to run 32-bit applications on a 64-bit Windows operating system. This technique shows that even though EMET offers an extra protection for legacy systems,

the performance of EMET is insufficient to protect 32-bit applications running under the WoW64 subsystem.

- The reports [33, 106] demonstrate two evasion techniques to safely call the LoadLibrary() function, which is monitored by EMET to protect against attacks loading malicious modules from remote sources. In order to evade EMET's caller protection, the study in [33] introduces an evasion technique that invokes a critical function using an "existing call instruction to the function" in the targeted application's code, instead of using a jmp instruction or directly returning to the function. Using this technique against EMET v4.1, malware is able to allocate executable memory by abusing the VirtualAlloc() function, and has executed a custom LoadLibrary shellcode. Likewise, in [106], the LoadLibrary() function monitored by EMET v4.0 has been called to load a malware module from local hard disk, after copying the malicious module from a remote server by invoking the MoveFile() function.

Differently from these analyses of EMET's security, our research focuses on deactivating EMET completely by disabling all userland protections of the latest version of Microsoft EMET v5.52 by implementing our two evasion methods.

## 2.2 Digitization of Life and Cyber Security

The Internet is a gigantic computer network developed in the 1970s and 1980s that covers more than billion devices worldwide, without any national or geographic boundaries [50]. In the beginning, the Internet was used by a small number of researchers for academic and research purposes. It became popular in the 1990s and has rapidly spread around the world. The increasing use of the Internet in all aspects of daily life has brought a deep change in people's lifestyles. A diverse range of educational, commercial, and governmental resources are now provided as a cloud service throughout the Internet. Not only computers but also smart phones, smart televisions, smart watches, smart home devices and smart cars are novel advancements that connect people's lives to the Internet. With continual new inventions and developments in technology, the lives of everyday people are becoming increasingly connected to this cyberworld [44]. Today the Internet is globally used by an estimated 3.6 billion people [84] in various settings, including academic, social, commercial, and entertainment.

In step with the rapid growth of the Internet, cybercrime has grown into a worldwide threat. Cyber criminals accomplish a variety of sophisticated attacks against victims across the Internet by combining technical abilities with social engineering techniques. Their methods include:

- Employing malicious or infected servers with attractive contents that act benign;
- Sending spam emails that contain viruses, phishing notes, or suspicious advertisements;
- Serving free or illegal copies of software, books, and movies that are embedded with malicious content.

Most cyber threats include malware software components to accomplish their malicious attacks in the cyberworld.

## 2.3 Malware

Most high-tech crimes that threaten information security involve malicious software. Malware is software whose code includes crafted instructions which can perform malicious activities. Malware exploits the vulnerabilities and hidden functionalities of an application in order to compromise targeted computer systems. After exploitation, malware may execute harmful components that vary based on the malicious purposes of malware. There are several forms of malware: viruses, worms, trojans, and rootkits [78].

In response to the threat posed by malicious software, developers have created an arsenal of detection and mitigation techniques ranging from simple pattern-matching to advanced behavior analysis. Likewise, malware authors have responded to techniques for the mitigation of malware by inventing new malignant methods, such as code encoding and obfuscations. Most advanced malware is able to alter its code while keeping exactly the same functionality in order to defeat detection by anti-malware software.

### 2.3.1 Malware Taxonomy

Malware is multifaceted software, so a sample of malicious code can be a member of more than one class, such as a targeted worm or an email virus. A malware sample may be classified in a variety of ways, depending on its targets, propagation methods, purposes, and functionalities.

#### 2.3.1.1 Malware Types

With respect to targets, malware may be classified as mass-spreading or narrowly targeted. Mass-spreading malware aims to aggressively infect a large number of computers, whereas targeted malware is designed to be destructive only for a specific target. Examples of the latter include Code Red, Slammer, and Stuxnet [86, 85, 65]. Similarly, malware may work against only one specific operating system or application, such as Microsoft Windows, Mac OS, and GNULinux or Microsoft Office, PDF Readers, and Internet Browsers. For example, depending on its targeted operating system or application, malware might be called "Mac OS malware", or "PDF malware" [12, 137]. Cross-platform malware takes advantage of the large attack vector offered by popular cross-platform software.

With respect to malware propagation methods, a common classification includes viruses, worms, and trojans. Viruses replicate themselves by copying their malicious code into the files [103] and critical system areas of an infected computer when they activate. A typical virus does not reside in its own file so its malicious code is injected into another file, such as an executable. Opening or execution of a virus-infected file results in the execution of the malicious viral code. Worms, on the other hand, run independently and spread stealthily between computers over a network, without any intervention from outside. As well-known example of such malware is the Blaster worm [18]. A worm exploits a remote vulnerability and replicates itself through network connections. Similarly, trojans are other destructive forms of malware. Trojans conceal their malicious payload and disguise as legitimate software. A trojan is not capable of copying itself so it infiltrates a computer by misleading users. Trojans typically spread by using social engineering techniques so the victims of trojans themselves download and install trojans on systems. Unlike worms, viruses and trojans require a human interaction or a system event to spread. Executing an infected program, opening an email attached to a virus, installing a crafted application, and visiting a malicious server result in destructive infections of viruses and trojans. Malware also spreads via various distribution channels on the Internet, such as emails, instant messages, IRC channels, or P2P networks [66].

Finally, malware is also designed for several harmful purposes with related functionalities. Malicious code usually has one or more functionalities so it can be categorized by its different functionalities and purposes. Some examples for the purposes of malicious attacks are spying, stalking, stealing (e.g., id, info, data, and money),

and sabotaging [38]. Spyware, adware and ransomware are good examples of malware classified based on its intended purpose. Like trojans, spyware infiltrates a system by tricking users. After being installed by an unsuspecting victim, spyware starts to secretly collect various types of information about users. Similarly, adware collects information about users for advertising purposes. Spyware and adware are capable of monitoring the activities of systems and users, accessing hardware components, such as webcams, and capturing personal information, such as passwords and email addresses. Ransomware is a new cyber threat that has rapidly and internationally grown since 2011 [74]. It encrypts the files of its victim on a compromised computer and destroys originals. Similar to kidnapping, then, ransomware demands a ransom payment to make the encrypted files accessible by decrypting them.

Malware can also be classified according to its functionality, including rootkits, spamware, keyloggers, backdoors, downloaders, and logic bombs. Rootkits hide their presence by loading special drivers and injecting hooks at the kernel level to bypass antivirus products. When a system is infected by a rootkit, the processes, network connections, and files of the rootkit will not be listed by standard system commands and tools. Backdoors provide attackers remote access to compromised systems. Thus, a backdoor is itself a remote control service or it offers this feature by installing a remote access application, such as vnc, rdp, or ssh. Similarly, downloaders are another malware type that download files and installs applications on the victims' computers for malicious purposes. Like a real time bomb, a logic bomb malware is triggered by a predefined time or condition to execute its malicious codes. Logic-bombs are usually part of a targeted attack, or they wait a certain amount of time with the goal of remaining undetected longer and thus able to infect more computers.

#### 2.3.1.2 Advanced Malware Types

Detection methods mostly rely on the unique binary pattern of malware so they can be bypassed if malware alters its appearance, while keeping its functionality. In order to escape detection by anti-malware products, advanced malware utilizes several stealth technologies, such as compression, encryption, and code obfuscations.

The classification of malware based on their stealth technologies follows [158]:

- Encrypted Malware
- Polymorphic Malware
- Metamorphic Malware.

**2.3.1.2.1 Encrypted Malware** Encrypted malware hides its malicious functionality and avoids detection using various encryption methods. A typical piece of encrypted malware includes encryption, decryption and functionality components, depicted in Figure 2.1. The decryption component is also called a decryption engine. In encrypted malware, code which implements a malicious functionality is encrypted, whereas its decryption engine is not. Some encrypted malware implementations support multi-layer encryption techniques. The encryption code of malware randomly generates an encryption key on every infection and stores this key inside the malware so that it may be used for decryption. Then, its malicious functionality component is encrypted by the encryption engine. When an infected application is executed, decryption engines decrypt the encrypted malicious code at run-time by using stored keys. Because encryption keys are random, the appearance of malware is different in new generations. Some anti-malware products are able to break various malware encryptions and then decrypt the constant functionality part that is stored in encrypted form. After decryption, the signatures of constant functionality parts can be used by anti-malware for detection.

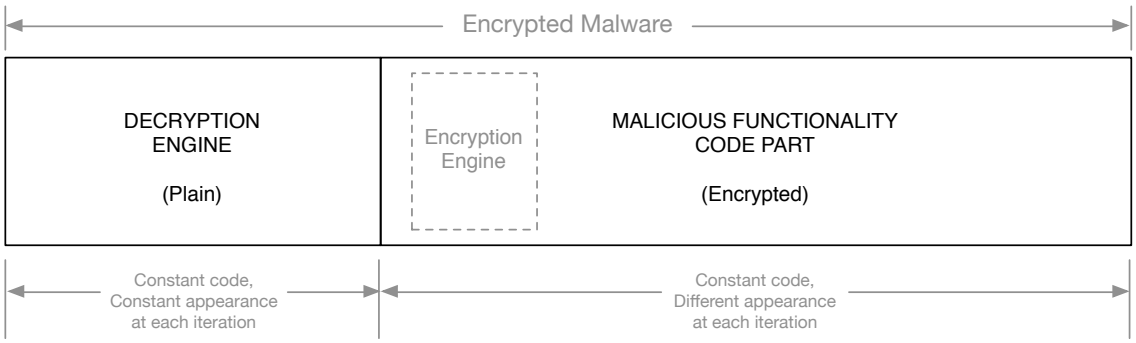


Figure 2.1: Structure of Encrypted Malware

**2.3.1.2.2 Polymorphic Malware** Polymorphic malware employs encryption methods in order to bypass detection, similarly to encrypted malware. It also consists of encryption, decryption and functionality components, depicted in Figure 2.2. The encryption-decryption engines are responsible for morphing. Although the binary pattern of the malicious component is changed with every infection, encrypted malware may be still identified by anti-malware using signatures based on the constant decryption engine that is not encrypted. Therefore, polymorphic malware changes the appearance of its decryption part as well, by inserting junk instructions and random

padding bytes. It has the capability of altering its appearance at each iteration while preserving the same functionality. During infection, newly created variants also have a new fingerprint, so anti-malware cannot properly identify them using the signature of the original malware variant. Polymorphic malware does not alter its malicious codes at each iteration, and it has to decrypt the malware functionality component in order to run. Thus, after decrypting itself in memory, polymorphic malware can be reliably detected using a signature based on its constant functionality part [63].

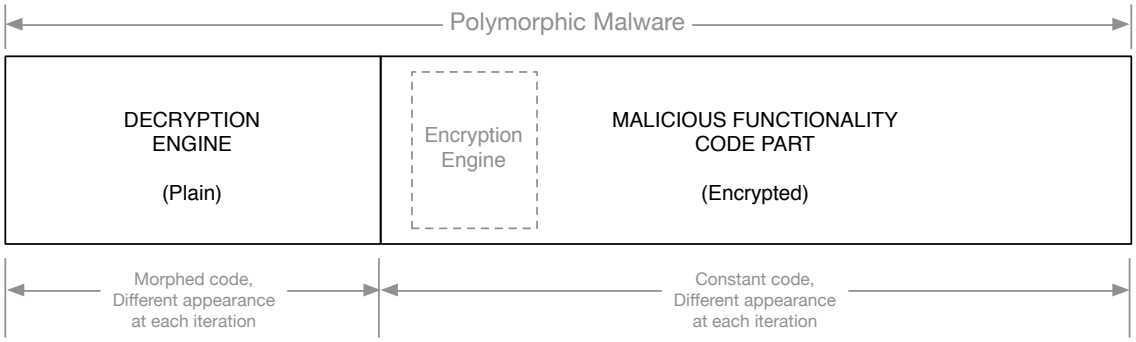


Figure 2.2: Structure of Polymorphic Malware

**2.3.1.2.3 Metamorphic Malware** Metamorphic malware mutates itself with every propagation, making signature-based detection impossible, which is similar yet more effective than both encrypted malware and polymorphic malware. There is no predictable patterns between generations. Unlike encrypted and polymorphic malware, metamorphic malware does not have decryption and constant functionality parts. It uses code-morphing techniques instead of encryption for evading malware detection. Although advanced metamorphic malware combines cryptographic and other code obfuscation techniques, the morphing methods of metamorphic malware usually rely on code obfuscation more than encryption. Thus, it has morphing and nondeterministic functionality components, depicted in Figure 2.3. Morphing engines employ various techniques for mutation, such as renaming registers or variables, or reordering instructions. A morphing engine may use techniques such as code compression and more complex obfuscations. In metamorphic malware, the morphing part is typically bigger than the functionality component. Both the malicious functionality part and its morphing code itself are dynamically mutated by the morphing engine, using code obfuscation methods, including function reordering and program flow modification. Thus, with each iteration, metamorphic malware completely re-

produces itself, by changing all its parts, so that no identical replica is generated. Metamorphic malware fully changes both its behavior and appearance. The new unidentical variants of a piece of metamorphic malware are likely undetectable by anti-malware programs. However, they provide the same malignant functionality as their parent malware.

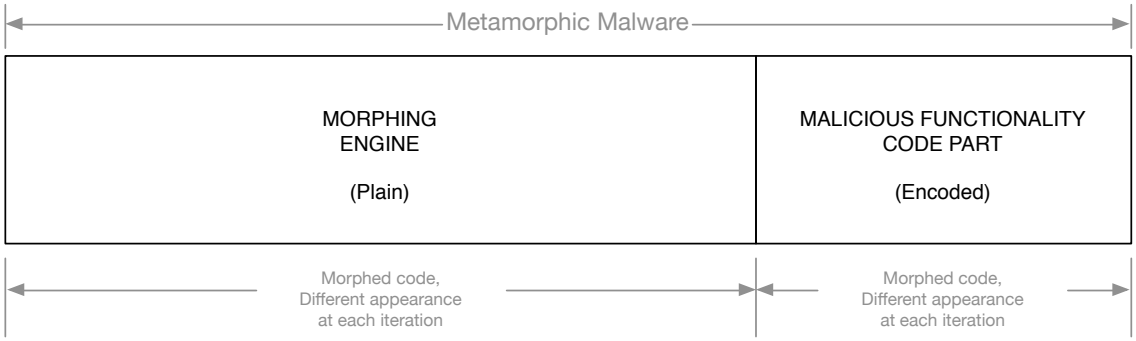


Figure 2.3: Structure of Metamorphic Malware

### 2.3.2 Malware Mitigation

Malware has evolved a great deal over the past decade. In response to the evolution of malware, more effective detection techniques have been developed and employed by anti-malware systems. There is a never-ending arms-race between anti-malware authors and malware authors. More sophisticated techniques used for detection have escalated the requirements for malware veiling methods. A considerable diversity of obfuscation and anti-reverse engineering techniques has been implemented by malware in order to avoid malware analysis and detection.

#### 2.3.2.1 Anti-Analysis

Reverse engineering has become an important approach to malware analysis. Therefore, malware employs several anti-reverse engineering techniques [168] against such approaches. Virtualization, disassembly, and debugging are common technologies used for malware analysis. By implementing anti-analysis techniques malware is able to delay and prevent analysis. Bypassing analysis software allows malware to conceal its malicious functionality and avoid detection as well.

Virtual machines are used by malware analysis systems to allow the execution of malware in a controlled environment. Malware detects whether it is running inside

a virtual machine and in response it may change its behavior at runtime. Malware may investigate footprints that indicate a virtualization environment by checking virtualization software components installed on a guest system and searching specific registry keys. Virtualization-specific cases are also tested by malware to detect virtual machines.

Likewise, disassemblers are used for malware analysis and heuristic-based detection. Malware may have special malicious code, or may obscure original entry points, in order to affect the performance of disassemblers, for example producing mistranslations. Heuristic-based detectors disassemble malware and run the translated assembly code in their emulators in order to perform similarity checks. Therefore, mistranslations alter detection results.

Debuggers are also popular software tools used for malware analysis. Using anti-debugging techniques, advanced malware is able to detect attached debuggers. Malware may then bypass detection and analysis by altering control flow misleads or causing program crashes. Techniques which examine data structures in memory enable malware to detect an attached debugger, such as the BeingDebugged field of PEB. Other approaches involve searching specific registry keys, and invoking specific system functions, such as CheckRemoteDebuggerPresent(), and IsDebuggerPresent(). Thus, critical data structures in memory and registry access are monitored by advanced anti-malware to detect similar malicious activities. Advanced malware exploits the vulnerabilities of debuggers to defeat them and also checks its own code integrity, for example determining if a debugging instruction, INT3, is inserted by a debugger. Anti-analysis techniques provide advanced protection to malware.

#### 2.3.2.2 Code Obfuscation

Obfuscation is a common behavior of malware to elude detection by anti-malware, especially signature-based detection and heuristic-based detection. It also complicates malware analysis. After code obfuscation, malware code appears completely different, while it maintains exactly the same malicious behaviour. Obfuscation techniques may be classified as code-level alteration and code encoding.

**2.3.2.2.1 Code Encoding** Advanced malware employs encoding in order to conceal and protect its malicious functionalities and techniques. In polymorphic and metamorphic malware, a morphing engine contains relevant decoding stubs to decode the encoded malicious part. Encoding makes static analysis complicated and

provides protection against signature-based detection. The encoding techniques used by morphing malware widely vary from simple encoding to packing. Custom encoding techniques are also developed and implemented by malware developers in order to decrease the likelihood of detection by well-known algorithms

Examples of simple encoding techniques commonly used by malware follow:

- XOR encoding
- BASE64 encoding
- ROT13 encoding (a Caesar cipher)
- Packing.

**2.3.2.2.1.1 XOR encoding** The eXclusive OR operation (XOR) encoding is one of the most used malware obfuscation techniques due to its easy implementation. The encoding performs XOR, which is a bitwise operation. Similarly to XOR, several bitwise operations, such as ROL, ROR, ROT, and SHIFT, are also commonly implemented by encoding algorithms. In the XOR encoding technique, the same key and algorithm are used for both encoding and decoding. For example, let us assume that we encode the letter 'H' using XOR encoding with the letter 'E' as the encoding key, depicted in Figure 2.4. In the ASCII character table, the hexadecimal value of the letters 'H' and 'E' are 0x48, and 0x45. Their binary values are '01001000' for and '01000101' respectively. The encoded code is computed by XORing these two values, which equals '00001101', 0x0D in hex. Similarly, for decoding, the encoded value is XORed with the same key, 'E'. The result of '00001101' xor '01000101' equals '01001000', which is the original value that represents the letter 'H'.

| Original Code | Key | Encoded Code by XOR using key E |
|---|---|---|
| H E L L O W O R L D<br>48 45 4C 4C 4F 57 4F 52 4C 44 | E<br>45 | 0D 00 09 09 0A 12 4F 17 09 01 |

Figure 2.4: XOR Encoding

**2.3.2.2.1.2 Base64 encoding** Base64 encoding is used by malware for obfuscation, even though it has mainly been designed for transferring data over a network [54]. The encoding results are also null-character-free, which makes shellcode development easier. Base64 encoding translates 24-bit groups of input data into groups

of 4 encoded characters using an alphabet table, depicted in Figure 2.5. Original input messages are divided into 6-bit groups that are considered as index values in the Base64 alphabet table. The Base64 alphabet is typically a 65-character subset of the ASCII table, including the padding character '=', as shown in Figure 2.6. Some advanced malware implements a custom base64-based encoding by reducing or modifying the standard Base64 alphabet in order to avoid the generic Base64 decoders.

| | | | | | | | | |
|---|---|---|---|---|---|---|---|---|
| 0 A | 8 I | 16 Q | 24 Y | 32 g | 40 o | 48 w | 56 4 | (pad) = |
| 1 B | 9 J | 17 R | 25 Z | 33 h | 41 p | 49 x | 57 5 | |
| 2 C | 10 K | 18 S | 26 a | 34 i | 42 q | 50 y | 58 6 | |
| 3 D | 11 L | 19 T | 27 b | 35 j | 43 r | 51 z | 59 7 | |
| 4 E | 12 M | 20 U | 28 c | 36 k | 44 s | 52 0 | 60 8 | |
| 5 F | 13 N | 21 V | 29 d | 37 l | 45 t | 53 1 | 61 9 | |
| 6 G | 14 O | 22 W | 30 e | 38 m | 46 u | 54 2 | 62 + | |
| 7 H | 15 P | 23 X | 31 f | 39 n | 47 v | 55 3 | 63 / | |

Figure 2.5: The Base64 Alphabet Table

```
Input String   : H               E               L
Input Binary   : 0 1 0 0 1 0 0 0 0 1 0 0 0 1 0 1 0 1 0 0 1 1 0 0
Computed Index :     18      |      4      |      21     |      12     |
Encoded output : SEVM
```

| Original Code | Encoded Code by Base64 |
|---|---|
| HELLO WORLD | SEVMTE8gV09STEQ= |

Figure 2.6: Base64 Encoding

**2.3.2.2.1.3 ROT13 encoding** The ROT13 encoding simply rotates characters to the right 13 times, depicted in Figure 2.7. The ROT13 encoding is a Caesar cipher and ROT13 stands for rotate 13. It uses the same code for both encoding and decoding, which again leads to an easy implementation [102].

| Original Code | Encoded Code by ROT13 |
|---|---|
| HELLO WORLD | URYYB JBEYQ |

Figure 2.7: ROT13 Encoding

**2.3.2.2.1.4 Packing** Packing is widely used by morphing malware because it allows both obfuscation and compression. Malware code compressed by packing software is smaller than its original code and mostly consists of random-appearing data, which complicates malware analysis by concealing the functionality of malware. Likewise, anti-malware cannot inspect and identify its malicious code without unpacking it. Packing software [99, 141] is typically designed for compression, but some advanced packing tools [165] also implement encryption and anti-analysis algorithms, such as anti-virtualization, anti-debugging and anti-disassembly, in order to seriously degrade the effectiveness of signature-based malware detectors. Packing software briefly compresses malware code and inserts its own unpacking stub to restore the original code when executing. Morphing malware that employs packing technology contains two parts: a morphing engine and a packed malicious code section. The morphing engine including a relevant unpacking stub opens the packed malware code when malware is loaded into memory, depicted in Figure 2.8. Several anti-malware programs are able to recognize the encoding and packing algorithms used by malware and unpack packed malware in order to perform detection. Thus, most advanced malware is packed by custom packing software developed to defeat the standard packing detectors.

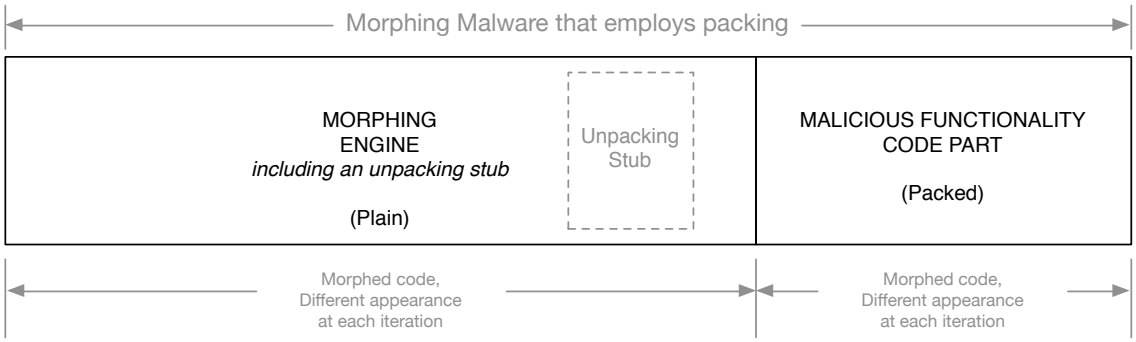


Figure 2.8: Morphing malware employed packing

**2.3.2.2.2 Code-level Alteration** Malware employs several obfuscation techniques at not only the binary, but also the code level. Code alteration can be categorized as statement permutation, statement exchanging, and junk code insertion. The categories of code-level alteration follow below:

- Statement permutation
- Statement exchange

- Junk code insertion.

**2.3.2.2.2.1 Statement permutation** Statement permutation briefly reorders the statements of malware to defeat detection and it has the following sub-categories:

- Independent code and block permutation
- Code block permutation using jmp instructions
- Function block permutation.

**Independent code and block permutation** At the code level, some instructions perform a functionality independent from other instructions. Therefore, reordering independent instructions has no effect on a malware behavior, but does alter its appearance. Changing the appearance of malware provides invisibility against some forms of detection. The number of new variants that can be generated depends on the number of independent instructions and instruction blocks of malware. For example, in Figure 2.9, instruction 1 (i1) and instruction 2 (i2) depend on each other. The order of i1 and i2 is unchangeable because the execution result of i1 affects i2. Instruction 1 and 2 are also considered as one independent code block (cb1). Similarly, instruction 3 (i3) and instruction 4 (i4) are independent so cb1, i3 and i4 can be permuted. However, instruction 5 (i5), call edx, is a restricted instruction with respect to code order because changing the position of i5 would affect the functionality of malware.

```
Instruction 1 - xor eax, eax --------> Independent group
Instruction 2 - add eax, 10h ---/
Instruction 3 - mov ebx, [esp] ------> Independent instruction
Instruction 4 - sub ecx, esi --------> Independent instruction
Instruction 5 - call edx ------------> Restricted instruction
```

Figure 2.9: Independent Code Permutation

**Code block permutation** In this technique, malware code is virtually divided into code blocks and their order is permuted. As shown in Figure 2.10, the execution flow of altered malware preserved to be exactly the same as the original flow by inserting additional branching instructions, such as jmp. Preserving control flow provides the same malicious behavior, but with a different code appearance. The size of malware variants are slightly more than original malware because of additional jmp instructions. This technique offers a number of different variants

(ndv) based on the number of blocks (n) that malware is virtually divided into. (ndv = n!)

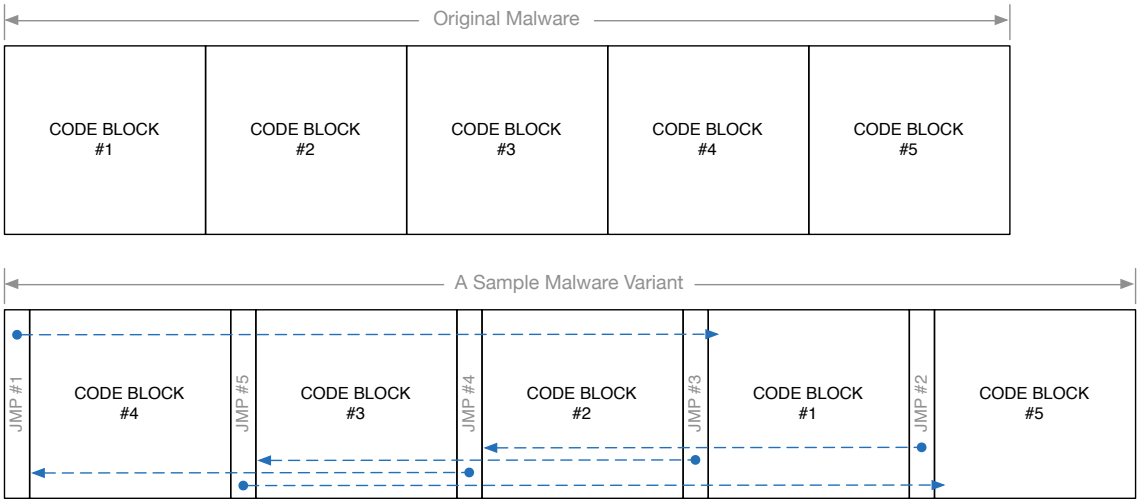


Figure 2.10: Code Blocks Permutation

**Function block permutation** The order of function blocks of malware is permuted in this technique, depicted in Figure 2.11. Reordering function blocks retains functionality, but it changes the byte pattern of malware. Unlike code block permutation, function block permutation does not require adding any branching instruction, such as jmp. For example, a piece of malware that includes ten (10) functions produces more than 3.5 million different variants (10!=3628800) via the function block permutation technique.

**2.3.2.2.2.2 Statement substitution** Briefly, statement substitution replaces statements occurring in malware code with functionally equivalent statement sequences in order to evade detection. It has the following sub-categories:

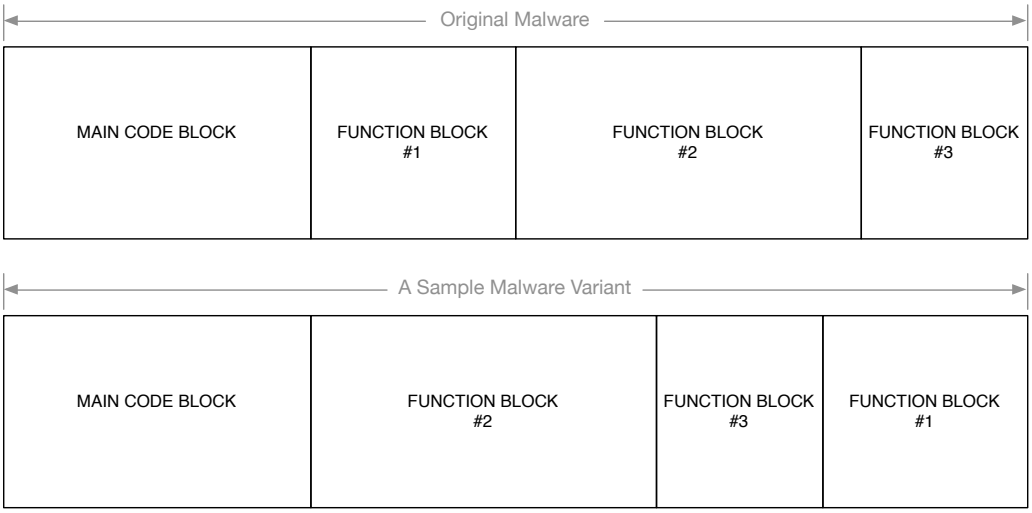


Figure 2.11: Function Blocks Permutation

- Equivalent statement replacement
- Register and variable renaming.

**Equivalent statement replacement** In this method, one instruction or a group of dependent instructions is replaced with other equivalent instruction sequences. Both instruction sets perform exactly the same functionality so there is no logical impact from this modification. Some simple examples of equivalent statement replacement are given in Figure 2.12.

**Register or variable renaming** Renaming variables or reassigning registers also changes the byte pattern of malware, while maintaining functionality. Registers used in malware instructions can be exchanged with other available registers to alter the appearance, as depicted in Figure 2.13. Similarly, changing the names of variables and functions has no impact on the functionality of malware that is developed in a scripting language, such as Javascript, as shown in Figure 2.14.

**2.3.2.2.2.3 Junk statement and data insertion** Additional instructions that do not have any logical effects on malware behaviors are considered junk statements. Junk statements do not alter any functionality. No operation (NOP) instruction, the instructions of adding or subtracting zero, and branching to other nonfunctional junk code blocks are simple examples of junk statement insertion, depicted in Figure 2.15. In order to change the byte pattern of malware for bypassing detection, a random number of random data is also inserted into the malware code, such as

| Original Code | Equivalent Code |
|---|---|
| `jmp eax` | `push eax`<br>`retn` |
| `mov eax, ebx` | `push ebx`<br>`pop eax` |
| `add eax, ebx` | `neg ebx`<br>`sub eax, ebx` |
| `sub eax, ebx` | `neg ebx`<br>`add eax, ebx` |
| `inc eax` | `add eax, 1` |
| `dec eax` | `sub eax, 1` |
| `xor eax, eax` | `sub eax, eax` |
| `cmp eax, 0`<br>`je sub_01`<br>`.`<br>`.`<br>`sub_01: ...` | `cmp eax, 1`<br>`jne sub_01`<br>`.`<br>`.`<br>`sub_01: ...` |

Figure 2.12: Simple Examples of Equivalent Code

| Original Code | Register Exchanging |
|---|---|
| `mov ebx, 10`<br>`xor eax, eax`<br>`add eax, ebx` | `mov ecx, 10`<br>`xor edx, edx`<br>`add edx, ecx` |

Figure 2.13: Simple Examples of Register Exchanging

random padding bytes. Both junk statement insertion and junk data insertion alter the appearance of malware, while preserving its functionality.

### 2.3.3 Malware Detection

The most important component of defence against malware is anti-malware software, including malware detectors. A malware detector is software that identifies malware using signatures and behaviorally-based detection techniques. Anti-malware products frequently update their malware signatures, rules, and behavior profile databases. A basic malware detector fulfills the three basic features of detection, identification and prevention.

| Original Code | Variable Renaming |
|---|---|
| `function foo(a, b) {`<br>`  var sum = a + b;`<br>`  return sum;`<br>`}`<br><br>`foo(5, 15);` | `function zMgajs2k(fg4rrDf, gFee3xVb) {`<br>`  var qWder = fg4rrDf + gFee3xVb;`<br>`  return qWder;`<br>`}`<br><br>`zMgajs2k(5, 15);` |

Figure 2.14: Simple Examples of Variable Renaming in Javascript

| Original Code | Junk Code Insertion |
|---|---|
| `mov ebx, 10`<br>`xor eax, eax`<br>`add eax, ebx` | `mov ebx, 10`<br>`nop --------------> Junk instruction`<br>`xor eax, eax`<br>`nop --------------> Junk instruction`<br>`nop --------------> Junk instruction`<br>`add eax, ebx`<br>`nop --------------> Junk instruction` |
| `mov ebx, 10`<br>`xor eax, eax`<br>`add eax, ebx` | `mov ebx, 10`<br>`add ebx, 0 -------> Junk instruction`<br>`xor eax, eax`<br>`sub eax, 0 -------> Junk instruction`<br>`inc ecx ----------> Junk instruction`<br>`mov ecx, [esp] ---> Junk instruction`<br>`add eax, ebx` |

Figure 2.15: Simple Examples of Junk Code Insertion

#### 2.3.3.1 Malware Detection Techniques

A wide variety of detection techniques are used by anti-malware:

- Checksum-based detection
- Signature-based detection
- Heuristic-based detection
- Behavioral-based detection.

**2.3.3.1.1 Checksum-based detection** Checksum-based detection is a fairly simple yet effective technique to determine if a program is malicious. In the checksum-based detection technique, users and malware detectors calculate the hash value of a suspicious file by using a checksum function, such as MD5 or SHA256. They then send the calculated hash value to an online malware analysis service [69, 160, 55]. Most online malware analysis services return a result against multiple anti-malware products, depicted in Figure 2.16.

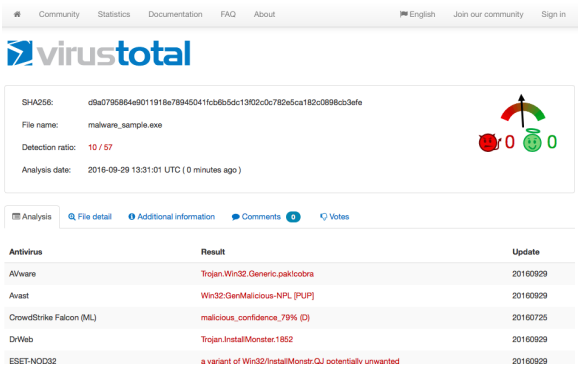


Figure 2.16: Checksum-based detection via VirusTotal, an online malware analysis service

**2.3.3.1.2 Signature-based detection** Malware detectors employ predefined malware signatures to detect a malicious threat in the signature-based detection technique. A signatures is a distinctive sequence of bytes that identifies the piece of malware in question. A malware-specific bit pattern is considered as a fingerprint that identifies malware. In signature-based detection, fingerprints are used as signatures for identification of malware. Signatures are defined as a result of malware analysis and stored in malware databases. Signature-based anti-malware products search for the presence of signatures, certain byte sequences, inside suspected files, network packets and critical memory areas when scanning for a malicious threat. If one of the known signatures in its database is found by anti-malware, the scanned object is likely infected. Modern signatures support wildcards. Unfortunately, this technique is only able to detect previously known malware, which has a defined signature. Morphing malware, which has a different code appearance in each run, is likely to bypass this technique too. Therefore, signature-based anti-malware products are usually ineffective against unknown and new malware.

**2.3.3.1.3 Heuristic-based detection** In heuristic-based detection, malware detectors use weighting algorithms and decision rules, which check for specific instructions that indicate malicious action inside a program, instead of using signatures. A rule defines a typically harmful action. A heuristic-based detector identifies potential malware functionalities based on characteristics that are only found in typical malicious codes [116]. Modern detectors also identify morphing codes, such as encryption and decryption. Each instance of a malicious functionality is given a weight based on its threat. The detector determines a program as malware if the sum of weights is

more than a critical threshold or there is a matched rule within its rules databases. Unlike signature-based methods, this approach is able to proactively detect malicious code without having a malware-specific signature. Heuristic-based detection also offers a detection ability against polymorphic and metamorphic malware, although it is limited.

**2.3.3.1.4 Behavioral-based detection** Many popular consumer-level anti-malware products have combined signature-based and heuristic-based detection techniques. However, these techniques are mostly susceptible to the obfuscation methods and barely sufficient to identify morphing malware. In behavior-based detection, malware detectors perform a detection technique based on the behavioral characteristics of malware. Similarly to heuristic-based detection, behavior-based detection does not require static malware signatures in order to detect malicious codes. It can identify unknown or similar harmful code as malware using behavioral patterns. Most behavior-based techniques are able to dynamically detect malicious activity during malware execution.

There are several academic studies based on behavior-based detection that use control flow analysis, API call sequences, API call frequency, and API call graphs to determine whether a threat is malicious. In most of these research studies, behavior-based detection methods are combined with data-mining and machine-learning algorithms in order to develop behavior models. In real-world settings, several anti-malware companies have offered their enterprise-level customers products [80, 131, 157] that implement behavioral-based detection methods in order to provide efficient and effective detection.

### 2.3.4 Malware Analysis

Most serious cyber threats involve harmful software components. It is critical to diagnose malicious threats by performing detailed analyses and to develop an effective cure to defeat them. Malware analysis collects information about suspected software in order to determine whether it is malware. Analyses are performed in order to provide a complete understanding of the exploitation methods and damaging effects of malware, so that detection and prevention techniques can be developed against it, for example patching exploited vulnerabilities and identifying compromised computers and network systems. Security experts employ several effective techniques to analyze

malware threats using different specific software tools. Static analysis and dynamic analysis are the essential approaches in malware analysis [140, 75]. In the static analysis approach, malware files are analyzed without running them by examining their meta-data information and assembly codes, whereas in the dynamic analysis approach, suspected applications are executed in a controlled environment in order to analyze and determine their run-time malicious behaviors. Each approach has its own advantages and disadvantages. Thus, in most cases, both analysis techniques are combined when investigating a cyber threat.

#### 2.3.4.1 Static Analysis

Static Analysis provides detailed knowledge about what malware does by analyzing but not running malicious files. Several software tools, such as antiviruses, online services, special file viewers, and disassemblers, are used together to gather intelligence when analyzing malware. In order to determine whether a file is malicious, anti-malware products and online malware analysis services [69, 171, 68] can be employed first by scanning and uploading suspected malicious files. Both of these services include enormous databases that contain data about a significant number of currently known malware threats, so they are effective mostly against the previously identified threats. Examining the metadata and content of a malware file using file viewers [83, 107, 142], and disassemblers [115, 8] reveals its interior structure, including malicious components. Different forms of malware target a broad range of software, so may appear in a variety of specific file formats based on the target application and operating system, for example, exe, pdf, doc, and html. In the Windows operating system, the Portable Executable (PE) format is used as a file format for executable files and modules. The header and sections of a PE file include pertinent information for malware analysis, such as loaded modules, and imported and exported functions. A imported function executes an internal functionality of the operating system. Most malware calls several specific functions of various system modules, such as ntdll, and kernel32 in Windows. Therefore, the imported functions and the loaded modules provide information about malicious functionality, for example, logging keystrokes, allocating new memory space, creating files, and loading modules. The names of the exported functions are useful to understand the structure of malware, when the author of advanced malware does not obfuscate them by giving meaningless or misleading function names. Another place to check are the strings in a malware file.

Additional malicious modules can be downloaded by malware from a remote server by using an embedded file name and URL string. Thus, extracting embedded text, such as messages, errors, names and URL strings, may expose the purpose of malware as well. Likewise, in static analysis, disassembling the executable machine code of malware may provide insight into its functionality. The exposed assembly code is manually analyzed by security experts, but not executed. However, when analyzing advanced malware employing encoding and packing techniques for obfuscation, only its encoding or packing engine code can be disassembled. Therefore, less information can be collected using static analysis than using dynamic analysis that allows malware to deobfuscate itself inside a debugger.

#### 2.3.4.2 Dynamic Analysis

In Dynamic Analysis involves, malware is run in a controlled setting on a target system. Runtime information is collected during execution, and when execution is complete, activities and changes present in the system are examined. This can be a manual or automatic process. Custom-designed systems or malware sandboxes [41, 73] can be utilized for malware analysis. Sandboxes are designed to automatically monitor and analyze the activities of malware, whereas debuggers are also used to manually debug the interior of malware. In Dynamic Analysis, the network, file system, registry, and process activities of suspected software are monitored and examined in order to confirm maliciousness. The analyses are performed in a controlled environment that has a dedicated virtual or physical machine with access to a simulated network. In such an environment, a spurious DNS server forwards DNS requests to a specific address that simulates common network services, by using specific simulation tools [39]. Malware connects to the mock services that log its remote connections and DNS requests. Thus, the network activities of malware can be captured and analyzed by tools, such as packet sniffers. File system activity can be an access, creation, and deletion of a file on the compromised system. For example, malware such as keyloggers and spyware, creates new files or reads sensitive system files on the victim's computer. Similarly, various registry keys of Windows are important for system security, so malware may also read and write certain registry keys. Registry activities may be tracked to and a list of changes that may indicate malicious action compiled. Furthermore, a running malicious process may spawn new processes, load necessary modules, and/or call critical system functions in order to perform malicious

tasks, such as manipulating registers and memory. Suspicious and destructive actions can be detected by monitoring and examination of all process activities by specific software tools. Dynamic Analysis provides a deep and illuminating insight into the malware behavior.

### 2.3.5 Malware Components

Malware typically consists of two fundamental components: exploit and shellcode. It may optionally include NOP slide and padding sections as well. An exploit section includes instructions that inject the shellcode and manipulate control flow by triggering a software vulnerability, whereas the shellcode itself performs the main harmful activity. Sometimes it is difficult to predict the address of injected shellcode, so a NOP slide may be added to malware in order to increase the predictability of shellcode locations. Similarly, including a padding section allows more precise alignment of shellcode in memory, which increases malware reliability.

#### 2.3.5.1 NOP Slide

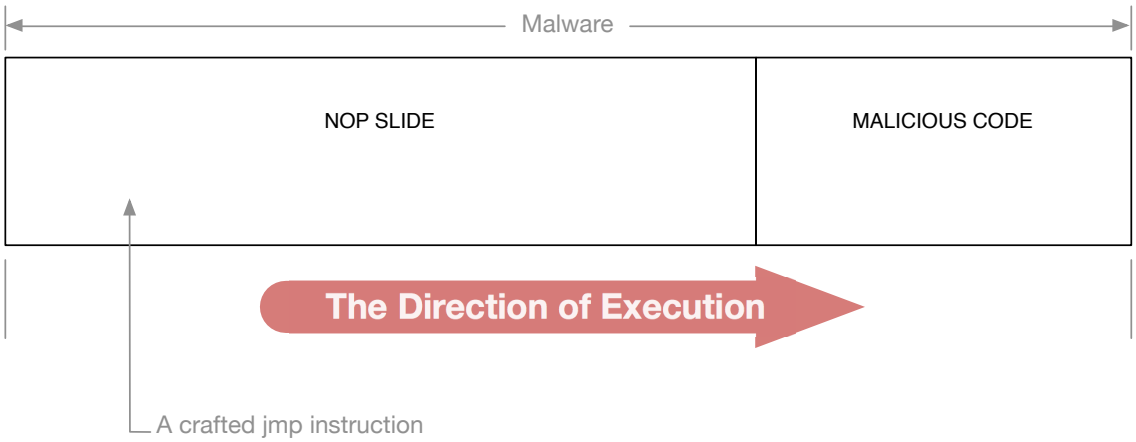


Figure 2.17: NOP Slide

A NOP slide section, also known as NOP sled, is optionally included in malware to increase the stability of malware and reduce dependence on the use of exact memory addresses for shellcode. When determining the correct address is difficult, a NOP slide allows malware to employ predictable locations. A NOP slide has a repetitive byte pattern and consists of a long sequence of non-functional instructions that precedes the malicious functionality code. When execution is redirected into any address in the NOP slide section, the remaining instructions in the NOP slide section will be executed, followed by the malicious section, as shown in Figure 2.17. The NOP instruction (0x90) is typically used for creating a NOP slide. In response, advanced anti-malware checks for the existence of a long NOP instruction chain. As a

counter-response, modern NOP slides often consist of a set of repeating non-functional instructions rather than just using the NOP instruction.

#### 2.3.5.2 Padding

Padding is another optional component, depicted in Figure 2.18. In contrast to a NOP slide section, a padding section simply consists of meaningless random bytes and it has no functionality in control flow. Although it does produce different-looking binaries, making simple signature-based detection more difficult, a padding section is typically included in order to align shellcode in memory and increase reliability, especially for heap-spraying attacks. Adding a padding section also allows malware to have a constant size shellcode, which simplifies the development process of malware.

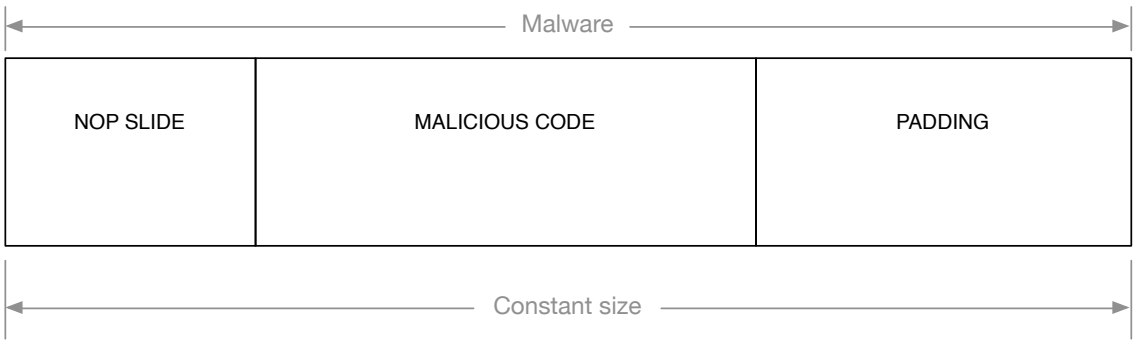


Figure 2.18: Padding

#### 2.3.5.3 Exploit

An exploit is a set of instructions used to exploit a software vulnerability for malicious purposes in order to alter the behavior of a benign program, causing it to act in an unintended manner. The exploit part of malware usually combines several languages, such as html, javascript, as well as machine code. When malware executes, the exploit first delivers its corresponding shellcode into the memory of victims' computers, and then triggers vulnerabilities in its target program and/or execution environment. Once the exploit successfully triggers a vulnerability, it manipulates the instruction pointer (IP) in order to redirect control flow to shellcode.

Because many systems now provide some form of Data Execution Prevention (DEP), advanced malware may first execute a ROP chain section before transferring execution to shellcode in order to obtain an executable memory space. For example,

after malware carefully injects shellcode onto the stack or heap, it modifies the access permissions of memory pages that shellcode relies on in order to be able to execute it. Using the Return Oriented Programming (ROP) technique, malware may invoke the VirtualProtect function to set the memory pages as executable. Similarly, another way to have memory space is that malware begins with allocating a memory area with the executable right by calling the VirtualAlloc or HeapCreate function via ROP and then copies shellcode into the allocated area to execute it.

**2.3.5.3.1 ROP chain:** Return Oriented Programming (ROP) is a code-reuse attack technique exploiting function call-return behavior which is similar to its ancestor, the return-into-library technique. In order to execute shellcode, malware usually requires an executable memory space. However, when DEP is enabled, the execution of code located on non-executable memory regions is disallowed by malware protection technologies implemented in both hardware and software. Thus, in order to bypass DEP, most malware usually employs ROP techniques. In ROP, chunks of code known as "ROP gadgets" are borrowed from modules loaded in memory for use by malware, in order to execute malicious, shellcode-like instruction sequences. Even though a ROP gadget consists of a set of benign instructions ending in a return instruction, ROP provides Turing-complete computation [167] through carefully crafted chaining of ROP gadgets. Today, most advanced malware includes a ROP chain section that alters the access protection of shellcode-related memory pages of the stack or heap in order to enable code execution.

When ASLR protection is enabled, program modules are mapped into different locations in virtual memory space every run. ROP chains are generated using code chunks belonging to loaded modules in the virtual address space of a process, so malware has to know the exact memory addresses of the loaded modules. However, advanced malware may still conduct a ROP attack by exploiting a memory leak to find a memory address of a known module, enabling it to bypass ASLR by generating a specific ROP chain at runtime. If malware controls the stack and injects the ROP gadgets, which are simply crafted return addresses, onto the stack, it can induce arbitrary behavior in a vulnerable program.

#### 2.3.5.4 Shellcode

Shellcode consists of a set of instructions that performs the main harmful activity of malware. It may be thought of as the "primary payload" delivered by malware.

Shellcode is injected into a victim's program memory and executed by the malware's exploit section. Depending on the vulnerability type, it can be injected into various memory sections, such as the stack, or the heap. When a vulnerability allows a limited memory space for injection, shellcode includes only a small instruction set for downloading and launching code, allowing it to download harmful parts from a remote network source and execute them. This is called drive-by download attack and also provides a level of protection for the functionality code of malware. Another method to deal with space restrictions, especially in stack overflow vulnerabilities, is to use a separate shellcode, which is called an egg [26]. Shellcode will do an "egg-hunt": searching process memory for the egg, prefixed by a constant number of bytes of a predefined tag value. The egg, a special shellcode, can then be injected into another memory area which does not have a byte-number limitation in size.

After injecting the shellcode, the exploit section triggers a vulnerability on the targeted system in order to gain control flow and transfer execution to the shellcode. Therefore, shellcode is typically in the form of machine code, which is executed directly by computer processors. Assembler is usually the preferred language for shellcode development, but ultimately it is assembled to produce machine code. Some instruction translations into machine code may produce all-zero bytes. Shellcode typically will not contain a null character due to problems arising from interpreting the null character as a string terminator. In order to obtain a null-free version of machine code translation, instructions producing null characters can be altered with equivalent null-free instructions, as shown in Figure 2.19. Likewise, data encoding, such as base64 encoding, reshapes the appearance of code providing the ability to generate null-free. Another technical challenge in designing shellcode arises from the fact that it is injected into different memory locations in each run. As a result, shellcode typically will use relative addressing rather than hardcoded addresses. In order to use relative addresses, shellcode needs to know a starting point, which is called base address. There are several ways to obtain a base address in shellcode. One of the most common methods for finding an address at runtime is to sequentially execute CALL and POP instructions. A CALL instruction pushes the address of the next instruction following the call instruction onto the stack and jumps to its target address. In the target location, a POP instruction pops the value pushed by CALL from the stack, giving an absolute address that will be used as the base address. After finding a base address in memory, shellcode will be able to compute absolute addresses by adding relative addresses to the base address.

| **Original Instructions** *causing zero bytes* | | **Null-free instructions** | |
|---|---|---|---|
| `mov eax, 0` | `b8 00 00 00 00` | `xor eax, eax` | `31 c0` |
| `mov eax, 1` | `b8 01 00 00 00` | `xor eax, eax`<br>`mov al, 1` | `31 c0`<br>`b0 01` |
| `mov eax, 0x10` | `b8 10 00 00 00` | `mov eax, 0xFFFFFFF0`<br>`neg eax` | `b8 f0 ff ff ff`<br>`f7 d8` |
| `mov eax, 0x1000` | `b8 00 10 00 00` | `xor eax, eax`<br>`mov ax, FFDh`<br>`inc eax`<br>`inc eax`<br>`inc eax` | `31 c0`<br>`b8 fd 0f`<br>`40`<br>`40`<br>`40` |

Figure 2.19: Null-free instructions

Shellcode is typically injected into the stack and the heap. Both of these memory regions are designed for storing process data so they are usually marked as read and write only. However, shellcode requires to be located in an executable memory area. Thus, before execution is passed to shellcode, malware usually first manipulates process memory by exploiting system functions in order to prepare an executable memory area. This requires kernel-level operations accessed by invoking system calls, syscalls, such as loading additional useful modules, altering memory protections, spawning threads, and allocating memory areas. Syscalls can be directly used in the Linux operating systems, whereas the Windows operating systems do not allow user-mode processes to directly access to system calls. Windows provides the Windows API implemented in a number of dynamically-linked system modules, or DLLs. The Windows system API is an interface between the user and kernel levels and gives access kernel-level operations in user space. In Windows, shellcode is able to discover addresses of loaded modules by traversing the data structures of the process environment block in memory [147], thereby locating the modules that it requires to successfully execute. For example, shellcode sets the access control to a memory region by invoking the VirtualProtect() function in order to obtain permission for the read, write, and execute operations. Shellcode also allocates arbitrary-sized memory blocks with executable access permission as a private heap region which may be used to execute malicious code by calling VirtualAlloc or HeapCreate. After obtaining its own executable memory region, shellcode has no limits.

Shellcode needs to call several Windows API functions in order to perform its

malicious behavior. Therefore, most behavior-based anti-malware products monitor these critical functions by using "hooking" methods.

## 2.4 Common Vulnerabilities Used by Malware

A software vulnerability is a design flaw or an improper implementation of a program design, for example in an application or operating system. Malware exploiting a vulnerability usually gains the same user rights as the victim's user. Exploits typically take advantage of memory corruptions, such as buffer overflows, in order to execute arbitrary, malicious code on a targeted system.

The following is a list of common software vulnerabilities that are exploited by malware:

- Buffer overflow vulnerability;
  - Stack-based buffer overflow;
  - Heap-based buffer overflow;
- Integer overflow vulnerability;
- Format string vulnerability;
- Use-After-Free vulnerability.

### 2.4.1 Buffer overflow vulnerability

Buffer overflows are one of the most exploited vulnerabilities [86, 85]. Software allocates a finite number of blocks in memory as a buffer area in order to store incoming information before it is processed. Buffers are contiguous memory blocks allocated for storing dynamic data at runtime. If the number of bytes that is written into a buffer is more than the space allocated in memory for the buffer, the overflow will corrupt adjacent data in memory, and the program will likely abort, as illustrated in Figure 2.20. However, instead of causing a simple program crash, malware usually abuses a buffer overflow vulnerability by corrupting memory in order to inject malicious code and gain control of execution. A buffer is usually allocated either on the stack or heap. The stack and heap are two memory segments used to dynamically store program data at runtime. During execution, the stack stores information for all nested function calls, including the return addresses, function parameters, and local variables, whereas dynamically allocated variables are located on the heap. Buffer

overflow vulnerabilities can be categorized into two types depending on where the buffer overflow occurs: stack-based and heap-based.

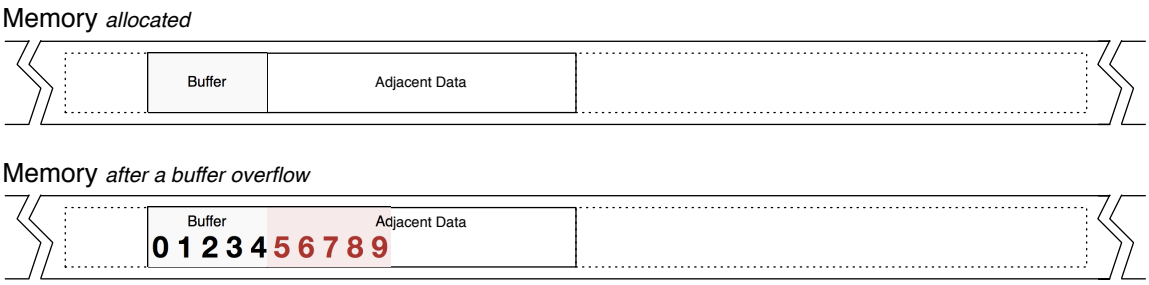


Figure 2.20: Buffer Overflow

In the C programming language, several memory management functions may cause a memory corruption, including strcpy(), gets(), scanf(), sprintf(), strcat(), malloc(), calloc() and realloc() due to the lack of explicit bounds checking. Figure 2.21 demonstrates example code stubs causing a buffer overflow.

| **Stack-based Buffer Overflow** | **Heap-based Buffer Overflow** |
|---|---|
| `char buffer[5];`<br>`strcpy(buffer, “0123456789abc”);` | `buffer = char* malloc(5);`<br>`strcpy(buffer, “0123456789abc”);` |

Figure 2.21: Example Code Stubs of Buffer Overflow

#### 2.4.1.1 Stack-based buffer overflow

Stack-based buffer overflow occurs when a program writes data requiring more space than allocated to a buffer on the stack. In such attacks, also known as ”stack smashing” [100] attacks, malware may manipulate the instruction pointer, the base pointer, and the Structured Exception Handler (SEH), which are stored on the stack, in order to redirect control flow to its malicious code [100, 40, 25]. The stack is used to maintain function calls and it grows downward, unlike other memory sections. During a function call operation, three registers, the instruction pointer (IP), stack pointer (SP), and base pointer (BP) have important roles to track the state of execution and the stack. IP typically stores the current instruction address in memory. BP is used for finding the location of function parameters and local variables, whereas SP always points to the top of the stack, which works on ”the last in first out” (LIFO) principle. When a function is called by a caller function, first the function parameters

are placed on the stack in reverse order, and then the return address is saved on the stack. A return address is the address of an instruction after the function call instruction. The saved return addresses are used for returning to the calling functions. The called function is responsible for preparing the stack and registers to use. Thus, after the caller function has invoked a function, the called function first saves the current value of BP onto the stack and then copies the current value of SP into BP. Next, a contiguous memory space is allocated for the local variables of the called function by decrementing SP. Figure 2.22 illustrates a stack frame after a function call.

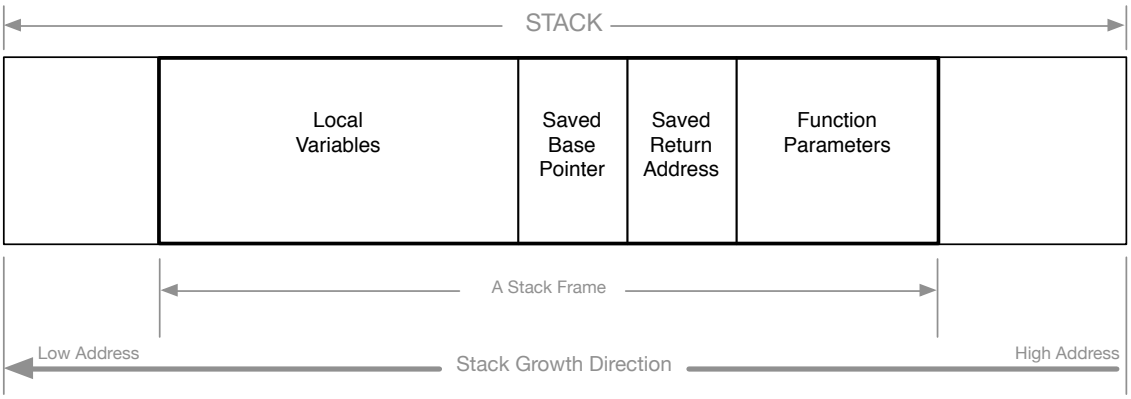


Figure 2.22: A Stack Frame of a Function on the Stack

In the same way, when returning to the calling program, SP is incremented by assigning the value of BP to SP. Then, the saved base pointer and the saved return address are popped off the stack into BP and IP, respectively. IP will point to the address of an instruction after the function call instruction. When a buffer overflow vulnerability occurs on the stack, malware is able to overwrite the saved return address with an arbitrary value as the stack grows downward, as depicted in Figure 2.23. If the number of bytes written into a local variable is more than the space allocated in memory for it, both of the saved base pointer and the saved return address will be corrupted in memory. Thus, IP will be assigned an arbitrary address controlled by malware, which usually points the address of shellcode. Execution may then be transferred to shellcode through manipulation of the instruction pointer (IP).

#### 2.4.1.2 Heap-based buffer overflow

The heap is a block of memory used for dynamic memory allocation at runtime and it grows upward. Similar to buffer overflow on the stack, heap-based buffer

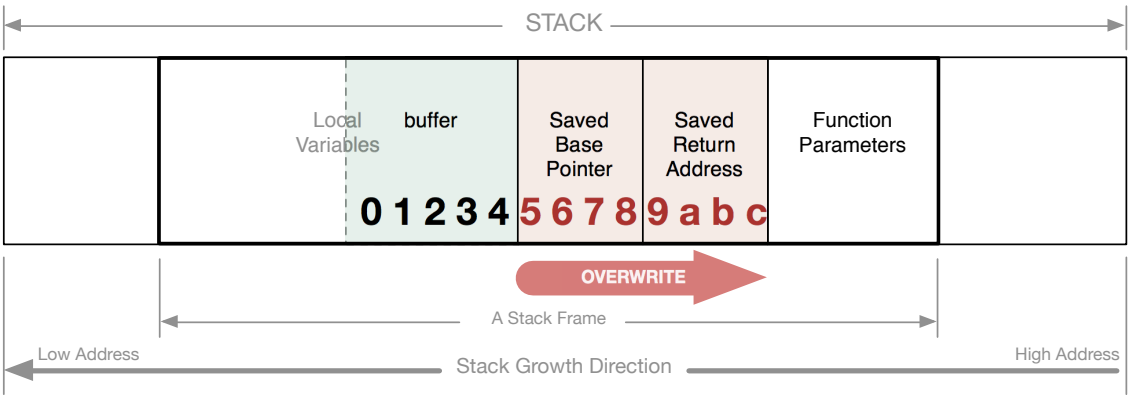


Figure 2.23: Stack-based Buffer Overflow

overflow occurs when a program writes more data than the allocated space into a vulnerable buffer on the heap [56]. Malware may also cause a buffer overflow by providing an arbitrary size for memory allocation on the heap. In Windows, every process has a default heap, and can also dynamically create new heaps and release these additional heaps, depending on memory needs at runtime [57]. In heap-based buffer overflow attacks, malware may corrupt the heap management structure [40] and adjacent objects containing a function pointer or a virtual table [113] on the heap in order to direct the execution flow to its malicious code.

Heap-based buffer overflow attacks that manipulate the heap metadata depend on the design and implementation of the targeted heap manager. Most operating systems typically provide heap memory management, and also applications sometimes include their own custom implementations for heap management. Each heap manager usually has its own vulnerabilities and exploitation techniques. For example, in Windows XP and 2000, heap management information, such as user dynamic data, is also stored on the heap. When heap-based buffer overflow occurs, one or more of the subsequent heap metadata sections can be overwritten. Therefore, malware may be able to write arbitrary data into arbitrary memory addresses by manipulating the heap management structure. Figure 2.24 illustrates a typical heap-based attack against the heap manager of Windows XP and 2000. If malware is able to overwrite a block and the metadata of the next block, it may write arbitrary data to an arbitrary location in memory [40].

In heap-based buffer overflow attacks, adjacent objects near a vulnerable buffer in memory may also be overwritten, such as C++ objects, as seen in Figure 2.25. C++ is a programming language that supports data abstraction, object-oriented,

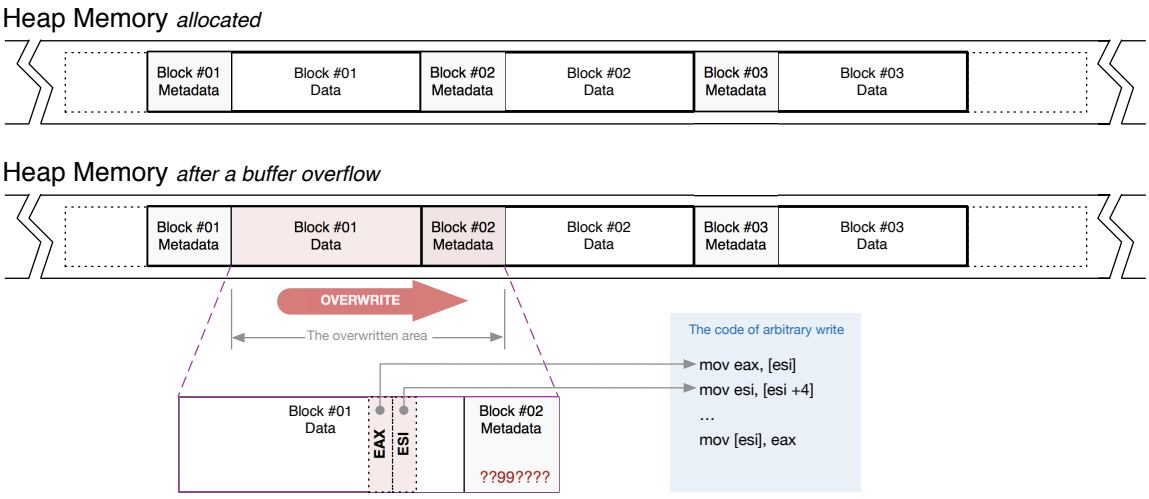


Figure 2.24: Heap-based Buffer Overflow - Heap Metadata

and generic programming [146, 145]. Runtime polymorphism is one of the most important features of C++. In runtime polymorphism, derived classes override the virtual methods of their parents, allowing method binding to happen at runtime. Thus, each instance of a C++ class that includes virtual methods has its own virtual function table, called vtbl, vftable or vtable. A virtual function table contains a list of function pointers to the virtual member methods of a class. The C++ standard does not define how a vtable is accessed by the instance of its owner class, so it depends on compiler implementation. A pointer that refers to the relevant vtable is added by the compiler into each instance of a class. This hidden pointer, called vpointer or vptr, is usually placed at the beginning of a C++ instance in memory [14], depending on the compiler. Since both this vtable structure and user data of a class are located in the same memory area, malware may corrupt vptr and vtable, or may even create a fake vtable. Overflowing into an adjacent object, such as a vptr and vtable, allows malware to transfer control flow to an arbitrary location in memory that malicious code relies on.

### 2.4.2 Integer overflow vulnerability

Integer overflows [13] occur when a value that is more than the maximum value of the integer type is stored into an integer variable. Writing values smaller than the minimum value also has a similar problematic effect, called integer underflow. Integer variables are used for storing real numbers, and there are fixed minimum and

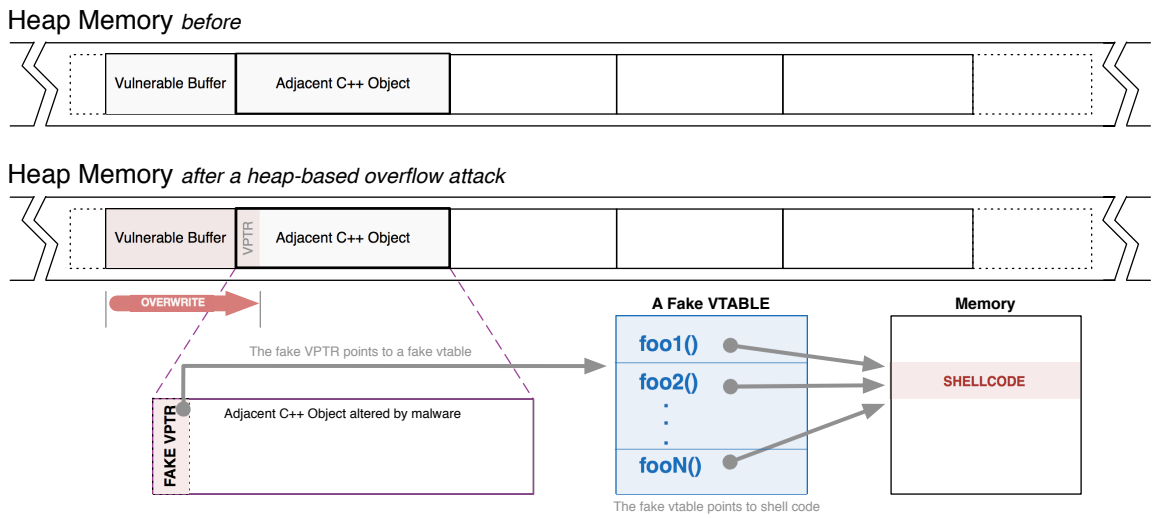


Figure 2.25: Heap-based Buffer Overflow - Adjacent Objects

maximum values storable in an integer variable. If a value smaller or greater than the ranges of an integer is provided for an integer variable, the provided value will be wrapped around by using modulo arithmetic, and so the variable will contain an unexpected value that may lead to abnormal situations during program execution. Integer overflow usually happens as a result of an arithmetic operation that falls behind or beyond the boundaries of an integer variable. As depicted in Figure 2.26, the maximum value for an unsigned short integer is 65535 ($2^{16} - 1$) in 32-bit systems. When 65550 is assigned to the unsigned short object buffer_size, it is implicitly converted from 65550 to 14. Thus, a smaller space than 65550 is allocated for vulnerable_buffer. Integer overflows may cause a buffer overflow when the affected integer variable is used as an array index or in memory allocation. In contrast to stack-based and heap-based buffer overflow, an integer overflow vulnerability may indirectly cause a memory corruption, although not all of them are exploitable.

```
unsigned short buffer_size = 65550;
char vulnerable_buffer[buffer_size];
. . .
. . .
memcpy(vulnerable_buffer,argv[1], 65550);
```

Figure 2.26: Example Code for Integer Overflow Vulnerability

### 2.4.3 Format string vulnerability

Format string vulnerabilities occur when the argument of a function in the printf family is directly used without a format string [118, 71, 159], demonstrated in Figure 2.27. In the C language, several standard library functions, which are defined in the ANSI standard, accept a format string as an argument specified for format and conversion operations [112]. These functions are called format functions, including printf, fprintf, and fwprintf. Format strings are employed for formatted input and output. Format strings allow C data types to be converted into string form. Ordinary characters in a format string are displayed directly. In addition to ordinary characters, a format string also consists of special characters for conversion specifications, called format specifiers, which specify how to interpret and format the data provided to the function. By providing a specific format string to a function, the same value can be printed in different formats, such as decimal, octal and hexadecimal. For example, when %x is used to specify an argument, it will be interpreted as an unsigned integer and displayed in hexadecimal format, whereas %d specifies an integer argument, which will be displayed in decimal format. When the function is called, both the format string and arguments of a function are stored on the stack. Therefore, the invoked function first processes the format string and then reads the corresponding argument from the stack for each format specifier in the format string. The number of arguments that are going to be read off the stack by the function is based on the number of format specifiers in the format string. However, if a format string is not provided to a function, one may pass a crafted argument including format specifiers into the function for exploitation by corrupting memory. Similar to buffer overflow vulnerabilities, format string vulnerabilities allow malware to read arbitrary locations in memory and also to inject malicious code.

| Safe Code | Vulnerable Code |
|---|---|
| `char my_name[256];`<br>`...`<br>`printf(“My name is %s”, my_name);` | `char my_name[256];`<br>`...`<br>`printf(my_name);` |

Figure 2.27: Example Vulnerable Code for Format String Vulnerability

Figure 2.28 demonstrates the stack layout for the safe code in the previous figure. When printf is invoked from the safe code, the address of the format string and argument will be pushed to the stack in reverse order. In the safe code demonstrated

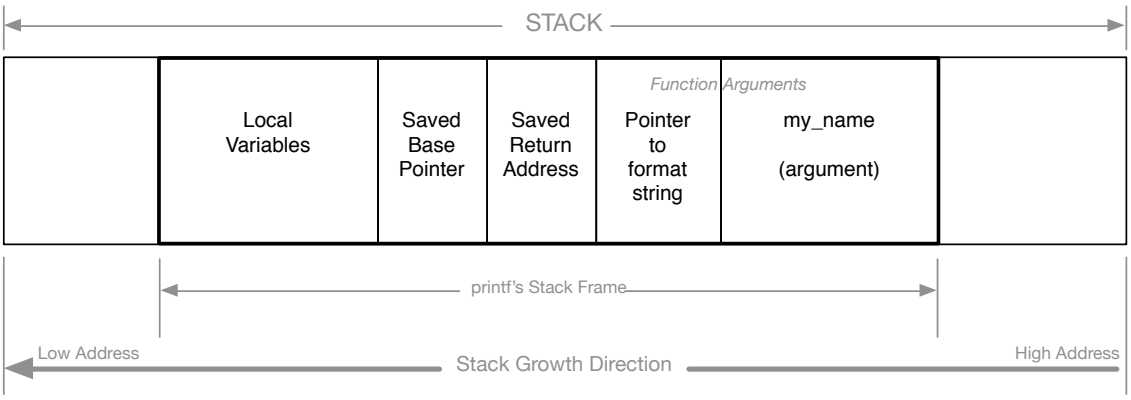


Figure 2.28: Stack Layout for Safe Code

in Figure 2.27, the format string includes a %s that is a format specifier corresponding to the argument my_name. Therefore, my_name will be interpreted as string and printf will display a string out combining "My name is" and the value of my_name. As indicated above, format specifiers may also be employed for exploitation. In a format string vulnerability attack, %x can be used to leak information from the stack, while %s allows reading of character strings from memory, and %n may be used to write arbitrary data to memory.

For example, in the vulnerable code demonstrated in Figure 2.27, there is no format string provided to the printf function so the argument my_name will be interpreted by the printf function as a format string. When printf is invoked in this vulnerable code, only the argument my_name will be pushed to the stack. If %x is stored in my_name, then printf will interpret it as a format specifier and it causes a memory leak by reading data located in the stack frame of the caller function. Since %x is used for displaying unsigned integers in hexadecimal format, printf will display a word above the saved return address in hexadecimal format, illustrated in Figure 2.29. A word is a fixed-size data unit used by processors and its size differs based on the processor design, such as four (4) bytes in 32-bit systems. Likewise, the %s format specifier corresponds for string formatting. If %s is stored in my_name, then printf will interpret a word above the saved return address as a pointer to a string and display bytes in the referenced memory area until reaching a Null byte, 0x00.

It is possible to access an arbitrary memory location using format specifiers. The %x format specifier reads a word each time so the format string "%x%x%x%x%x" shows five sequential words from the top of the stack. More generally, by using "%[number]$x" as a format string, the [number]th word of the stack can be read.

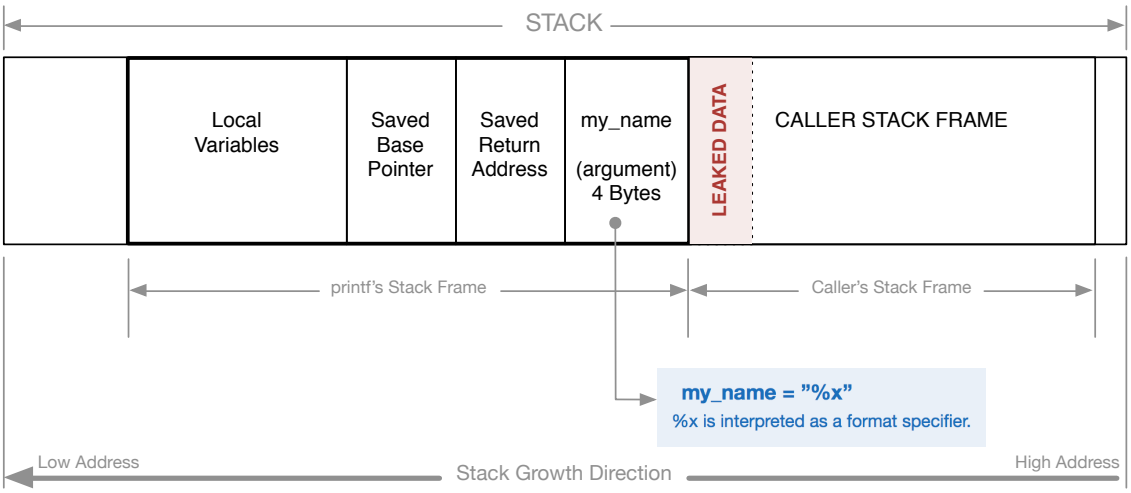


Figure 2.29: Stack Layout for Vulnerable Code

"%5$x" is a good example of reaching an arbitrary location on the stack, directly accessing the 5th word without taking any data off from the stack. Using similar techniques, malware may read the canary value, which protects the saved return address and the saved base pointer in a stack frame from buffer overflows. It may then generate a crafted string at runtime that can bypass canary protection by including the leaked canary value. It is even possible to access memory locations not on the stack by manipulating local variables in vulnerable code. Local variables will be placed in the caller's stack frame on the stack. Thus, the format string "[address]%[number]$s" dumps memory from a defined address, where the [number]th word of the stack stores the address value. For example, "\x40\xFA\xFF\xBF%10$s" provides the content of the memory address 0xBFFFFA40 because the first part "\x40\xFA\xFF\xBF" is stored on the stack as 10th word and the second part "%10$s" reads the content of 10th word from the stack the like reading a string, as seen in Figure 2.30.

Similar to both the %x and %s format specifiers, %n is another format specifier employed in the attacks. Typically, %n is used to write the number of characters formatted by the function so far in a signed integer so the argument provided to a printf-like function is usually a signed int pointer. In contrast to the other format specifiers, %n allows malware to write arbitrary data to memory, and so it may be used to cause malicious code execution. For example, let us assume that my_name includes "ABCDEFGH%n" in vulnerable code demonstrated in Figure 2.27. There are eight (8) characters before %n in "ABCDEFGH%n". Thus, printf will interpret a word above the saved return address as a pointer to a signed integer, and then

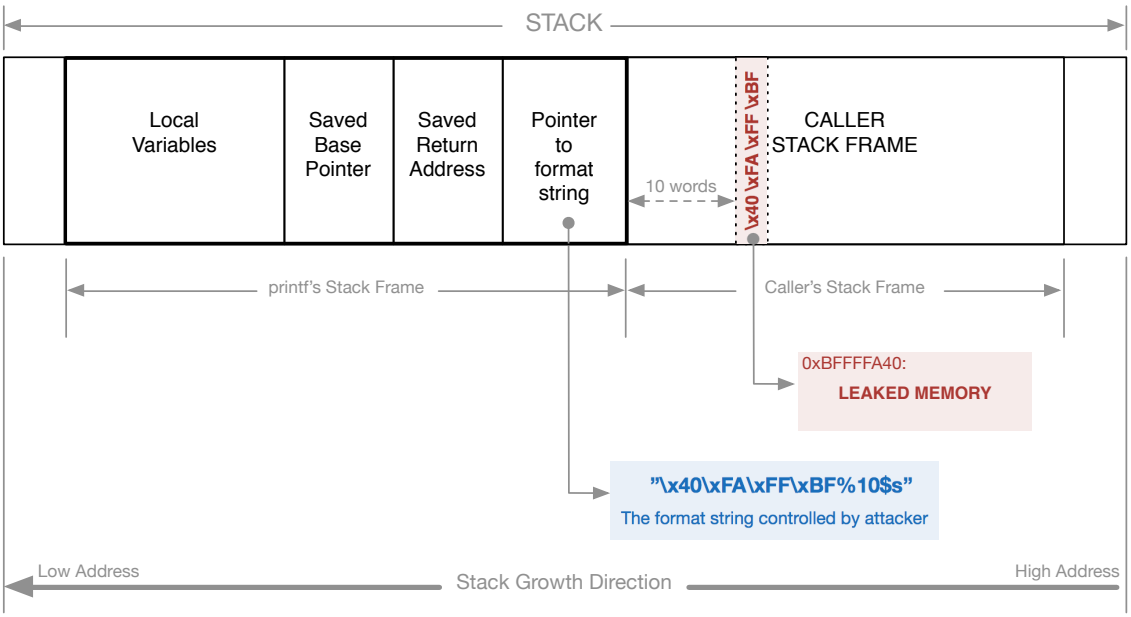


Figure 2.30: Reading Arbitrary Memory Locations

write eight (8) into the referenced memory. Similarly, the format string "%8u%n" has exactly the same effect on the stack because "%8u" occupies eight (8) characters, like "ABCDEFGH". Likewise, if the input controlled by attacker is stored in a local variable in vulnerable code, then it is possible to write an arbitrary value to an arbitrary memory location by providing a crafted format string. For example, the format string "\x40\xFA\xFF\xBF_%16u_%10\$n" modifies the value of 0xBFFFFA40 memory address as sixteen (16). This technique is very similar to the technique used for reading data from arbitrary memory locations in the format string attacks.

### 2.4.4 Use-After-Free vulnerability

Use-After-Free (UAF) vulnerabilities occur when a previously freed object in memory is still accessed by a program even after its deallocation, as demonstrated in Figure 2.31. The UAF vulnerabilities [4] are usually exploited in web-based attacks by malware in order to gain control of execution of a targeted web browser. Especially in C and C++, after an object has been created in memory a reference, called a pointer, is used to access the object. Briefly, pointers point to the locations of objects in memory. Every object has its own lifetime, and then it will be freed by deallocating its previously allocated memory. If the object in memory has been accidently deallocated, yet its pointer continues to point to the freed object, the resulting dangling

pointer may allow memory corruption giving malware the ability to perform arbitrary code execution. When a dangling pointer is dereferenced, it may be forwarded to a fake object controlled by malware. In UAF vulnerability attacks, web-based malware commonly overwrites a freed object with a fake malicious object to manipulate execution. Thus, after delivering a piece of shellcode at a predictable location in memory, malware usually forces the targeted browser to free a specific vulnerable object. Then, it creates a specially crafted object at exactly the same address space of the freed object, using a technique such as via heap spraying. For example, for C++ objects, malware may overwrite the freed object with a malicious object including a vtable pointer that refers a fake vtable created by malware. The malicious vtable usually contains a list of function pointers to shellcode, as illustrated in Figure 2.32. Therefore, when accessing the freed object, malware will be able to redirect execution to a piece of shellcode loaded at a predictable address.

```
MyObject *pVul = new MyObject();
. . .
. . .
delete pVul;
. . .
pVul->foo();
```

Figure 2.31: Example Code for Use-After-Free Vulnerability

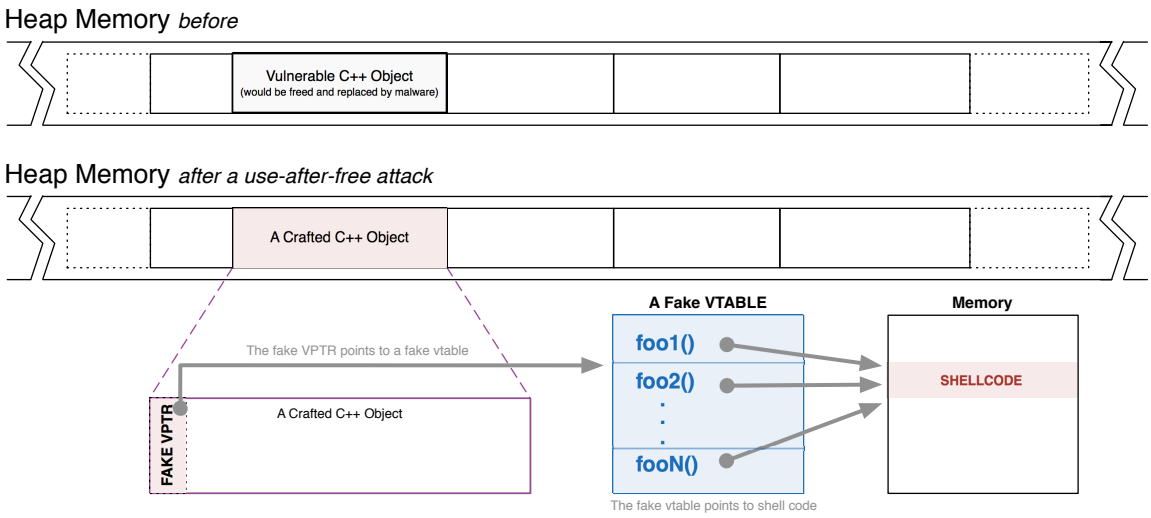


Figure 2.32: Attack to a Use-After-Free Vulnerability

## 2.5 Anti-Malware Techniques

Malware usually corrupts memory to exploit a vulnerability and execute malicious code. A significant number of memory protection techniques have been implemented in order to protect computer systems from malicious attacks [144, 82]. In this section we will explain several protection mechanisms, including DEP, ASLR, Stack Canary, SafeSEH, SEHOP, and heap-based security enhancements.

### 2.5.1 DEP - Data Execution Prevention

Data Execution Prevention (DEP) [91] is a memory protection mechanism that protects computer systems against the execution of code located in data regions in memory, including heap, stack, and memory pools. It is a combination of hardware-level protection and software-level protection. Hardware-level DEP support for an application may be enabled by using the "/NXCOMPAT" [94]. By marking memory pages that include data as non-executable, DEP only allows execution of code which is located in executable memory areas, and it prevents code execution from data pages, making exploitation more difficult. The first generation of buffer overflow malware had simply injected and then executed shellcode onto the stack or heap, which is typically used for storing data. In order to provide improved protection against malware, first AMD (in 2004) and then Intel included a page-protection feature in their chips for the x86 architecture, such as NX (No eXecute) bit for AMD and XD (eXecute Disable) bit for Intel. After this hardware improvement, Microsoft added NX/XD-bit support, called DEP, into the Windows operating system to service pack releases for Windows XP and Windows Server 2003 [6, 149]. In systems protected by DEP, when code is executed from a non-executable memory location, an exception is raised. Unless this exception is handled, the process possibly compromised by malware is terminated by the operating system.

Windows provides four different configuration options for DEP, including AlwaysOn, AlwaysOff, OptIn, and OptOut. The AlwaysOn configuration means that DEP provides full protection for the whole system, but the AlwaysOff configuration completely disables DEP protection system-wide. In both of theses configurations, protection cannot change at runtime. Alternatively, in the OptIn and OptOut configurations, DEP works at an application-based level so it can be selectively enabled and disabled for individual applications, and this can be altered at runtime. In the OptIn configuration, only the processes of applications that are opted in are protected

by DEP, whereas DEP protects all processes by default in the configuration OptOut, except applications that are opted out.

### 2.5.2 ASLR - Address Space Layout Randomization

Another memory protection mechanism, Address Space Layout Randomization (ASLR) protects process memory against malicious attacks by including entropy (randomness) into the memory addressing of mapped objects. When mapping objects into a virtual address space used by a process, placing them at random locations in memory makes it more difficult for malware to know the exact locations of interesting addresses during exploitation. Thus, ASLR relocates objects belonging to a process and loads them at different locations in memory at every execution. This includes executable and module images, heaps, stacks, as well as PEBs and TEBs.

For example, in ROP-based attacks, code chunks borrowed from loaded modules in memory are used to bypass DEP and execute arbitrary code. Therefore, malware must know the loading address of modules in order to execute the borrowed code chunks, called ROP gadgets, in place of shellcode. ASLR relocates executable and module images to obscure the current locations of loaded modules, which increases the unpredictability of ROP-gadgets locations. Similarly, ASLR also randomizes the location of heaps and stacks in memory to add randomness to their addresses. Malware usually injects shellcode to a predictable address on the stack and heap using various injection methods, such as heap-spraying. Thus, relocating the heap provides extra protection against heap spraying techniques. Other critical data structures mostly used by malware, such as the PEB and TEB are relocated by ASLR to decrease their predictability.

In Windows, ASLR support can be enabled at compile-time using the "/DYNAMICBASE" compiler option for an application. The first ASLR implementation was published in Windows Vista in 2006 [114, 154], Vista Beta 2. In this version, only images compiled with ASLR support were relocated. An additional feature called Force ASLR, started with Windows 8. This forces ASLR protection for every running process regardless of whether applications have been compiled with ASLR support. In Windows, 32-bit applications have better entropy than 64-bit applications, and since Windows 8, High Entropy ASLR (HEASLR) also provides less predictability by significantly increasing the number of entropy bits, which makes the address space wider.

### 2.5.3 Stack-based Buffer Overflow Protection

Stack-based memory vulnerabilities are one of the most serious memory corruption types exploited by malware. Several hardening improvements have been implemented in order to complicated stack-based exploitation, including stack canaries, variable reordering, SafeSEH, and SEHOP.

#### 2.5.3.1 Stack Canary and Variable Reordering

A Stack Canary is a protection mechanism that detects buffer overflows at runtime by inserting a special value between the local variables of a function and the state information of a function call on the stack [30]. It is a compile-time protection and can be enabled using the "/GS" guard stack compiler option in Windows [92]. This compiler option also provides variable reordering, which alters the stack frame layout in order to detect variable overwriting. In a stack-based buffer overflow, an exploit typically overwrites the saved base pointer, return address, and function variables on a function stack frame with a crafted address value pointing to shellcode. String variables are located at higher addresses than other variables by reordering and a canary value is placed on the stack for each function call in order to protect the relevant saved return address and the saved base pointer. A canary is a random four-byte value that is checked before function return. When a stack-based buffer overflow happens, the canary will be overwritten as well, as shown in Figure 2.33. Thus, if the canary value is corrupted, then the compromised process will be terminated by the operating system as the verification has failed. Canary protection that monitors buffer overflows is also called security cookies in Windows, and is included to the Windows operating systems within Windows 2003 Server.

#### 2.5.3.2 SEH - Structured Exception Handling Protection

Another type of the stack-based buffer overflow attacks is achieved by corrupting the Structured Exception Handling (SEH) [72]. Therefore, the SafeSEH and SEHOP protections have been developed by Microsoft in order to protect the SEH structure in memory against overwriting attacks. SEH is the integral exception handling mechanism of Windows for both hardware and software exceptions [96, 104]. Unexpected, exceptional events rarely occurring at runtime, such as divide-by-zero, are called exceptions and they require a special treatment other than the normal flow of execution. Thus, during execution, when an exception is raised in kernel-mode or

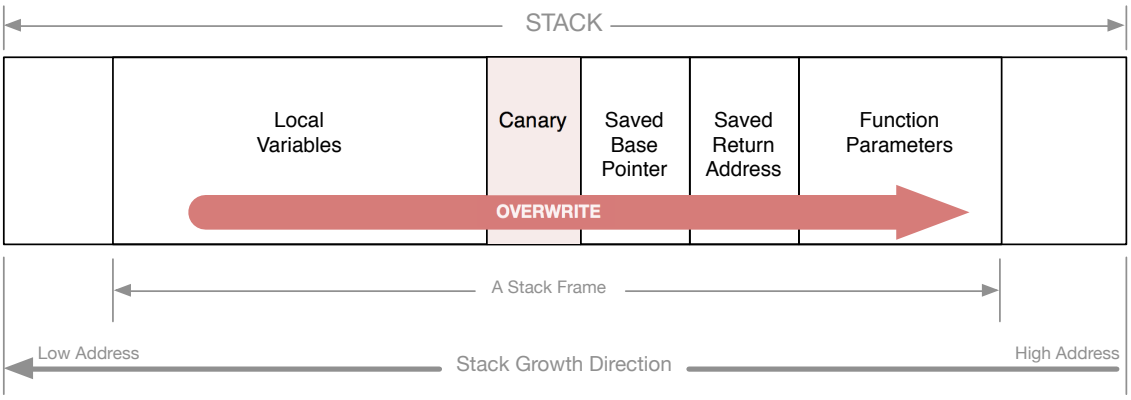


Figure 2.33: Stack-based Buffer Overflow Protection

user-mode code, the code flow is transferred to handling code specific to each exception type. Each exception handler is dedicated to a particular function of process. Exception handlers are stored on the stack in the Exception_Registration structure which is a linked list, as illustrated in Figure 2.34. In SEH overwrite attacks, an exploit overwrites the exception handlers chain on the stack with an arbitrary address value pointing to shellcode.

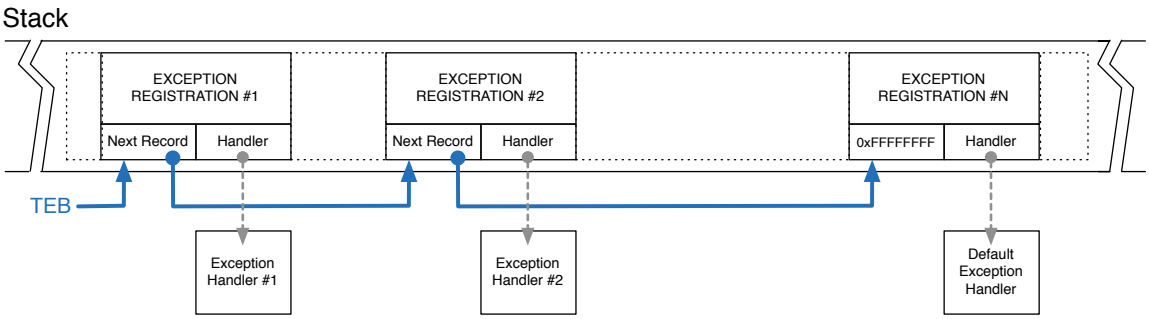


Figure 2.34: Exception_Registration Structure

SafeSEH is a protection mechanism against modification of an exception handler records on the stack, introduced in 2003. It is another compile-time protection and can be enabled using the "/SafeSEH" compiler option in Windows [95]. An executable protected by SafeSEH contains a special control table including its all valid exception handlers for verification at runtime. When an exception occurs, exception dispatcher code checks whether the exception handler record on the stack for the exception is valid by using the SafeSEH table. Otherwise, the process compromised by overwriting the exception handler record will be terminated by the dispatcher code.

Another memory protection mechanism in Windows is SEH Overwrite Protection (SEHOP), which prevents the exploitation of SEH by verifying the integrity of the SEH linked list on the stack [150]. SEHOP, which was introduced in 2008, is a runtime protection, and it can be enabled for every application regardless of compiler options used to compile it. The special registry subkey "DisableExceptionChainValidation" is used to enable and disable it [148]. SEH is in a linked list in memory. When SEHOP is enabled, the last exception registration record of the SEH linked list points to the ntdll.FinalExceptionHandler function. In order to detect a corruption in the SEH linked list, the exception dispatcher checks whether the last record refers ntdll.FinalExceptionHandler by traversing through the SEH linked list. When reaching the end of the SEH linked list, if the last record does not point the ntdll.FinalExceptionHandler function, the exception dispatcher terminates the compromised process. This will stop any SEH overwriting attack detected by SEHOP.

### 2.5.4 Heap-based Buffer Overflow Protection

Heap-based memory vulnerabilities are another type of serious memory corruption targeted by malware for exploitation, similar to stack-based vulnerabilities. In order to protect objects on the heap and detect inconsistency of heap management data, various heap protection mechanisms have been implemented and added to the Windows operating system starting with Windows XP SP2. These include safe unlinking checks, safe lookaside lists, virtual table guards (vtguard), guard pages, heap entry cookies, and heap metadata encryption [151, 155, 82].

Early heap malware traditionally corrupted the heap management data by overwriting chunk pointers with arbitrary values to gain control flow and execute arbitrary code. The safe unlinking check is designed to mitigate this type of attack. The heap management data for free chunks is stored in a doubly linked list structure on the heap. In the safe unlinking check, before unlinking a chunk, the backward and forward pointers of its neighbors are verified to see whether they point to the same chunk. Similar to safe unlinking, heap metadata encryption and heap cookies are used to protect heap management data. In particular, a heap cookie is inserted into the header of each heap entry in the heap metadata, and before unlinking a heap entry from the list, the 8-bit random-valued cookie added its header is validated to thwart heap exploitation. Likewise, the Lookaside List, a singly linked list, was used as the front-end heap allocator in early Windows versions, yet it had led several vital

security issues [7, 24], with attacks that overwrite the heap management data. In response, the Lookaside List has been replaced with the Low Fragmentation Heap (LFH) [93] in Windows Vista.

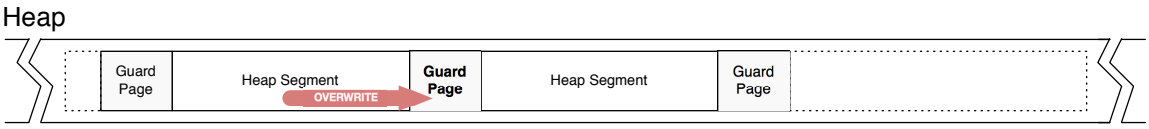


Figure 2.35: Guard Pages as Heap-based Buffer Overflow Protection

Due to security improvements in Vista which protect heap management data, corrupting application-specific data on the heap has become one of the most used techniques in the heap-based attacks [155]. For example, in order to gain control flow, heap malware also overwrites critical objects on the heap, such as virtual table pointers (vptr) and virtual function table (vtable). Therefore, virtual table guards (vtguard) have been implemented to protect the vtables of C++ objects on the heap by checking a secret entry added into the vtable of a C++ object, before performing virtual function call each time. Another heap protection is the use of guard pages since Windows 8, by placing guard pages around heap segments and subsegments. During a heap-based buffer overflow attack, if malware overwrites too much data in memory and tries to write on a guard page, an exception is raised, and hence the attack will be detected, as depicted in Figure 2.35.

# Chapter 3

# Techniques For Bypassing Behavioral-based Protections

In this chapter, we will consider *unhooking* and *funneling* to bypass behavioral-based prevention methods. We have chosen two enterprise level antivirus products for our tests, Microsoft EMET and McAfee HIPS, and we have used the Metasploit framework to obtain malicious files that exploit the some vulnerabilities of both the Windows operating system and its browser, Internet Explorer. Original malware files, which have been generated by Metasploit, are generally well-known by most anti-malware products in commercial and open software product levels. In our experiments, we have recoded the malicious payloads of the original generated malware files and have added a new malicious payload part that disables AV detection, prevention and protection methods on the fly.

In Chapter 3, the first section describes the main working principals of the evasion techniques. Section 3.2 briefly describes the Metasploit Framework. The following two sections 3.3 and 3.4 provide detailed information on Microsoft EMET and McAfee HIPS. Section 3.5 discusses some evasion techniques against security mitigations and their implementations for both Microsoft EMET and McAfee HIPS. In the following chapter, Chapter 4, the implementations of the unhooking and funneling methods are explained.

## 3.1 Description of the Evasion Techniques

### 3.1.1 The Unhooking Method

In the unhooking method, malware first patches all system functions hooked by anti-malware by replacing the hooking instructions with originals in order to remove inline-hooks added for monitoring critical system activities. After the unhooking process, behavioral anti-malware protection is deactivated, so any malicious payload can be executed by malware without triggering anti-malware protections. Unhooking malware includes original opcodes and their offsets in a table for patching. The code sections of executables and modules may be different in each release of software. Thus, unhooking malware is designed for specific systems. Due to possible changes in different versions, unhooking malware would also require maintenance to keep the original code table up to date after every operating system upgrade. However, anti-malware version changes slightly affect the unhooking method.

The development process of unhooking malware involves the following tasks:

- Task 1) Finding critical functions hooked by targeted anti-malware product(s),
- Task 2) Finding critical data structures monitored by targeted anti-malware product(s),
- Task 3) Finding the offsets of critical locations in modules,
- Task 4) Generating an original code table including the original code of modified parts of critical functions, and their beginning offsets,
- Task 5) Discovering a vulnerability in the targeted system and its exploitation technique,
- Task 6) Designing malware,
- Task 7) Implementing exploitation code,
- Task 8) Implementing antivirus-aware code that removes anti-malware protections, such as inline hooks and guard pages, for bypassing,
- Task 9) Implementing malicious payload code.

Patching hooking code in memory is a technique known by anti-malware and malware developers [60]. The development tasks of unhooking malware are accomplished manually, and they are time-consuming. In particular, tasks 1, 2, and, 3 require a deep technical knowledge of system programming for reverse engineering.

In this thesis, we introduce a just-in-time debugging tool, a *pintool*, that automatically discovers critical functions hooked by anti-malware and lists the offsets of critical locations. The debugging tool also reports the original and modified opcodes in the changed parts of hooked functions in memory, and it provides these altered code parts in assembly, as depicted in Figure 3.28 in Chapter 3. The generation and coding process needed to produce an original code table, task 4, can also be automated by adding a new feature to this pintool in the future. We have developed our tool using the Intel PIN dynamic instrumentation framework [48].

Thus, the unhooking method that we introduce is a new approach that automates task 1 and task 3 using a debugging pintool that provides useful information employed for developing malware, including offsets, original, and modified opcodes.

A piece of unhooking-enhanced malware consists of instructions in HTML, Javascript code, ROP gadgets, and hex opcodes. During an attack, unhooking malware performs the following actions in order:

- In HTML and Javascript code;
  - (Optional) Generating a malicious ROP chain of shellcode dynamically at runtime if necessary, depending on the vulnerability;
  - Delivering shellcode into the virtual address space of a victim application;
  - Triggering a vulnerability of the victim application;
- In the ROP (Return Oriented Programming) payload using ROP gadgets;
  - Bypassing the Data Execution Protection (DEP) of the running operating system;
    - Finding the base address of the ntdll module, which provides functionality for changing memory access protection (ntdll.ZwProtectVirtualMemory);
    - Calculating the address of a ntdll function to invoke it as if the function is the ntdll.ZwProtectVirtualMemory function;
    - Changing the access protection of the heap to executable by performing a malicious function call emulation technique using a "ntdll.ZwProtectVirtualMemorylike" function to eliminate DEP, allowing execution of malicious code on the heap;
- In the Hex-code payload

- Bypassing protections of the running operating system and anti-malware product;
  - Finding the base addresses of critical modules monitored by anti-malware, such as kernel32, kernelbase, and ntdll;
  - Changing the access protection of the critical modules to writable by performing the same malicious function call emulation technique;
  - (Optional) Removing the protections of anti-malware on critical modules, such as Guard Pages, by performing the same malicious function call emulation technique if necessary, depending on the anti-malware's protections;
- Disabling the functionality of the anti-malware product using the unhooking method;
  - Overwriting hooking code of the critical functions with the corresponding original code to patch them; an original code table embedded in shellcode is used for overwriting;
- Executing the rest of its malicious payload in hex.

### 3.1.2 The Funneling Method

In the funneling method, malware deactivates anti-malware by modifying its main control code in memory, instead of removing hooks. After disabling the anti-malware detection mechanism, anti-malware hooks will be nonfunctional; therefore, any malicious payload can be executed by malware while remaining undetectable by anti-malware. Funneling malware exploits the inline-hooking instructions in critical functions in order to discover the location of the anti-malware's main control function. Technically, funneling malware does not patch anti-malware hooks on critical functions one by one so it does not include an original code table. It can work against any system, although there are differences in code between the different versions of operating systems and modules. Thus, funneling malware is smaller than unhooking malware and the funneling method requires less effort for malware development and maintenance. Funneling malware may require an update only after major changes to anti-malware code. Small modifications in anti-malware code, such as new hook additions, will not affect the functionality of funneling malware.

The development process of funneling malware involves the following tasks:

- Task 1) Finding critical functions hooked by targeted anti-malware product(s),
- Task 2) Finding critical data structures monitored by targeted anti-malware product(s),
- Task 3) Finding the offsets of critical locations in modules,
- Task 4) Developing malicious techniques that deactivate anti-malware product(s) by reviewing anti-malware code injected in the virtual address space of a targeted process using reverse engineering methods,
- Task 5) Discovering a vulnerability in the targeted system and its exploitation technique,
- Task 6) Designing malware,
- Task 7) Implementing exploitation code,
- Task 8) Implementing antivirus-aware code that removes guard pages and alters anti-malware code in memory to disable its functionalities for bypassing,
- Task 9) Implementing malicious payload code.

The funneling method is a new evasion technique, in which finding critical functions and data structures, task 1 and task 3, is automated using the just-in-time debugging tool, similar to the unhooking method.

A piece of funneling-enhanced malware consists of instructions in HTML, Javascript code, ROP gadgets, and hex opcodes. The actions performed by funneling malware during an attack do follow below:

- In HTML and Javascript code;
  - (Optional) Generating the malicious ROP chain of shellcode dynamically at runtime if necessary, which depends on the vulnerability;
  - Delivering shellcode into the virtual address space of a victim application;
  - Triggering a vulnerability of the victim application;
- In the ROP (Return Oriented Programming) payload using ROP gadgets;
  - Bypassing DEP protection of the running operating system;
    - Finding the base address of the ntdll module;
    - Calculating the address of a NTDLL function to invoke it as if the function is the ntdll.ZwProtectVirtualMemory function;
    - Changing the access protection of the heap as executable by performing a malicious function call emulation technique using

the "ntdll.ZwProtectVirtualMemorylike" function to eliminate DEP and execute its malicious code on the heap;

- In the Hex-code payload;
  - Bypassing protections of the running operating system and anti-malware product;
    - Finding the base addresses of critical modules monitored by anti-malware, such as kernel32, kernelbase, and ntdll;
    - Changing the access protection of the critical modules as writable by performing the same malicious function call emulation technique;
    - (Optional) Removing the protections of anti-malware on the critical modules, such as Guard Pages, by performing the same malicious function call emulation technique if necessary, which depends on anti-malware's protections;
  - Disabling the functionality of anti-malware product using the funneling method;
    - Finding the address of anti-malware code by reviewing hooking instructions in memory
    - Changing the access protection of the memory area of anti-malware code to writable by performing the same malicious function call emulation technique;
    - Altering anti-malware code in memory to deactivate it;
  - Executing the rest of its malicious payload in hex.

### 3.1.3 New Malicious Function Call Emulation Technique

In this thesis, we introduce a new evasion technique used for invoking critical functions monitored by anti-malware employing inline hooking. This technique exploits two weaknesses in order to perform a system call: 1) only monitoring a limited number of critical functions and 2) code similarity between various ntdll functions in memory, allowing it to bypass inline hooking protection supported with Deep Hooks and Anti-Detour features. Instead of invoking a critical function hooked by anti-malware, the malicious function call emulation employs another ntdll function that is very similar in code, yet unhooked and not protected by anti-malware.

The lax monitoring of system functions allows malware to exploit non-critical system functions, and it decreases the effectiveness of other protection methods, such as Anti-Detour.

In both the *unhooking* and *funneling* methods, the access protections of critical memory areas are manipulated by malware in order to perform code modification in code segments and code execution on the heap or stack. Both *unhooking* malware and *funneling* malware use the malicious function call emulation technique in order to call the ntdll.ZwProtectVirtualMemory function for memory manipulation.

The implementation details of the function call emulation technique will be provided in Section 4.2.2.2.1.

### 3.1.4 New Evasion Technique Against Protections Based on Guard Pages to Monitor Critical Memory Regions

In this thesis, a new evasion technique is introduced that deactivates anti-malware protections employing guard pages to monitor critical data structures and critical functions in memory. This technique exploits the weakness of guard page protection methods. Typically, in Windows, an exception is raised by the guard page mechanism only if data is read and written, or code is executed in a memory area within the guard page, whereas altering the access protection of a memory area within the guard page does not trigger any alarm, which leads a security hole that allows malware to remove guard pages.

Anti-malware creates several guard pages for sensitive memory regions, on which critical data structures and functions are located, in order to detect malicious activities. The evasion technique unsets the PAGE_GUARD values of the protected memory regions by invoking the ntdll.ZwProtectVirtualMemory function to disable the guard pages.

Its implementation details will be provided in Sections 3.3.3 and 3.5.1.2.

### 3.1.5 Implementation of the Unhooking and Funneling Methods

Today, cyber criminals do not only install their own web servers, but also compromised vulnerable websites, to inject their malicious contents and spread them through the Internet. In order to evaluate the security of behavioral-based dynamic anti-malware

techniques in user space, we have conducted several web-based attacks as part of our research. These web-based attacks simulate a real-world scenario in which an adversary maintains a malicious web service and serves harmful web pages, and a victim opens such a crafted web page surfing on the Internet while using a vulnerable Internet browser.

In the experiments, the crafted web pages have been designed to exploit vulnerabilities in a targeted Windows computer system to deliver a malicious payload armed with *unhooking* and *funneling* methods. In the attacks, 32-bit Internet Explorer version 8 running on a 64-bit Windows 7 operating was the targeted browser. This is a configuration with well-known and serious security holes. Protection of the target systems is provided by two leading real-world antivirus solutions, Microsoft EMET and McAfee HIPS, that offer very effective mitigation using behavioral-based anti-malware techniques. Evading the current versions of these widely-used, real-world antivirus solutions is important in order to demonstrate how effective our methods are when in the evaluation defensive antivirus technologies.

We have designed and implemented several new variants of known malware using our *unhooking* and *funneling* evasion methods, in Ruby and Assembly. In order to develop new malware variants for test purposes, we have employed the Metasploit Framework. This tool provides both a development and an attack environment. We have created new malware variants from known and detectable originals provided by the Metasploit Framework. The new malware variants are able to bypass enterprise level anti-malware products that use behaviorally based detection techniques. Our methodology to generate a new malicious variant bypassing protections is as follows:

1. we have taken an exploit module of the Metasploit Framework that is detectable by anti-malware products;

2. we have modified the detectable Metasploit exploit module, by implementing the *unhooking* and *funneling* methods in order to evade protections provided by the targeted anti-malware products;

3. finally, we have combined the new improved exploit module with a few Metasploit payload modules, which may be done without any code modifications.

Developing our own malicious techniques as an exploit module of Metasploit Framework has enabled us to insert any Metasploit payload module into new malware variants with both *unhooking* and *funneling* functionalities.

## 3.2 Metasploit Framework

In our research, the Metasploit framework [111] has been used to produce malware files for the Windows operating system and its browsers. The Metasploit framework is one of most preferred supplementary software tools for penetration tests. As a versatile penetration testing framework, it can be used for task automation, information gathering, vulnerability scanning, custom tool creation and exploit writing. Many security researchers and penetration experts employ the Metasploit framework to discover vulnerabilities in their systems which can be targeted and exploited in a possible malicious attack. It is also used for simulating real-world attacks in order to assess the performance of running defence infrastructures. Knowing what attacks might happen in advance and taking necessary precautions against these threats in a timely way are absolutely vital for information security professionals. Metasploit users can choose to use the significant number of modules which are currently available in the Metasploit database, or to develop custom exploits leveraging Metasploit's development environment.

This de facto penetration testing framework [111] contains one of the largest exploit databases, currently consisting of more than 1300 exploits and 2000 complementary modules. An exploit module is a small program, especially designed to access a target system by abusing a specific vulnerability, and to deliver its malicious payload to the vulnerable target system. Metasploit modules extend the framework functionality. Depending on module types and their purposes, the comprehensive library of Metasploit can be categorized into exploit modules, auxiliary modules, encoder modules, nops modules, payload modules, and post-exploitation modules. Programmatically, exploits are also considered as modules in the Metasploit framework too. The main difference between the exploit and auxiliary modules of the Metasploit framework is that exploit modules use payload modules but auxiliary ones do not. A payload module is the main attack code used to damage or compromise the target system. Payload modules can be non-staged or staged so that there are three main types of payload modules. Single payloads are monolithic and work as stand-alones within the full attack code. Thus, a single payload module may have size problems and therefore cannot support every exploit module. However, staged payloads combine stager and stage modules. During attack executions, stages are downloaded by stagers which establish a remote communication between victim and attacker computers. Therefore, while stage modules avoid size problems, they may have stability issues

[135]. The Metasploit Framework can eliminate specified bad characters, like the null byte (00h), by using encoders when generating payload code in hex. It automatically selects the best-matched encoder to produce bad-character-free malicious code. An encoder module and its iteration number can also be manually chosen by user. The size of encoded payload code can be increased or decreased, depending on the encoder used and iteration number applied. The output size varies with different encoders and each iteration increases the size of final code produced, and also changes the code's appearance. An alteration in the code makes its signature different, which affects the effectiveness of signature-based protections. The Metasploit framework comes with various encoders and nop modules. Nop modules are used for generating NOP slide code, to increase the stability of payloads and keep payload sizes consistent, like padding does. Advanced access to and further information about the target system can be obtained by using post-exploitation modules. Privilege escalation, enabling remote desktop and pivoting, are features of the Metasploit framework providing good examples that bear out the powerful abilities of post-exploitation modules. Being able to provide different options to its modules, such as IP addresses, port numbers, and executable names extends modules' flexibilities and effectivenesses. Users can combine the different types of Metasploit modules when conducting an attack to create effective malware code.

In addition to its comprehensive workspace, the Metasploit framework also provides users with a productive exploit development environment to build upon and utilize for their custom exploits. The Metasploit framework has a very intuitive file system directory structure and its main components, modules, and configuration files are located in different directories with appropriate names, including data, documentation, lib, modules, plugins, scripts, and tools. For example, its actual modules are located in the "modules" directory, whereas the "lib" directory includes its base framework code. Also, user specified modules can be ideally placed under the " /.msf4/modules/" directory [135]. The Metasploit framework has been fully developed in the Ruby programming language by Rapid7 so it depends on various Ruby libraries as well. Depending on what users and researchers need, miscellaneous kinds of new custom exploits can be developed faster in Ruby, leveraging Metasploit libraries, and then easily integrated by users into its extensive exploit database. In the Metasploit development environment, libraries such as Rex, Msf::Core, Msf::Base, give significant flexibility to security researchers to implement their own proof of concept malware code in a reasonable time. Msf::Core and Msf::Base provide framework

APIs, while Rex is a library for basic common tasks, including Base64, XOR, SSL, and Unicode.

A working exploit for newest or zero-day vulnerabilities is usually impossible to find in the Metasploit database, and in this case, writing a customized exploit would be a must. Exploit developers can simply modify a similar exploit or module available or can develop their own from scratch, depending on their testing scenarios. Using available libraries, plugins, and mixings in the Metasploit framework makes the development process faster and easier.

## 3.3 Microsoft EMET - The Enhanced Mitigation Experience Toolkit

EMET [80] is a state-of-the-art security software system by Microsoft, which protects computer systems against cyber attacks regardless of whether the attacks have been addressed by anti-malware software yet or not. The reason we chose Microsoft EMET for testing against our evasion methods is that EMET implements behaviorally-based detection techniques. Also, it has been installed on both home and enterprise computers and works on a wide range of Microsoft Windows operating systems as well [80, 81]. EMET promises that it can detect and stop even new undiscovered threats by using its advanced security mitigation technologies.

### 3.3.1 Microsoft EMET - Technical background

Microsoft EMET mitigations can be system wide or application based. The system wide mitigations can be enabled for all processes running on the protected computer, while per-process-based mitigations can be applied individually on any application. The current version of EMET, version 5.52, includes the following security mitigations [80, 81].

System Wide Mitigations:

- Data Execution Prevention (DEP) Security Mitigation
- Structured Exception Handling Overwrite Protection (SEHOP) Security Mitigation
- Mandatory Address Space Layout Randomization (ASLR) Security Mitigation
- Windows 10 Untrusted Fonts

Application Based Mitigations:

- Data Execution Prevention (DEP) Security Mitigation
- Structured Exception Handling Overwrite Protection (SEHOP) Security Mitigation
- NullPage Security Mitigation
- Heapspray Allocation Security Mitigation
- Mandatory Address Space Layout Randomization (ASLR) Security Mitigation
- Export Address Table Filtering (EAF) Security Mitigation
- Export Address Table Filtering (EAF+) Security Mitigation
- Bottom Up ASLR Security Mitigation
- Load Library Check - Return Oriented Programming (ROP) Security Mitigation
- Memory Protection Check - Return Oriented Programming (ROP) Security Mitigation
- Simulate Execution Flow - Return Oriented Programming (ROP) Security Mitigation
- Stack Pivot - Return Oriented Programming (ROP) Security Mitigation
- Caller Checks - Return Oriented Programming (ROP) Security Mitigation
- Attack Surface Reduction (ASR) Mitigation
- Windows 10 Untrusted Fonts

Microsoft EMET provides the following extra mitigations to applications on which one of the ROP security mitigations is configured.

- Deep Hooks - Additional Return Oriented Programming (ROP) Security Mitigation
- Anti Detours - Additional Return Oriented Programming (ROP) Security Mitigation
- Banned Functions - Additional Return Oriented Programming (ROP) Security Mitigation

Descriptions of security mitigations in the latest version, 5.52, are given below [81].

**Data Execution Prevention (DEP) Security Mitigation:** Microsoft EMET applies DEP protection on all processes, even though an application is not compiled with a DEP flag. In DEP protection, data areas in memory, such as the stack and heap, are marked as read- and write-only, not executable. Therefore, Microsoft EMET prevents malware from running shellcode injected on the stack or heap. This is a system-wide and application-based mitigation and works on both 32-bit and 64-bit architectures.

**Structured Exception Handling Overwrite Protection (SEHOP) Security Mitigation:** Microsoft EMET provides SEHOP protection, including Windows versions since Vista SP1, to old version Windows systems. SEHOP protection validates the consistency of the exception handlers' chain before every exception handler call by the operating system [150]. Therefore, it disallows control flow manipulation via the SEH overwritten exploitation method, which is caused by overwriting the Structured Exception Handler (SEH) with a malicious value by malware in order to forward execution flow to the malware's shellcode. SEHOP mitigation is a both a system-wide and application-based mitigation and it is used only for protection of 32-bit processes, not 64-bit ones.

**Mandatory Address Space Layout Randomization (ASLR) Security Mitigation:** Microsoft EMET provides ASLR protection for applications by randomizing the loading addresses of their modules, even ones that are not ASLR compatible. Still, being compiled with a relocations section is a must [1] for this mitigation. It prevents malware from using ROP gadgets by preventing prediction of module locations. This is a system-wide and application-based mitigation and applicable with both 32-bit and 64-bit processes.

**Windows 10 Untrusted Fonts:** Microsoft EMET blocks untrusted fonts, which are located outside of the Windows font directory, "%windir%/Fonts" to prevent both remote and local attacks and it disallows possible parsing of a crafted font file. This is a system wide and application-based mitigation and applicable within both 32-bit and 64-bit processes, but only available on Windows 10.

**NullPage Security Mitigation:** Microsoft EMET protects against potential null dereference issues in user mode, even though there is no known exploitation

method at present. This is an application-based mitigation and works on both 32-bit and 64-bit architectures.

**Heapspray Allocation Security Mitigation:** Microsoft EMET pre-allocates memory addresses on the heap, which are commonly used in heapspray attacks, which spread out many copies of some shellcode on the heap. Therefore, it prevents malware from injecting its shellcode into these memory areas in order allow execution at a predictable and specified location. This is an application-based mitigation only, and applicable with both 32-bit and 64-bit processes.

**Export Address Table Filtering (EAF) Security Mitigation:** Microsoft EMET monitors the Export Address Tables (EATs) of kernel32.dll, kernelbase.dll, and ntdll.dll to determine whether a read or write operation is valid. Thus, it prevents malware from obtaining critical function addresses by reading these three EATs. This is an application-based mitigation and applicable with both 32-bit and 64-bit processes.

**Export Address Table Filtering (EAF+) Security Mitigation:** This is an extension of the EAF security mitigation. Microsoft EMET checks whether the stack pointer is within stack boundaries, and matched with the frame pointer. Also, Microsoft EMET monitors read accesses to EATs of kernel32.dll, kernelbase.dll, and ntdll.dll from user specified modules and it tracks read accesses to the PE Header of user-specified modules. This is an application-based mitigation and applicable with both 32-bit and 64-bit processes.

**Bottom Up ASLR Security Mitigation:** Microsoft EMET randomizes the base address of bottom-up allocations such as heaps, stacks, and other memory allocations, with an 8-bit entropy. This is an application-based mitigation, and applicable with both 32-bit and 64-bit processes.

**Load Library Check  Return Oriented Programming (ROP) Security Mitigation:** Microsoft EMET monitors every load operation by hooking critical functions related with library loading, such as kernel32.LoadLib. Thus, it disallows the loading of a module from a UNC path or a remote address. Only module-loading operations from local storage are considered benign. This is an application-based mitigation, and works for both 32-bit and 64-bit processes.

**Memory Protection Check  Return Oriented Programming (ROP) Security Mitigation:** Microsoft EMET disallows setting of the access protection of a memory region on the stack or heap as executable, as malware generally changes protections to be able to execute its shellcode located in these data areas. This is an application-based mitigation, and applicable with both 32-bit and 64-bit processes.

**Simulate Execution Flow  Return Oriented Programming (ROP) Security Mitigation:** Microsoft EMET monitors critical function calls and simulates execution in advance to detect ROP attacks. For example, a regular return address is preceded by a call instruction; otherwise it will be a suspicious execution. It disallows possible suspicious executions. This is an application-based mitigation, and works only for 32-bit processes, not 64-bit ones.

**Stack Pivot  Return Oriented Programming (ROP) Security Mitigation:** Microsoft EMET checks the stack pointer to determine whether it is within legal stack boundaries, specified in the PE header, in order to eliminate any usage of the heap memory area as a fake stack by malware. This is an application-based mitigation, and applicable with both 32-bit and 64-bit processes.

**Caller Checks  Return Oriented Programming (ROP) Security Mitigation:** Microsoft EMET monitors critical function calls by checking caller instructions, based on its return address to detect invalid function calls. In a regular function call, the caller instruction has to be a call (CALL) instruction, not a return (RET) or a jump (JMP) instruction. This could be evidence of an execution flow manipulation by malware. This is an application-based mitigation, and works only for 32-bit processes, not 64-bit ones.

**Attack Surface Reduction (ASR) Mitigation:** Microsoft EMET manages which modules or plugins can be used within a specified target application to reduce attack surface. Its Security Zones support allows users to configure behaviors for different zones, such as Internet and Intranet zones. This is a application-based mitigation, and applicable with both 32-bit and 64-bit processes.

**Deep Hooks - Additional Return Oriented Programming (ROP) Security Mitigation:** Microsoft EMET protects critical functions at a deeper level, by hooking not only critical kernel32 functions but also critical kernelbase and ntdll

functions. It thereby prevents malware from using deeper level critical functions, such as kernelbase.VirtualProtect, and ntdll.ZwProtectVirtualMemory.

**Anti Detours - Additional Return Oriented Programming (ROP) Security Mitigation:** Microsoft EMET detects simple jump-around bypassing techniques, executing overwritten instructions by a hook, and jumping to the next safe instruction in the hooked critical function.

**Banned Functions - Additional Return Oriented Programming (ROP) Security Mitigation:** Microsoft EMET monitors function calls to ntdll.LdrHotPatchRoutine and blocks them in order to prevent malware from abusing this function. This mitigation has been included in Microsoft EMET since version 4.0. In the MS13-063 vulnerability, attackers can call ntdll.LdrHotPatchRoutine through a pointer in SharedUserData, located in a predictable memory region. This pointer points the address of ntdll.LdrHotPatchRoutine. By calling ntdll.LdrHotPatchRoutine, attackers can remotely load their malicious modules so they bypass both DEP and ASLR protections [153].

### 3.3.2 Microsoft EMET - Monitoring Critical Functions via API Hooking

Several operating system functions are considered critical by Microsoft EMET. The list of the critical functions follows below. Microsoft EMET hooks fifty-one (51) functions of three (3) Windows modules in total. Figure 3.1 provides a full list of functions hooked by Microsoft EMET for 32-bit Windows and WOW64 subsystems of 64-bit Windows. All hooked functions are from the three main Windows modules, namely kernel32.dll, kernelbase.dll, and ntdll.dll. In particular, this means that deeper level API hooking is used by Microsoft EMET, providing protection against deeper functions calls, instead of API functions appearing only in kernel32.dll. Similar to the situation with McAfee HIPS, kernel32.dll is the most hooked module with 22 hooked functions for Microsoft EMET. Ntdll.dll and kernelbase.dll contain the second and third most hooked functions with 15 and 14 functions respectively. While Microsoft EMET hooks the functions only in the three main Windows modules, McAfee hooks

more Windows modules (twelve in total.) The numbers of hooked functions for each module are as follows:

- Ntdll.dll - 15 hooked functions
- Kernelbase.dll - 14 hooked functions
- Kernel32.dll - 22 hooked functions.

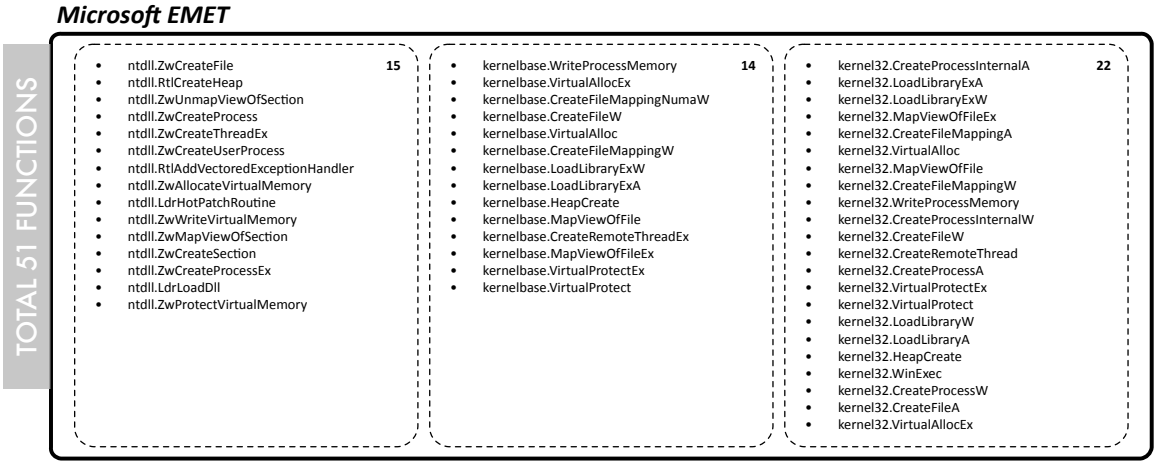


Figure 3.1: The list of the critical functions, hooked by Microsoft EMET in 32-bit Windows and WOW64

In order to place a hook into a critical function, Microsoft EMET adds a jump instruction. The size of a short jump instruction is five (5) bytes in a 32-bit architecture. Therefore, Microsoft EMET changes the first five (5) bytes of every critical function with a jmp instruction, jumping to its control code. Also, Microsoft EMET changes $n$ bytes with $n$ copies of INT3, a special one byte instruction, where $n$ is a non-deterministic value.

Microsoft EMET - Original kernel32.VirtualProtect Code

| ADDRESS | HEX | INSTRUCTION |
|---|---|---|
| 7707435F | 8B FF | MOV EDI,EDI |
| 77074361 | 55 | PUSH EBP |
| 77074362 | 8B EC | MOV EBP,ESP |
| 77074364 | 5D | POP EBP |
| 77074365 | E9 5E CD FF FF | JMP kernel32.CreateProcessA |

Microsoft EMET - Hooked kernel32.VirtualProtect Code

| ADDRESS | HEX | INSTRUCTION |
|---|---|---|
| 7707435F | E9 A4 C3 FE BF | JMP 35d70708 |
| 77074364 | CC | INT 3 |
| 77074365 | CC | INT 3 |
| 77074366 | CC | INT 3 |
| 77074367 | CC | INT 3 |
| 77074368 | CC | INT 3 |
| 77074369 | CC | INT 3 |

Figure 3.2: The comparison of the changed (5+6) bytes of the original and hooked functions of kernel32.VirtualProtect in 32-bit Windows and WOW64

It seems that Microsoft EMET adds more protection by replacing a non-deterministic $n$ bytes of original code with INT3; therefore, malware will not predict a jump point

after the Microsoft EMET hooking bytes. The INT3 instruction is used for setting a trap for debuggers. The processor invokes the interrupt handler of Windows when it executes a INT3 instruction. If malware tries to jump the next instruction, fortunately, it will jump to this trap and trigger an exception, meaning it is likely that Microsoft EMET will detect this malicious activity. Figure 3.2 depicts a comparison of the changed (5+6) bytes of the original and hooked functions of kernel32.VirtualProtect.

In the kernel32.VirtualProtect code, the first three instructions, "mov edi, edi", "push ebp", and "mov ebp, esp" take five (5) bytes and so they are changed with "jmp xxxxxxxx" by Microsoft EMET. The xxxxxxxx value is an address in the intrusion prevention (IPS) code of Microsoft EMET. Therefore, this jmp instruction will direct the execution flow to IPS code every time that kernel32.VirtualProtect is invoked. Moreover, two more instructions, 6 bytes, "pop ebp", and "jmp kernel32.CreateProcessA" are changed to 6 INT3 instructions as a trap. In Figure 3.3, the original and hooked versions of kernel32.VirtualProtect functions can be compared.

Microsoft EMET - Original kernel32.VirtualProtect Code

| ADDRESS | INSTRUCTION |
|---|---|
| 7707435F | MOV EDI,EDI |
| 77074361 | PUSH EBP |
| 77074362 | MOV EBP,ESP |
| 77074364 | POP EBP |
| 77074365 | JMP kernel32.CreateProcessA |
| 7707436A | MOV ECX,DWORD PTR DS:[ESI] |
| 7707436C | MOV DWORD PTR DS:[EAX],ECX |
| 7707436E | MOV ECX,DWORD PTR DS:[ESI+4] |
| 77074371 | MOV DWORD PTR DS:[EAX+4],ECX |
| 77074374 | JMP kernel32.LocalFree |
| 77074379 | MOV EAX,DWORD PTR SS:[EBP-130] |
| 7707437F | JMP kernel32.LocalFree |
| 77074384 | CALL kernel32.RegKrnGetGlobalState |
| 77074389 | JMP kernel32.GetProfileStringW |
| 7707438E | NOP |
| 7707438F | NOP |
| 77074390 | NOP |
| 77074391 | NOP |
| 77074392 | NOP |

Microsoft EMET - Hooked kernel32.VirtualProtect Code

| ADDRESS | INSTRUCTION |
|---|---|
| 7707435F | JMP 35d70708 |
| 77074364 | INT 3 |
| 77074365 | INT 3 |
| 77074366 | INT 3 |
| 77074367 | INT 3 |
| 77074368 | INT 3 |
| 77074369 | INT 3 |
| 7707436A | MOV ECX,DWORD PTR DS:[ESI] |
| 7707436C | MOV DWORD PTR DS:[EAX],ECX |
| 7707436E | MOV ECX,DWORD PTR DS:[ESI+4] |
| 77074371 | MOV DWORD PTR DS:[EAX+4],ECX |
| 77074374 | JMP kernel32.LocalFree |
| 77074379 | MOV EAX,DWORD PTR SS:[EBP-130] |
| 7707437F | JMP kernel32.LocalFree |
| 77074384 | CALL kernel32.RegKrnGetGlobalState |
| 77074389 | JMP kernel32.GetProfileStringW |
| 7707438E | NOP |
| 7707438F | NOP |
| 77074390 | NOP |
| 77074391 | NOP |
| 77074392 | NOP |

Figure 3.3: A sample Microsoft EMET Hook on kernel32.VirtualProtect function

On a protected Windows client, the following service of Microsoft EMET is always running.

- Microsoft Enhanced Mitigation Experience Toolkit Agent (EMET_Agent.exe)

Moreover, to effect userland protections, Microsoft EMET protects some running processes, which are listed on the "Application Configuration" screen. The "Apps" button is used for opening the "Application Configuration" screen. Users can add

and remove applications to/from the protected application list. Also, on this screen, different combinations of protection methods can be defined for each application by checking and unchecking the small boxes, representing fourteen protection methods.

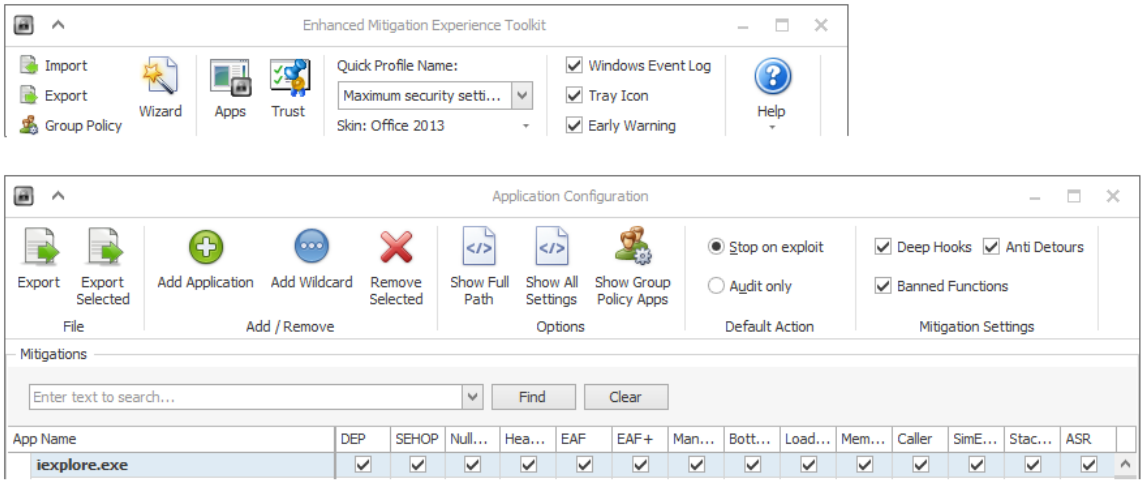


Figure 3.4: Application Configuration Screen of Microsoft EMET

Microsoft EMET injects its intrusion protection module inside the memory space of all protected processes in userland. For a running Internet Explorer process, EMET.dll or EMET64.dll is injected by Microsoft EMET. According to its file description, this module is the "Microsoft Enhanced Mitigation Experience Toolkit" module (EMET.dll). Figure 3.5 below depicts the virtual address space of a protected Internet Explorer 8 process in 32 bit Windows and WOW64.

Microsoft EMET also allocates an additional memory area with read and execution (RE) rights, and it uses this memory area in the process virtual address space to add its API hooking code. This reserved memory areas is 65536 bytes (64KB) and its owner and name are hidden. The layout of the hidden memory area in 32-bit Internet Explorer process virtual address space is given in Figure 3.6.

The hidden memory area includes function-specific hooking code for 51 critical functions in user space, hooked by Microsoft EMET. Every function-specific hooking code has a CALL instruction to the main control code in EMET.dll, depicted in Figure 3.7. The function-specific hooking code parts are a fixed size, 0x78 (120) bytes. Each hooking code part has a 38-byte hooking code and some data of the hooked critical function, such as the original code of the overwritten instructions.

In 32-bit Windows, for 51 hooked critical functions of several Windows mod-

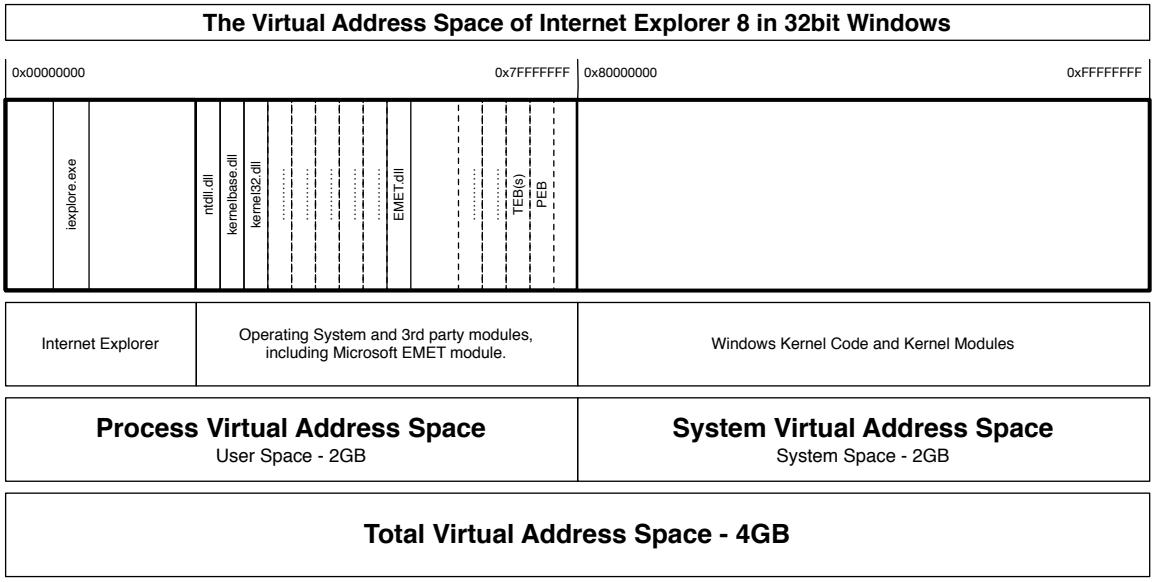


Figure 3.5: Virtual Address Space of Internet Explorer 8 protected by Microsoft EMET

ules, there are 51 hooking code sections in Microsoft EMET's hidden memory area. Therefore, every critical function hook is able to jump to corresponding hooking code, depicted in Figure 3.8. The layout of the hooking code is fixed in memory. Microsoft EMET always uses the same offsets for each critical function.

The memory addresses of the modules and the hidden section change for every execution of Internet Explorer due to the address space layout randomization (ASLR), a protection against buffer overflow attacks. However, for adversaries, it is possible to find the addresses of the hidden memory areas of Microsoft EMET by following its hooks on critical functions. Each hook is simply a jump instruction to an address in these hidden memory areas. For every critical function, there is a corresponding code section. These code sections are very similar, and most of their code are exactly the same.

### 3.3.3 Microsoft EMET - Monitoring Critical Data Structures via Guard Pages

Microsoft EMET employs guard pages [90] as a trap to monitor critical data structures in process memory, such as Export Address Tables (EAT) in kernel32.dll, kernelbase.dll, and the Portable Executable (PE) Header of ntdll.dll. Prior to using guard pages, starting with version 5.2 Microsoft EMET used debug registers to pro-

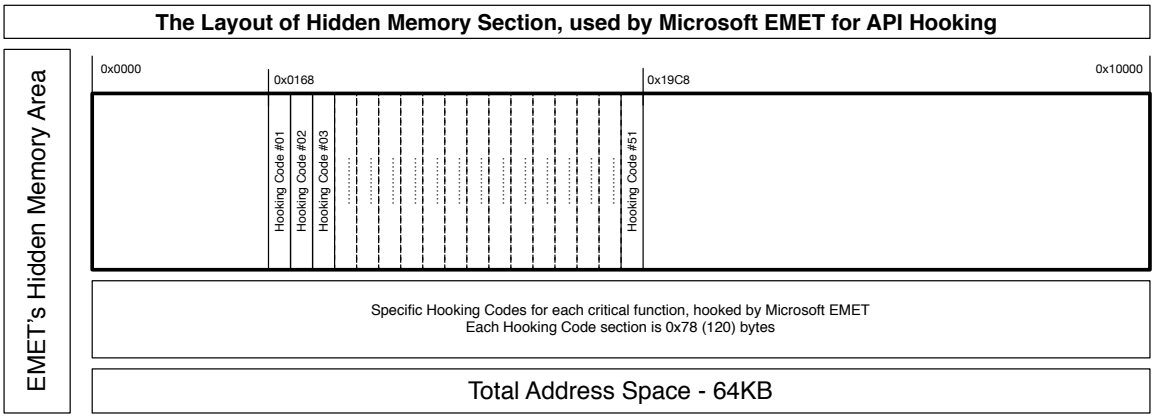


Figure 3.6: The Layout of Microsoft EMET's Hidden Section

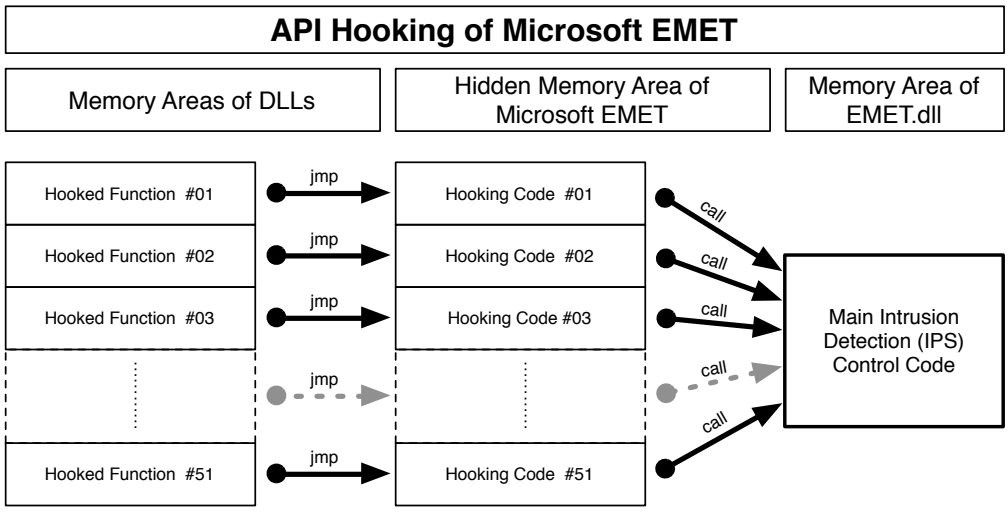


Figure 3.7: API Hooking of Microsoft EMET

vide similar protection. With version 5.5, Microsoft EMET started to use the guard page mechanism as a new protection. Microsoft EMET's current version 5.52 uses the same guard page protection mechanism as version 5.5. Although there is no limit on the use of guard pages, the number of debug registers in Intel Processors, such as DR01, DR02, DR03, and DR04, is limited. Therefore, with guard pages more memory areas can be monitored. However, bypasses of guard page and debug register protections are equally possible. In Table 3.1, the list of guard pages of Microsoft EMET is shown for a WoW64 subsystem of Windows 64-bit. They provide protection against information leaks which result from the reading of an EAF or PE Header. All of these three guard pages are 0x1000 bytes.

Microsoft EMET primarily monitors three important operating system modules,

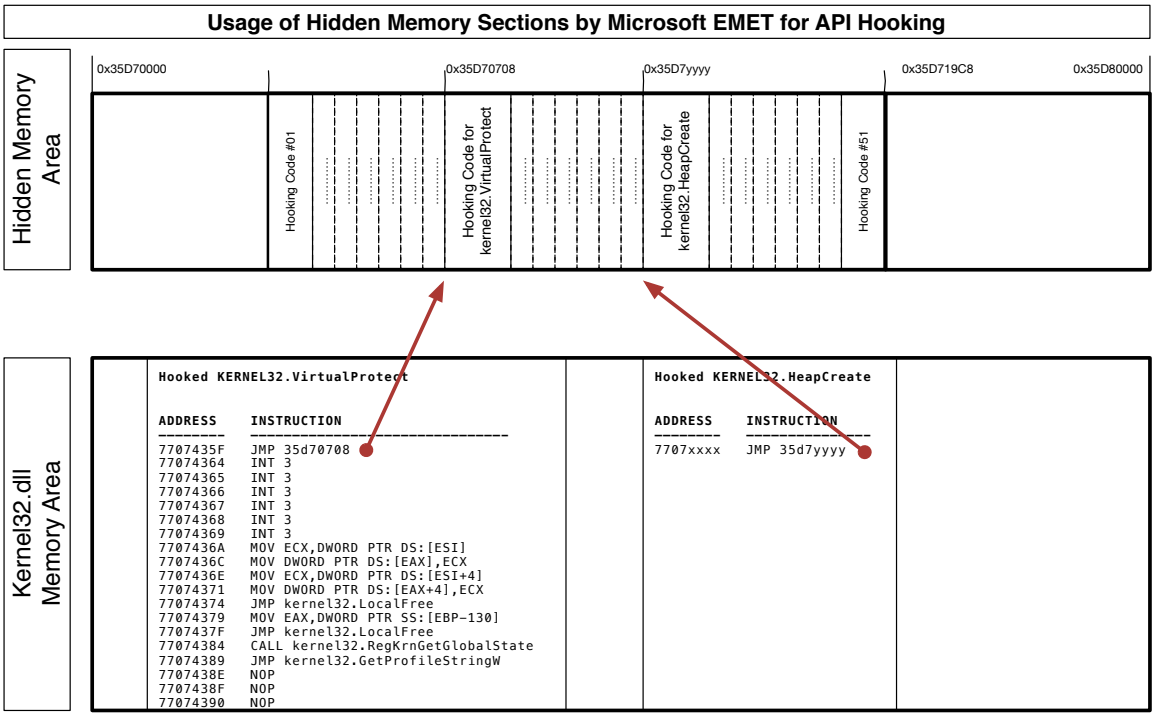


Figure 3.8: Usage of Hidden Sections by Microsoft EMET for API Hooking

| Module | EAF Address | GUARD PAGE ADDRESS Starting Offset | Ending Offset | Size |
|---|---|---|---|---|
| ntdll.dll | 0x101F8 | 0x0000 | 0x1000 | 0x1000 bytes |
| kernelbase.dll | 0x3A6E0 | 0x3A000 | 0x3B000 | 0x1000 bytes |
| kernel32.dll | 0xBFA00 | 0xBF000 | 0xC0000 | 0x1000 bytes |

Table 3.1: Microsoft EMET's Guard Page List for a WoW64 subsystem

namely kernel32.dll, kernelbase.dll, and ntdll.dll. Their fifty-one (51) exported functions are considered critical functions by Microsoft EMET. It not only monitors these fifty-one (51) critical functions using API hooking, it also monitors the critical data structures of these three modules using guard pages as well. Figure 3.9 presents the memory map of ntdll.dll in a WoW64 subsystem. When a module is loaded into process memory, it has several sections with different protections, which is very similar within the PE file format. Most common sections are PE header, .text, .data, .rsrc, .reloc. Ntdll sections in memory can be seen in the figure with their protections. For example, the .text section has "READ and EXECUTE" rights, whereas the .data section has "READ and WRITE" one. Normally, the PE Header of ntdll.dll

has "READONLY" rights. However, Microsoft EMET changes the protection for the memory region between 0x0000 and 0x1000 offsets to a guard page. The guard page violation exception will be raised, if any access attempt occurs to this memory area within the guard page of Microsoft EMET, enabling Microsoft EMET to execute its intrusion control code.

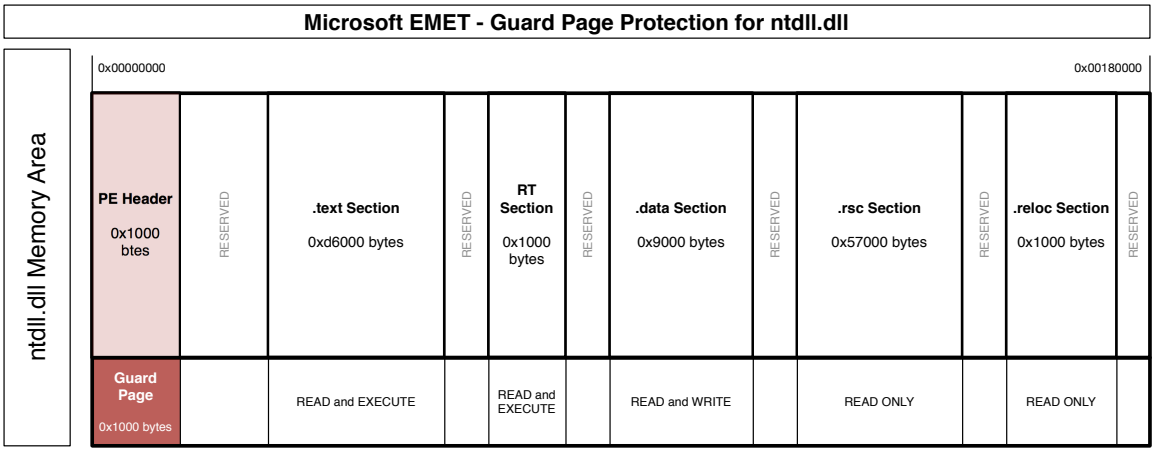


Figure 3.9: Microsoft EMET's Guard Page Protection for ntdll.dll in a WoW64 subsystem

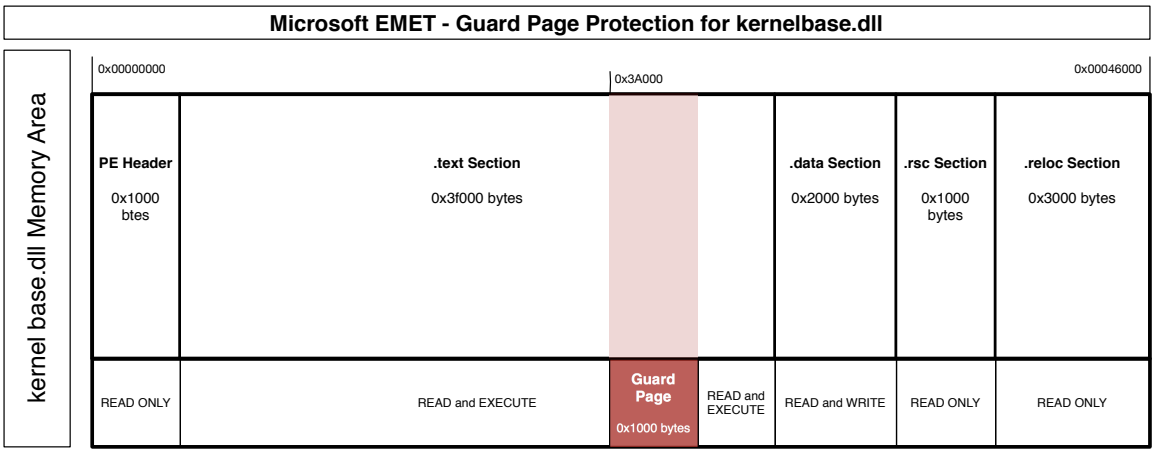


Figure 3.10: Microsoft EMET's Guard Page Protection for kernelbase.dll in a WoW64 subsystem

Similarly, Figure 3.10 and Figure 3.11 depict the memory maps of kernelbase.dll and kernel32.dll, respectively, in a WoW64 subsystem. In Figure 3.10, the kernelbase guard page is located between 0x3A000 and 0x3B000 offset addresses, while the location of the kernel32 guard page starts from the 0xBF000 offset address, which

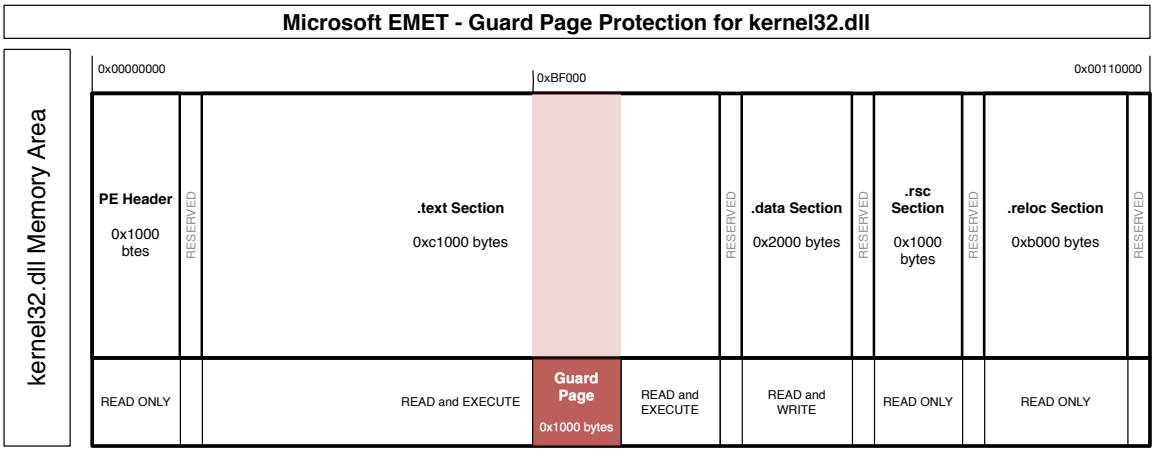


Figure 3.11: Microsoft EMET's Guard Page Protection for kernel32.dll in a WoW64 subsystem

is also called a relative address. The offset addresses of EATs for kernelbase.dll and kernel32.dll are 0x3A6E0 and 0xBFA00, respectively. Both EATs are in these 0x1000-byte memory areas, which are set as guard pages. Thus, any reading or writing operations will cause an execution of the intrusion control code of Microsoft EMET.

The methods for bypassing the protections provided by Microsoft EMET in user-land are explained in section 3.5.1 of this chapter, Bypassing Microsoft EMET.

## 3.4 McAfee HIPS - McAfee Host Intrusion Prevention

McAfee's Host Intrusion Prevention System, abbreviated as McAfee HIPS, is an enterprise level host based intrusion detection and prevention product from Intel Security designed to enable business customers to protect their computer systems [131]. According to case studies performed by Intel Security, McAfee HIPS products are installed, and protect, a significant number of computer systems for various worldwide and local organizations in several sectors, including government, healthcare, education, finance, retail and telecommunications [126, 121, 122, 127, 125, 129, 124, 123, 128]. McAfee HIPS promises users high-level protection against both zero-day and well-known threats against Windows and non-Windows computers.

McAfee HIPS consists of intrusion protection (IPS) and firewall components. Its

IPS component detects and terminates previously known and even unknown cyber attacks. Identified remote attacks are then blocked at the network level by its firewall component before malware reaches endpoints such as desktop systems. McAfee HIPS can also work together with McAfee Antivirus on the same computer to provide combined signature and behavioral based protection. McAfee HIPS provides fewer protection features on non-Windows operating systems, for example there it does not yet provide buffer overflow prevention and network protection on Linux clients. However, it supports a wide range of Microsoft Windows versions from XP to 10 in 32 or 64-bit architectures [130, 76, 77].

McAfee HIPS has no standalone installation option so it can be only installed and managed via McAfee ePolicy Orchestrator, a centralized management system. On Microsoft Windows, McAfee HIPS can be configured with different security levels using IPS and firewall rules, which are distributed from a central location. Thus, the McAfee ePolicy Orchestrator delivers and enforces policies to define the runtime reactions of McAfee HIPS. Its policies and content can be updated by users employing the orchestrator. McAfee HIPS detects and blocks remote attacks leveraging behavioral rules, signatures, and a system firewall on Windows computers [76]. Encountered security issues are reported to the management software in order to track, log, and analyze them in a company-wide scope as a part of its enterprise level management [77].

### 3.4.1 McAfee HIPS - Technical background

McAfee HIPS is a policy based software system and has three policy types, including general, firewall and IPS policy types. A policy is a set of configurable security settings. General policies are used for managing client user interface settings and defining the lists of trusted networks and applications. Some policies only work on Windows environments, such as firewall policies. Firewall policies define firewall rules and a list of blocked DNS servers. Similarly, IPS policies are categorized as IPS options, IPS protection, and IPS rules. IPS protection defines behaviors to cope with security violations, whereas IPS rules are applied to detect known and unknown malicious attacks. The "Policy Catalog" screen of McAfee ePolicy Orchestrator is used for managing policies. For example, on the policy management screen, a new IPS rules policy can be created and specific rules can be enabled or disabled with various options for this new policy.

An IPS rule can be a signature, a behavioral rule, an application protection rule, or an exception rule [77]. Signatures can be a fingerprint or pattern of an attack, such as a malicious string of a known HTTP thread, or the execution of a program with a double file extension. Signatures are used for the detection of known attacks, while behavioral and application protection rules are used for the detection of unknown attacks. Behavioral rules can be considered as a database of normal application activities, such as application shielding and enveloping [77]. Any anomaly in application behaviors can be accepted as a security violation. To avoid false positive alarms, exception rules are used for defining the normal work routine of a user. Also, application protection rules protect specified processes using API inline hooks in userland. API hooking is a defensive technique to monitor system calls and detect malicious activities, like buffer overflow exploits. Signatures and application protection rules are frequently updated by content updates every month.

McAfee HIPS employs several IPS protection methods, including enveloping and shielding, service specific engines, system call interception, and kernel drivers [77]. Enveloping and shielding ensures that an application can only access its own resources, such as files, data, registry settings and services, and these application specific resources are not accessible by others, preventing exploit modifications. The HTTP and SQL engines of McAfee HIPS protect against malicious HTTP servers and SQL clients [77, 120]. The HTTP engine is located after the SSL decryption mechanism, providing a plaintext version of incoming HTTP streams. Therefore, the HTTP engine can parse received web pages and perform malicious pattern checks on them to detect suspicious data sequences. Similarly, the SQL engine is placed in front of a database engine to control every SQL request in order to protect against malicious queries, such as SQL injections. As a system call interception method, HIPS uses its own kernel driver in kernel land and API hooking in userland, to detect most known and zero-day attacks.

In kernel land, the HIPS kernel-level driver (HipShieldK.sys) monitors system call chains via the kernel patching method, modifying critical system structures in the kernel of the operating system [89]. It uses redirected entries in the user-mode system call table [77]. For each system call, the driver performs some controls based on IPS rules, such as signatures and behavioral rules. The protection in kernel land via the HIPS kernel driver has some limitations for 64-bit Windows clients because the Windows operating system has patch protection to preserve kernel integrity, preventing third parties from changing the code and critical structures of the operating

system kernel [120, 89]. In userland, McAfee HIPS monitors running executables by injecting its own DLLs and API hooks into their memory space in order to intercept and validate system calls. Thus, it tracks a group of important system API functions to detect invalid Windows API calls. Some API functions, such as Loadlibrary, VirtualProtect, and HeapCreate are important because they are subject to malicious misuse. Control code inserted by HIPS is executed for every system call in order to determine whether the call is invoked from malware code on the memory stack or heap. This involves checking API call conventions defined in behavioral rules, such as controlling return addresses, and target addresses.

### 3.4.2 McAfee HIPS - Monitoring Critical Functions via API Hooking

Several operating system functions are considered critical by McAfee HIPS. The list of critical functions of McAfee is given below. A total of fifty-three (53) functions in twelve (12) Windows modules are hooked by McAfee HIPS. Figure 3.12, gives a full list of functions hooked by HIPS for 32-bit Windows and WOW64 subsystem of 64-bit Windows. Most of these functions belong to three main Windows modules, kernel32.dll, kernelbase.dll, and ntdll. This shows that McAfee HIPS uses a deeper level API hooking method. McAfee HIPS does not only hook the API functions of kernel32.dll, but also it hooks more functions in deeper levels, such as kernelbase.dll and ntdll. Generally, the kernel32.dll functions call or jump to the functions of kernelbase, similarly, the kernelbase functions call or jump to the functions of ntdll. Therefore, McAfee HIPS hooks at different levels to prevent deeper functions calls, not just API functions in kernel32.dll. The numbers of hooked functions based on modules are found below.

- Ntdll.dll - 4 hooked functions
- Kernelbase.dll - 9 hooked functions
- Kernel32.dll - 19 hooked functions
- Msvcrt.dll - 2 hooked functions
- Msvcr71.dll - 1 hooked function
- Ole32.dll - 3 hooked functions
- Rpcrt4.dll - 2 hooked functions
- Sspicli.dll - 1 hooked function
- Urlmon.dll - 2 hooked functions

- User32.dll - 2 hooked functions
- Wininet.dll - 3 hooked functions
- Ws2_32.dll - 5 hooked functions.

Kernel32.dll is the most hooked module with nineteen (19) functions. Kernelbase.dll and Ws2_32.dll are the second and third most hooked functions with respectively nine (9) and five (5) functions. Another very important module of Windows, ntdll.dll has four (4) hooked functions. The number of ntdll functions, four (4), hooked by McAfee HIPS is much less than the fourteen (14) hooked by Microsoft EMET. However, unlike Microsoft EMET, the API hooks of McAfee HIPS are not limited to only three (3) modules. McAfee HIPS hooks and monitors many more modules than Microsoft EMET.

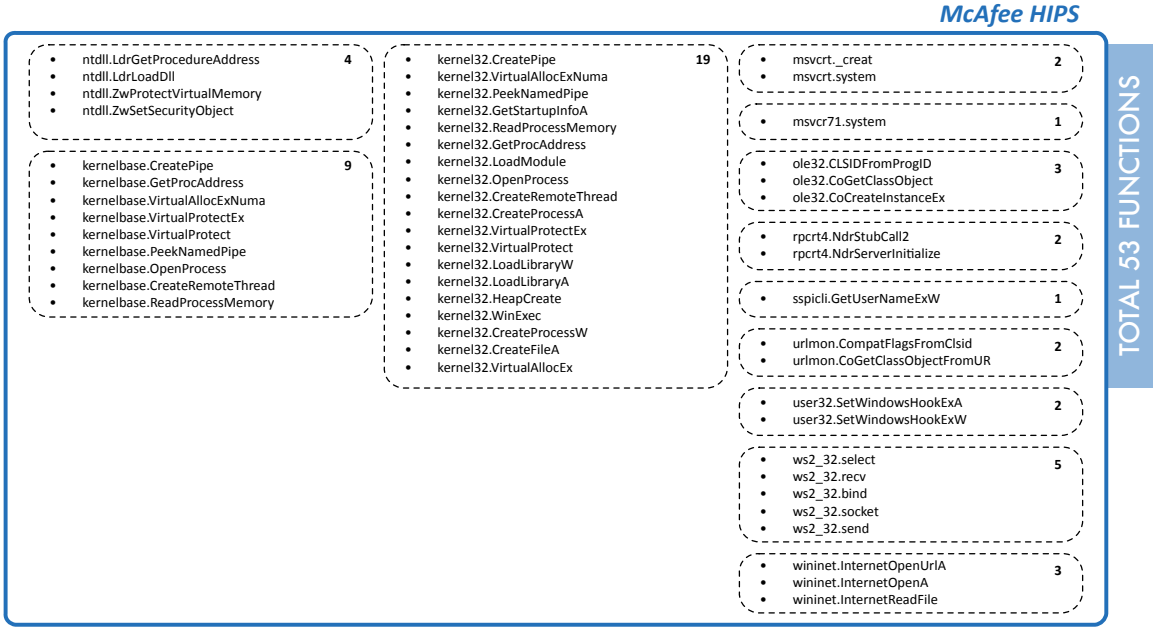


Figure 3.12: The list of the critical functions, hooked by McAfee HIPS in 32-bit Windows and WOW64

In order to place a hook into a critical function, McAfee HIPS simply adds a jump instruction. The size of a short jump instruction is five (5) bytes in a 32-bit architecture. Therefore, McAfee HIPS changes the first five (5) bytes of every critical function with a jmp instruction, jumping its control code. The following Figure 3.13 depicts a comparison of the first five (5) bytes of the original and hooked functions of kernel32.VirtualProtect.

McAfee HIPS - Original kernel32.VirtualProtect Code

| ADDRESS | HEX | INSTRUCTION |
|---|---|---|
| 7707435F | 8B FF | MOV EDI,EDI |
| 77074361 | 55 | PUSH EBP |
| 77074362 | 8B EC | MOV EBP,ESP |

McAfee HIPS - Hooked kernel32.VirtualProtect Code

| ADDRESS | HEX | INSTRUCTION |
|---|---|---|
| 7707435F | E9 71 C9 20 89 | JMP 00280CD5 |

Figure 3.13: The comparison of the first five (5) bytes of the original and hooked functions of kernel32.VirtualProtect in 32-bit Windows and WOW64

In the kernel32.VirtualProtect code, the first three instructions, "mov edi, edi", "push ebp", and "mov ebp, esp" take five (5) bytes and so they are changed with "jmp xxxxxxxx" by McAfee HIPS. The xxxxxxxx value is an address in the intrusion prevention (IPS) code of McAfee HIPS. Therefore, this jmp instruction will direct the execution flow to IPS code every time kernel32.VirtualProtect is invoked. In Figure 3.14, the original and hooked versions of kernel32.VirtualProtect function can be compared.

McAfee HIPS - Original kernel32.VirtualProtect Code

| ADDRESS | INSTRUCTION |
|---|---|
| 7707435F | MOV EDI, EDI |
| 77074361 | PUSH EBP |
| 77074362 | MOV EBP,ESP |
| 77074364 | POP EBP |
| 77074365 | JMP kernel32.CreateProcessA |
| 7707436A | MOV ECX,DWORD PTR DS:[ESI] |
| 7707436C | MOV DWORD PTR DS:[EAX],ECX |
| 7707436E | MOV ECX,DWORD PTR DS:[ESI+4] |
| 77074371 | MOV DWORD PTR DS:[EAX+4],ECX |
| 77074374 | JMP kernel32.LocalFree |
| 77074379 | MOV EAX,DWORD PTR SS:[EBP-130] |
| 7707437F | JMP kernel32.LocalFree |
| 77074384 | CALL kernel32.RegKrnGetGlobalState |
| 77074389 | JMP kernel32.GetProfileStringW |
| 7707438E | NOP |
| 7707438F | NOP |
| 77074390 | NOP |
| 77074391 | NOP |
| 77074392 | NOP |

McAfee HIPS - Hooked kernel32.VirtualProtect Code

| ADDRESS | INSTRUCTION |
|---|---|
| 7707435F | JMP 00280CD5 |
| 77074364 | POP EBP |
| 77074365 | JMP kernel32.CreateProcessA |
| 7707436A | MOV ECX,DWORD PTR DS:[ESI] |
| 7707436C | MOV DWORD PTR DS:[EAX],ECX |
| 7707436E | MOV ECX,DWORD PTR DS:[ESI+4] |
| 77074371 | MOV DWORD PTR DS:[EAX+4],ECX |
| 77074374 | JMP kernel32.LocalFree |
| 77074379 | MOV EAX,DWORD PTR SS:[EBP-130] |
| 7707437F | JMP kernel32.LocalFree |
| 77074384 | CALL kernel32.RegKrnGetGlobalState |
| 77074389 | JMP kernel32.GetProfileStringW |
| 7707438E | NOP |
| 7707438F | NOP |
| 77074390 | NOP |
| 77074391 | NOP |
| 77074392 | NOP |

Figure 3.14: A sample McAfee HIPS Hook on kernel32.VirtualProtect function

On a protected Windows client, the following services of McAfee HIPS are always running [120].

- McAfee Host Intrusion Prevention Service (FireSvc.exe)
- McAfee Firewall Core Service (mfefire.exe)
- McAfee Validation Trust Protection Service (mfevtps.exe)
- McAfee Host Intrusion Prevention Local Procedure Call (LPC) Service (HipMgmt.exe)

Moreover, as part of its userland protections, McAfee HIPS protects some running processes, except its trusted applications. Applications which do not need protections

are listed in Trusted Applications General Policies. Users can add and remove applications to/from this list using the management software. McAfee HIPS injects its intrusion protection modules inside the memory space of all protected processes in userland. For a running Internet Explorer process, HcApi.dll, HcThe.dll, and HIPHandlers.dll are injected by McAfee HIPS. According to their file descriptions, these modules are:

- McAfee HIP Engine module (HcApi.dll)
- McAfee HIP Thin Hook Environment module (HcThe.dll)
- HIPS Signatures module (HIPHandlers.dll)

The Figure 3.15 below depicts the virtual address space of a protected Internet Explorer 8 process in 32 bit Windows and WOW64.

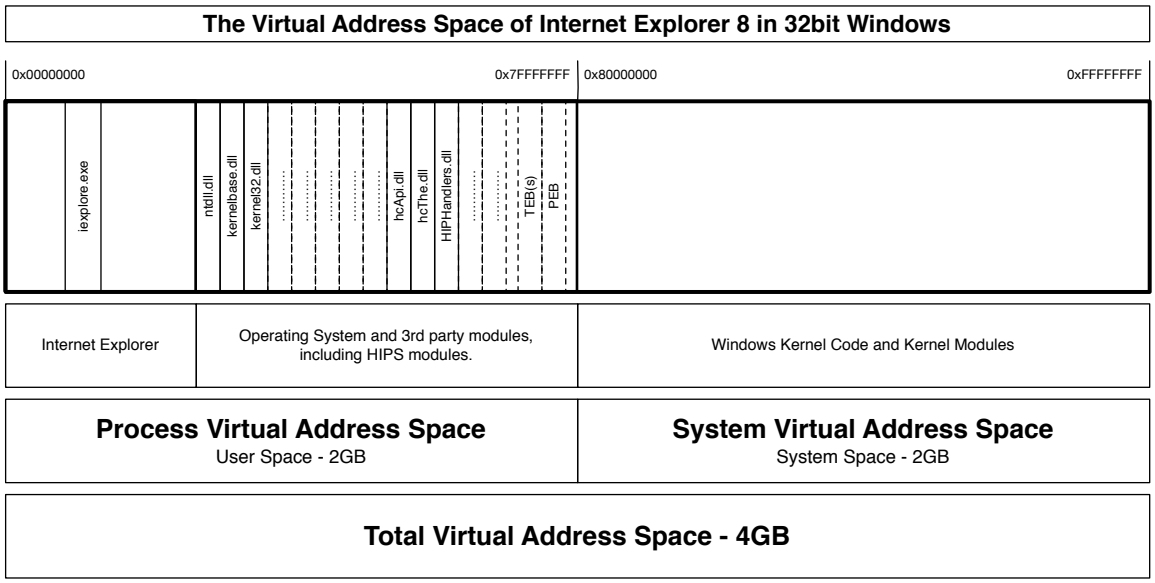


Figure 3.15: Virtual Address Space of Internet Explorer 8 protected by McAfee HIPS

Additionally, McAfee HIPS allocates and uses two additional memory areas in the process virtual address space to add its API hooking codes. Each of these reserved memory areas is 8192 bytes (8KB) and their owners and names are hidden. They have read and execution (RE) rights in memory, generally code sections of programs and DLLs have the same rights, RE. The simplified layouts of these hidden memory areas in 32-bit Internet Explorer process virtual address space are as in the following Figure 3.16. They are loaded into different address spaces, which are not necessarily

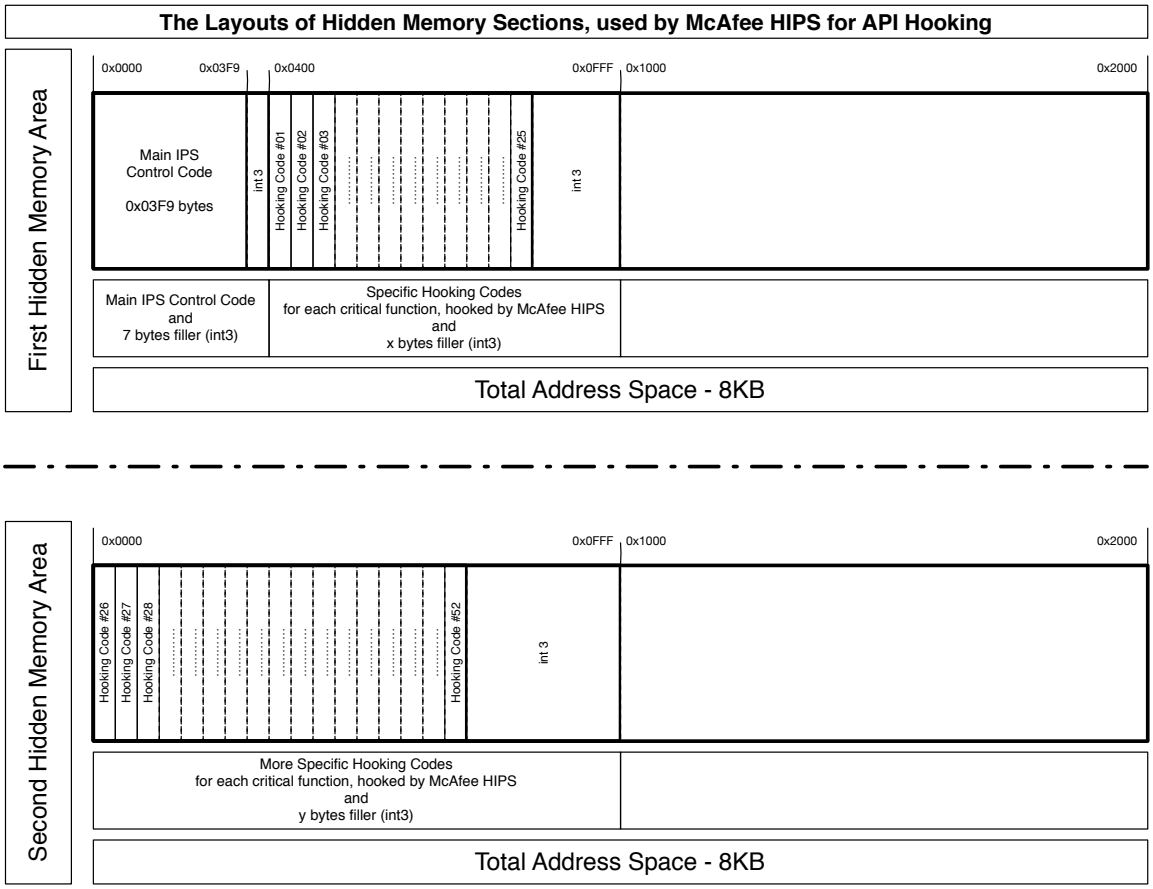


Figure 3.16: The Layout of McAfee HIPS's Hidden Sections

sequential in memory. Even though they are 8KB in total, the main control code part and the hooking code parts are located in the first 4KB of the hidden areas.

In the first hidden memory area, there is a main control code, and this part is not repeated in the second one. Both of the hidden memory areas include function-specific hooking codes for fifty-three (53) critical functions in user space, hooked by McAfee HIPS. Every function-specific hooking code has a CALL instruction to the main control code, depicted in Figure 3.17. The function-specific hooking code parts are fixed size, 0x77 bytes.

In a 32-bit Windows, for fifty-three (53) hooked critical functions of several Windows modules, there are fifty-three (53) hooking code sections in the hidden memory areas. Therefore, every hook on the critical functions is able to jump to its corresponding hooking code, depicted in Figure 3.18. Even though two functions belong to the same Windows module, their hooking code can be in the same hidden memory area or different hidden memory areas of McAfee HIPS.

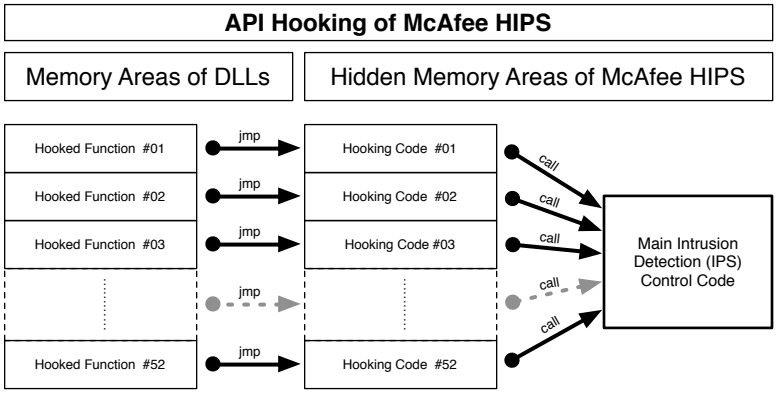


Figure 3.17: API Hooking of McAfee HIPS

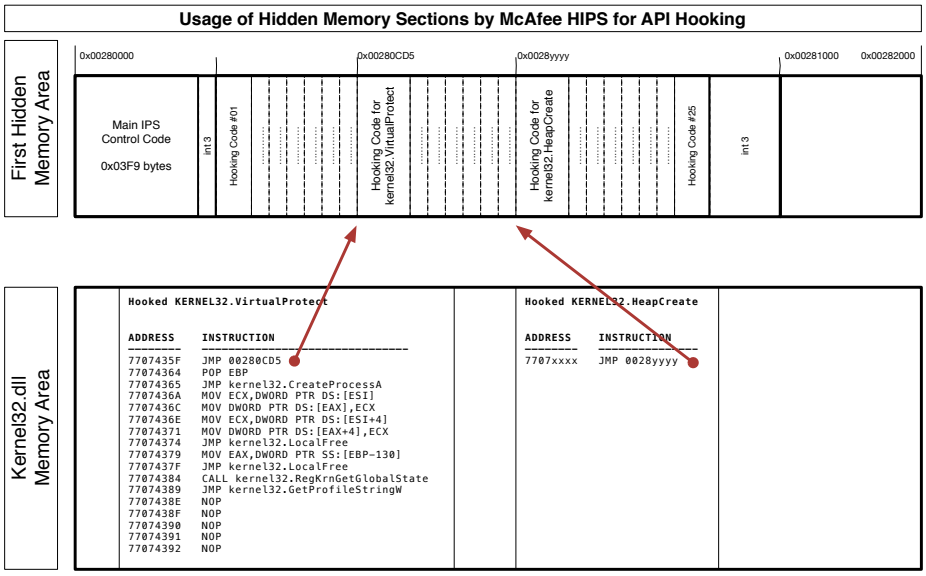


Figure 3.18: Usage of Hidden Sections by McAfee HIPS for API Hooking

The memory addresses of the modules and the hidden sections change in every execution of Internet Explorer due to address space layout randomization (ASLR), a protection against buffer overflow attacks. However, for adversaries, it is possible to find the addresses of the hidden memory areas of McAfee HIPS by following its hooks on the critical functions. Each hook is simply a jump instruction to an address in these hidden memory areas. For every critical function, there is a specific code part. These code parts are very similar, and most of their codes are exactly the same.

The bypassing methods of McAfee HIPS in userland are explained in the Bypassing McAfee HIPS section 3.5.2 in this chapter.

## 3.5 Bypassing Behavioral Based Detections of Antivirus Products in User Space

The following two subsections, 3.5.1 and 3.5.2, explain how it is possible to bypass behavioral-based detections of antivirus products in user space using specific software details of Microsoft EMET, and McAfee HIPS, two enterprise level antivirus products, widely used in real world, described in the preceding section. Although these products have very similar user space protection methods, such as API hooking, there are important differences in their respective approaches, including Anti-Detours, Guard Pages, and different critical functions lists.

### 3.5.1 Bypassing Microsoft EMET

In Section 3.3, we discussed the mitigations of Microsoft EMET. In this section, we will discuss how an attacker may bypass the inline hooks and the guard page protection of Microsoft EMET.

#### 3.5.1.1 Bypassing Microsoft EMET - Jump-Around API Hooking

Microsoft EMET implements a basic inline hooking method, using a single jmp instruction, which is five bytes in length, into antivirus monitoring code. A well-known bypassing technique against the basic inline hooking method is to emulate the first instruction (five bytes in 32-bit systems) of a critical function prolog, and jump to the following instruction in memory. This bypassing method is also known as the trampoline function method. However, Microsoft EMET changes a random number of bytes with INT3 instruction to prevent this simple jump-around bypass, an approached called "Anti-Detour". In Figure 3.19, the instructions of kernel32.VirtualProtect in the addresses between 0x7707435F and 0x77074369 are changed by Microsoft EMET to provide protection via API hooking and Anti-Detour. Still, an attacker can emulate more instructions, overwritten by Microsoft EMET, and then jump to the following instruction to conduct their attack. From an attacker's perspective, the difficulty here is that the number of added INT3s is random. Attackers may decide on an appropriate number and organize their trampoline code according to this assumption. However, we have found and implemented a simpler method to call a hooked function of ntdll.dll during our test. Our method is different from other known jump-around methods and it cannot be affected by the anti-detour protection of Microsoft EMET.

More information about the new way to call a hooked function, even with anti-detour protection, is provided in Section 4.2.2.2.

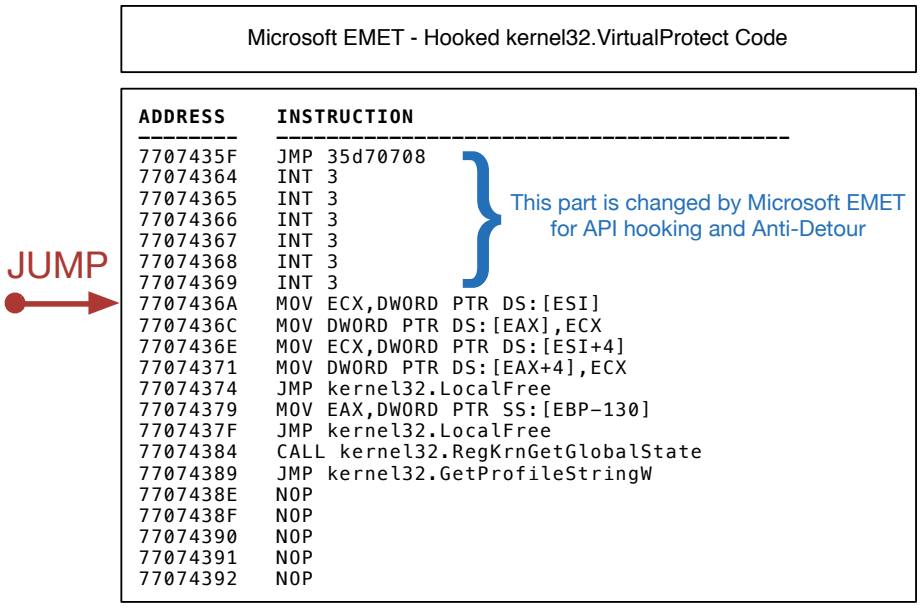


Figure 3.19: Microsoft EMET-aware Jump-around bypass

#### 3.5.1.2 Bypassing Microsoft EMET - Disabling Critical Structures Protection (Guard Pages)

The other important protection method of Microsoft EMET is the use of guard pages to monitor reading and writing activities on the critical data structures of kernel32.dll, kernelbase.dll, and ntdll.dll in process memory. The critical data structures, in particular the Export Address Tables (EAT) of kernel32.dll, kernelbase.dll, and Portable Executable (PE) Header of ntdll.dll, may be used by malware to leak information to bypass DEP protection. Thus, Microsoft EMET changes page access protections by adding the PAGE_GUARD protection value (0x100) on memory regions allocated by these critical data structures.

The following WinDbg output shows the virtual protection information for the guard page of ntdll. The region size is 0x1000 bytes and its access protection value is 0x102, which means PAGE_READONLY and PAGE_GUARD. However, a regular PE Header of ntdll, not protected by Microsoft EMET, has only PAGE_READONLY protection, value 0x2.

```
0:013> !vprot ntdll +0x0
BaseAddress:       770e0000
AllocationBase:    770e0000
AllocationProtect: 00000080  PAGE_EXECUTE_WRITECOPY
RegionSize:        00001000
State:             00001000  MEM_COMMIT
Protect:           00000102  PAGE_READONLY + PAGE_GUARD
Type:              01000000  MEM_IMAGE
```

When the address of kernelbase's EAT is checked using its offset address, 0x3A6E0, it is seen that the kernelbase's EAT is located in a memory region between 0x74f8a000 and 0x74f50000 if the base address of kernelbase is 0x74f50000. This 0x1000-byte memory region is exactly the guard page of Microsoft EMET for kernelbase.dll. Thus, its access protection value is 0x120, which means PAGE_EXECUTE_READ and PAGE_GUARD. Normally, an EAT of kernelbase, not protected by Microsoft EMET, is in the .text section with only PAGE_EXECUTE_READ protection, value 0x20.

```
0:013> !vprot kernelbase +0x3A6E0
BaseAddress:       74f8a000
AllocationBase:    74f50000
AllocationProtect: 00000080  PAGE_EXECUTE_WRITECOPY
RegionSize:        00001000
State:             00001000  MEM_COMMIT
Protect:           00000120  PAGE_EXECUTE_READ + PAGE_GUARD
Type:              01000000  MEM_IMAGE
```

Similarly, the address of kernel32's EAT is placed in a 0x1000-byte memory region of the Microsoft EMET guard page for kernel32.dll. This memory region protection is 0x120, which means PAGE_EXECUTE_READ and PAGE_GUARD too. On a computer with no anti-malware protection, an EAT of kernel32 is in the .text section, with only PAGE_EXECUTE_READ protection, value 0x20, like kernelbase.

```
0:013> !vprot kernel32 +0xBFA00
BaseAddress:       76b8f000
AllocationBase:    76ad0000
```

```
AllocationProtect: 00000080  PAGE_EXECUTE_WRITECOPY
RegionSize:        00001000
State:             00001000  MEM_COMMIT
Protect:           00000120  PAGE_EXECUTE_READ + PAGE_GUARD
Type:              01000000  MEM_IMAGE
```

Unfortunately, an attacker may remove Microsoft EMET's guard pages on critical data structures by setting the original access protections back on their memory regions. Malware may call ntdll.ZwProtectVirtualMemory with appropriate parameters, even though ntdll.ZwProtectVirtualMemory is hooked by Microsoft EMET, as explained in previous section 3.5.1.1. We have implemented this bypassing technique using both unhooking and funneling malware variants.

#### 3.5.1.3 Bypassing Microsoft EMET - Disabling AV Main Control Code (Funneling Method)

In the funneling method, antivirus code in process memory is modified to disable detection. In protected process memory, every API hook jumps to its own hooking code section in the memory area, which belongs to Microsoft EMET. Each of these hooking code sections has the exact same call instruction that invokes an EMET function, at an offset address of 0x27027. Thus, by following EMET's API hooks on critical functions, it is possible to reach the address of Antivirus code, as depicted in Figure 3.20.

Therefore, attacking code may change the access protection of a memory area, in which the antivirus code exists, as writable and modify the antivirus control code in memory thereby disabling its functionality. Before doing any modification, malware has to set this memory region as writable because its protection is normally only "Execute and Read".

```
0:020> !vprot EMET +0x27070
BaseAddress:       73D00000
AllocationBase:    73D00000
AllocationProtect: 00000004  PAGE_READWRITE
RegionSize:        0000?000
State:             00001000  MEM_COMMIT
Protect:           00000020  PAGE_EXECUTE_READ
Type:              00020000  MEM_PRIVATE
```

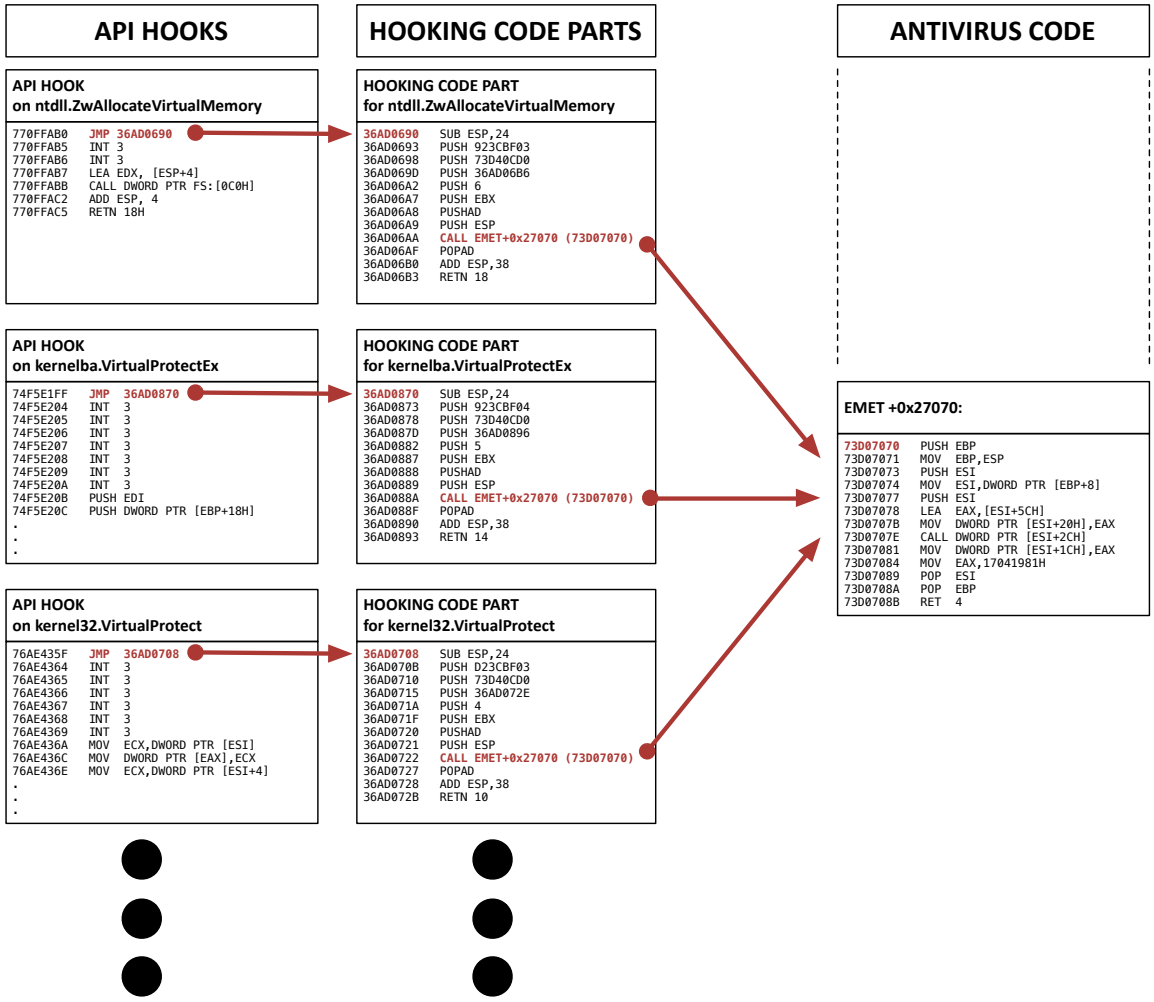


Figure 3.20: Microsoft EMET - Funneling - Invoked EMET+0x27070 by every hooking code parts

Figure 3.21 presents the modification of the main control code of Microsoft EMET in process memory. Just by changing the first 10 bytes of the main control code, a malware variant can disable Microsoft EMET detection. Every hooking code section also has the original instructions of the corresponding critical function, overwritten to add an API hook and Anti-Detour protection. Therefore, it is possible to make a change to the antivirus code in memory in order to execute the original code of a critical function without branching to the main control code of Microsoft EMET. We have implemented this bypassing technique in the funneling malware variants.

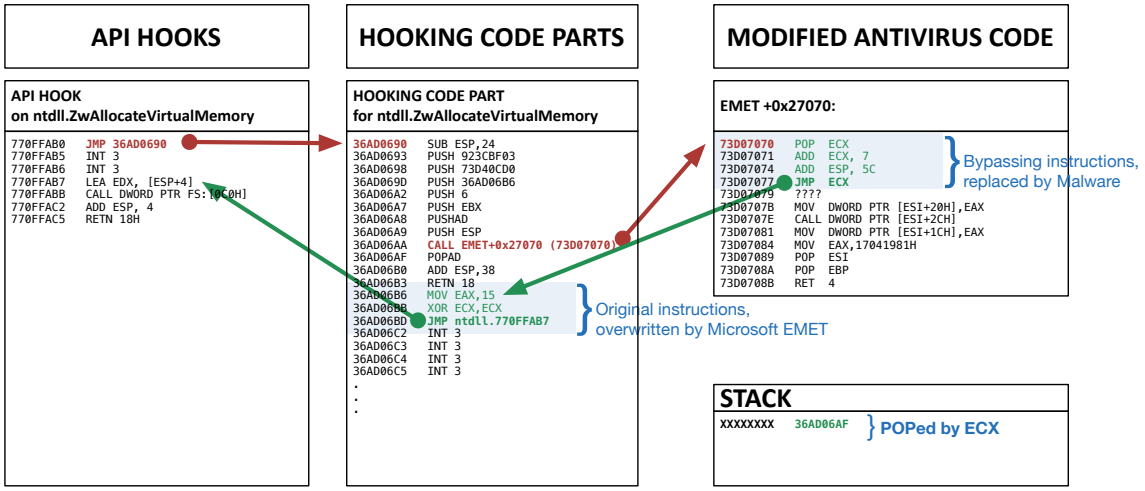


Figure 3.21: Microsoft EMET - Funneling - Modification of AV Control Code in process memory

## 3.5.2 Bypassing McAfee HIPS

The security mitigations of McAfee HIPS were discussed in Section 3.4. In this section, we will discuss some possibilities of how an attacker may still bypass inline API hooking protection, and the HTTP Engine of McAfee HIPS.

### 3.5.2.1 Bypassing McAfee HIPS - Jump-Around API Hooking

McAfee HIPS uses the basic inline hooking method, using a five-byte jmp instruction into antivirus monitoring code. The best-known bypassing technique, the trampoline function method, does work against the API hooking method of McAfee HIPS as well as against Microsoft EMET. Briefly, attackers can emulate the first instruction (five bytes in 32-bit systems), and jump the next instruction of a critical function prolog. In contrast, McAfee HIPS does not implement an Anti Detour protection. Figure 3.22 depicts a simple McAfee-aware Jump-around bypassing method. Moreover, our jump-around method to call a hooked function of ntdll.dll also works successfully against the API hooking method of McAfee HIPS as well as Microsoft EMET. More information about the new way to call a hooked function is provided in Section 4.2.2.2.1.

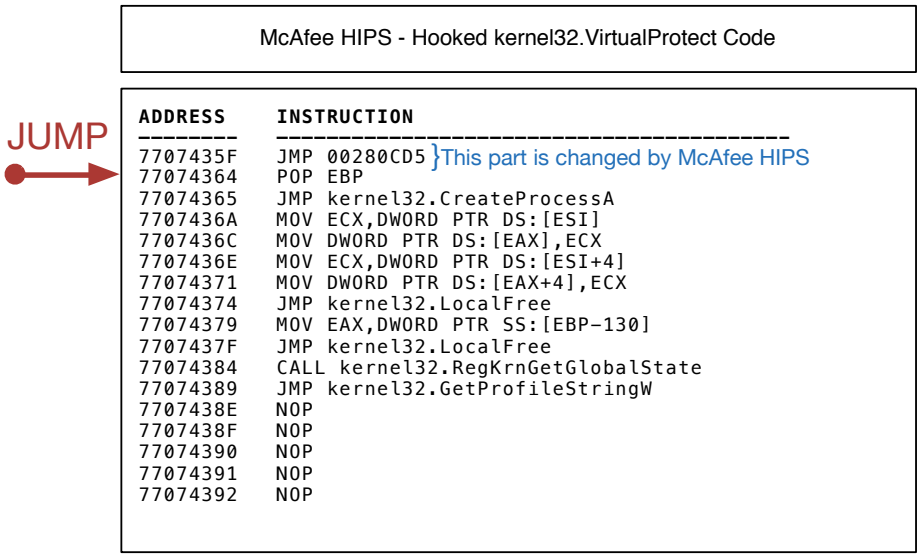


Figure 3.22: McAfee HIPS-aware Jump-around bypass

#### 3.5.2.2 Bypassing McAfee HIPS - Bypassing HTTP Engine

McAfee HIPS has an HTTP engine component to detect suspicious data sequences by checking incoming HTTP streams for malicious patterns. For example, if the shellcode of a web-based malicious file includes "0xAAAAAAAA", McAfee HIPS detects it as malware and produces the following output message: "6001: Suspicious Data Sequence in JavaScript." As a simple evasion, attackers can just change all occurrences of "0xAAAAAAAA" or similar suspicious data sequences with meaningless random bytes.

#### 3.5.2.3 Bypassing McAfee HIPS - Disabling AV Main Control Code (Funneling Method)

As mentioned before, the funneling method directly modifies antivirus control code in process memory, instead of removing antivirus hooks to disable the behavioral detection functionality of the antivirus product. In process memory protected by McAfee HIPS, every API hook jumps to its own hooking code section, similarly to Microsoft EMET. In HIPS, all hooking code sections have a call instruction, which is exactly the same. They call the main control code of McAfee HIPS that is located in 0x000000 offset address of either its hidden memory area. Thus, it is possible to find the address of the antivirus main control code by examining API hooks of McAfee HIPS on its critical functions, depicted in Figure 3.23.

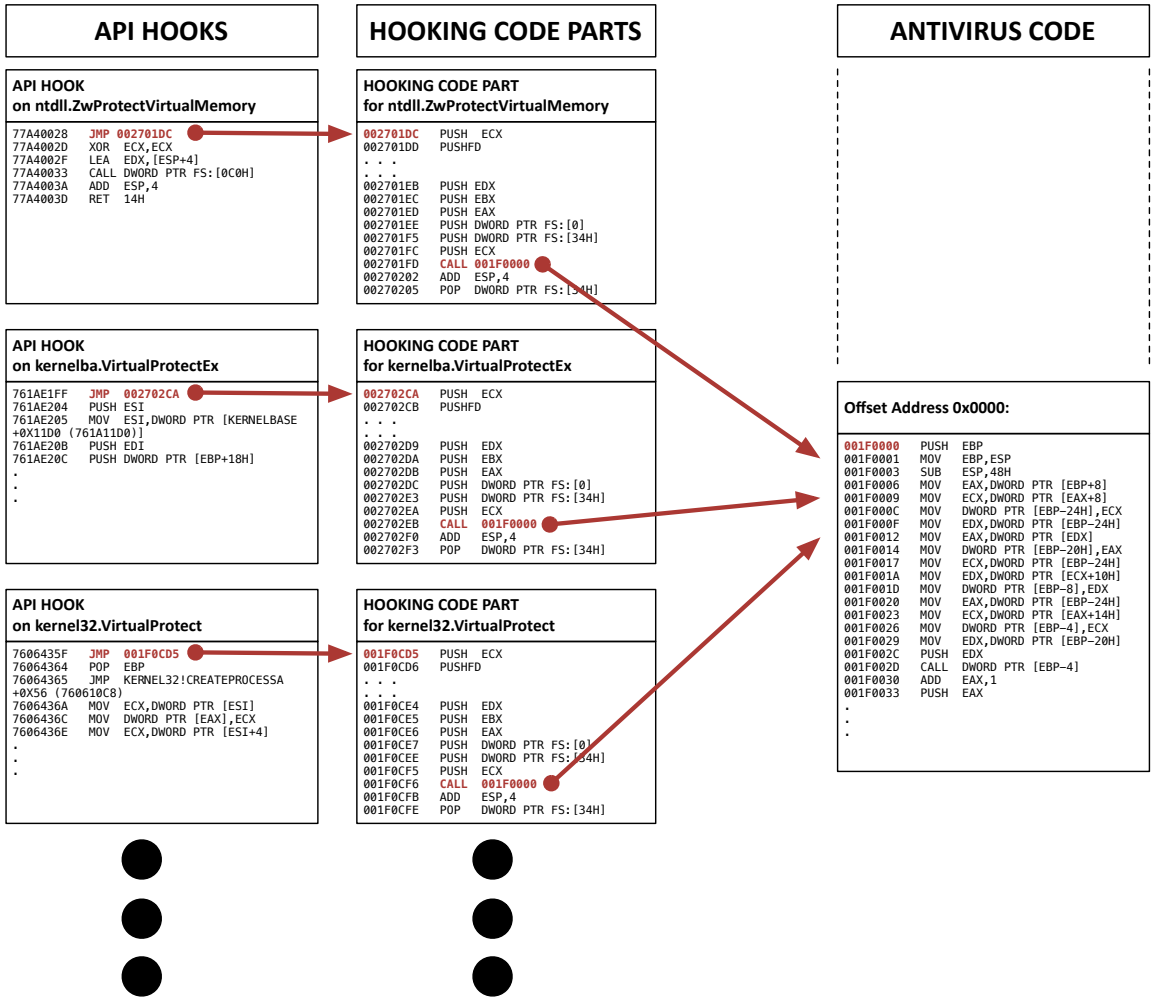


Figure 3.23: McAfee HIPS - Funneling - Invoked Main Control Code by every hooking code parts

```
0:020> !vprot 002701DC
BaseAddress:       00270000
AllocationBase:    00270000
AllocationProtect: 00000004  PAGE_READWRITE
RegionSize:        00001000
State:             00001000  MEM_COMMIT
Protect:           00000020  PAGE_EXECUTE_READ
Type:              00020000  MEM_PRIVATE
```

Therefore, attacking code may change the access protection of the antivirus code memory area and modify the antivirus control code in memory in order to disable its

functionality. The main control code of McAfee HIPS is located in one of two hidden memory areas in process memory. The access protection of the hidden memory area has been set as "Execute and Read" so malware has to change this memory region as writable in order to complete some code modifications. Figure 3.24 presents the modification of the main control code of McAfee HIPS. By changing just the first one (1) byte of the main control code to 0xCC, a RET instruction, a malware variant can eliminate all detection provided by McAfee HIPS, since the main control code will then immediately return to the next instruction in hooking code parts, and not produce any warning. In summary, it is possible to make a change to antivirus code in memory to execute the original code of a critical function without any execution of intrusion control code by McAfee HIPS. We have implemented this bypassing technique in the funneling malware variant.

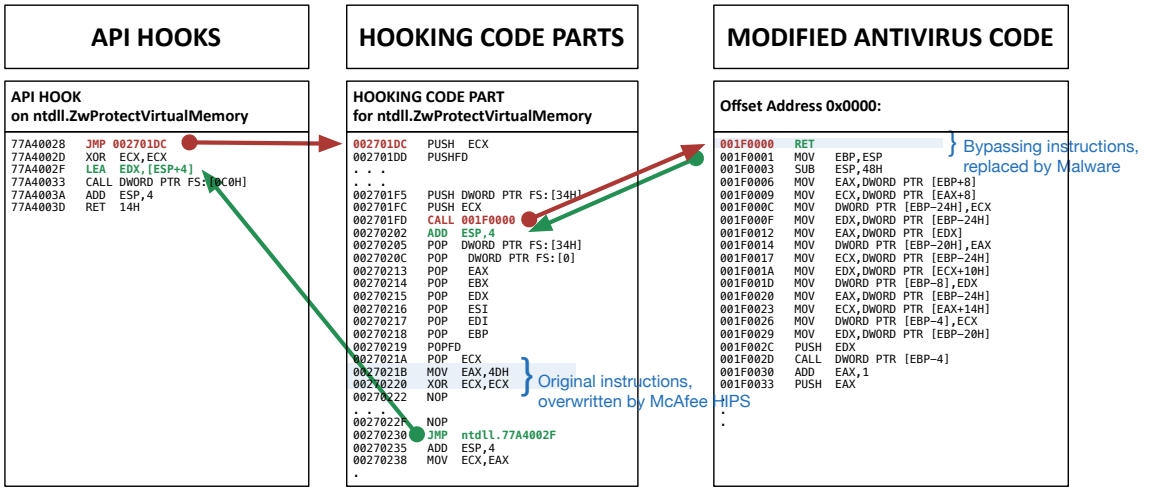


Figure 3.24: McAfee HIPS - Funneling - Modification of AV Control Code in process memory

### 3.5.3 Bypassing - Disabling API Hooks (Unhooking method)

Most behavioral-based antivirus products use API hooks to monitor critical functions on protected computer systems. API hooking allows antivirus products to branch to their behavioral-based control code, in order to perform protection checks and prevention activities. In order to avoid hooks, malware code may cut these branching connections between critical functions and the main control code of anti-malware, by using a live patching approach. The unhooking method replaces all of antivirus hooking code with the original code of every hooked function in the memory space

of a targeted process. Other protected processes will not be affected because the unhooking operation is performed only in targeted process memory space. Patching antivirus hooking code for all critical functions will prevent antivirus detection code from executing, so this patching disables the behavioral detection functionality of an active antivirus product.

The unhooking method for disabling API hooks can be broken into two phases, summarized below: a preparation phase and a malware-coding phase.

- The preparation phase consists of:
  - Finding the offsets of critical locations in modules
  - Finding antivirus-specific critical functions
  - Generating an original code table that includes the original code of critical functions and their beginning offsets.
- The malware coding phase consists of:
  - Including the original code table of malware code within a data structure
  - Finding base addresses of modules, using related Import Address Tables
  - Finding starting addresses of critical functions using their offsets
  - Changing access protection of related memory areas
  - Overwriting hooking code of critical functions with the corresponding original code.

#### 3.5.3.1 Preparation Phase

In the preparation phase, malware developers usually discover the offsets of critical locations and the list of critical functions for targeted antivirus products using several companion tools, such as debuggers. The offsets are converted to memory addresses in malicious code in order to call critical functions or read critical data structures of modules. Likewise, the list of critical functions helps to generate a table that consists of some information about altered sections of critical functions, such as the original code sections and their corresponding locations.

**3.5.3.1.1 Finding the offsets of critical locations in modules:** Malware usually employs several critical functions and data structures during an attack. Knowing the addresses of critical functions are necessary in order to call them. It is similar for the usage of critical data structures of modules as well. However, the absolute

addresses are not embedded in malware code, except for non-ASLR modules. Even some antivirus products, such as Microsoft EMET, force protected systems to not load non-ASLR modules into specific addresses. Most modules are dynamically loaded into different memory regions each time, whereas the positions of functions in modules, the offset values, still stay fixed. The offset values are deterministic so that they still can be used by malicious code. If malware obtains a certain memory address, then it will be able to calculate other important addresses using related offset values.

The offsets of memory locations can be obtained using a debugger, such as Microsoft WinDbg. Having the offsets of IATs and functions can be very useful during a malware development. When attached a running process of one target application, malware developers are able to run several commands in order to collect necessary offsets. Figure 3.25 presents example WinDBG commands to obtain offsets of some critical functions. "Command #1" gives the base address of kernel32.dll, whereas "Command #2" gives the absolute address of kernel32.VirtualProtect. The result of subtracting the module base address from this value is the offset of kernel32.VirtualProtect, 0x110c8.

```
0:019> ? kernel32  ----------------------------> COMMAND #1
Evaluate expression: 1985282048 = 76550000

0:019> ? kernel32!VirtualProtect  --------------> COMMAND #2
Evaluate expression: 1985351880 = 765610c8

0:019> ? kernel32!VirtualProtect - kernel32  ---> COMMAND #3
Evaluate expression: 69832 = 000110c8
```

Figure 3.25: An example critical function offset, collected by using Microsoft WinDbg

Moreover, Figure 3.26 depicts example WinDBG commands used to discover an entry offset in the IAT, a critical data structure. "Command #4" gives the offset of the IAT, 0x10000, as well as the size of the IAT, 0xDF0 for kernel32.dll. Using this informations, a list of the imported functions of kernel32 can be queried by "Command #5". As seen in the corresponding output, the first column on the right is the absolute address of the IAT's entries. "Command #6" gives the offset of the IAT entry for kernelbase.VirtualProtect, 0x10918, by subtracting the base address of kernel32.dll from the absolute address of the IAT entry for kernelbase.VirtualProtect.

```
0:019> !dh kernel32   ---------------------------> COMMAND #4

[Snipped]

      0 [       0] address [size] of Bound Import Directory
  10000 [     DF0] address [size] of Import Address Table Directory
      0 [       0] address [size] of Delay Import Directory

[Snipped]



0:019> dps kernel32+10000 kernel32+10000+DF0  ---> COMMAND #5

76560000  77276d39 ntdll!RtlUnwind
76560004  77276b2b ntdll!RtlCaptureContext
76560008  77294f8f ntdll!RtlCaptureStackBackTrace

[Snipped]

76560904  74dfde3e KERNELBASE!UnmapViewOfFile
76560908  74dfe365 KERNELBASE!VirtualAlloc
7656090c  74dfe2c8 KERNELBASE!VirtualAllocEx
76560910  74dfe2aa KERNELBASE!VirtualFree
76560914  74dfe174 KERNELBASE!VirtualFreeEx
76560918  74dfe326 KERNELBASE!VirtualProtect

0:019> ? 76560918-kernel32   ---------------> COMMAND #6
Evaluate expression: 67864 = 00010918  -----> The offset of IAT entry for kernelbase.VirtualProtect
```

Figure 3.26: An example critical data offset, collected by using Microsoft WinDbg

**3.5.3.1.2 Finding critical functions of antivirus products:** System functions considered as critical functions are usually different for every antivirus product. In other words, each antivirus product possibly has its own critical functions list to monitor. Therefore, a generic list of critical functions cannot be used by malware. Indeed, generation of critical function lists is antivirus-specific. A malware variant can be developed against multiple antivirus products by including more than one list of critical functions. In Sections 3.3.2, and 3.4.2, we provide the lists of critical functions for two sample antivirus products. Both lists have some similarities and differences in the functions selected as critical, depicted in Figure 3.27. For example, critical functions usually belong to different modules, including ntdll.dll, kernelbase.dll and kernel32.dll. McAfee HIPS monitors twelve (12) modules by hooking over fifty functions, whereas Microsoft EMET tracks only three (3) modules via a similar number of hooked functions. Thus, malware developers need to know the list of critical functions to implement antivirus-specific unhooking code. In summary, finding the hooked functions of an antivirus is an important phase in the development process of evasive malware.

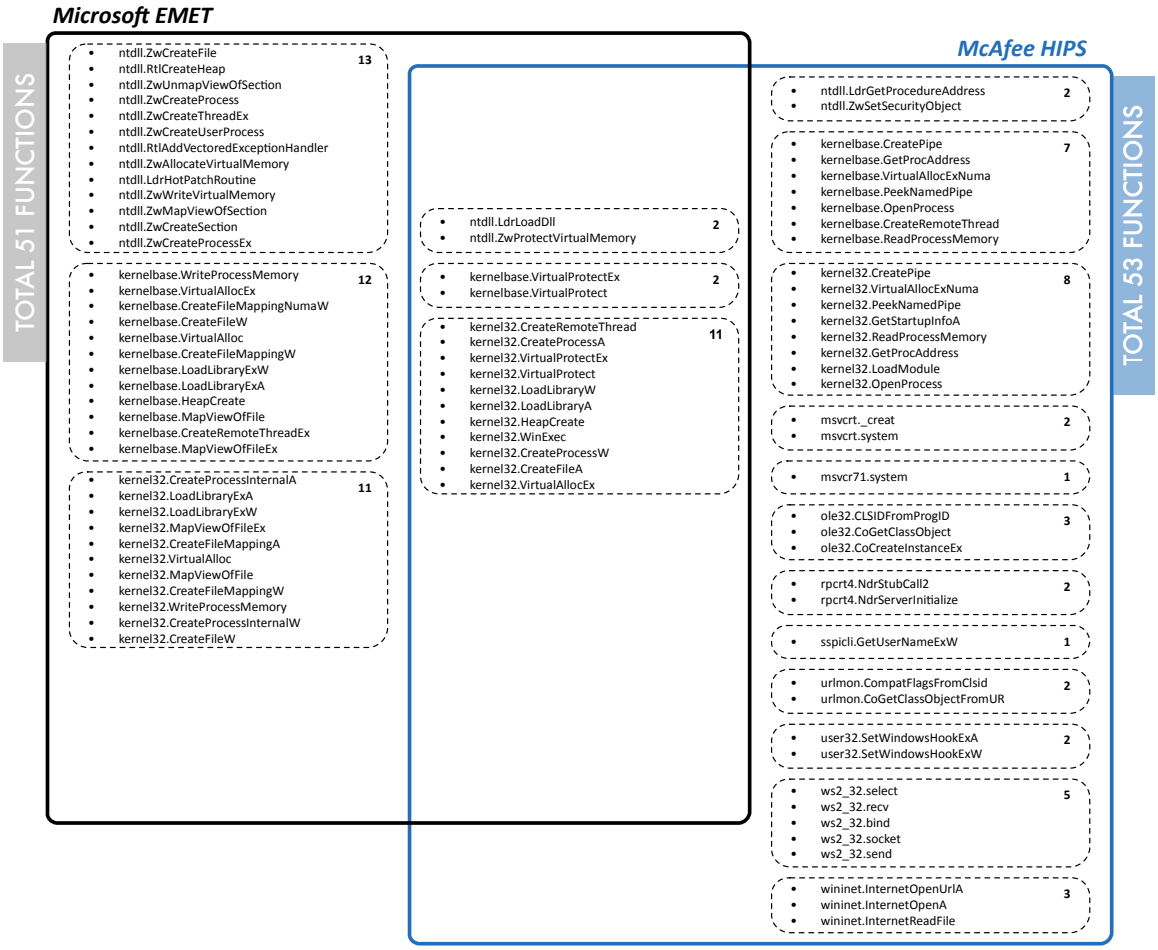


Figure 3.27: Comparing Critical Functions, hooked by Microsoft EMET and McAfee HIPS

There are several methods to obtain the list of critical functions of an antivirus product. API hooks are used for tracking critical functions so they are strong indicators of whether a function is critical. Antivirus products change the original prologue code of critical functions in memory at runtime to insert their API hooks. In order to track code modifications in memory by an antivirus in real-time, we have implemented a debugging tool using the Intel Pin framework. Our just-in-time debugging tool detects inserted antivirus hooks by discovering runtime modifications to loaded module codes in process memory. A simple hook consists of a five-byte jmp instruction. Therefore, our detection tool immediately stores the first 8 bytes of every function of a module in a data structure, when the module is loaded. Then it compares the stored original values with ones in memory to detect whether a modification has happened. They are mismatched if a running antivirus overwrites these

bytes with API hooking code. Due to performance concerns, these prologue comparisons are performed every 2000 instructions by the detection tool. A critical functions list can be generated in a very short time using the output of the detection tool, as depicted in Figure 3.28.

```
[Snipped]

KERNELBASE.dll::ReadProcessMemory has been changed!
Original: 8B FF 55 8B EC 8D 45 14  >> Modified: E9 1A 2C D7 8C 8D 45 14
NEW HEX CODE:
KERNELBASE.dll::0x76E2DFC8  E9 1A 2C D7 8C        jmp 0x3ba0be7                      Jumping into ::
KERNELBASE.dll::0x76E2DFCD  8D 45 14              lea eax, ptr [ebp+0x14]
KERNELBASE.dll::0x76E2DFD0  50                    push eax


KERNELBASE.dll::VirtualProtect has been changed!
Original: 8B FF 55 8B EC FF 75 14  >> Modified: E9 21 2A D7 8C FF 75 14
NEW HEX CODE:
KERNELBASE.dll::0x76E2E326  E9 21 2A D7 8C        jmp 0x3ba0d4c                      Jumping into ::
KERNELBASE.dll::0x76E2E32B  FF 75 14              push dword ptr [ebp+0x14]
KERNELBASE.dll::0x76E2E32E  FF 75 10              push dword ptr [ebp+0x10]

[Snipped]
```

Figure 3.28: A snip of an output of the API hooks detection tool, developed by using the Intel PIN framework

**3.5.3.1.3 Original Codes Tables:** After a list of critical functions for a specific antivirus product is exposed, an informative table that includes function entry offsets and original code can be prepared offline for a specific Windows version and system architecture, such as 32-bit, 64-bit, and Wow64. Figure 3.29 presents a sample of an original code table. A simple API hook changes at least the first five bytes of a critical function's code because an unconditional jump instruction is five bytes. Malware needs a list of the original machine code of critical functions in order to put them back when overwriting antivirus hooks. The location of original code is also needed for this purpose. Malware may have only the beginning offsets of critical functions, instead of their static beginning addresses, due to ASLR protection. Offsets, which are sometimes called relative virtual addresses (RVA), are used to define the function locations in memory. In order to obtain a specific absolute address of a memory location, an offset value is added to a base address. The table of original code and their offsets can be stored in a special data structure embedded in malware.

```
DDRESS    RVA      NAME              MODIFICATION
--------- -------  --------------------------------    ------------------------------------------------------------------------
0x77E70028 0x20028  ntdll.dll::ZwProtectVirtualMemory    Original: B8 4D 00 00 00 33 C9 8D  >> Modified: E9 AF 01 34 8C 33 C9 8D
0x77E71B8C 0x21B8C  ntdll.dll::NtSetSecurityObject       Original: B8 69 01 00 00 33 C9 8D  >> Modified: E9 E6 E8 D2 8B 33 C9 8D
0x77E801AA 0x301AA  ntdll.dll::LdrGetProcedureAddress    Original: 8B FF 55 8B EC 6A 00 FF  >> Modified: E9 4A 05 33 8C 6A 00 FF
0x77E8C43A 0x3C43A  ntdll.dll::LdrLoadDll                Original: 8B FF 55 8B EC A1 EC F6  >> Modified: E9 C1 3F D1 8B A1 EC F6
0x76E27838 0x7838   KERNELBASE.dll::CreatePipe           Original: 8B FF 55 8B EC 83 EC 3C  >> Modified: E9 3A 88 38 8D 83 EC 3C
0x76E283C8 0x83C8   KERNELBASE.dll::PeekNamedPipe        Original: 6A 1C 68 A0 75 E5 76 E8  >> Modified: E9 5B 8B D7 8C 90 90 E8
0x76E2DFC8 0xDFC8   KERNELBASE.dll::ReadProcessMemory    Original: 8B FF 55 8B EC 8D 45 14  >> Modified: E9 1A 2C D7 8C 8D 45 14
0x76E2E109 0xE109   KERNELBASE.dll::VirtualAllocExNuma   Original: 8B FF 55 8B EC 83 7D 0C  >> Modified: E9 EB 29 D7 8C 83 7D 0C
0x76E2E1FF 0xE1FF   KERNELBASE.dll::VirtualProtectEx     Original: 8B FF 55 8B EC 56 8B 35  >> Modified: E9 C6 20 38 8D 56 8B 35
0x76E2E326 0xE326   KERNELBASE.dll::VirtualProtect       Original: 8B FF 55 8B EC FF 75 14  >> Modified: E9 21 2A D7 8C FF 75 14
0x76E2E505 0xE505   KERNELBASE.dll::OpenProcess          Original: 8B FF 55 8B EC 83 EC 20  >> Modified: E9 30 29 D7 8C 83 EC 20
0x76E336AC 0x136AC  KERNELBASE.dll::CreateRemoteThread   Original: 8B FF 55 8B EC FF 75 20  >> Modified: E9 F5 CD 37 8D FF 75 20
0x771E0E00 0x10E00  kernel32.dll::GetStartupInfoA        Original: 8B FF 55 8B EC 64 A1 18  >> Modified: E9 7D F8 FC 8C 64 A1 18
0x771E1986 0x11986  kernel32.dll::OpenProcess            Original: 8B FF 55 8B EC 5D EB 05  >> Modified: E9 38 F4 9B 8C 5D EB 05
0x771E435F 0x1435F  kernel32.dll::VirtualProtect         Original: 8B FF 55 8B EC 5D E9 5E  >> Modified: E9 71 C9 9B 8C 5D E9 5E
0x771E492B 0x1492B  kernel32.dll::LoadLibraryW           Original: 8B FF 55 8B EC 6A 00 6A  >> Modified: E9 DB BC FC 8C 6A 00 6A

[Snipped]
```

Figure 3.29: A sample original codes table for McAfee HIPS

### 3.5.3.2 Malware Coding Phase

The original code and locations are usually carried by a basic unhooking malware variant in a special data structure, called an original code table. Otherwise, advanced malware may download the original code table from a remote server or directly read module files from victim computers to obtain originals. This table may also include more details about related modules, such as module names, and module sizes, excepting module base addresses. Because of ASLR protection, the base address of a loaded module will always change on every run. The base addresses of modules are used for calculation of absolute addresses so without base address information, knowing only function offsets will be not enough for malware. However, malware may still obtain interesting addresses using a memory information leak.

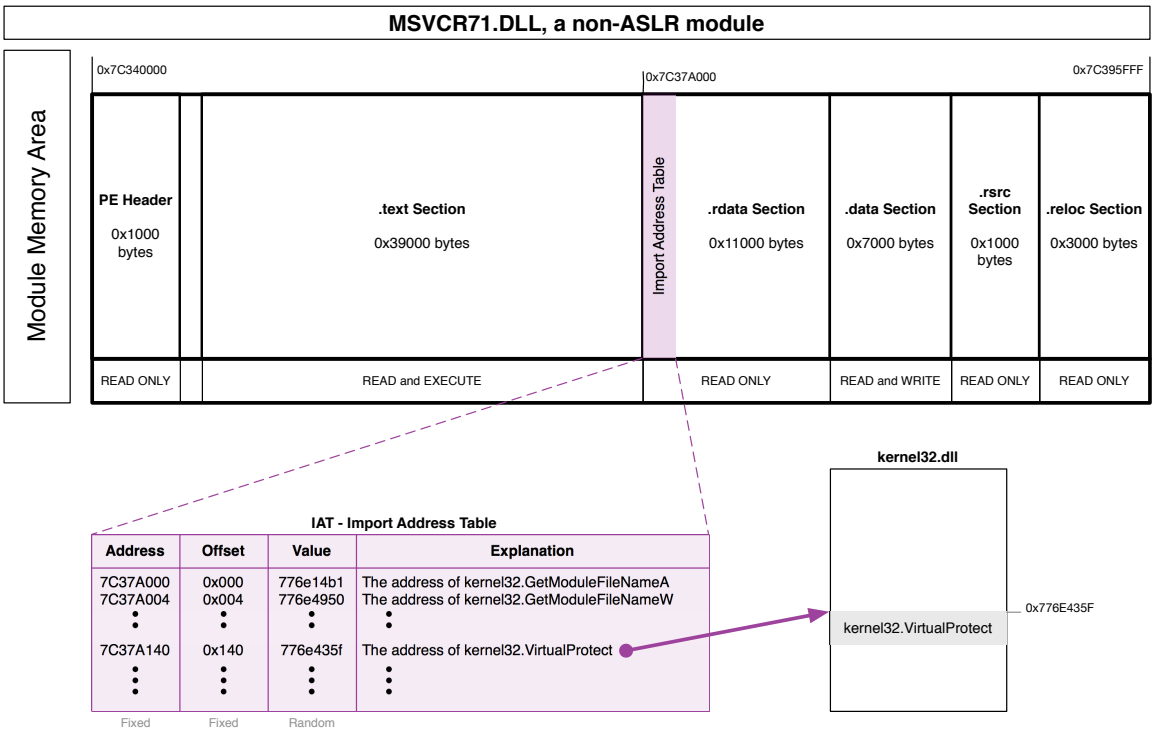


Figure 3.30: Import Address Table of msvcr71.dll, a non-ASLR module

In several vulnerabilities, the base address of a loaded module may be leaked or there may be a loaded non-ASLR module, which has a static base address. Having even one base address in process memory space could be enough for malware to predict the base addresses of some critical modules. Modules with ASLR support, such as kernel32.dll and ntdll.dll, are always loaded at different address in every run.

However, there is still a way to find their absolute addresses at runtime for malware if it knows the base address of any one module in memory. Figure 3.30 presents how to exploit the Import Address Table (IAT) of msvcr71.dll, a non-ASLR module, to find the address of kernel32.VirtualProtect. Like all non-ASLR modules, msvcr71.dll uses static addresses, so its IAT is always placed at the same memory address, 0x7C37A000 when msvcr71.dll is loaded in process memory. An IAT includes the addresses of all the imported functions of its owner module. Kernel32.VirtualProtect is imported by msvcr71.dll so a record that points to its address is always stored at the same offset in the IAT, namely 0x140. Thus, malware can find the absolute address of kernel32.VirtualProtect at 7C37A140, which is the sum of 0x7C37A000 and 0x140.

Moreover, the IAT offsets and functions are deterministic in a loaded module. Therefore, using the IAT of a leaked module, the base addresses and function addresses of other loaded modules can dynamically be found by malware at runtime. Although a function is not imported by a leaked module, such as msvcr71.dll, malware may still find the memory address of this function. In order to do this, the malware finds the absolute address of a function of another module by exploiting the IAT of the leaked module. This new module has to import a function of the same module that is not imported by msvcr71.dll. Then, by knowing one address in a module, malware can calculate other addresses in this module, like locations of the IAT and other needed functions. Figure 3.31 illustrates how to find a memory address by exploiting this deterministic behavior. Let us assume that the leaked module is msvcr71.dll and malware needs to discover the absolute address of kernelbase.VirtualProtect. Msvcr71.dll only imports the functions of kernel32.dll and ntdll.dll. Thus, kernelbase.VirtualProtect is not one of the imported functions by msvcr71.dll and its address is not stored in the IAT of msvcr71.dll. However, kernel32.dll does import kernelbase.VirtualProtect so malware may first find the address of a kernel32 function, such as kernel32.VirtualProtect, which is imported by msvcr71.dll too. Functions are located at the same offsets so kernel32.VirtualProtect is always placed at offset 0x110C8 in the memory region of a loaded image of kernel32.dll. Now, using this information, malware may calculate the base address of kernel32.dll by subtracting the function offset value from the absolute address of kernel32.VirtualProtect. If we assume that 0x765610C8 is the absolute address of kernel32.VirtualProtect, and then 0x769210C8 minus 0x110C8 is the base address of kernel32.dll, 0x76550000. In the IAT of kernel32.dll, the entry at offset 0x10918 points the location of kernelbase.VirtualProtect in process memory. If malware reads the value at address 0x76560918 then it obtains

the absolute address of kernelbase.VirtualProtect. Likewise, malware is able to discover other functions of kernel32.dll. For example, kernel32.HeapCreate is located at offset 0x14a3a so its address is 0x76564A3A, the sum of 0x76550000 and 0x14A3A.

Malware may exploit IATs to discover needed memory addresses for an attack in at run time. In response, some antivirus products, such as Symantec Endpoint Protection [157, 16], monitor system activities by using the EAT and IAT hooking methods. Such mitigation techniques are out of the scope of this thesis.

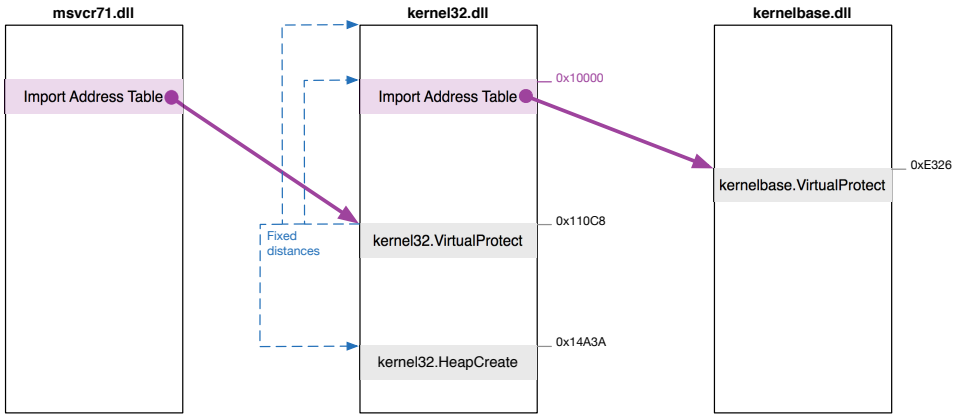


Figure 3.31: Finding Memory Locations by Exploiting Import Address Tables

After malware discovers the base addresses of critical modules, it easily calculates the addresses of all critical functions by adding their offsets. As a next step, the access protections of related regions in memory are changed by malware as writable in order to be able to write on these memory regions, which are normally only readable and executable. There are two options when performing this access change: 1) changing protection on the entire .code sections of modules, or 2) changing protection only on the individual memory regions where functions are located. Both methods work successfully. Then, using information from the original code table, malware may overwrite API hooks with original code to disable the API hooking protection of a running antivirus product.

We have started this chapter by providing a formal definition of the evasion techniques as well as an detailed insight to the Metasploit framework that is used for custom exploit creation throughout this thesis. Secondly, we have investigated the protection techniques implemented by the behavioral-based antivirus products for malware detection, specifically Microsoft EMET and McAfee HIPS. In addition, de-

tails about unhooking and funneling methods bypassing these protection techniques were discussed. The reader has established required knowledge to be introduced to the details about the implementation of these bypassing methods which are examined in the next chapter.

# Chapter 4

# Implementation and Results

In this chapter, we begin with describing our test environment and methodologies. Then, we discuss the implementation details of a bypass capable, antivirus-aware malware variant by examining each part and their functionalities to evade detection. In depth analysis of the unhooking and funneling methods are provided in order to demonstrate the effectiveness of these bypassing methods against behavioral-based detection. And finally, the chapter is concluded with the results section.

Today the Internet is one of the primary vectors for the delivery of malicious code by attackers. Generally, a crafted web page is designed to misuse one or more vulnerabilities of targeted operating systems, internet browsers, and/or their 3rd party plugins to deliver a malicious payload. An attacker may publically serve crafted web pages which generally look like a typical benign website. If a victim, who surfs on the Internet using a vulnerable Internet browser, opens such a crafted web page, the embedded attack exploits the browser's vulnerability and delivers malicious code to the victim's computer. If the malicious code has been written in an antivirus-aware way, the web-based attack will likely be successful, even though the victim has an installed and running antivirus product on the targeted system. In the real world, adversaries can also compromise a benign website or manipulate network traffic between server and client nodes to inject their malicious content as well. Also, crafted web advertisements or content can be located by malicious users on cloud websites. Such malignant web pages often are distributed via social media, or sent via phishing emails as a web link, attracting target users to open them.

In our web-based attack scenarios, the adversary has a HTTP web service, running on a Linux computer, which serves the harmful web pages. These antivirus-aware malicious web pages implement the unhooking and funnelling methods, bypassing

behavioral-based detection methods in user space, especially against two enterprise level antiviruses, Microsoft EMET and McAfee HIPS.

## 4.1 Test Environment and Methodologies

In the testing environment, in order to run anti-malware tests safely we use virtualization technologies to have victim and attacker computers on a controlled virtual network, illustrated in Figure 4.1. The virtual computers (VMs) are categorized into two different roles, including victim and attacker computers. All of the virtual victim computers are running 64-bit Microsoft Windows operating system, whereas a penetration specific Linux distribution is installed on the attacker computer. On the victim computers, an enterprise-level antivirus, leveraging behavioral-based detection methods is installed and configured for its full protection. Microsoft EMET and McAfee HIPS are installed on different virtual computers, never on the same one. Attacker virtual computers utilize the Metasploit Framework to provide a development and attack environment.

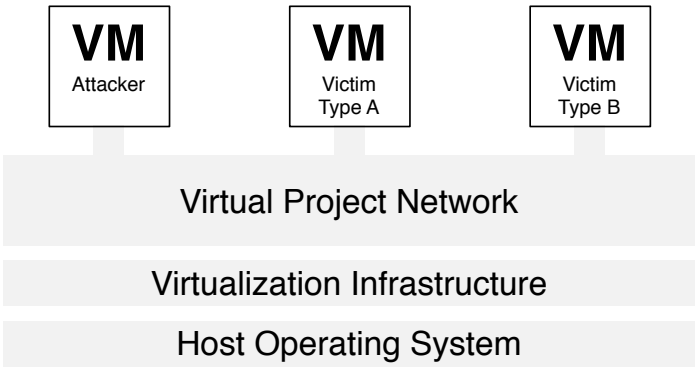


Figure 4.1: Project Virtualization Environment

- Attacker Computer:
  - Kali Linux, version 2.0, 64-bit
  - Metasploit Framework Community Edition, version 4.11.1-2015021201
- Victim Computer Type A:
  - Microsoft Windows 7 Professional, 64-bit, with Service Pack 1
  - Internet Explorer 8.0, 32-bit

  - Microsoft EMET, version 5.52.6156.38091
- Victim Computer Type B:
  - Microsoft Windows 7 Professional, 64-bit, with Service Pack 1
  - Internet Explorer 8.0, 32-bit
  - McAfee HIPS, version 8.0.0

Victim Computer Type A and Type B have the following vulnerabilities, which are exploited by antivirus-aware malware variants that we have developed in order to conduct several web-based test attacks.

1. **Vulnerability 01 - CVE-2013-1347:** Affects Microsoft Internet Explorer version 8 (IE8). It is a use-after-free (uaf) vulnerability which uses the fact that Microsoft Internet Explorer may access an improperly allocated or deleted object in memory, allowing attackers to perform a remote code execution attack [29]. As reported in May, 2013, the U.S. Department of Labor was compromised using this vulnerability, allowing attackers to inject malicious code into the US government website pages to hack visitors' computers running IE8. It was believed that the intended target users were possibly officials of the Department of Energy [36].

2. **Vulnerability 02 - CVE-2012-1875:** Microsoft Internet Explorer Same ID Property Remote Code Execution Vulnerability affects Microsoft Internet Explorer version 8. It is another uaf vulnerability, which results from a problem with object handling in IE, namely accessing freed objects in memory [27].

3. **Vulnerability 03 - CVE-2012-1876:** Microsoft Internet Explorer Col Element Remote Code Execution Vulnerability affects a wide variety of versions of Microsoft Internet Explorer, from version 6 to version 9, and even 10 Consumer Preview. Leveraging the heap spraying method, attackers can remotely execute their malicious code using of a heap overflow software bug in IE [28].

The Metasploit Framework has a number of exploits, which can employ the three vulnerabilities described above. The vulnerabilities above have already been fixed by Microsoft in newer versions of Windows and Internet Explorer, in particular via published patches or service packs for the affected Windows versions.

- For CVE-2013-1347, "MS13-038: Security update for Internet Explorer" was published on May 14, 2013 [162]
- For CVE-2012-1875, and CVE-2012-1876 vulnerabilities, "MS12-037 Cumulative Security Update for Internet Explorer" was published on June 12, 2012 [161]

Our methodology to generate a new malicious variant bypassing protections:

1. We have taken an exploit module of Metasploit Framework, written in Ruby, which can generate malicious code for a vulnerability of Windows and Internet Explorer described above in order to exploit this vulnerability to deliver antivirus-aware code.
2. We have improved the generic Metasploit exploit module, by modifying its malicious code to be able to bypass protections of a specific/targeted antivirus product.
3. We have used the new improved exploit module with a few Metasploit payload modules, which may be done without any code modifications.

Therefore, we have chosen and improved three exploit modules of the Metasploit Framework, including ms12_037_ie_colspan [108], ms12_037_same_id [109], and ie_cgenericelement_uaf [110] to test the effectivenesses of the unhooking and funneling methods against two behavioral-based antivirus products.

- **ms12_037_same_id:** MS12-037 Microsoft Internet Explorer Same ID Property Deleted Object Handling Memory Corruption
- **ms12_037_ie_colspan:** MS12-037 Microsoft Internet Explorer Fixed Table Col Span Heap Overflow
- **ie_cgenericelement_uaf:** MS13-038 Microsoft Internet Explorer CGenericElement Object Use-After-Free Vulnerability

These three exploit modules deliver their shellcode into the heap. In order to place their shellcode, ms12_037_ie_colspan and ms12_037_same_id use a "Heap Spraying" technique, whereas ie_cgenericelement_uaf uses "Mstime No-Spray" technique [174], which only can work against Microsoft Internet Explorer 8 or prior versions.

| MSF Exploit | Vulnerability | Microsoft Patch |
|---|---|---|
| ms12_037_ie_colspan | CVE-2012-1876 | MS12-037 |
| ms12_037_same_id | CVE-2012-1875 | MS12-037 |
| ie_cgenericelement_uaf | CVE-2013-1347 | MS13-038 |

Table 4.1: Modified Exploit Modules of Metasploit Framework (MSF)

## 4.2 Design of Antivirus-aware Malware

Thus far, we have introduced the test and development environment, the malware files with their related vulnerabilities, and our methodology in order to create a new malware variant from a detectable Metasploit malware. From here on, we focus on describing antivirus-aware malware improved by arming a capability of bypassing behavioral-based techniques.

### 4.2.1 Malware Sections

We have used the Metasploit Framework in this project to obtain ordinary malware code, which does not implement techniques for avoiding behavioural detection. When web-based malware code targeting Internet Explorer is generated by Metasploit, it generally has both web code and shellcode sections. Figure 4.2 presents the basic sections of a Metasploit web-based malware file.

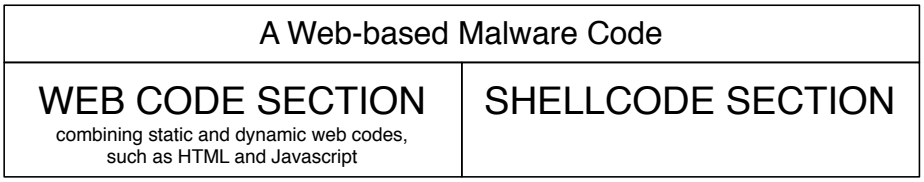


Figure 4.2: A web-based Malware file basis sections

The web code section is mostly some combination of HTML and Javascript, and is used for triggering a specific vulnerability which then delivers malicious code into a victim's process memory, mostly into the heap or stack. Even in the web code section, it is possible to generate a malicious ROP payload part on the fly. The delivered malicious code is known as shellcode. However, the shellcode section is hex code. A shellcode section may have several different sections, including sections that implement Pivot, Rop Payload, and Malicious Hex Payload, depicted in Figure 4.3. The Pivot part optionally creates a fake stack to employ the return behavior of a

function call to control the program execution flow. A ROP Payload part is created by using ROP gadgets, whereas a Malicious Payload part is a pure machine code. The Malicious Payload part can be developed in assembly language and compiled to machine code using Metasploit. In order to execute code in a memory area, the memory area has to be marked as executable. Therefore, the ROP Payload part prepares some registers and memory to execute injected malicious codes by evading the Data Execution Prevention (DEP) protection of Windows. However, the malicious Hex Payload part consists of the main damaging code, such as stealing information.

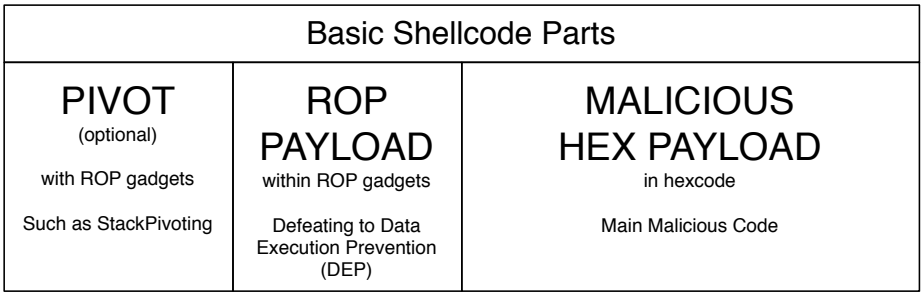


Figure 4.3: Basic Shellcode Parts

Even though Metasploit provides ROP Payload and Malicious Payload parts, most enterprise-level antivirus products are able to detect them using their behavioral-based detection methods. We have developed two new parts in order to add to bypassing capabilities to malware. Figure 4.4 presents shellcode sections improved with the new features. The new parts use both ROP gadgets and hex machine code. The Metasploit ROP Payload, used for bypassing DEP protection has been changed to a new ROP Payload, which does not trigger any antivirus protection, such as API hooking or Guard Page protection. Also, an extra malicious hex payload has been added to the shellcode section to disable all protections provided by a specific antivirus product, such as Microsoft EMET or McAfee HIPS. Depending on which method is implemented, such as unhooking or funneling, the new hex payload removes the antivirus traps, including API hooks, Guard Pages, or it modifies the antivirus code in the exploited process memory in user space to avoid the behavioral-based detections of the running antivirus product on a targeted system. Once the new hex payload is executed successfully, the antivirus product has been disabled. Therefore, any malicious hex payload of Metasploit Framework or a custom hex payload can be executed safely by malware without the need for a special modification in the malicious hex payload code. In the next section, we describe the structure and

implementation details of improved shellcode by parts.

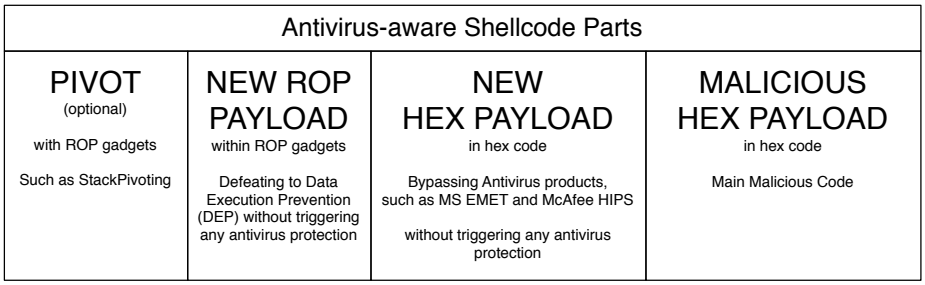


Figure 4.4: Antivirus-aware Shellcode Parts

## 4.2.2 Shellcode Parts

### 4.2.2.1 Stack Pivot Part

This part consists of ROP gadgets to perform the stack pivoting exploitation technique, commonly used by ROP-based malware. Stack pivoting pivots the stack pointer to an attacker-controlled memory area, on which a fake stack relies, generally on the heap. Then, by exploiting the program return behavior, malware with a crafted fake stack is able to have control on a targeted process execution flow [119].

Simply, the following ROP gadgets chain may be used for a stack pivoting, in Figure 4.5.

```
ROP Gadget - xchg REG, esp # retn
ROP Gadget - push REG # pop esp # retn
ROP Gadget - add esp, MALICIOUS_VALUE # retn
ROP Gadget - sub esp, MALICIOUS_VALUE # retn
ROP Gadget - mov esp, REG # retn
```

Figure 4.5: A Simple ROP Gadgets Chain For Stack Pivoting

### 4.2.2.2 New ROP Payload Part to Bypass DEP

We have implemented a new ROP payload to defeat DEP protection without triggering any antivirus protection. The new ROP payload employs a fake ntdll.ZwProtectVirtualMemory function to set executable right to the access protections of a memory region. Using a fake ntdll function emulation allows malware to bypass both DEP and Anti-Detour protections in an easy and generic way. To the

best of our knowledge, the fake ntdll function invocation that we use is a new exploitation technique to call hooked functions, and it has not been previously used in any malicious code appearing in academic or public malware domains.

The following list summarizes the pseudocode of the new ROP Payload.

- Stack pivoting
- Changing the access protection of the memory region, on which the shellcode relies, by using a fake ntdll.ZwProtectVirtualMemory function call without triggering an antivirus detection.
- Changing the instruction pointer value to the address of shellcode.

The following Sections 4.2.2.2.1 and 4.2.2.2.2 explain detailed information about the fake ntdll function invocation and a fake call stack usage of the new ROP payload.

**4.2.2.2.1 Fake Ntdll Function Invocation** It is a new evasion technique to invoke critical functions monitored by anti-malware using inline hooking methods. Let us first examine the generic ROP payload of Metasploit for bypassing DEP. Metasploit generates the following ROP gadgets chain using its ROP database for Windows 7. The Metasploit ROPdb database, storing specific ROP chains can be found in the directory, "Metasploit_Home_Director/data/ropdb". However, this generic ROP payload against DEP uses the hooked kernel32.VirtualProtect() function to change the access rights of the memory area, in which its shellcode is injected. Therefore, eventually, the generic ROP payload may be detected by any behavioral-based antivirus product, depicted in Figure 4.6.

An naive attack using a jump-around method to bypass the API hooking protection on kernel32.VirtualProtect cannot work against API hooks when kernel32.VirtualProtect is called in a ROP payload because kernel32.VirtualProtect invokes some other hooked functions, including kernelbase.VirtualProtect and ntdll.ZwProtectVirtualMemory, to modify access protections. Even though the attacker jumps over the first hook on kernel32.VirtualProtect, the other two hooks on kernelbase.VirtualProtect and ntdll.ZwProtectVirtualMemory will be still effective. Therefore, a behavioral-based antivirus with deep hooks can easily catch these jump-around methods at deeper levels. Figure 4.7 briefly depicts the control flow of kernel32.VirtualProtect function.

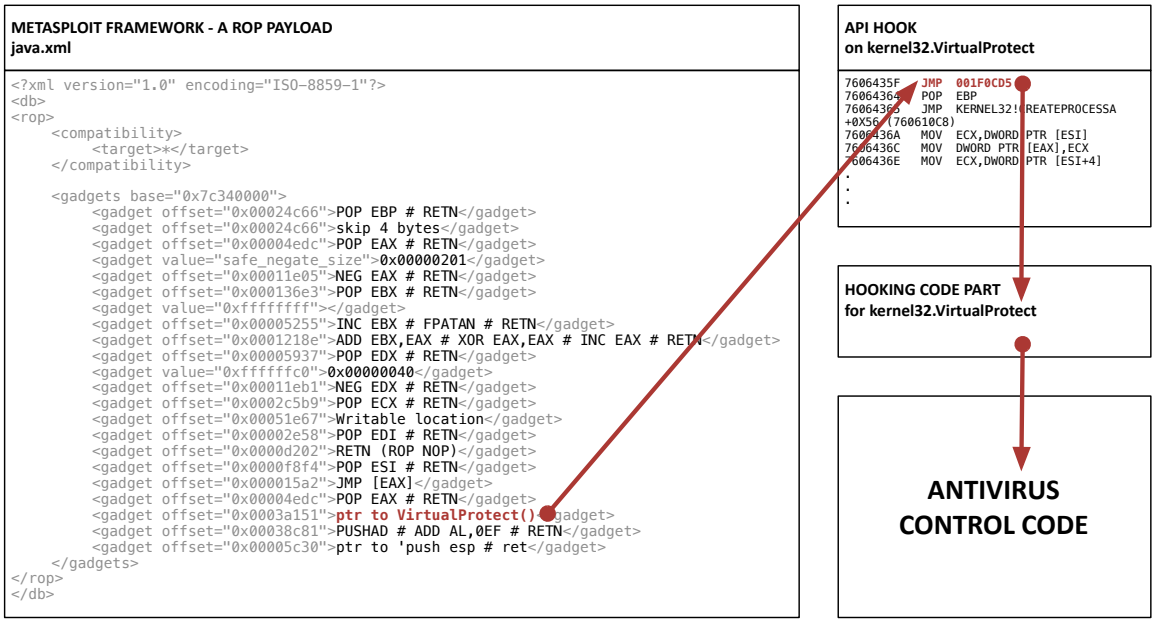


Figure 4.6: ROP Payload of Metasploit Framework

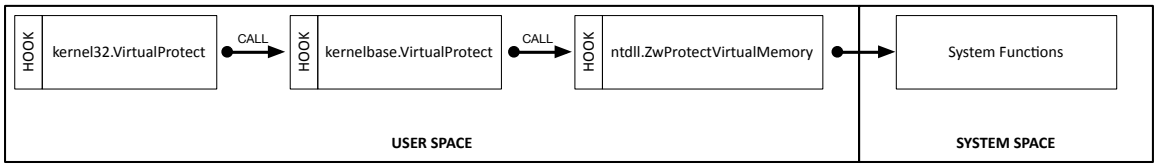


Figure 4.7: A Brief Kernel32.VirtualProtect Control Flow

As can be seen in Figure 4.7, ntdll.ZwProtectVirtualMemory is the last function in user space before code execution in system space. Therefore, attackers may directly call ntdll.ZwProtectVirtualMemory via the simple jump-around exploitation method to avoid deep level hooking. In response, some antivirus products implement anti-detour protections. For example, Microsoft EMET overwrites a random number of bytes as INT3 after its API hook to make the target address unpredictable for malware. If malware jumps to an address, overwritten with INT3 instruction, it raises an exception. The exception handler will forward the execution flow to its detection code so Microsoft EMET detects this malicious activity.

Thus, in the implementations, the new antivirus-ware payload invokes a fake ntdll.ZwProtectVirtualMemory to silently bypass both the DEP and Anti-Detour protections, instead of invoking kernel32.VirtualProtect. A fake ntdll function emulates a few instructions of a hooked ntdll function and simply jumps into one of the unhooked ntdll functions, depicted in Figure 4.8.

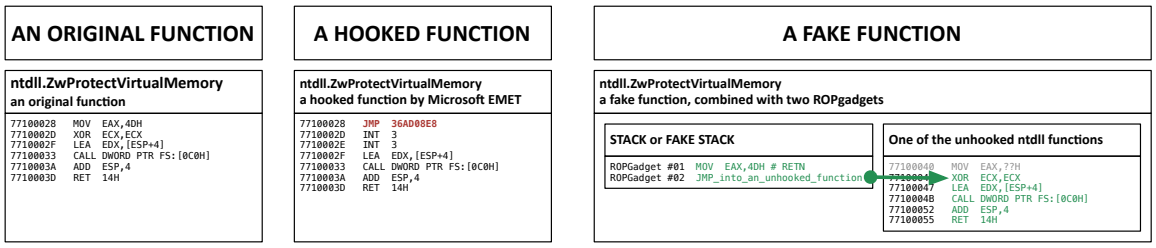


Figure 4.8: Comparison of An Original, A Hooked, and A Fake NTDLL Function in a WoW64 subsystem of Windows 7

Most important Ntdll functions use system call instructions to switch the code execution to system space. Other than these parts, many ntdll functions are very similar, as depicted in Figure 4.9. For example, ntdll.ZwProtectVirtualMemory, ntdll.ZwQuerySection, and ntdll.ZwQueryEvent are almost the same, except their first instructions, in a WoW64 subsystem of Windows 7. It is possible to find more functions, very similar to ntdll.ZwProtectVirtualMemory. Therefore, when a ntdll function is emulated, there is no need to use exactly the same function of ntdll. Also, only a very small number of ntdll functions are protected by antivirus products, for example Microsoft EMET and McAfee HIPS monitor only 15 and 4 ntdll functions, respectively. Thus, all of the unhooked ntdll functions are very good candidates to use in this improved jump-around-like method for malware because there is no API hooking or anti-detour protection in the code of these unprotected ntdll functions.

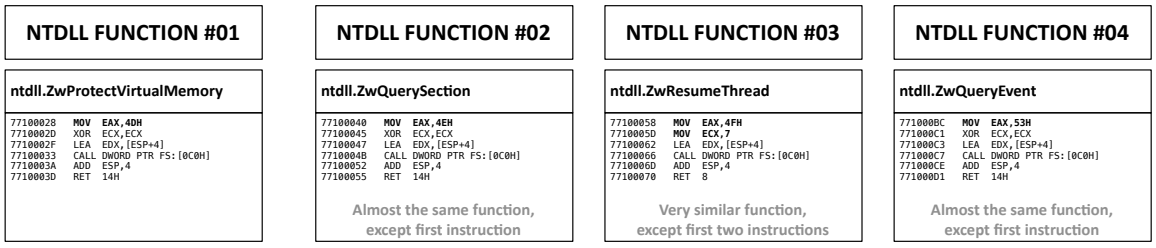


Figure 4.9: Comparison of Different NTDLL Functions in a WoW64 subsystem of Windows 7

In order to perform a fake ntdll function invocation, in 32-bit and WoW64 systems, emulating only the first instruction of a ntdll function may be enough. However, in a 64-bit system, more instructions would be emulated for the same ntdll function, depicted in the following figure 4.10.

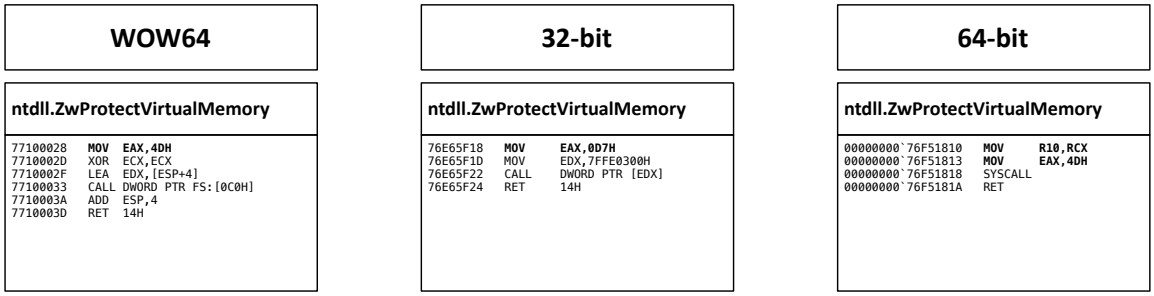


Figure 4.10: Comparison of NTDLL.ZwProtectVirtualMemory Functions in Windows 7 for different architectures

Figure 4.11 depicts a fake ntdll.ZwProtectVirtualMemory, which is used for modifications of the access protection of a memory region, in a Wow64 subsystem of Windows 7.

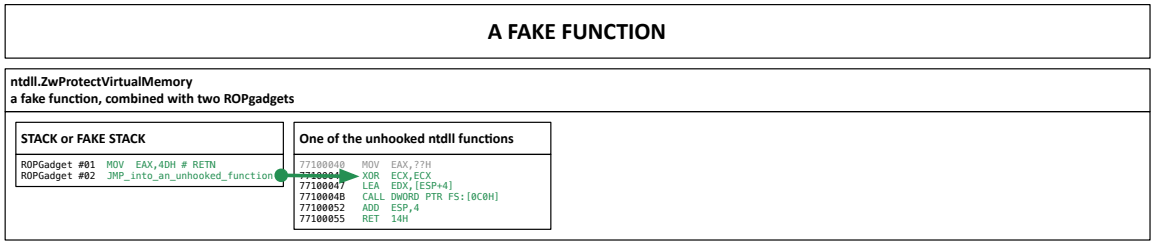


Figure 4.11: A Fake NTDLL.ZwProtectVirtualMemory Function in a WoW64 subsystem of Windows 7

For the new ROP payload, the pseudo code to invoke a fake ntdll.ZwProtectVirtualMemory with a fake call stack in order to change the access protection of the heap in a WoW64 subsystem or a 32-bit system of Windows is as follows:

- Finding and storing the address of an unhooked ntdll function as a fake function, similar to ntdll.ZwProtectVirtualMemory
- Storing the stack pointer (ESP) value, the address of the shellcode, located on the heap.
- Emulating "MOV EAX, 4Dh", the first instruction of ntdll.ZwProtectVirtualMemory.
- Jumping to the stored address of the unhooked function with a fake stack frame, which is a fake ntdll.ZwProtectVirtualMemory function.

**4.2.2.2.2 Fake Call Stack** Controlling a process stack allows attackers to execute malicious code in a ROP-based exploitation. The stack is a data structure, used for storing function parameters, local variables, and important registers for control flow, such as the instruction pointer (IP) and the base pointer (BP). Every function has its own stack frame in the stack. Normally, a caller function first pushes a number of function parameters and then pushes a current instruction pointer (IP) value into the stack when invoking another function. Malware mimics this behaviour when it invokes a function so the parameters of called functions are included in the malware's ROP payload. Thus, a ROP chain consists of return addresses, local variables, and parameter values.

ntdll.ZwProtectVirtualMemory is used for modifying access protections of memory regions. It is an undocumented function in the Microsoft kernel and has five function parameters, namely a process handle, a pointer to base address to protect, a pointer to number of bytes to protect, a new access protection value, and an old access protection value [98]. The related part of a ROP chain is given in Figure 4.12.

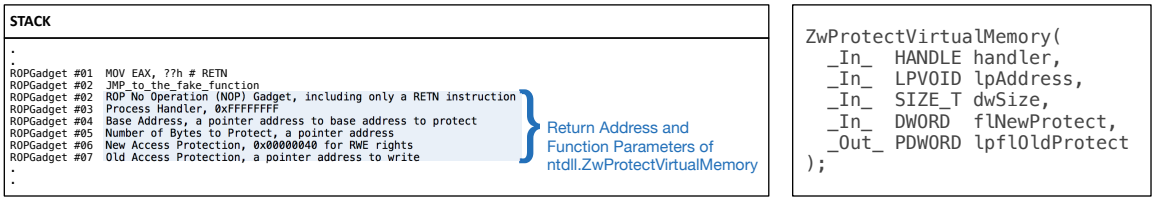


Figure 4.12: Function Parameters for NTDLL.ZwProtectVirtualMemory in a ROP chain

### 4.2.2.3 New Hex Payload Part to Remove Antivirus Traps

We have implemented a new Hex payload to eliminate antivirus traps without triggering any antivirus protection. It is written in Assembly language and translated to machine code using Metasploit. Similarly, the new Hex payload, like the ROP payload, uses a fake ntdll.ZwProtectVirtualMemory function method to make the access protections of a memory region writable in order to modify it. A fake ntdll function invocation works, regardless of the presence of anti-detour protection.

The following list summarizes the pseudocode of the new Hex Payload, which disables an antivirus detection feature by modifying its main control code or removing antivirus traps, such as API hooks and guard pages.

- Correcting the stack pointer (ESP) value
- Checking if a specific antivirus is running, such as Microsoft EMET.
- Fixing kernel32.dll, kernelbase.dll, and ntdll.dll
  - Changing access protections of memory regions of modules as writable
  - Removing guard pages if necessary
  - Removing API hooks on critical functions (only for the unhooking method)
- Modifying antivirus control code in process memory (only for the funneling method)

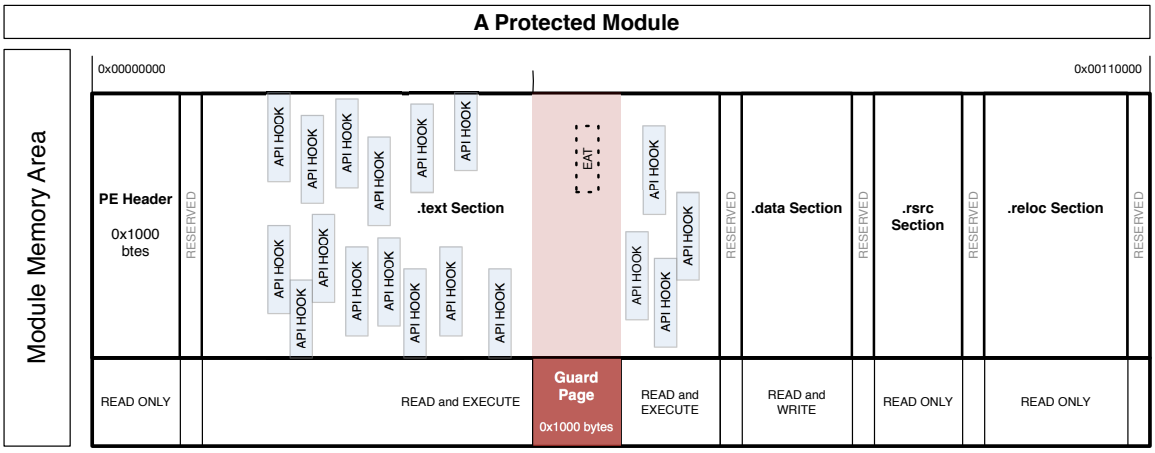


Figure 4.13: A module, protected by monitoring its critical functions and EAT

In the unhooking method, all antivirus hooks are removed by replacing them with original code, whereas in the funnelling method, none of antivirus hooks are patched because the main antivirus control code will be modified to disable its detection feature. However, in both methods, all page guards are removed if a page guard protection exists. Figure 4.13 depicts a protected module, whose critical functions and EAT are monitored using API hooks and a guard page.

#### 4.2.2.4 Regular Malicious Payload Part

After all of the antivirus hooks and guard page protections are removed by the previously described evasion shellcode, any regular malicious payload can be executed by malware and remains undetectable. For example, any Metasploit payload module that calls critical functions can now be used without any code modification. However, there is one exception, if a Metasploit payload spawns a new process, antivirus products may inject their protection code to the new spawned process. Such a payload will still need a code modification. Metasploit Meterpreter launches a new hidden Notepad process to migrate its malicious code from the targeted vulnerable process, like Internet Explorer, in case the user closes it. McAfee HIPS protects Notepad as well so it will detect this kind of malicious code, while Microsoft EMET does not.

## 4.3 Implementation of the Unhooking Method

Most behavioral-based antivirus products monitor critical system activities to detect suspicious ones. Malware usually uses several API functions to perform and support its malicious activities, such as allocating memory areas, changing access protections of memory regions, loading modules, having a network connection and so on. Therefore, these antivirus products use the API hooking method to monitor critical functions, commonly used by malware. In the unhooking method, malware removes all hooks of a running antivirus products in targeted process spaces, as illustrated in Figure 4.14.

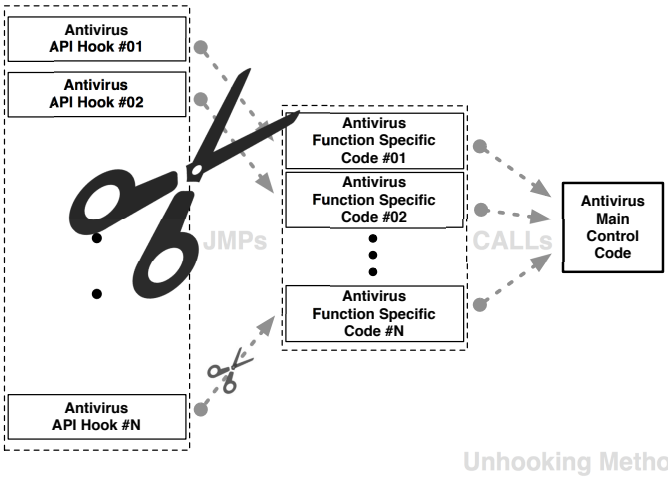


Figure 4.14: The Unhooking Method

For the unhooking method implementation, the malware variants that we have developed briefly have the following features.

- Triggering a specific vulnerability
- Delivering its shellcode, using a heapspray method
- Executing its shellcode on the heap
- Removing all hooks from critical functions in the targeted process memory
- Executing its malicious payload

In order to evaluate the unhooking method, we have developed an antivirus-aware malware variant with a special shellcode, which is armed with the unhooking method. The regular ms12_037_same_id exploit module of Metasploit has been used as a template malicious module. Ms12_037_same_id is a Metasploit exploit module, which exploits a use-after-free vulnerability only in Microsoft Internet Explorer 8. There is a flaw in the object handling of Internet Explorer 8 and it still accesses an object in memory after the object was freed. Thus, the vulnerability allows attackers to perform remote code executions on victim computers.

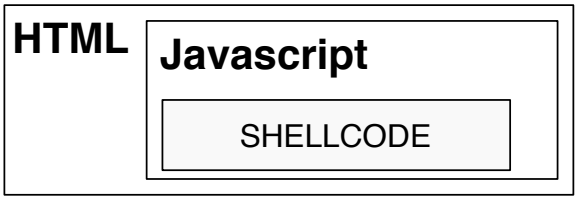


Figure 4.15: The layout of a ms12_037_same_id malware code

Ms12_037_same_id combines HTML and Javascript code, depicted in Figure 4.15. Both the shellcode and the exploitation libraries are placed in its Javascript part. The Javascript part triggers the vulnerability as well as delivering the malicious code to the victim computer. The ms12_037_same_id exploit module supports the heapspray attacking method with an embedded copy of the Javascript Heap Exploitation Library. This library provides precision heap spraying against legacy Internet Explorer and Windows versions [143]. Heapspraying is used for a malicious code delivery method in web-based malware. Javascript runtime and MSHTML engines use the same heap area, the default process heap, in Internet Explorer 8. Thus, using its javascript code, ms12_037_same_id sprays the heap with multiple copies of its shellcode to fill most parts of the heap with the same malicious code, depicted in Figure 4.16.

The ms12_037_same_id exploit module of Metasploit has been improved by adding a new ROP payload and an extra new hex payload in its shellcode. Figure 4.17

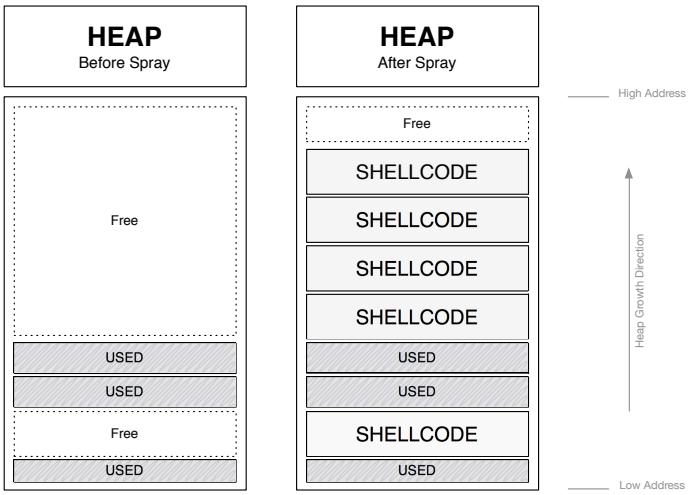


Figure 4.16: A heapspray attack

presents the layout of the new shellcode. The new shellcode has five parts. The first part is a "Malicious Hex Payload", and this part can be any regular Metasploit payload or a custom payload. It is not necessary that the malicious payload is an antivirus-aware payload. The next part, a "Padding" part consists of meaningless random values as it is used to obtain a fixed size shellcode. A new "ROP Payload" that defeats DEP protection follows the padding part. This ROP payload includes a new ROP chain that changes access protection of the heap to be able to execute the hex payloads, including the new unhooking payload, a nop slide payload, and a malicious hex payload. It uses the techniques that are discussed in Section 4.2.2.2. The unhooking method is implemented in the new "Unhooking Hex Payload" part. The size of the new shellcode is 8192 bytes. It is larger than the shellcode size of the original ms12_037_same_id since the unhooking hex payload carries all of the original prologues of the critical functions that are hooked by a running antivirus product. The API hooks are usually placed in the code sections. Code sections have read and execution rights in memory, and they are not writeable. Therefore, the unhooking payload first fixes the access protection of the code sections of relevant modules by using a technique explained in Section 4.2.2.3. Then it replaces the original code of the hooked functions with API hooks. More information about the API hooking bypassing method is provided in Section 3.5.3. The last part is a "NOP Slide Payload" in machine code, used to transfer the execution flow to the malicious hex payload of the next shellcode copy.

The heap manager typically allocates adjacent memory areas for each allocation

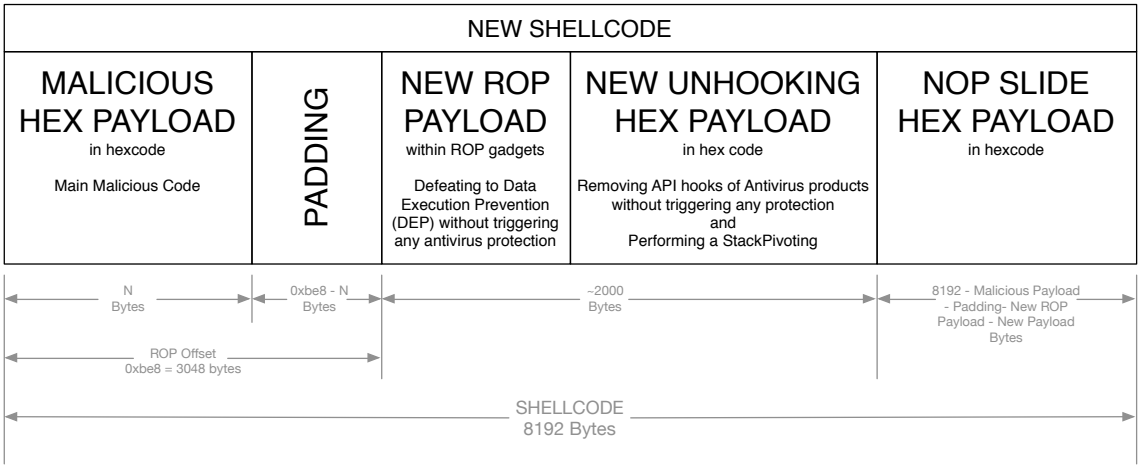


Figure 4.17: The layout of the new shellcode for the unhooking method

to prevent the memory fragmentation on the heap in Windows 7. A significant number of the shellcode copies are sequentially allocated on the default process heap, depicted in Figure 4.18. The Javascript Heap Exploitation Library allows a precision heap spraying attack. Thus, one of these shellcode copies will be at a predictable address so the new ms12_037_same_id variant will be able to jump to the new ROP payload that is placed on a predictable memory location on the heap. The address 0x1C180024 likely contains a copy of the shellcode so a new ROP payload begins at the address 0x1C180C0C. Ms12_037_same_id employs a fake vtable to gain control over the instruction pointer (IP). Therefore, the new malware variant first creates a fake vtable that points to this address 0x1C180C0C and then triggers the use-after-free vulnerability using Javascript code.

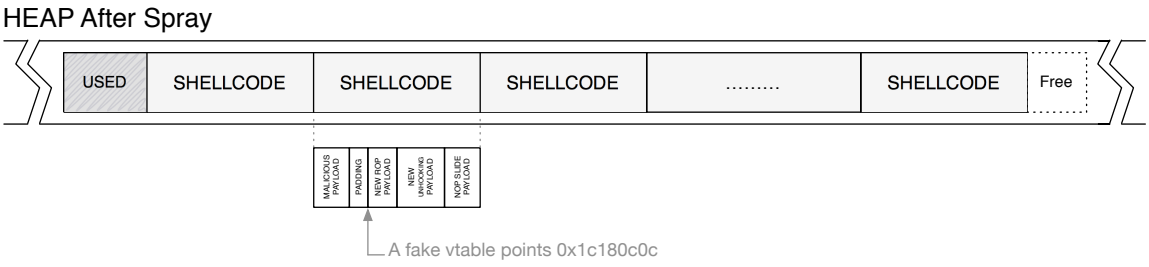


Figure 4.18: HEAP after a heapspray attack

The Mshtml module is the rendering engine of Microsoft Internet Explorer [88]. The vulnerability [67] discovered by Code Audit Labs is triggered in the following code of CElement::Doc function of mshtml.dll, shown in Figure 4.19. The ECX register points to the address of the deleted object and in this address there is a fake vtable that is crafted by ms12_037_same_id, when the CVE-2012-1875 vulnerability is exploited. In Figure 4.20, the code of mshtml.CElement::Doc function can be seen.

Instruction 1) The 4-byte value, 1c180c0c, at the address contained in ECX register is moved into the EAX register.

Instruction 2) The 4-byte value at the address contained in (EAX +70h) is moved into the EDX register. The EDX register contains a ROP gadget because the EAX register points to the ROP payload part of a shellcode copy in memory.

Instruction 3) The instruction, call edx, invokes the code in this ROP gadget and the execution process of the shellcode starts.

```
(e4c.aa4): Access violation - code c0000005 (first chance)
First chance exceptions are reported before any exception handling.
This exception may be expected and handled.
eax=1c180c0c ebx=00000000 ecx=03ce3c48 edx=00000001 esi=03ce3c48 edi=051eb8f0
eip=7179b68f esp=051eb888 ebp=051eb894 iopl=0         nv up ei pl zr na pe nc
cs=0023  ss=002b  ds=002b  es=002b  fs=0053  gs=002b             efl=00010246

mshtml!CElement::Doc+0x2:
7179b68f 8b5070          mov     edx,dword ptr [eax+70h] ds:002b:1c180c7c=????????
```

Figure 4.19: The use-after-free vulnerability in Microsoft Explorer

```
0:014> u mshtml!CElement::Doc
mshtml!CElement::Doc:
7179b68d 8b01            mov     eax,dword ptr [ecx] --------> Instruction 1
7179b68f 8b5070          mov     edx,dword ptr [eax+70h] ----> Instruction 2
7179b692 ffd2            call    edx ------------------------> Instruction 3
7179b694 8b400c          mov     eax,dword ptr [eax+0Ch]
7179b697 c3              ret
```

Figure 4.20: The vulnerable function in mshtml that is the rendering engine of Microsoft Explorer

The original ms12_037_same_id of Metasploit uses "0x0c0c0c0c" as an address of the fake vtable. However, we have changed this address with an address that is not on the antivirus blacklists in order to bypass the basic Heapspray protections of behavioral-based antivirus products.

An attack execution begins in the middle of the shellcode, the ROP payload part. Therefore, the order of the shellcode parts is chosen based on the heapspray attack method. As mentioned before, the shellcode copies are sequentially located on the heap. Thus, two copies of the shellcode in memory can be used for an attack, depicted in Figure 4.21. The ROP, unhooking and NOP Slide payload parts of "Shellcode #n" and the malicious payload of "Shellcode n+1" are executed respectively. The malicious payload can be successfully executed without any detection by an antivirus mitigation, after the previous payload parts remove API hooks of antivirus products, using the unhooking method.

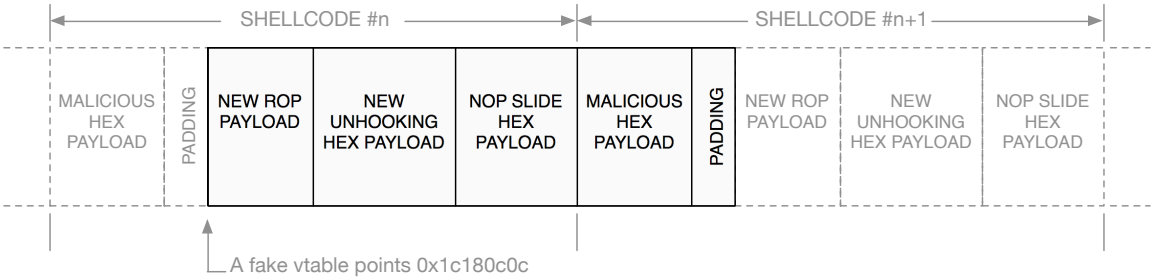


Figure 4.21: Fake vtable points to the address of a new ROP payload

## 4.4 Implementation of the Funneling Method

API hooking methods allows behavioral-based antivirus products to monitor critical systems in order to detect suspicious activities. However, it also provides certain sensitive information about the current protection to attackers. Malware can find the address of the main control code of a running antivirus product in memory by following API hooks in user space. Like a big funnel, all API hooks eventually jump to the main control code that performs the security mitigation checks. In the funneling method, malware directly attacks the main control code of running antivirus products in targeted process spaces to disable antivirus protections, as seen in Figure 4.22.

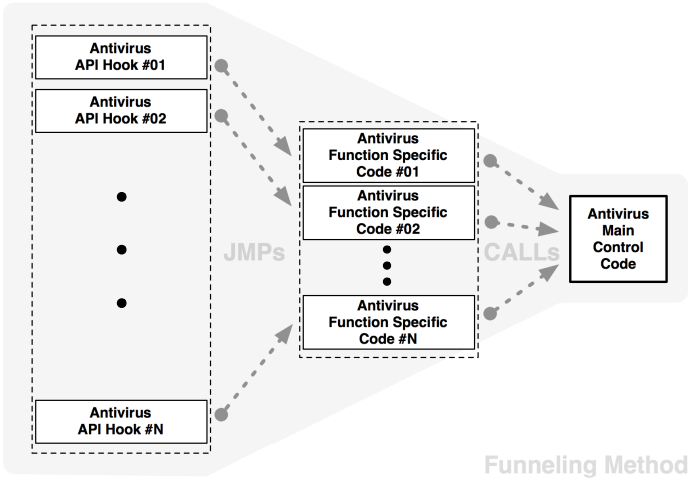


Figure 4.22: The Funneling Method

For the funneling method implementation, the malware variants that we have developed briefly have the following features.

- Triggering a specific vulnerability
- Delivering its shellcode, using the heapspray method
- Executing its shellcode on the heap
- Modifying the main control code of antivirus products in the targeted process memory
- Executing its malicious payload

In the malware code, the vulnerability is triggered twice:

1. the first one discloses the base address of mshtml module in order to bypass ASLR protection.

2. the second one obtains execute control and passes it to its shellcode, a malicious payload.

In order to evaluate the funneling method, we have developed another antivirus-aware malware variant with a new shellcode, in which the funneling method is implemented. Moreover, the new variant can bypass the Mandatory Address Space Layout Randomization (ASLR) Security Mitigation of Microsoft EMET as an improvement. Therefore, the original ms12_037_ie_colspan exploit module of Metasploit [108] and the exploit 24017 of Exploit Database [139] have been combined as a template malicious module. Ms12_037_ie_colspan is a Metasploit exploit module, which exploits a heap overflow vulnerability, CVE-2012-1876, in Microsoft Internet Explorer from version 6 to version 9. The ms12_037_ie_colspan of Metasploit Framework uses ROP gadgets from a specific version msvcr71.dll that is a module of Sun Java Runtime Environment (JRE).

In Sun JRE version 6 update 21 (6u21), msvcr71.dll has not been compiled with ASLR support so it is used in ROP-based attacks due to its predictable base address in memory. It is always loaded in the fixed address 0x7c340000. Metasploit's ms12_037_ie_colspan needs msvcr71.dll installed on the victim computer to exploit CVE-2012-1876 vulnerability successfully. As a result of the Mandatory ASLR Security Mitigation of Microsoft EMET, msvcr71.dll is always loaded in different base addresses on a protected system. Thus, the original ms12_037_ie_colspan cannot work on computers, protected by Microsoft EMET. However, the exploit with id number 24017 is an improved variant of ms12_037_ie_colspan of Metasploit Framework with a full ASLR and DEP bypass capability. The exploit 24017 allows attackers to generate ROP-gadgets on the fly using a memory leak, therefore, it cannot be affected by the Mandatory ASLR Security Mitigation and is able to bypass this protection. Fortunately, both ms12_037_ie_colspan and the exploit 24017 use the hooked VirtualProtect function of kernel32 and so they are detectable by behavioral-based antivirus products, monitoring critical functions via API hooks.

In the variant that we have developed, the ms12_037_ie_colspan exploit module of Metasploit has been improved by embedding a dynamic ROP chain construction method that is used in the exploit 24017. Also, as another improvement, a new ROP

payload, and an extra new hex payload have been added to its shellcode. As in the ms12_037_same_id exploit module, HTML and Javascript code are combined in the original ms12_037_ie_colspan and the exploit 24017, its improved version. Both of them also use a heapspray attack as a malicious code delivery method, depicted in Figure 4.18. Similarly, the Javascript part is used to trigger the vulnerability and to deliver the malicious code to the victim computer's memory. The vulnerability is triggered twice during an attack. In the first attempt, the vulnerability leaks the base address of the loaded mshtml module in process memory. After gaining the mshtml base address, the new malware variant creates a fake vtable and also constructs the ROP payload part of its shellcode on the fly using ROP-gadgets from the mshtml module, and sprays its shellcode copies through the heap. Then, in the second attempt, the new malware variant executes its shellcode.

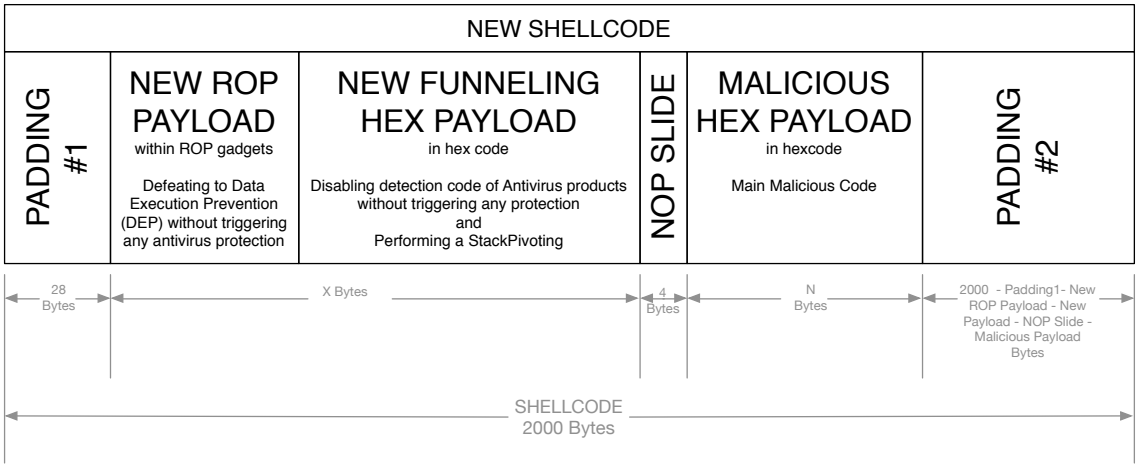


Figure 4.23: The layout of the new shellcode for the funneling method

Figure 4.23 depicts the shellcode of the new variant with its six parts. The first part is a 28-byte padding that is used to adjust the offset of the new ROP payload part. The "Padding #1" part is followed by the "new ROP Payload" part that will be generated on the fly during an attack. The new "ROP Payload" defeats DEP protection by changing access protection of the heap. Thus, the hex payloads, such as the new funneling payload, a nop slide payload, and a malicious hex payload, can be executed. The new ROP Payload uses the techniques which are discussed in Section 4.2.2.2. The new "Funneling Hex Payload" part has code implementing the funnelling methods. The main detection code of an antivirus product is usually located in the code section, which is not a writable region in memory. Therefore, the

funneling payload first changes the access protection of this code section as writable using the technique explained in Section 4.2.2.3. Then it modifies the original code of the main detection code in process memory to disable antivirus detection features. More information about techniques for disabling the main control code of Microsoft EMET and McAfee HIPS is provided in Sections 3.5.1.3 and 3.5.2.3, respectively. The following part is a "NOP Slide Payload" in machine code, and it is optional. A "Malicious Hex Payload" part can be any regular Metasploit payload or a custom payload. It is not necessary that the malicious payload be antivirus-aware. The last part, a second "Padding" part, consists of meaningless random values to obtain a fixed size shellcode.

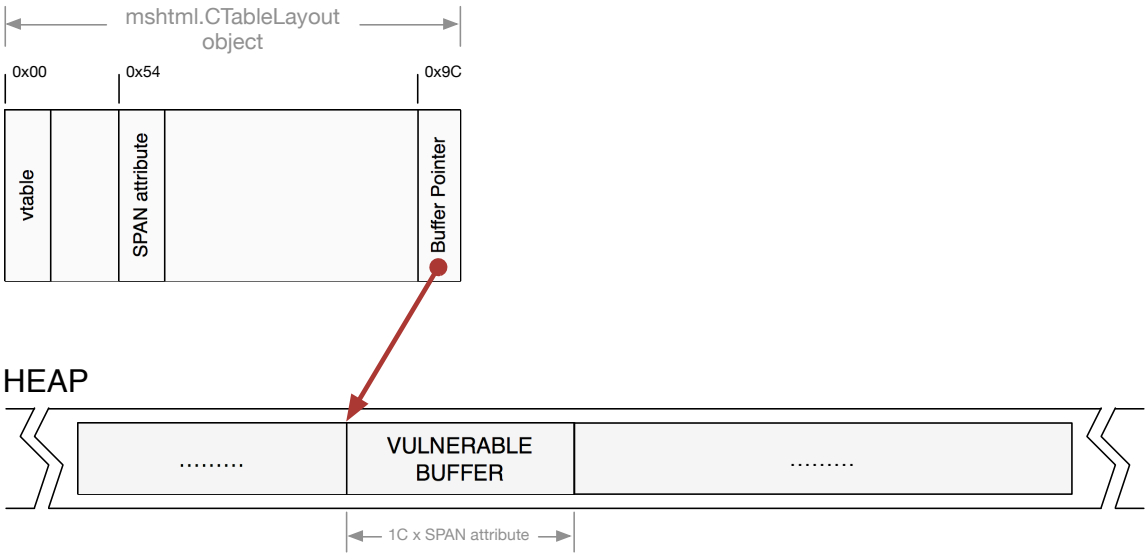


Figure 4.24: A CTableLayout object of mshtml module in Windows 7

The CVE-2012-1876 vulnerability was discovered and reported by the Vupen Security Research team in 2012. According to their technical analysis [136] that was published on July 10th, 2012, it is a heap overflow vulnerability in mshtml.dll, a HTML rendering engine. Using Javascript code, the vulnerability is triggered by changing two attributes of a table column: SPAN and WIDTH. The table rendering code of mshtml dynamically allocates a buffer that is based on the SPAN value. The buffer is used to store the style information of columns of a CTableLayout object, depicted in Figure 4.24. The size of an allocated buffer is computed by multiplying the SPAN attribute by 0x1C. However, a flaw in the mshtml rendering code allows writing beyond the boundary of a buffer area on the heap if the SPAN attribute is changed

at runtime. The WIDTH attribute is also used to manipulate data on the heap. An arbitrary value can be written over an arbitrary memory area by changing dynamically the supplied WIDTH attribute of the table column. Thus, the CVE-2012-1876 vulnerability allows attackers to perform an arbitrary remote code execution using a crafted web page.

Attackers can also use this heap overflow vulnerability to read an arbitrary memory address [136, 172]. This ability enables attackers to bypass ASLR protection as well as DEP protection. The vtable of a CButtonLayout object of mshtml is always assigned with a fixed offset, which may vary for each mshtml version. It is possible to discover the current base address of mshtml by reading this vtable, even though ASLR protection is enabled. Therefore, malware may manipulate the heap within a specific layout in order to read the vtable of a CButtonLayout. Two 0x100-byte BSTR strings and a CButtonLayout object are created. After this operation is repeated a number of times, several copies of the pattern are allocated on the heap and arranged adjacent to each other, as shown in Figure 4.25. Eventually, the first 0x100-byte BSTR strings in the pattern will be freed. This operation creates the specific layout with 0x100-byte free block holes, depicted in Figure 4.26.

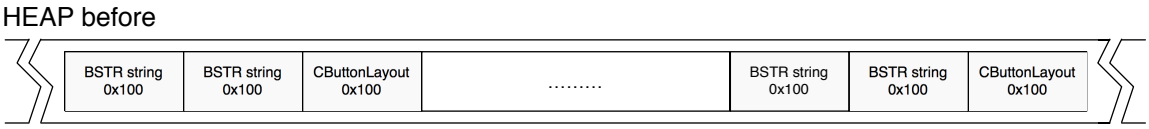


Figure 4.25: The HEAP before the first BSTR strings are freed

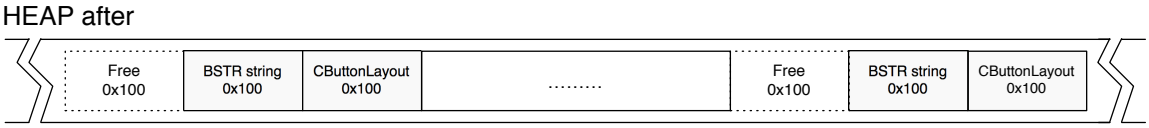


Figure 4.26: The HEAP after the first BSTR strings are freed

When the vulnerability is triggered the first time, one of the free block holes on the heap will be allocated as the "vulnerable" buffer and it will be followed by a BSTR string and CButtonLayout object. The BSTR string can be used to read the vtable of CButtonLayout object. By modifying the SPAN and WIDTH attributes, the memory area of the BSTR string can be overwritten. The header of a BSTR string, its first four (4) bytes, refers to the length of this string. Overwriting these four (4) bytes changes the length information of the BSTR string and allows malware

to read the further locations on the heap, depicted in Figure 4.27. Thus, the vtable of the following CButtonLayout object can be read in order to obtain the base address of mshtml.

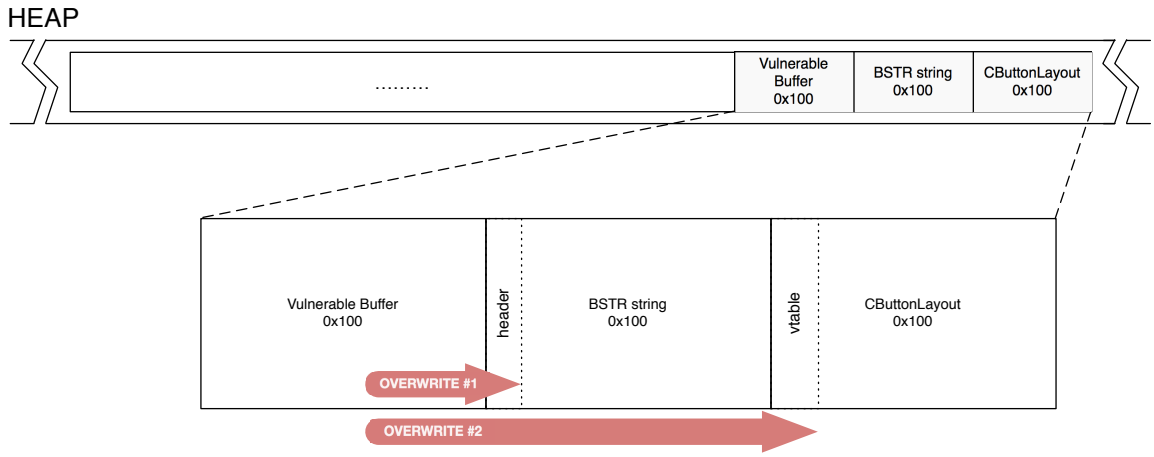


Figure 4.27: Overwriting the sensitive memory areas, such as the header of BSTR string and the vtable of CButtonLayout

After the base address is disclosed, malware can calculate the addresses of the ROP gadgets that are needed by the ROP payloads, and dynamically generate a shellcode, based on the leaked address in mshtml.dll. Then, malware sprays the heap with a significant number of shellcode copies. The CVE-2012-1876 vulnerability is triggered the second time in order to gain execution control by overwriting the vtable of the CButtonLayout object, depicted in Figure 4.26.

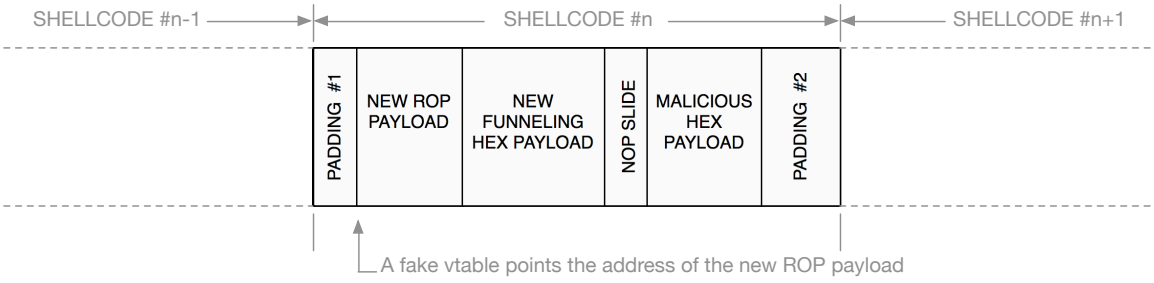


Figure 4.28: The crafted vtable points to the address of a new ROP payload

Similar to modifying the length information in a BSTR header, manipulating the supplied SPAN and WIDTH attributes allow the modification of the vtable with a specific value. The crafted vtable points to the address of a new ROP payload so

an attack execution starts in the beginning of a shellcode, which contains the new ROP payload. Only one of the shellcode copies on the heap is used for an attack, as depicted in Figure 4.28. The execution order of the shellcode parts during an attack is: the new ROP payload, the new funneling payload, the NOP slide, and a malicious payload. The malicious payload can be successfully executed without any detection by an antivirus mitigation, after the previous payload parts disable the detection features of antivirus products using the funneling method.

## 4.5 Results

Our previous work has demonstrated how attackers may generate a massive number of undetectable malware variants from a well-known malware file which is detectable by signature-based anti-malware products, while preserving its functionality [35]. In this research, we demonstrate that it is also strongly possible for attackers to create undetectable malware variants from a piece of publicly known malware and bypass behavioral-based protection, only by adding a malicious code part that directly attacks the anti-malware's code and protection mechanisms in memory to disable them. These new and improved antivirus-aware malware variants are also undetectable and functional too.

In this thesis, we concentrate on the effectiveness of behavioral-based protection techniques at user-level. This thesis addresses the limitations of anti-malware's user-land mitigation techniques by unhooking and funneling methods. Typically, most anti-malware products insert their protection mechanisms to memory locations in the virtual address spaces of processes in order to monitor system activities and identify serious malicious threats. Anti-malware and malware are placed and run in the same memory space at user level. The key problem with this approach is that anti-malware and malware have exactly the same memory access permissions at user-level. It allows malware to manipulate memory and disable protections. The unhooking and funneling methods take advantage of this weakness in the protection design, and they directly attack anti-malware components to deactivate them.

In our experiments, our web-based attacks using the unhooking and funneling methods have successfully bypassed two enterprise-level anti-malware products which implement behavioral-based protection techniques. Both of these products inject an intrusion protection module (DLL) into the virtual address space to protect processes in userland. They monitor critical system functions and data structures, they also

offer behavioral-based protection against common attacks, such as stack pivoting, heapspraying, anti-hooking, and ROP.

Due to performance concerns, it is impractical for anti-malware to run a set of threat control algorithms after every instruction. Therefore, "Inline hooks", "Debug Registers" and "Guard Pages" are used by anti-malware as a trap to accomplish its malware detection checks for only every invocation of a critical system function and access to an important data structure in memory during execution. The detection checks apply several behavioral-based anti-malware techniques to identify a malicious threat. These anti-malware traps in memory are user space solutions, and they transfer code execution to the control sections of anti-malware's code when they are triggered by an critical invocation or access. This technique facilitates an efficient way of performing detection checks.

Most enterprise-level anti-malware employ inline hooking method to monitor critical functions. Briefly, a hook is located by inserting a jump code in functions that will be monitored. It allows anti-malware to intercept critical function invocations and redirect function code to its own control code. Anti-malware products usually have hooking engines with their own implementations and most implementation of inline hooking method allows only one anti-malware installed on the same computer. Even though the use of inline hooking method complicates malware development, there are well-known techniques to bypass inline hooking. For example, malware simply invokes a deeper level critical function, or it jumps beyond the hook instruction to evade. Thus, advanced anti-malware techniques have been developed and implemented to detect the anti-hooking attacks, such as EMET's Deep Hooks and Anti-Detour features, described in Section 3.3.1.

One interesting implication of our results is that monitoring a limited number of critical functions is not complete, and it is still possible to call a critical function that is protected by inline hooking powered with the Deep Hooks and Anti-Detour features. In Windows, most Ntdll functions include system call instructions to switch the code execution from user space to system space. Therefore, they are commonly invoked by malware to perform sensitive system activities, such as bypassing DEP and ASLR protections. What is surprising is that many ntdll functions have very similar instruction sets. For example, the difference between ntdll.ZwProtectVirtualMemory and ntdll.ZwQuerySection is only their first instructions. Since ntdll.ZwQuerySection is not considered as critical, it is also not protected by anti-malware. Therefore, when a ntdll function is emulated, unprotected functions can be exploited, instead

of using exactly the same function of ntdll. Based on this finding, as discussed in Section 4.2.2.2.1, the new malicious function invocation method we introduced may be employed by malware to evade the improved inline hooking protection.

Another important finding regarding the weakness of inline hooking is that it is possible for malware to find important code sections by following inline hooks. Our research demonstrates how the use of inline hooks may leak the address of anti-malware code in memory, as discussed in Section 4.4. In the funneling method experiments, malware could discover the location of anti-malware's main control code.

We have previously discussed how "Debug Registers" and "Guard Pages" are used for monitoring critical data structures in the virtual address spaces of processes. In Windows, Critical data structures, such as the PEB, are exploited by malware to find interesting memory addresses. For example, traversing the data structures of the PEB allows malware to discover loaded modules in memory. Guard Pages are considered as limitless, whereas only four debug registers, namely DR01, DR02, DR03, and DR04, can be used for protection in total. An example of the use of "Guard Pages" is that EMET tracks every access to the memory regions of PE headers and EATs of sensitive modules by placing guard pages. Prior to using guard pages, EMET employed debug registers for protection of critical data structures. A bypassing technique already exists in the literature against the debug register protection. One unanticipated finding is that guard pages are not an effective protection against attacks that alter the access protections of memory regions. Although an exception is raised by the guard page mechanism when accessing a memory area within the guard page, surprisingly, changing the access protection of the memory area within the guard page does not trigger any alarm. In this thesis, we introduce a new technique to eliminate the guard page protections, described in Section 3.5.1.2.

Similar to the PE header, PEB, and EAT, the Import Address Table (IAT) may also be exploited by malware in order to find addresses of interest in memory. It is interesting to note that none of the enterprise-level anti-malware products we tested monitor the IAT, even though other critical data structures, including the PE header, and EAT, are protected by them. In the experiments, our malware variants were able to abuse this security hole by traversing the IAT to as a way of bypassing behavioral-based protections of the anti-malware products.

Another interesting finding is that even though the base address of a loaded module always changes on every run because of ASLR, the general structures of executables and loaded modules in memory are still deterministic. The offsets of module

functions and data structures as well as the distances between them are fixed, and these values are well-known during malware development. ASLR only randomizes the locations of loaded modules, stacks, and heaps. However, the modules of both the operating system and anti-malware programs are still loaded to memory with exactly the same deterministic structure. Therefore, advanced malware is able to calculate interesting addresses at runtime, if it obtains a base address using a memory information leak. In both the unhooking and funneling methods, we demonstrate how malware may employ the deterministic structures of executables and loaded modules in memory in order to bypass DEP and ASLR protections, as explained in Sections 3.5.1.3, 3.5.2.3, and 3.5.3.

The results of this research indicate that behavioral-based anti-malware techniques at user-level are incomplete, and they do not present a serious impediment to malware which takes advantage of the above-described weaknesses. In particular, we have shown that it is possible to modify known malware produced using the Metasploit framework so that it completely bypasses the behavioral detection techniques employed by two well-known enterprise-level anti-malware products.

# Chapter 5

# Conclusion and Future Work

In this thesis, we have evaluated the efficacy of behavioral-based dynamic anti-malware techniques, and demonstrated the limitations of defensive mechanisms deployed by anti-malware products in user space. In addition, we have shown how it is possible for malicious software to disable anti-malware protection.

The digitization of life and the latest technological innovations continue to reshape people's lifestyles in every aspects. Since the 1980s, the Internet been rapidly evolved into a cyber metropolis that globally connects half of the world's population to each other without any boundaries, which is an estimated 3.6 billion people. This massive computer network offers considerable benefits and new opportunities to individuals, corporates, and nations worldwide. In spite of its great conveniences, unfortunately, it also attracts the attentions of criminals by providing a new attacking vector with high financial gain. On the Internet, millions of people are at risk of cyber attacks that target a wide range of victims from individuals to government organizations, and cyber-lawbreakers employ well-established, evolving methods in the attacks to accomplish their harmful intentions. Most cyber attacks entail malicious software, called malware, to compromise computer systems; therefore, an effective protection against malware-initiated threats is essential in the area of system security.

Over the past decades, there has been a dramatic increase in attacks involving malware on the Internet. Cyber criminals spread malware code through the Internet by maintaining crafted web and ftp servers, sending malignant emails and phishing messages, as well as serving malware-injected content. Malware is a pervasive, evolving, and destructive threat to the continued viability of the Internet. In response to malware threats, a variety of anti-malware analysis, detection and mitigation tech-

niques have been developed. Anti-malware products are widely used by security experts to defend systems by locating on computer networks, servers, and endpoints, which offers a comprehensive protection against malware. Most anti-malware solutions identify malicious activities in systems by applying the latest detection and mitigation techniques, which are classified into two main categories: Signature-based and behavioral-based.

Signature-based detection determines if a suspicious file or network traffic is malicious using predefined signatures of well-known malicious code for identifying patterns. Even though signature-based detection techniques are fast and prone to false positives, they are only effective against previously known malware. Additionally, advanced morphic malware is able to mutate itself at each iteration to have no predictable patterns between generations, so that signature-based detection becomes impossible. In contrast, behavioral-based detection provides a proactive protection by employing behavioral patterns, instead of signatures. Thus, an anti-malware product armed with behavioral-based detection is not negatively affected by morphing, and as well it is capable of detecting not only previously discovered threats, but also new and unknown attacks. Even if malware is able to evade signature-based detection, advanced anti-malware products still have an opportunity to detect it using behavioral based detection techniques. For a complete malware protection system, most advanced anti-malware products combine both signature-based and behavioral-based detection techniques.

In our previous research, we introduced a practical way to automatically create many malware variants that evade signature-based detections [35]. The work of this thesis complements our earlier approach by concentrating on techniques for evading user-level behavioral-based mitigation techniques. In this context, the efficacy of two enterprise-level anti-malware products installed on vulnerable operating systems have been evaluated by conducting several web-based attacks using new malware variants deploying two evasion methods, unhooking and funneling. The new enhanced malware variants have been employed against leading real-world anti-malware products, Microsoft EMET and McAfee HIPS, in order to test their behavioral-based protections against the unhooking and funneling evasion methods. In our experiments, new unhooking- and funneling-enhanced malware variants successfully defeated these enterprise-level anti-malware products, confirming that it is possible to evade behavioral-based detection methods that are deployed in user space.

In order to design our evasion techniques, we performed an in-depth analysis of behavioral-based user-level mitigation techniques used by the targeted products. Both products use behavior-based protection against attack techniques commonly used by malware, such as stack pivoting, heapspraying, anti-hooking, and ROP-based code execution. They also monitor critical system functions and sensitive data structures in process memory to detect abuse by malware. For monitoring purposes, inline hooking and guard-page-based mechanisms are used as a trap to run a set of threat control and detection algorithms at every access to critical functions and data structures.

In this context, we introduced two new techniques to be able to call critical system functions hooked by anti-malware, and to remove guard pages protecting critical data structures in memory: a new malicious function call emulation technique and a new evasion technique disabling guard-page protections. Anti-malware products employ inline hooking protection methods improved with additional features, including deep hooking and anti-detour, in order to monitor critical functions as well as they create guard pages in memory to monitor critical data structures, such as PE headers, and EATs. Surprisingly, both anti-malware products do not monitor IATs. The method of guard page protection may also be used for tracking critical functions in memory. Due to only monitoring a limited number of critical functions and code similarity between various ntdll functions in memory, the malicious function call emulation technique employs an unhooked ntdll function that is similar in code to perform a system call, instead of invoking a critical function hooked by anti-malware. Likewise, the evasion technique disabling guard-page protections alters the access protection of a memory area within the guard page without triggering any alarm by calling the ntdll.ZwProtectVirtualMemory function to disable the guard pages on sensitive memory regions. The research also shows that a mitigation approach monitoring a limited number of system functions is incomplete, as well as it is possible that inline hooking and guard-page-based protections can be evaded by malware in userland.

In this research, we have designed and implemented several new enhanced variants of known malware using the unhooking and funneling evasion methods in order to evaluate the performance of behavioral-based mitigation techniques in the experiments. The new unhooking- and funneling-enhanced malware variants bypassing protections have been generated publicly known originals that are detectable by anti-malware products. Several exploit modules provided by the Metasploit Framework has been improved by adding a bypassing ability against behavioral mitigation. We have combined our antivirus-aware exploit modules with regular payload modules

provided by the Metasploit Framework to create a new malware variant. The new unhooking- and funneling-enhanced exploit modules are compatible with all of Metasploit payload modules. These experiments against two well-known enterprise-level anti-malware products demonstrated that it is possible to produce a new piece of enhanced malware completely bypassing the behavioral detection techniques by altering publicly known detectable malware originals.

In the context of malware development, this thesis demonstrates the utility for malware development of just-in-time debugging tools which automate the discovery of internal information about targeted anti-malware, such as hooked critical functions and offsets of critical locations. We developed such a debugging tool using the Intel PIN dynamic instrumentation framework. Typically, the development process of malware against an advanced anti-malware product is very labor intensive and demands a deep technical system knowledge. The information gathering tasks in development are very time-consuming when accomplished by manual reverse-engineering. Advanced mitigation techniques make the malware development process even more complicated. The just-in-time debugging framework developed in this thesis automates information gathering, and provides insight into behavioral mitigation techniques.

The results of this research indicate that behavioral-based anti-malware techniques are incomplete when they are deployed only in user space, and it is possible to develop enhanced malware completely bypassing the behavioral detection techniques implemented by two well-known enterprise-level anti-malware products.

## 5.1 Future Work

The scope of this thesis was limited in terms of behavioral-based dynamic anti-malware techniques in user space, and so this study did not evaluate the use of mitigation techniques in kernel space as well as did not include the domain of signature-based techniques. The kernel-level protections of operating systems and anti-malware products augment system and application security on computers and complicate the process of malware development and exploitation for malware authors. Future research should focus on behavioral-based mitigation in kernel space and analyzing its performance and possible limitations.

This results of this research demonstrate that malware is able to disable anti-malware protections when both malware and anti-malware run in the same mode

with equal privileges, such as at user or kernel mode. For cloud systems, which employ virtualization technologies, behavioral-based dynamic anti-malware techniques may be directly deployed in the hypervisors of virtual operating platforms in order to eliminate the possibility of a user or kernel mode evasion performed by advanced malware. In terms of directions for future research, further work could investigate and experiment with "hypervisor-embedded behavioral-based dynamic mitigation techniques in virtualization environments".

We have concentrated specifically on the efficacy of behavioral-based mitigation techniques and corresponding evasion methods. We have not covered the vulnerability discovery process and exploitation techniques in operating systems and applications. Therefore, when conducting experimental web-based attacks, we assume that there are one or more vulnerabilities on a victim computer and adversaries are able to exploit the vulnerabilities and execute malicious code on victim computers. Our research will be enhanced in the future by investigating the automatization of vulnerability discovery and software debugging.

Since the study was confined to only one tested operating system and vulnerable application type in the experiments, the results of this study may be somewhat limited to the Windows operating system on 64-bit architectures, and the 32-bit browser provided by the operating system. More research is required to determine the possible effects of the use of different operating systems and applications on the efficacy of behavioral-based dynamic mitigation techniques.

The experiments also focus on web-based malware, which attacks a client-side application, namely an Internet browser. Server-side and client-side applications have different behaviors with respect to exploitation, and different exploitation techniques bring various advantages and disadvantages. An example of this is the recovery of processes. Server-side service applications, such as http, automatically recover their child processes and restart them when they crash, which offers an excellent opportunity for brute-force attacks. In contrast, client-side applications, such as Internet browsers, are under the control of users, allowing a limited number of attacks because browser processes are terminated when malware is unsuccessful and as well, users may terminate browsers if they operate too slowly due to exploitation. Our research should be extended using server-side applications, to investigate the effectiveness of behavioral-based mitigation techniques for server-side applications.

# Bibliography

[1] 0xdabbad00 (Dabbadoo). Emet 4.1 Uncovered. http://0xdabbad00.com/wp-content/uploads/2013/11/emet_4_1_uncovered.pdf, November 2013.

[2] Martín Abadi, Mihai Budiu, Úlfar Erlingsson, and Jay Ligatti. Control-flow integrity. In *Proceedings of the 12th ACM Conference on Computer and Communications Security, CCS 2005, Alexandria, VA, USA, November 7-11, 2005*, pages 340–353, 2005.

[3] Martín Abadi, Mihai Budiu, Úlfar Erlingsson, and Jay Ligatti. Control-flow integrity principles, implementations, and applications. *ACM Trans. Inf. Syst. Secur.*, 13(1):4:1–4:40, 2009.

[4] J. Afek and A. Sharabani. Dangling pointer - smashing the pointer for fun and profit. *Black Hat USA*, 2007.

[5] Abdulellah Alsaheel and Raghav Pande. Using EMET to disable EMET. *Black Hat USA*, 2016.

[6] Starr Andersen and Vincent Abella. Part 3: Memory protection technologies. *Microsoft TechNet Magazine*, September 2004.

[7] Alexander Anisimov. Defeating Microsoft Windows XP SP2 Heap protection, 2005.

[8] Cryptic Apps. Hopper disassembler. https://www.hopperapp.com/, 2017.

[9] Elias Bachallany. Inside EMET 4.0, June 2013.

[10] Piotr Bania. Security mitigations for return-oriented programming attacks, 2010.

[11] Piotr Bania. BYPASSING EMET Export Address Table Access Filtering feature. http://piotrbania.com/all/articles/anti_emet_eaf.txt, March 2014.

[12] Bitdefender. New backdoor allows full access to mac systems, Bitdefender warns. https://labs.bitdefender.com/2016/07/new-mac-backdoor-nukes-os-x-systems, 2017.

[13] blexim. Basic integer overflows. *Phrack Magazine*, 11(60):10, 2002.

[14] Walter Bright. Vptr jamming. http://www.artima.com/cppsource/backyard2.html, May 2006.

[15] Erik Buchanan, Ryan Roemer, Hovav Shacham, and Stefan Savage. When good instructions go bad: generalizing return-oriented programming to RISC. In *Proceedings of the 2008 ACM Conference on Computer and Communications Security, CCS 2008, Alexandria, Virginia, USA, October 27-31, 2008*, pages 27–38, 2008.

[16] J. Butler and S. Sparks. Windows rootkits of 2005, part one. http://www.symantec.com/connect/articles/windows-rootkits-2005-part-one, November 2005.

[17] Nicholas Carlini and David Wagner. ROP is still dangerous: Breaking modern defenses. In *Proceedings of the 23rd USENIX Security Symposium, San Diego, CA, USA, August 20-22, 2014.*, pages 385–399, 2014.

[18] CERT.org. W32/blaster worm. http://www.cert.org/historical/advisories/CA-2003-20.cfm, August 2013.

[19] Stephen Checkoway, Ariel J. Feldman, Brian Kantor, J. Alex Halderman, Edward W. Felten, and Hovav Shacham. Can DREs provide long-lasting security? the case of return-oriented programming and the AVC advantage. In *2009 Electronic Voting Technology Workshop / Workshop on Trustworthy Elections, EVT/WOTE '09, Montreal, Canada, August 10-11, 2009*, 2009.

[20] Yueqiang Cheng, Zongwei Zhou, Miao Yu, Xuhua Ding, and Robert H. Deng. Ropecker: A generic and practical approach for defending against ROP attacks. In *21st Annual Network and Distributed System Security Symposium, NDSS 2014, San Diego, California, USA, February 23-26, 2014*, 2014.

[21] Tzi-Cker Chiueh and Fu-Hau Hsu. Rad: a compile-time solution to buffer overflow attacks. In *Proceedings 21st International Conference on Distributed Computing Systems*, pages 409–417, Apr 2001.

[22] M. Christodorescu, S. Jha, S. A. Seshia, D. Song, and R. E. Bryant. Semantics-aware malware detection. In *2005 IEEE Symposium on Security and Privacy (S P'05)*, pages 32–46, May 2005.

[23] Mihai Christodorescu and Somesh Jha. Testing malware detectors. In *Proceedings of the ACM/SIGSOFT International Symposium on Software Testing and Analysis, ISSTA 2004, Boston, Massachusetts, USA, July 11-14, 2004*, pages 34–44, 2004.

[24] Matt Conover and Oded Horovitz. Windows heap exploitation (Win2kSP0 through WinXPSP2), 2004.

[25] Corelan Team (corelanc0d3r). Exploit writing tutorial part 3 : SEH based exploits. https://www.corelan.be/index.php/2009/07/25/writing-buffer-overflow-exploits-a-quick-and-basic-tutorial-part-3-seh, July 2009.

[26] Corelan Team (corelanc0d3r). Exploit writing tutorial part 8 : Win32 egg hunting. https://www.corelan.be/index.php/2010/01/09/exploit-writing-tutorial-part-8-win32-egg-hunting/, January 2010.

[27] MITRE Corporation. CVE - CVE-2012-1875. http://www.cve.mitre.org/cgi-bin/cvename.cgi?name=CVE-2012-1875, March 2012.

[28] MITRE Corporation. CVE - CVE-2012-1876. http://www.cve.mitre.org/cgi-bin/cvename.cgi?name=CVE-2012-1876, March 2012.

[29] MITRE Corporation. CVE - CVE-2013-1347. http://www.cve.mitre.org/cgi-bin/cvename.cgi?name=CVE-2013-1347, January 2013.

[30] Crispan Cowan. Stackguard: Automatic adaptive detection and prevention of buffer-overflow attacks. In *Proceedings of the 7th USENIX Security Symposium, San Antonio, TX, USA, January 26-29, 1998*, 1998.

[31] Lucas Davi, Ahmad-Reza Sadeghi, Daniel Lehmann, and Fabian Monrose. Stitching the gadgets: On the ineffectiveness of coarse-grained control-flow in-

tegrity protection. In *Proceedings of the 23rd USENIX Security Symposium, San Diego, CA, USA, August 20-22, 2014.*, pages 401–416, 2014.

[32] Lucas Davi, Ahmad-Reza Sadeghi, and Marcel Winandy. Ropdefender: A detection tool to defend against return-oriented programming attacks. In *Proceedings of the 6th ACM Symposium on Information, Computer and Communications Security*, ASIACCS '11, pages 40–51, New York, NY, USA, 2011. ACM.

[33] Jared DeMott. Bypassing emet 4.1. http://bromiumlabs.files.wordpress.com/2014/02/bypassing-emet-4-1.pdf, February 2014.

[34] Solar Designer. Getting around non-executable stack (and fix). Bugtrack, 1997.

[35] Erkan Ersan, Lior Malka, and Bruce M. Kapron. Semantically non-preserving transformations for antivirus evaluation. In *Foundations and Practice of Security - 9th International Symposium, FPS 2016, Québec City, QC, Canada, October 24-25, 2016, Revised Selected Papers*, pages 273–281, 2016.

[36] Eweek.com. Zero-day exploit enabled cyber-attack on U.S. labor department. http://www.eweek.com/security/zero-day-exploit-enabled-cyber-attack-on-us-labor-department, May 2013.

[37] FBI. Cyber Crime. https://www.fbi.gov/investigate/cyber, 2017.

[38] FBI. Identity theft. https://www.fbi.gov/investigate/white-collar-crime/identity-theft, 2017.

[39] Inc. FireEye. Apatedns. https://www.fireeye.com/services/freeware/apatedns.html, 2017.

[40] Halvar Flake. Third generation exploitation smashing heap on Win2K. *Black Hat USA*, 2002.

[41] Cuckoo Foundation. Cuckoo sandbox. https://cuckoosandbox.org, 2017.

[42] Aurélien Francillon and Claude Castelluccia. Code injection attacks on harvard-architecture devices. In *Proceedings of the 2008 ACM Conference on Computer and Communications Security, CCS 2008, Alexandria, Virginia, USA, October 27-31, 2008*, pages 15–26, 2008.

[43] I. Fratric. ROPGuard: Runtime prevention of return-oriented programming attacks. http://www.ieee.hr/_download/repository/Ivan_Fratric.pdf, 2012.

[44] Gartner. Gartner says 6.4 billion connected "things" will be in use in 2016, up 30 percent from 2015. http://www.gartner.com/newsroom/id/3165317, November 2015.

[45] Gartner. Gartner says worldwide information security spending will grow 7.9 percent to reach $81.6 billion in 2016. http://www.gartner.com/newsroom/id/3404817, August 2016.

[46] Government of Canada. Cyber security risks. https://www.getcybersafe.gc.ca/cnt/rsks/index-en.aspx/index-en.aspx, 2017.

[47] Helpnetsecurity.com. Rise of cyber attacks against the public sector. https://www.helpnetsecurity.com/2016/09/23/cyberattacks-public-sector, September 2016.

[48] Intel. Pin - A dynamic binary instrumentation tool. https://software.intel.com/en-us/articles/pin-a-dynamic-binary-instrumentation-tool.

[49] Intel. *Intel 64 and IA-32 architectures software developers manual, combined volumes: 1, 2A, 2B, 2C, 2D, 3A, 3B, 3C and 3D*, 325462-061US edition, December 2016.

[50] Internet Systems Consortium, Inc. ISC Internet Domain Survey. https://www.isc.org/network/survey, 2016. [Online; accessed 23-September-2016].

[51] Internetworldstats.com. World Internet users statistics and 2016 world population stats. http://www.internetworldstats.com/stats.htm, 2017. [Online; accessed 30-January-2017].

[52] Interpol. Cybercrime / Crime areas / Internet. https://www.interpol.int/Crime-areas/Cybercrime/Cybercrime, 2017.

[53] S. Jalayeri. Bypassing EMET 3.5's ROP mitigations. [Online]. Available: https://repret.wordpress.com/2012/08/08/bypassing-emet-3-5s-rop-mitigations, August 2012.

[54] Simon Josefsson. The base16, base32, and base64 data encodings. Internet RFC 4648, October 2006.

[55] Jotti. Jotti's malware scan. https://virusscan.jotti.org, 2017.

[56] Michel "MaXX" Kaempf. Vudo - An object superstitiously believed to embody magical powers. *Phrack Magazine*, 11(57):8, 2001.

[57] Randy Kath. Managing heap memory. *Microsoft Developer Network*, April 1993.

[58] Darren Kemp and Mikhail Davidov. WoW64 and So Can You - Bypassing EMET with a Single Instruction. https://duo.com/blog/wow64-and-so-can-you, November 2015.

[59] Engin Kirda and Christopher Kruegel. Behavior-based spyware detection. In *Proceedings of the 15th USENIX Security Symposium, Vancouver, BC, Canada, July 31 - August 4, 2006*, 2006.

[60] J. Koret and E. Bachaalany. *The Antivirus Hacker's Handbook*, chapter 9 Evading Heuristic Engines, pages 165–182. John Wiley & Sons, 1st edition, 2015.

[61] T. Kornau. Return oriented programming for the ARM architecture. Master's thesis, Ruhr-University Bochum, Germany, 2009.

[62] Christopher Krügel, Engin Kirda, Darren Mutz, William K. Robertson, and Giovanni Vigna. Polymorphic worm detection using structural information of executables. In *Recent Advances in Intrusion Detection, 8th International Symposium, RAID 2005, Seattle, WA, USA, September 7-9, 2005, Revised Papers*, pages 207–226, 2005.

[63] Christopher Krügel, Engin Kirda, Darren Mutz, William K. Robertson, and Giovanni Vigna. Polymorphic worm detection using structural information of executables. In *Recent Advances in Intrusion Detection, 8th International Symposium, RAID 2005, Seattle, WA, USA, September 7-9, 2005, Revised Papers*, pages 207–226, 2005.

[64] Christopher Krügel, William K. Robertson, Fredrik Valeur, and Giovanni Vigna. Static disassembly of obfuscated binaries. In *Proceedings of the 13th*

*USENIX Security Symposium, August 9-13, 2004, San Diego, CA, USA*, pages 255–270, 2004.

[65] D. Kushner. The real story of stuxnet. *IEEE Spectrum*, 50(3):48–53, March 2013.

[66] Kaspersky Lab. Types of Malware. https://usa.kaspersky.com/internet-security-center/threats/types-of-malware, 2017.

[67] Code Audit Labs. [CAL-2012-0026] Microsfot IE same id property remote code execution vulnerability. http://seclists.org/fulldisclosure/2012/Jun/252, June 2012.

[68] Malware Tracker Limited. Malware tracker. http://www.malwaretracker.com, 2017.

[69] Rotarua Limited. Virus Total. https://www.virustotal.com/, 2017.

[70] Felix 'FX' Lindner. Developments in Cisco IOS forensics. *Black Hat USA*, 2008.

[71] David Litchfield. Windows 2000 format string vulnerabilities. David Litchfield, May 2002.

[72] David Litchfield. Defeating the stack based buffer overflow prevention mechanism of Microsoft Windows 2003 server, September 2003.

[73] Joe Security LLC. Joe sandbox. https://www.joesecurity.org/, 2017.

[74] Microsoft Malware Protection Center. Ransomware. https://www.microsoft.com/security/portal/mmpc/shared/ransomware.aspx, 2017.

[75] Victor Marak. *Windows Malware Analysis Essentials*. Packt Publishing, 2015.

[76] McAfee. *McAfee Host Intrusion Prevention 8.0 Installation Guide*, 2010.

[77] McAfee. *McAfee Host Intrusion Prevention 8.0 Product Guide for use with ePolicy Orchestrator 4.5*, 2010.

[78] Gary McGraw and J. Gregory Morrisett. Attacking malicious code: A report to the infosec research council. *IEEE Software*, 17(5):33–41, 2000.

[79] Microsoft. Microsoft BlueHat Prize Contest. https://www.microsoft.com/security/bluehatprize, 2012.

[80] Microsoft. Enhanced Mitigation Experience Toolkit. https://www.microsoft.com/emet, 2017.

[81] Microsoft. *Enhanced Mitigation Experience Toolkit (EMET) 5.52 User Guide*, November 2017.

[82] Matt Miller and Ken Johnson. Exploit mitigation improvements in Win 8. *Black Hat USA*, 2012.

[83] Steve Miller. Dependency Walker. http://www.dependencywalker.com, 2017.

[84] Miniwatts Marketing Group. World Internet users statistics and 2016 world population stats. http://www.internetworldstats.com/stats.htm, 2016. [Online; accessed 23-September-2016].

[85] D. Moore, V. Paxson, S. Savage, C. Shannon, S. Staniford, and N. Weaver. Inside the slammer worm. *IEEE Security Privacy*, 1(4):33–39, July 2003.

[86] David Moore, Colleen Shannon, and Kimberly C. Claffy. Code-red: a case study on the spread and victims of an internet worm. In *Proceedings of the 2nd ACM SIGCOMM Internet Measurement Workshop, IMW 2002, Marseille, France, November 6-8, 2002*, pages 273–284, 2002.

[87] Andreas Moser, Christopher Kruegel, and Engin Kirda. Limits of static analysis for malware detection. In *23rd Annual Computer Security Applications Conference (ACSAC 2007), December 10-14, 2007, Miami Beach, Florida, USA*, pages 421–430, 2007.

[88] Microsoft Developer Network. Internet Explorer architecture. https://msdn.microsoft.com/en-us/library/aa741312(v=vs.85).aspx. [Online; accessed 02-February-2017].

[89] Microsoft Developer Network. Kernel patch protection: frequently asked questions. https://msdn.microsoft.com/en-us/library/windows/hardware/Dn613955%28v=vs.85%29.aspx, January 2007.

[90] Microsoft Developer Network. Creating guard pages. https://msdn.microsoft.com/en-us/library/windows/desktop/aa366549(v=vs.85).aspx, 2017. [Online; accessed 06-February-2017].

[91] Microsoft Developer Network. Data Execution Prevention. https://msdn.microsoft.com/library/windows/desktop/aa366553(v=vs.85).aspx, 2017.

[92] Microsoft Developer Network. /GS (buffer security check). https://msdn.microsoft.com/en-us/library/8dbf701c.aspx, 2017.

[93] Microsoft Developer Network. Low-fragmentation heap. https://msdn.microsoft.com/en-us/library/windows/desktop/aa366750(v=vs.85).aspx, 2017.

[94] Microsoft Developer Network. /nxcompat (compatible with data execution prevention). https://msdn.microsoft.com/library/ms235442.aspx, 2017.

[95] Microsoft Developer Network. /safeseh (image has safe exception handlers). https://msdn.microsoft.com/en-us/library/9a89h429.aspx, 2017.

[96] Microsoft Developer Network. Structured exception handling, 2017.

[97] BBC News. Major cyber spy network uncovered. http://news.bbc.co.uk/2/hi/americas/7970471.stm, March 2009.

[98] Ntinternals.net. NTAPI Undocumented Functions - NtProtectVirtualMemory. http://undocumented.ntinternals.net/index.html?page=UserMode/Undocumented Functions/Memory Management/Virtual Memory/NtProtectVirtual Memory.html, December 2000.

[99] Markus F.X.J. Oberhumer, László Molnár, and John F. Reiser. Upx : the Ultimate Packer for eXecutables. https://upx.github.io, 2017.

[100] Aleph One. Smashing the stack for fun and profit. *Phrack Magazine*, 7(49):14, 1996.

[101] Vasilis Pappas, Michalis Polychronakis, and Angelos D. Keromytis. Transparent ROP exploit mitigation using indirect branch tracing. In *Proceedings of the 22th USENIX Security Symposium, Washington, DC, USA, August 14-16, 2013*, pages 447–462, 2013.

[102] Phishme.com. Dridex – Password bypass, extracting macros, and Rot13. https://phishme.com/dridex-password-bypass-extracting-macros-rot13, February 2015.

[103] Josef Pieprzyk, Thomas Hardjono, and Jennifer Seberry. *Fundamentals of computer security*, pages 615–617. Springer, 2003.

[104] Matt Pietrek. A crash course on the depths of Win32 Structured Exception Handling. *Microsoft Developer Network*, January 1997.

[105] J. Pincus and B. Baker. Beyond stack smashing: recent advances in exploiting buffer overruns. *IEEE Security Privacy*, 2(4):20–27, July 2004.

[106] Aaron Portnoy. Bypassing all of the things, 2013.

[107] Wayne J. Radburn. Peview. http://wjradburn.com/software/, 2017.

[108] Rapid7. Ms12-037 Microsoft Internet Explorer fixed table col span heap overflow. https://www.rapid7.com/db/modules/exploit/windows/browser/ms12_037_ie_colspan. [Online; accessed 02-February-2017].

[109] Rapid7. Ms12-037 Microsoft Internet Explorer same id property deleted object handling memory corruption. https://www.rapid7.com/db/modules/exploit/windows/browser/ms12_037_same_id. [Online; accessed 02-February-2017].

[110] Rapid7. Ms13-038 Microsoft Internet Explorer CGenericElement object use-after-free vulnerability. https://www.rapid7.com/db/modules/exploit/ windows/browser/ie_cgenericelement_uaf. [Online; accessed 02-February-2017].

[111] Rapid7. Metasploit project. https://www.metasploit.com, 2017.

[112] Dennis M. Ritchie and Brian W. Kernighan. *The C Programming Language*, chapter 7, pages 153–155. Prentice Hall Professional Technical Reference, 2nd edition, 1988.

[113] rix. Smashing C++ VPTRS. *Phrack Magazine*, 10(56):8, 2000.

[114] Mark Russinovich. Windows administration inside the Windows Vista kernel: Part 3, Address Space Load Randomization. *Microsoft TechNet Magazine*, April 2007.

[115] Hex-Rays SA. IDA disassembler. https://www.hex-rays.com/products/ida/, 2017.

[116] Markus Schmall. Heuristic techniques in AV solutions: An overview. http://www.securityfocus.com/infocus/1542, February 2002.

[117] Felix Schuster, Thomas Tendyck, Jannik Pewny, Andreas Maaß, Martin Steegmanns, Moritz Contag, and Thorsten Holz. Evaluating the effectiveness of current anti-rop defenses. In *Research in Attacks, Intrusions and Defenses - 17th International Symposium, RAID 2014, Gothenburg, Sweden, September 17-19, 2014. Proceedings*, pages 88–108, 2014.

[118] Scut and Team Teso. Exploiting Format String Vulnerabilities, version 1.2, September 2001.

[119] Intel Security. Emerging 'stack pivoting' exploits bypass common security — McAfee blogs. https://securingtomorrow.mcafee.com/mcafee-labs/emerging-stack-pivoting-exploits-bypass-common-security/, May 2013.

[120] Intel Security. FAQs for Host Intrusion Prevention 8.0. https://kc.mcafee.com/corporate/index?page=content&id=KB73399, December 2016.

[121] Intel Security. Case Study - Chicago protects critical infrastructure and services with security connected. http://www.mcafee.com/ca/resources/case-studies/cs-city-of-chicago.pdf, 2017.

[122] Intel Security. Case Study - Global retailer secures virtual business infrastructure with McAfee MOVE AntiVirus. http://www.mcafee.com/ca/resources/case-studies/cs-large-retail-chain.pdf, 2017.

[123] Intel Security. Case Study - Government statistics agency embraces enterprise security connected strategy. http://www.mcafee.com/ca/resources/case-studies/cs-government-statistics-agency.pdf, 2017.

[124] Intel Security. Case Study - Integrated security architecture transforms commercial bank's security posture. http://www.mcafee.com/ca/resources/case-studies/cs-regional-commercial-bank.pdf, 2017.

[125] Intel Security. Case Study - Leading hosted IT provider drives growing MSP business with security connected. http://www.mcafee.com/ca/resources/case-studies/cs-macquarie-telecom.pdf, 2017.

[126] Intel Security. Case Study - Partners for the Long Haul: Kleberg Bank relies on Intel Security for comprehensive protection. http://www.mcafee.com/ca/resources/case-studies/cs-kleberg-bank.pdf, 2017.

[127] Intel Security. Case Study - SF Police Credit Union passes audits easily with comprehensive security management. http://www.mcafee.com /ca/resources/case-studies/cs-sf-police-credit-union.pdf, 2017.

[128] Intel Security. Case Study - State government places its trust in Intel Security for IT consolidation. http://www.mcafee.com/ca/resources/case-studies/cs-state-government-trust-intel-security.pdf, 2017.

[129] Intel Security. Case Study - TTUHSC reliance on McAfee to connect security and compliance. http://www.mcafee.com/ca/resources/case-studies/cs-texas.pdf, 2017.

[130] Intel Security. Data Sheet - McAfee Host Intrusion Prevention for Desktop. http://www.mcafee.com/ca/resources/data-sheets/ds-host-intrusion-for-desktop.pdf, 2017.

[131] Intel Security. McAfee Host Intrusion Prevention. http://www.mcafee.com /ca/products/host-ips-for-desktop.aspx, 2017.

[132] Offensive Security. Disarming and Bypassing EMET 5.1. https://www.offensive-security.com/vulndev/disarming-and-bypassing-emet-5-1, November 2014.

[133] Offensive Security. Disarming EMET v5.0. https://www.offensive-security .com/vulndev/disarming-emet-v5-0, September 2014.

[134] Offensive Security. Disarming Enhanced Mitigation Experience Toolkit (EMET). https://www.offensive-security.com/vulndev/disarming-enhanced-mitigation-experience-toolkit-emet, July 2014.

[135] Offensive Security. Metasploit Unleashed - Free Online Ethical Hacking Course. https://www.offensive-security.com/metasploit-unleashed, 2017.

[136] VUPEN Security. Advanced exploitation of Internet Explorer heap overflow (pwn2own 2012 exploit). https://web.archive.org/web/20150521040626/http://www.vupen.com/blog/20120710.Advanced_Exploitation_of_Internet_Explorer_HeapOv_CVE-2012-1876.php, July 2012.

[137] Karthik Selvaraj and Nino Fred Gutierrez. The rise of pdf malware. Technical report, Symantech Security Response, September 2010.

[138] Hovav Shacham. The geometry of innocent flesh on the bone: return-into-libc without function calls (on the x86). In *Proceedings of the 2007 ACM Conference on Computer and Communications Security, CCS 2007, Alexandria, Virginia, USA, October 28-31, 2007*, pages 552–561, 2007.

[139] sickness. Microsoft Internet Explorer 8 - Fixed col span id (full ASLR + DEP bypass) (MS12-037). https://www.exploit-db.com/exploits/24017/, January 2013.

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

[141] Aspack Software. ASPack - File compressor, exe compressor. http://aspack.com/aspack.html, 2017.

[142] Heaventools Software. Pe explorer. http://www.heaventools.com/pe-explorer-feature-list.htm, 2017.

[143] Alexander Sotirov. Heap feng shui in javascript. *Black Hat USA*, 2007.

[144] Alexander Sotirov and Mark Dowd. How to impress girls with browser memory protection bypasses. *Black Hat USA*, 2008.

[145] Bjarne Stroustrup. *The C++ Programming Language*. Addison-Wesley Longman Publishing Co., Inc., Boston, MA, USA, 3rd edition, 2000.

[146] Bjarne Stroustrup. The C++ Programming Language. http://www.stroustrup.com/C++.html, December 2016.

[147] Dafydd Stuttard. Writing small shellcode, 2005.

[148] Microsoft Support. How to enable Structured Exception Handling Overwrite Protection (SEHOP) in Windows operating systems. https://support.microsoft.com/en-us/kb/956607, June 2011.

[149] Microsoft Support. A detailed description of the Data Execution Prevention (DEP) feature in Windows XP Service Pack 2, Windows XP Tablet PC Edition 2005, and Windows Server 2003. https://support.microsoft.com/en-us/kb/875352, 2017.

[150] swiat. Preventing the Exploitation of Structured Exception Handler (SEH) Overwrites with SEHOP. *Microsoft TechNet, Security & Research Defense*, February 2009.

[151] swiat. Preventing the exploitation of user mode heap corruption vulnerabilities. *Microsoft TechNet, Security & Research Defense*, August 2009.

[152] swiat. EMET 3.5 Tech Preview leverages security mitigations from the BlueHat Prize. *Microsoft TechNet, Security & Research Defense*, July 2012.

[153] swiat. Mitigating the LdrHotPatchRoutine DEP/ASLR bypass with MS13-063. *Microsoft TechNet, Security & Research Defense*, August 2013.

[154] swiat. Software defense: mitigating common exploitation techniques. *Microsoft TechNet, Security & Research Defense*, December 2013.

[155] swiat. Software defense: mitigating heap corruption vulnerabilities. *Microsoft TechNet, Security & Research Defense*, October 2013.

[156] Symantec. Web threats. In *Internet Security Threat Report*, volume 21, April 2016.

[157] Symantec. Symantec Endpoint Protection. https://www.symantec.com/products/threat-protection/endpoint-family/endpoint-protection, 2017.

[158] Peter Szor. *The Art of Computer Virus Research and Defense*, pages 226–260. Addison-Wesley Professional, 2005.

[159] Peter Szor. *The Art of Computer Virus Research and Defense*, pages 352–358. Addison-Wesley Professional, 2005.

[160] Inc. Team Cymru. The Malware Hash Registry. http://www.team-cymru.org/MHR.html, 2017.

[161] Microsoft TechNet. Microsoft security bulletin MS12-037 - critical. https://technet.microsoft.com/library/security/ms12-037.aspx, June 2012.

[162] Microsoft TechNet. Microsoft security bulletin MS13-038 - critical. https://technet.microsoft.com/library/security/ms13-038, May 2013.

[163] Microsoft TechNet. Microsoft security bulletin MS14-021 - Critical. https://technet.microsoft.com/library/security/ms14-021, May 2014.

[164] Microsoft TechNet. Microsoft security bulletin MS14-024 - Important. https://technet.microsoft.com/library/security/ms14-024, May 2014.

[165] Oreans Technology. Themida. http://www.oreans.com/themida.php, 2017.

[166] The Globe and Mail. U.S. report rips China, Russia on economic espionage, hacking. http://www.theglobeandmail.com/technology/tech-news/us-report-rips-china-russia-on-economic-espionage-hacking/article4182647/, November 2011.

[167] Minh Tran, Mark Etheridge, Tyler K. Bletsch, Xuxian Jiang, Vincent W. Freeh, and Peng Ning. On the expressiveness of return-into-libc attacks. In *Recent Advances in Intrusion Detection - 14th International Symposium, RAID 2011, Menlo Park, CA, USA, September 20-21, 2011. Proceedings*, pages 121–141, 2011.

[168] Joshua Tully. An anti-reverse engineering guide. https://www.codeproject.com/articles/30815/an-anti-reverse-engineering-guide, November 2008.

[169] US-CERT. South Korean Malware Attack. https://www.us-cert.gov/security-publications/South-Korean-Malware-Attack, April 2013.

[170] Vendicator. Stack shield a "stack smashing" technique protection tool for linux. http://www.angelfire.com/sk/stackshield/info.html.

[171] Virusshare.com. VirusShare. https://virusshare.com/, 2017.

[172] Peter Vreugdenhil. Pwn2Own 2010 Windows 7 Internet Explorer 8 exploit. http://vreugdenhilresearch.nl/Pwn2Own-2010-Windows7-InternetExplorer8.pdf, 2010.

[173] A. Walenstein, R. Mathur, M. R. Chouchane, and A. Lakhotia. Normalizing metamorphic malware using term rewriting. In *2006 Sixth IEEE International Workshop on Source Code Analysis and Manipulation*, pages 75–84, Sept 2006.

[174] Wchen-r7. js_mstime_malloc_test.rb - github. https://gist.github.com/wchen-r7/ac29eb40fb33ddb5ab29, 2013. [Online; accessed 02-February-2017].

[175] We Are Social UK. Digital, Social and Mobile Worldwide in 2015. http://wearesocial.com/uk/special-reports/digital-social-mobile-worldwide-2015, 2015. [Online; accessed 31-January-2017].

[176] Mingwei Zhang and R. Sekar. Control flow integrity for COTS binaries. In *Proceedings of the 22th USENIX Security Symposium, Washington, DC, USA, August 14-16, 2013*, pages 337–352, 2013.