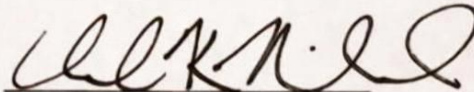
APPROVAL SHEET

Title of Thesis: Automate the tracing of Windows System Calls to identify malicious activities

Name of Candidate: SIDDHANT GOENKA
Master of Science in Computer Science, 2019

Thesis and Abstract Approved: 

DR. CHARLES NICHOLAS
Professor
Department of Computer Science and Electrical Engineering

Date Approved: April 29, 2019

ABSTRACT

Title of Thesis: Automate the tracing of Windows System Calls to identify malicious activities

SIDDHANT GOENKA, Master of Science in Computer Science, 2019

Thesis directed by: DR. CHARLES NICHOLAS,
Professor
Department of Computer Science and
Electrical Engineering

We describe the problems addressed by various malware or malicious applications on the Microsoft Windows Operating System. Our work focuses on automatic the dynamic malware analysis by intercepting Windows system calls that help to cover a larger range of malware, including the newly evolved fileless variants. Intercepting system calls allow us to monitor malicious activities in a way that malicious behavior can be easily identified without the manual efforts of disassembling binaries. The results will show how our work can help in automating the process of API Hooking for the open source community to detect Byzantine behaviors, rather than focusing on improving the detection mechanism.

Automate the tracing of Windows System Calls to identify malicious activities

by

SIDDHANT GOENKA

Thesis submitted to the Faculty of the Graduate School
of the University of Maryland in partial fulfillment
of the requirements for the degree of
Master of Science in Computer Science
2019

Advisory Committee:

Dr. Charles Nicholas

Dr. Anupam Singh

Dr. Richard Forno

© Copyright SIDDHANT GOENKA, 2019

ACKNOWLEDGMENTS

We are very grateful to *T. Rowe Price Group, Inc*, who supported this research and motivated to work in CyberSecurity domain.

We thank our professors and colleagues from *University of Maryland, Baltimore County* who provided insight and expertise that greatly assisted and improved the research.

TABLE OF CONTENTS

ACKNOWLEDGMENTS . . . . **ii**

Chapter 1 INTRODUCTION **1**

1.1 About Malware 1

1.2 Should we be concerned? 1

1.3 Fileless malwares 2

1.4 Analyzing Malware 3

1.5 Our work in a nutshell 4

Chapter 2 RELATED WORK **6**

2.1 API call sequence analysis 6

2.2 Memory Analysis 7

Chapter 3 POSSIBLE METHODS **10**

3.1 Override Windows PE loader 10

3.2 Using System APIs 11

3.3 Modify a process memory 12

3.3.1 Accessing memory via another process 13

3.3.2 Using Sandboxes 14

3.4 Change during compilation . . . 14

Chapter 4 IMPLEMENTATION AND EVALUATION . . . 16

4.1 Implementation . . . 16

4.1.1 Microsoft Detours . . . 16

4.1.2 Adding Hooks . . . 17

4.1.3 Activating Hooks . . . 18

4.2 Proof Of Concept . . . 19

4.2.1 Processes and Binaries . . . 19

4.2.2 Shellcodes . . . 21

4.2.3 PowerShell scripts . . . 22

4.2.4 Office/PDF files . . . 22

Chapter 5 CONCLUSION AND FUTURE WORK . . . 25

5.1 Conclusion . . . 25

5.2 Limitations . . . 25

5.3 Future Work . . . 26

Appendix A APIS HOOKED . . . 27

REFERENCES . . . 31

Chapter 1

INTRODUCTION

Gone are those days when *Security* was a second thought for the Computer Engineers. *Malware*[1] infections are coming like a torrent to us and we have to be totally aware of the precautionary measures in order to deal with them.

1.1 About Malware

Malware[1], or “malicious software”, is used as an umbrella term that describes any malicious program or code that is harmful to computers, systems or software controlled devices, including smart-phones, smartwatch or IOT devices. Malware invades a computer system or a computer network like the human flu and can take partial or total control of the system/network. It can steal, hide, alter or delete your data without your permission or knowledge.

As per *MalwareBytes*[2], “Malware attacks would not work without the most important ingredient: *you*”

1.2 Should we be concerned?

Unlike the first known malware a.k.a *Creeper*[3], a self-replicating program, today’s malware are extremely dangerous. They focus on stealing our data and making money.

According to SonicWall[4], $27,680$ new attack variants have been discovered in the year 2018, with an increase of 54% in malware attacks from 2017, where Ransomware attacks increased by 108%. Ransomware is a variant of malware that threatens to publish the victim's data unless a ransom/money is paid to them. The *MS-IAC*[6] observed a decrease in malware infections from December 2017 to January 2018 [5], but also states that Kovter continued to dominate the SLTT government landscape with 55% of total malware infections. KOVTER is an involving malware Trojan, gone through various changes during its lifespan, and eventually evolved into much more effective and evasive fileless malware.[7] Given it's gone almost fileless, it has become much more difficult to detect and mitigate them.

1.3 Fileless malwares

Traditionally, malware attacks we have known, have been using disks in one or other form to carry out the execution. Fileless Malware is specific kinds of malware that reside only in system memory, ideally leaving no traces on the disk/HDD after its execution. By leaving as little traces as possible, this malware make the post-forensics more difficult postpone the detection. Although staying only in memory makes the malware prone to be deleted on system restart, malware authors use the rootkit techniques to hide in the Windows registry and stealthy fileless persistence on a compromised system.

As per Symantec's Internet Security Threat Report [11], "Cybercriminals are adopting these tactics to spread threats like *ransomware* and *financial Trojans* but nation-state targeted attack groups also make use of them. There has been a growing interest in fileless infection techniques over the past few years. Fileless malware is not a new concept. For example, the *Code Red worm*, which first appeared in 2001, resided solely in memory and did not write any files to disk. In 2014 there was yet another spike of fileless attacks, this

time with fileless persistence methods used by threats such as *Trojan.Poweliks*[12] which resides completely in the registry". Later *Trojan.Kotver* and *Trojan.Bedep* utilized the same method extensively.

Unfortunately, it is not difficult to conduct fileless attacks. Scripting languages like *Powershell*[39], VBScript, JavaScript, Shell script and Python, are most exploited by malware authors. PowerShell is a command-line shell and a scripting language, that help Windows Administrators to automate tasks on a Windows System. PowerShell is developed by Microsoft and is also available for Linux and MacOs. However, it is widely used on Windows systems and is installed by default. Frameworks like *Metasploit*[13] provides many injection options, such as DLL injection and generating different payloads/*shellcodes*[16], making these attacks more prominent. Infiltration of system memory or incursions can be achieved by exploiting Remote Code Execution (RCE) vulnerabilities, and run the malicious payloads or scripts directly into the memory and misusing the attacked system.

1.4 Analyzing Malware

Malware analysis can be performed using various approaches. Detection techniques can be majorly categorized as "Static Analysis" and "Dynamic Analysis". Static analysis involves examining a binary code of malware, identify all the possible execution paths and identify/calibrate the intended behavior, without executing the malware. Signature-based techniques rely on pre-built databases and regular updates. This method fails for scripting languages, as it can be written in infinite ways to achieve similar functionality, as shown in Fig. [**??**]. Moreover, the anti-virus system scan only binaries to identify malicious behavior and not codes/scripts/macros. As *Obfuscation techniques*[15] are becoming stronger and easier to implement, the malware uses them extensively to hide their behavior as an additional layer of defense, until they are eventually executed. Obfuscation can be

achieved by using some cryptographic standards or by some advanced bit manipulations using XOR operation. These additional defense makes static analysis further difficult and time-consuming and involves a lot of manual efforts.

On the other hand, Dynamic Analysis is widely used to achieve more effective detection, involving the execution of the malware in a sand-boxed environment and trace its behavior. By doing so, we allow the malware to unleash its covert behavior. Efforts are then inclined to analyze the control flow of the malware for faster detection. It is also important to note that control flows can be easily misled by malware authors by inserting dummy or redundant system calls. Such tricks make static analysis slower and gives dynamic analysis an upper edge. Intercepting and logging the system calls, while the execution of malware, will give a better view of the malware intentions. Many of the currently available dynamic analysis techniques involve intercepting system calls in order to understand the exact malicious behavior, but fail to detect the redundant use of system calls. Also, no such techniques have been used to detect or identify fileless malware and formally proved to be working.

1.5 Our work in a nutshell

System API or system call is a way for a program or process to interact with the operating system. These system calls are used by the malware to gain control over an operating system. In this study, we explain different methodologies that allow us to intercept Windows system calls; and then understand the use-case of every method. We use one of the methods to automate the tracing of system calls. The study focuses on automating the tracing of system calls to log the behavior of malware, including fileless malware and their variants. The study does not focus on improving a pre-existing detection algorithm but only focuses on scaling the approach. We provide an open source tool to monitor malicious behavior with an assurance that the approach works for all kinds of malware that

exist today.

We test it on benign processes and various malware (including the fileless variants) to log their behavior. We categorize the test cases in order to cover a wider range of malware and verify our approach. Our tests would focus on two goals: the tracing process is automated, and verify that tracing works for fileless malware.

In upcoming chapters, we will be using *system calls* and *system APIs* terms interchangeably. *Hooking* will be used interchangeably used with “interception” and “tracing”. The term *Malware* will denote both filed and the fileless variant of malware. In the next chapter, Chapter-2, we briefly describe the related work that has been done in this field. Chapter-3 describes the methodologies that can be used to trace system calls. Chapter-4 gives a brief description of our implementation, followed by the tests and results. Finally, Chapter-5 concludes our work, and describes the possible work that can be done in the near future.

Chapter 2

RELATED WORK

In this Chapter, we review the prior approaches that have been taken to intercept system calls, and explain how they deal with the problem of malware identification.

2.1 API call sequence analysis

Unlike conventional dissembling of binaries, system API hooking is one of the memory-resident techniques increasingly being used. There are many reasons of API hooking being used for both legitimate and malicious purposes. It can be used either by the malware to intercept user sensitive data or to modify the behavior of the system call to identify suspicious behavior.

strace[46] is a powerful command line tool for debugging and troubleshooting programs. It captures and records all system calls made by a process and the signals received by the process. It was developed by Paul Kranenburg in 1991 for SunOS. *strace* was primarily developed for debugging a process and can be seen as a light weight debugger. It is very similar to a LINUX system call *ptrace*.

We review Youngjoon's [17] work, *CBM*'s[18] work, YongQ's[19] work, and *TTAnalyze*'s[20] work, based on system API interception in this section. Youngjoon's paper provides a good insight into hooking system calls, by describing its working and benefits.

Youngjoon uses API call sequence analysis to detect malware using a data-set of around 23 thousand malware; as compared to YongQ and CBM's paper with the data-set size of only $3,131$. The sample malware used is a filed malware, as there are no specific mentions of fileless malware or shellcodes. Youngjoon's work monitors Windows System APIs and then uses the Longest Common Subsequence (LCS) to detect malware and claims an accuracy of 99.8%. Although Youngjoon work shows good results in detecting malware it misses detection of fileless malware. It also lacks formal definitions and a detailed description of the LCS implementation.

YongQ and CBM's paper focuses more on clustering malware as compared to detecting them. Unlike Youngjoon, they use existing sandboxing techniques like "Cuckoo" to analyze malware. CBM compares two different sandboxes, on a scale of APIs hooked and some other benchmarks. Both the papers provide a strong mathematical explanation to classify malware. TTAnalyze built a system call tracing to detect malware, with only ten filed malware samples, popular in 2006. TTAnalyze did a great work a decade ago and proved that system API tracing is really effective in detecting malware. It has a detailed description of working with windows processes and tracking them. Unfortunately, TTAnalyze used an emulated Windows environment to test and not an original Windows system. Therefore having a lot of limitations, already mentioned in the paper. They claim to use emulated environment because monitoring on original Windows platform was not entirely straightforward.

2.2 Memory Analysis

Volatility Labs said, "Although it would make our jobs quite interesting if every investigation involved analyzing new malware samples and families, the reality is that many malware investigations only require analyzing memory samples in order to verify (or hunt

for) an infection by malware previously discovered in the wild."[21]

Memory analysis has been an active methodology to automate the detection of malware([21][22][23]). Intel recently launched a *"Thread Detection Technology"*(TDT)[23] that focuses on a solution for faster memory analysis. *TDT* tries to reduce the high CPU utilization consumed while scanning for malware on an active system. Intel guarantees to have accelerated memory analysis using their improved integrated graphics. Although, Intel accelerates analytic by improving hardware, it requires integration with its graphics drivers, with limited support for the systems.

Volatility[21] is a powerful tool written in Python especially for memory analysis. Volatility opens up a window of analyzing nits-and-grits of memory without worrying about the permissions, as every bit available in memory, can be read; making it easier to analyze fileless malwares as they reside only in memory. With enough knowledge required to understand the process management in memory and the default memory structure of Windows, it becomes easier to identify threats and point the odds. Volatility provides a plugin "apihooks" to find all the user and kernel mode API hooks. It finds several types of hooks from a memory dump of an operating system. It does that by detecting CALLs and JMPs to direct and indirect locations, and detects PUSH/RET instruction sequences.

On one hand, if analyzing malware inside the memory have an ample number of benefits, it also has its own repercussions. The problem begins with analyzing memory dumps/snapshots of huge sizes, that are supposed to be taken at frequent intervals for further comparisons. Although *Volatility Labs* made efforts to automate the analysis by integrating Python scripts, to read the memory dumps and filter the data in our own way; it still involve a lot of manual efforts.

Sandboxing technique allows us to monitor malware and execute malware in an isolated environment. *Sandbox Cuckoo*[32] provides a feature not only to execute a malware in an isolated environment but also to intercept system APIs and understand its behavior.

Cuckoo uses C framework to intercept system calls, specified in their official documentation[32]. In Chapter-3.3.2 we talk more on Sandboxes[27], and compare it with other approaches.

Unlike all aforesaid work, we found Microsoft to be actively working on fighting fileless malware.

Microsoft Defender Advanced Threat Protection(ATP)[47] an Antivirus system developed by Microsoft, aims to defeat fileless malware by introducing *AntiMalware Scan Interface*(AMSI), that can allow Windows to inspect fileless threats even with heavy obfuscation. It combines behavior and memory analysis with machine learning techniques to identify the threats. As per Microsoft, AMSI is part of the range of dynamic next-generation features that enables antivirus capabilities in Windows Defender. Windows leverages AMSI extensively in JavaScript, VBScript and PowerShell to fight fileless malware. Since the work done by Microsoft is not public, it becomes beyond the scope of our research.

Other than Microsoft AMSI, all the aforesaid work are either limited to filed malware or involves a lot of manual efforts. Our work aims to automate the procedure, provide an open source framework that makes malware analysis much easier, and cover a wider range of malware. In the next Chapter, we discuss a few principles through which we can hook the system calls, and understand how we can use them for analyzing malware.

Chapter 3

POSSIBLE METHODS

In this Chapter, we review the principal ways through which we can intercept system calls and monitor malicious behavior, and explain the pros and cons of every method.

We begin with a simple way that Microsoft Windows allows us to add any functionality to the system and then other ways that developed with time to achieve a similar result for different purpose.

3.1 Override Windows PE loader

Before we dig deep into how to override Windows PE loader, we will take some time to explain how *PE loader* works. *"Windows Internal"*[24] is a great resource/book to understand Process management in Windows.

As we know, *"PE"* in PE loader stands for *Portable Executable*; and for any executable to execute two major functionality has to be performed by an operating system: *Linking* and *Loading*. Linking is the process of taking several smaller object files and joining them together as a single executable. Loading is copying the sections of a single executable into the system memory, prior to execution. Once copied, the loader then produces a process control block (PBC) to control program execution. Finally, the execution starts, usually by jumping to its main address or entry point, as shown in Fig. [**??**]

In Windows, the loading process is exposed by a set of user APIs in *"kernel32.dll"*, under the *CreateProcess* family. The instant we call *CreateProcess* API, we are technically asking Windows to run the loader. This involves creating the process object in the kernel and registering the process. Then creating the main thread of the process including stack, execution context and the thread object. Then starting the main thread and loading the appropriate DLLs in the context of the process.

Dynamic Linked Library(DLL) is the medium that allows us to add our desired functionality into any Windows process. Asking the Windows to load an additional DLL created by us, while loading any process into the memory, is a simple solution to achieve hooking of system calls. Although it sounds simple, there are few things we should keep in mind. Firstly, Windows restricts unsigned DLLs to be injected into the processes. The DLL needs to signed by Microsoft for efficient use. Secondly, It is very risky to play with user processes by injecting a custom DLL into them. It might lead to system crash or corruption. Lastly, Windows restricts users to inject custom DLL into system processes, even with elevated privileges, with the reason of error as "ERROR_ACCESS_DENIED".

Keeping all these things in mind, we can create a DLL which hooks the system calls, and then ask Windows PE loader to link it into every user process in the system. Changes in Windows registry is required in order to ask Windows PE loader to link the custom DLL. Note that, Microsoft has documented the best practices[25] to create a DLL, a great resource to have a look at.

3.2 Using System APIs

"You can install a hook procedure by calling the *SetWindowsHookEx* function and specifying the type of hook calling the procedure, whether the procedure should be associ-

ated with all threads in the same desktop as the calling thread or with a particular thread, and a pointer to the procedure entry point"[26].

Microsoft provides a rich set of APIs that allows us to hook/intercept events in Windows OS, such as messages, mouse actions, and keystrokes. There are 13 types of predefined hooks, defined by Microsoft to monitor different events in a Windows system. Internally, these hooks modify the *Import Address Table* (IAT) located in the user-space of a process. IAT stores the addresses of particular library functions, imported from the DLLs. Similar tables exist in kernel space to keep a record of system interrupts and input/output request packets. On modifying the IAT, the memory location of the original function changes to the intermediate intercepting function. The intercepting function can then internally call the original target function and return the response, leading to the interception of system calls, as shown in Fig. [**??**]. Malicious applications can hence try to intercept the system calls and return arbitrary responses, thereby fooling a process of using the API incorrectly and crashing.

It is quite easy to write a small piece of code to monitor keyboard strokes and log user passwords and other important data. On one hand, it is a very useful set of APIs to detect malware, on the other hand this set of APIs is one of the most commonly used APIs by malware authors too. A scripting language like *PowerShell*[39] has also incorporated these APIs for administration purpose. But it turns out that PowerShell scripts have been one of the most commonly used technology by fileless malware in Windows Systems.

3.3 Modify a process memory

Windows PE Loader and system libraries allow us to achieve API hooking at the system level, where we can monitor multiple processes together. There are other ways

in which we can manually hook into a single process, by manipulating or modifying its memory.

3.3.1 Accessing memory via another process

We have already seen that we can modify IAT in order to hook system calls, by using the system APIs. But it's plausible to think if it can be done programmatically or manually without using any system library. To understand how it is possible, it is important to understand how a DLL is loaded into process memory.

Well, we know that every process executes in its own virtual memory space. The operating system, therefore, maintains a mapping of the physical memory addresses to its corresponding virtual memory addresses. It is because, it eases the loading of a DLL into the process memory. Note that, the memory address space of every DLL is pre-defined by the operating system; this is done to simplify the job of the Windows PE loader, as no additional table is required to map them to virtual space.

For example, the DLL 'A' will always be loaded from $0x1000$ to $0x2400$ memory address in every process, as defined by the operating system. Therefore, every process refers to the address space of the DLL in their own virtual address space, to access the DLL functions, as shown in Fig. [**??**].

Now that the address space of every DLL is fixed and already known, it becomes very easy to replace a particular memory address referring to a system API, with another memory address referring to a random function of a custom DLL. This can be achieved using a few system-calls namely, *GetProcAddress*, *memset*, and *VirtualProtect*. This method is very old and needs to be implemented very carefully, as it is not safe to modify a process memory. Therefore a workaround exists to do a similar job; where an external DLL is injected into the target process and executed using *VirtualAlloc* and *CreateRemoteThread* functions, by an external process. Here the process memory is not directly changed, but a

new section of memory is allocated by the external process, to execute the DLL having the hooking functionality.

3.3.2 Using Sandboxes

A Sandbox[27] is a security mechanism that allows us to isolate an application(s) from critical system resources and other programs. It acts as an additional layer of security, that prevents malware or harmful applications to negatively affect the operating system. It has been a wide-spread solution for malware analysts to test malicious codes. Cuckoo[29] is a leading open source Sandbox, for automated malware analysis. For this study, we would just understand the benefits of a sandbox and its limitations.

Cuckoo allows us to isolate malicious applications, and also enables system API logs to understand malicious behavior. Although it sounds very promising, Sandboxing is only applicable if we execute a program specifically inside the sandbox. In general, it is not practical to run every process inside a sandbox. It can lead to a significant increase of memory consumption and poor system performance. Usually a fileless malware tries to exploit a vulnerability and leave no trace of its presence, but otherwise, it hides behind a dropper file which is usually office files, PDFs or an e-mail. Lack of awareness of fileless malware, have a natural advantage over malicious binaries, which have been a conventional threat to society. [30] and [31] provide a nice idea on Cuckoo Sandbox and its comparison with other sandboxes, for better understanding.

3.4 Change during compilation

"Is it possible to add an intercepting function through a compiler, while creating an executable?" To answer this question, we need to understand what compilers are.

Compilers are programs that convert high-level language to machine-readable lan-

guage. So, even if we assume that there exists a super-compiler that has the hooking functionality, without putting any efforts to develop one. There are still many drawbacks in this approach. Firstly, the executable binary that any compiler provides, can only contain statically linked libraries and not dynamically linked libraries (DLL). Therefore limiting the hooking functionality till statically linked libraries. Lastly, we will have to recompile all the existing applications using this super-compiler to achieve hooking at a global scale, which is not practicle.

In this Chapter, we understood the possible ways we can achieve hooking in a Windows system. There exist more ways to hook system calls, for example hooking via Rootkits, or hooking Network Interfaces (NDIS). But those approaches are currently beyond our approach. We focus on hooking at the user level and not system level. In the next Chapter, we will walk through the hooking implementation and evaluation of the results.

Chapter 4

IMPLEMENTATION AND EVALUATION

We understood in Chapter-3, the possible ways to intercept system calls. We will be primarily using the first approach i.e. Chapter-3.1 to achieve automated hooking of system calls.

4.1 Implementation

We used *Visual Studio Code*[33] and C++ to implement API hooking. *Visual Studio Code* is a source-code editor, developed by Microsoft, to develop Windows-based applications and Dynamic Libraries (DLLs). Our implementation also uses *Microsoft Detours*[34], a framework built by Microsoft to intercept function calls in a DLL of the windows system, and tested it on Windows 7 SP1 (x86).

In this section, we briefly understand the working of Detours and then deep dive into our implementation

4.1.1 Microsoft Detours

Detours is a very powerful tool and an equally dangerous tool. It is a library of useful APIs that can be utilized to hook any system call on ARM, $x86$, $x64$ and IA64 machines. It is commonly used to intercept Win32 APIs to add debugging instrumentation. As it

simplifies the hooking mechanism, it is beneficial for both malware analysts and authors. The internals of the Detours is highly complicated and technical, which is beyond the scope of our work. To use Detours, we must have sufficient knowledge of how Windows works, already described in Chapter-3.

Fig. [**??**] shows the workflow of basic hooking, but Detours work a bit differently, and the comparison has been shown in Fig. [**??**]. The dotted lines in Fig. [**??**] show the normal execution flow of any process without the Detours; and solid lines show the execution with the Detours. The detour-function is the function that is called, instead of the target function. The detour function must exist to intercept a system call. Unlike the trampoline function, which can be avoided and therefore is optional.

Detour is a set of C/C++ code and is important to build it on the same system that is supposed to be monitored or hooked. It gives us an easy framework to hooking system APIs, and allow the developers to choose the system calls. So, we integrate and build our set of codes with Detours, to achieve system call hooking, specifically focusing on identifying fileless malware. *Code Project*[35] provides a great resource to implement Microsoft Detours.

4.1.2 Adding Hooks

Detour implements every function call as a transaction. So, it is important to understand the need of following functions:-

- *DetourTransactionBegin*: Initiates a transaction
- *DetourUpdateThread*: Updates the transaction, with the details of the current thread
- *DetourTransactionCommit*: Brings the hook into action
- *DetourAttach*, *DetourDetach*: Used to start and stop a hook, respectively

Both *DetourAttach* and *DetourDetach* functions take two pointer arguments. The first argument is a pointer to the original target function; while the second argument is a pointer to the detour-function that we create. Note that, the detour function is supposed to have the same method signature as the target function, as shown in Fig. [**??**].

Fig. [**??**] shows an example of hooking *CheckRemoteDebuggerPresent* function, defined in *kernel32.dll*. *"pCheckRemoteDebuggerPresent"* is the pointer to the target-function, and *"MyCheckRemoteDebuggerPresent"* is the detour-function (left undefined) with the same signature as the target function i.e. *CheckRemoteDebuggerPresent*. This simplifies the hooking procedure, as all the complexities behind the hooking process are being handled by the Detours efficiently. The detour-function was created for all the system APIs specified in *(Appendix-A)*, with an intention to monitor malicious behavior. The trampoline-function keeps a log of all the system calls, with the time-stamps and the frequency/usage of each API, in a map. We finally built the DLL named *WinHook.dll* using Visual Studio Code and moved ahead to the final step. It is important to note that, *WinHook.dll* writes the recorded data from the map into a file, once a process exits.

4.1.3 Activating Hooks

Once the DLL named *WinHook.dll* was created, we placed it in the Windows system partition (C drive). Then, we set two specific Windows Registry values: *AppInit_DLLs* and *LoadAppInit_DLLs* to activate the hook during the testing phase. These registries are located at: *"HKEY_LOCAL_MACHINE\Software\Microsoft\Windows NT\CurrentVersion\Windows"*, as shown in Fig. [**??**].

The value of *AppInit_DLLs* tells the Windows PE loader to load the respective DLL into every user process, of the current window session. Even after being informed, the PE loader does not load the specified DLL into user processes, unless we set another registry value named *LoadAppInit_DLLs*, with a value of 1. Also recall from Chapter-3.1, that Win-

dows restricts injecting a custom DLL into the processes unless signed by Microsoft. The workaround is that we set another registry value, named *RequiredSignedAppInit_DLLs* to 0, as shown in Fig. [**??**].

Finally, after setting the values of all three registries, we successfully hooked system APIs for all the user processes. Note that, since Windows 7 does not allow to inject an unsigned DLL like "WinHook.dll" into the system processes, we were only able to test, hook and log the user processes.

4.2 Proof Of Concept

In this section, we explain how we tested our developed framework and covered a large variety of malware. Our testing involves two major goals. The first goal is to automate the tracing of system calls and the second goal is to verify that this method works for fileless malware.

We divided the tests into four major categories. First, we inject the *WinHook.dll* into a running process in order to verify the results with pure binaries. Second, we executed shellcodes using a C program to verify the automation. Third, we executed a PowerShell script to verify the working of hooks for fileless malware. Lastly, we compare the results of hooking a simple PDF against a malicious PDF file.

4.2.1 Processes and Binaries

We chose Web Browsers to trace system calls and analyze the results, as they use a variety of functionalities in background. *Google Chrome* was purely a random choice, with

no specific reason to test it.

We first activated the hook by modifying the registry value as specified in Chapter-4.1.3, and then started the chrome browser. The Windows Task Manager in Fig. [**??**] shows all six running processes of Google Chrome, named *"chrome.exe"*. As there were six chrome processes that were created on starting the Google Chrome browser, our framework also created six text files. Each log file contained the Process Id (PID) of the respective process, with the name *"winhook_(PID)_log.txt"*, located at the Desktop of the Window user. It is not possible to view the log files during the hooking process, until all the chrome processes are closed.

The results showed that one log file out of all six logs was totally blank. The remaining five log files are shown in Fig. [**??**]. Each log file contained the list of hooked Windows APIs that were used by the process, with the count/usage of each API.

We can conclude that only one chrome process with process-id as 3044 tried to connect to the internet; as only that process used system calls like *send*, *connect* and *recv*. Rest of the API calls are pretty common in every instance of chrome. Additionally, it is interesting to see that chrome also used system API named *IsDebuggerPresent*, commonly used by malware to hide its behavior, and checked if it's running inside a Memory Debugger.

We verified the working of our approach on the Windows 10 operating system also, using the same implementation. This time we tested the working on *Mozilla Firefox* web browser, as shown in Fig. [**??**] and Fig. [**??**]. We can see a change in the results, that only two out of the five processes of firefox.exe were logged as a text file. The reason for this change was that Firefox did not import the *user32.dll* in the other three processes, therefore not allowing Windows PE loader to inject the hooking functionality. We verified the absence of the DLL using *Process Explorer* tool found in Windows SysInternals.

4.2.2 Shellcodes

The *Shellcodes*[16] were not simple to test, as compared to the binaries. A Shellcode can neither be directly executed like a binary, nor it can be executed like a DLL by injecting into another process. So, we created a C program and executed the shellcode as an executable binary.

A typical shellcode looks like a set of unicode characters representing machine code, as shown in Fig. [**??**]. These unicode characters can be directly executed using a C program, by storing them as a string and executing the character pointer as a function. We used this approach to identify the behavior of the shellcode. This approach is totally acceptable to prove the concept, as a shellcode is eventually executed inside a binary by exploiting some vulnerability on a victim's system. In order to simplify this testing, it was totally fair to run the shellcode directly as an executable, rather than exploiting a vulnerability by our self. The only aim is to identify the unexpected behavior using the hooking logs.

We used *MSFVenom*[37] to generate a malicious shellcode and created a reverse TCP connection at port 9999, commonly used to gain access to a victim's system. Once the shellcode was executed through the binary, the TCP port 9999 is reserved by the process. Fig. [**??**] shows Windows Command Prompt executing the shellcode via a process named "WinHook.exe" and the Windows Task Manager showing the process-id 4552 of the "WinHook.exe". The winhook log file was also created at the user Desktop, with the usage of hooked APIs being logged in the file. We verified the usage of the port 9999 by the process 4552 through the *"netstat"* tool shown in Fig. [**??**].

The system APIs used by the shellcode is shown in Fig. [**??**]. The result shows the use of *connect* system call and confirms the creation of the reverse TCP. We could have seen the usage of additional WinSock APIs*(Appendix-A)* if the reverse TCP was successful. Since

there was no other program/process to accept the TCP connection, in this test scenario, the shellcode timed-out at *connect*.

4.2.3 PowerShell scripts

PowerShell scripts turned out to be more challenging than shellcodes because of our direct dependency on the Windows PE loader to automate the hooking process.

A limiting factor to our approach is that the custom DLL is only linked to the process importing "user32.dll". As the PowerShell console did not import *user32.dll*, "WinHook.dll" was not injected by the PE loader. However, just to verify the working of our approach, we manually injected "WinHook.dll" into malicious PowerShell scripts and tested the scenario. Fig. [**??**] shows a PowerShell script executing as a KeyLogger[36].

The results were positive as the text file was created successfully with the usage of the system calls. Fig. [**??**] shows the usage of system APIs named *GetAsyncKeyState*, *GetKeyState*, and *GetKeyboardState*. These three system APIs are commonly used by malware to steal passwords and important user data, verifying that the method can be used to identify malicious PowerShell activities.

4.2.4 Office/PDF files

PDF and Microsoft Office files are one of the major sources of fileless attacks([41][42]), breaking the general convention of using malicious binaries. These are known as semi-fileless malware, as it involves a dropper file to execute the malware. Once executed, this malware resides only in memory. These particular malicious files hide a script or a shellcode behind the them. The scripts or shellcode can be activated either by exploiting a vulnerability or through escalated privileges.

In the following sections, we tested a benign PDF and a malicious PDF and then compared the results.

Benign PDF We created the benign PDF named *"sample.pdf"* using Microsoft Office and executed it after enabling the hook. The log file for the Adobe Acrobat Reader (*AcroRd32.exe*) with Process Id 3372 was successfully created, also shown in Fig. [**??**]. We found the usage of Hooking related APIs*(Appendix-A)* even in a benign PDF file, along with the usage of *OpenProcess* and *CreateRemoteThreadEx*. It is difficult for us to reason the usage of few system calls by the benign PDF, shown in Fig. [**??**].

Malicious PDF We created the malicious PDF named *"sample-mal.pdf"* using *Metasploit Framework*[38].

An existing exploit named *windows/fileformat/adobe_pdf_embedded_exe* was used to embed a malicious payload/behavior into a benign PDF. The malicious payload in the PDF created a *Reverse TCP* connection without the consent of the user. Just for testing purpose, we set the Destination IP of reverse TCP as *"127.0.0.1"* and the destination port as 8989. Fig. [**??**] shows the malicious PDF running on the Windows Desktop after the hook was activated.

The malicious PDF executed using the Adobe Acrobat Reader (*AcroRd32.exe*) having process-id 3180. It is important to note that, two log files and one additional PDF named *"sample.pdf"* were created on the execution of the malicious PDF. We found *sample.pdf* executing in the background (not visible as a PDF on the Windows Desktop) with process-id 2564 through Windows Task Manager. *Netstat* tool verified that the *sample.pdf* process was creating the intended Reverse TCP at port 8989. The Netstat tool and the logs of both the processes: 3180 (the malicious pdf) and 2564 (the additional process) are shown in Fig.

[**??**].

Comparison We understood the working of the malicious PDF in the previous section and discovered the silent creation of another process *sample.pdf*. We found that the API usage of the malicious PDF (Pid= 3180) and the benign PDF (Pid= 3372) *do not* have any major difference. It is probably because of *AcroRd32.exe* being a common benign software used to view PDF files.

The good part was that we *were* able to monitor the behavior of the additional process (Pid= 2564) created by the malicious PDF to initiate Reverse TCP. The usage of the *socket* function by the process 2564 is shown in Fig. [**??**]. Very similar to Chapter-4.2.2, we could have seen the usage of additional network APIs, if the reverse TCP was successful.

By performing the above tests, we tried to cover a wider range of malware and verified that it is possible to identify them by hooking system calls. We also analyzed the challenges that we faced during the automation of tracing the system calls. We found a good and a bad thing about the Windows system that helped and blocked our motive respectively. In the next Chapter, we summarize our study and explain the limitations and future work.

Chapter 5

CONCLUSION AND FUTURE WORK

5.1 Conclusion

We revisited the problem of detecting malware. We analyzed the challenges faced in malware analysis, especially due to fileless malware. We understood the process of hooking Windows system calls and reviewed the work already been done. We discovered that system call hooking hasn't been tested on the fileless malware and analyzed existing methodologies and understood their pros & cons. We chose one of the methods and implemented it to automate the process of tracing system APIs to detect malicious behavior. Finally, we evaluated our method and proved its effectiveness, to detect both filed and fileless malware.

5.2 Limitations

Although we were able to raise the usefulness of API hooking to identify malicious behavior, we were not able to truly automate the process for fileless malware. The following factors limit our approach to automate API hooking.

First, the Windows PE Loader injects a custom DLL only into the process importing *user32.dll*. This made our approach capable enough to monitor user processes but not automatized enough to monitor PowerShell scripts, as explained in Chapter-4.2.3.

We were also restricted to test only the user processes, as Microsoft Windows restricted to profile the system processes. It can be taken as a positive aspect too because it makes it harder for malware to infect a system process.

5.3 Future Work

Malware authors always try to be unpredictable and unusual. It becomes difficult for us to predict the system API/calls that could be used by the malware authors. Therefore, it is very important for us to regularly improve and update the framework and walk one step ahead of the malware authors. Our framework is currently limited to $x86$ Windows architecture and does not support $x64$ architecture, making it open for an extension toward other platforms.

Also, devising a new method to automate the tracing of the PowerShell scripts is very important. TrendMicro[40] states that "Malicious PowerShell scripts are a key ingredient to much fileless malware". Therefore a workaround has to be designed to identify a wider range of fileless malware.

Appendix A

APIS HOOKED

This is the list of all the Win32 APIs we hooked:

Registry Apis

- RegCreateKeyExA
- RegCreateKeyExW
- RegSetValueExA
- RegSetValueExW
- RegCreateKeyA
- RegCreateKeyW
- RegDeleteKeyA
- RegDeleteKeyW
- RegCloseKey

System Apis

- CreateToolhelp32Snapshot
- CreateProcessA
- CreateThread
- CreateFileA
- CreateFileW

- OpenFile
- _lopen
- DeleteFileA
- DeleteFileW
- OpenProcess
- CreateRemoteThread
- CreateRemoteThreadEx
- WriteProcessMemory
- ReadProcessMemory
- NtOpenFile
- NtCreateFile
- NtRenameKey
- WinExec
- LookupPrivilegeValueA
- LookupPrivilegeValueW
- ExitWindowsEx

Hooking Apis

- SetWindowsHookExA
- SetWindowsHookExW
- CallNextHookEx
- UnhookWindowsHookEx
- GetAsyncKeyState
- GetKeyState
- GetKeyboardState

Crypto Apis

- CryptBinaryToStringA

- CryptBinaryToStringW
- CreateEventA
- CreateEventW
- CreateEventExA
- CryptDecrypt
- CryptEncrypt
- CryptDecryptMessage
- CryptEncryptMessage

WinSock Apis

- socket
- send
- recv
- listen
- connect
- bind
- gethostbyname
- gethostbyaddr

Debugging Apis

- IsDebuggerPresent
- CheckRemoteDebuggerPresent
- OutputDebugStringA
- OutputDebugStringW

Network Apis

- URLDownloadToFile
- HttpOpenRequestA

- HttpOpenRequestW
- HttpSendRequestA
- HttpSendRequestW
- InternetConnectA
- InternetConnectW
- InternetCrackUrlA
- InternetCrackUrlW
- InternetOpenA
- InternetOpenW
- InternetOpenUrlA
- InternetOpenUrlW

Resource Apis

- SecureZeroMemory
- memcpy
- wmemcpy
- memcpy_s
- wmemcpy_s
- VirtualProtect
- VirtualProtectEx
- VirtualAlloc
- VirtualAllocEx
- VirtualQuery
- VirtualQueryEx
- LoadLibraryA
- LoadLibraryW
- LoadLibraryExA
- LoadLibraryExW

REFERENCES

[1] SearchSecurity: *Malware*, `https://searchsecurity.techtarget.com/definition/malware`

[2] MalwareBytes: *All about malware*, `https://www.malwarebytes.com/malware/`

[3] Malware Wiki : *Creeper*, `http://malware.wikia.com/wiki/Creeper`

[4] SonicWall : *Cyber Threat Data*, `https://blog.sonicwall.com/en-us/2018/10/september-2018-cyber-threat-data-ransomware-threats-doubl` Sept 2018

[5] Center Of Internet Security : *Top 10 Malware*, `https://www.cisecurity.org/blog/top-10-malware-january-2018/`, Jan 2018

[6] *Multi-State Information Sharing and Analysis Center*, `https://www.cisecurity.org/ms-isac/`

[7] TrendMicro: *KOVTER, Gone Fileless*, `https://www.trendmicro.com/vinfo/us/security/news/cybercrime-and-digital-threats/kovter-an-evolving-malware-gone-fileless`, Aug 2017

[8] Emotet, `https://www.anquanke.com/post/id/94022`, Jan 2018

[9] Cybercrime: *Fileless infections*, `https://blog.malwarebytes.com/cybercrime/2016/03/fileless-infections-an-overview/`, Mar 2018

[10] *Fileless malware an insidious threat*, https://blog.malwarebytes.com/threat-analysis/2018/08/fileless-malware-getting-the-lowdown-on-this-insidious-threat/, Oct 2018

[11] Internet Security Threat Report $(ISTR)$: *Living off the land and fileless attack techniques*, https://www.symantec.com/content/dam/symantec/docs/security-center/white-papers/istr-living-off-the-land-and-fileless-attack-techniques-en.pdf, July 2017

[12] VMRay: *Poweliks Malware - Filelessly Persistent*, https://www.vmray.com/cyber-security-blog/poweliks-fileless-malware-analysis/

[13] *Metasploit Framework by Rapid7*, https://www.metasploit.com/

[14] Microsoft PowerShell, https://docs.microsoft.com/en-us/powershell/scripting/overview?view=powershell-6, Aug 2018

[15] MalwareBytes: *Malware Obfuscation Techniques*, https://blog.malwarebytes.com/threat-analysis/2013/03/obfuscation-malwares-best-friend/

[16] Exploit Database: *Shellcode*, https://www.exploit-db.com/docs/english/13019-shell-code-for-beginners.pdf

[17] Youngjoon Ki, Eunjin Kim and Huy Kang Kim. "A Novel Approach to Detect Malware Based on API Call Sequence Analysis," in *International Journal of Distributed Sensor Networks*, April 2015.

[18] Y. Qiao, Y. Yang, J. He, C. Tang, and Z. Liu. "CBM: free, automatic malware analysis framework using API call sequences," in *Knowledge Engineering and Management, pp. 225236*, Springer, Berlin, Germany, 2014.

[19] Y. Qiao, Y. Yang, L. Ji, and J. He. "Analyzing malware by abstracting the frequent itemsets in API call sequences," in *Proceedings of the 12th IEEE International Conference on Trust, Security and Privacy in Computing and Communications (TrustCom 13), pp. 265270*, July 2013.

[20] Ulrich Bayer, Christopher Kruegel and Engin Kirda TTAnalyze. "A Tool for Analyzing Malware," in January 2006

[21] Volatility: *Automating Detection of Known Malware through Memory Forensics*, `https://volatility-labs.blogspot.com/2016/08/automating-detection-of-known-malware.html`, August 2016

[22] TechRepublic: *New detection method identifies cryptomining and other fileless malware attacks*, `https://www.techrepublic.com/article/new-detection-method-identifies-cryptomining-and-other-fileless-m` in Feb 2019

[23] Intel *ThreatDetectionTechnology*, `https://www.intel.com/content/dam/www/public/us/en/documents/product-briefs/tdt-product-brief.pdf`

[24] Windows Internals : `https://docs.microsoft.com/en-us/sysinternals/learn/windows-internals`, 2017

[25] Microsoft: *DLL Best Practises*, `https://docs.`

microsoft.com/en-us/windows/desktop/dlls/dynamic-link-library-best-practices, 2018

[26] Microsoft: *Using Hooks*, https://docs.microsoft.com/en-us/windows/desktop/winmsg/using-hooks, 2018

[27] SearchSecurity: *Sandbox*, https://searchsecurity.techtarget.com/definition/sandbox

[28] *Most common sandbox evasion techniques*, https://www.apriorit.com/dev-blog/545-sandbox-evading-malware

[29] Cuckoo Sandbox, https://github.com/cuckoosandbox/cuckoo

[30] X. Yang and S. Deb, "Cuckoo search: State-of-the-art and opportunities," *2017 IEEE 4th International Conference on Soft Computing & Machine Intelligence (ISCMI), Port Louis*, 2017, pp. 55-59.

[31] Cuckoo vs Reality, https://blog.avira.com/cuckoo-sandbox-vs-reality-2/, Nov 2014

[32] Cuckoo Sandbox Documentation, https://buildmedia.readthedocs.org/media/pdf/cuckoo-monitor/latest/cuckoo-monitor.pdf

[33] Microsoft Visual Studio Code, https://code.visualstudio.com/, 2017

[34] Microsoft Detours, https://github.com/Microsoft/Detours/, last updated 2019

[35] API Hooking with MS Detours, https://www.codeproject.com/Articles/30140/API-Hooking-with-MS-Detours, Oct 2008

[36] *Creating a Key Logger via a Global System Hook using PowerShell*, https://hinchley.net/articles/creating-a-key-logger-via-a-global-system-hook-using-powershell/

[37] MSFVenom: *Binary Payloads*, https://www.offensive-security.com/metasploit-unleashed/binary-payloads/

[38] Metasploit: *Client Side Exploits*, https://www.offensive-security.com/metasploit-unleashed/client-side-exploits/

[39] Microsoft PowerShell, https://docs.microsoft.com/en-us/powershell/scripting/overview?view=powershell-6

[40] TrendMicro: *Fileless Threats abuse PowerShell*, https://www.trendmicro.com/vinfo/pl/security/news/security-technology/security-101-the-rise-of-fileless-threats-that-abuse-powershell, June 2017

[41] *Fileless malware attacks explained*, https://www.comparitech.com/blog/information-security/fileless-malware-attacks/, June 2018

[42] Carbon Black: *What is Fileless Malware*, https://www.carbonblack.com/resources/definitions/what-is-fileless-malware/

[43] LCG Kit: *Create malicious Office documents*, https://www.proofpoint.com/us/threat-insight/post/lcg-kit-sophisticated-builder-malicious-microsoft-office-document
Dec 2018

[44] TrendMicro: *Abusing Actions to run scripts*, https://blog.trendmicro.com/trendlabs-security-intelligence/

analysis-abuse-of-custom-actions-in-windows-installer-msi-to-run-
April 2019

[45] MalwareBytes: State of Malware 2019, https://resources.malwarebytes.com/files/2019/01/Malwarebytes-Labs-2019-State-of-Malware-Report-2.pdf

[46] strace, https://linux.die.net/man/1/strace

[47] Microsoft on Fileless threats, https://docs.microsoft.com/en-us/windows/security/threat-protection/intelligence/fileless-threats, Aug 2019