Dakota State University
Beadle Scholar

Masters Theses & Doctoral Dissertations

3-2023

MEASURING THE PERFORMANCE COST OF MANUAL SYSTEM CALL DETECTIONS VIA PROCESS INSTRUMENTATION CALLBACK (PIC)

Jacob Williams

Follow this and additional works at: https://scholar.dsu.edu/theses

MEASURING THE PERFORMANCE COST OF MANUAL SYSTEM CALL DETECTIONS VIA PROCESS INSTRUMENTATION CALLBACK (PIC)

A dissertation submitted to Dakota State University in partial fulfillment of the requirements for the degree of

Doctor of Philosophy

in

Cyber Operations

March, 2023

By

Jacob Williams

Dissertation Committee:

Dr. Cody Welu, Chair

Dr. Cherie Noteboom, Committee Member

Dr. Tyler Flaagan, Committee Member

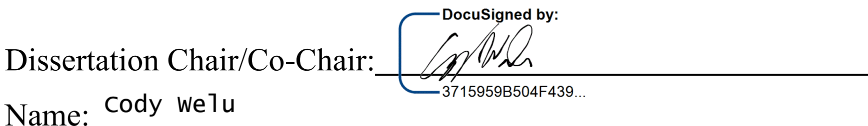
DocuSign Envelope ID: C5544F00-DAE1-44D2-8A76-41EAB4676A66

DISSERTATION APPROVAL FORM

This dissertation is approved as a credible and independent investigation by a candidate for the Doctor of Philosophy degree and is acceptable for meeting the dissertation requirements for this degree. Acceptance of this dissertation does not imply that the conclusions reached by the candidate are necessarily the conclusions of the major department or university.

Student Name: Jacob Williams

Dissertation Title: MEASURING THE PERFORMANCE COST OF MANUAL SYSTEM CALL DETECTIONS VIA PROCESS INSTRUMENTATION CALLBACK (PIC)

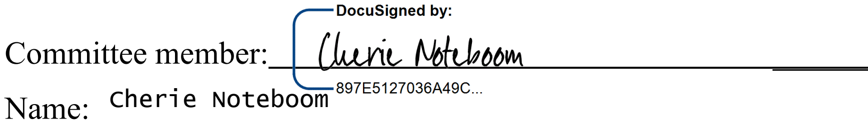
Dissertation Chair/Co-Chair:


Date: April 20, 2023

Name: Cody Welu

Dissertation Chair/Co-Chair: ____________ Date: ____________

Name:

Committee member:


Date: April 21, 2023

Name: Cherie Noteboom

Committee member:

Date: April 20, 2023

Name: Tyler Flaagan

Committee member: ____________ Date: ____________

Name:

Committee member: ____________ Date: ____________

Name:

Original to Office of Graduate Studies and Research
Acid-free copies with written reports to library

ACKNOWLEDGMENT

Writing an entire dissertation is nothing compared to trying to find the right words to express how grateful I am for the support I received throughout this entire process. A good start is to acknowledge the commitment that each of my committee members demonstrated in seeing this through. Dr. Noteboom, thank you for your sharp eye for detail and valuable insight you provided for not only my paper but my aspirations outside of defending my proposal. From the first time I submitted a rough draft of my work, you provided me with a well of confidence that I happily withdrew from any time I was lacking any of my own. Dr. Flaagan, I know you joined my committee as a late addition but that would be impossible to tell from the amount of feedback and support I received from you. Thank you for being a part of my journey to complete this paper. Dr. Welu, if I had not watched you go through this same process a few years ago I would not have had the courage to apply to the program and pursue my doctoral degree in the first place. You inspired me to push myself and create a body of work that I can truly be proud of. Thank you for being a fantastic dissertation chair and most of all, a lifelong friend. I only get a single page to write acknowledgements otherwise there are a lot of people that made cybersecurity a career for me and DSU a second home. To my DSU family that I have accumulated over the last nine years, thank you for everything.

This list would not be complete without acknowledging my family, especially my wife Rachel. I can’t promise you a future without having to listen to me blab about technology (that’s a lifelong commitment) but we can finally put this process behind us. You always knew just what I needed to keep myself motivated whether it was engaging with me about potential ideas or subtly leaving sticky notes on my computer monitors saying that you were proud of me. To my parents: my ability to write this section is the direct result of the encouragement and support

you have shown me throughout my life. To my dad specifically, thank you for the countless hours that you spent reviewing my work and receiving my phone calls. You have been my best friend throughout these last few years, and I cherish the time that we have been able to spend together relating to each other's academic experiences.

ABSTRACT

This quasi-experimental before-and-after study measured the performance impact of using Process Instrumentation Callback (PIC) to detect the use of manual system calls on the Windows operating system. The Windows Application Programming Interface (WinAPI), the impacts of system call monitoring, and the limitations of current detection mechanisms and their downsides were reviewed in-depth. Previous literature was evaluated that identified PIC as a unique solution to monitor system calls entirely from User-Mode, being able to rely on the Windows Kernel to intercept a target process. Unlike previous monitoring techniques, PIC must handle all system calls when performing analysis which requires an increase in processing. The impact on a single process was evaluated by recording CPU time, memory utilization, and clock time. Three different iterations that performed additional analysis were developed and tested to determine the cost of increased fidelity in detection. Results showed a statistically significant increase when PIC was applied in each version. However, the rate of impact was drastically reduced by restricting dynamic lookups to process initialization and the elimination of the Microsoft Debugging Engine. Future integration with existing detection mechanisms such as User-Mode hooks and Event-Tracing for Windows is encouraged and discussed.

DECLARATION

I hereby certify that this dissertation constitutes my own product, that where the language of others is set forth, quotation marks so indicate, and that appropriate credit is given where I have used the language, ideas, expressions or writings of another.

I declare that the dissertation describes original work that has not previously been presented for the award of any other degree of any institution.

Signed,

Jacob Williams

TABLE OF CONTENTS

MEASURING THE PERFORMANCE COST OF MANUAL SYSTEM CALL DETECTIONS VIA PROCESS INSTRUMENTATION CALLBACK (PIC) I

DISSERTATION APPROVAL FORM II

ACKNOWLEDGMENT III

ABSTRACT V

DECLARATION VI

TABLE OF CONTENTS VII

LIST OF TABLES X

LIST OF FIGURES XI

CHAPTER 1 1

INTRODUCTION 1

BACKGROUND OF THE PROBLEM 3
STATEMENT OF THE PROBLEM 6
RESEARCH QUESTION 7
NATURE OF THE STUDY 8
SUMMARY 10

CHAPTER 2 11

LITERATURE REVIEW 11

WINDOWS ARCHITECTURE 11
User-Mode vs Kernel-Mode 12
Windows Application Programming Interface 14
Native API 15
System Calls 16
SURVEY OF HOOKING METHODS 18
SSDT Hooking 19
User-Mode Hooks 20
User-Mode Hooks Performance 24
BYPASSING USER-MODE HOOKS 27
Unhooking 27
Manual System Calls 28

Detection of Manual System Calls 31
Event-Tracing for Windows *31*
Process Instrumentation Callback *34*
Summary 39

CHAPTER 3 41

RESEARCH METHODOLOGY 41

Research Method and Design Appropriateness 41
Research Questions, Hypothesis, and Variables 44
Population 45
Sample Collection 48
Instrumentation 49
Validity and Reliability 50
Assumptions 51
Scope and Limitations 52
Summary 53

CHAPTER 4 54

RESULTS 54

Data Collection 55
PICE Detection and Analysis Methods 57
Version 1 ..... *57*
Version 2 ..... *59*
Version 3 ..... *62*
System Performance 64
User-Mode and Kernel-Mode Performance *64*
Clock Time CPU Performance *65*
Memory Performance *67*
Statistical Analysis 69
Analysis Goals *69*
Survey of Statistical Methods *69*
Evaluation of Statistical Results *71*
Summary *73*

CHAPTER 5 75

CONCLUSION 75

Review of Findings 75

CPU Utilization *76*
Memory Utilization *78*
Detection Capabilities *80*
DISCUSSION 83
Using PIC to detect Manual System Calls *83*
Integration with Existing Detection Mechanisms *85*
Future Research *87*
LIMITATIONS 88
SUMMARY 90

REFERENCES 92

APPENDIX A: AVERAGE USER-MODE AND KERNEL-MODE TIMES 103

APPENDIX B: AVERAGE MEMORY UTILIZATION 105

APPENDIX C: AVERAGE REAL-TIME 107

APPENDIX D: SOURCE CODE LOCATION 109

APPENDIX E: GLOSSARY 110

LIST OF TABLES

Table 1. Windows subsystem DLLs and their use cases (Catlin et al., 2017a; "Windows API," 2021) 14

Table 2. NTDLL User-Mode Callbacks (Everdox, 2013) 35

Table 3. Actions performed by TIP and the corresponding system calls 56

Table 4. Average User-Mode and Kernel-Mode execution time in microseconds 65

Table 5. Clock Time execution in microseconds 66

Table 6 . Average Virtual and Physical memory usage in bytes 67

Table 7. T-test results for User-Mode and Kernel-Mode execution time 72

Table 8. Percentage of CPU utilization increase over Windows baseline without PICE 78

LIST OF FIGURES

Figure 1. WinAPI transition to Windows Kernel 16
Figure 2. SCN Relation to SSDT (Allievi et al., 2021) 17
Figure 3. Basic User-Mode Hooks Example Diagram 22
Figure 4. x64 Windows Calling Convention 28
Figure 5. Process Instrumentation Callback Information Structure (Ionescu, 2016)....... 36
Figure 6. Quasi-experimental diagram showing pre-test and post-test in the study's context........ 44
Figure 7. Diagram of PICEv1 Analysis 59
Figure 8. SysCallList Structure Layout........ 60
Figure 9. Diagram of PICEv2 Analysis 62
Figure 10. Diagram of PICEv3 Analysis 64
Figure 11. Comparison of CPU utilization time for each version of PICE 77
Figure 12. Average memory utilization for each version of PICE 79
Figure 13. Jumping directly to syscall instruction to avoid PICEv2 detections (Josh, 2021) 82

CHAPTER 1

INTRODUCTION

Information security has been an intense cat-and-mouse game between security professionals and malicious actors for many years. Malicious actors create new bypasses of security systems constantly, requiring a greater investment in detecting and blocking these techniques by security professionals. Often the first step of a detection method involves the ability to monitor the programs running on a computer system.

A common technique for monitoring is the interception and analysis of function calls (Abimbola et al., 2006; Eder et al., 2013). This is referred to as ‘hooking’ and is commonly used by antivirus companies as the primary way to determine if a program is malicious or not (Madani P, Vlajic N 2016). These hooks operate before the requested function call is executed, operating as an intermediary. If no malicious indicators are identified, then it is allowed to continue execution without interference.

These hooking techniques focus on ‘system calls’, which are functions provided by the operating system (OS) to facilitate interaction with the Windows Kernel (Bremer, 2012; Hand, 2020; Marhusin et al., 2008a; Tang, 2017). The Windows Kernel operates at the lowest code level of the OS and is referred to as Kernel-Mode. It is responsible for tasks such as allocating memory, creating a new process, or displaying information on the screen on behalf of User-Mode processes (Allievi et al., 2021). These tasks are exposed to developers through a series of application programming interfaces, commonly referred to as the Windows

Application Programming Interface (WinAPI). Performing essentially any task on a computer system requires programs to interface with the WinAPI. This creates an ideal choke point for monitoring suspicious behavior that occurs in User-Mode.

A technique attackers use to avoid detection is by manually invoking system calls (de Plaa, n.d.; Gavriel, 2018; MDSec, 2020). By skipping the convenience provided by WinAPI, attackers can operate undetected by utilizing the numerical representation of a system call instead. This number, known as the System Call Number (SCN), is used by the native WinAPI function to instruct the kernel what routine it would like to have called. Bypassing the WinAPI and using the SCN to instruct the kernel results in the bypass of hooking implementations and detection failure. Consequently, manual invocation of system calls would only be detectable from code running in the kernel.

Methods such as Event-Tracing for Windows (ETW) exist to receive notifications via the kernel about certain activities, which provide a convenient mechanism for the detection of potentially malicious activity (Alshehri et al., n.d.; Bode & Warnars, n.d.). The most helpful logs require the user's software to be signed by Microsoft (Microsoft, n.d.-b). Security researchers have published techniques that limit the effectiveness of ETW's reporting mechanism and subvert its results (Chester, 2020; de Plaa, 2020/2020; Teodorescu et al., n.d.). Further restrictions of ETW are covered in depth in Chapter 2.

Process Instrumentation Callback (PIC) presents an alternative detection technique by presenting a User-Mode (UM) capability allowing for the hooking of system calls when they return from the kernel. (Ionescu, 2016; MDSec, 2020; nick.p.everdox, 2013; Noah, 2017/2022; Ullrich, 2021). PIC has been a part of the Windows OS since the introduction of Windows Vista, utilized internally by Microsoft with no official documentation mentioning

its existence (Ionescu, 2016). Prior research has demonstrated PIC to be useful in detecting manual system call usage, but it has significant drawbacks that question it's viability as a business solution (MDSec, 2020; Noah, 2017/2022; Richard, n.d.; Ullrich, 2021). At large, the monitoring and inspection of processes across an entire OS is a laborious task that requires a low overhead (Marhusin et al., 2008b). Before PIC can realistically be implemented in existing detection stacks, defenders must first understand the potential impact on operational costs and benefits.

This study measured the performance impact of using Process Instrumentation Callback to detect the use of manual system calls on the Windows operating system. Chapter 1 will provide further background and the significance of the problem. Assumptions and scope are clarified to contextualize the model and the research questions associated with it.

Background of the Problem

Microsoft Windows is the most popular computer OS and is used throughout the world. Roughly 74% of home computers worldwide rely on it to be able to complete daily tasks such as browsing the Internet, checking up on social media, and working from home (Statista, 2021a). Similarly, 72.1% of servers across the world rely on Windows to host services and conduct business (Statista, 2020). This ubiquitous reliance has resulted in malware created by bad actors specifically target the Windows operating system.

Malware campaigns have increased significantly in recent years, targeting industries such as healthcare, finance, and public infrastructure (Arghire, 2022; Bartolik, 2022; HealthITSecurity, 2020). Once installed, this software relies on the ability to operate undetected while making changes to the OS. These changes vary depending on the goals of

the attack but often include things like credential stealing, encryption of documents, and crypto-coin mining (ChainAnalysis, 2022; HealthITSecurity, 2020).

The execution of the Windows OS is separated into two main modes: User-Mode (UM) and Kernel-Mode (KM) (Lopez et al., 2017; Microsoft, n.d.-d). Normal applications like internet browsers and mail clients execute within UM. KM contains lower-level system services that provide interactions between software and physical hardware. The relationship between UM and KM is a critical mechanism for the stability of the operating system. Chapter 2 will further expand on the security boundaries and separation of the two technologies.

To interact with the Windows Kernel from UM, Microsoft exposes a series of Windows application programming interfaces (WinAPI). These WinAPI consists of a set of functions exported through special dynamic link libraries (DLL) that are pre-installed on the system (Microsoft, n.d.-a). These each perform a targeted action on the system, such as creating a file on disk or initializing a page of memory. Parameters are then validated and passed to a low-level instruction called a system call. A numeric value corresponding to a routine in the kernel is assigned to each system call to differentiate which action should be performed. This is the last instruction that executes prior to exiting UM (Tang, 2017). Tasks will often involve the calling of multiple system calls in a certain order to fulfill an objective. For example, reading a file from disk require also requires a handle to be opened and closed to function properly. By requiring specific sets of parameters and permissions to execute, Microsoft created a predictable way for applications to run on the operating system.

To combat the threat of malware, antivirus companies have attempted to monitor the system calls by installing UM hooks in the API's prologue (Lopez et al., 2017). These hooks

intercept the execution of the function by first performing a series of checks to determine its legitimacy. The results of these checks determine whether execution is given back to the calling program or if the antivirus instead chooses to terminate execution. Installation requires the interception of the startup routine of a process and then the injection of code to search for the memory location of desired Windows API DLLs to overwrite them. This system-wide monitoring requires Administrator access to the system and quite often is accompanied by companion software running in the kernel.

Malware developers avoid detections by retreating lower in the call stack to prevent triggering system call detection. Since the WinAPI operates as a wrapper around the x86_64 instruction *syscall*, this presents a weakness. By discovering the correct System Call Number (SCN) and executing it manually, attackers can ensure that the hooked function is never called (de Plaa, n.d., n.d.; Gavriel, 2018; Hydra, 2020; MDSec, 2020). SCNs are an undocumented feature of Windows, subject to change between Windows releases (Allievi et al., 2021). This requires an application to know what version of the OS it is operating on so it can utilize the correct SCN. Numerous projects exist in the public domain that categorizes these numbers for ease of use and provides source code to automate their retrieval (Jurczyk, 2020; T, 2021/2022)

Process Instrumentation Callback (PIC) is a native feature of the Windows OS that allows developers to require a process to divert execution to a specified memory address every time it traverses from KM to UM (Bhansali et al., 2006). It is undocumented by Microsoft and there are no official references how to make use of it. The Windows Kernel naturally transitions from KM to UM every time it finishes the execution of a system call on behalf of a UM application (Allievi et al., 2021; Everdox, 2013). This is incredibly useful for

the post-processing of the system call because it allows for analysis of the UM process that requested it. Just like with UM hooks, a decision can be made to terminate the process if desired. PIC has gone largely unnoticed in academia, even though it has existed in the OS since Windows Vista (Bhansali et al., 2006). A few security researchers have spoken about this capability and created proofs-of-concept (POCs) that showcase its usefulness for offensive and defensive capabilities (Cocomazzi, n.d.; Ullrich, 2021). Security monitoring at scale requires consideration of the system impact despite the benefits of increased telemetry.

Statement of the problem

The problem is that manual system calls are used by malware to evade modern detection mechanisms on Windows and undermine the confidentiality of its users. Operating at the lowest level of UM, manual system calls force defensive technologies to explore alternative solutions.

Current implementations of UM hooks lack complete coverage of the execution stack (MDSec, 2020). This has created a situation where malicious code can operate undetected by purposely extracting and manually invoking system call instructions. Current detection of manual system calls relies heavily on ETW to provide KM telemetry (de Plaa, n.d.; Hydra, 2020). KM telemetry that can assist with the detection of manual system calls is limited to a subset of Microsoft-approved vendors that submit themselves to an in-depth verification process and membership of the Microsoft Virus Initiative (MVI) (Microsoft, n.d.-b). Research also shows that ETW's reporting mechanism is unreliable and is frequently bypassed (Chester, 2020; de Plaa, 2020/2020; Teodorescu et al., n.d.). Additionally, ETW's telemetry fails to provide detection for manual system call activity, rather reporting specific events that

are provided from the kernel. PIC fills the gap left by UM hooks while preserving the ability to monitor entirely from UM.

Although companies could previously rely on UM hooks to monitor process activity, it is no longer a sufficient detection mechanism for applications that utilize manual system calls. Process Instrumentation Callback represents a solution to this problem by presenting an alternative way for companies to be able to achieve comparable results without the requirement of Microsoft code signature or kernel modification.

Proper adoption of new technologies requires proper baselining to be able to gauge their performance and therefore usefulness in the real world (GenPact, 2014). By its nature, hooking introduces overhead complexities that must be accounted for. Previous UM hooks have the advantage of being able to designate a subset of functions to be hooked. PIC has no such luxury, requiring the process as a whole to be hooked, resulting in the interception of every system call the process makes (Ionescu, 2016; Marhusin et al., 2008b).

Research Question

This study measured the performance impact of using Process Instrumentation Callback to detect the usage of manual system calls on the Windows operating system. This involved a comparative analysis between process execution without intervention and with PIC interception and analysis. Measuring the performance impact of PIC is guided by the following research question:

What are the performance impacts of monitoring execution using Process Instrumentation Callback to detect the use of manual system call with regards to CPU time, memory utilization, and clock time?

An additional research question was created to guide the enhancements and upgrades to PIC's analytical capabilities:

Can we perform additional analysis using PIC to detect manual system calls and what is their effect on performance?

These questions together guided the model's construction and analysis by defining the focus of this research.

Nature of the Study

This study utilized a quantitative approach to measuring the results of the model, with and without PIC treatment. Creswell (2018) defines experimental research as seeking to determine if a specific treatment influences an outcome. Experimental research can vary from true experiments where samples are chosen completely at random or the opposite where samples are non-randomized and chosen purposefully.

Quasi-experimental before-and-after designs specialize in the evaluation of pre-post intervention scenarios, where randomization is either not possible or irrelevant to the outcome of the study (Harris, 2006). The study of before and after design allows a researcher to construct a baseline measurement before treatment and then compare the measurements from treatment to determine the impact (Kumar, 2014). This study reused the same programs and operating system before and after treatment to measure performance impact of PIC. Therefore, randomization was not practicable for this study. The quasi-experimental methodology was chosen to guide this study for this reason.

To maintain validity, researchers must maintain strict guidelines that mitigate internal and external influences (Creswell, 2015). Internal influences may involve things like time

passing between experiments and unequal selection of sample groups. External influences can include things like unbalanced treatment of the population between pre- and post-testing. To mitigate these, this study utilized a virtualization environment that allowed identical testing scenarios to be repeated without change. This allowed the exact instrumentation to be applied evenly across groups and to maintain the validity of results.

Reliability can be established when a research instrument can provide similar results when used repeatedly under similar conditions (Kumar, 2014). This is accomplished by repeating each testing scenario a total of 30 times and recording the results. This number was chosen based on previous studies that also monitor the performance impact of hooks to assess consistency (Marhusin et al., 2008). Any irregularity from the average is apparent and can be investigated. Assuming the same experimental environment, the results should have consistency between tests.

With a quasi-experimental design in mind, the researcher created a model that: [1] monitored the execution of system calls, [2] intercepted or activated upon return from KM, [3] validated the origin of the call, and [4] recorded the runtime overhead. A sample program was developed to simulate system call activity on Windows. Existing POCs that utilize PIC were included in the model and further expanded upon in three different versions. Each version explores stability and enhances the analytical capabilities for manual system call detection.

Summary

The purpose of this research is to analyze the performance impact of detecting manual system calls utilizing PIC. This chapter presented an introduction to the topic area and the background of the research problem. The methodology that this study used is a quasi-experimental before-and-after study. Variables such as CPU time, memory utilization, and clock time were recorded to determine the impact.

Chapter 2 will present a literature review that reviews the current state of the research and illustrate the significance of the problem area. Previous research in both PIC and system call detection are surveyed and explained in detail. Additional topics include the progression of WinAPI monitoring, manual system calls and how their use, existing detection methods, and an overview of the Windows architecture separation of UM and KM.

CHAPTER 2

LITERATURE REVIEW

Chapter 1 introduced the topic area and a brief background of the significance of the problem. This study measures the performance impact on a single computer system when PIC is monitoring for manual system call usage. The chapter also defined the study's problem statement, research question, and research design. Chapter 2 provides additional background and significance for the topics covered in Chapter 1. This includes a basic overview of the Windows operating system and how programs interact with it. Discussion of the design decisions to separate execution into UM and KM is explored, with added context from relevant literature in the topic area. The history of hooking is presented to provide the context of the environment and the relevance of the problem. This chapter reviews previous work that involves both the use of manual system calls and their current detection mechanisms, including PIC. Lastly, this section looks at how similar studies have collected and analyzed the performance of hooks.

Windows Architecture

Microsoft officially released the Windows 10 operating system in July 2015 and declared it to be the last version of the operating system they would be releasing, with the exception of releasing updates for existing Windows 10 installations (Catlin et al., 2017a). Often these updates include security mitigations for vulnerabilities that are discovered in the

operating system. Windows 10 saw a rapid increase in the adoption rate compared to previous versions, nearly doubling the number of end-users in two years (Statista, 2021b).

Windows dominates the market by holding 74% of all personal computers (Statista, 2021a). It has undergone many changes throughout the years as technology has progressed and better hardware has become available. The literature is described as it relates to the Windows 10 version of the operating system. This is the version used in the development and execution of the experiment.

User-Mode vs Kernel-Mode

Microsoft considers stability paramount to the success of the Windows operating system (OS) (Catlin et al., 2017a). To protect user applications from accessing or modifying critical OS data, Windows execution is managed in two modes: User-Mode (UM) and Kernel-Mode (KM). User applications such as an internet browser or email client run in UM. Critical system components like device drivers and services run in KM. These critical service components are known as the Windows Kernel.

Running applications are separated by their location in memory. Read and writes to memory locations occur using virtual addresses (Microsoft, n.d.-e). These virtual addresses are a logical representation of a physical address inside the memory used by the OS. When an address is translated from virtual to physical during a read or write operation, it is known as mapping (Catlin et al., 2017a). Windows includes a memory manager that assists with this operation and controls the separation of physical and virtual memory.

When executing a UM application, the OS automatically creates an object to contain all of the information required for it to operate (Catlin et al., 2017a; Microsoft, n.d.-d). This

object is called a process, and it is in a private virtual address space within memory. A process can be executed without worrying about altering the memory of another running process. Being private means that if something goes wrong and the process crashes, it is limited to the individual application that ran. Memory privacy provides stability to the OS by restricting memory access and isolating failure to the affected processes. Also, when the processor is running in UM it cannot access virtual addresses that are reserved for the OS (Catlin et al., 2017a). This denial of access is what creates the boundary between UM and KM.

Unlike UM, all processes running in KM share a single virtual address space (Microsoft, n.d.-d). This means that all the crash protection afforded to UM applications is not present. The code must be written with care to avoid any accidental overwrites of other processes or critical services. Due to the extremely delicate nature of this address space, Microsoft has restricted the ability of applications to run in KM. Consequently, developers commonly use driver code to access this capability when necessary (Catlin et al., 2017a).

Drivers are software that is allowed to execute within kernel space and usually provides some sort of service for the OS. They are released by hardware vendors to instruct the OS on how to interact with their products. (Catlin et al., 2017a). A simple example is a keyboard that comes with a driver that translates the physical actions of a button press to what Windows recognizes as input.

Windows 10 requires all drivers to be certified by Microsoft through a code review process called the Windows Hardware Compatibility Program (WHCP) (Microsoft, n.d.-f). WHCP is designed to maintain stability and security within the OS by conducting a thorough examination of every submission. Once a driver passes inspection, it is digitally signed by Microsoft. Every version of the operating system is aware of this signature and once it verifies

its authenticity, it is then allowed to execute in KM. Not only does this inspection ensure that driver code does not threaten the stability of the OS, but also allows Microsoft to prevent malicious code from executing in the kernel.

Windows Application Programming Interface

Nearly all the programs running in UM on Windows rely on services provided by the kernel and require a standard way to make requests to KM services (Catlin et al., 2017a; Lopez et al., 2017). These services include things such as allocating memory, input/output (I/O) operations, or the creating of processes. This is provided by the OS through the Windows application programming interface (WinAPI). The WinAPI abstracts away many of the complex details of how the OS works and provides a set of callable functions (Catlin et al., 2017a; Reddy, 2011). These functions correspond to the specific operation that is to be completed by the kernel before returning execution to the UM application. Common examples are *CreateProcess*, *VirtualAlloc,* and *WriteFile*.

To organize and package functions according to the different jobs that can be performed by the OS, Microsoft exports dynamic link libraries (DLL). A DLL is a binary file that can be included by applications that wish to use the exported functions provided by them (Catlin et al., 2017a). DLLs can be preinstalled in applications via the compilation process or loaded dynamically from the disk in UM. Microsoft-provided DLLs that provide interaction between UM and KM are called ‘subsystem DLLs’. Common examples are shown in Table 1.

Table 1. Windows subsystem DLLs and their use cases (Catlin et al., 2017a; “Windows API,” 2021)

NAME	PURPOSE	DLL

Table 1 (continued).

BASE SERVICES	File System, Devices, Processes, and Thread Management	Kernel32.dll
ADVANCED SERVICES	Registry Read/Write, Shutdown/Reboot, Service Management	Advapi32.dll
USER INTERFACE	Create windows, buttons, mouse/keyboard	Comctl32.dll
NETWORK SERVICES	NetBIOS, RPC, Sockets	Netapi32.dll

Native API

Another application programming interface (API) that is exposed to UM applications is the Native API (NTAPI). NTAPI is an undocumented interface that operates on a level below the subsystem DLLs, acting as the last layer before execution transitions to KM (Catlin et al., 2017a). Like WinAPI, a set of functions are exposed through a DLL named NTDLL. Over 450 functions are exposed by NTDLL that correspond directly to functions with the same name in the Windows Kernel that provides system services for user applications (Catlin et al., 2017a). Functions exposed to WinAPI from NTDLL are referred to as NT Functions because they all follow a similar naming convention of prepending the letters “NT” to the beginning of their name. This is to distinguish them from their WinAPI counterparts which have the same. Examples include *NtCreateProcess*, *NtAllocateVirtualMemory* and *NtCreateFile*. Figure 1 provides a visual representation of a WinAPI function call transition to KM.

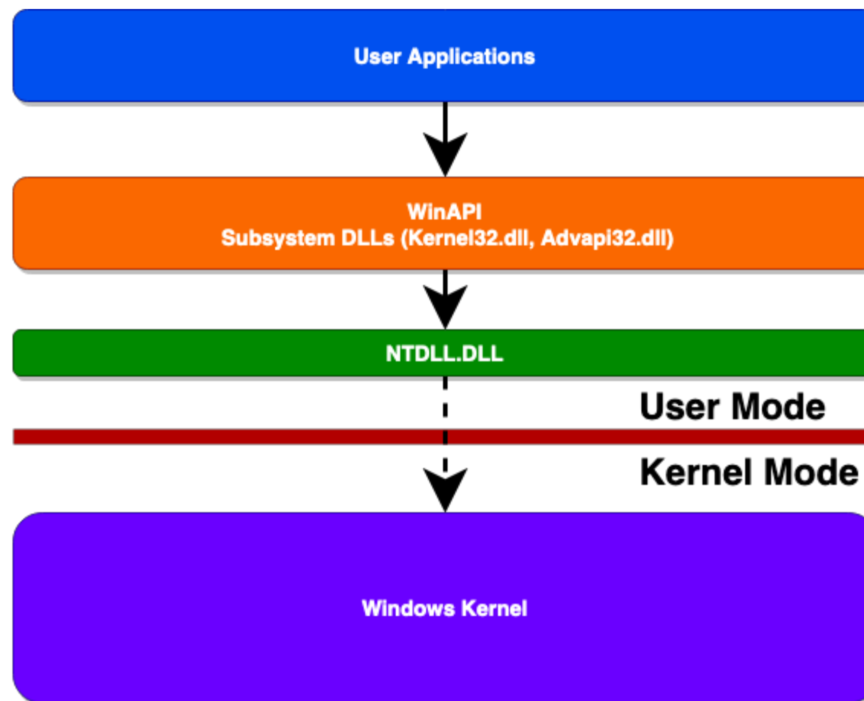


Figure 1. WinAPI transition to Windows Kernel

Microsoft does not support the use of the NTAPI for user applications officially since they reserve the right to make changes at any time between updates. Therefore, little to no official documentation exists for developers that would like to use it (Allievi et al., 2021). For most commercial software, the WinAPI provides all the necessary functionality to run their applications. The exception to this rule is programs created by Microsoft themselves, known as Native Images (NI). These applications often only require functionality exposed by the NTAPI and use it to manage other processes on the system (Allievi et al., 2021).

System Calls

On a 64-bit Windows 10 operating system, the assembly instruction *syscall* is the last instruction executed before the processor changes execution from UM to KM (Allievi et al., 2021; de Plaa, n.d.). When a UM application calls a WinAPI function, the corresponding NTAPI function is called which executes the *syscall* instruction. The *syscall* instruction

switches the processor to KM. Transitioning from UM to KM to invoke a system service is defined as a system call.

The Windows Kernel establishes a reference to the requested calling operation via the EAX register that is loaded with a number named the System Call Number (SCN) (Allievi et al., 2021). Just before the execution of the *syscall* instruction, the EAX register is loaded with the SCN (Allievi et al., 2021; de Plaa, n.d.). Once transitioned to KM, the SCN is used to determine which system service is executed. Next, any arguments provided to the operation are validated and passed along to the corresponding KM function. NTDLL is responsible for the placement of function arguments, insertion of the SCN into EAX, and finally, the *syscall* instruction is executed to inform the processor to transition to KM (Allievi et al., 2021).

To keep track of which SCN corresponds to what function, the Windows Kernel uses the System Service Dispatch Table (SSDT). The SSDT, pictured in Figure 2, contains the location of the system services and their corresponding SCN (Allievi et al., 2021).

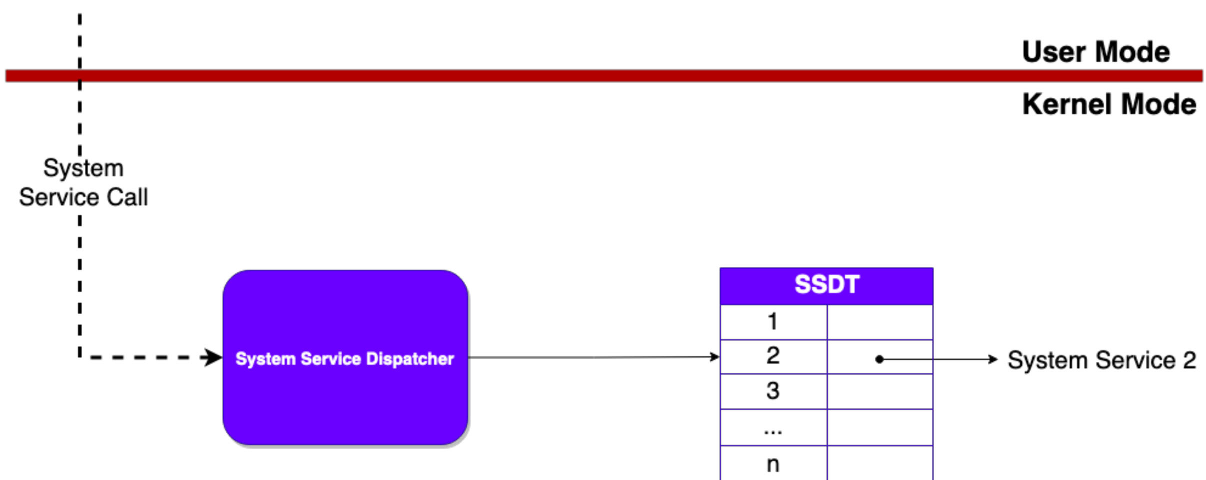


Figure 2. SCN Relation to SSDT (Allievi et al., 2021)

Since officially the SSDT is an undocumented feature, Microsoft reserves the right to alter or completely remove system services and their SCNs from the SSDT at any time (Allievi et al., 2021). Often between OS releases the SSDT is altered or even randomized to decrease predictability and break applications that use hardcoded SCNs to avoid detection. Future sections in this chapter provide more information regarding hardcoded SCNs and their use for detection avoidance. The KM process in charge of this lookup is known as the System Service Dispatcher (SSD). Not only does the SSD identify which system service should be called, but it is also in charge of saving the UM information required to return execution to the application and copying any function arguments into KM (Allievi et al., 2021; Lukan, 2014).

The interaction of the application process, WinAPI, management of the system calls via SCNs, and transitions between UM and KM are the primary concepts of the design of this study. These foundational concepts are referenced heavily in the following sections as their role in monitoring capabilities is described.

Survey of Hooking Methods

Hooking is the interception of specific functions or system calls to monitor and/or alter the execution of the specified call (Lopez et al., 2017). This is often helpful to provide information when the source code of an application is unavailable. Many debuggers make use of this functionality to examine what could cause an application to crash, allowing the user to quickly inspect the content being passed to these calls (Eder et al., 2013; Lopez et al., 2017; x64dbg, n.d.-b). A popular use of hooking functions is to perform analysis for cybersecurity.

Hooking provides security products with valuable insight into what an application is performing on a computer system. For example, if an API call is made, the parameters being

passed to it can be extracted and inspected for malicious content. This is a technique that has been used by antivirus (AV) companies for years to detect malicious software running on Windows (Catlin et al., 2017a; Lopez et al., 2017). The only thing that has changed has been wherein the execution stack of the system calls that a process is hooked.

SSDT Hooking

As previously discussed, the SSDT is a KM lookup table responsible for holding the locations of system services and their associated SCNs (Allievi et al., 2021). Historically, the best place to insert a system call hook would be to modify the SSDT to point to a different function location (Allievi et al., 2021; Lukan, 2014). This is known as SSDT Hooking. The location to which a call is diverted is called a detour function and is responsible for the analysis and inspection of the call. After it has finished, the detour function is free to either stop execution and terminate the process or direct execution to the system service as originally requested. This technique is invisible to UM applications since it all happens within KM, resulting in a seamless monitoring capability.

SSDT Hooking is not a technique that is unique to AV companies. Special malware variants, known as rootkits, have been taking advantage of the ability to stop and inspect processes on computer systems for ages (Catlin et al., 2017a; Kleymenov & Thabet, 2019; Monnappa, 2018). Rootkits run in the Windows Kernel and have full access to all KM components, including the SSDT. This level of access allows them to modify the SSDT just like an AV product and divert execution flow to their own malicious detour functions. Rootkits represent some of the most dangerous forms of malware for computer systems (Singh et al., 2017; Win et al., 2015).

All these third-party entities running code in the Windows Kernel created a stability problem for the OS. When critical structures such as the SSDT are hooked, it is extremely important to prevent code from crashing or containing bugs. If not, the entire system can crash, and cause outages that are out of Microsoft's control. With the release of the 64-bit version of Windows XP, Microsoft introduced a new kernel protection mechanism known as PatchGuard (PG) to help prevent this (Catlin et al., 2017a; Lukan, 2014). PG is responsible for detecting unwanted code from running in the kernel, keeping a close eye on specific KM components. If access is detected within KM that is not allowed, PG will crash the system entirely with what's known as a Blue Screen of Death (BSOD). This crash screen displays a message informing the user what had happened and immediately disclose the crash information to Microsoft's threat intelligence team to begin analysis (Catlin et al., 2017a). Microsoft discloses certain checks that it makes, such as the verification of the SSDT and the PG code itself (Catlin et al., 2017a). However, most checks that PG makes are unknown to the public, as are the intervals that it performs them. Frequent updates to PG are released to constantly evolve its detections and prevent any predictability by the engine (Catlin et al., 2017a). This is done purposefully to prevent third-party individuals from attempting to reverse engineer the execution of PG and bypass it.

User-Mode Hooks

PatchGuard (PG) left AV companies and malicious actors without a reliable way to monitor process execution on Windows and created the need for a new technique. While malicious actors continue to attempt to bypass PG to continue to use SSDT Hooking, legitimate companies could not afford the risk. The unpredictable nature of PG and the risk of

crashing the OS entirely are not good for their products. Knowing this, Microsoft offered an alternative solution to monitor system call activity by hooking the call itself within UM.

Figure 1 shows the next closest location to monitor for activities is NTDLL. This is the last place that UM execution passes before the transitioning to KM to perform a lookup within the SSDT. As discussed earlier in the chapter, NTDLL contains the NT functions that are responsible for setting up and performing the *syscall* instruction to transition to KM (Allievi et al., 2021). Just like with SSDT Hooking, UM hooks seek to alter the flow of execution by temporarily diverting it to a detour function. This requires the modification of NTDLL.

Modifying the on-disk version of NTDLL is impractical since it would invalidate Microsoft's digital signature of the file. Windows verifies the signature of Microsoft binaries before execution, refusing to continue if any issues are found (Allievi et al., 2021). This is solved by patching the DLL at runtime when it is loaded into the application's memory space (MDSec, 2020; Willems et al., 2007). Patching is defined in this paper as overwriting an existing set of instructions to alter execution.

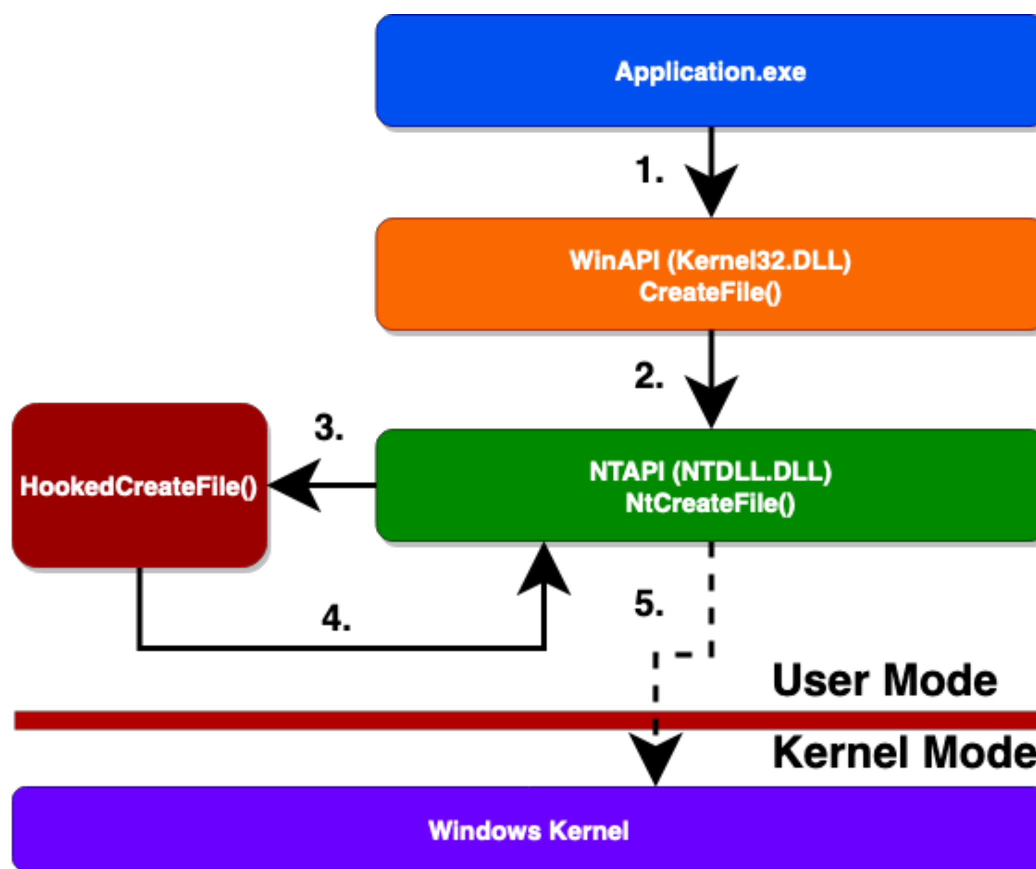


Figure 3. Basic User-Mode Hooks Example Diagram

As seen in Figure 3, a high-level overview of UM hooks is shown. The diagram uses *Createfile* and *NtCreateFile* as an example of a system call. The five basic steps are:

1. Application.exe calls the *CreateFile* WinAPI function and passes its input.
2. *CreateFile* calls the NTAPI function *NtCreateFile* and passes the Application's input.
3. A patched *NtCreateFile* diverts execution to a detour function *HookedCreateFile* which inspects the parameters for potential malicious indicators.
    a. An example could be a file path being written that is known to be abused by malicious actors.
4. If no malicious indicators are found, *HookedCreateFile* returns execution to NTDLL and allows it to perform a system call.

5. NTDLL performs executes the *syscall* instruction for NtCreateFile and execution transitions to KM.

Patching the in-memory version of NTDLL is the first step and the second step is finding an appropriate place to direct execution. The goal of UM hooking is to monitor the execution of all running processes on the OS. This goal represents an obvious chokepoint for system performance due to the sheer number of requests that must are inspected. It is normal for Windows to make hundreds of thousands of system calls a second, performing even more depending on the number of processors (Allievi et al., 2021). A single service would not be able to handle that. Therefore, a special DLL must be loaded into each process that can assist.

Endpoint Detection and Response (EDR) applications, which are the products AV vendors use to gather telemetry from the OS, are responsible for implementing this assistance. A popular way is using the framework developed by Microsoft called Detours (Catlin et al., 2017b; MDSec, 2020; Microsoft, 2002). A DLL can be written with the necessary logic to inspect hooked functions and then wrapped with the Detours framework to assist with the actual hooking. This DLL is then forcibly loaded into each process that starts on the OS, a technique that is known as DLL injection. Then it patches the in-memory version of NTDLL to direct execution to itself. When the execution flow is diverted, such as in the case in Figure 3, it calls a version of *HookedCreateFile* that is specific to the hooked process. This means that every process that is targeted for hooking is responsible for the analysis and inspection of its function calls. This solves the issue of widespread deadlock due to each process having its own dedicated DLL for hooking.

User-Mode Hooks Performance

Even though system-wide deadlock is avoided, UM hooks still provide an impact on the performance (Lopez et al., 2017; Marhusin et al., 2008a). EDRs need to prioritize as little increase of execution time as possible, this can often be the difference between a solution being deployed for use or disabled for convenience. One way to cut down on the number of system calls being tracked is by limiting APIs hooked. Certain NT functions are predisposed to be more helpful for a malicious application than others. For example, *NtProtectVirtualMemory* is a function that changes the memory permissions of a page of memory in a process. While legitimate uses for this API exist, it is of use to an attacker looking to exploit a system that has protected memory. Not every function can be easily identified as useful for malware, for example, *NtAllocateVirtualMemory*. The ability to allocate memory is something most applications, whether malicious or benign, might wish to do. It's a well-known tactic for malware to create a space in memory for itself to operate (Klein et al., n.d.). By focusing on NT functions used by current malware variants and threat actors EDR solutions can create a subset of functions to monitor. As tactics and techniques change with time, these lists are subject to alteration through product updates. Variations between vendors can be expected as well.

Previous research into UM hooks performance compared it against alternative implementations to gauge overhead (Lopez et al., 2017; Marhusin et al., 2008a). A non-zero impact on the execution time of an application is expected, but recording it is still important. Lastly, the effectiveness of a solution against current threats should be studied.

Marhusin (2008) looked at comparing three different test scenarios: a computer with nothing extra installed, with antivirus, and with an API hook program. The API hook program

only targeted a small subset of functions from KERNEL32. The researchers recorded the performance impact on ten commonly used programs at the time the research was performed. These programs were measured from the beginning of execution time up until the graphic user interface was displayed. Validity concerns were eased by reproducing this process a total of 30 times to be able to identify an average length of execution time (Marhusin et al., 2008). The paper found that executing each test a total of 30 times provided consistent results without requiring too much time.

It is important to note that when this research was performed, UM hooks were a new concept and the antivirus solution used file signature based detection. A hash of every file written on disk was compared to a static list of known malware samples to determine legitimacy. This means that UM hooks were being compared to scenarios in which the target program's API execution was not being inspected. At best, the signature based detection introduced a time delay when the file is loaded from the disk.

The results of the paper found that by targeting a small subset of functions rather than all API calls, they were able to avoid a high overhead (Marhusin et al., 2008a). Additionally, performance impacts were similar to those produced by the antivirus product.

Another paper surveyed the current hooking approaches across multiple platforms and operating systems (Lopez et al., 2017). Since the research included systems other than Windows, researchers were able to observe additional hooking techniques. The operating systems studied included Windows, Linux, macOS, iOS, and Android (Lopez et al., 2017). For each system, a variety of open-source and closed-source solutions were chosen. Unlike Marhusin, who used publicly available software as their target, these researchers created a

custom test program (Lopez et al., 2017). For each operating system, the program performed the following tasks:

- Creates 10,000 text files
- Opens each file, writes the message "Hooking Testing," and closes the file
- Opens each file, read the contents into a buffer and closes the file
- Deletes all 10,000 files

Before the execution of these operations, a baseline of system performance was recorded. These statistics were recorded once more when execution was completed, providing two sets of results. The variables used to measure the impact were: physical memory size, virtual memory size, CPU user time, CPU kernel time, and clock time (Lopez et al., 2017).

Each operating system presents challenges and techniques to hook API calls (Lopez et al., 2017). Since this study focuses on the Windows 10 architecture, only results pertinent to the Windows OS are explored from the paper. To record the variables of interest, Lopez (2017) utilized the built-in WinAPI functions that Microsoft provides.

- *GetProcessMemoryInfo* – Memory Utilization
- *GetProcessTimes* – CPU Time (User & Kernel)
- *QueryPerformanceCounter* – (clock time, start/end time)

On the surface, using WinAPI functions to monitor the execution of UM hooks seems like a recursive loop. This is not the case due to the proper scoping of the paper. Since the paper was interested in specifically observing the execution of certain functions, measurements were taken before and after file creation. Therefore the functions that recorded the study's variables did not interfere with the timekeeping.

The results of the paper presented the hooking performance of three Windows utilities: Rohitab, WinAPIOverride, and Frida (Lopez et al., 2017). All were shown to have an impact on the performance of the system in multiple categories, with Frida being the most memory intensive. While the paper tries to include Microsoft Detours in its literature review, it did not include it in its analysis. This is disappointing since most EDR solutions utilize Detours as their API hooking engine (Hand, 2020; Microsoft, 2002). However, the use of multiple metrics to measure performance was an improvement over Marhusin's research.

Bypassing User-Mode Hooks

Unhooking

As the name suggests, UM hooks run exclusively outside of KM. While this aids Microsoft's mission of creating a more stable KM environment, it now leaves the hooks themselves vulnerable to attack (Apostolopoulos et al., 2021; MDSec, 2020; Tang, 2017). Plenty of research has been done into identifying the presence of UM hooks in a process and then dynamically removing them (Tang, 2017). This usually involves restoring the overwritten portions of NTDLL to their original "clean" state before executing any NT functions. This renders the UM hooks blind and unable to analyze any system calls made thereafter by that specific process. A major downside to this technique is the identification that hooks exist, which often rely on static signatures specific to a vendor or version of the OS (Tang, 2017).

Manual System Calls

As defined earlier, one purpose of NTDLL is to set up the necessary parameters to perform the *syscall* instruction that switches the processor's execution state to KM (Catlin et al., 2017a). Figure 3 demonstrates how UM hooks take advantage of this chokepoint and inject themselves before the execution of NT functions. The parameters are analyzed for malicious indicators and if nothing is found, step 4 is followed. Execution returns to NTDLL to perform the system call. To bypass analysis completely, attackers can execute step 5 themselves.

This is known as a manual system call. Without the help of the WinAPI or NTAPI to construct the necessary parameters to transition into KM, an attacker application needs to be able to recreate this operation by hand. Fortunately, Microsoft requires that all x64 Windows system calls follow the same general calling convention(MDSec, 2020; Microsoft, n.d.-g).

```
MOV R10, RCX
MOV EAX, <SCN>

SYSCALL

RETN
```

Figure 4. x64 Windows Calling Convention

Figure 4 shows the exact x86_64 assembly instructions that are executed by the processor before it crosses to KM. The predictability of this format, also known as a stub, is a boon for attackers since it limits the number of instructions they need to supply. Windows mandates that the EAX register contains the SCN of what system service is to be executed (Allievi et al., 2021). As discussed in the Windows Architecture section of this chapter, these SCNs correspond to a row in a KM structure known as the SSDT. Microsoft provides no guarantees on the stability of the relationship between what SCN corresponds to a system

service, often intentionally changing them between releases (Allievi et al., 2021). This leaves two options, knowledge of what OS is being targeted during development or the dynamic retrieval of them at runtime.

NTDLL provides all of the necessary information with every release of Windows (Allievi et al., 2021; de Plaa, n.d.). Statically reverse engineering each NT function can reliably produce the stub illustrated by Figure 4. Security researchers have created many public resources around the retrieval of these stubs. The next section will survey different approaches to accomplish this.

Windows System Call Tables

Jurczyk (2020) maintains both a popular GitHub repo and a website that catalogs the SCNs for all NT functions. This database extends from Windows NT SP3 to Windows 10 20H2. If the target OS version can be identified before execution, Jurczyk's research allows an attacker to determine the appropriate SCN without having to extract the information from NTDLL. This is a key project that is referenced quite heavily by other researchers, acting either as inspiration or the source of their SCNs.

SysWhipsers & SysWhispers2

The original SysWhispers project henceforth referred to as "SysWhispers1", continued Jurczyk's research by automating several key parts (T, 2019/2022). By inputting the NT Function and the OS version desired to execute a manual system call, SysWhispers1 generates the necessary code assembly code to invoke it. As demonstrated in Figure 4, only a few instructions are required. SysWhispers1 substitutes the correct SCN into the MOV EAX parameter based on Jurczyk's data that it stores locally (T, 2019/2022). Security researchers can then import this stub directly into their code, skipping the process of performing this

resolution themselves. The major downside to this project is that to contain all of the necessary stubs, the generated output can be quite large. This can lead to problems if the size is of concern.

SysWhispers2 improves on this process greatly by eliminating the requirement to specify what version of Windows is being targeted (T, 2021/2022, p. 2). It can perform the generation of the correct stub during execution time instead of during development time. This means that during runtime the application is responsible for figuring out the version of the OS. This decreased complexity came with a shrinkage in size. SysWhispers2 no longer relies on Jurczyk's pre-generated list to determine what SCN to utilize. Based on a technique first published by MDSec (2020), the SCN can be identified by performing the following steps:

1. Parse the export address table of NTDLL for all functions names that begin with "Zw".
2. Replace "Zw" with "Nt" for every function and generate a hash of the name.
3. Sort them by memory address in ascending order.
4. Store the index of the table as the SCN.

This technique is possible because Windows loads NTAPI functions into memory it does so in ascending order based on the SCN. This method can work across Windows versions and drastically reduces the amount of code that is required to compensate for all the possible OS versions (MDSec, 2020; T, 2021/2022, p. 2). Additionally, it eliminates potential indicators of compromise that can be generated by patching NTDLL to remove hooks.

Detection of Manual System Calls

Event-Tracing for Windows

The circumvention of UM hooks is a serious problem that requires supplemental monitoring capabilities to detect the use of manual system calls (Gavriel, 2018). The reality is that without the ability to execute code in KM, AV companies cannot compete. Microsoft recognized this gap and created Event-Tracing for Windows (ETW).

ETW is a tracing mechanism that provides UM applications and KM drivers the ability to provide, consume, and manage log and trace events (Allievi et al., 2021). This is a powerful detection mechanism for monitoring execution across the OS. It is exposed to UM applications running with administrative privileges through an API, which is heavily utilized by Microsoft-developed applications to report their activities. ETW is broken down into three distinct parts:

- *Controller* – Start/Stop event producing.
- *Provider* – Generate events.
- *Consumers* – Consume events.

A good example of ETW usage is the Windows Event Log. The Windows Event Log operates as a consumer, displaying all of the events generated by EventLog-Application and EventLog-System providers (Palantir, 2019). The controller in this context is the Event Log service, which is started at the boot of the OS, to ensure its execution. Applications that consume ETW events can operate as their controller, enabling the flow of events when needed (Allievi et al., 2021; Palantir, 2019).

ETW as it relates to this study is heavily researched for the detection of malicious activities (Ahmed et al., 2021; Alshehri et al., n.d.; Bode & Warnars, n.d.). Researchers used ETW to detect the six most popular DLL injection techniques in strains of malware by consuming events related to memory. (Alshehri et al., n.d.) This outperformed other open-source memory analysis tools that scan adjacent process memory spaces. Another pair of researchers were able to use ETW to detect fileless malicious .NET C2 agents (Bode & Warnars, n.d.). Since .NET operates as a provider for events about usage on the system, it was possible to consume these events and gain insight into what was happening.

None of the previous research utilizes the execution of system calls as a detection mechanism for malicious activity. Although ultimately not necessary for the scope of their detections, they would be ineffective against malware using manual system calls. This is because the ETW provider that contains the necessary events, *Microsoft-Windows-Threat-Intelligence*, is restricted to normal UM applications (H, 2020). This provider is a primary source of telemetry Windows Advanced Threat Protection (WATP) and any approved Early Launch Antimalware (ELAM) Kernel Driver.

An ELAM driver is a special kernel driver that has been submitted to Microsoft upon approval and is signed using a special signature (Microsoft, n.d.-b). The in-depth verification process requires membership in the Microsoft Virus Initiative (MVI) to ensure the legitimacy of the company behind the submission. Afterward, the code for the driver is evaluated for security vulnerabilities by Windows Hardware Quality Lab (WHQL). The WHQL is then responsible for digital-signing the driver package, ultimately allowing it to run on the Windows OS (WHQL, n.d.).

This signing process limits the number of applications that can effectively monitor malicious activity on the OS when compared to UM hooks. Any application running with administrative-level access could utilize UM hooks to gather telemetry (Microsoft, 2002). This requirement creates a gap where there is no effective way to detect the use of manual system calls entirely from UM.

Due to its wide adoption by EDR solutions, ETW has been a large focus of security research (Chester, 2020; Palantir, 2019; Teodorescu et al., n.d.). One researcher discovered that the reporting mechanism for ETW events about NTDLL occurred from within UM. Additionally, the events were issued from the same process that was generating the events. By patching over the function *EtwEventWrite,* Chester (2020) was able to effectively mute the ETW events from being created. This is highly reminiscent of the unhooking bypasses for UM hooks, which took advantage of the fact that the code performing the hooking was inside the target binary's memory space.

Not only is it possible to overwrite *EtwEventWrite*, but it's also possible to falsify its contents (de Plaa, 2020/2020). A piece of malware can intercept the function's arguments and alter them before forwarding them to the original call. Categories such as *EVENT_DESCRIPTOR* or *EVENT_DATA_DESCRIPTOR* can be modified to change the name of the application being loaded completely.

These attacks display a clear weakness when an application can have direct access to its hooks. The ideal solution is to hook applications from within KM, as discussed however, due to Microsoft restrictions this is no longer possible. Receiving telemetry from *Microsoft-Windows-Threat-Intelligence* via ETW could help detect the actions performed. However, it

requires Microsoft's signature to function. Even still it would be unable to determine the method of invocation for a system call which is essential to determine if it is manual or not.

Process Instrumentation Callback

Ionescu (2016) introduced the concept of Process Instrumentation Callback (PIC) in a presentation called Hooking Nirvana: Stealthy Instrumentation Hooks. In this presentation, he illustrated how Microsoft has internal tools that interact with Windows that are yet undiscovered. This is far from a new concept, as covered previously in this paper, Microsoft often leaves features undocumented to preserve its authority to alter features between updates (Allievi et al., 2021).

One of the technologies unveiled is the Nirvana Runtime Engine (NRE). NRE is defined in the talk as a native monitoring framework that can be used to control the execution of a UM process without access to its source code (Ionescu, 2016). This internal capability was alluded to by Microsoft in a paper originally published in 2006, but it's utilization method remains unknown. (Bhansali et al., 2006). In Ionescu's research, it was discovered that Microsoft accidentally leaked one of the components involved in NRE through the Windows 7 Software Development Kit (SDK). The leak revealed that NRE was able to monitor running processes by using a dynamic UM callback.

A UM callback occurs whenever the Windows Kernel must interact with UM data to complete an operation (Mandt, 2011; Valasek, 2010). NTDLL supports a number of these callbacks by exporting functions to be called after the Windows Kernel completes the transition back to UM (Mandt, 2011; Ullrich, 2021). Common examples include:

Table 2. NTDLL User-Mode Callbacks (Everdox, 2013)

Name	Purpose	Location
LdrInitializeThunk	Thread and initial process thread creation starting point	NTDLL
KiUserExceptionDispatcher	Kernel exception dispatcher will return here if a process has no debug port or if the debugger chose not to handle the exception	NTDLL
KiRaiseUserExceptionDispatcher	When an exception occurs in a system service that can be handled by a user exception chain	NTDLL
KiUserCallbackDispatcher	Win32K and thread-based operations	NTDLL
KiUserAPCDispatcher	Queued APCs dispatched from here	NTDLL

To keep track of a process's execution, the Windows Kernel utilizes two structures: EPROCESS and KPROCESS (Catlin et al., 2017a; Chappel, n.d.). EPROCESS contains a list of pointers to data structures related to its execution like Tokens and the Process Execution Block (PEB). KPROCESS stores the information related to KM execution for a process such as process flags and execution time. Every time the Windows Kernel returns from UM it checks the KPROCESS of the current process for a field called *InstrumentationCallback* (Everdox, 2013; Ionescu, 2016). The default value for this field is NULL, which allows execution to continue unaffected. This is the missing piece that was revealed by the Windows 7 SDK.

To register the callback, a special structure is required. On Windows 10 the structure is defined as seen in Figure 5.

```
Typedef struct
_PROCESS_INSTRUMENTATION_CALLBACK_INFORMATION
{
	ULONG Version;
	ULONG Reserved;
	PVOID Callback;
} PROCESS_INSTRUMENTATION_CALLBACK_INFORMATION;
```

Figure 5. Process Instrumentation Callback Information Structure (Ionescu, 2016)

The Version fields define what architecture OS it is being run on, 0 for x64 and 1 for x86 (Ionescu, 2016). Reserved is always set to 0. The Callback field accepts a pointer to a location in memory. This is where the execution flow of an instrumented application is set when a UM callback occurs (Everdox, 2013). It is important to note that this is triggered every time the processor returns to UM from KM, not only when the functions in Table 2 are called. The R10 register will contain the return address that the execution originally would have gone to if it had not been diverted (Everdox, 2013).

The KPROCESS structure is a KM component, which makes it difficult to access from UM. As discussed earlier, interactions with the Windows Kernel are normally performed using NTAPI (Allievi et al., 2021). Ionescu (2016) identified it is possible to use the NT function *NtSetInformationProcess* to register the callback within the KPROCESS structure of a target process. This was the final piece to the puzzle, execution flow from a target application could now be diverted to a predetermined memory address.

Although originally named Nirvana Hooks by Ionescu (2016), this study henceforth refers to the technology as Process Instrumentation Callback (PIC). This helps to provide a

logical separation between Microsoft's internal tool "Nirvana Engine" and the technique of using the *ProcessIntrumentationCallback* structure to monitor a program's execution (Bhansali et al., 2006).

There is a limited amount of research on the benefits of using PIC, partially due to its undocumented nature. Two open-source projects were surveyed: Syscall-Detect and ScyllaHide.

Syscall-Detect.

Syscall-Detect is a proof-of-concept (POC) tool that uses PIC to detect the use of manual system calls from UM (Ullrich, 2021). It is accompanied by a blog post of the same name that details the steps the author took to create the project. The same basic details about the origin of PIC and the *ProcessInstrumentationCallback* structure are defined. This includes the usage of *NtSetProcessInformation* to register the *InstrumentationCallback* field within the KPROCESS structure.

The POC included two files: callback.cpp and Thunk.asm. The assembly (ASM) program contains a single function, *InstrumentationCallbackThunk*. It is responsible for preserving the register state between calls, preventing unintended recursion, and maintaining proper stack alignment (Ullrich, 2021). PIC does not provide these capabilities and requires any detour function implemented to be responsible for preserving execution flow (Ullrich, 2021). The author explained the lack of inline assembly support in the 64-bit MSVC compiler as the primary motivation for writing this as a separate file. Once the execution state is preserved, a local function defined in callback.cpp is called. *InstrumentationCallback* is responsible for the analysis portion of PIC. The second was a C++ program that oversaw

defining the *ProcessInstrumentationCallback* structure and passing it as a parameter to the *NtSetProcessInformation* function. The Callback parameter of *ProcessIntrumentationCallback* is declared as a pointer to the location of *InstrumentationCallBackThunk*.

By linking these two files at compilation time, they are compiled as a single DLL file. Similar to UM hooking, *syscall-detect* is intended to be loaded into a target process at runtime (Ullrich, 2021). This prevents issues previously discussed with the enormous number of system calls performed by the operating system. It also ensures that the execution flow never leaves the memory space of the calling process.

Once execution is returned from *InstrumentationCallBackThunk*, the registers are immediately inspected (Ullrich, 2021). By inspecting the R10 registers for the return address of the instrumented process, the reason for the UM callback can be determined. (Everdox, 2013). This is essential to avoid accidentally performing analysis on any of the functions covered in Figure 2. This project highlighted this as a potential future research opportunity but did not implement it.

Instead, the authors sought to validate that the address exists in a known module, in this case, NTDLL (Ullrich, 2021). Validating the integrity of the address can be used to detect the use of manual system calls since they are never legitimately executed outside of NTDLL. The POC was tested on a public tool that utilizes manual system calls to dump the contents of the LSASS.exe process and was successful in its detection (Ullrich, 2021).

ScyllaHide.

PIC was created by Microsoft to assist with debugging UM programs. Therefore, it's no surprise a debugger took advantage of its capabilities to enhance its analysis. ScyllaHide defines itself as an advanced open-source anti-anti-debug library, hooking various functions in UM to hide debugging from a program (x64dbg, n.d.-a). Some software will compile itself with anti-debug capabilities to avoid being analyzed by debuggers. Various actions like checking the PEB for the *BeingDebugged* value can help malware variants determine if it is safe to execute as well.

ScyllaHide performs the same steps as *syscall-detect* to set up and instrument the target process. The only change is rather than checking if R10 returns to NTDLL, the base address of the process currently being debugged is checked for (x64dbg, n.d.-a). This produces the same outcome since a system call should never originate outside of NTDLL. Ultimately, the project uses ScyllaHide to provide additional telemetry if a debugged process is attempting to execute manual system calls to bypass analysis.

Summary

Chapter 2 gave a broad overview of Windows Architecture and how it operates. This was necessary to deliver the proper understanding of how system services are utilized by UM applications. Separate ways to monitor the execution of UM system calls were surveyed, giving a historical perspective on the current state of research. This chapter also discussed how UM hooks can be bypassed and demonstrated various themes when detection capabilities rely on a sensor in UM. Manual system calls detection methods and how they can be utilized

by EDR solutions were also covered. Finally, two PIC projects were surveyed and provided context for this study.

Chapter 3 describes the methodology and why it was chosen for this study. It also describes the guiding research questions and hypotheses that this research answers. The topics covered in Chapter 2 are related to this study's primary goals and help frame the foundation for the creation of a model.

CHAPTER 3

RESEARCH METHODOLOGY

Chapter 2 provided background on the literature that applies to this study. The foundational concepts were explored to give background and provide the current state of the research. Chapter 3 details the methodology and design plan for this research, explaining the reasoning behind choosing the methods. Components include the population of the study, collection methods, instrumentation of variables, and data analysis.

Research Method and Design Appropriateness

To best conduct research, a proper methodology must first be identified. Three approaches are available for the analysis of a study's results: qualitative, quantitative, and mixed methods. These should not be viewed as rigid categories, but rather a sliding scale that tends to lean towards one way or another (Creswell, 2015). Research that focuses on numerical results and data tends to focus on a quantitative approach. Studies that involve varied responses and open-ended questions from individuals favor a qualitative approach. Both styles offer valuable tools to organize and plan things out, which is often the reason a mixed-methods approach is taken. By integrating the two forms, insight can be gained that otherwise may be lost by focusing on any single methodology (Creswell & Creswell, 2018). Since this study analyzes numerical data about the performance impact of PIC, a quantitative approach was chosen.

Conducting a quantitative study requires the proper identification of variables: independent and dependent. An independent variable is synonymous with the treatment that is applied. Its effect on the variables around it is what provides the data that is studied to measure results. Dependent variables represent the outcome or results of the influence of the independent variables. Creswell (2018) recommends having multiple dependent variables to measure in a study. The relationship between these variables and how it is measured provides the outcome of a study. By being able to quantify the variation in a phenomenon, situation, problem, or issue; hypotheses and research questions can then be answered (Kumar, 2014).

After selecting a proper research methodology, the next step is to choose the correct research design. These are types of inquiry that provide specific direction for procedures in a research study (Creswell & Creswell, 2018). The two main types of design are survey and experimental. A population's feelings or attitude is best studied using a survey design. It allows for the generalization of a large number of responses to numerically represent results. Experimental design splits the focus of a study into a sample and a control group. The sample group has applied treatment and the effects are compared to the control group. Since this study examined the impact PIC has on performance, a comparison between two groups based on numerical results was required. Therefore, an experimental design was chosen.

There are five steps involved in a classic experimental design according to Mauldin (2020). The steps include Sampling, Assignment, Pretest, Intervention, and Posttest. The first step involves the collection of what you would like to study known as Sampling. There are guidelines for choosing accurate samples that are covered later in this chapter. Assignment is the selection of which group each sample should be placed within. This is often a crucial step in classic experimental design, requiring randomization to maintain free of posterior

influences. As defined previously in the chapter, a dependent variable is a measurement of what changes throughout a study. In the pretest, instrumented dependent variables create a baseline measurement. Next, the intervention, otherwise known as a treatment, is applied to the experimental group only. This can happen multiple times or just once. The last step is an analysis of both groups to observe any changes in the dependent variables (Mauldin, 2020).

Creswell (2018) defines two subtypes of experimental research that can be used: true experimental and quasi-experimental. A true experimental follows the basic steps illustrated above with a key distinction. In the Assignment step, the design randomizes the population sample between the sample and control group before a treatment is applied. This is useful when comparing the results of the two groups because any differences between them can be attributed to random chance (Mauldin, 2020).

Quasi-experimental designs however are the exact opposite, using no randomization at all. These experiments are favorable when randomization is not possible or irrelevant to the outcome of a study (Harris, 2006). The experimental group is still given the treatment while the control group is not given the treatment. Quasi-experimental studies are often referred to as a before-and-after since results are recorded pretest and posttest. Before and after studies do not require the retroactive establishment of a 'before' observation, allowing researchers to construct it before the application of treatment (Kumar, 2014). This allows for the collection of results to answer research questions. To ascertain if the implementation of PIC has any performance impact, measurements are taken before it is applied and after. For the reasons explained above, the quasi-experimental before-and-after study was chosen for this research.

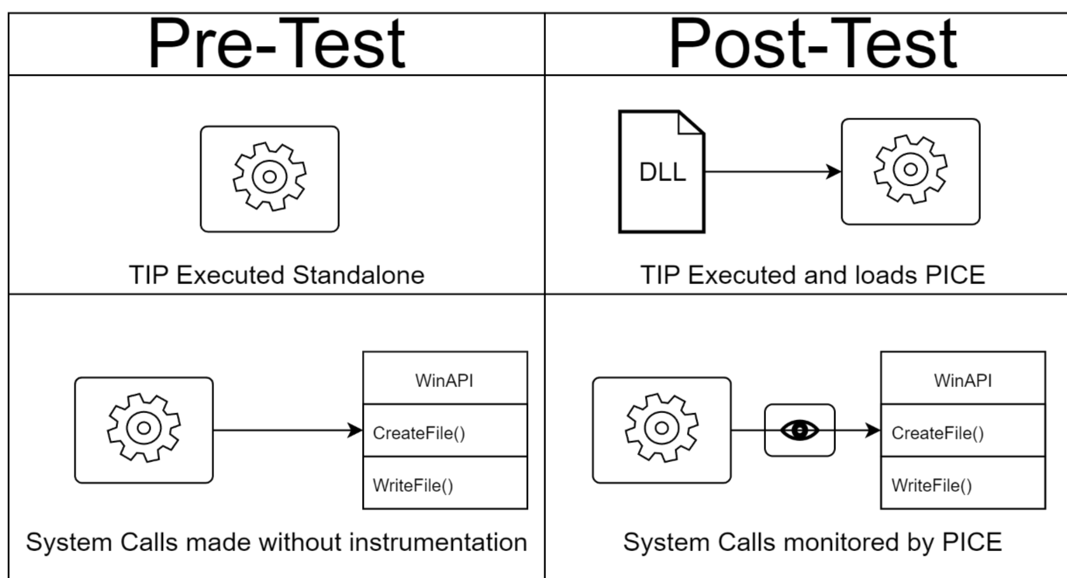


Figure 6. Quasi-experimental diagram showing pre-test and post-test in the study's context

Research Questions, Hypothesis, and Variables

To conduct this study properly, a guiding research question needs to be established. The primary research question for this study was: *What are the performance impacts of monitoring execution using Process Instrumentation Callback to detect the use of manual system calls with regards to CPU time, memory utilization, and clock time?* This question was answered by creating a model that contains a program monitored using PIC and one that does not.

The secondary research question employed in this study was: *Can we perform additional analysis using PIC to detect manual system calls and what is their effect on performance?* The underlying assumption may be that testing the bare-bones PIC is adequate for the primary research question, but the literature analysis demonstrated the innovation and extension of such methods for refinement of detection. Presentation of possible such

extensions demonstrate that PIC remains viable, thereby showing a certain resilience to such improvements in terms of usefulness.

The hypothesis for this study was: *Detecting the use of manual system calls utilizing Process Instrumentation callback will affect the performance of the system.* To determine the accuracy of this question the CPU time in both UM and KM were measured, along with the memory utilization and clock time.

Population

Mauldin (2020) defines an experiment as a method of data collection to test hypotheses under controlled conditions. During the normal execution of a computer, there may be changes that the user is unaware of that cause variability in an experiment. This study utilizes virtualization technology to prevent this from happening. Virtualization technology is used to split a physical server into isolated execution environments known as Virtual Machines (VM) (Zhang, 2018). Each runs independently from the other and can be used to run their isolated operating system and programs. This requires that physical components such as memory, disk space, and CPU time are shared between them. Modern systems can run multiple of these VMs without noticeable performance impact, providing realistic execution environments to simulate scenarios (Hirofuchi, 2018; Zhang, 2018). The ability to be able to run multiple versions of the same environment allows for maximum control over the fidelity between tests.

The management of hardware utilized by VMs was provided via a hypervisor. A hypervisor or Virtual Machine Manager (VMM) runs on top of hardware resources to allocate them to the appropriate VM as needed (Zhang, 2018). This study used the enterprise-class

hypervisor ESXi to service in this capacity. ESXi is a bare-metal hypervisor which means it installs directly onto a physical server, as opposed to running on top of an existing operating system.

This study utilized a single Windows 10 20H2 64-bit operating system for observation. Since 2020, newer versions of Windows have been released twice a year, naming the version after which half of the year it was released in (Hoffman, 2020). For example, 20H2 was released in October 2021. Future releases of the operating system were ignored to provide consistency and control for the study. The VM operated using an i5 2.20GHz CPU and 8GB Ram. These are consistent with the Microsoft recommended hardware requirements for optimal system usage (Microsoft, n.d.-c). The installed software on the computer was limited to what comes pre-installed and what is necessary to execute the study. Boilerplate libraries such as VMware Tools and VC++ Libraries were installed to create the environment.

The Process Instrumentation Callback Engine (PICE) was installed and compiled using the 64-bit Microsoft Visual C++ Compiler that comes installed with Visual Studio 2019. PICE featured the same basic structure as Ullrich's original proof-of-concept (Ullrich, 2021).

- *NtSetInformationProcess* to set the *InstrumentationCallback* field in KPROCESS.
- Instrumentation Hook
    - *InstrumentationCallbackStub* written in assembly to preserve registers.
    - *InstrumentationCallback* C++ function to perform main analysis.

As covered previously in Chapter 2, Ullrich (2021) determined the validity of a system call by performing bounds check on the return address stored in R10. If the address resided in the currently known location of NTDLL, it was allowed to execute. If not, it was assumed to

be a manual system call and was terminated. This was vulnerable to manipulation if an attacker overwrote a benign system call in NTDLL and made the call from that location (Ullrich, 2021).

PICE overcomes this vulnerability by examining the instructions immediately preceding the address stored in R10. Since R10 always points to the *ret* instruction that came after instrumentation, it is possible to peek at the previous instructions by walking backward. Windows x64 calling convention is static and provides a reliable expectation of what instructions should be found (Hand, 2020; Microsoft, n.d.-g). By cross-referencing these instructions with a structure that contains an expected location for every system call prologue, it is possible to determine if a legitimate function has been patched.

As previously discussed in Chapter 2, SysWhispers2 is an open-source project that automatically discovers and extracts the SCNs from Windows and generates header/ASM files that can be used to make manual system calls (T, 2021/2022, p. 2). A modified version of SysWhispers2's extraction technique was utilized by PICE to generate a structure of system call locations and their SCNs. The modified technique used is as follows:

1. Parse the export address table (EAT) of NTDLL for all functions names that begin with “Zw”.
2. Store them in a custom structure for later retrieval.
3. Sort the system call entries by memory address in ascending order.
4. The system call number (SCN) is the index of the table.
5. The memory address used for sorting is the function prologue location.

This structure served as the ground truth for what SCN should be discovered when the address stored in the R10 register is analyzed.

An equally important part of this study is the target instrumentation program (TIP). TIP is a representation of a standard program running on the Windows that is monitored by PICE. It was compiled with 64-bit MSVC that comes with Visual Studio 2019 and installed on the system. Once executed, TIP imported a DLL version of PICE into its memory space and mapped it for execution. WinAPI functions were executed to trigger PIC and begin analysis.

Sample Collection

The process of identifying a population and selecting a subset to conclude from is known as sampling (Mauldin, 2020). A population represents what is being studied in a research experiment. Often, it is not possible or even necessary to gather data about a population in its entirety. Sampling frames are used to fill that gap by further clarifying what in a population is relevant and allowing a sample to be chosen from there (Mauldin, 2020). By studying a sample, a researcher can estimate what is likely to be the situation in the total population (Kumar, 2019).

The objective of this study is to observe the impact of monitoring for manual system calls via PIC. Since manual system calls are not performed by legitimate applications, their existence is a powerful heuristic . If manual system calls are detected, the process should be immediately terminated. By nature, this requires the hooking of all system calls executed by a process to perform analysis and determine the legitimacy of the call. If the process was terminated, it would have been impossible to capture measurements in a test. Therefore, TIP does not execute manual system calls to provide the largest number of non-manual system calls' to calculate overhead. This paper defines 'non-manual system calls' as the native chain

of execution as defined in Figure 1. By determining the overhead of monitoring for their execution, the cost of implementing PIC as a solution can be measured.

It is unreasonable to monitor the execution of every system call on the Windows 10 operating system. It is an extremely complex piece of closed-source software that contains many undocumented features that cannot be accounted for in a model (Schulman et al., 1992). To frame the problem appropriately, the system calls executed by TIP are based on previous studies surveyed in Chapter 2 that examine the performance of API hooks on Windows (Lopez et al., 2017; Marhusin et al., 2008a).

Instrumentation

The objective of this study was to measure the performance impact of detecting manual system calls with PIC through several different dependent variables modeled after previous research on the topic (Lopez et al., 2017; Marhusin et al., 2008b). Memory utilization was measured by collecting the physical and virtual memory sizes before and after the intervention. Similarly, the amount of time the CPU spends in UM and KM was recorded. The amount of clock time from start to finish was also collected, providing an overall assessment of the time spent running.

The WinAPI provides multiple functions to measure the performance of programs running on the operating system. These native functions were used to record the variables in this experiment by executing them within TIP both at the beginning and at the end. The following functions were used:

- *GetProcessMemoryInfo*
    - Records memory utilizations.

- *GetProcesstimes*
    - Records UM and KM execution time.
- *QueryPerformanceCounter*
    - Records wall-clock or real execution time.

Many previous studies have employed these tools and they are widely utilized by Windows developers to analyze performance (Botor & Habiballa, 2018; Jia et al., 2014; Lopez et al., 2017).

Validity and Reliability

The validity of an experimental study is critical to its value. Researchers need to identify threats to their design and minimize that as soon as possible (Creswell & Creswell, 2018). There are two main threats to validity: internal and external (Creswell & Creswell, 2018). Internal threats include things like the passage of time and cross-contamination between the control and experimental groups. External threats are more often an issue with being able to generalize the results from a sample and apply them to a population.

To mitigate the risk of internal threats this study employed virtualization technology. A snapshot of a VM was taken that freezes all of the virtual hardware components such as memory, CPU state, and hard disks (VMware, n.d.; Zhang, 2018). Creating a snapshot at the beginning of the experiment and then reverting to that after each test eliminated any risk posed by the passage of time. This limited the number of uncontrolled variables, providing fidelity measurements.

To ensure that cross-contamination between the two groups did not occur, PICE was installed on Windows 10 after snapshot initialization. By ensuring PICE was never present on

the system when baseline measurements were taken, it can be assured that it had no outcome on the results.

Ensuring the reliability of results is extremely important in research. The quality of a measurement is determined by its repeatability and accuracy (Kumar, 2014). This study accomplished this by repeating the experiment 30 times under the exact same conditions for each iteration. Like validity, virtualization technology was implemented to maintain a controlled environment between tests. Repetition ensured that in the chance that any anomalous results were produced, they could be mitigated by averaging the tests.

Assumptions

This study made several assumptions about the tools and environment that were measured. The first assumption was that between experiments, the context surrounding the measured variables was reset completely. This means that the operating system can revert to a point in time before the introduction of treatment. This ensured that the study was conducted in a controlled environment with limited outside influences. The second assumption was that the use of WinAPI functions to measure the variables of interest, such as the time of execution, remained accurate despite PIC's implementation. Lastly, the internal mechanisms of how Windows handles system calls and the assigned numerical values was assumed to remain static between tests. By using the same instance of the Windows operating system and disallowing upgrades or internet access, these assumptions were assured.

Scope and Limitations

It's important to emphasize that PIC is not the sole detection method for manual system call use (Teodorescu et al., n.d.). Other methods like ETW can monitor their usage if the consumer application has gone through the Microsoft verification process. Chapter 2 covered the limitations of ETW as they exist today, including their reliability on UM functions for NTDLL telemetry (de Plaa, 2020/2020). PIC is unique because it can monitor system calls from UM while relying on the Windows Kernel to hook the target process.

Individual system calls execution time was not measured since it is rare for them to operate independently. To create a realistic model, multiple system calls were called in succession to synthesize the processing workload. These system calls were purposefully chosen not to interfere with the recording of variables and avoid situations that would cause the OS to terminate the program.

Ideally, PIC would be configured to only inspect the execution of a subset of system calls. However, PIC does not allow individual UM callbacks to be filtered, therefore the detour function was executed for every callback. Normal UM hooks do not have this problem since they can statically alter function prologues of interest located in NTDLL. PICE must perform extra analysis to quickly determine if an intercepted UM callback should be analyzed. This limitation is unavoidable.

The focus of this study is the detection of the manual execution of system calls. Therefore, which system call TIP executes is irrelevant. As long as it follows the standard Windows calling convention the results of the experiment can be abstracted to apply to any NTAPI function. The fact that these calls were made without following the proper NTAPI call chain, as covered in Chapter 2, was the driving indicator for an alert.

Summary

The purpose of this research is to analyze the performance impact of detecting manual system calls utilizing PIC. This chapter described the overall methodology to be employed in this study and why it was the most appropriate choice. The various components were explained in detail and their relationship with the objective was established. The scope of this research was clarified and limitations were defined. The next chapter discusses data collection and analysis as well as the quantitative results from the experiment. Statistical analysis is performed to highlight the significance of the results.

CHAPTER 4

RESULTS

This quasi-experimental before-and-after study aimed to determine the relationship between system performance and the detection of manual system calls using PIC. This chapter covers the methods used to capture and analyze the metrics presented in this experiment. As stated in Chapter 3, these performance indicators included CPU and Memory utilization by the instrumented process. CPU utilization was further broken down into time spent in User-Mode and Kernel-Mode. Enhancements to the fidelity of PIC's detection capabilities were observed in three separate instances to illustrate the statistical significance of additional time spent analyzing system calls for detecting manual invocation. PICEv1 used a combination of bounds checking of the UM return address and public symbol files to determine if it was located within the region of memory allocated for NTDLL. PICEv2 demonstrated the impact of removing the reliance on public symbols and performing additional anti-tamper analysis on the loaded image of NTDLL itself. PICEv3 added the analysis of the returned stack pointer and performed bounds checking of the WinAPI DLLs.

Chapter 4 presents the data collected to answer the following research question: What are the performance impacts of monitoring execution using Process Instrumentation Callback to detect the use of manual system call with regards to CPU time, memory utilization, and clock time?

Data Collection

To conduct the experiment a virtualized environment hosted on a bare-metal hypervisor was constructed as described in Chapter 3. The environment contained a single virtual machine running a clean installation of Windows 10 20H2 Testing was conducted in an isolated environment with zero network access or third-party applications. Process Instrumentation Callback Engine (PICE) and Target Instrumentation Process (TIP) were loaded onto the machine using a virtual-disk image. An updated version of Microsoft Visual C++ 2015 Redistributable was installed for PICE to function. A virtual snapshot of the operating system was created and was reverted to between tests. A snapshot of a VM freezes all of the virtual hardware components such as memory, CPU state, and hard disks (VMware, n.d.; Zhang, 2018). By implementing the aforementioned safeguards, the number of uncontrolled variables was limited, such as the passage of time, providing fidelity when measurements are taken.

The objective of this study is the observation of the performance of the Windows operating system during instrumentation by PIC for the detection of manual system calls. Two primary components described in Chapter 3 were developed: Target Instrumentation Process (TIP.exe) and Process Instrumentation Callback Engine (PICE.dll). TIP was implemented as a binary that imported PICE.dll and subsequently makes a large amount of legitimate system calls using the Windows API (WinAPI). The calls made were modeled after previous research into system call performance that suggested the following (Lopez et al., 2017):

Table 3. Actions performed by TIP and the corresponding system calls

Action	*WinAPI (NtAPI)*
Create 10,000 text files	CreateFile (NtCreateFile), CloseHandle (NtClose)
Open each file, write "Hello World!" and close the file	CreateFile (NtCreateFile), WriteFile (NtCreateFile), CloseHandle (NtClose)
Open each file, read the contents into a buffer, and close the file	CreateFile (NtCreateFile), ReadFile (NtCreateFile), CloseHandle (NtClose)
Delete all 10,000 files	DeleteFileW (NtCreateFile)

Before the execution of these operations, a baseline of system performance was recorded. These statistics were also recorded when execution was completed, providing two sets of results. The variables measured to measure impact were: CPU user time, CPU kernel-time, physical memory size, virtual memory size, and clock time (Lopez et al., 2017). Collectively, these measurements represent the overall performance of the system. Each variable was collected using the corresponding WinAPI functions as described in Chapter 3. The functions used by TIP during execution included GetProcessTime, GetProcessMemoryInfo, and QueryPerformanceCounter. Once execution completed, the results were outputted to screen and recorded by the researcher. The operating system was then reverted to its snapshot to preserve any independent variables and the experiment was repeated.

PICE Detection and Analysis Methods

Four different tests were run that encompassed different iterations of PICE. Four different test category iterations were used, including the benchmark and three versions of PICE (PICEv1, PICEv2, PICEv3). Each iteration was executed a total of 30 times to create an ideal statistical population (Lopez et al., 2017). In every experiment, an identical version of TIP.exe was executed that would import the corresponding PICE.dll. These different iterations were created to properly answer the research question: *Can we perform additional analysis using PIC to detect manual system calls and what is their effect on performance?* Each contained an improvement that increased the amount of checks performed by PICE to ensure that a system call was made legitimately. Each of these checks included the previous iteration's changes as well, building on top of one another. A few exceptions to this included coding decisions made specifically for performance benefits such as the removal of the Microsoft Debugging Engine in PICEv2 and PICEv3. See Appendix D for the source code for each iteration of PICE and TIP.

Version 1

As reviewed in Chapter 2, previous researchers have used PIC to successfully detect the use of manual system calls on Windows 10 (Ionescu, 2016; Noah, 2017/2022; Richard, n.d.; Ullrich, 2021). These implementations functioned as proof of the concepts for the technique and were not necessarily concerned with optimization or resilience. PICEv1 represented these previous works, featuring much of the author's original codebase with minimal alterations. It was important for the secondary research question to be able to

quantify the performance of previous works to properly observe the differences made in future versions.

Manual system call usage was detected by first determining the current base address of NTDLL in the current process and the size of the image. By cross-referencing the return address (R10) that is instrumented by PIC, PICEv1 could determine if the location resides within NTDLL. Since all NTAPI function calls should return to NTDLL after performing a system call, any detraction from this was assumed to be malicious.

Microsoft Debugging Engine (MDE) was used to resolve the return address location to the corresponding function name using publicly exported symbols. MDE is implemented in a dynamic link library named DBGHELP.DLL that is installed by default on Windows 10. The symbols are exported by Microsoft for every version of NTDLL and are expected to be consistent in future versions of Windows. If PICEv1 was unable to resolve the function to a symbol, then the call was also considered malicious. See Figure 7 for a flowchart that illustrates the basic layout of PICEv1.

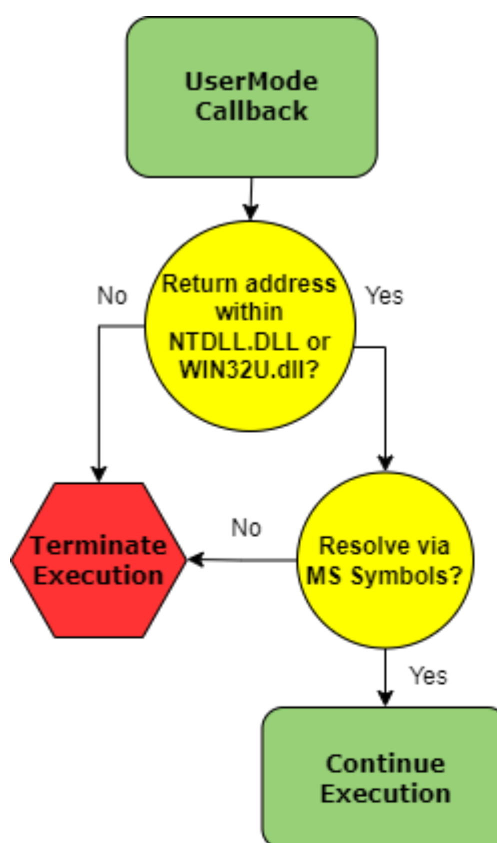


Figure 7. Diagram of PICEv1 Analysis

Version 2

Previous authors have theorized bypassing the bounds checking of NTDLL by modifying a seldom-used NTAPI function's SCN. It is not uncommon for malicious binaries to modify NTDLL to avoid detection (Apostolopoulos et al., 2021; MDSec, 2020; T, 2021/2022) For example, NtCreateFile has an SCN of 0x55 and could replace the index in another function and invoke it normally. When PICEv1 performed its inspection, it would consider it legitimate due to the return address resolving within NTDLL.

PICEv2 checked for the modification of NTDLL by comparing the SCN that was executed against a ground truth of function addresses and their corresponding SCNs. This

ground truth was a custom structure (SysCallList) that was populated every time a system call was intercepted. The technique described in Chapter 3 to dynamically retrieve SCNs from NTDLL was implemented and modified to populate SysCallList with the location of the function prologue instead of the function name. See Figure 8 for a visualization of the structure.

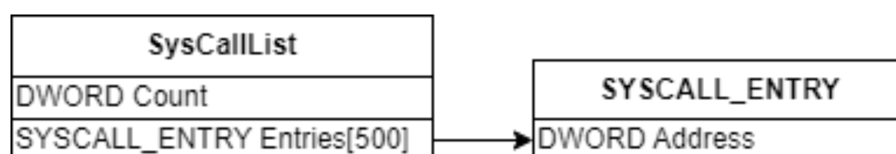


Figure 8. SysCallList Structure Layout

As an added benefit, MDE was no longer imported since SysCallList acted as a dynamically generated replacement to resolve addresses to functions. This eliminated having to import DBGHELP.DLL into the instrumented process which is a well-known indicator that malware often checks for (Czeczko, 2020). At this point, the return address captured by PIC (R10/RIP) is inspected and compared for detection purposes.

This comparison was completed in four steps:

1. Find and store the location of the instrumented NTAPI function prologue (mov rcx, r10) by subtracting 0x14 from RIP.
2. Find and store the value moved into RAX before the *syscall* instruction by subtracting 0x10 from RIP. This should be the SCN that was executed.
3. Use the address found in step 1 to iterate through SysCallList until a match is found.
4. Retrieve the SCN that should be at that location and stop execution if they do not match.

PICEv2 can reliably instrument each NTAPI system call due to the reliability of the Windows calling convention (Figure 4). RIP will always point to the *ret* instruction which is

consistently 0x14 bytes away from the function prologue *mov rcx, r10*. The function prologue always matches the value in the exported address table because without it programs would be unable to locate them for use. The same storage and retrieval are possible by subtracting 0x10 from RIP to find the SCN. If it matches the SCN stored in SysCallList then PICEv2 can be positive that it is legitimate and has not been tampered with. PICEv2 is illustrated in Figure 9 which shows a flowchart of the decision-making process.

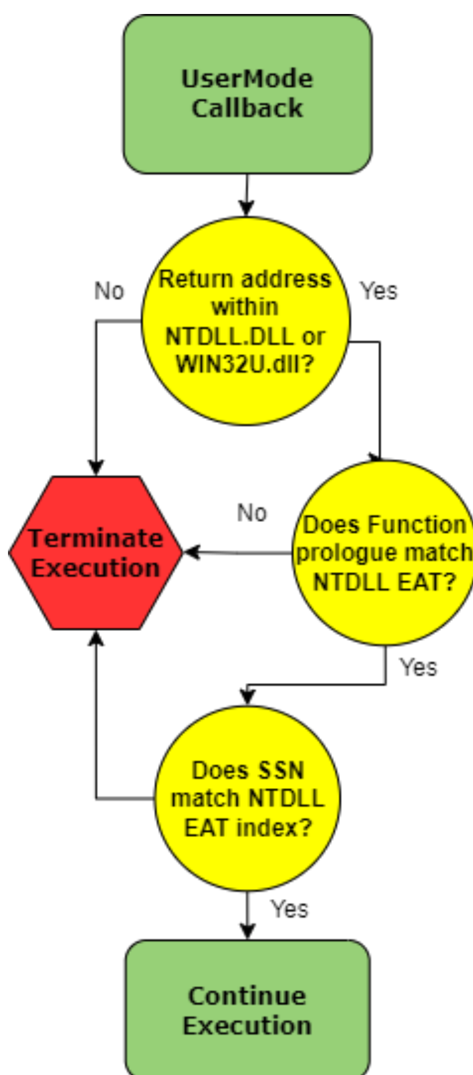


Figure 9. Diagram of PICEv2 Analysis

Version 3

PICEv3 altered the execution of PICEv2 in two major ways. The first is that the dynamic parsing of NTDLL's export table to populate SysCallList was performed when PICE.dll first attaches to the target process. SysCallList itself was declared as a global variable and can be shared between multiple system calls. This was expected to have a decrease in performance impact compared to PICEv2 which needed to repopulate the structure for every system call. The second addition to PICEv3 was the inspection of the stack

pointer returned by PIC. When the instrumented call returns to UM, RSP points to the stack frame that existed before executing the *syscall* instruction. This is important because after performing a system call the next instruction is always *ret*. This means that execution will jump to the memory address that is on top of the stack. Since RSP pointed to the top of the stack, PICEv3 could perform analysis of that memory address. A legitimate system call into NTDLL would originate from KERNELBASE.DLL, as seen in Figure 1. The same bounds checking performed on NTDLL was performed to make sure the address occurs within KERNELBASE.DLL. Two exceptions to this were identified which are internal calls from within NTDLL and lateral calls from WIN32U.DLL. Bound checking was performed on these as well to ensure compliance. A diagram illustrating the decision-making process of PICEv3 is shown in Figure 10.

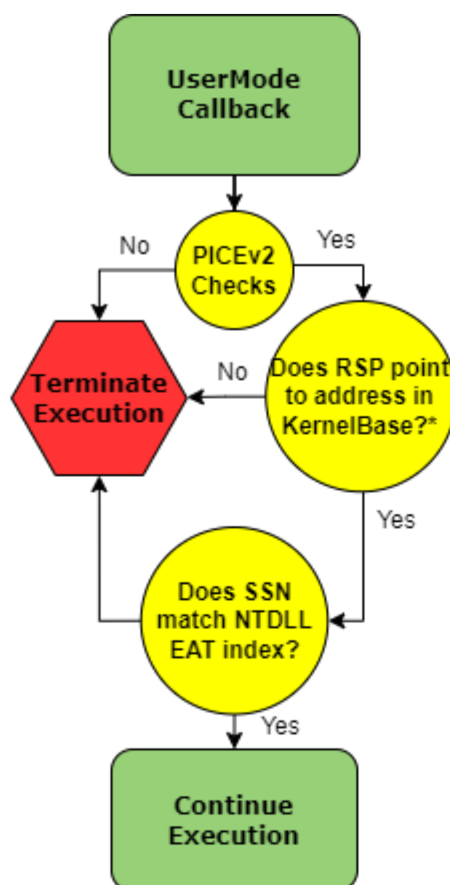


Figure 10. Diagram of PICEv3 Analysis

System Performance

After the completion of all tests, the data was imported into Microsoft Excel to perform additional analysis. Separate groups were created for each of the different versions of PICE and the baseline measurement. This section will discuss the degree of impact that each test had on both CPU and memory of the operating system.

User-Mode and Kernel-Mode Performance

CPU utilization was measured using the WinAPI function *GetProcessTimes*. *GetProcessTimes* provides the timing information for a specified process including the

amount of time that it has spent executing in user-mode and kernel-mode during the call. By executing this function before and after treatment it was possible to ascertain the accurate execution time while eliminating contamination from process startup and teardown. The amount of time in 100-nanosecond intervals was returned by the function and converted to microseconds by dividing by ten. The larger amount of time spent in either mode indicates a higher impact on the processor.

Table 4. Average User-Mode and Kernel-Mode execution time in microseconds

PICE Version	**User-Mode Time**	**Kernel-Mode Time**	**Combined**
Baseline	164,583.333	4,713,541.667	4,878,125.000
1	671,875.000	4,839,062.500	5,510,937.500
2	27,476,041.670	5,315,104.167	32,791,145.830
3	408,854.167	4,824,008.333	5,232,862.5

A significant increase in User-Mode Time was expected with the introduction of PICE since all analysis takes place outside of KM. Kernel-Mode Time experienced much smaller increases in processing time since the Windows kernel checks for the existence of PIC whether it is configured or not. PICEv2 had the highest analysis time, most likely due to the parsing of NTDLL's export address table (EAT) to populate SysCallList.

Clock Time CPU Performance

Processors often contain a 64-bit Time Stamp Counter (TSC) register that increases at a rate equal to the processor clock (Microsoft, 2022b). This value can be read via the RDTSC or RDTSCP machine instructions to provide a low-level access time and measurement of the computational cost of executing a program on a processor and is known as a performance

counter. The performance counter, or ticks, was measured using the WinAPI function *QueryPerformanceCounter*. This function provided the number of ticks that have occurred and can be converted to microseconds by multiplying by 1,000,000 (Microsoft, 2022b). This is referred to as wall time, otherwise known as clock time and is the amount of actual that has passed according to the processor itself. This represents the amount of time that an end user would see an application taking and is complex due to potential strain from concurrent activity happening on the system. Real time data was recorded at the end of execution and is represented in Table 5.

Table 5. Clock Time execution in microseconds

PICE Version	**Duration (Average)**	**Standard Deviation**
Baseline	10,288,090.400	488,989.409
1	10,528,254.900	280,275.243
2	38,195,913.870	455,914.696
3	10,439,245.800	366,688.800

Without instrumentation, TIP had an average execution time of 10,288,090.4 microseconds from beginning to end. This included the lowest clock time of 8,767,614 microseconds and the highest clock time of 12,021,302 microseconds. PICEv1 which performed NTDLL bounds-checking via the Microsoft Debugging Engine had an execution range from 10,052,854 at the lowest, to a high of 11,383,973 microseconds. PICEv2, which included additional dynamic SCN retrieval and verification had the highest execution time at 38,195,913.87 microseconds and a peak of 39,764,391 microseconds. Even at its lowest execution time of 37,424,581 microseconds, it was still triple the largest measurement from

any other version of PICE. Lastly, the additional verification of the return stack and global variable usage of PICEv3 brought its average execution down to 10,439,245.8 microseconds. Its range was 9,953,495 microseconds at the lowest and 11,761,360 microseconds at the highest.

Overall, clock time execution times were consistent with the User-Mode Time and Kernel-Mode Time measurements. PICEv2 had the greatest impact on performance compared to the other versions. PICEv3 had the smallest standard deviation at 366,688.800. This consistency can be attributed to the simplified method of analysis that it uses to resolve function addresses.

Memory Performance

Memory utilization was split into two categories: physical and virtual. Both can be measured using the WinAPI function *GetProcessMemoryInfo*. Unlike the other functions used for data collection, it does not require multiple invocations and can be called once at the end of execution. This is because it tracks the peak usage in bytes of both types of memory, only incrementing it when the previous value has been surpassed (Microsoft, 2022c). As covered in Chapter 2, virtual memory is an abstract representation of the physical memory on the system that is not realized until accessed. Virtual memory is a poor representation of system impact on its own . Memory performance data is averaged over the 30 tests and presented for each version of PICE in Table 6.

Table 6 . Average Virtual and Physical memory usage in bytes

PICE Version	*Virtual Memory (Bytes)*	*Physical Memory (Bytes)*
Baseline	522,103.5	2,500,198.4

Table 6 (continued).

1	5,830,109.87	8,260,403.2
2	560,742.4	2,614,886.4
3	555,144.5	2,609,561.6

Baseline execution of TIP without PICE had an average virtual memory usage between 516,096 bytes and 561,152. Physical memory usage ranged from 2,482,176 Bytes (B) to 2,789,376 B. PICEv1 has the largest impact on memory usage that ranged from 5,804,032 B to 6,316,032 B for virtual memory and 8,232,960 B to 8,953,856 B for physical memory. With the improvements in PICEv2, the average was brought back down to below one million for virtual memory utilization. This can be partially attributed to the lack of including the DBGHELP.DLL to resolve function names. Its memory usage ranged from 548,864 B to 610,304 B for virtual memory and 2,592,768 B to 2,891,776 B for physical memory. Lastly, PICEv3 averaged 544,768 B to 598,016 B for virtual memory and 2,580,480 B to 2,899,968 B for physical memory.

Overall, PICEv1 had the highest memory utilization increase when PICE is used. Though an initial increase of memory from the inclusion of PICE.DLL is to be expected, both PICEv2 and PICEv2 have much smaller usage due to their lack of reliance on MDE.

Statistical Analysis

Analysis Goals

The hypothesis of this study is: "Detecting the use of manual system calls utilizing Process Instrumentation Callback will affect the performance of the system." A null hypothesis is the opposite of the hypothesis in a study (Boslaugh, 2012). The null hypothesis used in this study is: Detecting the use of manual system calls utilizing Process Instrumentation Callback will not affect the performance of the system. To prove the validity of a hypothesis the null hypothesis must first be proven to be false. This can be accomplished by demonstrating the statistical significance of the data (Fralick et al., 2017). By determining the significance, it can be assured that the results were not a matter of chance and can be attributed to the application of the PIC treatment.

Survey of Statistical Methods

Choosing the correct statistical method is extremely important in the determination of the significance of research data (Fralick et al., 2017). Prior research into system calls did not report utilizing any methods other than the calculated average of test results (Lopez et al., 2017; Marhusin et al., 2008b). These studies focused on the comparison of alternative hooking approaches and therefore did not require intensive validation of the system impact of intercepting system calls. On the contrary, this study considered two test types to determine the significance of the results: Student's t-test (t-test) and analysis of variance (ANOVA).

Both tests wish to study the relationships between variables but the question they seek to answer are slightly different. A t-test is used to compare the means between two groups, whereas ANOVA is used to compare the means among three or more groups (Mishra et al.,

2019). As previously mentioned, this study created three different versions of PICE that were compared to a base installation of Windows 10. Initially, ANOVA would seem the most appropriate to consider there are four groups involved. However, each of these groups is only compared to a single group at a time, the pre-test and post-test values. This is because the goal of this analysis is to disprove the null hypothesis that there is zero effect on the performance of the Windows operating system. Therefore, a t-test was chosen to be the most appropriate method instead of ANOVA.

A t-test is a statistical technique used to test the mean difference between two groups for statistical significance (Mishra et al., 2019). It involves the use of an alternative hypothesis and a null hypothesis to determine the significance of two samples. The null hypothesis always states that the means are equal, whereas the alternative hypothesis states that both means are not statistically equal. It can be further broken into different types: independent samples, one-sample, and paired samples. The independent-sample (or two-sample) test is used when the population means, or standard deviation is unavailable. A common use case is to compare two independent population samples for a common dependent variable. When the mean is available for a studied population, the one-sample can be used to compare the mean to a static or hypothetical value of a population. This can be used to evaluate a sample against a larger dataset that has produced an expected mean. Lastly, a paired sample is used to compare the means of the same or related groups at different times. This is often used to analyze the effects of treatment on dependent samples of a single population (Fralick et al., 2017). Paired-sample was chosen as the most appropriate type of t-test since this study observes system performance before and after PIC treatment.

Evaluation of Statistical Results

The next step when preparing to perform analysis with a t-test is to determine the confidence interval (Boslaugh, 2012). This is the range of numbers that are acceptable when calculating the difference between the means of the two groups. A minimum confidence interval (p-value) of 95% is generally required by most publishing journals and is supported by previous research (Boslaugh, 2012; de Winter, n.d.; Mishra et al., 2019). Ensuring a minimum p-value of 0.05 (95% confidence interval) minimizes the risk of sample difference being the result of chance. This is required to statistically disprove the null hypothesis and show that the treatment influenced the population.

The t-test also calculates the degrees of freedom allowed which for this experiment was 29. The degrees of freedom are used to calculate the acceptable values that must be outputted from a t-test to demonstrate statistical significance (de Winter, n.d.). These output values are known as t-values. The result of this comparison creates the p-value for the study and determines whether the groups are statistically different or not. All comparisons used a one-tailed directional approach when analyzing the results. This is because the study is only intended to observe the dependent variables which showcase system impact.

Analysis was performed on the results of CPU utilization for both UM and KM a total of three times. Each test performed a t-test between the baseline version of TIP and the versions that were monitored using PICE. Since each group had the same number of samples the same degrees of freedom (29) were used to calculate a minimum t-value needed of 1.699127027. This means if the resulting t-value is greater, and the p-value is lower than 0.05 it can be assured to be statistically significant. The resulting t-value and p-values can be seen below in Table 7.

Table 7. T-test results for User-Mode and Kernel-Mode execution time

PICE Version	*UM t-value*	*UM p-value*	*KM t-value*	*KM p-value*
1	33.95363969	3.3391E-25	2.180466737	0.018740946
2	364.6892804	5.93804E-55	9.214008673	2.04361E-10
3	15.06430652	1.50397E-15	1.832424938	0.038587737

PICEv2 experienced the largest increase in processing time and likely can be attributed to having to populate SysCallList each time a system call is instrumented. PICEv3 had the shortest processing time of all versions in both UM and KM. Each iteration demonstrated a t-value well above the required minimum (1.699127027) and therefore successfully rejects the null hypothesis. Since the null hypothesis is disproven, it is not necessary to perform a t-test on the other dependent variables. However, to be thorough, a t-test was performed for the other measurements: clock time and memory utilization.

Real-time (wall) analysis produced similar results as CPU utilization and demonstrated an increase in both PICEv1 and PICEv2. PICEv2 had a t-value of 242.7146425 compared to PICEv1's t-value of 2.392751469, proving they are statistically significant. Alternatively, PICEv3 produced a t-value of 1.42555938 and a p-value of 0.082335591. Both numbers produced via PICEv3 are too low to be significant.

Memory utilization for virtual and physical memory usage was compared and showed statistically significant results. Increases in memory activity can be partially attributed to the inclusion of PICE.DLL in all tests. PICEv1 produced the highest t-values of 343.7473227 (virtual) and 412.734209 (physical), most likely due to its inclusion of MDE. T-test analysis

of PICEv2 and PICEv3 resulted in t-values of 21.50751942 and 18.25574897 respectively for virtual memory. Physical memory analysis of the duo produced t-values of 60.88478071 and 10.38907161. All t-tests ran for memory utilization produced statistically significant results.

Summary

This chapter discussed the results of the quasi-experimental before-and-after study. The layout of the model was described in detail and included the Windows 10 operating system and the use of virtual snapshots to preserve integrity between tests. Three different tests were performed using various iterations of PICE and a single test that used instrumentation to measure a baseline. The results were gathered into a Microsoft Excel document for further analysis.

Additional scrutinization of the return address of an instrumented system call was documented and described. PICE development resulted in three different versions, each containing increased fidelity that improved its detection capabilities. Additional analysis and source code are available in Appendix D.

To prove the hypothesis that there is an effect on system performance when PIC is used to monitor for manual system calls, the null hypothesis must be disproven. As others have shown this is most commonly done using by demonstrating the statistical significance of the data gathered by using some like a t-test (Boslaugh, 2012). CPU utilization for both UM and KM was analyzed using the t-test and the quantitative results were presented to prove that there is a performance impact. Additional t-tests were performed in this chapter on clock time and memory utilization, with their results presented.

Overall, each of the dependent variables measured showed a large increase when PICE was imported and began monitoring system call usage. The next chapter will conclude this paper and discuss the findings in additional detail. Recommendations for how PIC can augment existing detections and potential future research topics are covered as well.

CHAPTER 5

CONCLUSION

This chapter presents the implications of the quantitative results covered in Chapter 4. The limitations of this quasi-experimental study and the performance impact introduced by using PIC to monitor manual system call usage are discussed. Each version of PICE introduces its own detection mechanisms that can both help detection but also contain downsides that must be considered. Therefore, PICE iterations and their performance are compared with each other.

This discussion further includes how PIC may fit into existing defensive technology ecosystems and augment telemetry for UM activity on Windows. Recommendations for future research for the purpose of expanding the capabilities of PIC and optimizing analysis are made.

Review of Findings

Executing a system call on Windows manually is a well-known technique used to bypass defense technologies such as User-Mode Hooks (MDSec, 2020; Ullrich, 2021). Microsoft does not officially support the use of manual system calls, instead providing the WinAPI for programs to interface directly with the operating system (Allievi et al., 2021). These hooks commonly run in the WinAPI, as described in Chapter 2, and temporarily divert execution to perform an analysis of function arguments to determine if it is malicious or not.

The detection of these manual system calls is incredibly important since without it any monitoring software relying on UM hooks is blind to its execution. The use of this technique can be considered anomalous due to unlikely legitimate usage because of instability between versions of Windows.

Process Instrumentation Callback was discovered by Alex Ionescu (2015) and provides the ability to hook system calls natively using undocumented built-in features of the Windows operating system. These hooks occur every time a call returns from KM and provide the location that it had originally intended to return to. By performing analysis on this address and it's possible to detect manual system call usage and terminate execution (Richard, n.d.; Ullrich, 2021).

A model was developed using PIC to monitor a process's system call usage (PICE) to evaluate the performance impact of the hook. The hooked process (TIP) performed a large amount of system calls via WinAPI for the purpose of being intercepted and analyzed using PIC. Chapter 3 describes this model in-depth and further describes the setup of the environment. Quantitative results showing an impact on system performance were showcased in Chapter 4, confirming the original hypothesis of this study.

CPU Utilization

CPU utilization was a primary variable measured in each test to determine the system performance when PICE was applied. The results were individually measured as user-mode time (UM time) and kernel-mode time (KM time). Real-time was also measured using the number of ticks reported by the processor itself to represent how long execution lasts for the

end user. Collectively, these results represent the overall CPU impact that PICE had on Windows when used to detect manual system call usage.

Approximately 40,000 system calls were made by TIP during each test. The data was captured after each test and averaged to observe the different impact between versions.

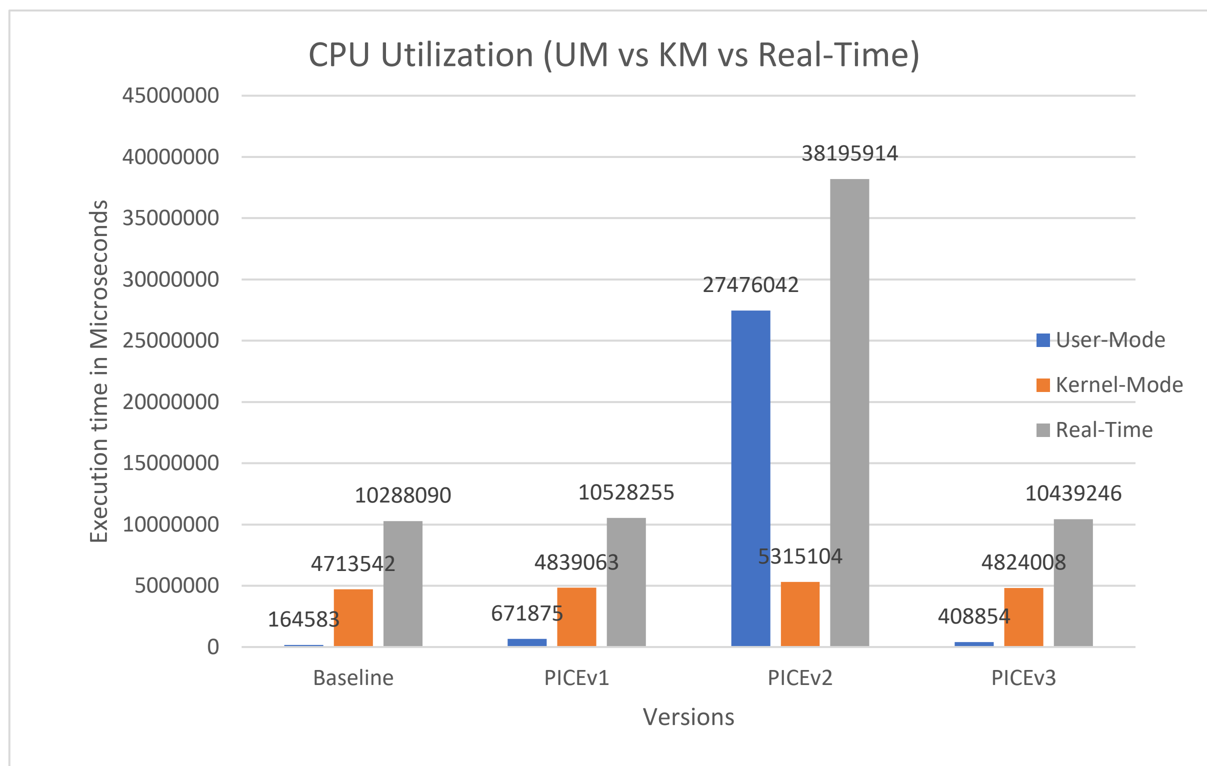


Figure 11. Comparison of CPU utilization time for each version of PICE

The bar graph in Figure 11 shows the comparison of CPU utilization for each version of PICE. As expected, PICEv2 had the largest impact on CPU utilization due to parsing the EAT of NTDLL for each system call it examined. Each version of PICE showed an increase in KM execution which can be attributed to the system calls made by PICE during analysis. Real-time performance scaled similarly to UM time and KM time with PICEv3 being the

exception. A different representation of the data is shown in Table 9 which examines the percentage of change between the baseline and the PICE iterations.

Table 8. Percentage of CPU utilization increase over Windows baseline without PICE

VERSION	UM-TIME	KM-TIME	CLOCK TIME
PICEV1	308%	3%	2%
PICEV2	16594%	13%	271%
PICEV3	148%	2%	1%

By its nature PICE performed most of its analysis in UM. Therefore, limiting execution spent in UM time is the primary focus when further improving PICE.

Memory Utilization

Another variable used to measure system performance was the amount of virtual and physical memory that was allocated by the process. Minimizing the memory usage of PICE was not a primary concern during development due to a small codebase. Unless a new image is loaded into the process space there is rarely any reason for an unexpected increase in the amount of memory utilized.

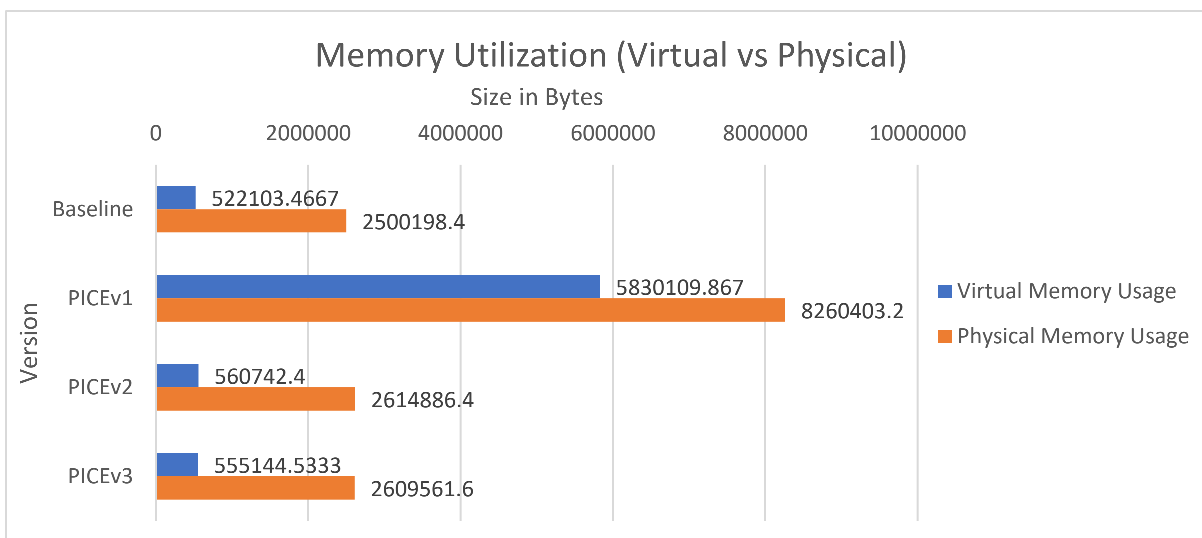


Figure 12. Average memory utilization for each version of PICE

The difference in memory usage between PICEv1 and the other versions can be attributed to its inclusion of the Microsoft Debugging Engine (MDE). As discussed in Chapter 4, MDE requires the import of DBGHELP.DLL to be loaded into the process's memory space. The removal of MDE was a clear success that caused memory utilization in subsequent versions to have a small increase over the baseline by comparison. PICEv2 and PICEv3 had an average increase of virtual memory by 7% and physical memory by 4%.

Windows 10 was configured with 8GB of RAM for the testing environment. This is consistent with the Microsoft recommended hardware requirements for optimal system usage (Microsoft, n.d.-c). Even at its peak in PICEV1 at 9 MB, the system had more than enough memory to compensate. In a real-world scenario, PICE is likely to be executed within multiple processes on a single system which would increase the system-wide amount of memory utilized.

Overall, these results have were proven statistically to impact the performance of Windows and future development should ensure that additional memory utilization is considered carefully to maximize portability.

Detection Capabilities

The secondary research question of this study was to determine if enhancements could be made to previous implementations of PIC to harden them from potential evasion. Both PICEv1 and PICEv2 contain changes that improve and add greater analysis than that included in previous research (Richard, n.d.; Ullrich, 2021). These changes are described in Chapter 4 and the logic flow can be observed in Figures 7 and 8. This section will briefly discuss the faults that these evolutions seek to address and the likelihood of success.

PICEV2's primary goal was to eliminate the use of MDE and verify the integrity of NTDLL. This was achieved by the implementation of an internal routine that dynamically parsed the EAT of NTDLL and stored the legitimate memory location of its exported functions and their corresponding SCNs (T, 2021/2022). After, due to the static calling convention in Windows (Figure 4) the return address was manipulated to the current function prologue and SCN. These two values should always be equivalent unless NTDLL has been tampered with as suggested by Ullrich (2021) by modifying the SCN.

The removal of MDE eliminated an indicator of analysis that could inform malware that it is being examined. DBGHELP.DLL is the implementation of MDE that is required to be imported into any program wishing to use its functionality. A process on Windows can easily query its own memory space for the presence of this DLL and alter its execution to stay hidden (Apostolopoulos et al., 2021). This technique can also be used to successfully check

for PICE.DLL. This was an accepted fault in PICE due to the fact it would require the introduction of uncontrolled variables such as external monitoring processes.

PICEv2 relied on the integrity of the EAT to provide a ground truth to use for analysis dynamically. A static version of SysCallList could be used, however, this would decrease the portability of the solution to future Windows versions that may have altered SCNs. This reliance can be abused if the integrity of the EAT is compromised in addition to the SCN. If the values match, PICEv2 would determine the call to not be manual. PICEv2 could instead use a different source to verify the integrity of NTDLL to protect against this. For example, the on-disk version of NTDLL could be loaded into memory to be compared for modification. This would likely introduce a higher impact on system performance and shift the weakness from the integrity of the EAT to the on-disk version of NTDLL. Alternatively, an existing solution outside of PICE could be utilized to augment its detection and provide a much greater level of assurance. This is discussed further in the Recommendations section.

The *syscall* instruction is responsible for the transfer of execution from UM to KM, but when performing this action, it must store the UM address to be executed upon return (Allievi et al., 2021). Malware can abuse this by ensuring that they do not execute the *syscall* instruction within their program's address space. Preparation of any arguments required for the system call (SCN) can be safely executed before jumping to a valid *syscall* instruction within NTDLL (Josh, 2021). Look to Figure 13 for the execution flow of this technique. After returning from KM, PICEv2 would check the return address and allow execution to continue since it exists within NTDLL. This is because the *syscall* instruction pushes the next instruction (*ret*) onto the stack before transferring execution to KM.

PICEv3 instead focused on performing an analysis of the values on the stack. After returning from KM, Windows calling convention (Figure 4) dictates that a *ret* instruction is executed. This will pop the current stack pointer (RSP) into RIP and execute it. As described in Chapter 4, bounds checking is performed on this location to determine if it belongs to KERNELBASE.DLL. A malicious binary that jumps to the *syscall* instruction within the desired NTDLL function would be detected by PICEv3, even though PICEv2 would not. A hypothetical representation of the stack is pictured in Figure 13 to illustrate the opportunity for analysis.

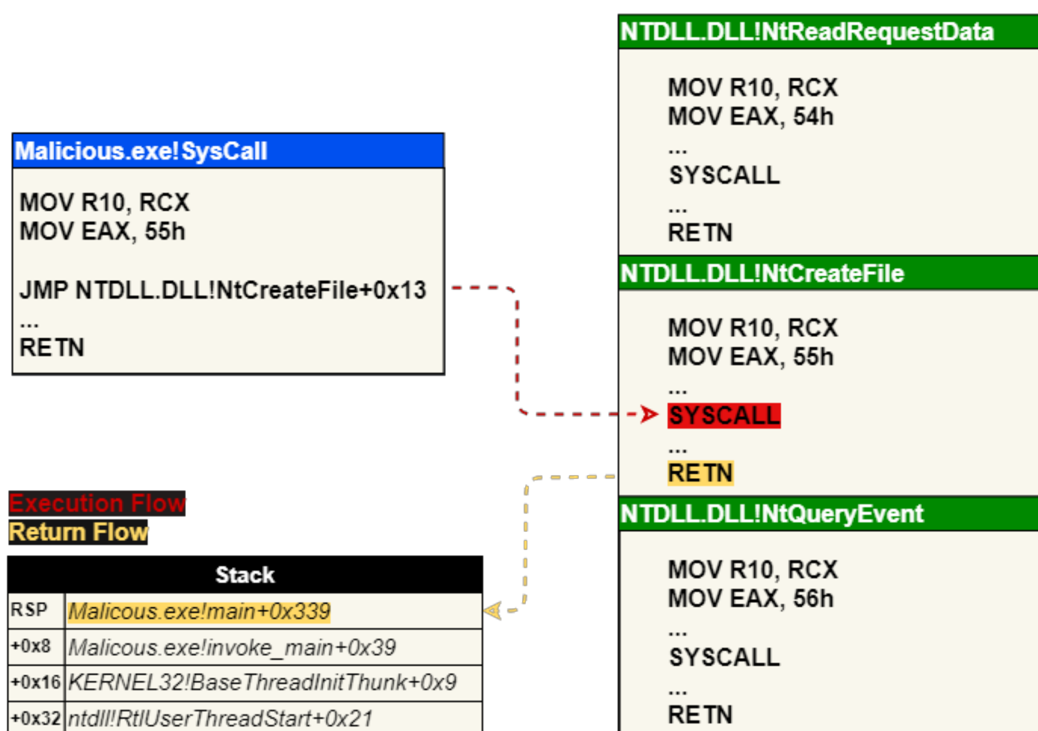


Figure 13. Jumping directly to syscall instruction to avoid PICEv2 detections (Josh, 2021)

This detection relies on the stability provided by WinAPI that legitimate applications require to ensure that their code can work in future versions of Windows (Allievi et al., 2021). If an attacker knew stack analysis was being performed, they could ensure the stack contains an address located within KERNELBASE.DLL. This can be achieved by manually resolving

and pushing onto the stack a pointer an address of a *ret* instruction within KERNELBASE.DLL. A more intensive stack analysis could be performed by PICEv3 to potentially identify this which is discussed in the Future Research section.

Discussion

This section will cover the researcher's recommendations regarding the use of PIC to detect the usage of manual system calls. Chapter 2 revealed that previous hooking methods implemented in UM are inadequate to intercept and analyze malicious programs. Previous research found that intercepting system calls adds a performance overhead to programs that can affect the overall performance of Windows (Lopez et al., 2017; Marhusin et al., 2008a). PICE was created to evaluate the potential impact caused by PIC interception and various levels of analysis.

Chapter 4 showed that all three versions of PICE had a significant impact on system performance. Depending on the implementation development, there can be a massive increase in both CPU and Memory utilization.

Using PIC to detect Manual System Calls

The literature review illustrated how a manual system call is a well-known bypass to modern user-mode hooks. Previous iterations have shown that PIC can be successful in detecting manual system calls by analyzing the location of the return address. This technique separates itself from the other defensive technologies such as ETW, who focus on what end action is performed by a system call. Instead, emphasis is placed on scrutinizing the differences in how the program performed the system call. PICEv2 performed this by

analyzing the memory around the return address to determine the last instructions executed. PICEv3 took a different approach and instead checked the UM stack for abnormal return locations. These detections were purposefully developed to be simple and heuristically based to highlight the parallels between increased analysis and performance impact.

TIP purposefully imported PICE.DLL before beginning its execution for this study. Real-life implementations of PICE.DLL would have to monitor programs that were not developed with PIC in mind. Like UM hooks, PICE.DLL can be injected into the memory space of a target process during runtime. It may be beneficial to automate the process by creating a Windows service to be responsible for monitoring process creation for targets. Once identified, the service can inject PICE.DLL and monitor for success. Furthermore, PIC could easily be configured to report telemetry to this service which could then ingest it into a database.

One of the limitations described earlier in this chapter was the ability to disable PIC using NtSetInformationProcess. This technique relies on modifying the KPROCESS structure of a process to overwrite the *InstrumentationCallback* field with a NULL value. To re-enable PIC another call must be made using NtSetInformationProcess to reset the field to Figure 5. A Windows service could be created to monitor the KPROCESS structure of any hooked processes to ensure that the *InstrumentationCallback field* has not been removed. NTDLL provides the function NtQueryInformationProcess that can be used to retrieve the KPROCESS structure from any process provided adequate permissions. Once it has been detected that PIC has been tampered with, the service can decide whether to terminate the process or reload PICE.DLL.

Integration with Existing Detection Mechanisms

It is important when considering the security of computer systems to utilize a defense-in-depth strategy. This means that security should be implemented as a layered approach to reduce the chances of a threat successfully bypassing your defenses (US-CERT, 2005). Examples of different layers that can be used to monitor host activity include UM hooks, kernel callbacks, Event-Tracing for Windows, and in-memory scanning. Developing these layers to be aware of each other can provide additional heuristics that detect when a program disables them. The act of trying to bypass a security control is itself a notable event that should be generated and made available to defenders.

This is the recommendation for implementing PIC into an existing environment. Its ability to detect the usage of manual system calls is only useful if an attacker is forced to decide to use them. Monitoring NTDLL functions via UM hooks creates a scenario where an attacker must use manual system calls to bypass it. Home field advantage is the primary strength of defenders and by limiting the techniques attackers can use it becomes much easier to write reliable detections. Therefore, PIC and UM hooks are complementary techniques that must be utilized together to be the most effective.

A commonality between the bypasses to both UM hooks and PIC is the modification of NTDLL in memory (Baranauskas, n.d.; MDSec, 2020; Tang, 2017). By default, when NTDLL is loaded, processes do not have write permissions to modify it. A program must use the system call *NtProtectVirtualMemory* to change the permissions and allow them to overwrite the desired sections. The literature surveyed in Chapter 2 shows that it is uncommon for this function to not be monitored by UM hooks by EDR vendors (Hamilton,

2021/2021). Consequently, this makes manual system calls the only option to remain undetected until the UM hooks can be removed from NTDLL.

It follows that a manual system call is necessary to execute *NtProtectVirtualMemory* without triggering detections by UM hooks. If these hooks are configured to scrutinize any processes attempting to modify the memory protections of NTDLL then attackers are left with two choices, even if they are aware of PIC's existence. The first choice is to use *NtSetInformationProcess* to disable PIC as described in previous sections to be able to use manual system calls. Like *NtProtectVirtualMemory*, this function is commonly hooked by EDR vendors using UM hooks and can be configured to terminate a process supplying the *InstrumentationCallback* field, protecting PIC. This creates a symbiotic relationship between both defensive technologies. The second way is to bypass the analysis that is performed when PIC intercepts a system call. The different iterations of PICE have shown that improvements can be made to previous work beyond just performing a bounds-check of the return address location. These are covered in-depth in the Detection Capabilities section found earlier in this chapter.

Potentially the most impactful way that PIC could be utilized with existing technologies is to verify that system calls were successfully intercepted for analysis. UM hooks perform analysis pre-system call and PIC performs analysis post-system call. If a log fails to be generated for either of the two technologies, then it can be assumed that one of them was bypassed and an alert should be generated. A hypothetical setup of this would require a managing agent that is responsible for ingesting telemetry from both hooks and providing API access to both implementation's detour functions to perform the check. This is

an extremely simple method of analysis and should be coupled with additional metrics to verify the authenticity of interception.

Future Research

There are increases in performance that can be found in the development process. The researcher was not concerned about achieving optimal performance but instead proved that PIC can affect the performance of a system. Future research into improving the performance of hooks is an important next step in adoption. Different algorithm choices for analysis could provide differences in overhead. Exclusions of boilerplate technologies such as the C runtime (CRT) could be achieved by rewriting PICE completely in assembly code which may speed up the processing time.

Although this study focuses on functions exported from NTDLL, there are additional UM callbacks in Table 2 that are yet to be analyzed. This was purposeful, to scope the research to manual system calls and provide the greatest benefit from a defensive perspective. Although the analysis was not performed, each one of these system calls was still intercepted by PICE to determine it belongs to NTDLL. This preliminary check becomes increasingly important when graphical applications are monitored. A much higher number of system calls are performed utilizing *KiUserCallbackDispatcher* to create and manage window drawing operations in a graphical program. Future studies should employ different metrics to study the visual impacts that are caused by increased processing time.

Lastly, the stack analysis performed by PICEv3 is not a foolproof method of analysis. Keeping performance in mind, additional checks should be performed using the addresses

present on the stack. Additional heuristics can be derived from the Windows API calling convention and used to prevent evasion.

Limitations

Performing a before-and-after study implies that the primary interest is the effect PIC exerts on the environment in which it has been installed and operated (Creswell & Creswell, 2018). It is not practical to explore every edge case for the detection of manual system calls or to include every system call in the TIP implementation. Defense security research is often a back-and-forth between defensive technologies and bypass techniques that leaves end-users caught in the middle. Effective security technologies should not negatively affect the stability or the performance of the operating system. For this reason, the performance of PIC must be studied further and appropriately considered before adoption.

The undocumented nature of PIC requires understanding that future versions of Windows may alter the implementation or remove it completely. Considering it has been around since at least Windows 7, it can be expected to continue to exist but caution must be observed when considering its use (Ionescu, 2016). Therefore, the findings of this study can only be guaranteed to be replicated using the operating system version: Windows 10 19042. Any of the analysis functions of PICE such as the resolution of SCNs and memory addresses were dynamically written to have the best chance at being version agnostic.

Microsoft allows for both 64-bit and 32-bit applications to be run on Windows 10 (Allievi et al., 2021). This is made possible by performing address translation of 32-bit system calls to their 64-bit counterpart before crossing into KM. PICE does not support the instrumentation of 32-bit system calls but existing research shows that it’s possible (Richard,

n.d.). A fully featured version of PICE would require 32-bit support to be able to properly monitor all potential applications for manual system calls.

It is possible to disable PICE by modifying the KPROCESS field using *NtSetInformationProcess* and providing a NULL value to the *InstrumentationCallback* field. There is little difference in the process described in Chapter 2 to place the hook, however instead of providing the structure described in Figure 5, instead, a NULL value is provided. Any process by default has the permission to set this value and it can be used to avoid detection. This study provides potential workarounds in the Recommendations section; however, no effort was made to make PICE resilient to it. Since the primary interest of this study was to observe the performance impact of PIC, this remains out of scope.

The development of PICE focused on the detection of anomalous behavior that accompanies the execution of manual system calls to be able to identify when they are used. Bounds checking of both the return address and the stack pointer was implemented in PICEv2 and PICEv3, but they do not represent the only way to perform detection. It is possible that further analysis could be performed using these two values to provide a stronger heuristic.

Functions that exist outside of NTDLL that were intercepted by PICE were not subject to the same analysis. For example, WIN32U.DLL primarily supports the graphical user interface (GUI) on Windows and does not provide the same capabilities as NTDLL which focuses on process management (Allievi et al., 2021). Existing research showcases the exploitation of WIN32U.DLL primarily used as a mechanism to execute malicious code in the kernel, which is out of the scope of this study (Kaspersky, 2021; Microsoft, 2022a). Additional exceptions were written for the functions in Table 2 that are not related to the detection of manual system calls.

Virtual snapshots limited the amount of outside influence on the model. These allowed a replica of the environment to be available for each test and limited the number of uncontrolled variables. By using a dedicated hypervisor to run the Windows 10 virtual machine the researcher ensured that the system had sole access to the underlying hardware. Even with all these precautions, there may be unknown routines taking execution time on the processor by the hypervisor. Additionally, Windows is an extremely complex operating system and often performs tasks unknown to the user in the background which could affect performance.

Summary

Chapter 5 concluded the documentation of this study and provided commentary on the results covered in the previous chapter. This quasi-experimental before-and-after study sought to determine the performance impact of using Process Instrumentation Callback to detect the use of manual system calls. The literature survey showed the gaps in existing detections on Windows and how PIC can augment them to achieve defense-in-depth. Previous studies comparing hook performance were reviewed to provide context and establish the need to measure the impact of intercepting system calls.

Three different iterations of PICE were developed to answer the secondary research objective to highlight the performance impacts of additional analysis. The quantitative results proved that in each version of PIC, there was a statistically significant impact on CPU utilization, memory utilization, and clock time. While it is still possible to evade the detections employed in PICE, this study illustrates several ways to improve by involving

other techniques like UM hooks. Overall, the researcher provides these results to encourage the addition of PIC to existing detection frameworks.

REFERENCES

Abimbola, A., Munoz, J. M., & Buchanan, W. (2006). NetHost-sensor: Monitoring a target host's application via system calls. Information Security Technical Report, 11, 166–175. https://doi.org/10.1016/j.istr.2006.10.003

Ahmed, M. E., Kim, H., Camtepe, S., & Nepal, S. (2021). Peeler: Profiling Kernel-Level Events to Detect Ransomware. In E. Bertino, H. Shulman, & M. Waidner (Eds.), Computer Security – ESORICS 2021 (pp. 240–260). Springer International Publishing. https://doi.org/10.1007/978-3-030-88418-5_12

Allievi, A., Ionescu, A., Russinovich, M. E., & Solomon, D. A. (2021). Windows internals, part 2 (7th ed.). Microsoft Press.

Alshehri, M., Brady, C., & Albert, J. (n.d.). Windows Memory-Injected Malware Detection Freeware Comparison. 9.

Apostolopoulos, T., Katos, V., Choo, K.-K. R., & Patsakis, C. (2021). Resurrecting anti-virtualization and anti-debugging: Unhooking your hooks. Future Generation Computer Systems, 116, 393–405. https://doi.org/10.1016/j.future.2020.11.004

Arghire, I. (2022). Ransomware Targeted 14 of 16 U.S. Critical Infrastructure Sectors in 2021 | SecurityWeek.Com. https://www.securityweek.com/ransomware-targeted-14-16-us-critical-infrastructure-sectors-2021

Baranauskas, M. (n.d.). Bypassing Cylance and other AVs/EDRs by Unhooking Windows APIs. Retrieved February 24, 2022, from https://www.ired.team/offensive-security/defense-evasion/bypassing-cylance-and-other-avs-edrs-by-unhooking-windows-apis

Bartolik, P. (2022, January 20). Financial Services Malware Just Won't Die. What to Do About It. CSO Online. https://www.csoonline.com/article/3648031/financial-services-malware-just-won-t-die-what-to-do-about-it.html

Bhansali, S., Chen, W.-K., de Jong, S., Edwards, A., Murray, R., Drinić, M., Mihočka, D., & Chau, J. (2006). Framework for instruction-level tracing and analysis of program executions. Proceedings of the 2nd International Conference on Virtual Execution Environments - VEE '06, 154. https://doi.org/10.1145/1134760.1220164

Bode, A., & Warnars, N. (n.d.). Detecting Fileless Malicious Behaviour of .NET C2 Agents using ETW. 9.

Boslaugh, S. (2012). Statistics in a Nutshell: A Desktop Quick Reference. O'Reilly Media, Inc.

Botor, T., & Habiballa, H. (2018). Comparison of time measurement methods in C++. AIP Conference Proceedings, 1978(1), 060003. https://doi.org/10.1063/1.5043705

Bremer, J. (2012). X86 API Hooking Demystified | Development & Security. X86 API Hooking Demystified. http://jbremer.org/x86-api-hooking-demystified/

Catlin, B., Hanrahan, J. E., Russinovich, M. E., Solomon, D. A., & Ionescu, A. (2017a). System architecture, processes, threads, memory management, and more (Seventh edition). Microsoft.

Catlin, B., Hanrahan, J. E., Russinovich, M. E., Solomon, D. A., & Ionescu, A. (2017b). System architecture, processes, threads, memory management, and more (Seventh edition). Microsoft.

ChainAnalysis. (2022, January 19). Meet the Malware Families Helping Hackers Steal and Mine Millions in Cryptocurrency. Chainalysis. https://blog.chainalysis.com/reports/2022-crypto-crime-report-preview-malware/

Chappel, G. (n.d.). KPROCESS. Retrieved February 24, 2022, from https://www.geoffchappell.com/studies/windows/km/ntoskrnl/inc/ntos/ke/kprocess/index.htm

Chester, A. (2020, March 24). Hiding Your .NET – ETW. Chester. https://www.mdsec.co.uk/2020/03/hiding-your-net-etw/

Cocomazzi, A. (n.d.). Weaponizing Mapping Injection with Instrumentation Callback for stealthier process injection. Weaponizing Mapping Injection with Instrumentation Callback for Stealthier Process Injection. Retrieved January 18, 2022, from https://splintercod3.blogspot.com/p/weaponizing-mapping-injection-with.html

Creswell, J. W. (2015). Educational research: Planning, conducting and evaluating quantitative and qualitative research.

Creswell, J. W., & Creswell, J. D. (2018). Research design: Qualitative, quantitative, and mixed methods approaches (Fifth edition). SAGE.

Czeczko, P. (2020, April 5). Malware development part 2—Anti dynamic analysis & sandboxes. https://0xpat.github.io/Malware_development_part_2/

de Plaa, C. (n.d.). Red Team Tactics: Combining Direct System Calls and sRDI to bypass AV/EDR | Outflank Blog. Retrieved January 18, 2022, from http://www.outflank.nl/publications

de Plaa, C. (2020). TamperETW [C]. Outflank B.V. https://github.com/outflanknl/TamperETW (Original work published 2020)

de Winter, J. C. F. (n.d.). Using the Student's t-test with extremely small sample sizes. https://doi.org/10.7275/E4R6-DJ05

Eder, T., Rodler, M., Vymazal, D., & Zeilinger, M. (2013). ANANAS - A Framework for Analyzing Android Applications. Proceedings of the 2013 International Conference on Availability, Reliability and Security, 711–719. https://doi.org/10.1109/ARES.2013.93

Everdox, N. (2013, February 13). Windows x64 System Service Hooks and Advanced Debugging. CodeProject. https://www.codeproject.com/Articles/543542/Windows-x64-System-Service-Hooks-and-Advanced-Debu

Fralick, D., Zheng, J. Z., Wang, B., TU, X. M., & Feng, C. (2017). The Differences and Similarities Between Two-Sample T-Test and Paired T-Test. Shanghai Archives of Psychiatry, 29(3), 184–188. https://doi.org/10.11919/j.issn.1002-0829.217070

Gavriel, H. (2018, November 27). Malware Mitigation when Direct System Calls are Used. Cyberbit. https://www.cyberbit.com/blog/endpoint-security/malware-mitigation-when-direct-system-calls-are-used/

GenPact. (2014). The-impact-of-technology-on-business-process-operations.pdf. https://www.genpact.com/downloadable-content/insight/the-impact-of-technology-on-business-process-operations.pdf

H, P. (2020, December 16). Experimenting with Protected Processes and Threat-Intelligence. Pat_h/to/File. https://blog.tofile.dev/2020/12/16/elam.html

Hamilton, C. (2021). EDRs [C]. https://github.com/Mr-Un1k0d3r/EDRs (Original work published 2021)

Hand, M. (2020, December 16). Adventures in Dynamic Evasion. Medium. https://posts.specterops.io/adventures-in-dynamic-evasion-1fe0bac57aa

Harris, A. D. (2006). The Use and Interpretation of Quasi-Experimental Studies in Medical Informatics. Journal of the American Medical Informatics Association : JAMIA, 13(1), 16–23. https://doi.org/10.1197/jamia.M1749

HealthITSecurity. (2020, October 7). US Ransomware Attacks Doubled in Q3; Healthcare Sector Most Targeted. HealthITSecurity. https://healthitsecurity.com/news/us-ransomware-attacks-doubled-in-q3-healthcare-sector-most-targeted

Hirofuchi, T. (2018). SimGrid VM: Virtual Machine Support for a Simulation Framework of Distributed Systems. https://www.ezproxy.dsu.edu:2063/document/7274701/

Hoffman, C. (2020). Read This to Understand Windows 10 Update Names and Numbers. How-To Geek. https://www.howtogeek.com/697411/read-this-to-understand-windows-10-update-names-and-numbers/

Hydra. (2020, September 18). Implementing Direct Syscalls Using Hell's Gate. Team Hydra. https://teamhydra.blog/2020/09/18/implementing-direct-syscalls-using-hells-gate/

Ionescu, A. (2016). Ionescu007/HookingNirvana [C]. https://github.com/ionescu007/HookingNirvana/blob/9e4e8e326b9dfd10a7410986486e567e5980f913/Esoteric%20Hooks.pdf

Jia, H., Mao, R., & Wu, W. (2014). Methods to monitor process's Spatial and temporal consumption. 2014 11th International Conference on Fuzzy Systems and Knowledge Discovery (FSKD), 892–897. https://doi.org/10.1109/FSKD.2014.6980957

Josh. (2021, June 9). Hiding Your Syscalls. PTHB. https://passthehashbrowns.github.io/hiding-your-syscalls

Kaspersky. (2021, October 12). CVE-2021-40449: Trojan delivery vulnerability. https://www.kaspersky.com/blog/mysterysnail-cve-2021-40449/42448/

Klein, A., Kotler, I., & Labs, S. (n.d.). Windows Process Injection in 2019. 34.

Kleymenov, A., & Thabet, A. (2019). Mastering Malware Analysis: The complete malware analyst's guide to combating malicious software, APT, cybercrime, and IoT attacks.

Kumar, R. (2014). Research Methodology: A Step-by-Step Guide for Beginners. SAGE. https://books.google.co.uk/books?id=MKGVAgAAQBAJ&printsec=frontcover&source=gbs_ge_summary_r&cad=0#v=onepage&q&f=false

Kumar, R. (2019). Research Methodology: A Step-by-Step Guide for Beginners.

Lopez, J., Babun, L., Aksu, H., & Uluagac, S. (2017). A Survey on Function and System Call Hooking Approaches. Journal of Hardware and Systems Security, 1. https://doi.org/10.1007/s41635-017-0013-2

Lukan, D. (2014). Hooking the System Service Dispatch Table (SSDT). Infosec Resources. https://resources.infosecinstitute.com/topic/hooking-system-service-dispatch-table-ssdt/

Mandt, T. (2011). Kernel Attacks through User-Mode Callbacks. 29.

Marhusin, M. F., Larkin, H., Lokan, C., & Cornforth, D. (2008a). An Evaluation of API Calls Hooking Performance. 2008 International Conference on Computational Intelligence and Security, 1, 315–319. https://doi.org/10.1109/CIS.2008.199

Marhusin, M. F., Larkin, H., Lokan, C., & Cornforth, D. (2008b). An Evaluation of API Calls Hooking Performance. 1, 315–319. https://doi.org/10.1109/CIS.2008.199

Mauldin, R. L. (2020). Foundations of Social Research. https://uta.pressbooks.pub/foundationsofsocialworkresearch/

MDSec. (2020, December 31). Bypassing User-Mode Hooks and Direct Invocation of System Calls for Red Teams. MDSec. https://www.mdsec.co.uk/2020/12/bypassing-user-mode-hooks-and-direct-invocation-of-system-calls-for-red-teams/

Microsoft. (n.d.-a). Dynamic-Link Libraries (Dynamic-Link Libraries)—Win32 apps. Retrieved February 12, 2022, from https://docs.microsoft.com/en-us/windows/win32/dlls/dynamic-link-libraries

Microsoft. (n.d.-b). ELAM driver submission Process—Windows drivers. Retrieved February 11, 2022, from https://docs.microsoft.com/en-us/windows-hardware/drivers/install/elam-driver-submission

Microsoft. (n.d.-c). How to Check Windows 10 Computer System Specs & Requirements—Microsoft. Windows. Retrieved February 15, 2022, from https://www.microsoft.com/en-us/windows/www.microsoft.com/en-us/windows/windows-10-specifications

Microsoft. (n.d.-d). User mode and kernel mode—Windows drivers. Retrieved February 17, 2022, from https://docs.microsoft.com/en-us/windows-hardware/drivers/gettingstarted/user-mode-and-kernel-mode

Microsoft. (n.d.-e). Virtual address spaces—Windows drivers. Retrieved February 18, 2022, from https://docs.microsoft.com/en-us/windows-hardware/drivers/gettingstarted/virtual-address-spaces

Microsoft. (n.d.-f). Windows Hardware Compatibility Program. Retrieved February 18, 2022, from https://docs.microsoft.com/en-us/windows-hardware/design/compatibility/

Microsoft. (n.d.-g). X64 calling convention. Retrieved February 16, 2022, from https://docs.microsoft.com/en-us/cpp/build/x64-calling-convention

Microsoft. (2002). Detours. Microsoft Research. https://www.microsoft.com/en-us/research/project/detours/

Microsoft. (2022a). CVE-2022-21882—Security Update Guide—Microsoft—Win32k Elevation of Privilege Vulnerability. https://msrc.microsoft.com/update-guide/vulnerability/CVE-2022-21882

Microsoft. (2022b, January 4). Acquiring high-resolution time stamps—Win32 apps. https://learn.microsoft.com/en-us/windows/win32/sysinfo/acquiring-high-resolution-time-stamps

Microsoft. (2022c, February 2). GetProcessMemoryInfo function (psapi.h)—Win32 apps. https://learn.microsoft.com/en-us/windows/win32/api/psapi/nf-psapi-getprocessmemoryinfo

Miller, C. J., Smith, S. N., & Pugatch, M. (2020). Experimental and quasi-experimental designs in implementation research. Psychiatry Research, 283, 112452. https://doi.org/10.1016/j.psychres.2019.06.027

Mishra, P., Singh, U., Pandey, C., Mishra, P., & Pandey, G. (2019). Application of student's t-test, analysis of variance, and covariance. Annals of Cardiac Anaesthesia, 22(4), 407. https://doi.org/10.4103/aca.ACA_94_19

Monnappa, K. A. (2018). Learning malware analysis: Explore the concepts, tools, and techniques to analyze and investigate Windows malware. Packt Publishing.

nick.p.everdox. (2013, February 13). Windows x64 System Service Hooks and Advanced Debugging. CodeProject. https://www.codeproject.com/Articles/543542/Windows-x64-System-Service-Hooks-and-Advanced-Debu

Noah. (2022). Secrary/Hooking-via-InstrumentationCallback [C++]. https://github.com/secrary/Hooking-via-InstrumentationCallback/blob/442607787b83a4cf8a2be77c9a1f917224e1fe90/instrumentationcallback/instrumentationcallback.cpp (Original work published 2017)

Palantir. (2019, February 28). Tampering with Windows Event Tracing: Background, Offense, and Defense. Medium. https://blog.palantir.com/tampering-with-windows-event-tracing-background-offense-and-defense-4be7ac62ac63

Reddy, M. (2011). API design for C++. Morgan Kaufmann.

Richard, C. (n.d.). X86 Nirvana Hooks & Manual Syscall Detection. Retrieved February 8, 2022, from https://blog.xenoscr.net/2022/01/17/x86-Nirvana-Hooks.html

Schulman, A., Maxey, D., & Pietrek, M. (1992). Undocumented Windows. Addison-Wesley.

Singh, B., Evtyushkin, D., Elwell, J., Riley, R., & Cervesato, I. (2017). On the Detection of Kernel-Level Rootkits Using Hardware Performance Counters. Proceedings of the 2017 ACM on Asia Conference on Computer and Communications Security, 483–493. https://doi.org/10.1145/3052973.3052999

Statista. (2020). Global server share by OS 2018-2019. Statista. https://www.statista.com/statistics/915085/global-server-share-by-os/

Statista. (2021a). Desktop OS market share. Statista. https://www.statista.com/statistics/218089/global-market-share-of-windows-7/

Statista. (2021b). Windows 7/10 adoption rate 2020. Statista. https://www.statista.com/statistics/897222/north-america-western-europe-windows-7-10-adoption/

T, J. (2022). SysWhispers2 [Assembly]. https://github.com/jthuraisamy/SysWhispers2 (Original work published 2021)

T, J. (2022). SysWhispers [Assembly]. https://github.com/jthuraisamy/SysWhispers (Original work published 2019)

Tang, J. (2017). Universal Unhooking: Blinding Security Software. https://blogs.blackberry.com/en/2017/02/universal-unhooking-blinding-security-software

Teodorescu, C., Korkin, I., & Golchikov, A. (n.d.). Veni, No Vidi, No Vici: Attacks on ETW Blind EDR Sensors. 51.

Ullrich, J. (2021, February 10). Detecting Manual Syscalls from User Mode. Winternl. https://winternl.com/detecting-manual-syscalls-from-user-mode/

US-CERT. (2005). Defense in Depth | CISA. https://www.cisa.gov/uscert/bsi/articles/knowledge/principles/defense-in-depth

Valasek, C. (2010). Understanding the Low Fragmentation Heap. 86.

VMware. (n.d.). Overview of virtual machine snapshots in vSphere (1015180). https://kb.vmware.com/s/article/1015180

WHQL. (n.d.). WHQL Release Signature—Windows drivers. Retrieved February 24, 2022, from https://docs.microsoft.com/en-us/windows-hardware/drivers/install/whql-release-signature

Willems, C., Holz, T., & Freiling, F. (2007). Toward Automated Dynamic Malware Analysis Using CWSandbox. IEEE Security Privacy, 5(2), 32–39. https://doi.org/10.1109/MSP.2007.45

Win, T. Y., Tianfield, H., & Mair, Q. (2015). Detection of Malware and Kernel-Level Rootkits in Cloud Computing Environments. 2015 IEEE 2nd International Conference on Cyber Security and Cloud Computing, 295–300. https://doi.org/10.1109/CSCloud.2015.54

Windows API. (2021). In Wikipedia. https://en.wikipedia.org/w/index.php?title=Windows_API&oldid=1053587257

x64dbg. (n.d.-a). Add x64 instrumentation callback hook · x64dbg/ScyllaHide@66ec9ad. GitHub. Retrieved January 25, 2022, from https://github.com/x64dbg/ScyllaHide/commit/66ec9ad5140a4deca88cf9e0178ec5728bba077b

x64dbg. (n.d.-b). X64dbg. Retrieved February 22, 2022, from https://x64dbg.com/

Zhang, F. (2018). A Survey on Virtual Machine Migration: Challenges, Techniques, and Open Issues. https://www.ezproxy.dsu.edu:2063/document/8260891/

APPENDIX A: AVERAGE USER-MODE AND KERNEL-MODE TIMES

The tables below are the raw values captured from each of the 30 tests conducted. The data is represented in microseconds and was captured using the *GetProcessTimes* function.

User-Mode CPU Time

Baseline	PICEv1	PICEv2	PICEv3
140625	578125	27375000	328125
234375	562500	26515625	281250
171875	812500	27890625	296875
125000	593750	27187500	421875
171875	718750	27578125	375000
187500	765625	27609375	500000
218750	765625	27296875	421875
171875	703125	27500000	500000
125000	656250	27203125	359375
140625	578125	27218750	562500
171875	562500	27359375	359375
109375	765625	26703125	328125
187500	609375	27859375	328125
218750	765625	27875000	406250
140625	656250	27171875	437500
156250	750000	27796875	500000
62500	609375	27265625	343750
125000	640625	28328125	515625
171875	750000	27750000	375000
125000	500000	27687500	421875
187500	750000	27421875	312500
125000	625000	27937500	390625
218750	671875	27171875	359375
156250	796875	27093750	437500
250000	687500	27718750	406250
109375	593750	27718750	484375
140625	593750	28265625	468750
203125	718750	27234375	515625
250000	656250	27375000	484375

140625	718750	27171875	343750

Kernel-Mode CPU Time

Baseline	PICEv1	PICEv2	PICEv3
5093750	5015625	5453125	5015625
4703125	5140625	5421875	4812500
4562500	4765625	5281250	4843750
4953125	4796875	5953125	4875000
4890625	4718750	5046875	5062500
4906250	4593750	5109375	4750000
4531250	4937500	5421875	4843750
4593750	4718750	5187500	4687500
4968750	5125000	5453125	4859375
4703125	4890625	5375000	5360875
4484375	4875000	5093750	4703125
4750000	4609375	5937500	4953125
4828125	5078125	5062500	4593750
4843750	4718750	5250000	5015625
4500000	4828125	5031250	4765625
4796875	4812500	5234375	4500000
5125000	4984375	6046875	4687500
4687500	4765625	5468750	4921875
4828125	4734375	5312500	4765625
4890625	5015625	5156250	4843750
4734375	4750000	5000000	4812500
5015625	4906250	5296875	4640625
4859375	4640625	5562500	4906250
4828125	4750000	5203125	4609375
3468750	4796875	5406250	4734375
4640625	4843750	5453125	4750000
4703125	4968750	5015625	4875000
4406250	4921875	5000000	4859375
4546875	4875000	5093750	4953125
4562500	4593750	5125000	4718750

APPENDIX B: AVERAGE MEMORY UTILIZATION

The tables below are the raw values captured from each of the 30 tests conducted. The data is represented in bytes and was captured using the *GetProcessMemoryInfo* function.

Virtual Memory Usage

Baseline	PICEv1	PICEv2	PICEv3
561152	6316032	610304	598016
520192	5808128	573440	544768
520192	5820416	548864	561152
520192	5812224	585728	552960
528384	5812224	561152	561152
524288	5812224	552960	561152
524288	5820416	552960	557056
520192	5816320	552960	557056
524288	5820416	557056	557056
520192	5812224	565248	598016
520192	5812224	581632	557056
520192	5808128	561152	548864
524288	5812224	557056	557056
520192	5812224	552960	552960
516096	5816320	552960	552960
520192	5812224	561152	557056
516096	5812224	561152	548864
520192	5812224	552960	552960
516096	5820416	565248	548864
516096	5820416	561152	548864
516096	5816320	557056	548864
524288	5816320	561152	544768
528384	5812224	552960	548864
520192	5808128	557056	552960
516096	5804032	557056	548864
520192	5812224	552960	552960
524288	5812224	552960	548864
520192	5808128	548864	544768
520192	5808128	561152	544768
520192	5816320	552960	544768

Physical Memory Usage

Baseline	PICEv1	PICEv2	PICEv3
2789376	8953856	2891776	2887680
2490368	8232960	2592768	2588672
2490368	8237056	2592768	2588672
2490368	8237056	2605056	2588672
2490368	8232960	2596864	2588672
2490368	8237056	2592768	2600960
2494464	8237056	2592768	2588672
2490368	8237056	2592768	2588672
2490368	8237056	2592768	2588672
2486272	8237056	2592768	2899968
2490368	8232960	2609152	2580480
2490368	8237056	2596864	2588672
2490368	8237056	2596864	2588672
2494464	8232960	2596864	2588672
2482176	8237056	2592768	2588672
2490368	8237056	2625536	2600960
2490368	8237056	2613248	2588672
2490368	8237056	2609152	2592768
2490368	8241152	2617344	2588672
2490368	8241152	2613248	2592768
2486272	8237056	2609152	2588672
2490368	8237056	2609152	2588672
2490368	8241152	2609152	2584576
2490368	8232960	2617344	2588672
2490368	8232960	2613248	2588672
2490368	8237056	2617344	2592768
2490368	8237056	2613248	2588672
2490368	8237056	2613248	2584576
2494464	8232960	2617344	2588672
2490368	8237056	2613248	2584576

APPENDIX C: AVERAGE REAL-TIME

The tables below are the raw values captured from each of the 30 tests conducted. The data is represented in microseconds and was captured using the *QueryPerformanceCounter* function.

Clock Time

Baseline	PICEv1	PICEv2	PICEv3
10519194	10516491	38163455	10907528
10419200	11101732	37572856	10312567
12021302	11082052	38624968	10289657
10278332	10454859	38106474	10273096
10265414	10537890	37890323	10439761
10302468	10149571	37920606	10263001
10017148	10519282	37856422	10411751
9991042	10614789	37948388	10375473
10360497	10461239	38127109	10395371
10623391	10277930	37879692	11761360
9879046	10458340	38330299	10241341
10104805	10902021	38343515	10374293
10275809	10330534	38377821	10177954
10579521	10322293	38895060	10803485
10196625	10350880	38012634	10280770
10191335	11383973	37870057	10254071
10484699	10328307	38723373	10751630
10339862	10607507	38977411	10506098
10185629	10431411	38572109	9953495
10337557	10625102	37942730	10099592
10472450	10343153	37823920	10403795
11018685	10332585	38263732	10261309
10290149	10052854	38128014	11420178
10573324	10339869	38045491	10404926
8767614	10617098	38257522	10389716
10010813	10538771	38199036	10331564
10192124	10661789	39764391	10022715
10261880	10597782	37424581	10378666
9758485	10636352	37808187	10364948

9924312	10271191	38027240	10327264

APPENDIX D: SOURCE CODE LOCATION

The source code for both the Process Instrumentation Callback Engine (PICE) and the Target Instrumentation Process (TIP) are located at the following link:

https://github.com/dsujacob/PICE

APPENDIX E: GLOSSARY

Term	Definition
C Runtime	C program's execution environment
Clock Time	Actual elapsed time
Degrees of Freedom	Independent data points
Dynamic Link Library	Shared code library
Export Address Table	List of exported functions
Hook	Intercept system calls
Kernel	Core of operating system
Kernel-Mode	Execution in kernel space
KPROCESS	Kernel process structure
Manual System Call	System call made without NTAPI
Native API	Direct OS interface
Physical memory	RAM hardware
PICE	Process Instrumentation Callback Engine
Process Instrumentation Callback	Post-process system call hook technique
System Call	Kernel service request
TIP	Target Instrumentation Process
T-test	Statistical hypothesis test
User-Mode Callback	Function executed in user-mode
User-Mode	Execution in user space

Virtual Memory	Memory abstraction
Windows API	OS service interface