Universidad de Buenos Aires
Facultad de Ciencias Exactas y Naturales
Departamento de Computación

Studying the logging capability of Windows Telemetry component using Reverse Engineering

Tesis de Licenciatura en Ciencias de la Computación

Pablo Agustín Artuso
LU: 282/11
artusopablo@gmail.com

Director: Rodolfo Baader <rbaader@dc.uba.ar>

Codirector: Aleksandar Milenkoski <amilenkoski@ernw.de>

Buenos Aires, 2022

ABSTRACT (ENGLISH VERSION)

Windows, one of the most popular OS, has a component called Telemetry. It collects information from the system with the goal of analyzing and later diagnosing and fixing software and hardware problems and improving the user experience, among others. The kind of information that can be obtained by this component is partially configurable by specifying one of four different levels: security, basic, enhanced and full, being "security" the level where less information is gathered and "full" the opposite case.

How Telemetry stores/process/administrates the information extracted? It employs a widely used framework called Event Tracing for Windows (ETW) [4]. Embedded not only in userland applications but also in the kernel modules, the ETW framework has the goal of providing a common interface to log events and therefore help to debug and log system operations.

In this work, we are going to analyze a part of the Windows kernel to better understand how Telemetry works from an internal perspective. Due to Kernel source code not being open, techniques like reverse engineering [5][6] will be used. As a consequence, other complex challenges will be involved such as kernel debugging, dealing with undocumented kernel internal structures, reversing of big frameworks (i.e: ETW), binary libraries which lack symbols, etc. This work will make Windows analysts, IT admins or even Windows users, more aware about the functionality of the Telemetry component. As a consequence, it will provide necessary resources to deeply understand and help deal with privacy issues, bug fixing, knowledge of collected data, etc.

ABSTRACT (SPANISH VERSION)

Windows, uno de los sistemas operativos más populares, tiene un componente llamado Telemetría. Dicho componente recolecta información del sistema con el objetivo de analizarla para después poder diagnosticar y reparar problemas de software y hardware, mejorar la experiencia de usuario, entre otros. El tipo de información obtenida por este componente es parcialmente configurable a través de la especificación de uno de estos 4 niveles: Seguridad, Básico, Mejorado y Completo, siendo "Seguridad" el nivel que menos información recolecta y "Completo" el que más.

¿Cómo hace Telemetría para guardar/procesar/administrar la información extraída? Hace uso de un mecanismo interno de Windows llamado "Seguimiento de Eventos para Windows" (ETW) [4]. Embebido tanto en aplicaciones de usuario como en módulos de Kernel, ETW tiene el objetivo de proveer una interfaz común de escritura de eventos y por lo tanto ayudar a depurar y dejar registro de operaciones del sistema.

En este trabajo, analizaremos una parte del Kernel de Windows con el objetivo de entender cómo funciona el componente de Telemetría desde una perspectiva interna. Dado que el código fuente del Kernel de Windows no es de público acceso, se aplicarán técnicas tales como ingeniería reversa [5], [6]; lo cual implica otros desafíos complejos tales como depuración de Kernel, lidiar con estructuras de Kernel no documentadas previamente, reverseo de mecanismos complejos (ETW), librerías sin símbolos, etc. Este trabajo hará que tanto analistas de Windows, administradores IT o incluso usuarios de Windows estén más conscientes sobre el comportamiento del componente. Como consecuencia, se proveerá de recursos necesarios para entender y ayudar a lidiar con temas de privacidad, corrección de errores, conocimiento de información recolectada, etc.

ACKNOWLEDGMENTS

A mis padres, Verónica y Jorge, que me impulsaron siempre a estudiar una carrera universitaria dándome su apoyo fiel e incondicional desde el principio. Haciendo todo lo posible, para que yo simplemente me preocupe en estudiar.

A mi esposa, Valentina, la cual me acompañó y soportó mis altos y bajos desde que empecé hasta que terminé la carrera.

A mi director, Rodolfo Baader, por su eterna paciencia y excelente acompañamiento durante todo el proceso de tesis. Por dedicarle tiempo a debatir y charlar aún en tiempos complicados.

A mi co-director, Aleksandar Milenkoski, por elegirme y darme la oportunidad de realizar pasantía junto a él y acompañarme durante todo el proceso de investigación.

A mis dos compañeros y amigos, Federico Landini y Laouen Belloli, por contagiarme esas ganas de querer aprender, estudiar, compartir ideas y debatir. Por enseñarme qué es ser un equipo. Por apoyarme y ayudarme cuando más lo necesitaba. Por aguantar esas interminables horas haciendo trabajos prácticos en los labos. Ustedes dos son el gran motivo por el cual hoy estoy pudiendo terminar este camino.

A mi país, Argentina, por brindarme la posibilidad de estudiar y aprender de los mejores profesores del país sin tener que pagar un arancel.

Al Departamento de Computación de la UBA y a todos sus profesores, que me enseñaron, explicaron y ayudaron para que yo hoy pueda estar acá.

A mis hermanos, Jor y Caro, por haberme incentivado a seguir una carrera de la UBA al igual que ellos.

A mis amigos que conocí gracias al mundo laboral, sobre todo Nahuel Sanchez y Andrés Blanco, por incentivarme en todo momento a terminar la última materia y motivándome a superar obstáculos aún cuando creía que no eran posibles.

A mis amigos y a todas aquellas personas que me acompañaron y apoyaron durante todo estos años.

CONTENTS

1. MOTIVATION

The analysis of the Windows OS Telemetry component presented in this work, was performed with the following goals:

- Understand how the process of generating logs is carried out.
- How, where and what logs are stored.
- Which applications are involved in gathering Telemetry information.
- Provide a comprehensive source of information of the kernel structures involved.
- Reverse engineer processes and structures in order to contribute to the community and help future research projects based on the same topic.

2. INTRODUCTION

The analysis presented in this project has the goal of better understanding and illustrating how Windows Telemetry works from an internal perspective. We will show how, even in difficult situations where documentation is scarce and/or source code is not publicly available, it is still possible to understand the way a component works. By leveraging the usage of advanced techniques such as Reverse Engineering and Kernel Live Debugging, we will demonstrate how Windows Telemetry builds, stores and sends data from the system. Additionally, several scripts developed in order to automatize the extraction of data will be shared with the objective of providing a base for future projects which may leverage them to perform analysis in the same or similar topics.

This investigation was carried out in a specific version of the Windows OS: Windows 10 64 bits Enterprise, 1607. Windows delivers upgrades of its Operating System, usually, twice a year. Originally, this project started in 2018 with an internship in ERNW GMbH. At that time, ERNW was working closely with the Federal Office of Information Security. There were several reasons why this specific version was chosen:

- It was one of the mainstream versions of Windows at the moment of starting the project.
- It was a long support version (EOS: April 2019).
- It was used by the German Police Office.

Although this version may sound a bit old as of today, all the analysis is also applicable to newer versions such as Windows 10 64 bits Enterprise 1909. This was confirmed using the final version of the scripts that were developed during this project, and adjusting some of the offsets where the breakpoints were set.

Following sections will introduce several basic concepts, tools and other Windows components that were key for our analysis.

2.1 Basic concepts

In order to carry out the analysis, two techniques were used: Reverse Engineering and Live Debugging. Both of them played a central role as they provided the resources that allowed us to study how Telemetry operated at a given point in time. Despite the fact that each of them provided different features, the combination of both was the foundation for the full analysis of the component. The following two chapters will introduce and explain the aforementioned techniques.

2.1.1 Reverse Engineering

Software engineering can be defined as the process of designing, building and testing computer software. A software engineer is a human being with skills that allow them to write, in a particular programming language, instructions that ultimately the computer processor will execute. At the very beginning, the only available languages the so-called "low level" languages (because they are really "close" to the processor itself). The software engineers of that moment had to know very specific instructions that the processor was able to execute, and write programs just with that. This means, that no advanced control features were available, just the basic instruction set provided by the processor. With the help of modularization and the evolution of computers, new abstraction layers began to appear, easing the process of developing programs. These abstraction layers allowed humans to avoid knowing what was happening beyond their current layer and, at the same time, it exposed ways to use the same capabilities as before in a much easier way. Currently, most software engineers (for instance: web developers) do not need to fully understand how the statements they write inside an IDE end up being executed by the CPU. This happens thanks to the hard work of years and years of the aforementioned software engineers that built these abstraction layers. One of the important abstraction layers built are the compilers. These programs convert high level programming language instructions that are human readable to 0's and 1's that are finally executed by the CPU. As mentioned before, this allows most current software engineers to ignore what is going inside them and just make use of their capabilities.

Currently, if somebody wants to understand what a program is doing, usually they read its source code (human readable). However, this is not always possible. It could happen that you do not have the source code of that program because of several reasons: You lost it or it is a closed project. In that case, how can you know what the program is actually doing?

Usually the flow of creating a new program involves:

- Develop the source code of the program.
- Compile the source code.
- Execute the compiled program (executable file)

As mentioned before, programs ultimately execute CPU's instructions. Therefore, although being a bit more difficult than reading the source code, it is possible to decompile the compiled program in order to read the CPU instructions that it is going to execute. The language which is expressed with CPU instructions is usually known as Assembly language. Sometimes a single line of source code of a high level language could end up in dozens of lines in Assembly code.

The process of reading the executable's Assembly code and learning the details on how that program works is also known as Reverse Engineering. The ultimate goal would be to get a high level code representation which does the same as the original executable, although this can be sometimes difficult to achieve. The name comes from the fact that is the exact reverse process of the creating program flow detailed before. With the difference that most of the time it is almost impossible to get the exact source code from the executable file. This is also the reason why performing Reverse Engineering is considered a very tough task. Deep understanding about the CPU instruction set, calling conventions, memory management, etc is needed.

Speaking of the topic of code analysis, there exists two ways of doing it: Static and Dynamic. The former involves reading code (either high level or Assembly), while the latter involves actually executing, stopping it and analyzing the current program's execution state. The combination of both is usually the best possible scenario for a full analysis of a program.

2.1.2 Debugging

Debugging refers to the process of analyzing a program during execution. It is usually related to the concept of finding and removing errors, but it is not the only use case. Debugging a program involves having the possibility to stop the program at some specific point, being able to read its internal state (variables, memory, registers, etc), jumping to the next instruction and keep comparing the changes between the states.

In this specific project, due to dealing with the Telemetry component which is a part of the Windows Kernel, Kernel debugging was used in order to analyze the state of the Kernel under different moments. Having the possibility of stopping the execution at some specific point and being able to read the state (like the value of CPU registers or the content of certain regions of memory) was crucial to the success of this project. Performing Kernel debugging is sometimes even harder than normal executable debugging because the program that is being used as debugger runs under the target you want to debug (the Kernel). Therefore, is highly probable that a crash may arise due to both having dependencies in each other. That is why, although Windows offers some capabilities to self-debug your Kernel, it is usually recommended to use virtual machines and remote debugging when analyzing Kernels.

2.2 Tools

Various tools were used throughout this work. Although they all had different goals, they all played a central role at some particular point in the investigation. As shown in section 2.1, reverse engineering and debugging were the two main techniques used. In the following sections, not only the tools to perform the aforementioned techniques will be presented, but also other tools that were used to develop and process the information carried out by the Telemetry events.

2.2.1 IDA pro

IDA which stands for Interactive Disassembler is a very powerful tool created by Ilfak Guilfanov which converts executable code to its Assembly representation. It is the most famous and complete disassembler in the market. Being able to run on Windows, Linux and MacOS, IDA has support for multiple different processors and technologies. One of the most important plugins that can be used within IDA is “X-RAYS”. This plugin allows building a pseudo-c (high-level) representation from an executable binary. This is super useful as it reduces the complexity of applying reverse engineering due to being much easier to read pseudo-c code rather than Assembly. IDA is not the only disassembler in the market, it has its competitors such as: Ghidra, Hooper, Binary Ninja and more. This tool was used all along the whole project and was, together with WinDbg, the main used tool. For additional information visit IDA’s documentation[1].

2.2.2 WinDBG

WinDbg is a tool for performing debugging of programs which run on top of the Windows OS created by Microsoft. It is one of the most commonly used debuggers out there,

[1] https://hex-rays.com/ida-pro/

mainly because of being native but also because of their outstanding user interface. Furthermore, WinDbg allows performing debugging through the network (pipes/sockets and more) including Kernel debugging. Additionally, it has support to connect to Microsoft systems and automatically download symbols whenever possible. Finally, WinDbg has a very straightforward scripting language (with full documentation) and even accepts scripts written in other general purpose languages such as Javascript. This scripting feature was leveraged during the entire analysis performed in this work, in order to extract and read specific parts of data. For instance, it helped in the study of the content of structures and portions of memory at certain points which contributed to understanding of Telemetry's workflow. As a consequence, several scripts were developed that helped to automatize most of the tests carried out. Because of all these reasons, the biggest portion of the work carried out within this project was done using WinDbg. For additional information visit Microsoft's debugging documentation[2].

2.2.3 XPERF and Message Analyzer

Although being used for very specific purposes these two Windows tools allowed us to confirm that some hypothesis developed within this project were correct. Xperf is a Windows tool, part of Windows Performance Analyzer[3] tools, that deals with tracing data (control tracing sources and also process its data). Message Analyzer[4] allows, among other things, to display data captured from protocol traffic, events and so on. In our scope, both tools were used together in section 5.2.2, in order to confirm some hypothesis regarding correctness of logged information.

2.3 Windows components

Our objective was to analyze the Telemetry component of the Windows operating system from an internal perspective. In order to log events, Windows Telemetry leverages a kernel framework called Event Tracking for Windows. Although this project focuses on the analysis from an internal perspective, understanding its architecture and main goals before moving on to its assembly code, makes the analysis easier. The following sections will describe both Telemetry and ETW, from a high-level perspective.

2.3.1 Event Tracing for Windows

ETW[5] is an efficient kernel-level tracing facility that provides both, kernel and userland applications, customizable event logging capabilities. In order to make use of them, developers must instrument the source code of the mentioned applications by using functions exposed by the ETW API.

[2] https://docs.microsoft.com/en-us/windows-hardware/drivers/debugger/getting-started-with-windows-debugging

[3] https://docs.microsoft.com/en-us/previous-versions/windows/desktop/xperf/windows-performance-analyzer–wpa-

[4] https://docs.microsoft.com/en-us/message-analyzer/microsoft-message-analyzer-operating-guide

[5] https://docs.microsoft.com/en-us/windows/win32/etw/event-tracing-portal

2.3.1.1 ETW Architecture

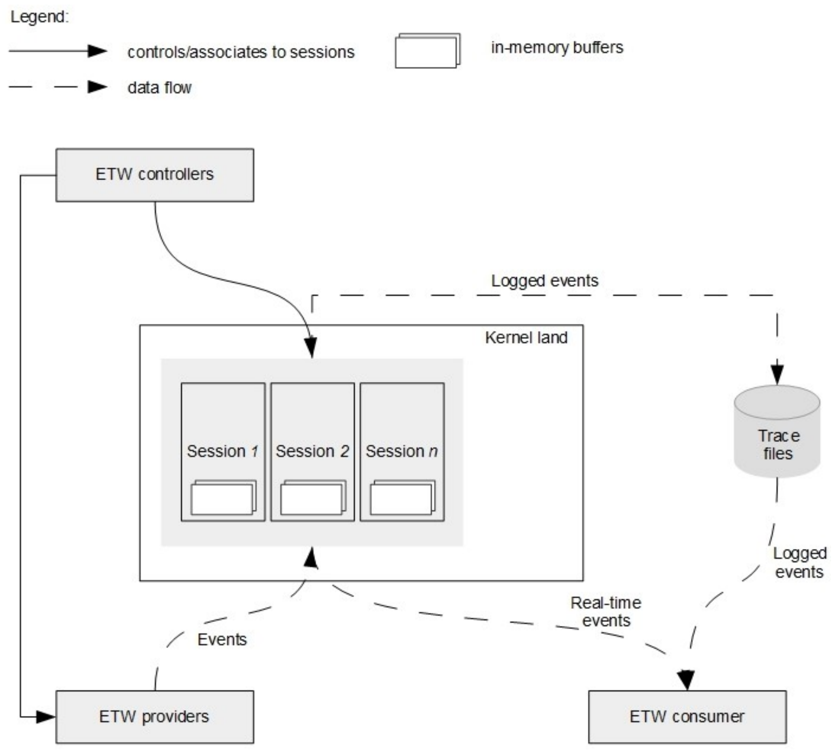


Fig. 2.1: Event Tracing for Windows architecture.

ETW is comprised of several components which are key to its functionality. However there are four important components, as depicted in figure 2.1, worth to highlight:

- Sessions
- Controllers
- Providers
- Consumers

2.3.1.2 ETW Sessions

Sessions are the entities where events are finally stored into. In fact, diving deeper in technical aspects, events are actually stored inside the session's in-memory buffers. The whole ETW architecture supports a maximum of 64 sessions working simultaneously. In order to differentiate one session from another, a "session number" a.k.a. Logger ID is assigned to each of them. Additionally, ETW sessions have a human readable name which can be used to quickly recognize the corresponding session. For example, the name assigned to the Windows Telemetry ETW session is "DiagTrack-Listener".

There are two types of ETW Sessions:

- **NT Kernel:** Used by all the built-in Kernel ETW providers for logging their events.
- **Not NT Kernel:** Must be used by all ETW providers that are not able to log to NT Kernel Logger sessions. For example: Custom ETW providers, providers which

log events that are not considered part of the core Kernel (like TCP/IP user events), etc.

2.3.1.3 ETW Controllers

Controllers are the ones in charge of "administering" ETW sessions. Among their tasks it possible to find:

- Start/Stop ETW Sessions.
- Enable/Disable providers (so they can/can not log events to a particular session).
- Manage the size of the buffer pool.
- Obtain execution statistics (number of buffers used, number of buffers delivered, etc)[6]

2.3.1.4 ETW Providers

ETW Providers are entities (applications, drivers, etc) that contain event tracing instrumentation, in other words, they make use of the ETW API. Each provider has a particular GUID which uniquely identifies it. Within the ETW workflow, these entities are the ones who play the role of writing down events to the ETW sessions. In order to be able to do such a thing, providers have to register themselves against the corresponding ETW session.

2.3.1.5 ETW Consumers

ETW Consumers are the entities who consume/read/parse the data stored in the ETW sessions. An example of an ETW consumer could be the Windows Tool **xperf** (explained in section 2.2.3).

To recap, **Providers** are those who write events to **Sessions** which are later read by the **Consumers**. **Controllers** are the managers/administrators of the Sessions.

2.3.2 Telemetry

Microsoft Telemetry is a Windows 10 component which has the goal of collecting information from the running OS and sending it to Microsoft Backend Remote Servers in a secure way. It collects technical information such as installed software and its performance, hardware specifics and more, in order to analyze it and learn how to improve the overall Windows user experience. All this logic is implemented inside the Windows Dynamic Library *diagtrack.dll* and wrapped up as an OS service running under the name "Diagtrack".

The telemetry data gathered by this service, which is provided by several Windows parts, is written to the Diagtrack related ETW sessions.

Windows Telemetry can be configured in one of the following four levels:

[6] https://msdn.microsoft.com/en-us/library/windows/desktop/aa363881(v=vs.85).aspx

- Security: Logs information needed to help protect Windows. For example: Data about the Telemetry and Connected User Experiences component, the Malicious Software Removal Tool and WindowsDefender.
- Basic: Device information, data related to quality, compatibility of the application, application usage data and security level data.
- Enhanced: Use of Windows, Windows Server, System center and applications. The performance, advanced reliability data and data from basic and security levels.
- Full: All the necessary data to identify problems and help solve them, in addition to data from basic, security and enhanced levels.

As can be deduced, they are ordered from in an ascendent way taking into account the amount and detail information that is being logged. Furthermore, the higher the level of telemetry the larger the amount of ETW Providers that are going to be registered against DiagTrack's ETW session.

3. LABORATORY CONFIGURATION

The goal of this project was to learn about specific Windows components that are usually embedded inside the Windows Kernel. Therefore, being able to perform kernel debugging was key. There are several ways to configure Kernel debugging on Windows Systems[1].

After researching the different possible approaches, it was concluded that the most comfortable, easiest to manage and most reliable way was to build a laboratory of Virtual Machines. Among the numerous advantages we can find:

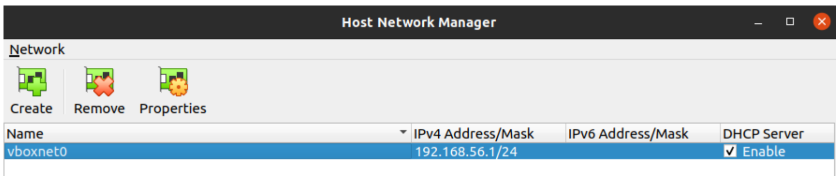
- Independency with host machine: Do not care about the OS of the Host machine.
- Independency Windows versions among VMs: The debuggee (machine to be debugged) and debugger machines can differ in OS versions.
- Flexibility: Being able to restart the one machine without affecting others.

Despite the fact that a particular Virtual Machine Manager tool had to be chosen, the following setup could be adapted to any other VM Manager software.

3.1 Building the setup

Among all possible ways to connect the debuggee machine with the debugger one, it was decided to go after the “network” approach. In other words, the debugger will be performing the Kernel debugging process (sending and receiving data to/from the debuggee) through a network.

After choosing VirtualBox[2] as the VM Manager tool, the network configuration started. Using a default feature of VirtualBox called “Host Network Manager” an internal network was created. In spite of being virtual, “vboxnet0” used IPv4 with its own DHCP server:



Name	IPv4 Address/Mask	IPv6 Address/Mask	DHCP Server
vboxnet0	192.168.56.1/24		✓ Enable

Fig. 3.1: Creation of internal network using the Host Network Manager of VirtualBox

Once this network was built, it was only a matter of connecting both machines (debugger and debuggee) to this network and performing the corresponding configuration to each of them.

[1] https://docs.microsoft.com/en-us/windows-hardware/drivers/debugger/setting-up-kernel-mode-debugging-in-windbg–cdb–or-ntsd

[2] https://www.virtualbox.org/

3.2 Configuring debuggee

The debuggee had to expose their Kernel debugging feature through the network. In order to perform such a thing, the program **bcdedit**[3] was used. This configuration was performed in two steps: Turn on debug and configure a particular setting for allowing remote Kernel debugging.

Turning on debugging can be achieved using the following command line (as administrator):

```
    $> bcdedit /debug on
```

Fig. 3.2: Issued shell command to activate the debug mode inside the debuggee

In order to set particular debugging settings:

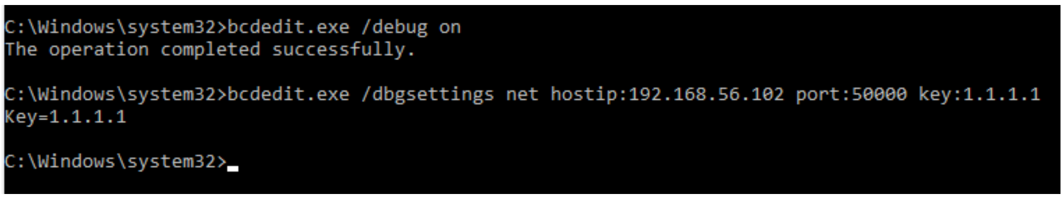
```
    $> bcdedit /dbgsettings net hostip:a.b.c.d port:n key:x.x.x.x
```

Fig. 3.3: Issued shell command to configure debug settings inside the debuggee

Figure 3.3 shows the command[4] that allows the host with IP Address **a.b.c.d** to debug its Kernel through the **net** (network) using port **n** and key **x.x.x.x**. While most command flags are self-explanatory, the **key** is used to encrypt the data transmitted over the channel.

In the specific case of this project, the following values were used:

- **hostip:** Was filled with the debugger's IP address of the vboxnet0 network.
- **port:** Was filled with 50000 (host's port).
- **key:** Was filled with 1.1.1.1. Due to being inside a full controlled environment and network, there was no need to care about encrypting this traffic.



```
C:\Windows\system32>bcdedit.exe /debug on
The operation completed successfully.

C:\Windows\system32>bcdedit.exe /dbgsettings net hostip:192.168.56.102 port:50000 key:1.1.1.1
Key=1.1.1.1

C:\Windows\system32>_
```

Fig. 3.4: Issued shell command to display the Debuggee's debugging configuration

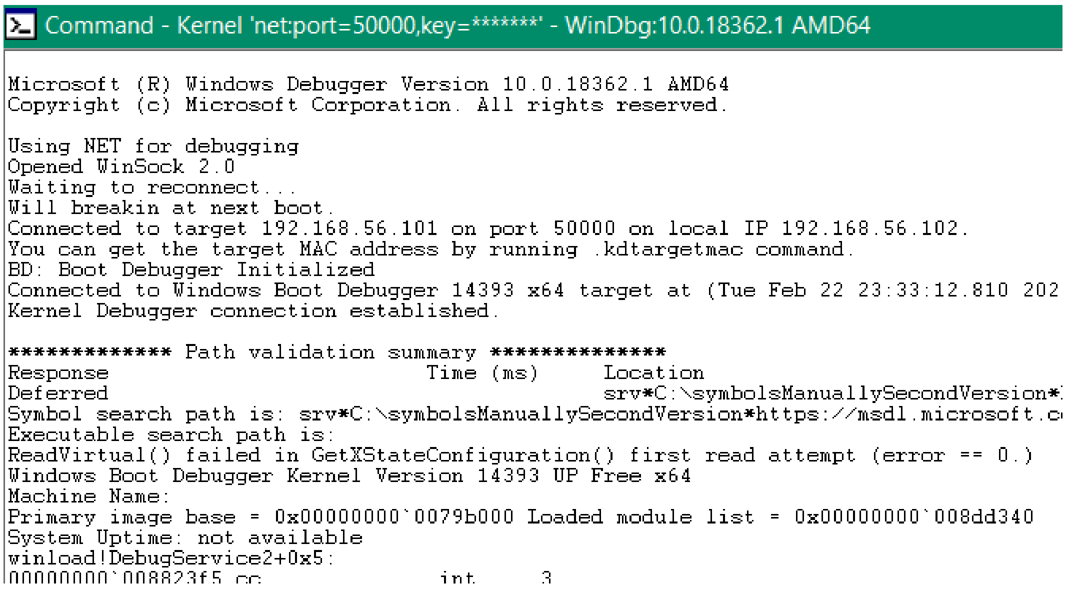
3.3 Configuring debugger

The debugger's configuration is performed inside the WinDBG, due to being the debugging tool to be used.

[3] https://docs.microsoft.com/en-us/windows-hardware/manufacture/desktop/bcdedit-command-line-options?view=windows-11

[4] https://docs.microsoft.com/en-us/windows-hardware/drivers/devtest/bcdedit–dbgsettings

In order to perform Kernel Debugging with WinDBG, it is necessary to go to **File** → **Kernel Debug**. Once inside the configuration panel (**"net"** tab) two of the previously filled items are requested: port number and key. Once this data is completed (and debuggee machine restarted) it will be possible to break as soon as the debuggee starts its boot process..



Command - Kernel 'net:port=50000,key=*******' - WinDbg:10.0.18362.1 AMD64

```
Microsoft (R) Windows Debugger Version 10.0.18362.1 AMD64
Copyright (c) Microsoft Corporation. All rights reserved.

Using NET for debugging
Opened WinSock 2.0
Waiting to reconnect...
Will breakin at next boot.
Connected to target 192.168.56.101 on port 50000 on local IP 192.168.56.102.
You can get the target MAC address by running .kdtargetmac command.
BD: Boot Debugger Initialized
Connected to Windows Boot Debugger 14393 x64 target at (Tue Feb 22 23:33:12.810 202
Kernel Debugger connection established.

************** Path validation summary **************
Response                         Time (ms)     Location
Deferred                                       srv*C:\symbolsManuallySecondVersion*
Symbol search path is: srv*C:\symbolsManuallySecondVersion*https://msdl.microsoft.c
Executable search path is:
ReadVirtual() failed in GetXStateConfiguration() first read attempt (error == 0.)
Windows Boot Debugger Kernel Version 14393 UP Free x64
Machine Name:
Primary image base = 0x00000000`0079b000 Loaded module list = 0x00000000`008dd340
System Uptime: not available
winload!DebugService2+0x5:
00000000`008823f5 cc              int     3
```

Fig. 3.5: WinDBG screen inside the debugger machine as soon as the debuggee connects to it.

4. PREVIOUS WORK

Previous work related to Windows Telemetry has always been focused on either two topics:

- Event analysis: Discover which events have been sent to Microsoft Backend servers, and inspect their content.
- Traffic Analysis: Network analysis of the events remotely sent to the backend servers.

Lot of academic and non-academic efforts have been invested into understanding what kind of information was being logged inside the Telemetry events (Event Analysis). This had raised a lot of concerns due to its relation with Windows users privacy and legal topics such as regulations. A clear example of this could be the General Data Protection Regulation (GDPR) which, in simple terms, provides a legal framework for data protection in the EU[1]. At the same time, analysis of information being sent to Microsoft backend servers has captured a lot of attention. There are two reasons behind this. First, the data is sent remotely and therefore the channel being used should be secure. That is why some projects were devoted to ensuring that the information being sent was encrypted. Second, "untrusted" data was being analyzed and parsed by Microsoft Backend servers. As a consequence, a potential risk could be raised if those servers were vulnerable to some sort of attack while trying to parse the data sent. Some examples of projects devoted to performing the aforementioned tasks are [1][2][3].

Our research, despite the fact of briefly dealing with events' content, is focused on different aspects. This project will analyze the technical matters of Windows Telemetry, specially focusing on how the process of logging events is carried out inside the Windows Kernel itself. In other words, this work will not focus on studying the data being sent nor the channel used to send it. On the contrary, it will focus on applications, kernel structures, internal processes and other entities that are somehow involved in the process of logging after some particular event has ocurred.

[1] https://gdpr.eu/

5. COMPONENT ANALYSIS AND METHODOLOGY

The main objective of this work is the analysis of Telemetry component in Microsoft Windows Operating System. The best and most accurate option to study this component would have been to analyze its source code. Unluckily this is not possible as the Windows kernel is not open source. However, it was still possible to reverse engineer the Windows kernel to understand how Telemetry works. Several files (dynamic libraries, executables, drivers) had to be reversed and analyzed. Nonetheless, there was one file that was the main focus: **ntoskrnl.exe**. This binary held the actual implementation of the Windows Kernel.

The following sections will depict different challenges faced and achievements accomplished during the analysis.

5.1 Understanding how Telemetry makes use of ETW

When Windows OS boots, multiple sessions are created inside ETW. Among them there is one related to Telemetry: DiagTrack. As explained in 2.3.1, every session has "providers" -those entities that actually provide information to the session-. In order to understand how Telemetry made use of ETW, it was important to learn about DiagTrack's session providers. At this point, the answers for the following questions were unknown:

1. Who are they?
2. Where are they?
3. What information are they logging?

Windows allowed to query the list of providers registered to a particular session by executing the following powershell command:

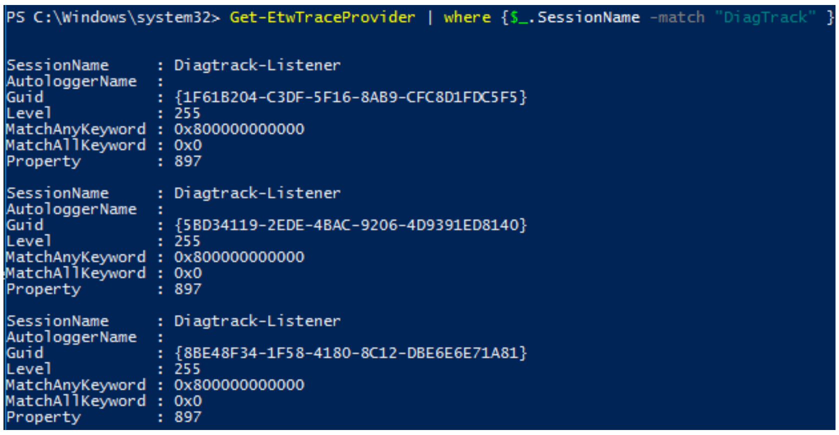
```
$> Get-EtwTraceProvider | where {$_.SessionName -match "<SESSION_NAME>"}
```

Fig. 5.1: Powershell command to list ETW providers registered against a particular session.

For each provider registered to the queried session, the following information is outputted:

- Provider's GUID.
- Session's name.
- Some extra data that was not relevant at this stage.

For example:



```
PS C:\Windows\system32> Get-EtwTraceProvider | where {$_.SessionName -match "DiagTrack" }

SessionName     : Diagtrack-Listener
AutologgerName  :
Guid            : {1F61B204-C3DF-5F16-8AB9-CFC8D1FDC5F5}
Level           : 255
MatchAnyKeyword : 0x800000000000
MatchAllKeyword : 0x0
Property        : 897

SessionName     : Diagtrack-Listener
AutologgerName  :
Guid            : {5BD34119-2EDE-4BAC-9206-4D9391ED8140}
Level           : 255
MatchAnyKeyword : 0x800000000000
MatchAllKeyword : 0x0
Property        : 897

SessionName     : Diagtrack-Listener
AutologgerName  :
Guid            : {8BE48F34-1F58-4180-8C12-DBE6E6E71A81}
Level           : 255
MatchAnyKeyword : 0x800000000000
MatchAllKeyword : 0x0
Property        : 897
```

Fig. 5.2: Output of Get-EtwTraceProvider Powershell command for DiagTrack session

With this list, the question **Who are they?** seemed to be answered. Still, two questions remained.

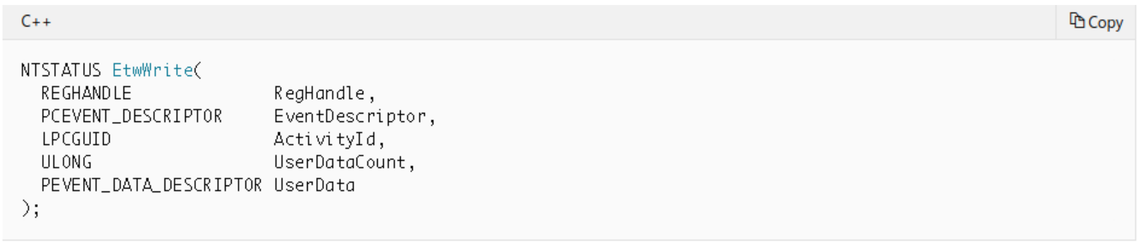
At that point, an interesting idea came up: To answer **Where are they running?** and **What are they logging?** it could be useful to hook into the exact moment when any of those providers are going to write to the DiagTrack's session. In other words, it could be useful to set a breakpoint in the function that performs the write to the DiagTrack's session. Once the breakpoint is hit, the following information could be extracted:

1. The piece of code that triggered the write (by inspecting the function's call stack). In other words, identify who was writing.
2. The actual content of the log being written.

That was when debugging (2.1.2) came into play. Analyzing the symbols exposed by the ntoskrnl binary, the function **EtwWrite** was found. This function seemed to be the one in charge of carrying out the process of writing inside sessions. Nonetheless, it was not the best option to set a breakpoint at this function as every provider of the system (not necessarily related to DiagTrack) could use it. It was necessary to find a way of only detecting the writes performed by DiagTrack's providers.

The powershell command (5.1) returned information for each registered provider. Part of that information was the GUID. Due to the previous objective being to filter writes only from providers that were registered against the DiagTrack Session, it could be useful to set a conditional breakpoint inside function **EtwWrite** and try to check if the GUID provided was of interest.

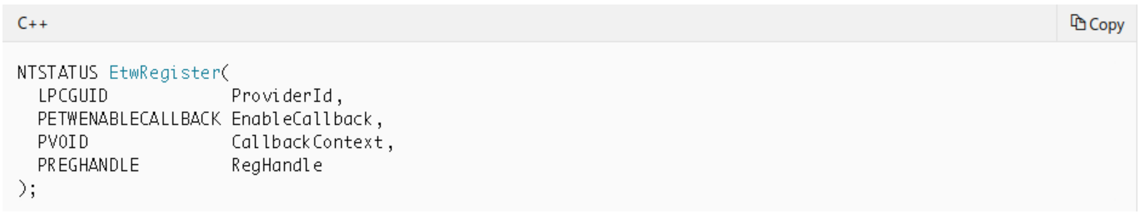
Unfortunately, this strategy had one minor issue. The function (**EtwWrite**) had five parameters and none of them would show the GUID directly:



```
NTSTATUS EtwWrite(
  REGHANDLE              RegHandle,
  PCEVENT_DESCRIPTOR     EventDescriptor,
  LPCGUID                ActivityId,
  ULONG                  UserDataCount,
  PEVENT_DATA_DESCRIPTOR UserData
);
```

Fig. 5.3: Documentation for EtwWrite function [1].

The first parameter was the registration handler. This object was returned once the provider executed the registration (**EtwRegister**) successfully. Taking a deeper look at **EtwRegister** it was possible to observe that it received the GUID as parameter:



```
NTSTATUS EtwRegister(
  LPCGUID            ProviderId,
  PETWENABLECALLBACK EnableCallback,
  PVOID              CallbackContext,
  PREGHANDLE         RegHandle
);
```

Fig. 5.4: Documentation for EtwRegister function [2].

This finding basically meant that having only one breakpoint in **EtwWrite** was not going to be enough as information from **EtwRegister** was also needed. In other words, to understand if the write was being done by a provider registered against the DiagTrack session it was necessary to:

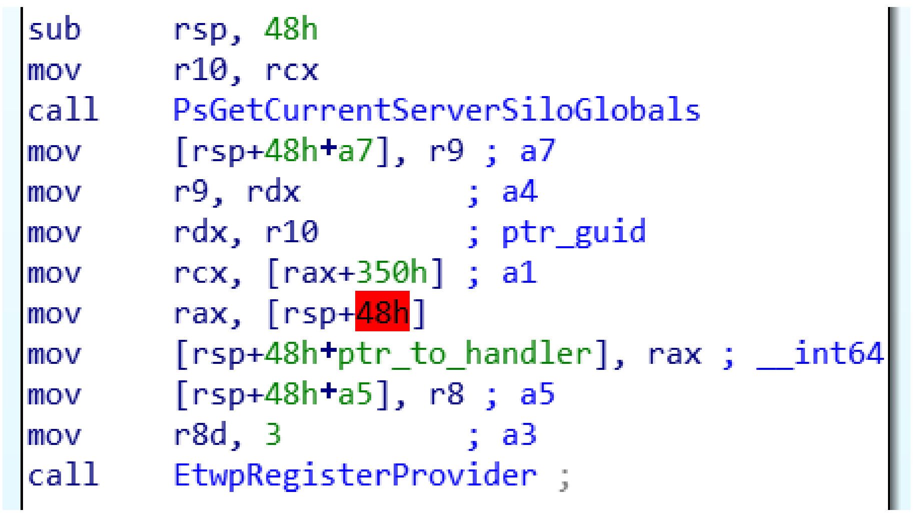
1. Extract the whole list of providers registered against the DiagTrack session.
2. Intercept all the **EtwRegister** executions and check if the GUID being used was inside the list.
3. If it was, save the handler.
4. Intercept all the **EtwWrite** executions and check if the handler being used is one of the stored handlers.
5. If it was, the provider that is writing is attached to the DiagTrack session.

Even though this strategy seemed to be theoretically promising, it was necessary to understand how to actually carry out each of these steps. Further sections will depict this process.

5.1.1 Reversing registration process

As mentioned in section 2.3.1, whenever a provider wants to register itself against a particular session it has to call the function **EtwRegister**. Because of this, the first step was to analyze the behavior of this function using **IDA**(2.2.1).

As can be seen in figure 5.5, the only interesting action being performed by **EtwRegister** was a call to another function named **EtwpRegisterProvider**.



```
sub     rsp, 48h
mov     r10, rcx
call    PsGetCurrentServerSiloGlobals
mov     [rsp+48h+a7], r9 ; a7
mov     r9, rdx          ; a4
mov     rdx, r10         ; ptr_guid
mov     rcx, [rax+350h] ; a1
mov     rax, [rsp+48h]
mov     [rsp+48h+ptr_to_handler], rax ; __int64
mov     [rsp+48h+a5], r8 ; a5
mov     r8d, 3           ; a3
call    EtwpRegisterProvider ;
```

Fig. 5.5: Basic block of function *EtwRegister*, obtained using IDA's disassembler, shown in order to highlight offsets and calls to key functions like *PsGetCurrentServerSiloGlobals* and *EtwpRegisterProvider*.

A quick analysis of the latter function showed that it was the function holding the actual implementation of the registration process. However, due to the lack of documentation, it was necessary to understand more in depth what was actually happening inside it.

Following sections will present a detailed description of reversing (only the interesting parts for this research) **EtwpRegisterProvider**. To make it easier, it will be divided in different parts:

- 1. Understanding the layout of the function.
- 2. Check if a GUID for this provider already exists.
- 3. If not, create a new one.
- 4. Return the handler.

```
// 1. Understanding the layout of the function
signed __int64 __fastcall EtwpRegisterProvider(__int64 a1, _QWORD *a2, int a3,
    void (__fastcall *a4)(ULONG_PTR, __int64, __int128 *,__int64), __int64 a5,
    __int64 a6, __int64 *a7){

[..]

// 2. Check if a GUID for this provider already exists.
ptr_guid_entry = (char *)EtwpFindGuidEntryByGuid(etw_silo, ptr_guid, 0);

// 3. If not, create a new one
if ( ptr_guid_entry || (ptr_guid_entry = EtwpAddGuidEntry(ptr_etw_silo_cpy2,
    ptr_guid_cpy, 0)) != 0i64 )
{
  v15 = __readgsqword(0x188u);
  --*(_WORD *)(v15 + 484)

  [..]

  // 4. Return the handler.
  v35 = EtwpAddKmRegEntry((ULONG_PTR)ptr_guid_entry, v10, (__int64)v9, a5,
      (__int64)&ptr_handler);
  v20 = v35;

  [..]

}
return v20;
```

Fig. 5.6: Pseudocode representation of an interesting part of *EtwpRegisterProvider* function. This snippet was created using IDA's disassembler and decompiler output. It illustrates the four different parts that will be analyzed.

5.1.1.1 Understanding the layout of the function

EtwpRegisterProvider received seven parameters:

```
signed __int64 __fastcall EtwpRegisterProvider(__int64 a1, _QWORD *a2,
      int a3, void (__fastcall *a4)(ULONG_PTR, __int64, __int128 *,
      __int64), __int64 a5, __int64 a6, __int64 *a7)
```

Usually when performing reverse engineering it is not necessary to understand every tiny detail but only the key points that are important to meet the proposed goals. This was not the exception.

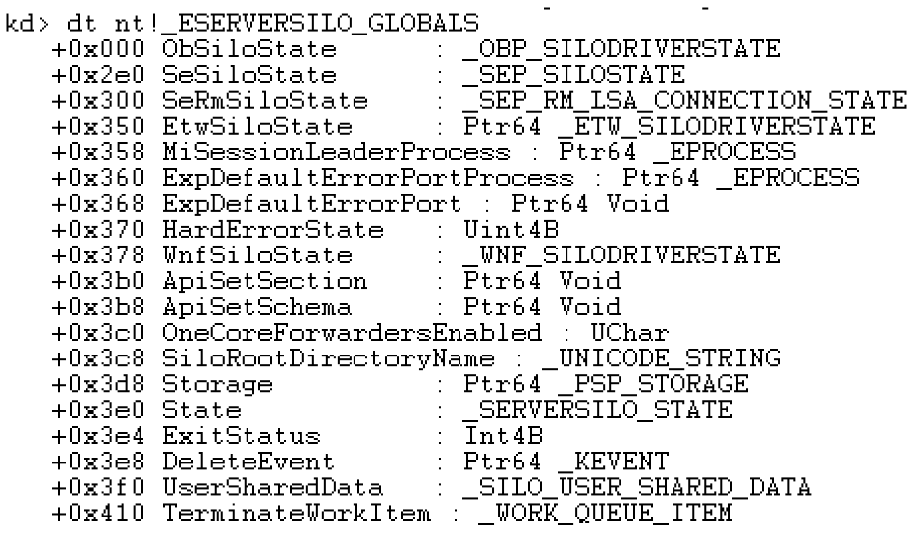
The main focus here was not to understand how the registration process fully worked but just to get an idea of it plus get to know the relation between GUID and registration handler.

After analyzing **EtwpRegisterProvider** it was possible to conclude that:

1. **a1**: Was the pointer to an unknown structure.

2. **a2**: Was the pointer to the GUID structure.
3. **a7**: Was the address where the pointer to the registration handler would be placed (can be thought as “function output”).

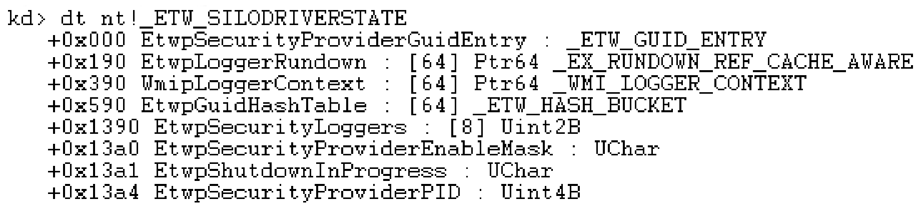
What is this **a1** structure? The figure 5.5 shows that before calling **EtwpRegister-Provider**, the function **PsGetCurrentServerSiloGlobals** is invoked. This latter one returns a pointer to a structure S of type **_ESERVERSILO_GLOBALS**.



```
kd> dt nt!_ESERVERSILO_GLOBALS
   +0x000 ObSiloState      : _OBP_SILODRIVERSTATE
   +0x2e0 SeSiloState      : _SEP_SILOSTATE
   +0x300 SeRmSiloState    : _SEP_RM_LSA_CONNECTION_STATE
   +0x350 EtwSiloState     : Ptr64 _ETW_SILODRIVERSTATE
   +0x358 MiSessionLeaderProcess : Ptr64 _EPROCESS
   +0x360 ExpDefaultErrorPortProcess : Ptr64 _EPROCESS
   +0x368 ExpDefaultErrorPort : Ptr64 Void
   +0x370 HardErrorState   : Uint4B
   +0x378 WnfSiloState     : _WNF_SILODRIVERSTATE
   +0x3b0 ApiSetSection    : Ptr64 Void
   +0x3b8 ApiSetSchema     : Ptr64 Void
   +0x3c0 OneCoreForwardersEnabled : UChar
   +0x3c8 SiloRootDirectoryName : _UNICODE_STRING
   +0x3d8 Storage          : Ptr64 _PSP_STORAGE
   +0x3e0 State            : _SERVERSILO_STATE
   +0x3e4 ExitStatus       : Int4B
   +0x3e8 DeleteEvent      : Ptr64 _KEVENT
   +0x3f0 UserSharedData   : _SILO_USER_SHARED_DATA
   +0x410 TerminateWorkItem : _WORK_QUEUE_ITEM
```

Fig. 5.7: Illustration of structure *_ESERVERSILO_GLOBALS* (S) using WinDBG’s *dt* command.

However, the first parameter provided to **EtwpRegisterProvider** was not the pointer to S but the pointer to another structure *S_2* of type **_ETW_SILODRIVERSTATE** which happens to be part of S, situated at offset 0x350.

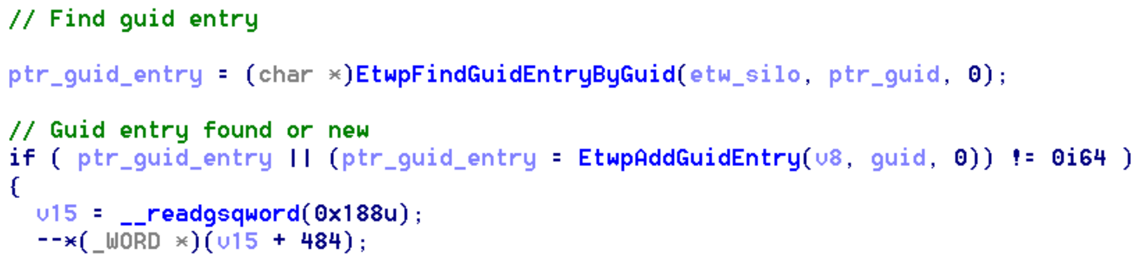


```
kd> dt nt!_ETW_SILODRIVERSTATE
   +0x000 EtwpSecurityProviderGuidEntry : _ETW_GUID_ENTRY
   +0x190 EtwpLoggerRundown : [64] Ptr64 _EX_RUNDOWN_REF_CACHE_AWARE
   +0x390 WmipLoggerContext : [64] Ptr64 _WMI_LOGGER_CONTEXT
   +0x590 EtwpGuidHashTable : [64] _ETW_HASH_BUCKET
   +0x1390 EtwpSecurityLoggers : [8] Uint2B
   +0x13a0 EtwpSecurityProviderEnableMask : UChar
   +0x13a1 EtwpShutdownInProgress : UChar
   +0x13a4 EtwpSecurityProviderPID : Uint4B
```

Fig. 5.8: Illustration of structure *_ETW_SILODRIVERSTATE* (*S_2*) using WinDBG’s *dt* command.

With this information it was possible to conclude that **a1** will point to a global structure holding configurations, settings and information in general directly related with the **ETW** framework. **a2** and **a7** will hold pointers to a GUID and to a place where a registration handler will be stored afterwards. The information gathered from this analysis was enough to move forward as having those two pieces of information would allow us to filter only the interesting function calls to *EtwWrite*.

Once inside the *EtwpRegisterProvider* function, after performing some sanity checks, it tries to get the guid entry related to the GUID provided. If it does not exist, it will create one.



```
// Find guid entry

ptr_guid_entry = (char *)EtwpFindGuidEntryByGuid(etw_silo, ptr_guid, 0);

// Guid entry found or new
if ( ptr_guid_entry || (ptr_guid_entry = EtwpAddGuidEntry(v8, guid, 0)) != 0i64 )
{
  v15 = __readgsqword(0x188u);
  --*(_WORD *)(v15 + 484);
```

Fig. 5.9: Pseudocode representation of an interesting part of *EtwpRegisterProvider* function. This snippet was created using IDA's disassembler and decompiler output. It illustrates the second and third parts that will be analyzed.

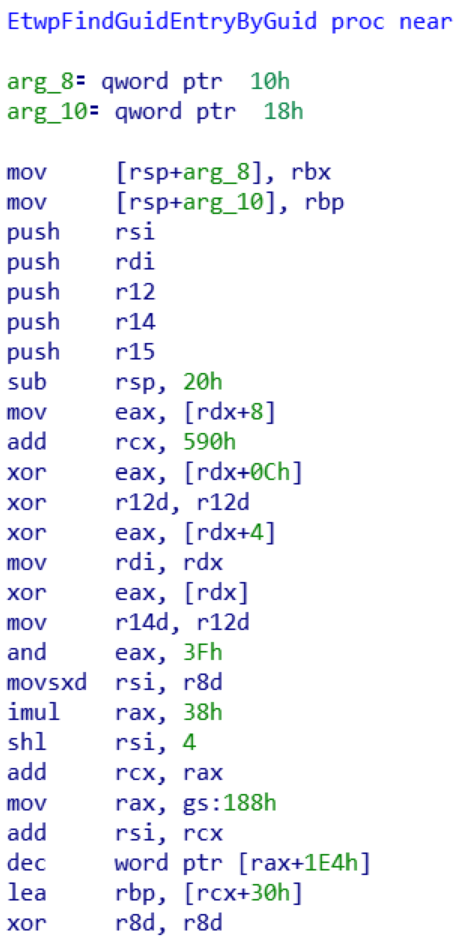
5.1.1.2 Check if GUID for this provider already exists

This part will be focused on understanding how the process of recovering the already existing "GUID entry" works.

The action of recovering is performed by a particular function called **EtwpFindGuidEntryByGuid**:

```
ptr_guid_entry = (char *)EtwpFindGuidEntryByGuid(etw_silo, ptr_guid, 0);
```

As can be inferred from the previous line, two important parameters were provided: the **ETWSILODRIVERSTATE**(a1) structure *S* and the pointer to the GUID *ptr_guid*.



```
EtwpFindGuidEntryByGuid proc near

arg_8= qword ptr  10h
arg_10= qword ptr  18h

mov     [rsp+arg_8], rbx
mov     [rsp+arg_10], rbp
push    rsi
push    rdi
push    r12
push    r14
push    r15
sub     rsp, 20h
mov     eax, [rdx+8]
add     rcx, 590h
xor     eax, [rdx+0Ch]
xor     r12d, r12d
xor     eax, [rdx+4]
mov     rdi, rdx
xor     eax, [rdx]
mov     r14d, r12d
and     eax, 3Fh
movsxd  rsi, r8d
imul    rax, 38h
shl     rsi, 4
add     rcx, rax
mov     rax, gs:188h
add     rsi, rcx
dec     word ptr [rax+1E4h]
lea     rbp, [rcx+30h]
xor     r8d, r8d
```

Fig. 5.10: First basic block of *EtwpFindGuidEntryByGuid*, obtained using IDA's disassembler. It highlights the operations that are performed in order to carefully analyze them.

Figure 5.10 depicts how the function gets the guid entry related to the provider (if it exists): *rcx* holds the pointer to *S* and *rdx* holds *ptr_guid*. Let's analyze this function deeper.

The first highlight is the *add* function which stores in *rcx* the pointer to the structure stored at the offset $0x590$ of *S*. Going back to the structure layout of *S* (figure 5.8), it can be appreciated that at the offset $0x590$, the structure *EtwpGuidHashTable* of type *_ETW_HASH_BUCKET[64]* is present. Figure 5.12 depicts its layout.

Just before the *add* function, *eax* is filled with the content of the address $rdx + 8$. *rdx* held the *ptr_guid*, meaning that *eax* will have the third group of 4 bytes inside of the guid structure. Why the third? Because the offset was 8. Why 4 bytes? Because the register *eax* (32 bits) was used.

In the following lines, the value of *eax* is being constantly modified by xoring it successively with the different group of 4 bytes that compose the guid structure[3]. After performing these successive xor operations, a boolean-and is applied against *eax* (result of

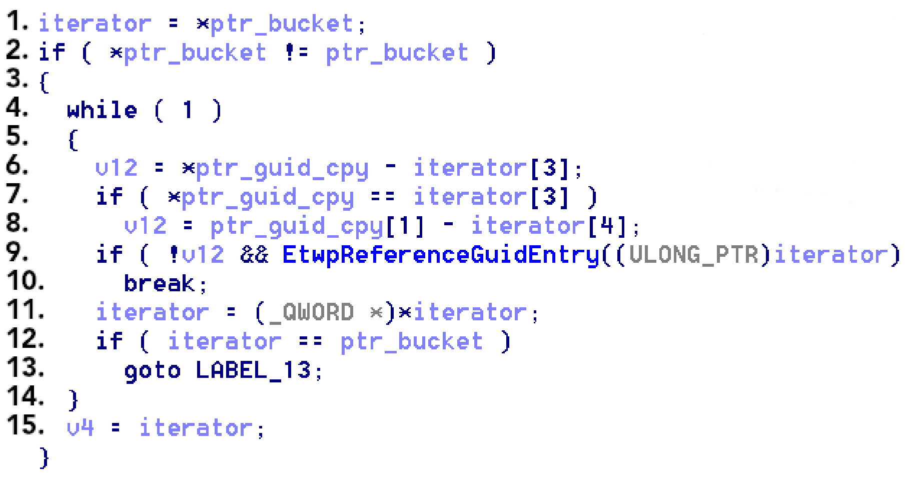
[3] Sometimes the structures were not documented at all. Sometimes they were, but it was not possible to find it until some kind of clue pointing to it was found. So far the layout of the structure pointed by *ptr_guid* was unknown, however from this function it was possible to conclude that the structure had a size of 16 bytes.

xoring) using a mask of $0x3f$. This mask will set all *eax* bits to 0 with the exception of the last 6 that will remain having its actual value. The reason to do this is because $2^6 = 64$. In other words, this mask is making the xoring result fit into the range of a valid bucket index. Afterwards, multiplies the result against $0x38$ (size of *_ETW_HASH_BUCKET* structure). Finally, the value of $rax(eax)$ is added to *rcx* which has the pointer to the *EtwpGuidHashTable* structure.

Writing the aforementioned function in a pseudo-code style (*ptr* is a short version of “pointer”):

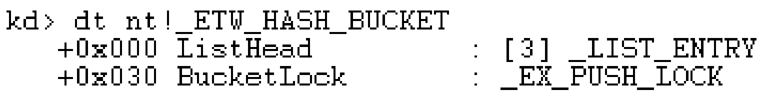
```
xor_guid_parts = ptr_guid[0] ^ ptr_guid[1] ^ ptr_guid[2] ^ ptr_guid[3]
ptr_hash_table = ptr_S + 0x590
ptr_bucket = ptr_hash_table + 0x38 * ((xor_guid_parts) & 0x3F)
```

Therefore, *ptr_bucket* is basically a pointer to a particular bucket inside the *EtwpGuidHashTable* calculated based on the GUID of the provider[4]. Once this value is obtained, a “look up” inside the structure is carried out in the following way:



```
1. iterator = *ptr_bucket;
2. if ( *ptr_bucket != ptr_bucket )
3. {
4.   while ( 1 )
5.   {
6.     v12 = *ptr_guid_cpy - iterator[3];
7.     if ( *ptr_guid_cpy == iterator[3] )
8.       v12 = ptr_guid_cpy[1] - iterator[4];
9.     if ( !v12 && EtwpReferenceGuidEntry((ULONG_PTR)iterator)
10.      break;
11.    iterator = (_QWORD *)*iterator;
12.    if ( iterator == ptr_bucket )
13.      goto LABEL_13;
14.  }
15.  v4 = iterator;
   }
```

Fig. 5.11: Pseudocode snippet of *EtwpFindGuidEntryByGuid* function, obtained using IDA’s decompiler. It highlights the way the GUID is being searched once the correct bucket was selected.



```
kd> dt nt!_ETW_HASH_BUCKET
   +0x000 ListHead         : [3] _LIST_ENTRY
   +0x030 BucketLock       : _EX_PUSH_LOCK
```

Fig. 5.12: Illustration of structure *_ETW_HASH_BUCKET* using WinDBG’s *dt* command.

At first an iterator is built. This iterator will point initially to the Flink of the first list entry[5] of the bucket (figure 5.11). Line **2** will capture the special case where the list is empty. In that particular case, the whole cycle will be skipped and the **LABEL_13** (routine to exit, which is not displayed in the figure) will be executed. It is worth mentioning

[4] There was also an additional value involved in the calculation of the bucket. However, in this particular context, the value was not taken into account as it was always 0.

[5] https://docs.microsoft.com/en-us/windows/desktop/api/ntdef/ns-ntdef-_list_entry

that this routine executes a return statement (line **15**) with the value of the variable $v4$ (which is initially defined as 0).

If the list is not empty, the first operation which is carried out is a subtraction between the first quadword of the GUID and a value of *iterator[3]* (line **6**). Due to the variable *iterator* is defined as a 8-bytes pointer, *iterator[3]* will point to the offset $0x18$ of the structure stored inside the Flink. In the case that both values are the same, the second comparison (between the second quadword of the GUID and the *iterator[4]*) is carried out as depicted in lines **7 and 8**.

At this point some things can be concluded:

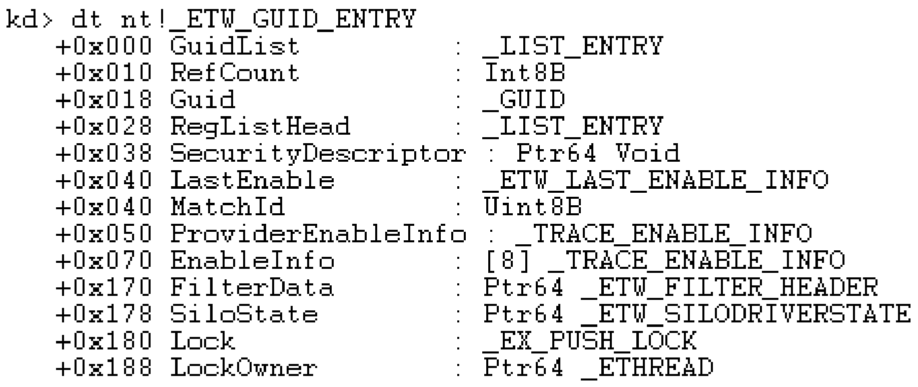
- The cycle is iterating a double-linked-list which holds a particular structure T.
- T has the GUID of the provider stored at offset $0x18$.
- Again, it seems that the GUID is 16 bytes long.
- From the function name it can be inferred that T is a structure that represents the GUID entry.

Moving forward with the code analysis, if some of the comparisons failed, the iterator changes its values to the next one in the list (line **11**). Before continuing, it ensures that the cycle is not finished by checking if the actual value of the iterator is the same one used as the starting point (line **12**). If they are equal, the exit routine is executed meaning that the return value will be 0.

If both comparisons are equal (the GUID of the provider and the one stored in T are the same), a function called *EtwpReferenceGuidEntry* with the current value of the iterator as parameter, is called (line **9**). After this execution, the cycle is finished by the break statement. However, before executing the exit routine, the value of $v4$ is filled up with the value of the *iterator* (line **15**), meaning that the return value will be pointer to the guid entry related to the GUID of the provider. The *EtwpReferenceGuidEntry* function just makes some sanity checks to the pointer that are not relevant for our investigation.

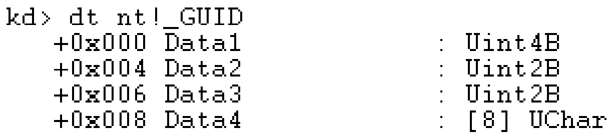
Therefore, to summarize, it is possible to say that:
The function *EtwpFindGuidEntryByGuid* looks for a particular structure (most probably called guid entry), which is stored inside a double-linked-list of a bucket inside the *EtwpGuidHashTable* of the *_ETW_SILODRIVERSTATE*, based on doing some mathematical operations with the GUID of the provider.

After finishing with this analysis, the documentation of the guid entry structure was found:



```
kd> dt nt!_ETW_GUID_ENTRY
   +0x000 GuidList          : _LIST_ENTRY
   +0x010 RefCount          : Int8B
   +0x018 Guid              : _GUID
   +0x028 RegListHead       : _LIST_ENTRY
   +0x038 SecurityDescriptor : Ptr64 Void
   +0x040 LastEnable        : _ETW_LAST_ENABLE_INFO
   +0x040 MatchId           : Uint8B
   +0x050 ProviderEnableInfo : _TRACE_ENABLE_INFO
   +0x070 EnableInfo        : [8] _TRACE_ENABLE_INFO
   +0x170 FilterData        : Ptr64 _ETW_FILTER_HEADER
   +0x178 SiloState         : Ptr64 _ETW_SILODRIVERSTATE
   +0x180 Lock              : _EX_PUSH_LOCK
   +0x188 LockOwner         : Ptr64 _ETHREAD
```

Fig. 5.13: Illustration of structure *_ETW_GUID_ENTRY* using WinDBG's *dt* command.



```
kd> dt nt!_GUID
   +0x000 Data1             : Uint4B
   +0x004 Data2             : Uint2B
   +0x006 Data3             : Uint2B
   +0x008 Data4             : [8] UChar
```

Fig. 5.14: Illustration of structure *_ETW_GUID* using WinDBG's *dt* command.

Luckily, all the previous guesses made, were correct:

- The guid entry (now *_ETW_GUID_ENTRY*) had the GUID of at offset $0x18$ (figure 5.13)
- The GUID was a structure of 16 bytes long (figure 5.14)

5.1.1.3 If GUID not found, create a new one

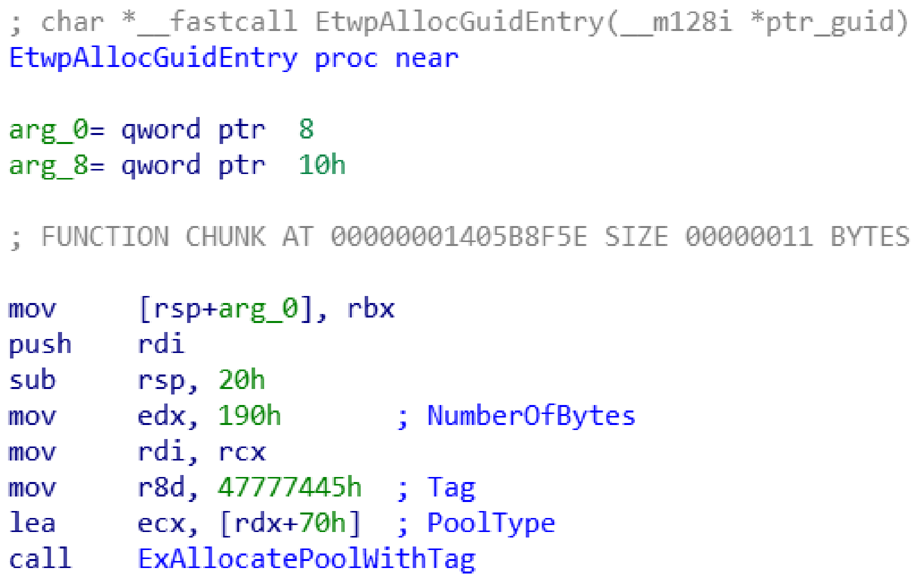
Previous section detailed how the process was to find an already existing guid entry based on the GUID of the provider. This one will explain the process of creating a new guid entry.

From figure 5.9 can be observed that the function in charge of this part is the function *EtwpAddGuidEntry*:

```
ptr_guid_entry = EtwpAddGuidEntry(ptr_etw_silo_cpy2, ptr_guid_cpy, 0)
```

As can be inferred from the previous line, two important parameters were provided: the pointer to the _ETW_SILODRIVERSTATE structure (for simplicity will be called *ptr_etw_silo* instead of *ptr_etw_silo_cpy2*) and the pointer to the GUID (for simplicity will be called *ptr_guid* instead of *ptr_guid_cpy*).

One of the first lines of *EtwpAddGuidEntry*, calls another function named *EtwpAllocGuidEntry*. As it can be quickly inferred from the name, it basically allocates a certain amount of memory inside the heap to be used by the guid entry afterwards and returns the pointer to it. The allocation part happens in the first basic block of *EtwpAllocGuidEntry*:



```
; char *__fastcall EtwpAllocGuidEntry(__m128i *ptr_guid)
EtwpAllocGuidEntry proc near

arg_0= qword ptr  8
arg_8= qword ptr  10h

; FUNCTION CHUNK AT 00000001405B8F5E SIZE 00000011 BYTES

mov     [rsp+arg_0], rbx
push    rdi
sub     rsp, 20h
mov     edx, 190h       ; NumberOfBytes
mov     rdi, rcx
mov     r8d, 47777445h  ; Tag
lea     ecx, [rdx+70h]  ; PoolType
call    ExAllocatePoolWithTag
```

Fig. 5.15: Interesting basic block from function *EtwpAllocGuidEntry*, obtained using IDA's disassembler, in charge of performing the allocation of the necessary memory.

As can be observed in figure 5.15 the Windows function *ExAllocatePoolWithTag*[6] is called with the following parameters:

- **PoolType**: 0x200 (**NonPagedPoolNx**). This value indicates that the system memory allocated will be non pageable and not executable[7]. This is really good from a security perspective, as it means that if somehow malicious code ends up being inside this memory region, it will not be neither executable nor pageable and therefore difficult to leverage it for a successful exploitation.
- **NumberOfBytes**: 0x190. This value is the size of the structure *_ETW_GUID_ENTRY* (figure 5.13).
- **Tag**: "0x47777445". According to the documentation just a four character long is used as a pool tag. Due to being specified in reverse order: 0x45747747 → "EtwG".

Therefore, as it was thought, *EtwpAllocGuidEntry* allocs the necessary memory for holding the *_ETW_GUID_ENTRY* structure and returns a heap pointer to it.

The remaining code of *EtwpAddGuidEntry* is devoted to populating and adjusting some parts of related structures. Some key points about it:

- A guid entry related to this GUID is looked up inside the guid entries double-linked list using the same technique as the one used in *EtwpFindGuidEntryByGuid*. If a structure is found, the pointer is freed.
- Only three parts of the *_ETW_GUID_ENTRY* structure are populated at this point:

[6] Documentation: https://docs.microsoft.com/en-us/windows-hardware/drivers/ddi/content/wdm/nf-wdm-exallocatepoolwithtag

[7] Documentation: https://docs.microsoft.com/en-us/windows-hardware/drivers/ddi/content/wdm/ne-wdm-_pool_type

1. The pointer to the previous guid entry in the double-linked list (offset 0x0)
2. The pointer to the following guid entry in the double-linked list (offset 0x8)
3. The pointer to the SILO STATE (offset 0x178)

To summarize, once *EtwpAllocGuidEntry* is executed, the pointer to heap memory holding the *_ETW_GUID_ENTRY* structure is returned. The next step is to insert this entry into *EtwpGuidHashTable*. To perform that action, first it looks for the correct place to insert it as depicted previously.

5.1.1.4 Return the handler

Going back to what the figure 5.4 states, the 4th parameter of *EtwRegister* it is something of type *PREGHANDLE*. Although it is not very clear, this parameter is the output of the function (usually referred as an "out" type of parameter). Furthermore, as mentioned previously, the real registration logic is implemented by *EtwpRegisterProvider* and therefore the output of *EtwRegister* is none other than the output of *EtwpRegisterProvider*.

Despite the fact the provider's GUID existed previously or not, at this point it exists a pointer to a *_ETW_GUID_ENTRY* structure holding its data and already inside the main structures of ETW. Once the code achieves this point, the next step is basically get the handler.

Just right after the pointer to the *_ETW_GUID_ENTRY* is found, the function *EtwpAddKmRegEntry* is called:

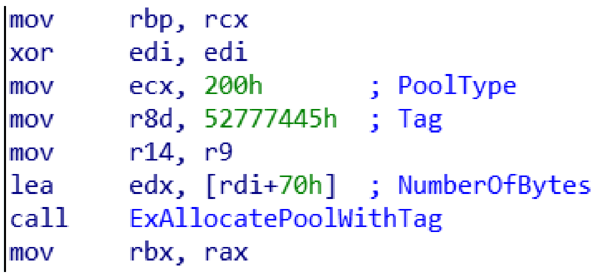
```
__int64 __usercall EtwpAddKmRegEntry(ULONG_PTR a1, int a2, __int64 a3,
__int64 a4, __int64 a5)
```

where :

1. **a1**: Is the pointer to *_ETW_GUID_ENTRY*, called *ptr_guid*.
2. **a5**: Is the memory address provided by *EtwWrite* (and afterwards by *EtwpRegisterProvider*) where the handler should be placed.

The remaining parameters are not interesting for the sake of our research.

Once inside *EtwpAddKmRegEntry*, the first important lines were:



```
mov     rbp, rcx
xor     edi, edi
mov     ecx, 200h         ; PoolType
mov     r8d, 52777445h    ; Tag
mov     r14, r9
lea     edx, [rdi+70h]    ; NumberOfBytes
call    ExAllocatePoolWithTag
mov     rbx, rax
```

Fig. 5.16: Interesting basic block from function *EtwpAddKmRegEntry*, obtained using IDA's disassembler, in charge of performing the allocation of the necessary memory.

As can be inferred from 5.15, this is also an allocation in the heap:

- **PoolType**: 0x200 (**NonPagedPoolNx**).
- **NumberOfBytes**: 0x70. As depicted in the image, *rdi* is first set to 0.
- **Tag**: “0x52777445”. According to the documentation just a four character long is used as a pool tag. Due to being specified in reverse order: 0x45747752 → “EtwR”.

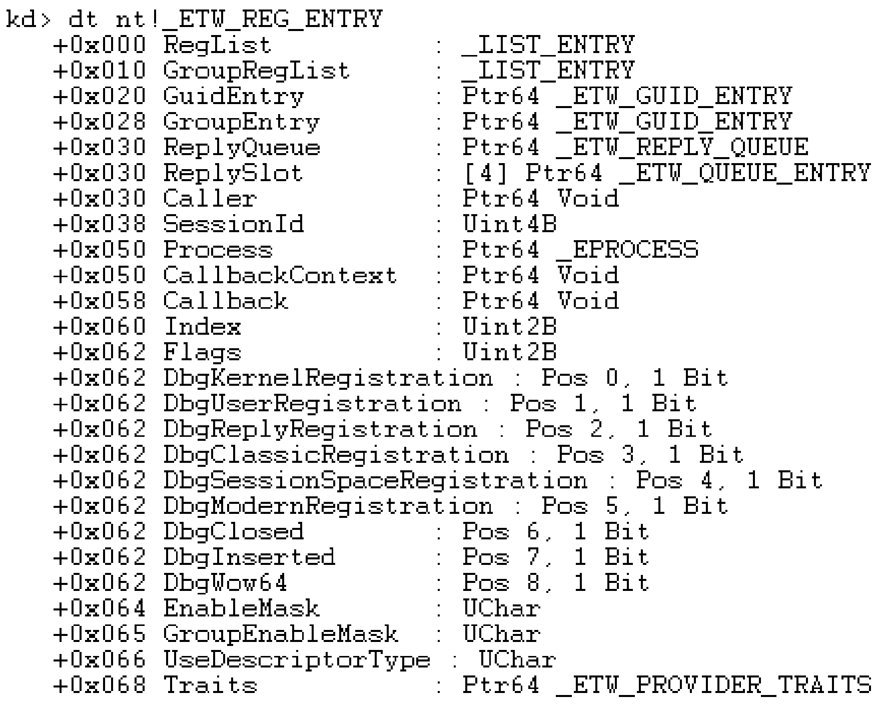
The analysis of the following lines of *EtwpAddKmRegEntry* showed how the aforementioned reserved space (structure) was being filled. While debugging this function the following line excelled from the rest:

```
mov     [rbx+20h], rbp
```

The reason to highlight it was that, at that point, *rbp* held the pointer to the GUID entry. Meaning that this structure, potentially the registration handler structure, has the pointer to the GUID entry at offset 0x20. After finishing filling the rest, the pointer to the structure is returned.

Going back to 5.6, it can be appreciated that the output of *EtwpAddKmRegEntry* is the output of *EtwpRegisterProvider* too. Which confirmed that this was the registration handler structure.

After finishing with this analysis, the documentation of the registration handler structure was found:



```
kd> dt nt!_ETW_REG_ENTRY
   +0x000 RegList          : _LIST_ENTRY
   +0x010 GroupRegList     : _LIST_ENTRY
   +0x020 GuidEntry        : Ptr64 _ETW_GUID_ENTRY
   +0x028 GroupEntry       : Ptr64 _ETW_GUID_ENTRY
   +0x030 ReplyQueue       : Ptr64 _ETW_REPLY_QUEUE
   +0x030 ReplySlot        : [4] Ptr64 _ETW_QUEUE_ENTRY
   +0x030 Caller           : Ptr64 Void
   +0x038 SessionId        : Uint4B
   +0x050 Process          : Ptr64 _EPROCESS
   +0x050 CallbackContext  : Ptr64 Void
   +0x058 Callback         : Ptr64 Void
   +0x060 Index            : Uint2B
   +0x062 Flags            : Uint2B
   +0x062 DbgKernelRegistration : Pos 0, 1 Bit
   +0x062 DbgUserRegistration : Pos 1, 1 Bit
   +0x062 DbgReplyRegistration : Pos 2, 1 Bit
   +0x062 DbgClassicRegistration : Pos 3, 1 Bit
   +0x062 DbgSessionSpaceRegistration : Pos 4, 1 Bit
   +0x062 DbgModernRegistration : Pos 5, 1 Bit
   +0x062 DbgClosed        : Pos 6, 1 Bit
   +0x062 DbgInserted      : Pos 7, 1 Bit
   +0x062 DbgWow64         : Pos 8, 1 Bit
   +0x064 EnableMask       : UChar
   +0x065 GroupEnableMask  : UChar
   +0x066 UseDescriptorType : UChar
   +0x068 Traits           : Ptr64 _ETW_PROVIDER_TRAITS
```

Fig. 5.17: Illustration of structure *_ETW_REG_ENTRY* using WinDBG’s *dt* command.

5.2 Hooking into providers' writes

In section 5.1 an idea on how to answer two important questions was presented: Hook in the exact moment when providers write to DiagTrack's session. However the idea would have been unfeasible without the analysis performed in section 5.1.1.

At this point it was possible to detect if a particular provider was writing by knowing just its GUID. However it was important to understand to which session this provider was writing.

For an initial analysis a combination of automatic and manual analysis was performed. A breakpoint in the function **EtwWrite** was set using the following Windbg(2.2.2) script:

```
bp nt!EtwWrite ".printf \"Handler: %N\\n\",@rcx"
```

This script just printed the address of the provider's registration handle that is performing the write (@*rcx* holds the first parameter of the function according to Windows x64 calling convention). Once the handler's address was obtained it was possible to get the GUID as well. Comparing the GUID with the output of the powershell command **Get-EtwTraceProvider** (without filters) threw an interesting result: Most of the time a provider with E02A841C-75A3-4FA7-AFC8-AE09CF9B7F23 as GUID was the one writing. Unfortunately, this provider was not related to DiagTrack at all.

With the goal of filtering out writes carried out by this provider, the following script was used:

```
bp nt!EtwWrite ".if (@rcx != ffffda839f0c2c50){.printf \"Handler:
    %N\\n\",@rcx;gc;}.else{gc;} "
```

However, this was not enough. Providers which were not related to DiagTrack continued flooding the output of the script. Clearly, this was not a good approach.

With the objective of pursuing interesting results, another ETW-related writing function called **EtwWriteTransfer** was used :

```
bp nt!EtwWriteTransfer ".if (@rcx != ffffda839f0c2c50){.printf \"Handler:
    %N\\n\",@rcx;gc;}.else{gc;}"
```

This time, a new provider attached to DiagTrack with GUID E9EAF418-0C07-464C-AD14-A7F353349A00 appeared. To get some extra information, the script was updated once more to get the Call Stack of the provider's process:

```
bp nt!EtwWriteTransfer ".if (@rcx == FFFFDA83A036F0D0){.printf \"Handler:
    %N\\n\",@rcx;kc;gc;}.else{gc;} "

 Call Site
00 nt!EtwWriteTransfer
01 nt!TlgWrite
02 nt!CmpInitHiveFromFile
03 nt!CmpCmdHiveOpen
04 nt!CmLoadAppKey
05 nt!CmLoadDifferencingKey
06 nt!NtLoadKeyEx
07 nt!KiSystemServiceCopyEnd
08 ntdll!NtLoadKeyEx
```

```
09 0x0
```

So far, the only way to detect when providers related to DiagTrack were writing consisted in two steps:

1. Hook in the *EtwWrite* function call.
2. Check if the provider is attached to DiagTrack based on its GUID and the output of the powershell command shown in Figure 5.1.

However, even if it was a DiagTrack's related provider the one writing, it was not enough to ensure that it is currently writing to DiagTrack's session (providers could be attached to several sessions). Furthermore, *EtwWriteTransfer* showed that *EtwWrite* was not the only function involved in the writing process. In other words, the following two questions were raised:

1. Is **EtwWrite** the only write function used? (clearly no!)
2. How can we be sure that a provider is actually writing to DiagTrack's session?

5.2.1 ETW functions to write

A quick analysis of symbols and cross references of **ntoskrnl.exe** binary showed that there were actually a "group of functions" that could be used to perform writes within the ETW framework:

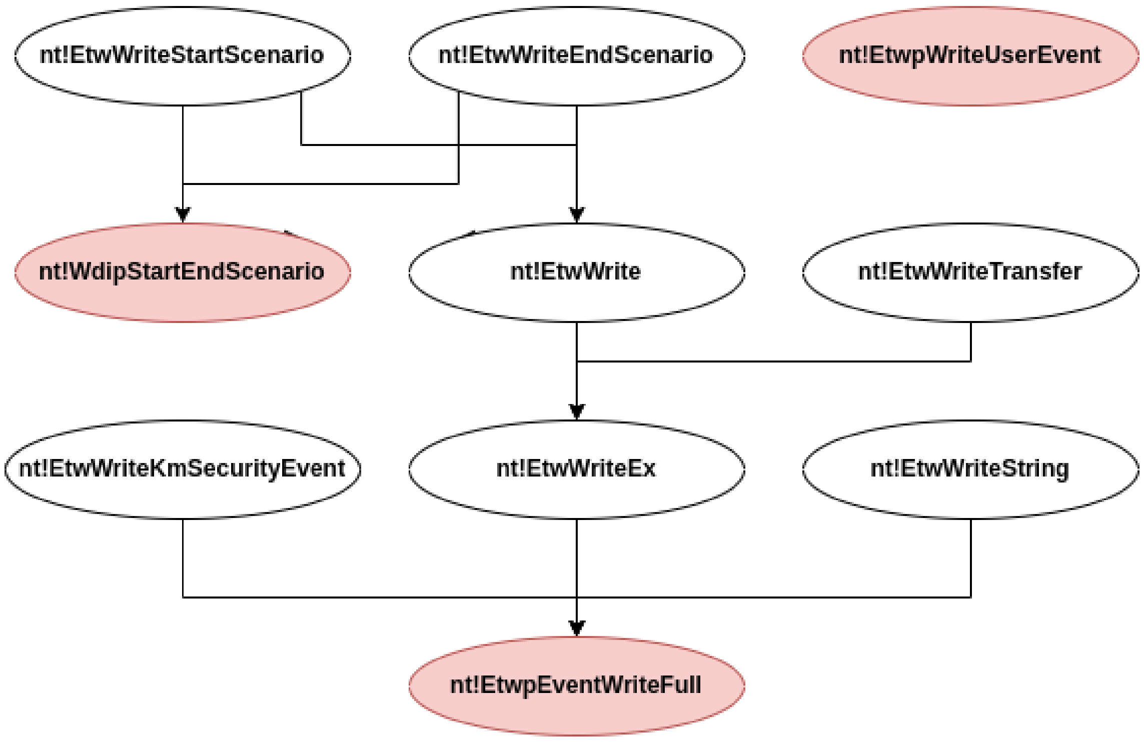


Fig. 5.18: Illustration of all ETW functions related to writing events and their call flow.

As can be appreciated in figure 5.18 all functions end up calling either **nt!EtwpEventWriteFull**, **nt!EtwpWriteUserEvent** or **nt!WdispStartEndScenario**.

It was necessary to understand if all of these functions were called whenever a provider wanted to log an event. Therefore, it was decided to forge a WinDBG script which set breakpoints in the three functions and logged every time one of them was used. This script was left to run debugging a machine while it was being used for normal daily tasks. After some days of debugging, the output was analyzed. It was possible to conclude that the function **nt!WdispStartEndScenario** was never called under the context of interest of this research. Because of this result, it was decided to continue the investigation focusing only on the first two functions: **nt!EtwpEventWriteFull** and **nt!EtwpWriteUserEvent**.

5.2.2 Ensuring providers write to DiagTrack session

The previous idea of only analyzing writes from providers that were returned by the powershell command had, at least, three problems:

1. The output of the powershell command returned providers that were registered in a particular session at a given time t. If a provider registers, writes and unregisters itself from the session in a time frame tf where $t \notin tf$, that write will not be took into consideration.
2. Even if the provider is registered against DiagTrack's session, this does not ensure the provider is currently writing to it as it could happen that, for some reason, it was deactivated.
3. It relied too much on manual analysis.

After several brainstorming sessions, an idea that made the difference came up: What if it is possible to relate the handler that it is used at the moment of writing, with the session where the provider is going to write. If that is possible, the first two problems would be solved. The third one could be solved as only one breakpoint at the writing function will be enough to make the full analysis.

Following sections will detail the process to find an answer to these problems.

5.2.2.1 Inspecting ETW structures

So far it happened that several ETW structures were the key to overcome different obstacles. We wanted to know if this was again the case.

The first structure analyzed was *_ETW_REG_ENTRY* (figure 5.17) as it would have been the most direct and easier way to relate session and provider. Unfortunately, none of its components were helpful.

The next structure analyzed was *_WMI_LOGGER_CONTEXT*. This structure seemed to be the actual representation of an ETW session. Due to its size, only necessary and representative offsets are depicted:

```
+0x000 LoggerId       : Uint4B
+0x004 BufferSize     : Uint4B
+0x008 MaximumEventSize : Uint4B
+0x00c LoggerMode     : Uint4B
+0x010 AcceptNewEvents : Int4B
[...]
+0x070 ProviderBinaryList : _LIST_ENTRY
+0x080 BatchedBufferList : Ptr64 _WMI_BUFFER_HEADER
+0x080 CurrentBuffer  : _EX_FAST_REF
+0x088 LoggerName     : _UNICODE_STRING
+0x098 LogFileName    : _UNICODE_STRING
+0x0a8 LogFilePattern : _UNICODE_STRING
+0x0b8 NewLogFileName : _UNICODE_STRING
[...]
+0x428 LastBufferSwitchTime : _LARGE_INTEGER
+0x430 BufferWriteDuration : _LARGE_INTEGER
+0x438 BufferCompressDuration : _LARGE_INTEGER
```

Fig. 5.19: Illustration of the *_WMI_LOGGER_CONTEXT* structure through using WinDBG's *dt* command.

Each ETW session will have an instance of this structure. At offset *0x70* there is an attribute called **ProviderBinaryList**. After a quick analysis this attribute seemed to hold all providers registered against the session in the format of a double linked list. In order to confirm that theory, the process of attaching new providers to an existing session was analyzed.

The function *nt!EtwpAddProviderToSession* seemed to be the one creating these links. During the analysis several problems were faced: a lot of unknown new functions, hard to reverse engineer them, lack of documentation, among others. Besides all mentioned issues, the key reason why this avenue of analysis was stopped was: The finding of the structure *_TRACE_ENABLE_INFO* (more details in the next chapter).

5.2.2.2 Provider GUID and Provider Group GUID

Usually a structure can have the answer, but if you cannot understand how that structure is being used you will not be able to find that. With that in mind, a further analysis over the ETW writes functions was carried out.

EtwWriteEx was one of the most commonly used functions by providers to write inside events inside sessions. That was the reason why the analysis was focused on it.

The provider and the actual event that wanted to be logged were some of the parameters that *EtwWriteEx* received. An interesting piece of its source code can be depicted easily as pseudocode, with the following listing:

```
1      // First part
2      v14 = *(_BYTE *)(ptr_handler + 0x64);
3      if(v14){
4        v15 = *(_QWORD **)(ptr_handler + 0x20);
5        if ( EtwpLevelKeywordEnabled((__int64)(v15 + 0x50), a2, a3) ){
6          EtwpEventWriteFull(...)
7        }
8      }
9      // Second part
10     v14 = *(_BYTE *)(ptr_handler + 0x65);
11     if (v14)}
12       v15 = *(_QWORD **)(ptr_handler + 0x28);
13       if ( EtwpLevelKeywordEnabled((__int64)(v15 + 0x50), a2, a3) ){
14         EtwpEventWriteFull(...)
15       }
16     }
```

Fig. 5.20: Pseudocode snippet of an interesting part of function *EtwWriteEx*, obtained analyzing IDA's disassembler and decompiler outputs.

At first, both parts (line 1-7 and 8-14) may seem equal, but they have two key differences:

1. The offset used for **v14**: The first part uses *0x64*, while the second one uses *0x65*.
2. The offset used for **v15**: The first part uses *0x20*, while the second one uses *0x28*

Going back to figure 5.17, it is possible to understand that the first part is using the attributes *GuidEntry* and *EnableMask* while the second one is using *GroupGuidEntry* and *GroupEnableMask*. This was the key to understand that providers do not always use the same GUID entry, but they can also use a "Group GUID".

Furthermore, it was found that the function *EtwpLevelKeywordEnabled* received three parameters:

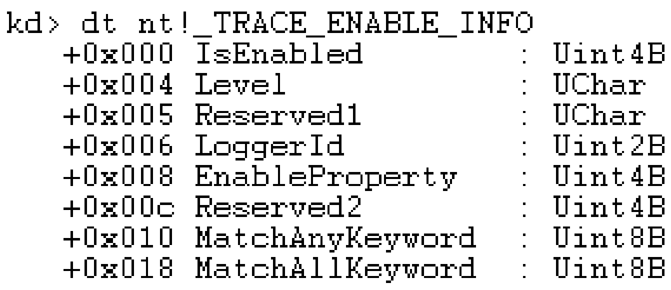
- *ProviderEnableInfo* of type *_TRACE_ENABLE_INFO*
- Event Level
- Event keyword

The main objective of *EtwpLevelKeywordEnabled* was to check if the event should be logged by the provider into the session according to the filtering rules[8] defined when the provider was first registered against it.

In other words, listing 5.20 could be translated to "If *EnableMask* is different from 0x0, check if the event should be logged using the *GuidEntry*. If it should, call *EtwpEventWriteFull*. If *GroupEnableMask* is different from 0x0, check if the event should be logged using the *GroupEntry*. If it should, call *EtwpEventWriteFull*."

The first parameter received in *EtwpLevelKeywordEnabled* was of type *_TRACE_ENABLE_INFO*. Analyzing its structure:

[8] https://docs.microsoft.com/en-us/message-analyzer/system-etw-provider-event-keyword-level-settings



```
kd> dt nt!_TRACE_ENABLE_INFO
   +0x000 IsEnabled        : Uint4B
   +0x004 Level            : UChar
   +0x005 Reserved1        : UChar
   +0x006 LoggerId         : Uint2B
   +0x008 EnableProperty   : Uint4B
   +0x00c Reserved2        : Uint4B
   +0x010 MatchAnyKeyword  : Uint8B
   +0x018 MatchAllKeyword  : Uint8B
```

Fig. 5.21: Illustration of *_TRACE_ENABLE_INFO* structure obtained using WinDBG's *dt* command.

This structure seemed to be promising as it had information that could relate to a session. Furthermore, this structure can be found inside *_ETW_GUID_ENTRY* at offset *0x50* and *0x70* (while the former refers to the provider GUID, the latter refers to the provider group GUID). However, the logic which filled or used this structure is not present inside *EtwWriteEx*. As shown in 5.20 after some checks, the function *EtwpEventWriteFull* was being called. In order to continue with this analysis, it was necessary to analyze this last function.

5.2.2.3 Identifying the destination session

The analysis of *EtwpEventWriteFull* was the hardest yet the most interesting one. Although it received 17 parameters, had more that 1000 lines of code and helped in understanding multiple pieces of the puzzle, only few lines are going to be analyzed and illustrated here.

It turned out, as it was believed, that the logic to understand which session the execution was going to write was inside this function. The goal of this section is to explain in depth how this process of choosing the session is carried out.

To begin with the analysis, lets start analyzing this simplified pseudocode which represents the key part of the process:

```
  [...]
  while ( 1 ){
    bsf_found = !_BitScanForward((unsigned int *)&enable_info_bucket_index,
        enable_mask);
    guid_ptr_shifted_by_enable_info_bucket_index = (__int64)&ptr_guid_entry[4
        * (unsigned int)enable_info_bucket_index]

    if ( _bittest64(&ptr_local_addr, *(unsigned __int8
        *)(guid_ptr_shifted_by_enable_info_bucket_index + 0x76)) )
      continue;

    [...]
```

Fig. 5.22: Pseudocode snippet of an interesting part of function *EtwpEventWriteFull*, obtained analyzing IDA's disassembler and decompiler outputs. This part illustrates how the function selects which ETW session to write.

The **1st** line represents previous definitions and statements that are not important for the current analysis.
The **2nd** line represents the cycle that will be executed. In each iteration, it will pick one of the sessions that the provider is registered against and check if it should write this event to that session. To perform such a thing, it was found that the attribute *enableMask* it is an 8-bit mask which represents the use of each of the 8 buckets from the *enableInfo* attribute (*GuidEntry* structure). If the first bit (position 0) of *enableMask* was set to 1, it meant that the information inside *enableInfo*[0] should be considered. If its value was 0, the information of *enableInfo*[0] should be dismissed. This was exactly what was happening in the **3rd** line. It performed a search[9] over the **enable_mask** variable which held the actual content of the *enableMask* attribute from the *GuidRegEntry*, and wrote inside *enable_info_bucket_index* the index of the first bit set to 1.
The **4th** line shifted a temporary pointer to the *GuidEntry* by *0x20* * *enable_info_bucket_index*. In this case, *0x20* was the size of the structure *_TRACE_ENABLE_INFO* (figure 5.21), which was the type of each *enableInfo* bucket. In other words, it was shifting the pointer depending on which bucket of *enableInfo* should be considered in this iteration of the cycle.
The **6th** line was the one that gave sense to all the aforementioned steps. It added *0x76* to the shifted pointer and checked[10] if the bit at the specified position was 1 or 0. This was basically checking if the *loggerId* of the corresponding *enableInfo* had some value. Let's do an example to better understand this:
If the result of the *enable_info_bucket_index* was equal to 0, it meant that the least significant bit of *enableMask* had a *1*. As explained before, this meant that the *enableInfo*[0] should be considered. Therefore, the following step would be to shift the pointer to *GuidEntry* by $0x20$ * 0, in other words, do not shift it. Finally, it checked if the position $0x76$ had something different from 0. $0x76 = 0x70 + 0x6$, $0x70$ was the offset of the first bucket of *enableInfo* attribute from the *GuidEntry*, while $0x6$ was the offset of *loggerId* inside the *_TRACE_ENABLE_INFO* structure. Now, if the *enable_info_bucket_index* was 1 instead of 0, the only difference from the previous example would have been that the pointer to the *guidEntry* would be shifted $0x20 * 1 = 0x20$. This meant that the position to check with *_bittest64* would have been $0x76 + 0x20$. Due to $0x20$ was the size of *_TRACE_ENABLE_INFO* structure it would have accessed the next bucket's *loggerId* of *enableInfo* (*enableInfo*[1])
If the **6th** line condition was accomplished, it meant that *loggerId* was 0 and therefore not interesting. Hence, it would have jumped directly to the next iteration of the cycle. If the condition was not met, it would have continued with the rest of the statements.
The **9th** line represents all operations that were executed once a valid session of the provider was found (i.e: checking if this event should be written to that session, the actual write to the session, etc).

The *loggerId* finally became the way to relate a write execution with the destination session. Recalling the structure which represented an ETW session (*_WMI_LOGGER_CONTEXT*, figure 5.19) the first of its attributes was the *loggerId*, an identification for the session. Although, until this point it was unknown the exact moment when the event was written, this logic was enough to understand how the session was chosen.

[9] https://docs.microsoft.com/en-us/cpp/intrinsics/bitscanforward-bitscanforward64?view=vs-2019
[10] https://docs.microsoft.com/en-us/cpp/intrinsics/bittest-bittest64?view=vs-2019

5.2.3 Detecting what is going to be written

Although the last analysis was key to understanding how the destination session was chosen, it was necessary to find the exact moment when *EtwpAllocGuidEntry* was actually writing the event to that session. After all, that was the main goal.

As shown in listing 5.22, **9th** line involved a lot of further processing once a candidate session was found. Among all that processing, it was possible to find the following code related to the way events were being written:

```
1   [...]
2   logger_id = *(unsigned __int16
        *)(guid_ptr_shifted_by_enable_info_bucket_index+0x76);
3   [...]
4   if ( (unsigned int)logger_id >= 0x40 ){
5       ptr_wmi_trace_of_session_cpy = 1i64;
6       ptr_wmi_trace_of_session = 1i64;
7   }
8   else {
9       ptr_wmi_trace_of_session = *(_QWORD *)(ptr_silo_state + 8 * logger_id +
            0x390);
10      ptr_wmi_trace_of_session_cpy = ptr_wmi_trace_of_session;
11  }
12  [...]
13  ptr_to_buffer = EtwpReserveTraceBuffer(
14                      (unsigned int *)ptr_wmi_trace_of_session_cpy,
15                      event_size,
16                      (__int64)&ptr_to_buffer_offset,
17                      &return_value_of_function_to_get_cpu_clock,
18                      0
19                  );
20  if ( ptr_to_buffer ){
21    *(_OWORD *)(ptr_to_buffer + 0x18) = *(_OWORD *)(ptr_guid_entry_cpy + 3);
22    *(_OWORD *)(ptr_to_buffer + 0x28) = *(_OWORD *)ptr_event_descriptor_cpy;
23    [...]
24  }
25  [...]
```

Fig. 5.23: Pseudocode snippet of an interesting part of function *EtwpEventWriteFull*, obtained analyzing IDA's disassembler and decompiler outputs. This part illustrates how the function actually writes event to the selected ETW session.

The **1st** line represented previous definitions and statements (like the whole listing 5.22) that were not important for the current analysis.
The **2nd** line assigned the *loggerId* value of the selected session to a variable. For the sake of clarity, this variable was called *logger_id*.
The **3rd** line represented definitions and statements that were not important for this analysis.
The **4th** line asked if *logger_id ¿= 0x40*. In case it did, it assigned weird values to a supposed pointer. The reason to do it was because the maximum number of ETW sessions that could exist was 64 ($0x40$). In other words, it was ensuring that *logger_id* had a valid value.

In case *logger_id* was valid, lines **9** and **10** were executed. The former one, assigned the value of *ptr_silo_state* (variable defined before but whose content could be inferred from its name) *+ 8 * logger_id + 0x390*. What did all these values mean? At offset $0x390$ of the structure *_ETW_SILODRIVERSTATE* (figure 5.8) there was an attribute called *WmipLoggerContext*. This attribute was an array of pointers (*Ptr64*) to structures of type *_WMI_LOGGER_CONTEXT* (figure 5.19). This array had 64 buckets (again, because of the maximum number of sessions possible) and its index in this array matched with their *loggerId* value. Definitely, this was not coincidence. When adding a new ETW session, the free bucket of this *WmipLoggerContext* is used and therefore its index within this array is used to fulfill the value of *loggerId*. Finally, 8 was the length of a *Ptr64* structure.
As a conclusion, the value of *ptr_wmi_trace_of_session* as the name suggests, will be the pointer to the corresponding *_WMI_LOGGER_CONTEXT* of the session associated with the *logger_id* found.
The **12th** line represented definitions and statements that were not important for this analysis.
The **13th** line represented a call to the most interesting function of this analysis:
EtwpReserveTraceBuffer. This function received parameters such as the pointer to the associated session's *_WMI_LOGGER_CONTEXT* and the size of the event to write. This function returns a pointer to the place (within the buffers of the ETW session) where the event data (and metadata) should be written afterwards (full analysis on the following section).
The **20th** line only checked if the pointer was different from null (in case the OS ran out of memory).
The **21th**, **22th** and **23th** lines represent the exact moment when all the data is being written inside the memory block returned by *EtwpReserveTraceBuffer*.
The **25th** represented statements that would have been executed in case the condition line **20** was not met.

As a summary, it is possible to deduce that line **21** represented the first moment where the event data (and metadata) was being written to the session's buffers.

5.2.4 Ensuring correctness of events logged

Despite the fact the previous analysis seemed to be correct, it was necessary to confirm that all data being logged to the ETW buffers was written as expected. It was possible to force the system to log an event to the DiagTrack session by crafting specific scripts (more on this in section 5.5.1). Therefore, the following strategy was followed in order to confirm that all the findings were correct:

1. Get the logger id of the DiagTrack session.
2. Set a breakpoint few instructions before the actual writing, taking into account the logger id.
3. Once hit, print the event descriptor and the amount of events in the DiagTrack's buffer.
4. Set a new breakpoint few instructions after the writing.
5. Print the event descriptor again and compare it against the previous value.

6. Print the event content to ensure everything is working as expected

A breakpoint was set a few instructions before calling the *EtwpReserveTraceBuffer* function, which meant that the event would not be written yet. Once the debugger got that point, the event descriptor was printed in order to have something to compare with afterwards:

```
!wmitrace.strdump
bp nt!EtwpEventWriteFull+0x286 ".if(r12d == 0x22){.echo 'YEP! it
    entered';.ech ''}.else{gc};gc;"
g
dt nt!_EVENT_DESCRIPTOR @r13
```

Fig. 5.24: WinDBG's combination of commands used to perform steps 1, 2 and 3.

At that point, $0x22$ was the logger id of the DiagTrack session. After printing the event descriptor, the second part was executed:

```
bp nt!EtwpEventWriteFull+0x3c1
g
dt nt!_EVENT_DESCRIPTOR @r13
!wmitrace.eventlogdump 0x22
```

Fig. 5.25: WinDBG's combination of commands used to perform steps 4, 5 and 6.

The new breakpoint (few instructions after writing) was hit. The event descriptor was exactly as before and the log content had the expected information. After these tests, it was possible to conclude that the analysis performed earlier was correct.

Although we believed that at this point we have ensured that events were logged as expected, we wanted to go one step deeper. The idea was to compare the events extracted with our strategy, to the ones that Telemetry actually sends to Microsoft backend servers. In order to do such test, two different tools were used:

- xperf[11]
- Message Analyzer[12]

With **xperf** it is possible to create an ETW session. Therefore, it is possible to create a new session with exactly the same providers as for Telemetry one. The following command was used to accomplish such requirement:

[11] https://docs.microsoft.com/en-us/windows-hardware/test/wpt/xperf-command-line-reference
[12] https://docs.microsoft.com/en-us/message-analyzer/microsoft-message-analyzer-operating-guide

```
xperf -start UserTrace -on
    Microsoft-Windows-Diagtrack+43ac453b-97cd-4b51-4376-db7c9bb963ac+da995380
-18dc-5612-a0db-161c5db2c2c1+6489b27f-7c43-5886-1d00-0a61bb2a375b -f
    C:\tmp\diaglog%d.etl -FileMode Newfile -MaxFile 50
```

Fig. 5.26: Shell command issued to execute the program *xperf* in order to log all events from DiagTrack to a file. The goal of this command, is to later be able to compare the content of this file against the information extracted throughout our WinDBG scripts.

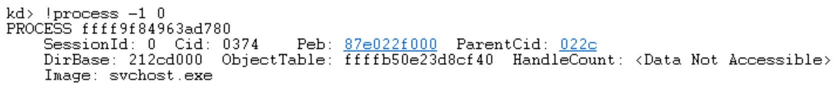
Additionally, this command will create a **.etl** file containing all the logs stored by the ETW providers attached to that session. In order to be able to read those logs, **Message Analyzer** was used. Just a simple comparison between the logs inside the **.etl** file and the logs captured with our strategy, although having some minor differences, concluded that we were doing things right. The following figure shows an example of how the same event would be presented in our strategy vs how Message Analyzer did it:

```
Our Strategy
============
{"UTCReplace_AppId": "{00000d8c-0001-0100-66e6-ff87e6fbd301}",
    "UTCReplace_AppVersion": 1, "UTCReplace_CommandLineHash": 1,
    "AppSessionId": "{00000d8c-0001-0100-66e6-ff87e6fbd301}",
    "AggregationStartTime": "06/04/2018 09:29:03.935",
    "AggregationDurationMS": 19765, "InFocusDurationMS": 313,
    "FocusLostCount": 1, "NewProcessCount": 1, "UserActiveDurationMS": 313,
    "UserOrDisplayActiveDurationMS": 313, "UserActiveTransitionCount": 0,
    "InFocusBitmap.Length": 8, "InFocusBitmap": "00000008 00 02 00 00 00 00
    00 00", "InputSec": 0, "KeyboardInputSec": 0, "MouseInputSec": 0,
    "TouchInputSec": 0, "PenInputSec": 0, "HidInputSec": 0, "WindowWidth":
    993, "WindowHeight": 519, "MonitorWidth": 1920, "MonitorHeight": 1080,
    "MonitorFlags": "0x00", "WindowFlags": "0x10",
    "InteractiveTimeoutPeriodMS": 60000, "AggregationPeriodMS": 120000,
    "BitPeriodMS": 2000, "AggregationFlags": "0x00000031",
    "TotalUserOrDisplayActiveDurationMS": 313, "SummaryRound": 25,
    "SpeechRecognitionSec": 0, "GameInputSec": 0, "EventSequence": 282}


Message Analyzer
================
{"AppId":"W:0000f519feec486de87ed73cb92d3cac802400000000!000000667a0f0c0d5e9da69
7e9ff54ecddd449259354!conhost.exe",
    "AppVersion":"2016/07/16:02:28:13!1375C!conhost.exe",
    "CommandLineHash":2216829733,
    "AppSessionId":"00000D8C-0001-0100-66E6-FF87E6FBD301",
    "AggregationStartTime":"2018-06-04T09:29:03.9355680Z",
    "AggregationDurationMS":19765, "InFocusDurationMS":313,
    "FocusLostCount":1, "NewProcessCount":1, "UserActiveDurationMS":313,
    "UserOrDisplayActiveDurationMS":313, "UserActiveTransitionCount":0,
    "InFocusBitmap":"0x0002000000000000", "InputSec":0,
    "KeyboardInputSec":0, "MouseInputSec":0, "TouchInputSec":0,
    "PenInputSec":0, "HidInputSec":0, "WindowWidth":993, "WindowHeight":519,
    "MonitorWidth":1920, "MonitorHeight":1080, "MonitorFlags":0,
    "WindowFlags":16, "InteractiveTimeoutPeriodMS":60000,
    "AggregationPeriodMS":120000, "BitPeriodMS":2000, "AggregationFlags":49,
    "TotalUserOrDisplayActiveDurationMS":313, "SummaryRound":25,
    "SpeechRecognitionSec":0, "GameInputSec":0, "EventSequence":282}
```

Fig. 5.27: Comparison between the event content extracted through our strategy of setting breakpoints in particular places inside *EtwpReserveTraceBuffer* against the content logged by Message Analyzer

5.3 Service isolation

At this point of the research we already understood how to know what information was going to be written and how. However, we were missing a very important part: the program that was actually writing the log. At first glance, this may seem a super trivial thing to extract. The **process** structure holds the information about the execution context and therefore knows the name of the process actually running. This could be easily gathered by making use of the WinDBG **process** extension:



```
kd> !process -1 0
PROCESS ffff9f84963ad780
    SessionId: 0  Cid: 0374    Peb: 87e022f000  ParentCid: 022c
    DirBase: 212cd000  ObjectTable: ffffb50e23d8cf40  HandleCount: <Data Not Accessible>
    Image: svchost.exe
```

Fig. 5.28: WinDBG's command showing an illustration of the information outputted using the process extension. In this particular case, it is showing information related to the process that it is currently writing.

Services inside Windows are usually implemented inside Dynamic Linked Libraries files. However, DLL's cannot run inside a process as executable binaries (.exe). As part of Windows 10 features, the **svchosts.exe** was introduced. In a few words, **svchosts.exe** allows several Windows services to run in a nested way, meaning that they will all run within the same process. From our strategy perspective, this represented a concern because if we wanted to have an accurate way to determine which process/service was the one actually logging an event, we had to find a way to deal with nested services.

Our first idea was to force all OS services to run in an isolated way (not nested), by issuing the following command:

```
$> sc config <service_name> type=own
```

Fig. 5.29: Shell command issued to force a service to run in an isolated way.

However, this options did not work as it only applied to currently running services.

The second idea was to proper configure the **HKLM:/SYSTEM/CurrentControlSet/Services** registry key which defines all service configurations and capabilities. By modifying the **Type** key you would be able to specify the type of the service:

- 0x10: For isolated service
- 0x20: For nested service

Unluckily, this option did not work as forcing every service to run in an isolated way requires a lot of RAM, making this option unfeasible.

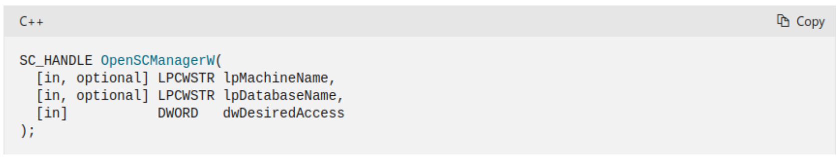
The following (and definitive) way of identifying services came up after weeks of research:

Alex Ionescu wrote a blog post[13] describing a tool developed by himself named **ScTagQuery** that was able to return the service related to a TID (thread id). In other words, this tool has the logic to map a thread to service. According to his blog, he carries out this action by using a value inside the TEB (Thread Environment Block) called **SubProcessTag**. It seemed that this value was an index inside a database of services. However, he never explains how this index actually works. Luckily he mentions that all this functionality is implemented in the SCM (Service Control Manager).

```
kd> dt nt!_TEB
    +0x000 NtTib                    : _NT_TIB
    +0x038 EnvironmentPointer       : Ptr64 Void
    +0x040 ClientId                 : _CLIENT_ID
    +0x050 ActiveRpcHandle          : Ptr64 Void
    ....
    +0x1710 ActivityId              : _GUID
    +0x1720 SubProcessTag           : Ptr64 Void
    +0x1728 PerflibData             : Ptr64 Void
    ...
```

Fig. 5.30: Illustration of the structure "_TEB" (Thread Environment Block) highlighting the *SubProcessTag* at offset 0x1720.

Reversing Alex's tool, an interesting function called **I_ScQueryServiceTagInfo** was found. This function was the one actually getting the service name by providing a service tag. In order to do so, it internally calls another function named **OpenSCManagerW** which is documented by Microsoft[14]:



```
SC_HANDLE OpenSCManagerW(
  [in, optional] LPCWSTR lpMachineName,
  [in, optional] LPCWSTR lpDatabaseName,
  [in]           DWORD   dwDesiredAccess
);
```

Fig. 5.31: OpenSCManagerW's documentation gathered from Microsoft Official Documentation.

The parameters provided to this function were fixed (0, 0 and 4 respectively):

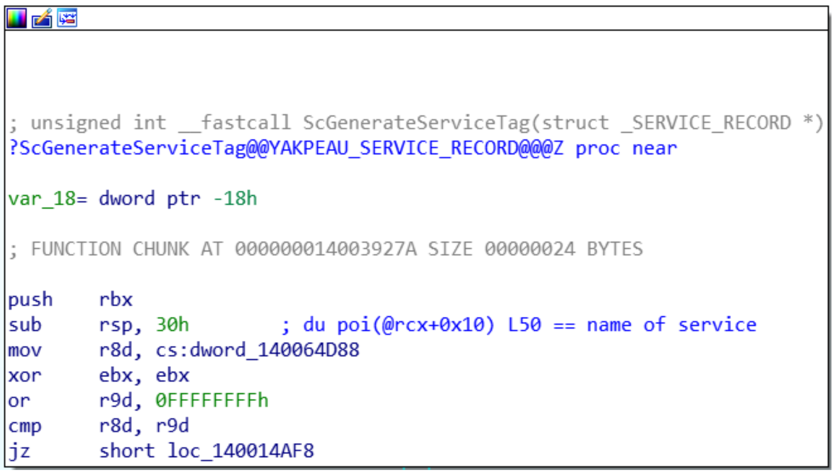
- **lpMachineName:** If the pointer is NULL or points to an empty string, the function connects to the service control manager on the local computer.
- **lpDatabaseName:** If it is NULL, the SERVICES_ACTIVE_DATABASE database is opened by default.
- **dwDesiredAccess:** Specifies the access right to the service control manager. In our case was 0x4, meaning it was using the SC_MANAGER_ENUMERATE_SERVICE which is required to call the EnumServicesStatus or EnumServicesStatusEx function to list the services that are in the database

[13] http://www.alex-ionescu.com/?p=52
[14] https://docs.microsoft.com/en-us/windows/win32/api/winsvc/nf-winsvc-openscmanagerw

Once the handler to the SCM database was returned, another function **I_ScQueryServiceTagInfo** was called. However, this last function was initializing an RPC communication and therefore made the analysis even harder. Instead of continuing down this road, we decided to take a look at the binary that was implementing the SCM: **services.exe**.

Within **services.exe** several promising functions were found such as **ScGenerate-ServiceDB** or **ScGetServiceNameTagMapping**. Although the latter seemed to be the answer, after some testing we realized that this function was being called only for certain particular services. By going deeper in the analysis, a third function was found: **ScGenerateServiceTag**.

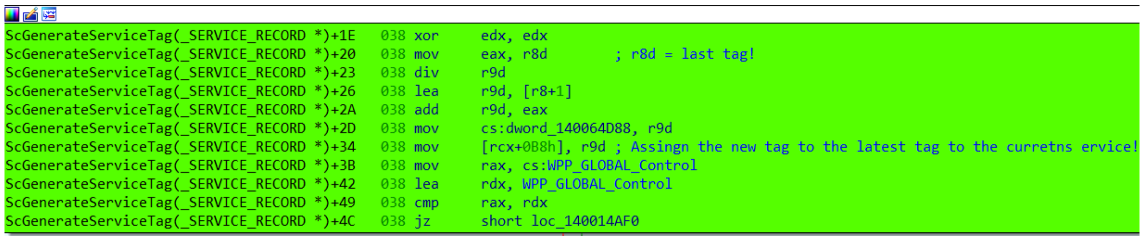
This function is executed each time a new service is registered to the OS. In simple words, **ScGenerateServiceTag** receives the *_SERVICE_RECORD* as the only parameter and looks for the latest value of tags assigned. The service's name comes at offset *0x10* within the *_SERVICE_RECORD* structure, and due to the first parameter comes in *rcx*, then *@rcx+10* will have the actual name of the current service.



```
; unsigned int __fastcall ScGenerateServiceTag(struct _SERVICE_RECORD *)
?ScGenerateServiceTag@@YAKPEAU_SERVICE_RECORD@@@Z proc near

var_18= dword ptr -18h

; FUNCTION CHUNK AT 000000014003927A SIZE 00000024 BYTES

push    rbx
sub     rsp, 30h        ; du poi(@rcx+0x10) L50 == name of service
mov     r8d, cs:dword_140064D88
xor     ebx, ebx
or      r9d, 0FFFFFFFFh
cmp     r8d, r9d
jz      short loc_140014AF8
```

Fig. 5.32: First basic block of the function *ScGenerateServiceTag*, obtained using IDA's disassembler, shown in order to illustrate the mentioned offsets.

The latest used tag is stored at *cs:dword_140064D88*a matter of incrementing that value. That is why the value of the last tag is stored inside *r8d* and later incremented. Finally, this new value is saved inside the provided service record at offset *0x0b8* (service tag's offset).



```
ScGenerateServiceTag(_SERVICE_RECORD *)+1E   038 xor     edx, edx
ScGenerateServiceTag(_SERVICE_RECORD *)+20   038 mov     eax, r8d        ; r8d = last tag!
ScGenerateServiceTag(_SERVICE_RECORD *)+23   038 div     r9d
ScGenerateServiceTag(_SERVICE_RECORD *)+26   038 lea     r9d, [r8+1]
ScGenerateServiceTag(_SERVICE_RECORD *)+2A   038 add     r9d, eax
ScGenerateServiceTag(_SERVICE_RECORD *)+2D   038 mov     cs:dword_140064D88, r9d
ScGenerateServiceTag(_SERVICE_RECORD *)+34   038 mov     [rcx+0B8h], r9d ; Assingn the new tag to the latest tag to the curretns ervice!
ScGenerateServiceTag(_SERVICE_RECORD *)+3B   038 mov     rax, cs:WPP_GLOBAL_Control
ScGenerateServiceTag(_SERVICE_RECORD *)+42   038 lea     rdx, WPP_GLOBAL_Control
ScGenerateServiceTag(_SERVICE_RECORD *)+49   038 cmp     rax, rdx
ScGenerateServiceTag(_SERVICE_RECORD *)+4C   038 jz      short loc_140014AF0
```

Fig. 5.33: A basic block of the function *ScGenerateServiceTag*, obtained using IDA's disassembler, shown in order to highlight the process of reading the last service tag and, assigning and storing the new one.

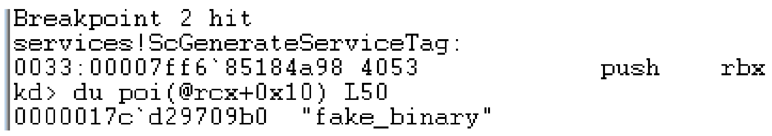
Although it seemed promising, this idea came up from reversing the binary. In order to confirm it, we did the following test:

1. Developed a simple .exe which created a new file (source code in Appedix)
2. Set a breakpoint in **ScGenerateServiceTag**.
3. Ran the following command:

```
    sc create fake_service binPath= <path_to_binary>
```

Fig. 5.34: Shell command issued in order to create an arbitrary service out of a binary.

After executing the last step, the breakpoint was reached successfully!



```
Breakpoint 2 hit
services!ScGenerateServiceTag:
0033:00007ff6`85184a98 4053            push    rbx
kd> du poi(@rcx+0x10) L50
0000017c`d29709b0  "fake_binary"
```

Fig. 5.35: WinDBG's illustration of the moment the previously discussed breakpoint in function *ScGenerateServiceTag* was hit just after executing the fake service.

As a summary, every time a new service is added, it will trigger the execution of **ScGenerateServiceTag** generating a tag for that service. Although a way to dump the entire database was not found, by setting a breakpoint in this function we could maintain a "copy" of the database. Having this database means that every time a **svchost.exe** process writes, we could ask for its service tag by inspecting the TEB (Thread Environment Block) and match it against our DB to know which is the real service.

5.4 Automation

Although a big part of all the previous analysis was carried out using static analysis, dynamic analysis also played a central role. Furthermore, one of the goals of the research was to have an automatic framework capable of monitoring every log of the DiagTrack session in real time.

The following sections will show every WinDBG script developed in order to overcome particular situations. There were several intermediate versions of each script, but for the sake of simplicity only the final version will be shown.

It is important to highlight that the output of these scripts work as a source of information for later processing. Most of the information printed by the WinDBG scripts cannot be immediately leveraged, but requires a full processing once the execution is finished.

5.4.1 Automating DiagTrack logger id search

One of the very first things that needed to be automated was the search for the logger id of the DiagTrack session. This value was key for the whole process due to being the one used to ensure that the write was related to DiagTrack's session.

The first step was to try to find at which moment this value was defined for each ETW session. By inspecting the symbols of ntoskrnl.exe the function *EtwpLookupLoggerIdByName* came up and, because of its name, it was the first candidate. In order to confirm that this was a good candidate, we started with the tasks of reverse engineering and debugging it.

```c

__int64 __fastcall EtwpLookupLoggerIdByName(__int64 ptr_silo_globals, const
    UNICODE_STRING *logger_name, unsigned int *addr_of_logger_id)
{
  unsigned int *logger_id_to_ret; // r14
  const UNICODE_STRING *logger_name_cpy; // r15
  __int64 ptr_silo_globals_cpy; // rbp
  unsigned int v6; // esi
  unsigned int logger_id_iterator; // ebx
  unsigned int *logger_context; // rax
  unsigned int *logger_context_cpy; // rdi

  logger_id_to_ret = addr_of_logger_id;
  logger_name_cpy = logger_name;
  v6 = 0xC0000296
  ptr_silo_globals_cpy = ptr_silo_globals;
  logger_id_iterator = 0;
  while ( 1 )                               // Iterating all loggers
  {
    logger_context = EtwpAcquireLoggerContextByLoggerId(ptr_silo_globals_cpy,
        logger_id_iterator, 0);
    if ( logger_context )
      break;
LABEL_3:
    if ( ++logger_id_iterator >= 0x40 )     // break conditions (there can not be
        more that 64 sessions)
      return v6;
  }
  [...]

  EtwpReleaseLoggerContext(logger_context_cpy, 0);
  v6 = 0;
  *logger_id_to_ret = logger_id_iterator;
  return v6;
```

Fig. 5.36: Pseudocode snippet of *EtwpLookupLoggerIdByName* obtained using a combination of IDA's disassembler and decompiler.

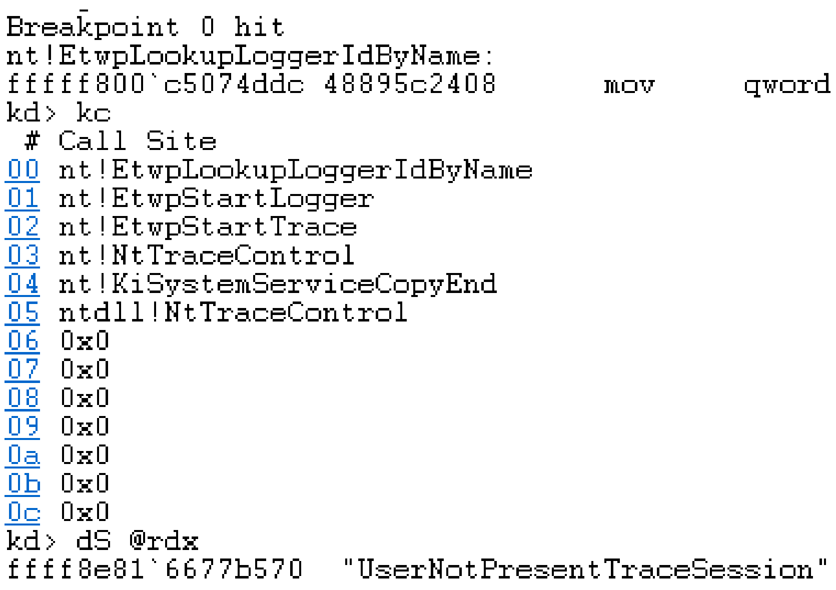
From it is code it was possible to understand that:

1. The first parameter (*rcx*) is the pointer to the **_ETW_SILODRIVERSTATE** structure (figure 5.8), the second one (*rdx*) a pointer to a unicode string (probably the logger name) and the third one seems to be just a buffer where the logger id for this new will be placed.
2. There is an iterator which represents the logger id number, that will go from 0 up to 0x40 (64). This iterator will not be able to go further than 0x40 because it is the maximum number of sessions that can be held inside the ETW framework at the same time.
3. As a summary, this function seems to be the one that actually maps a logger name against a logger id and registers this relation inside the **_ETW_SILODRIVERSTATE** structure.

To help validate this information, we proceed to perform a dynamic analysis. Most probably, the call to this function was going to happen during (or right after) the operating system was booting. In order to confirm this idea, a breakpoint at function *EtwpLookupLoggerIdByName* was set and the debugee system was turned on:

```
bp nt!EtwpLookupLoggerIdByName
```

After a couple of seconds the breakpoint was hit. To confirm the aforementioned statements, two commands were issued:



```
Breakpoint 0 hit
nt!EtwpLookupLoggerIdByName:
fffff800`c5074ddc 48895c2408      mov     qword
kd> kc
 # Call Site
00 nt!EtwpLookupLoggerIdByName
01 nt!EtwpStartLogger
02 nt!EtwpStartTrace
03 nt!NtTraceControl
04 nt!KiSystemServiceCopyEnd
05 ntdll!NtTraceControl
06 0x0
07 0x0
08 0x0
09 0x0
0a 0x0
0b 0x0
0c 0x0
kd> dS @rdx
ffff8e81`6677b570  "UserNotPresentTraceSession"
```

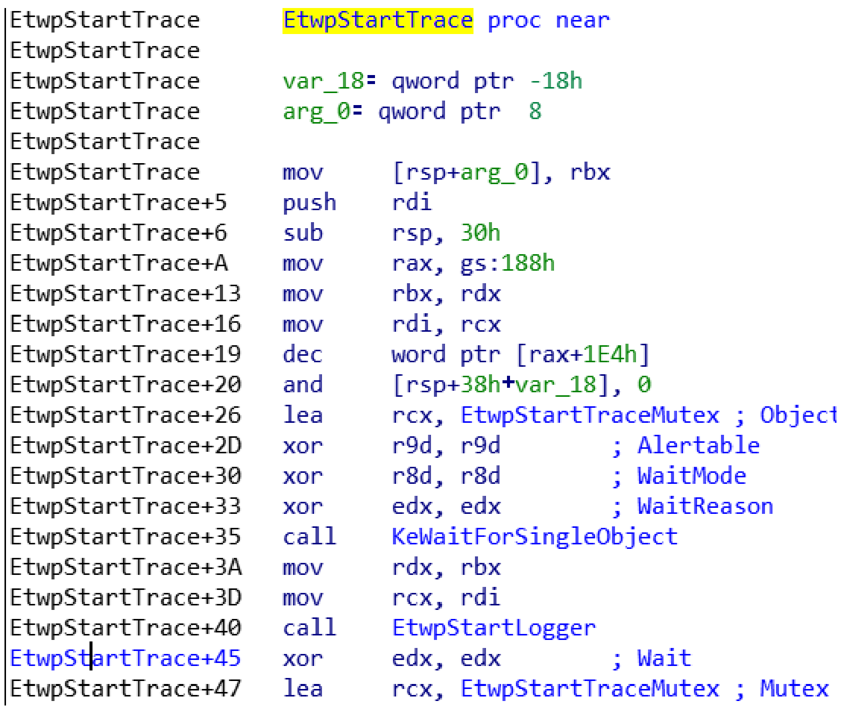
Fig. 5.37: Execution of two Windbg commands *kc* (to display the callstack) and *dS* (to display a string) after the breakpoint of *EtwpLookupLoggerIdByName* was hit.

The first one was **kc**. This prints out the call stack of the function. As can be appreciated, functions *nt!EtwpStartTrace* and *nt!EtwpStartLogger* were the callers. As the name suggests, these functions were creating and registering for the first time the loggers inside the ETW framework, confirming what was stated before.

The second one was **dS @rdx** which basically prints out a string which is unicode-encoded. In this particular case, the string "UserNotPresentTraceSession" was printed. This name is none other than the logger name of one of the several ETW sessions that are initialized by default in Windows.

After this analysis, it was possible to conclude that by setting a breakpoint inside *EtwpLookupLoggerIdByName* and comparing the logger name against the hardcoded one used for the DiagTrack session (**Diagtrack-Listener**) it was possible to detect when the logger was being created. However, the logger id will not be filled when the breakpoint is hit, but when this function ends.

The next step was to find a place (function) that will be executed once the function *EtwpLookupLoggerIdByName* finishes (so that the logger id for the Diagtrack session is already set). Analyzing the call stack shown in figure 5.37, it can be appreciated that *nt!EtwpStartTrace* would be a good candidate. Once the instruction that makes the call to *nt!EtwpStartLogger* finishes, it means that the logger id was already created and related to the corresponding session.



```
EtwpStartTrace       EtwpStartTrace proc near
EtwpStartTrace
EtwpStartTrace       var_18= qword ptr -18h
EtwpStartTrace       arg_0= qword ptr  8
EtwpStartTrace
EtwpStartTrace       mov     [rsp+arg_0], rbx
EtwpStartTrace+5     push    rdi
EtwpStartTrace+6     sub     rsp, 30h
EtwpStartTrace+A     mov     rax, gs:188h
EtwpStartTrace+13    mov     rbx, rdx
EtwpStartTrace+16    mov     rdi, rcx
EtwpStartTrace+19    dec     word ptr [rax+1E4h]
EtwpStartTrace+20    and     [rsp+38h+var_18], 0
EtwpStartTrace+26    lea     rcx, EtwpStartTraceMutex ; Object
EtwpStartTrace+2D    xor     r9d, r9d        ; Alertable
EtwpStartTrace+30    xor     r8d, r8d        ; WaitMode
EtwpStartTrace+33    xor     edx, edx        ; WaitReason
EtwpStartTrace+35    call    KeWaitForSingleObject
EtwpStartTrace+3A    mov     rdx, rbx
EtwpStartTrace+3D    mov     rcx, rdi
EtwpStartTrace+40    call    EtwpStartLogger
EtwpStartTrace+45    xor     edx, edx        ; Wait
EtwpStartTrace+47    lea     rcx, EtwpStartTraceMutex ; Mutex
```

Fig. 5.38: Disassembly of *nt!EtwpStartTrace* function obtained using IDA's disassembler.

Inspecting the assembly code of *nt!EtwpStartTrace*, it was possible to confirm that *nt!EtwpStartTrace+45* would be a very good place to set the second breakpoint as it is the immediate instruction after the execution of the aforementioned functions.

In summary, in order to finally get the logger id of the Diagtrack session the following steps should be performed:

1. Set a breakpoint at *EtwpLookupLoggerIdByName*.
2. Until the logger name (which is held in *rdx*) is not "**Diagtrack-Listener**", resume execution.
3. Once it matches, create a one-time breakpoint at *nt!EtwpStartTrace+45* and jump to it.
4. Once this second breakpoint is hit, extract the information about the logger id from the _ESERVERSILO_GLOBALS structure.

The final version of the script to accomplish this task was splitted in two parts. First part illustrates steps 1, 2 and 3. The second part implements the 4th step.

```
$$ RCX --> Pointer to ETW_SILODRIVERSTATE
$$ RDX --> Pointer to the logger name (unicode) (+0x8)

$$ Save the pointer to the logger name
r $t0 = poi(@rdx+0x8);

$$ Compare the string using alias
as /mu ${/v:LOGG_NAME} $t0;
.block{r $t10 = 0x0;r $t10 = $spat(@"${LOGG_NAME}", "Diagtrack-Listener")}
ad LOGG_NAME

$$ If it was the Diagtrack-Listener
.if($t10 == 1){
    $$ Save the pointer to ETW_SILODRIVERSTATE
    r $t1 = rcx;

    $$ The array is in offset 0x1b0 of ETW_SILODRIVERSTATE
    r $t2 = $t1+0x1b0;

    $$ Lets put a breakpoint just after the logger was created
    bp /1 nt!EtwpStartTrace+0x45 "$$><\"path_to_second_part_script"
};
gc
```

Fig. 5.39: First half of the WinDBG's script in charge of automatically getting (and storing inside a particular register) the logger id. Specifically, setting a dynamic breakpoint after finding the correct strucutre related to the Diagtack service.

```
.for(r $t19 = 0;@$t19 < 0x40;r $t19 = @$t19+1){
  $$ First, check if it is empty
  .if ( poi(poi(@$t2)+@$t19*0x8) == 1){
      .continue
  }
  .else{
      $$ Save the pointer to the WMI_LOGGER_CONTEXT in t4
      r $t4 = poi(@$t2) + (@$t19*0x8);

      $$ Save the STRING UNICODE object in t5
      r $t5 = poi(@$t4)+0x98;

      $$ Save the UNICODE buffer
      r $t6 = poi(@$t5+0x8);

      $$ Create the alias of Unicode String
      as /mu ${/v:LOGG_NAME_NEW} $t6;
      .block{r $t10 = 0x0;r $t10 = $spat(@"${LOGG_NAME_NEW}",
          "Diagtrack-Listener")}
      ad LOGG_NAME_NEW

      .if($t10 == 1){
          .printf "Logger id of Diagtrack-Listener found!!: %N\n", @$t19
          $$ <PLACE FOR FUTURE CODE >
          .break
      };
  };
};
gc
```

Fig. 5.40: Second half of the WinDBG's script in charge of automatically getting (and storing inside a particular register) the logger id. Specifically, it iterates all available ETW sessions until it finally founds the DiagTrack one. After finding the correct one, it stores DiagTrack's logger id inside register *t19* and leaves the place for future code.

5.4.2 Automating extraction of information related to services

After realizing that services isolation was critical to accurately detect the actual entity that was performing the write operation, it was necessary to build a way to automate this process.

As illustrated in section 5.3, the main function to consider was *ScGenerateServiceTag*. This function is called every time that a new service is added to the database of services. Furthermore, *ScGenerateServiceTag* has all the information needed: the name of the binary and the service index.

It was just a matter of setting a breakpoint in the correct place, in order to print both values each time a new service was added. As can be seen in figure 5.33, the offset *0x3b* seems to be a good candidate as it is the following instruction after storing the index in the *_SERVICE_RECORD* structure. In order to get the name of the service, we need to print the value stored in offset *0x10* from the *_SERVICE_RECORD* structure provided by parameter (which happens to be pointed by *rcx*). Additionally, at offset *0x3b* of *ScGenerateServiceTag*, the value of the new index is stored inside *r9d*. Therefore, one script that could be used once the breakpoint set at *ScGenerateServiceTag+0x3b* gets hit could be:

```
$$ Script that prints out the new service name and the corresponding tag
$$ Print out the placeholder
.printf "NEW_SERVICE_ADDED_START\n";
$$ Print out the name
.printf "%mu\n", poi(@rcx+0x10)
$$ Print out the tag
.printf "%N\n", r9d
$$ Print out the placeholder
.printf "NEW_SERVICE_ADDED_END\n";
$$ Continue
gc
```

Fig. 5.41: Main WinDBG's script in charge of printing out interesting information every time a new service is registered.

5.4.3 Automating extraction of events from write functions

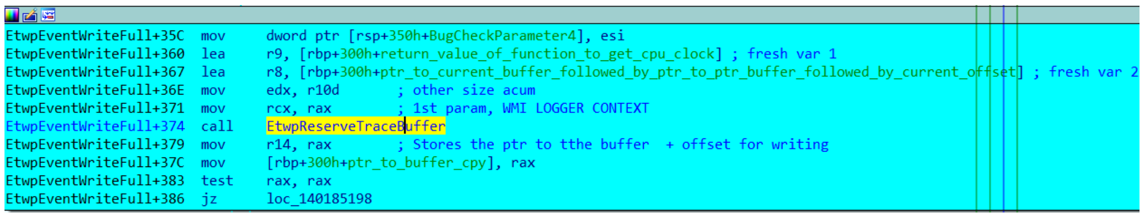
One of the most, if not the most, important analysis carried out was the one related to the writing functions. ETW providers used different functions to perform their write actions; and thanks to the analysis performed in section 5.2.1 it was possible to conclude that there were two important functions: *nt!EtwpEventWriteFull* and *nt!EtwpWriteUserEvent*. For the sake of simplicity and due to being almost identical, we will be focusing on the details of the script created for detecting writes from *nt!EtwpEventWriteFull*.

The goal of this script was to print all valuable information every time a DiagTrack event was detected:

- Timestamp
- Provider's GUID
- Call Stack
- Information about the process who was writing (including process/service name)
- The PEB
- The Event's Descriptor structure
- The Event's Payload
- Service tag if svchost.exe was the process behind the writing function

In order to ensure that the event was going to be written to the DiagTrack's session, we needed to validate that the logger id of the current write matched the DiagTrack's one. Luckily, we already knew how to perform such an action (check figure 5.24).

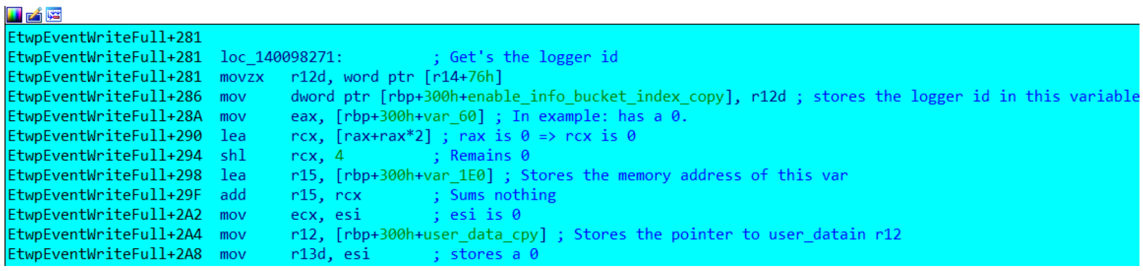
It was crucial to catch every write coming from *nt!EtwpEventWriteFull*. Nevertheless, choosing the correct place where to set the breakpoint was challenging. After an in-depth analysis of different candidates, it was concluded that the best offset would be just before calling *EtwpReserveTraceBuffer*. The main reason for choosing this offset was that it was the last instruction of *nt!EtwpEventWriteFull* where the buffers were still empty and, at the same moment, information such as the logger id was still easy to get.



```
EtwpEventWriteFull+35C mov     dword ptr [rsp+350h+BugCheckParameter4], esi
EtwpEventWriteFull+360 lea     r9, [rbp+300h+return_value_of_function_to_get_cpu_clock] ; fresh var 1
EtwpEventWriteFull+367 lea     r8, [rbp+300h+ptr_to_current_buffer_followed_by_ptr_to_ptr_buffer_followed_by_current_offset] ; fresh var 2
EtwpEventWriteFull+36E mov     edx, r10d       ; other size acum
EtwpEventWriteFull+371 mov     rcx, rax        ; 1st param, WMI LOGGER CONTEXT
EtwpEventWriteFull+374 call    EtwpReserveTraceBuffer
EtwpEventWriteFull+379 mov     r14, rax        ; Stores the ptr to tthe buffer  + offset for writing
EtwpEventWriteFull+37C mov     [rbp+300h+ptr_to_buffer_cpy], rax
EtwpEventWriteFull+383 test    rax, rax
EtwpEventWriteFull+386 jz      loc_140185198
```

Fig. 5.42: Basic block of *EtwpEventWriteFull*, extracted with IDA's disassembler, showing the offset of the last instruction executed just before calling *EtwpReserveTraceBuffer*.

Thanks to the analysis performed by the test illustrated in Figure 5.24, we already knew that when the *RIP* pointed to *nt!EtwpEventWriteFull+0x374* (chosen offset), the logger id was going to be stored inside *r12d*. Furthermore, this could be double checked by inspecting the exact offset were the assignation was carried out:



```
EtwpEventWriteFull+281
EtwpEventWriteFull+281 loc_140098271:          ; Get's the logger id
EtwpEventWriteFull+281 movzx   r12d, word ptr [r14+76h]
EtwpEventWriteFull+286 mov     dword ptr [rbp+300h+enable_info_bucket_index_copy], r12d ; stores the logger id in this variable
EtwpEventWriteFull+28A mov     eax, [rbp+300h+var_60] ; In example: has a 0.
EtwpEventWriteFull+290 lea     rcx, [rax+rax*2] ; rax is 0 => rcx is 0
EtwpEventWriteFull+294 shl     rcx, 4          ; Remains 0
EtwpEventWriteFull+298 lea     r15, [rbp+300h+var_1E0] ; Stores the memory address of this var
EtwpEventWriteFull+29F add     r15, rcx        ; Sums nothing
EtwpEventWriteFull+2A2 mov     ecx, esi        ; esi is 0
EtwpEventWriteFull+2A4 mov     r12, [rbp+300h+user_data_cpy] ; Stores the pointer to user_datain r12
EtwpEventWriteFull+2A8 mov     r13d, esi       ; stores a 0
```

Fig. 5.43: Basic block of *EtwpEventWriteFull*, extracted with IDA's disassembler, showing when and how the logger id is stored in *r12d*.

Taking into account that the current script would be executed after the one presented in the previous section (5.4.1) and therefore *$t19* would contain DiagTrack's logger id, the task of determining if the current write's destination was the DiagTrack session could be summed up in the following statement:

```
$$ Compare the current LOGGER ID with the DiagTrack one.
.if (r12d == @$t19){
  \\DO something
}
```

Fig. 5.44: First version of the WinDBG's script in charge of automatically extracting interesting inforation from the moment a provider is writing an event through calling *EtwpEventWriteFull*.

Now that it is known how to be sure whether or not a write call is interesting in our context, it was time to extract the information out of it. At this point there are at least two things out of the list of valuable information, that can be extracted:

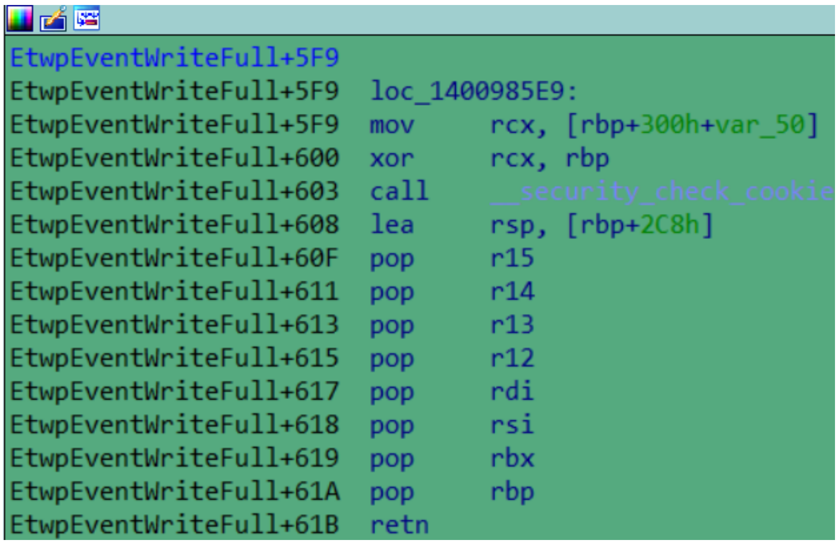
1. The GUID_ENTRY
2. The Event Descriptor

The GUID_ENTRY was stored at *rbp+300h-280h* and the EVENT_DESCRIPTOR at *rbp+300h-2C0h*. Therefore, we could add this logic to the current version of the script:

```
$$ Compare the current LOGGER ID with the DiagTrack one.
.if (r12d == @$t19){
  $$ In this function, the guid entry is situated in [rbp+300h-280h]
  r $t1 = poi(rbp+300h-280h);

  $$ In this function, the event_descriptor is situated in [rbp+300h-2C0h]
  r $t2 = poi(rbp+300h-2C0h);

  // Do more things
}
```

Fig. 5.45: Second version of the WinDBG's script in charge of automatically extracting interesting inforation from the moment a provider is writing an event through calling *EtwpEventWriteFull*.

Finally, as one piece of information that was needed to be extracted from this function was the actual content of the event being logged, it was necessary to continue with the execution of *EtwpEventWriteFull*, at least, until *EtwpReserveTraceBuffer* finished. In order to be sure that everything had already executed, it was decided to dynamically set a new breakpoint at the last basic block of *EtwpEventWriteFull*. Setting this breakpoint at that point will ensure that the function is finished and most of the information is ready to be read. That was why it was decided to set the breakpoint at *EtwpEventWriteFull+0x600*:



```
EtwpEventWriteFull+5F9
EtwpEventWriteFull+5F9  loc_1400985E9:
EtwpEventWriteFull+5F9  mov     rcx, [rbp+300h+var_50]
EtwpEventWriteFull+600  xor     rcx, rbp
EtwpEventWriteFull+603  call    __security_check_cookie
EtwpEventWriteFull+608  lea     rsp, [rbp+2C8h]
EtwpEventWriteFull+60F  pop     r15
EtwpEventWriteFull+611  pop     r14
EtwpEventWriteFull+613  pop     r13
EtwpEventWriteFull+615  pop     r12
EtwpEventWriteFull+617  pop     rdi
EtwpEventWriteFull+618  pop     rsi
EtwpEventWriteFull+619  pop     rbx
EtwpEventWriteFull+61A  pop     rbp
EtwpEventWriteFull+61B  retn
```

Fig. 5.46: Last basic block of function *EtwpEventWriteFull*, obtained using IDA's disassembler, shown in order to highlight the best offset where to set the breakpoint to extract information.

For the sake of simplicity, it was decided to split the script in two. The "main" script which would contain all the logic already explained and the "inner script" which would contain the logic of the extraction of the rest of information.

```
    $$ Compare the current LOGGER ID with the DiagTrack one.
    .if (r12d == @$t19){
      $$ In this function, the guid entry is situated in [rbp+300h-280h]
      r $t1 = poi(rbp+300h-280h);

      $$ In this function, the event_descriptor is situated in [rbp+300h-2C0h]
      r $t2 = poi(rbp+300h-2C0h);

      $$ So, once we know that the provider is going to ask for buffers related
      $$ with the telemetry, we jump to a part of the same function where the
      $$ event should be already written to the buffers, and therefore we can
      $$ print out the valuable information.
      bp /1 nt!EtwpEventWriteFull+0x600 "$$><\"${$arg1}\""
      gc;
  }
  .else{
      gc;
  }
```

Fig. 5.47: Final version of the WinDBG's script in charge of automatically extracting interesting information from the moment a provider is writing an event through calling *EtwpEventWriteFull*.

The "main" script only extracted two out of the seven elements that we wanted to extract from the write function. Therefore, it still needed to extract:

- Timestamp
- Provider's GUID
- Call Stack
- Information about the process who was writing (including process/service name)
- The PEB
- The Event's Payload
- Service tag if svchost.exe was the process behind the writing function

Some of them were easy:

- Timestamp: Only needed to execute the command **.echotime**
- Call Stack: Only needed to execute the command **kc**
- The PEB: Only needed to execute the command **!peb**
- Information about the process: Only needed to execute the command **!process -1 0**

But...what about the rest?

5.4.3.1 Extracting Provider's GUID

The provider GUID extraction was not challenging. We already had the pointer to the GUID_ENTRY stored in register *t1* (check figure 5.45). Recalling the full layout of the structure (figure 5.13), it can be easily seen that the provider's GUID was at *t1+0x18* and *t1+0x18+0x8* (due to being two words long).

In a nutshell, the following line could be used to extract the full provider GUID:

```
    .printf "%N %N\n", poi(@$t1+0x18), poi(@$t1+0x18+0x8);
```

Fig. 5.48: WingDBG command printing the Provider's GUID out of a structure *_GUID_ENTRY* stored in register *t1*.

5.4.3.2 Extracting the Event's Payload

By making use of the **wmitrace** Windbg's extension and due to being already completed because of the execution of *EtwpReserveTraceBuffer*, it was possible to read the information logged by reading the ETW buffers. For the sake of simplicity and correctness some useful aliases were used:

```
    as /x ${/v:LOGGER_ID} @$t19;
    .block{!wmitrace.logdump ${LOGGER_ID} -t 1}
    ad LOGGER_ID
```

Fig. 5.49: WingDBG command printing the event's payload using the *wmitrace* plugin.

5.4.3.3 Extracting the service tag if needed

One of the biggest challenges was the presence of **svchosts.exe** processes. As they may hold several services inside, it was needed to implement a way to detect which of all the possible services was behind the writing function. With the isolation of services technique explained in section 5.3, it was possible to match an index with the corresponding service name. However, we needed to get the service tag out of the process information. Recalling what was explained in section 5.3, this information was stored inside the TEB (Thread Environment Block).

In simple steps we needed to:

1. Extract the process name out of the process information.
2. Determine (by comparison) if the process name was "svchost.exe".
3. If true, read the TEB of the process and extract the tag.
4. Print the service tag.

```
1    .foreach /pS 13 (proc_name {!process -1 0 }) { aS ${/v:ProcessName}
        "${proc_name}" };
2    .block{r $t10 = 0x0;r $t10 = $spat("${ProcessName}", "*svchost.exe*")}
3    .if ($t10 == 1){
4       .foreach /pS 5 (teb_addr {!thread @thread }) { r $t12 = teb_addr ;.break }
5       .printf "SERVICE_TAG_START\n"
6       .foreach /pS 1 (tag { dd @$t12+0x1720 L1 }) { .echo tag}
7       .printf "SERVICE_TAG_END\n"
8    }
```

Fig. 5.50: WingDBG snippet in charge of printing the process name. In case the process is "svchost.exe", it will print out the service tag out of the Thread Environment Block.

- Line **1** solves the first step.
- Line **2** solves the second step.
- Lines **3 and 4** solve the third step (check figure 5.30).
- Lines **5, 6 and 7** solve the fourth step.

It was time to complete the puzzle. Putting together all snippets developed to extract each part the final script would be finished:

```
r $t11 = poi(rbp+300h-280h);
r $t12 = poi(rbp+300h-2C0h);

$$ Just to ensure that we are still in the same write
.if ((@$t11 == @$t1) & (@$t12 == @$t2)){
    .reload
    .printf "WRITE_INFO_START\n";

    $$ If the process is in svchost, we want to have the service related to
        the actual thread.
    $$ In order to do that, we should print the tag of the service related to
        the tread and
    $$ afterwards compare it again to the dump of the services db.

    .foreach /pS 13 (proc_name {!process -1 0 }) { aS ${/v:ProcessName}
        "${proc_name}" };
    .block{r $t10 = 0x0;r $t10 = $spat("${ProcessName}", "*svchost.exe*")}
    .if ($t10 == 1){
        .foreach /pS 5 (teb_addr {!thread @thread }) { r $t12 = teb_addr
            ;.break }
        .printf "SERVICE_TAG_START\n"
        .foreach /pS 1 (tag { dd @$t12+0x1720 L1 }) { .echo tag}
        .printf "SERVICE_TAG_END\n"
    }

    .printf "DATE_TIMESTAMP_START\n";
    .echotime;
    .printf "DATE_TIMESTAMP_END\n";
    .printf "PROVIDER_GUID_START\n"
    .printf "%N %N\n", poi(@$t1+0x18), poi(@$t1+0x18+0x8);
    .printf "PROVIDER_GUID_END\n"
    .printf "CALL_STACK_START\n";
    kc;
    .printf "CALL_STACK_END\n";
    .printf "PROCESS_INFO_START\n";
    !process -1 0;
    .printf "PROCESS_INFO_END\n";
    .printf "PEB_INFO_START\n";
    !peb;
    .printf "PEB_INFO_END\n";
    .printf "EVENT_DESCRIPTOR_START\n";
    dt nt!_EVENT_DESCRIPTOR @$t2;
    .printf "EVENT_DESCRIPTOR_END\n";
    .printf "EVENT_JSON_FORMAT_START\n";
    as /x ${/v:LOGGER_ID} @$t19;
    .block{!wmitrace.logdump ${LOGGER_ID} -t 1}
    ad LOGGER_ID
    .printf "EVENT_JSON_FORMAT_END\n";
    .printf "WRITE_INFO_END\n";
    gc;
}
.else{
    gc;
}
```

Fig. 5.51: Final version of the WinDBG's inner script in charge of printing out all the interesting information from a write performed through calling *EtwpEventWriteFull*. It is basically a combination of all the aforementioned snippets.

This inner script is going to be "called" from the main script illustrated in figure 5.47. However, this main script has to be also called from somewhere. The last step that was necessary to perform in order to chain all scripts explained in this section, was to add the main *EtwpeventwriteFull* extraction script inside the DiagTrack's logger id search script depicted in figure 5.40. In particular, the "$$ <*PLACE FOR FUTURE CODE*>" placeholder should be replaced with:

```
    $$ Once we find this, we should create the breakpoint for the other script.
    $$ If the breakpoint is already created, it does not matter, it will do
        anything.
    $$ But it is totally worthless to set the breakpoint before this happens.
    bp nt!EtwpEventWriteFull+0x371 "$$>a<\"${$arg1}\" \"${$arg2}\""
    bp nt!EtwpWriteUserEvent+0x48d "$$>a<\"${$arg3}\" \"${$arg4}\""
```

Fig. 5.52: Illustration on how the final part of the WinDBG's script depicted by Figure 5.40 should be replaced. This basically chains the execution of the first script (in charge of looking for the logger id) with the scripts that print out interesting information out of a writing call.

Two important disclaimers:

1. The offset *0x371* is obtained from the explanation of 5.42.
2. Although not being shown here, the same analysis performed with *EtwpeventwriteFull* was carried out with function *EtwpWriteUserEvent*.

Main and inner scripts for extracting information from *EtwpWriteUserEvent* are shown below:

```
$$
$$ ARG1: Path to the internal script for this function
$$
$$ Compare the current LOGGER ID with the DiagTrack one.
.if (ebx == @$t19){
   $$ In this function, the guid entry is situated in [rbp+360h-340h]
   r $t1 = poi(rbp+360h-340h);

   $$ In this function, the event_descriptor is situated in [rbp+360h-330h]
       + 0x28
   r $t2 = poi(rbp+360h-330h)+0x28;

   $$ So, once we know that the provider is going to ask for buffers related
   $$ with the telemetry, we jump to a part of the same function where the
   $$ event should be already written to the buffers, and therefore we can
   $$ print out the valuable information.
   bp /1 nt!EtwpWriteUserEvent+0xd47 "$$><\"${$arg1}\""
   gc;
}
.else{
   gc;
}
```

Fig. 5.53: Main WinDBG's script in charge of setting the breakpoint inside the *EtwpWriteUserEvent* in order to later extract the interesting information out of the writing call.

```
r $t11 = poi(rbp+360h-320h);
r $t12 = poi(rbp+360h-328h)+0x28;

$$ Just to ensure that we are still in the same write
.if ((@$t11 == @$t1) & (@$t12 == @$t2)){
    .reload
    .printf "WRITE_INFO_START\n";

    $$ If the process is in svchost, we want to have the service related to
        the actual thread.
    $$ In order to do that, we should print the tag of the service related to
        the tread and
    $$ afterwards compare it against the dump of the "services db".

    .foreach /pS 13 (proc_name {!process -1 0 }) { aS ${/v:ProcessName}
        "${proc_name}" };
    .block{r $t10 = 0x0;r $t10 = $spat("${ProcessName}", "*svchost.exe*")}
    .if ($t10 == 1){
        .foreach /pS 5 (teb_addr {!thread @thread }) { r $t12 = teb_addr
            ;.break }
        .printf "SERVICE_TAG_START\n"
        .foreach /pS 1 (tag { dd @$t12+0x1720 L1 }) { .echo tag}
        .printf "SERVICE_TAG_END\n"
    }

    .printf "DATE_TIMESTAMP_START\n";
    .echotime;
    .printf "DATE_TIMESTAMP_END\n";
    .printf "PROVIDER_GUID_START\n"
    .printf "%N %N\n", poi(@$t1+0x18), poi(@$t1+0x18+0x8);
    .printf "PROVIDER_GUID_END\n"
    .printf "CALL_STACK_START\n";
    kc;
    .printf "CALL_STACK_END\n";
    .printf "PROCESS_INFO_START\n";
    !process -1 0;
    .printf "PROCESS_INFO_END\n";
    .printf "PEB_INFO_START\n";
    !peb;
    .printf "PEB_INFO_END\n";
    .printf "EVENT_DESCRIPTOR_START\n";
    dt nt!_EVENT_DESCRIPTOR @$t2;
    .printf "EVENT_DESCRIPTOR_END\n";
    .printf "EVENT_JSON_FORMAT_START\n";
    as /x ${/v:LOGGER_ID} @$t19;
    .block{!wmitrace.logdump ${LOGGER_ID} -t 1}
    ad LOGGER_ID
    .printf "EVENT_JSON_FORMAT_END\n";
    .printf "WRITE_INFO_END\n";
    gc;
}
.else{
    gc;
}
```

Fig. 5.54: WinDBG's Inner script in charge of printing out interesting information out of a writing call performed through *EtwpWriteUserEvent*. This script is analogous to the one depicted in figure 5.51.

5.5 Triggers

5.5.1 Searching for new triggers

In previous sections, we put a lot of focus on explaining how the different ETW providers were writing events to the ETW sessions. However, a very important detail was overlooked. Despite the fact that knowing where the provider is going to write is critical, if we would set a breakpoint just before the execution of the writing function, we would still need to wait until the provider actually had to write. An analysis without being able to control when the action that you want to inspect happens, it is really difficult to carry out.

That is the reason why it was decided to build what we call "Triggers". Developed using different languages, once these binaries are executed, they force specific providers to log an event to the DiagTrack ETW Session. Having developed these binaries allowed us to do a better and more accurate analysis in less time.

By the end of this research, we had developed around five or six different triggers. Even though having their own particularities, some of these triggers were found using the same strategies. The following sections will describe in detail the most representative Triggers found during the project.

5.5.2 Notepad

This trigger was one of the very first ones found. In order to find it, a breakpoint was set at the beginning of *EtwpEventWriteFull*. After several minutes using the debugee computer as a normal user, the breakpoint got hit.

Once inside *EtwpEventWriteFull* and with the help of WinDBG (section 2.2.2) it was possible to recover the full call stack of that write and understand which was the initial program/class/object that originated the call.

After analyzing this behavior several times, it was possible to conclude that just with opening and closing a Notepad process, some ETW providers will write an event to the DiagTrack's ETW Session. However, this was not always the case. Depending on the level of Telemetry configured 0, 1 or 2 providers will write to the ETW session. The following table illustrates this behavior with some extra information:

Telemetry Level	Number of events logged	Providers Guid
Security	0	-
Basic	1	487D6E37-1B9D-46D3-A8FD-54CE8BDF8A53
Enhanced	1	487D6E37-1B9D-46D3-A8FD-54CE8BDF8A53
Full	9	487D6E37-1B9D-46D3-A8FD-54CE8BDF8A53 2839FF94-8F12-4E1B-82E3-AF7AF77A450F

The following listing is an extraction of the information gathered while capturing a write using this trigger:

```
    PROVIDER_GUID: 46D31B9D487D6E37 538ADF8BCE54FDA8
        nt!EtwpEventWriteFull+0x371:

    PROCESS ffffdd868344c480
    SessionId: 1 Cid: 0774  Peb: ffffdd868344c480; !peb
        821eee7000">821eee7000 ParentCid: 0e68">0e68
    DirBase: 53cc5000 ObjectTable: ffffc7004a4e8b00 HandleCount: <Data Not
        Accessible>
    Image: notepad.exe

    +0x000 Id             : 0x8bd
    +0x002 Version        : 0 ''
    +0x003 Channel        : 0xb ''
    +0x004 Level          : 0x5 ''
    +0x005 Opcode         : 0 ''
    +0x006 Task           : 0
    +0x008 Keyword        : 0x00008000`00000800
    fffff802`6ceab361 488bc8       mov    rcx,rax

    kd> !wmitrace.logdump 0x22 -t 1
    [0]1310.0FF8:: 131717129297892613
        [Win32kTraceLogging/AppInteractivitySummary/]{"UTCReplace_AppId":
        "{00001310-0001-00f[redacted]}
```

Fig. 5.55: Output from a particular WinDBG script showing the Provider GUID, process information, event descriptor and event payload gathered after executing the Notepad trigger.

In order to help us with future automatization of these test, a super simple C# script which mimics this same action was developed.

```
using System;
using System.Diagnostics;
namespace TriggerNotepad
{
    class Program
    {
        static void Main(string[] args)
        {
            Process.Start("notepad.exe");
            String cmd = "/C taskkill /im notepad.exe";
            System.Threading.Thread.Sleep(2000);


            Process process = new Process();
            ProcessStartInfo startInfo = new ProcessStartInfo();
            startInfo.WindowStyle = ProcessWindowStyle.Hidden;
            startInfo.FileName = "cmd.exe";
            startInfo.Arguments = cmd;
            process.StartInfo = startInfo;
            process.Start();
        }
    }
}
```

Fig. 5.56: C# source code of the Notepad trigger

5.5.3 Census

While looking for information about possible events that would be logged to DiagTrack's ETW Session on the Internet, a very interesting web resource from Microsoft was found[8]. Inside this documentation there was a dedicated section for Census events. These kinds of events were related to applications installed inside the system, hardware information, memory information and more.

After analyzing this information it was found that just executing the DeviceCensus.exe binary which comes by default inside the system32 Windows' directory, one event was being written to the DiagTrack's ETW Session. As explained in the previous trigger, this also would depend on the level of Telemetry configured. The following table explains for each level, which ETW providers will write a new event once this trigger is executed:

Telemetry Level	Number of events logged	Providers Guid
Security	0	-
Basic	1	262CDE7A-5C84-46CF-9420-94963791EF69
Enhanced	1	262CDE7A-5C84-46CF-9420-94963791EF69
Full	1	262CDE7A-5C84-46CF-9420-94963791EF69

The following listing is an extraction of the information gathered while capturing a write using this trigger:

```
PROVIDER_GUID: 46CF5C84262CDE7A 69EF913796942094

PROCESS ffff8e0d8a6e9080
    SessionId: 1 Cid: 0f58  Peb: f0276b9000 ParentCid: 04e4
    DirBase: 67a88000 ObjectTable: ffffc1882c3fa580 HandleCount: <Data Not
        Accessible>
    Image: DeviceCensus.exe

START_EVENT_DESCRIPTOR
   +0x000 Id            : 0x1fe
   +0x002 Version       : 0 ''
   +0x003 Channel       : 0xb ''
   +0x004 Level         : 0x5 ''
   +0x005 Opcode        : 0 ''
   +0x006 Task          : 0
   +0x008 Keyword       : 0x00008000`00000000
END_EVENT_DESCRIPTOR
START_EVENT_JSON_FORMAT
(WmiTrace) LogDump for Logger Id 0x21
Found Buffers: 4 Messages: 18, sorting entries
[0]0F58.133C:: 131728645762917499 [Census/App/]{"__TlgCV__":
    "X0BX596o1kyBj176.0", "IEVersion": "11.0.14393.0", "CensusVersion":
    100143930000}
Total of 18 Messages from 4 Buffers
```

Fig. 5.57: Output from a particular WinDBG script showing the Provider GUID, process information, event descriptor and event payload gathered after executing the Census trigger.

The following listing contains the source code of the binary developed to be used as the Census trigger.

```
using System;
using System.Diagnostics;
using System.Runtime.InteropServices;

namespace triggerCensus
{
    class Program
    {
        static void Main(string[] args)
        {
            Process process = new Process();
            ProcessStartInfo startInfo = new ProcessStartInfo();
            startInfo.WindowStyle = ProcessWindowStyle.Hidden;
            startInfo.FileName = "cmd.exe";
            startInfo.Arguments = "/C
                C:\\Windows\\System32\\DeviceCensus.exe";
            process.StartInfo = startInfo;
            process.Start();
            process.WaitForExit();
        }
    }
}
```

Fig. 5.58: C# Source code of Census trigger

5.5.4 One Drive

Another interesting trigger that was possible to develop thanks to the Microsoft Documentation[8] was the One Drive one. This trigger, although being a bit unstable, was developed because it had a very interesting capability: An event is being logged even when the Security level of Telemetry is configured. However, after analyzing and investing quite a lot of time understanding the behavior of this trigger it was not possible to develop a stable version. In other words, we were not able to understand which conditions it needs in order to work.

In order to make use of this trigger, the binary OneDriveStandaloneUpdater.exe was used. The following listing is an extraction of the information gathered while capturing a write using this trigger:

Telemetry Level	Number of events logged	Providers Guid
Security	1	D34D654D-584A-4C59-B238-69A4B2817DBD
Basic	1	D34D654D-584A-4C59-B238-69A4B2817DBD
Enhanced	1	D34D654D-584A-4C59-B238-69A4B2817DBD
Full	1	D34D654D-584A-4C59-B238-69A4B2817DBD

The following listing is an extraction of the information gathered while capturing a write using this trigger:

```
PROVIDER_GUID: 4C59584AD34D654D BD7D81B2A46938B2 nt!EtwpWriteUserEvent+0x48d:

PROCESS ffff8002598ac080
   SessionId: 1 Cid: 1240  Peb: 0040d000 ParentCid: 0370
   DirBase: 4e24e000 ObjectTable: ffff9301e7f9d300 HandleCount: <Data Not
       Accessible>
   Image: OneDriveStandaloneUpdater.exe

   +0x000 Id             : 0xa7
   +0x002 Version        : 0 ''
   +0x003 Channel        : 0xb ''
   +0x004 Level          : 0x5 ''
   +0x005 Opcode         : 0 ''
   +0x006 Task           : 0
   +0x008 Keyword        : 0x00008000`00000000
   fffff801`b080295d 418bd7       mov    edx,r15d
```

Fig. 5.59: Output from a particular WinDBG script showing the Provider GUID, process information, event descriptor and event payload gathered after executing the OneDrive trigger.

The following source code references to the program developed in order to help with the automation of the OneDrive trigger:

```csharp
   using System;
   using System.Diagnostics;
   using System.Runtime.InteropServices;

   namespace onedrivestandalonetrigger
   {
       class Program
       {
           static void Main(string[] args)
           {
               Process process = new Process();
               ProcessStartInfo startInfo = new ProcessStartInfo();
               startInfo.WindowStyle = ProcessWindowStyle.Hidden;
               startInfo.FileName = "cmd.exe";
               string userNameAndHost =
                   System.Security.Principal.WindowsIdentity.GetCurrent().Name;
               int indexOfSlash = userNameAndHost.IndexOf("\\");
               string userName = userNameAndHost.Substring(indexOfSlash+1);
               startInfo.Arguments = "/C C:\\Users\\"+ userName +
                   "\\AppData\\Local\\Microsoft\\OneDrive\\OneDriveStandaloneUpdater.exe";
               process.StartInfo = startInfo;
               process.Start();
               process.WaitForExit();
           }
       }
   }
```

Fig. 5.60: C# Source code of OneDrive trigger

6. CONCLUSIONS

In this project we analyzed how Windows Telemetry works from an internal perspective. Due to being a part of the Windows Kernel and its source code being closed, a combination of several techniques had to be used in order to understand how this component operated. Reverse Engineering and Kernel Debugging were the main two techniques that allowed us to dig deeply into Telemetry's heart and led us to understand which and how its internal processes and tasks were carried out. Moreover, by unveiling complex Kernel structures and developing several scripts to automate information gathering, we set a base that will simplify investigations and therefore be leveraged by future works either for the same or different components.

Reverse Engineering is a powerful tool. Despite the fact that Windows Kernel source code is closed, it was possible to understand the internal functionality of several components by being able to read its assembly code. This means that usually, with appropriate time, no matter which process, service, application or executable you are trying to analyze, you will be able to understand how it works and operates even without its source code. In some cases, the analysis could get harder or even impossible when obfuscation techniques are in place, however this is not the case of the Windows Kernel. Finally, combining Reverse Engineering with Dynamic Analysis (being able to debug) makes things even easier as it is possible to stop at some particular points and inspect the current state of the machine with the goal of better understanding the current situation.

Throughout the journey, several challenges came up. One of them being the usage of Kernel structures that were not documented at all. As mentioned in the introduction, reading and analyzing assembly code is more difficult than reading high-level code. Hence, the process of learning how some structures are used when they are unknown becomes even more difficult. Understanding their definition, their offsets or even the semantics of each pointer stored within them, was one of the most difficult but most rewarding parts of the whole investigation. We believe that having unveiled all these structures could help others avoid going through the same process.

Throughout the journey, various challenges arose. One of them is the use of kernel structures that were not documented at all. As mentioned in the introduction, reading and parsing assembly code is more difficult than reading high-level code. Therefore, the process of learning how some structures are used when they are unknown becomes even more difficult. Understanding their definition, their offsets, or even the semantics of each pointer stored within them, was one of the most difficult but most rewarding parts of the entire discussion. We believe that having unveiled all these structures could help others avoid going through the same process.

Another big challenge faced and overcame was one related to detecting the process that was writing an event. The ability of **svchost.exe** to "hide" several services inside of it, was prohibiting us to accurately detect the entity behind a submission of a new event. As explained in section 5.3, after trying several options, it was possible to find a way to detect which service was running at some particular point (even in the case of a write coming from **svchost.exe**) by inspecting two things: the process' Thread Environment Block (TEB) and services' tag database. Based on the research performed against the services.exe executable and its tag assignment implementation, we understood how different services were being assigned with tags once they were registered.

Leveraging the power of Windbg and its script language, it was possible to create an automatic tool for event analysis. By combining all the Windbg scripts presented, we created an automated tool capable of catching and analyzing events logged by ETW providers, just before being actually written to the trace buffers. Despite the fact that in order to be able to use this tool Kernel debugging should be activated, being able to have control execution right before the event is being logged gives a lot of power and flexibility.

Finally, the development of Telemetry Triggers were key during the whole project. By being able to perform dynamic analysis, and thus, read the call stack when an ETW write happened, it was possible to understand which requirements need to be met in order to force particular ETW providers to write an event. Developing these "triggers" allowed us to perform innumerable analysis and tests in a controlled way due to knowing who, when and what they were logging.

7. FUTURE WORK

As stated in section 6, one of the goals of this project was to set a base, contribute and help future investigations.

Having explained how to combine Reverse Engineering and Kernel Debugging with the goal of performing an analysis of an OS Kernel, we expect to encourage and let other researchers or enthusiastics to replicate these techniques to better understand other components either in the same or different OS. Additionally, all kernel structures and functionalities reversed in this project could assist security researchers to better understand the details of Windows Telemetry or ETW and therefore help them with the process of assessing their overall security. For instance, it could help find a memory corruption or any other potential vulnerability which could end up in a serious flaw affecting the Windows OS.

We believe that the development of the Windbg scripts could inspire people into creating new and innovative projects just by modifying or expanding the current features. Despite the fact that all Windbg scripts developed during this work were devoted to analyzing the **DiagTrack-Listener** ETW session, they were also thought of with the idea of low coupling in mind. As an example, a simple change in the name of the Logger being compared in 5.39 (line 9), allows to perform the same analysis with any other arbitrary ETW session. In more general terms, this project was devoted to understanding how Telemetry worked from an internal perspective. However, there are other types of analysis that could be carried out against Telemetry, where our work could help to perform them. For example, privacy analysis of the data contained inside the events that are actually being sent to the backend systems. Although several previous works already focused on that part, by leveraging the ability of knowing how Telemetry stores them, would allow it to automatically inspect them even before being sent. As a consequence, several tools such as an "Event Blocker" tool could be developed. Another interesting example could be: To perform a security analysis against the way events are sent to the backend servers. Having the power to break and modify the content of an event just before being sent could provide, at least, two different features. First, it could be used to anonymize or obfuscate some piece of information that for some reason somebody wants to hide. Second, it could provide a way to send arbitrary content and therefore check for security issues in the way the backend server parses the events.

Finally, the developed "triggers", although may look not too crucial, could be key to further investigations. First and foremost, they provide a way to easily perform tests by forcing new events to be written to the corresponding ETW session. By doing that, they could ease the work of a researcher who is trying to perform an analysis of the events. Furthermore, they provide different examples of processes/binaries (ETW providers) that are actually writing into the ETW session. Therefore, they could be used as a base to get patterns out of them with the goal of building a model of an ETW provider (related to Telemetry or not).

8. APPENDIXES

Simple .exe that creates a new file

Written in C++.

```cpp
#include "pch.h"
#include <iostream>
#include <fstream>

int main()
{
    std::ofstream outfile("C:\\Users\\<user>\\Desktop\\test.txt");
    outfile << "Hello world!\n";
    outfile.close();
}
```

Fig. 8.1: C# source code of the simple program used to be executed as a service.

BIBLIOGRAPHY

[1] Jaehyeok Han, Jungheum Park, Hyunji Chung and Sangjin Lee. Forensic analysis of the Windows telemetry for diagnostics. 2020.

[2] Faris Anis Khasawneh. OS Call Home: Background Telemetry Reporting in Windows 10. 2019.

[3] Ministry of Justice and Security Strategic Vendor Management Microsoft. DPIA Windows 10 Enterprise v.1809 and preview v. 1903. 2019.

[4] Tarik Soulami. Inside Windows debugging. Chapter 12, 2012

[5] Hausi A. Miiller, Jens H. Jahnke, Kenny Wong ,Dennis B. Smith , Scott R. Tilley , Margaret-Anne Storey. Reverse Engineering: A Roadmap. In Proceedings of the Conference on The Future of Software Engineering, Pages 47-60, 2000.

[6] Bruce Dang, Alexandre Gazet, Sbastien Josse, Elias Bachaalany. Practical Reverse Engineering: x86, x64, ARM, Windows Kernel, Reversing Tools, and Obfuscation. 2014.

[7] Online Microsoft Documentation about ETW: https://docs.microsoft.com/en-us/windows/win32/etw/event-tracing-portal. 2021.

[8] Online Microsoft Documentation about Events that are logged to ETW Session: https://docs.microsoft.com/en-us/windows/privacy/basic-level-windows-diagnostic-events-and-fields-1703. 2022.