



UNIVERSIDAD DE BUENOS AIRES
FACULTAD DE CIENCIAS EXACTAS Y NATURALES
DEPARTAMENTO DE COMPUTACIÓN

Estudiando las capacidades de logging de Windows Telemetry usando Ingenieria Inversa

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.

AGRADECIMIENTOS

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. Motivación	7
2. Introducción	8
2.1 Conceptos básicos	8
2.1.1 Ingeniería Inversa	8
2.1.2 Debugging	10
2.2 Herramientas	10
2.2.1 IDA pro	10
2.2.2 WinDBG	11
2.2.3 XPERF y Message Analyzer	11
2.3 Componentes de Windows	11
2.3.1 Event Tracing for Windows	12
2.3.1.1 Arquitectura de ETW	12
2.3.1.2 Sesiones de ETW	12
2.3.1.3 Controladores de ETW	13
2.3.1.4 Proveedores de ETW	13
2.3.1.5 Consumidores de ETW	13
2.3.2 Telemetría	14
3. Configuración de laboratorio	15
3.1 Creando el entorno	15
3.2 Configuración del debuggee	16
3.3 Configuración del debugger	17
4. Trabajo Previo	18
5. Análisis del componente y metodología	19
5.1 Entendiendo cómo Telemetría hace uso de ETW	19
5.1.1 Analizando el proceso de registración con ingeniería inversa	22
5.1.1.1 Entendiendo la estructura de la función	23
5.1.1.2 Chequeando si el GUID del proveedor ya existe	25
5.1.1.3 Si el GUID no se encontró, crear uno nuevo	29
5.1.1.4 Devolver el handler	31
5.2 Interponiéndose entre la escrituras de los proveedores	33
5.2.1 Funciones de escritura en ETW	34
5.2.2 Asegurando que los proveedores escriben a la sesión del Diagtrack	35
5.2.2.1 Inspeccionando estructuras de ETW	36
5.2.2.2 GUID del proveedor y GUID de grupo	37
5.2.2.3 Identificando la sesión destino	38
5.2.3 Detectando qué se escribe	40
5.2.4 Asegurando la correctitud de los eventos loggeados	42
5.3 Analizando servicios	45
5.4 Automatizando	49

5.4.1	Automatizando la búsqueda del logger id	49
5.4.2	Automatizando la extracción de información relacionada a los servicios	54
5.4.3	Automatizando la extracción de eventos en funciones de escritura . .	55
5.4.3.1	Extracción del GUID de los proveedores	59
5.4.3.2	Extracción del contenido de los eventos	59
5.4.3.3	Extracción del tag del servicio	60
5.5	Triggers	65
5.5.1	Buscando nuevos triggers	65
5.5.2	Notepad	65
5.5.3	Census	67
5.5.4	One Drive	69
6.	Conclusiones	72
7.	Trabajo futuro	74
8.	Apéndices	75

1. MOTIVACIÓN

El análisis del componente Telemetría del SO Windows presentado en este trabajo, fue llevado a cabo con los siguientes objetivos:

- Entender como funciona el proceso de loggear logs.
- Responder a las preguntas de ¿Qué?, ¿Cómo? y ¿Dónde? se guardan los eventos.
- Entender qué aplicaciones estan involucradas en el proceso de obtención de información por parte de Telemetría
- Proveer una fuente integral de información de las estructuras de kernel involucradas.
- Descubrir y documentar como procesos y estructuras de Kernel funcionan utilizando Ingeniería inversa con el fin de contribuir con la comunidad con información no publicada anteriormente y ayudar a futuros trabajos relacionados (o no) al mismo tema.

2. INTRODUCCIÓN

El análisis presentado en este proyecto tiene como objetivo comprender mejor e ilustrar cómo funciona Telemetría en Windows desde una perspectiva interna. Mostraremos cómo, incluso en situaciones difíciles donde la documentación es escasa y/o el código fuente no está disponible públicamente, aún es posible comprender la forma en que funciona un componente. Al aprovechar el uso de técnicas avanzadas como ingeniería inversa y la depuración del kernel, demostraremos cómo Telemetría en Windows genera, almacena y envía datos desde el sistema. Además, se compartirán varios scripts desarrollados para automatizar la extracción de datos con el objetivo de proporcionar una base para futuros proyectos que puedan aprovecharlos para realizar análisis en los mismos temas o similares.

Esta investigación se llevó a cabo en una versión específica del SO Windows: Windows 10 64 bits Enterprise, 1607. Windows publica actualizaciones de su Sistema Operativo, por lo general, dos veces al año. Originalmente, este proyecto comenzó en 2018 con una pasantía en ERNW GmbH. En ese momento, ERNW trabajaba en colaboración con la Oficina Federal de Seguridad de la Información. Hubo varias razones por las que se eligió esta versión específica:

- Era una de las versiones principales de Windows en el momento de iniciar el proyecto.
- Era una versión con soporte de larga duración (EOS: April 2019).
- Era utilizado por la Oficina Alemana de la Policía.

Aunque esta versión puede parecer un poco antigua a día de hoy, todo el análisis también es aplicable a versiones más nuevas como Windows 10 64 bits Enterprise 1909. Esto se confirmó utilizando la versión final de los scripts que se desarrollaron durante este proyecto y ajustando algunos detalles mínimos sobre los offsets donde se establecieron los breakpoints.

Las siguientes secciones presentarán varios conceptos básicos, herramientas y otros componentes de Windows que fueron claves para nuestro análisis.

2.1 Conceptos básicos

Para llevar a cabo el análisis se utilizaron dos técnicas: Ingeniería Inversa y Debugging. Ambos jugaron un papel central ya que proporcionaron los recursos que nos permitieron estudiar cómo operaba la Telemetría en un momento dado. A pesar de que cada uno de ellos aportaba características diferentes, la combinación de ambos fue la base para el análisis completo del componente. Los siguientes dos capítulos presentarán y explicarán las técnicas antes mencionadas.

2.1.1 Ingeniería Inversa

La ingeniería de software se puede definir como el proceso de diseño, construcción y prueba de software. Un ingeniero de software es un ser humano con habilidades que le permiten escribir, en un lenguaje de programación particular, instrucciones que finalmente ejecutará

el procesador de la computadora. Al principio, los únicos lenguajes disponibles eran los llamados lenguajes de "bajo nivel" (porque son realmente "cercaños" al procesador mismo). Los ingenieros de software de ese momento tenían que saber instrucciones muy específicas que el procesador podía ejecutar y escribir programas solo con eso. Esto significa que no había funciones de control avanzadas disponibles, solo el conjunto de instrucciones básico proporcionado por el procesador. Con la ayuda de la modularización y la evolución de las computadoras, comenzaron a aparecer nuevas capas de abstracción, lo que facilitó el proceso de desarrollo de programas. Estas capas de abstracción permitieron a los humanos evitar saber lo que estaba sucediendo más allá de su capa actual y, al mismo tiempo, expusieron formas de usar las mismas capacidades que antes de una manera mucho más fácil. Actualmente, la mayoría de los desarrolladores de software (por ejemplo, los desarrolladores web) no necesitan comprender completamente cómo las declaraciones que escriben dentro de un IDE terminan siendo ejecutadas por la CPU. Esto sucede gracias al arduo trabajo de años y años de los ingenieros de software antes mencionados que construyeron estas capas de abstracción. Una de las capas de abstracción importantes construidas son los compiladores. Estos programas convierten instrucciones de lenguaje de programación de alto nivel que son legibles por humanos en 0s y 1s que finalmente son ejecutados por la CPU. Como se mencionó anteriormente, esto permite que la mayoría de los ingenieros de software actuales ignoren lo que sucede dentro de ellos y simplemente hagan uso de sus capacidades.

Actualmente, si alguien quiere entender qué está haciendo un programa, generalmente lee su código fuente (legible por humanos). Sin embargo, esto no siempre es posible. Puede pasar que no tengas el código fuente de ese programa por varias razones: Se perdió o es un proyecto cerrado entre otras. En esos casos, ¿Cómo puede saber qué está haciendo realmente el programa?

Por lo general, el flujo de creación de un nuevo programa implica:

- Desarrollar el código fuente del programa..
- Compilar el código fuente.
- Ejecutar el programa compilado (archivo ejecutable).

Como se mencionó anteriormente, los programas finalmente ejecutan las instrucciones de la CPU. Por lo tanto, aunque es un poco más difícil que leer el código fuente, es posible decompilar el programa compilado para leer las instrucciones de la CPU que va a ejecutar. El lenguaje que se expresa con las instrucciones de la CPU generalmente se conoce como lenguaje ensamblador. A veces, una sola línea de código fuente de un lenguaje de alto nivel puede terminar en docenas de líneas en código ensamblador.

El proceso de leer el código ensamblador del ejecutable y aprender los detalles sobre cómo funciona ese programa también se conoce como ingeniería inversa. El objetivo final sería obtener una representación de código de alto nivel que haga lo mismo que el ejecutable original, aunque a veces esto puede ser difícil de lograr. El nombre proviene del hecho de que es exactamente el proceso inverso del flujo del programa de creación detallado anteriormente. Con la diferencia de que la mayoría de las veces es casi imposible obtener el código fuente exacto del archivo ejecutable. Esta es también la razón por la que realizar ingeniería inversa se considera una tarea muy difícil. Se necesita una comprensión profunda del conjunto de instrucciones de la CPU, las convenciones de llamada, la gestión de la memoria, etc.

Hablando del tema del análisis de código, existen dos formas de hacerlo: Estático y Dinámico. El primero implica leer código (ya sea de alto nivel o ensamblador), mientras que el segundo implica ejecutarlo, detenerlo y analizar el estado de ejecución del programa actual. La combinación de ambos suele ser el mejor escenario posible para un análisis completo de un programa.

2.1.2 Debugging

Debugging se refiere al proceso de analizar un programa durante la ejecución. Suele estar relacionado con el concepto de encontrar y eliminar errores, pero no es el único caso de uso. Debuggear un programa implica tener la posibilidad de detener el programa en algún punto específico, poder leer su estado interno (variables, memoria, registros, etc), saltar a la siguiente instrucción y seguir comparando los cambios entre los estados.

En este proyecto, debido a que se trata del componente de Telemetría que forma parte del Kernel de Windows, se utilizó el debugging del Kernel para analizar el estado del Kernel en diferentes momentos. Tener la posibilidad de detener la ejecución en algún punto específico y poder leer el estado (valor de los registros de la CPU o el contenido de ciertas regiones de la memoria) fue crucial para el éxito de este proyecto. Realizar el debugging del Kernel suele ser incluso más difícil que hacer debugging de un ejecutable normal porque el programa que se utiliza como debugger se ejecuta en el destino que uno quiere debuggear (el Kernel). Por lo tanto, es muy probable que surjan limitaciones debido a que ambos tienen dependencias entre sí. Por eso, aunque Windows ofrece algunas capacidades para debuggear su propio Kernel, generalmente se recomienda usar máquinas virtuales y debugging remoto al analizar Kernels.

2.2 Herramientas

A lo largo de este trabajo se utilizaron diversas herramientas. Aunque todos tenían objetivos diferentes, todos jugaron un papel central en algún punto particular de la investigación. Como se muestra en la sección 2.1, la ingeniería inversa y el debugging fueron las dos técnicas principales utilizadas. En las siguientes secciones se presentarán no solo las herramientas para realizar las técnicas antes mencionadas, sino también otras herramientas que se utilizaron para desarrollar y procesar la información llevada a cabo por los eventos de Telemetría.

2.2.1 IDA pro

IDA, siglas provenientes de Interactive Disassembler, es una herramienta muy poderosa creada por Ilfak Guilfanov que convierte el código ejecutable en su representación Assembler. Es el desensamblador más famoso y completo del mercado. Al poder ejecutarse en Windows, Linux y MacOS, IDA tiene soporte para múltiples procesadores y tecnologías diferentes. Uno de los plugins más importantes que se pueden usar dentro de IDA es "X-RAYS". Este plugin permite construir una representación pseudo-c (de alto nivel) a partir de un binario ejecutable. Esto es muy útil ya que reduce la complejidad de la aplicación de ingeniería inversa debido a que es mucho más fácil de leer código pseudo-c en lugar de Assembler. IDA no es el único desensamblador en el mercado, tiene competidores como: Ghidra, Hooper, Binary Ninja y más. Esta herramienta se utilizó a lo largo de todo

el proyecto y fue, junto con WinDbg, la principal herramienta utilizada. Para obtener información adicional, visite la documentación de IDA¹.

2.2.2 WinDBG

WinDbg es una herramienta para realizar la depuración de programas que se ejecutan sobre el sistema operativo Windows creado por Microsoft. Es uno de los debuggers más utilizados, principalmente por ser nativo, pero también por su excelente interfaz de usuario. Además, WinDbg permite realizar debugging a través de la red (pipes/sockets y más), incluido debugging del kernel. Además, cuenta con soporte para conectarse a sistemas de Microsoft y descargar los símbolos automáticamente siempre que sea posible. Finalmente, WinDbg tiene un lenguaje de scripting muy sencillo (con documentación completa) e incluso acepta scripts escritos en otros lenguajes de propósito general como Javascript. Esta capacidad de scripting se aprovechó durante todo el análisis realizado en este trabajo, para extraer y leer partes específicas de los datos. Por ejemplo, ayudó en el estudio del contenido de estructuras y porciones de memoria en ciertos puntos que contribuyeron a la comprensión del flujo de trabajo de Telemetría. Como consecuencia, se desarrollaron varios scripts que ayudaron a automatizar la mayoría de las pruebas realizadas. Por todas estas razones, la mayor parte del trabajo realizado en este proyecto se realizó con WinDbg. Para obtener información adicional, visite la documentación de depuración de Microsoft².

2.2.3 XPERF y Message Analyzer

Aunque se utilizan para propósitos muy específicos, estas dos herramientas de Windows nos permitieron confirmar que algunas hipótesis desarrolladas dentro de este proyecto eran correctas. Xperf es una herramienta de Windows, parte de las herramientas Windows Performance Analyzer³, que se ocupa de los datos de tracing (controlar las fuentes de tracing y también procesar sus datos). Message Analyzer⁴ permite, entre otras cosas, mostrar datos capturados del tráfico de protocolo, eventos, etc. Para el alcance de este trabajo, ambas herramientas se utilizaron juntas en la sección 5.2.2, con el fin de confirmar algunas hipótesis sobre la correctitud de la información registrada.

2.3 Componentes de Windows

Nuestro objetivo fue analizar el componente de Telemetría del sistema operativo Windows desde una perspectiva interna. Para registrar eventos, Telemetría aprovecha un framework de kernel llamado Event Tracing for Windows (ETW). Aunque este proyecto se centra en el análisis desde una perspectiva interna, comprender su arquitectura y objetivos principales antes de pasar a su código assembler facilitó el análisis. Las siguientes secciones describirán tanto Telemetría como ETW, desde una perspectiva de alto nivel.

¹ <https://hex-rays.com/ida-pro/>

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>

2.3.1 Event Tracing for Windows

ETW⁵ es un framework de tracing muy eficiente que funciona a nivel de kernel, que proporciona a aplicaciones de kernel y de usuario, registro de eventos personalizable. Para hacer uso de ellos, los desarrolladores deben instrumentar el código fuente de las aplicaciones mencionadas mediante el uso de funciones expuestas por la API de ETW.

2.3.1.1 Arquitectura de ETW

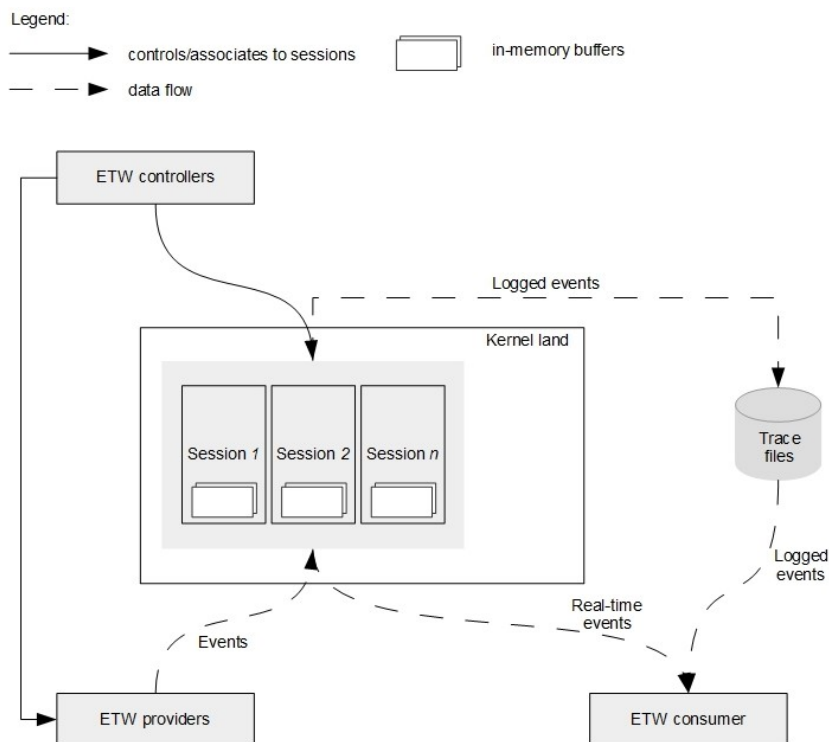


Fig. 2.1: Arquitectura de ETW (Event Tracing for Windows).

ETW está compuesto de varios componentes que son clave para su funcionalidad. Sin embargo, hay cuatro componentes importantes, como se muestra en la figura 2.1, que vale la pena destacar:

- Sesiones
- Controladores
- Proveedores
- Consumidores

2.3.1.2 Sesiones de ETW

Las sesiones son entidades en las que finalmente se almacenan los eventos. De hecho, profundizando en los aspectos técnicos, los eventos en realidad se almacenan dentro de

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

los buffers de la sesión. Toda la arquitectura ETW admite un máximo de 64 sesiones trabajando simultáneamente. Para diferenciar una sesión de otra, se asigna un “número de sesión”, también conocido como logger id, a cada una de ellas. Además, las sesiones ETW tienen un nombre legible por humanos que se puede usar para reconocer rápidamente la sesión correspondiente. Por ejemplo, el nombre asignado a la sesión ETW de Telemetría es “DiagTrack-Listener”.

Hay dos tipos de sesiones ETW:

- **NT Kernel:** Lo utilizan todos los proveedores de ETW integrados a nivel Kernel para registrar sus eventos.
- **Not NT Kernel:** Debe ser utilizado por todos los proveedores de ETW que no pueden iniciar sesión en las sesiones de NT Kernel Logger. Por ejemplo: proveedores de ETW personalizados, proveedores que registran eventos que no se consideran parte del kernel central (como eventos de usuario de TCP/IP), etc.

2.3.1.3 Controladores de ETW

Los controladores son los encargados de “administrar” las sesiones ETW. Entre sus tareas se pueden encontrar:

- Encender/Pausar sesiones.
- Activar/Desactivar proveedores (así permitir que escriban o no eventos a una sesión particular).
- Administrar el tamaño del pool del buffer.
- Obtener estadísticas de ejecución (número de buffers utilizados, número de buffers entregados, etc.)⁶

2.3.1.4 Proveedores de ETW

Los proveedores de ETW son entidades (aplicaciones, controladores, etc.) que contienen instrumentación de tracing de eventos, en otras palabras, hacen uso de la API de ETW. Cada proveedor tiene un GUID particular que lo identifica de manera única. Dentro del flujo de trabajo de ETW, estas entidades son las que desempeñan el papel de escribir eventos en las sesiones de ETW. Para poder llevar a cabo dicha acción, los proveedores deben registrarse en la sesión ETW correspondiente.

2.3.1.5 Consumidores de ETW

Los consumidores de ETW son las entidades que consumen/leen/analizan los datos almacenados en las sesiones de ETW. Un ejemplo de un consumidor ETW podría ser la herramienta de Windows **xperf** (explicado en la sección 2.2.3).

En resumen, **Proveedores** son aquellos que escriben eventos en **Sesiones** que luego son leídos por los **Consumidores**. Los **controladores** son los administradores de las sesiones.

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

2.3.2 Telemetría

Telemetría es un componente de Windows 10 que tiene como objetivo recopilar información del sistema operativo en ejecución y enviarla a los servidores remotos backend de Microsoft de forma segura. Collecta información técnica, como el software instalado y su rendimiento, detalles del hardware y más, para analizarla y aprender cómo mejorar la experiencia general del usuario de Windows. Toda esta lógica se implementa dentro de la librería dinámica de Windows *diagtrack.dll* y se integra como un servicio del sistema operativo el cual se ejecuta con el nombre “Diagtrack”.

Los datos de telemetría recopilados por este servicio, proporcionados por varias partes de Windows, se escriben en las sesiones ETW relacionadas con Diagtrack.

Windows Telemetry se puede configurar en uno de los siguientes cuatro niveles:

- Seguridad: Registra la información necesaria para ayudar a proteger Windows. Por ejemplo: Datos sobre el componente Telemetría y Experiencias de Usuarios Conectados, Windows Defender, etc.
- Básico: Información del dispositivo, datos relacionados con la calidad, compatibilidad de la aplicación, datos de uso de la aplicación y datos del nivel de seguridad.
- Mejorado: Uso de Windows, Windows Server, System center y aplicaciones. El rendimiento, datos de confiabilidad avanzada y datos de nivel básico y de seguridad.
- Completo: Todos los datos necesarios para identificar problemas y ayudar a solucionarlos, además de datos de nivel básico, seguridad y mejorado.

Como se puede deducir, se ordenan de forma ascendente teniendo en cuenta la cantidad y el detalle de la información que se va registrando. Además, cuanto mayor sea el nivel de telemetría, mayor será la cantidad de proveedores de ETW que se registrarán en la sesión de ETW de DiagTrack.

3. CONFIGURACIÓN DE LABORATORIO

El objetivo de este proyecto fue siempre aprender sobre componentes específicos de Windows que generalmente están integrados dentro de su Kernel. Por lo tanto, poder realizar debugging del kernel fue una de las ventajas más importantes. Hay varias formas de configurar el debugging del kernel en sistemas Windows¹.

Después de investigar los diferentes enfoques posibles, se concluyó que la forma más cómoda, fácil de administrar y más confiable era construir un laboratorio de Máquinas Virtuales. Entre las numerosas ventajas podemos encontrar:

- Independencia con la máquina host: no hay que preocuparse por el sistema operativo de la máquina host.
- Independencia de versiones de Windows entre las VMs: La máquina debuggee (a ser debuggeada) y la debugger puede tener versiones de OS distintas.
- Flexibilidad: La posibilidad de poder reiniciar o cambiar el estado de una máquina sin afectar a la otra.

A pesar de que se tuvo que elegir una herramienta particular de Virtual Machine Manager, la siguiente configuración podría adaptarse a cualquier otro manejador de VMs.

3.1 Creando el entorno

Entre todas las formas posibles de conectar la debugger con la debuggee, se decidió optar por el enfoque de "red". En otras palabras, se realizará el proceso de debugging del Kernel a través de una red.

Después de elegir VirtualBox² como la herramienta VM Manager, se procedió a hacer la configuración de la red. Usando una función predeterminada de VirtualBox llamada "Administrador de red de host", se creó una red interna. A pesar de ser virtual, "vboxnet0" usará IPv4 con su propio servidor DHCP:

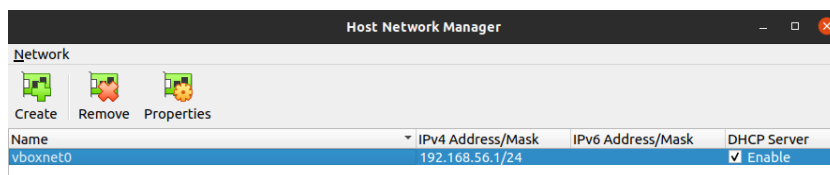


Fig. 3.1: Creación de red interna usando el Host Network Manager de VirtualBox

Una vez construida esta red, solo era cuestión de conectar ambas máquinas (debugger y debuggee) a esta red y realizar la configuración correspondiente a cada una de ellas.

¹ <https://docs.microsoft.com/en-us/windows-hardware/drivers/debugger/setting-up-kernel-mode-debugging-in-windbg--cdb-o-ntsd>

² <https://www.virtualbox.org/>

3.2 Configuración del 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.

La debuggee debía exponer su función de debugging de Kernel a través de la red. Para realizar tal cosa, el programa **bcdedit**⁴ fue utilizado. Esta configuración se realizó en dos pasos: activar el debugging y luego activar una configuración particular para permitir el debugging de Kernel de forma remota.

La activación del debugging se puede lograr usando la siguiente línea de comando (como administrador):

```
1 $> bcdedit /debug on
```

Fig. 3.2: Comando de shell emitido para activar el debugging

Para establecer configuraciones de debugging particulares:

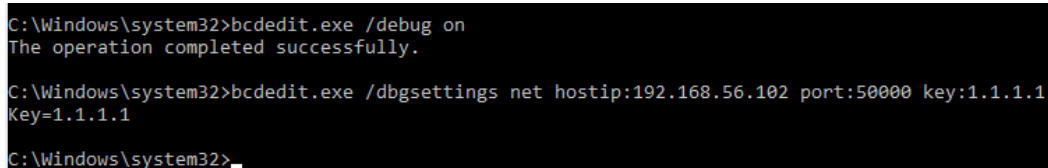
```
1 $> bcdedit /dbgsettings net hostip:a.b.c.d port:n key:x.x.x.x
```

Fig. 3.3: Comando de shell emitido para activar la configuración de debugging remoto

La figura 3.3 muestra el comando⁵ que permite que el host con dirección IP **a.b.c.d** puede debuggear su Kernel a través de **net** (red) usando el puerto **n** y la clave **x.x.x.x**. Si bien la mayoría de los indicadores de comando se explican por sí mismos, la **clave** se usa para cifrar los datos transmitidos por el canal.

En el caso específico de este proyecto, se utilizaron los siguientes valores:

- **hostip:** Dirección IP de la máquina debugger de la red vboxnet0.
- **port:** Puerto 50000.
- **key:** 1.1.1.1. Debido a que se encontraba dentro de un entorno y una red completamente controlados, no había necesidad de preocuparse por cifrar este tráfico.



```
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: Resumen de configuraciones realizadas en la máquina a debuggear.

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

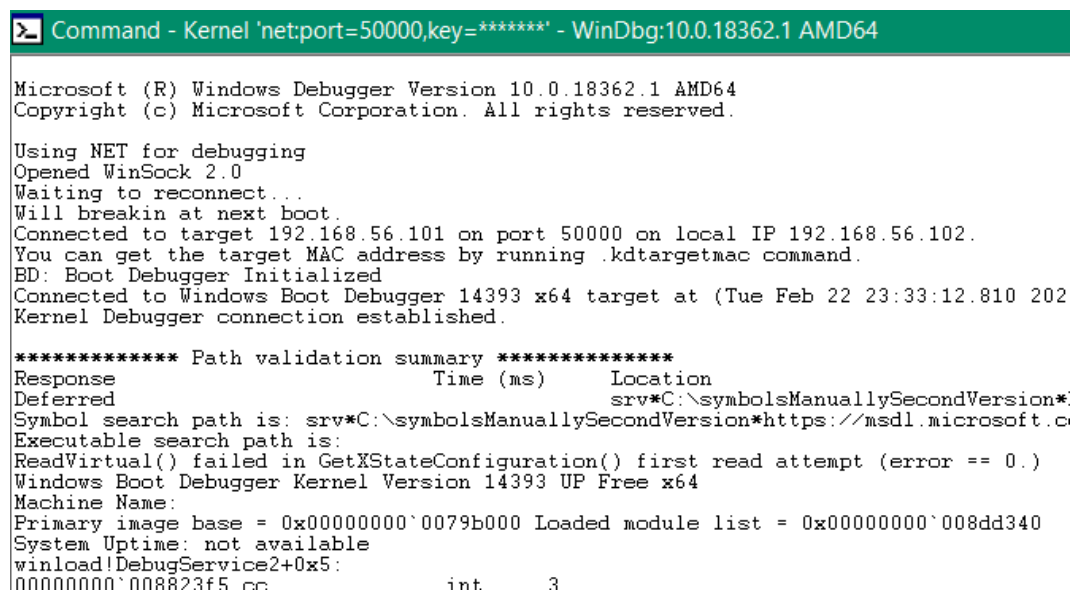
⁴ <https://docs.microsoft.com/en-us/windows-hardware/manufacture/desktop/bcdedit-command-line-options?view=windows-11>

⁵ <https://docs.microsoft.com/en-us/windows-hardware/drivers/devtest/bcdedit-dbgsettings>

3.3 Configuración del debugger

La configuración de la máquina debugger se realiza dentro del WinDBG, por ser la herramienta de debugging a utilizar.

Para realizar debugging del Kernel con WinDBG, es necesario ir a **File** → **Kernel Debug**. Una vez dentro del panel de configuración (pestaña “**net**”) se solicitan dos de los elementos rellenados anteriormente: puerto y clave. Una vez que se completen estos datos (y se reinicie la máquina debuggee) será posible realizar una interrupción (breakpoint) tan pronto como la máquina debuggee comience con su proceso de booteo.



```

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                0              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: Pantalla WinDBG dentro de la máquina debugger tan pronto como la debuggee se conecta.

4. TRABAJO PREVIO

Trabajos previamente realizados relacionados con la telemetría de Windows generalmente se han centrado en dos temas:

- Análisis de eventos: Entender qué eventos se han enviado a los servidores backend de Microsoft haciendo foco en su contenido.
- Análisis de tráfico: análisis de red de los eventos enviados de forma remota a los servidores backend.

Se han invertido muchos esfuerzos académicos y no académicos para comprender qué tipo de información se registraba dentro de los eventos de telemetría (Análisis de eventos). Esto generó muchas preocupaciones debido a su relación con la privacidad de los usuarios de Windows y temas legales como las regulaciones. Un claro ejemplo de esto podría ser el Reglamento General de Protección de Datos (GDPR) que, en términos simples, proporciona un marco legal para la protección de datos en la UE¹. Al mismo tiempo, el análisis de la información que se envía a los servidores back-end de Microsoft ha captado mucha atención. Hay dos razones detrás de esto. Primero, los datos se envían de forma remota y, por lo tanto, el canal que se utiliza debe ser seguro. Es por eso que algunos proyectos se dedicaron a asegurar que la información que se enviaba estaba encriptada. En segundo lugar, los servidores Backend de Microsoft analizan datos "no confiables", dado que dicho input proviene de computadoras ajenas. Como consecuencia, podría surgir un riesgo potencial si esos servidores fueran vulnerables a algún tipo de ataque al intentar analizar los datos enviados. Algunos ejemplos de proyectos dedicados a realizar tareas antes mencionadas son [1][2][3].

Nuestra investigación, a pesar de tratar brevemente con el contenido de los eventos, se centra en diferentes aspectos. Este proyecto analizará los aspectos técnicos de la Telemetría de Windows, centrándose especialmente en cómo se lleva a cabo el proceso de registro de eventos dentro del propio Kernel de Windows. En otras palabras, este trabajo no se centrará en estudiar los datos que se envían ni el canal utilizado para enviarlos. Por el contrario, se centrará en aplicaciones, estructuras del kernel, procesos internos y otras entidades que de alguna manera están involucradas en el proceso de logging después de que haya ocurrido algún evento en particular.

¹ <https://gdpr.eu/>

5. ANÁLISIS DEL COMPONENTE Y METODOLOGÍA

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 Entendiendo cómo Telemetría hace uso de ETW

Cuando se inicia el sistema operativo Windows, se crean varias sesiones dentro de ETW. Entre ellas hay una relacionado con la Telemetría: DiagTrack. Como se explica en 2.3.1, cada sesión tiene “proveedores” -aquellas entidades que brindan información a la sesión-. Para comprender cómo Telemetry hizo uso de ETW, fue importante conocer los proveedores de sesión de DiagTrack. Hasta este punto, se desconocían las respuestas a las siguientes preguntas:

1. ¿Quiénes son?
2. ¿Dónde corren?
3. ¿Qué información loggean?

Windows permite consultar la lista de proveedores registrados en una sesión en particular ejecutando el siguiente comando de powershell:

```
1 $> Get-EtwTraceProvider | where {$_.SessionName -match "<SESSION_NAME>"}
```

Fig. 5.1: Comando de Powershell para enumerar los proveedores de ETW registrados en una sesión en particular.

Para cada proveedor registrado en la sesión consultada, se genera la siguiente información:

- El GUID del proveedor.
- El nombre de la sesión.
- Más información que no es relevante en este momento.

Por ejemplo:

```

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: Salida del comando Get-EtwTraceProvider Powershell para la sesión de DiagTrack

Con esta lista, la pregunta **¿Quiénes son?** parecía estar respondida. Aún así, restaban responder dos preguntas.

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:

En ese momento, surgió una idea interesante: para responder **¿Dónde se están ejecutando?** y **¿Qué están registrando?** podría ser útil interponerse en el momento exacto en que cualquiera de esos proveedores escriba a la sesión de DiagTrack. En otras palabras, podría ser útil establecer un breakpoint en la función que realiza la escritura a la sesión de DiagTrack. Una vez que se alcance dicho punto de interrupción, se podría extraer información interesante como:

1. El fragmento de código que inició la escritura (inspeccionando el stack de llamadas de la función). En otras palabras, identificar quién estaba escribiendo.
2. El contenido real del evento que se está escribiendo.

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.

Fue entonces cuando entró en juego el debugging (2.1.2). Analizando los símbolos expuestos por el binario ntoskrnl, se encontró la función **EtwWrite**. Esta función parecía ser la encargada de llevar a cabo el proceso de escritura dentro de las sesiones. No obstante, no era la mejor opción establecer un punto de interrupción en esta función, ya que todos los proveedores del sistema (no necesariamente relacionados con DiagTrack) podrían estar utilizándolo. Era necesario encontrar una manera de detectar sólo las escrituras realizadas por los proveedores de DiagTrack.

El comando de powershell (5.1) devolvió información para cada proveedor registrado. Parte de esa información era el GUID. Debido a que el objetivo anterior era filtrar escrituras

uras solo de proveedores registrados en la sesión de DiagTrack, podría ser útil establecer un punto de interrupción condicional dentro de la función **EtwWrite** e intentar verificar si el GUID proporcionado coincidía con alguno de los GUIDs devueltos en la ejecución del comando.

Desafortunadamente, esta estrategia tenía un problema menor. La función (**EtwWrite**) tenía cinco parámetros y ninguno de ellos mostraba el GUID directamente:

```

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

```

Fig. 5.3: Documentación para la función EtwWrite¹.

El primer parámetro era el handler de registro. Este objeto se devolvía una vez que un proveedor se registraba a una sesión (a través de la función **EtwRegister**) con éxito. Echando un vistazo más profundo a **EtwRegister** fue posible observar que ésta recibía el GUID como parámetro:

```

C++
NTSTATUS EtwRegister(
    LPCGUID      ProviderId,
    PETWENABLE_CALLBACK EnableCallback,
    PVOID        CallbackContext,
    PREGHANDLE    RegHandle
);

```

Fig. 5.4: Documentación para la función EtwRegister².

Este hallazgo básicamente significaba que tener solo un punto de interrupción en **EtwWrite** no sería suficiente ya que también se necesitaba información de **EtwRegister**. En otras palabras, para entender si la escritura la estaba realizando un proveedor registrado en la sesión de DiagTrack, era necesario:

1. Extraer la lista completa de proveedores registrados en la sesión de DiagTrack.
2. Interceptar todas las ejecuciones de **EtwRegister** y verificar si el GUID que se usa estaba dentro de la lista.
3. Si lo estaba, guardar el handler.
4. Interceptar todas las ejecuciones de **EtwWrite** y comprobar si el handler que se está utilizando es uno de los handlers almacenados.
5. Si lo era, el proveedor que esta escribiendo está registrado en la sesión de DiagTrack.

Aunque esta estrategia parecía ser teóricamente prometedora, era necesario entender cómo llevar a cabo cada uno de estos pasos. Las siguientes secciones describirán este proceso.

5.1.1 Analizando el proceso de registración con ingeniería inversa

Como se mencionó en la sección 2.3.1, siempre que un proveedor quiera registrarse en una sesión en particular, debe llamar a la función **EtwRegister**. Por esto, el primer paso fue analizar el comportamiento de esta función usando **IDA**(2.2.1).

Como se puede ver en la figura 5.5, la única acción interesante realizada por **EtwRegister** fue una llamada a otra función llamada **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 de función *EtwRegister*, obtenido usando el desensamblador de IDA, mostrado para resaltar los offsets y llamadas a funciones clave como *PsGetCurrentServerSiloGlobals* y *EtwpRegisterProvider*.

Un análisis rápido de esta última función mostró que era la función que realizaba la implementación real del proceso de registro. Sin embargo, debido a la falta de documentación, fue necesario comprender más a fondo qué estaba sucediendo realmente en su interior.

Las siguientes secciones presentarán una descripción detallada del proceso de reversing de (solo las partes interesantes para esta investigación) **EtwpRegisterProvider**. Para hacerlo más fácil, se dividirá en 4 partes:

- 1. Entender el diseño de la función.
- 2. Comprobar si ya existe un GUID para este proveedor.
- 3. Si no, crear uno nuevo.
- 4. Devolver el handler.

```

1 // 1. Entender la aridad de la funcion.
2 signed __int64 __fastcall EtwpRegisterProvider(__int64 a1, _QWORD *a2, int a3,
    void (__fastcall *a4)(ULONG_PTR, __int64, __int128 *, __int64), __int64 a5,
    __int64 a6, __int64 *a7){
3
4 [...]
5
6 // 2. Comprobar si ya existe un GUID para este proveedor.
7 ptr_guid_entry = (char *)EtwpFindGuidEntryByGuid(etw_silo, ptr_guid, 0);
8
9 // 3. Si no, crear uno nuevo.
10 if ( ptr_guid_entry || (ptr_guid_entry = EtwpAddGuidEntry(ptr_etw_silo_cpy2,
    ptr_guid_cpy, 0)) != 0i64 )
11 {
12     v15 = __readgsqword(0x188u);
13     --*(_WORD *)(v15 + 484)
14
15     [...]
16
17 // 4. Devolver el handler.
18 v35 = EtwpAddKmRegEntry((ULONG_PTR)ptr_guid_entry, v10, (__int64)v9, a5,
    (__int64)&ptr_handler);
19 v20 = v35;
20
21 [...]
22
23 }
24 return v20;

```

Fig. 5.6: Representación en pseudocódigo de una parte interesante de la función *EtwpRegisterProvider*. Este fragmento se creó utilizando la salida del desensamblador y decompilador de IDA. Ilustra las cuatro partes diferentes que serán analizadas.

5.1.1.1 Entendiendo la estructura de la función

EtwpRegisterProvider recibía siete parámetros:

```

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.

Por lo general, cuando se realiza ingeniería inversa, no es necesario comprender cada pequeño detalle, sino solo los puntos clave que son importantes para cumplir con los objetivos propuestos. Esta vez, no fue la excepción.

El enfoque principal aquí no fue comprender cómo funcionaba completamente el proceso de registro, sino solo tener una idea y conocer la relación entre GUID y el handler de registro.

Luego de analizar **EtwpRegisterProvider** fue posible concluir:

1. **a1**: Era un puntero a una estructura desconocida.
2. **a2**: Era un puntero a la estructura de GUID.
3. **a7**: Era la dirección de memoria donde el puntero al handler de registración sería guardado en el futuro (se puede pensar como la “salida” de la función).

¿Qué es esta estructura **a1** ?

La figura 5.5 muestra que antes de llamar a **EtwpRegisterProvider**, se invoca la función **PsGetCurrentServerSiloGlobals**. Esta última devuelve un puntero a una estructura *S* de tipo **_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: Ilustración de la estructura **_ESERVERSILO_GLOBALS** (*S*) usando el comando *dt* de WinDBG.

Sin embargo, el primer parámetro proporcionado a **EtwpRegisterProvider** no era el puntero a *S* sino el puntero a otra estructura *S.2* de tipo **_ETW_SILODRIVERSTATE** que resulta ser parte de *S*, situado en el offset 0x350.

```
kd> dt nt!_ETW_SILODRIVERSTATE
+0x000 EtwpSecurityProviderGuidEntry : _ETW_GUID_ENTRY
+0x190 EtwpLoggerRundown : [64] Ptr64 _EX_RUNDOWN_REF_CACHE_AWARE
+0x390 WnipLoggerContext : [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: Ilustración de la estructura **_ETW_SILODRIVERSTATE** (*S.2*) usando el comando *dt* de WinDBG.

Con esta información fue posible concluir que **a1** apuntará a una estructura global que contiene configuraciones, ajustes e información en general directamente relacionada con el

framework **ETW**. **a2** y **a7** contendrán punteros a un GUID y a un lugar donde se almacenará un handler de registro más adelante. La información recopilada de este análisis fue suficiente para avanzar, ya que tener esas dos piezas de información nos permitiría filtrar solo las llamadas de funciones interesantes a *EtwWrite*.

Una vez dentro de la función *EtwpRegisterProvider*, después de realizar algunos chequeos, intenta obtener la estructura de GUID relacionada con el GUID proporcionado. Si no existe, creará una.

```
// 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: Representación en pseudocódigo de una parte interesante de la función *EtwpRegisterProvider*. Este fragmento se creó utilizando la salida del desensamblador y decompilador de IDA. Ilustra la segunda y tercera parte que serán analizadas.

5.1.1.2 Chequeando si el GUID del proveedor ya existe

Esta parte se centrará en comprender cómo funciona el proceso de recuperación de la “entrada GUID” ya existente.

La acción de recuperar/buscar dicha estructura, es realizada por una función particular llamada **EtwpFindGuidEntryByGuid**:

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

Como se puede deducir de la línea anterior, se proporcionaron dos parámetros importantes: la estructura **ETWSILODRIVERSTATE**(a1) *S* y el puntero al 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: Primer basic block de *EtwpFindGuidEntryByGuid*, obtenido usando el desensamblador de IDA. Destaca las operaciones que se realizan con el fin de analizarlas detenidamente.

La figura 5.10 muestra cómo la función obtiene la entrada GUID relacionada con el proveedor (si existe): *rcx* mantiene el puntero a *S* y *rdx* mantiene *ptr_guid*. Analicemos esta función con más profundidad.

El primer punto destacado es la función *add* que almacena en *rcx* el puntero a la estructura almacenada en el offset *0x590* de *S*. Volviendo al diseño de la estructura de *S* (figura 5.8) se puede apreciar que, en el offset *0x590*, la estructura *EtwpGuidHashTable* de tipo *_ETW_HASH_BUCKET[64]* está presente. La figura 5.12 muestra su diseño.

Justo antes de la función *add*, *eax* se llena con el contenido de la dirección *rdx + 8*. *rdx* contenía *ptr_guid*, lo que significa que *eax* tendrá el tercer grupo de 4 bytes dentro de la estructura guid. ¿Por qué el tercero? Porque el offset era 8. ¿Por qué 4 bytes? Porque se utilizó el registro *eax* (32 bits).

En las siguientes líneas, el valor de *eax* se modifica constantemente xoraendolo sucesivamente con los diferentes grupos de 4 bytes que componen la estructura guid³. Después de realizar estas sucesivas operaciones de xor, se aplica un and booleano contra *eax* (resul-

³ Algunas veces las estructuras no estaban documentadas en absoluto. A veces lo estaban, pero no era posible encontrar su documentación hasta que se encontraba algún tipo de pista que apuntara a él. Hasta este momento se desconocía el diseño de la estructura apuntada por *ptr_guid*, sin embargo a partir de esta función se pudo concluir que la estructura tenía un tamaño de 16 bytes.

tado de xoring) usando una máscara `0x3f`. Esta máscara establecerá todos los bits de `eax` en 0 con la excepción de los últimos 6 que permanecerán con su valor real. La razón para hacer esto es porque $2^6 = 64$. En otras palabras, esta máscara hace que el resultado de xoring se ajuste al rango de un índice de bucket válido. Luego, multiplica el resultado con `0x38` (tamaño de la estructura `_ETW_HASH_BUCKET`). Finalmente, el valor de `rax(eax)` se agrega a `rcx` que tiene el puntero a la estructura `EtwpGuidHashTable`.

A continuación, la función antes mencionada en un estilo de pseudocódigo (*ptr* es una versión corta de “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)
```

Por lo tanto, *ptr_bucket* es básicamente un puntero a un bucket en particular dentro de

EtwpGuidHashTable calculado en función del GUID del proveedor⁴. Una vez obtenido dicho valor, se realiza un “búsqueda” dentro de la estructura de la siguiente forma:

```
1. iterator = *ptr_bucket;
2. if ( *ptr_bucket != ptr_bucket )
3. {
4.     while ( 1 )
5.     {
6.         u12 = *ptr_guid_cpy - iterator[3];
7.         if ( *ptr_guid_cpy == iterator[3] )
8.             u12 = ptr_guid_cpy[1] - iterator[4];
9.         if ( !u12 && EtwpReferenceGuidEntry((ULONG_PTR)iterator)
10.            break;
11.         iterator = (_QWORD *)iterator;
12.         if ( iterator == ptr_bucket )
13.             goto LABEL_13;
14.     }
15.     u4 = iterator;
}
```

Fig. 5.11: Fragmento de pseudocódigo de la función *EtwpFindGuidEntryByGuid*, obtenido mediante el descompilador de IDA. Resalta la forma en que se busca el GUID una vez que se seleccionó el depósito correcto.

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

Fig. 5.12: Ilustración de la estructura `_ETW_HASH_BUCKET` usando el comando *dt* de WinDBG.

Al principio se construye un iterador. Este iterador apuntará inicialmente al Flink de la primera entrada de la lista⁵ del bucket (figura 5.11). La línea **2** capturará el caso especial donde la lista está vacía. En ese caso en particular, se saltará todo el ciclo y se ejecutará

⁴ También hubo un valor adicional involucrado en el cálculo del depósito. Sin embargo, en este contexto particular, el valor no se tuvo en cuenta ya que siempre fue 0.

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

la **LABEL_13** (rutina para salir, que no se muestra en la figura). Cabe mencionar que esta rutina ejecuta una declaración de retorno (línea **15**) con el valor de la variable *v4* (que inicialmente se define como 0).

Si la lista no está vacía, la primera operación que se lleva a cabo es una resta entre la primera quadword del GUID y un valor de *iterador[3]* (línea **6**). Debido a que la variable *iterador* se define como un puntero de 8 bytes, *iterador[3]* apuntará al offset 0x18 de la estructura almacenada dentro de Flink. En el caso de que ambos valores sean iguales, la segunda comparación (entre el segundo quadword del GUID y el *iterador[4]*) se realiza como se muestra en las líneas **7 y 8**.

En este punto se pueden sacar algunas conclusiones:

- El ciclo itera una lista doblemente enlazada que contiene una estructura particular *T*.
- *T* tiene el GUID del proveedor almacenado en el offset 0x18.
- Nuevamente, parece que el GUID tiene una longitud de 16 bytes.
- Del nombre de la función se puede inferir que *T* es una estructura que representa un GUID.

Avanzando con el análisis del código, en el caso de que fallen algunas de las comparaciones, el iterador cambia sus valores al siguiente en la lista (línea **11**). Antes de continuar, se asegura que el ciclo no haya terminado, comprobando si el valor real del iterador es el mismo que el utilizado como punto de partida (línea **12**). Si son iguales, se ejecuta la rutina de salida, lo que significa que el valor de retorno será 0.

Si ambas comparaciones son iguales (el GUID del proveedor y el almacenado en *T* son iguales), se ejecuta una función llamada *EtwpReferenceGuidEntry* con el valor actual del iterador como parámetro (línea **9**). Después de esta ejecución, el ciclo finaliza con la sentencia break. Sin embargo, antes de ejecutar la rutina de salida, el valor de *v4* se completa con el valor del *iterador* (línea **15**), lo que significa que el valor devuelto será un puntero a la entrada de GUID relacionada con el GUID del proveedor. La función *EtwpReferenceGuidEntry* solo realiza algunos chequeos en el puntero que no son relevantes para nuestra investigación.

Por lo tanto, para resumir, se puede decir que:

La función *EtwpFindGuidEntryByGuid* busca una estructura particular (probablemente llamada GUID), que se almacena dentro de una lista doblemente enlazada de un bucket dentro de *EtwpGuidHashTable* de *_ETW_SILODRIVERSTATE*, basado en hacer algunas operaciones matemáticas con el GUID del proveedor.

Después de terminar con este análisis, se encontró la documentación de la estructura de entrada del GUID:

```
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: Ilustración de la estructura `_ETW_GUID_ENTRY` usando el comando `dt` de WinDBG.

```
kd> dt nt!_GUID
+0x000 Data1             : UInt4B
+0x004 Data2             : UInt2B
+0x006 Data3             : UInt2B
+0x008 Data4             : [8] UChar
```

Fig. 5.14: Ilustración de la estructura `_ETW_GUID` usando el comando `dt` de WinDBG.

Afortunadamente, todas las conjeturas anteriores realizadas fueron correctas:

- La entrada de GUID (ahora `_ETW_GUID_ENTRY`) tenía el GUID en el offset `0x18` (figura 5.13)
- El GUID era una estructura de 16 bytes de tamaño (figura 5.14)

5.1.1.3 Si el GUID no se encontró, crear uno nuevo

La sección anterior detalló cómo era el proceso para encontrar una entrada de GUID ya existente basada en el GUID del proveedor. Esta sección explicará el proceso de creación de una nueva entrada GUID. De la figura 5.9 se puede observar que la función encargada de esta parte es la función `EtwpAddGuidEntry`:

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

Como se puede inferir de la línea anterior, se proporcionaron dos parámetros importantes: el puntero a la estructura `_ETW_SILODRIVERSTATE` (para simplificar se llamará `ptr_etw_silo` en lugar de `ptr_etw_silo_cpy2`) y el puntero al GUID (por simplicidad se llamará `ptr_guid` en lugar de `ptr_guid_cpy`).

Una de las primeras líneas de `EtwpAddGuidEntry` ejecuta un llamado a la función `EtwpAllocGuidEntry`. Como se puede inferir rápidamente del nombre, básicamente asigna una cierta cantidad de memoria dentro del montón para que la use la entrada GUID y le devuelve el puntero. La parte de asignación ocurre en el primer basic block de `EtwpAllocGuidEntry`:

```

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

arg_0= qword ptr 8
arg_8= qword ptr 10h

; FUNCTION CHUNK AT 000000001405B8F5E 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: Interesante basic block de la función *EtwpAllocGuidEntry*, obtenido mediante el desensamblador de IDA, encargado de realizar la asignación de la memoria necesaria.

Como se puede observar en la figura 5.15 la función de Windows *ExAllocatePoolWithTag*⁶ se llama con los siguientes parámetros:

- **PoolType:** 0x200 (**NonPagedPoolNx**). Este valor indica que la memoria del sistema asignada no será paginable ni ejecutable⁷. Esto es realmente bueno desde una perspectiva de seguridad, ya que significa que si algún código malicioso termina estando dentro de esta región de memoria, no será ni ejecutable ni paginable y, por lo tanto, difícil de aprovechar para una explotación exitosa.
- **NumberOfBytes:** 0x190. Este valor es el tamaño de la estructura *_ETW_GUID_ENTRY* (figura 5.13).
- **Tag:** “0x47777445”. De acuerdo con la documentación, solo se usa una longitud de cuatro caracteres como etiqueta de grupo. Debido a que se especifica en orden inverso: 0x45747747 → “EtwG”.

Por lo tanto, como se pensaba, *EtwpAllocGuidEntry* asigna la memoria necesaria para contener la estructura *_ETW_GUID_ENTRY* y le devuelve un puntero al heap.

El código restante de *EtwpAddGuidEntry* está dedicado a completar y ajustar algunas partes de las estructuras relacionadas. Algunos puntos clave al respecto:

- Se busca una entrada GUID relacionada con el GUID dentro de la lista doblemente enlazada de entradas GUID utilizando la misma técnica que la utilizada en *EtwppFindGuidEntryByGuid*. Si se encuentra una estructura, el puntero se libera.
- Solo tres partes de la estructura *_ETW_GUID_ENTRY* se completan en este punto:

⁶ Documentación: <https://docs.microsoft.com/en-us/windows-hardware/drivers/ddi/content/wdm/nf-wdm-exallocatepoolwithtag>

⁷ Documentación: <https://docs.microsoft.com/en-us/windows-hardware/drivers/ddi/content/wdm/nf-wdm-pool-type>

1. El puntero a la anterior entrada GUID en la lista de doblemente enlazada (offset 0x0)
2. El puntero a la siguiente entrada GUID en la lista de doblemente enlazada (offset 0x8)
3. El puntero a SILO STATE (offset 0x178)

Para resumir, una vez que se ejecuta *EtwpAllocGuidEntry*, se devuelve el puntero a la memoria alocada en el heap que contiene la estructura *_ETW_GUID_ENTRY*. El siguiente paso es insertar esta entrada en *EtwpGuidHashTable*. Para realizar esa acción, primero se busca el lugar correcto para insertarlo como se muestra anteriormente.

5.1.1.4 Devolver el handler

Volviendo a lo que dice la figura 5.4, el 4º parámetro de *EtwRegister* es algo de tipo *PREGHANDLE*. Aunque no está muy claro, este parámetro es la salida de la función (usualmente referido como un tipo de parámetro “out”). Además, como se mencionó anteriormente, la lógica de registro real está implementada por *EtwpRegisterProvider* y por lo tanto la salida de *EtwRegister* no es otra que la salida de *EtwpRegisterProvider*.

A pesar de que el GUID del proveedor existió anteriormente o no, en este punto existe un puntero a una estructura *_ETW_GUID_ENTRY* que contiene sus datos y ya está dentro de las estructuras principales de ETW.

Una vez que el código alcanza este punto, el siguiente paso es básicamente obtener el handler.

Justo después de encontrar el puntero a *_ETW_GUID_ENTRY*, se llama a la función *EtwpAddKmRegEntry*:

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

donde :

1. **a1**: Es el puntero a *_ETW_GUID_ENTRY*, llamado *ptr_guid*.
2. **a5**: Es la dirección de memoria brindada por *EtwWrite* (y luego por *EtwpRegisterProvider*) donde el handler debería ser devuelto.

Los parámetros restantes no son interesantes para el fin de nuestra investigación.

Una vez dentro de *EtwpAddKmRegEntry*, las primeras líneas importantes fueron:

```
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: Interesante basic block de la función *EtwpAddKmRegEntry*, obtenido mediante el desensamblador de IDA, encargado de realizar la asignación de la memoria necesaria.

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

Como se puede inferir de 5.15, esta también es una alocaión de memoria en el heap:

- **PoolType:** 0x200 (**NonPagedPoolNx**).
- **NumberOfBytes:** 0x70. Como muestra la figura, *rdi* tiene asignado el valor 0.
- **Tag:** “0x52777445”. En base a la documentación, la etiqueta de grupo. Por especificarse en orden inverso: 0x45747752 → “EtwR”.

El análisis de las siguientes líneas de *EtwpAddKmRegEntry* demostró cómo se estaba llenando el espacio reservado (estructura) antes mencionado. Al debuggear esta función, la siguiente línea sobresalió del resto:

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

La razón para resaltarlo fue que, en ese momento, *rbp* mantuvo el puntero en la entrada GUID. Lo que significa que esta estructura, potencialmente la estructura del handler de registro, tiene el puntero a la entrada GUID en el offset 0x20. Después de terminar de llenar el resto, se devuelve el puntero a la estructura.

Volviendo a 5.6, se puede apreciar que la salida de *EtwpAddKmRegEntry* también es la salida de *EtwpRegisterProvider*. Lo que confirmó que esta era la estructura del handler de registro.

Luego de terminar con este análisis se encontró la documentación de la estructura del handler de registro:

```
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: Ilustración de la estructura *_ETW_REG_ENTRY* usando el comando *dt* de WinDBG.

5.2 Interponiéndose entre la escrituras de los proveedores

En la sección 5.1 se presentó una idea sobre cómo responder a dos preguntas importantes: Interponerse en el momento exacto en que los proveedores escriben en la sesión de DiagTrack. Sin embargo, la idea hubiera sido inviable sin el análisis realizado en la sección 5.1.1.

En este punto, fue posible detectar si un proveedor en particular estaba escribiendo conociendo solo su GUID. Sin embargo, era importante comprender a qué sesión estaba escribiendo este proveedor.

Para un análisis inicial se realizó una combinación de análisis automático y manual. Se estableció un punto de interrupción en la función **EtwWrite** mediante el siguiente script Windbg(2.2.2):

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

Este script simplemente imprimía la dirección del handler de registro del proveedor que realiza la escritura (@rcx contiene el primer parámetro de la función de acuerdo con la convención de llamadas de Windows x64). Una vez que se obtuvo la dirección del handler, también fue posible obtener el GUID. La comparación del GUID con la salida del comando powershell **Get-EtwTraceProvider** (sin filtros) arrojó un resultado interesante: la mayoría de las veces, un proveedor con E02A841C-75A3-4FA7-AFC8-AE09CF9B7F23 como GUID era el que escribía. Desafortunadamente, este proveedor no estaba relacionado con DiagTrack en absoluto.

Con el objetivo de filtrar las escrituras realizadas por este proveedor, se utilizó el siguiente script:

```
1 bp nt!EtwWrite ".if (@rcx != fffffda839f0c2c50){.printf \"Handler:
  %N\\n\",@rcx;gc;}.else{gc;}" "
```

Sin embargo, esto no fue suficiente. Los proveedores que no estaban relacionados con DiagTrack continuaron inundando la salida del script. Claramente, este no era un buen enfoque.

Con el objetivo de buscar resultados interesantes, se utilizó otra función de escritura relacionada con ETW llamada **EtwWriteTransfer**:

```
1 bp nt!EtwWriteTransfer ".if (@rcx != fffffda839f0c2c50){.printf \"Handler:
  %N\\n\",@rcx;gc;}.else{gc;}" "
```

Esta vez, apareció un nuevo proveedor registrado a DiagTrack con GUID E9EAF418-0C07-464C-AD14-A7F353349A00. Para obtener información adicional, el script se actualizó una vez más para obtener la pila de llamadas del proceso del proveedor:

```
1 bp nt!EtwWriteTransfer ".if (@rcx == FFFFFDA83A036F0D0){.printf \"Handler:
  %N\\n\",@rcx;kc;gc;}.else{gc;}" "
```

```
2
3 Call Site
4 00 nt!EtwWriteTransfer
5 01 nt!TlgWrite
6 02 nt!CmpInitHiveFromFile
7 03 nt!CmpCmdHiveOpen
8 04 nt!CmLoadAppKey
```

```
9 05 nt!CmLoadDifferencingKey
10 06 nt!NtLoadKeyEx
11 07 nt!KiSystemServiceCopyEnd
12 08 ntdll!NtLoadKeyEx
13 09 0x0
```

Hasta ahora, la única forma de detectar cuándo estaban escribiendo los proveedores relacionados con DiagTrack consistía en dos pasos:

1. Interponerse en llamadas a la función *EtwWrite*.
2. Verificar si el proveedor estaba registrado a DiagTrack comparando su GUID y el resultado del comando powershell que se muestra en la Figura 5.1.

Sin embargo, incluso si era un proveedor relacionado con DiagTrack el que escribía, no era suficiente para garantizar que está escribiendo en la sesión de DiagTrack (los proveedores pueden estar registrados a varias sesiones). Además, *EtwWriteTransfer* mostró que *EtwWrite* no era la única función involucrada en el proceso de escritura. En otras palabras, se plantearon las siguientes dos preguntas:

1. ¿Es **EtwWrite** la única función de escritura utilizada? (claramente no!)
2. ¿Cómo podemos estar seguros de que un proveedor realmente está escribiendo en la sesión de DiagTrack?

5.2.1 Funciones de escritura en ETW

Un análisis rápido de los símbolos y las referencias cruzadas del binario **ntoskrnl.exe** mostró que en realidad había un “grupo de funciones” que podía usarse para realizar escrituras dentro de ETW:

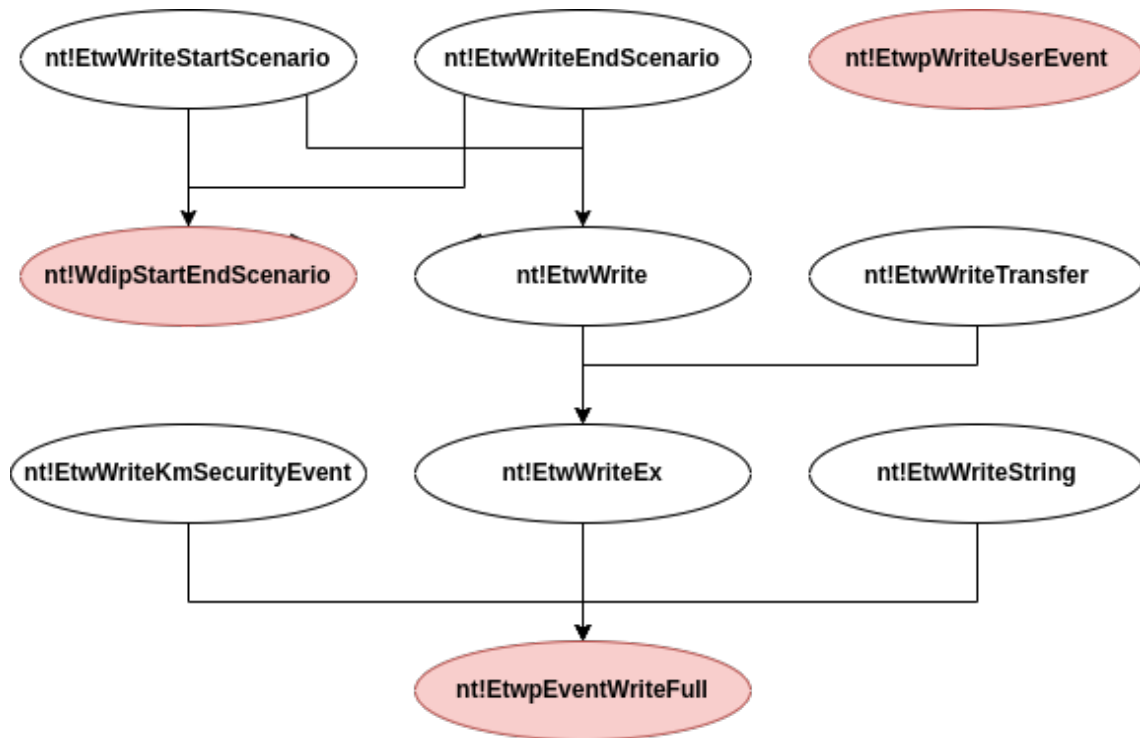


Fig. 5.18: Ilustración de todas las funciones de ETW relacionadas con la escritura de eventos y su flujo de llamadas.

Como se puede apreciar en la figura 5.18 todas las funciones terminan llamando a **nt!EtwEventWriteFull**, **nt!EtwWriteUserEvent** o **nt!WdipStartEndScenario**. Era necesario comprender si todas estas funciones se llamaban cada vez que un proveedor deseaba registrar un evento. Por lo tanto, se decidió forjar un script WinDBG que estableciera puntos de interrupción en las tres funciones y registrara cada vez que se usaba una de ellas. Este script se dejó ejecutando en una máquina mientras se usaba para las tareas diarias normales. Después de algunos días de debugging, se analizó la salida. Fue posible concluir que la función **nt!WdipStartEndScenario** nunca fue llamada bajo el contexto de interés de esta investigación. Debido a este resultado, se decidió continuar la investigación enfocándose solo en las dos primeras funciones: **nt!EtwEventWriteFull** y **nt!EtwWriteUserEvent**.

5.2.2 Asegurando que los proveedores escriben a la sesión del Diagtrack

La idea anterior de analizar solo las escrituras de los proveedores que devolvía el comando powershell tenía, al menos, tres problemas:

1. La salida del comando powershell devolvía proveedores que estaban registrados en una sesión particular en un momento dado t . Si un proveedor se registra, escribe y se da de baja de la sesión en un momento tf donde $t \notin tf$, esa escritura no se tendrá en cuenta.
2. Incluso si el proveedor está registrado en la sesión de DiagTrack, esto no garantiza que el proveedor esté escribiendo en ella, ya que podría suceder que, por alguna razón, estuviera desactivado.

3. Dependía demasiado del análisis manual.

Después de varias sesiones de brainstorming, surgió una idea que marcó la diferencia: ¿Y si es posible relacionar el handler que se utiliza en el momento de escribir, con la sesión donde el proveedor va a escribir? Si eso es posible, los dos primeros problemas estarían resueltos. El tercero podría resolverse ya que solo un punto de interrupción en la función de escritura será suficiente para realizar el análisis completo.

Las siguientes secciones detallarán el proceso para encontrar una respuesta a estos problemas.

5.2.2.1 Inspeccionando estructuras de ETW

Hasta ahora sucedió que varias estructuras ETW fueron la clave para superar diferentes obstáculos. Queríamos saber si esto era de nuevo el caso.

La primera estructura analizada fue `_ETW_REG_ENTRY` (figura 5.17) ya que hubiera sido la forma más directa y sencilla de relacionar sesión y proveedor. Desafortunadamente, ninguno de sus componentes fue útil.

La siguiente estructura analizada fue `_WMI_LOGGER_CONTEXT`. Esta estructura parecía ser la representación real de una sesión de ETW. Debido a su tamaño, solo se representan los atributos necesarios y representativos:

```

1  +0x000 LoggerId      : Uint4B
2  +0x004 BufferSize    : Uint4B
3  +0x008 MaximumEventSize : Uint4B
4  +0x00c LoggerMode    : Uint4B
5  +0x010 AcceptNewEvents : Int4B
6  [...]
7  +0x070 ProviderBinaryList : _LIST_ENTRY
8  +0x080 BatchedBufferList : Ptr64 _WMI_BUFFER_HEADER
9  +0x080 CurrentBuffer  : _EX_FAST_REF
10 +0x088 LoggerName     : _UNICODE_STRING
11 +0x098 LogFileName    : _UNICODE_STRING
12 +0x0a8 LogFilePattern : _UNICODE_STRING
13 +0x0b8 NewLogFileName : _UNICODE_STRING
14 [...]
15 +0x428 LastBufferSwitchTime : _LARGE_INTEGER
16 +0x430 BufferWriteDuration : _LARGE_INTEGER
17 +0x438 BufferCompressDuration : _LARGE_INTEGER

```

Fig. 5.19: Ilustración de la estructura `_WMI_LOGGER_CONTEXT` mediante el uso del comando `dt` de WinDBG.

Cada sesión de ETW tendrá una instancia de esta estructura. En el desplazamiento `0x70` hay un atributo llamado **ProviderBinaryList**. Después de un análisis rápido, este atributo parecía contener a todos los proveedores registrados en la sesión en el formato de una listadoblemente enlazada. Para confirmar esa teoría, se analizó el proceso de adjuntar nuevos proveedores a una sesión existente.

La función `nt!EtwpAddProviderToSession` parecía ser la que creaba estos enlaces. Durante el análisis se enfrentaron varios problemas: muchas funciones nuevas desconocidas, dificultad para aplicarles ingeniería inversa, falta de documentación, entre otros. Además

de todos los problemas mencionados, la razón clave por la que se detuvo esta vía de análisis fue: El hallazgo de la estructura `_TRACE_ENABLE_INFO` (más detalles en el próximo capítulo).

5.2.2.2 GUID del proveedor y GUID de grupo

Por lo general, una estructura puede tener la respuesta, pero si no puede comprender cómo se utiliza esa estructura, no podrá encontrar nunca dicha respuesta. Con eso en mente, se llevó a cabo un análisis adicional sobre las funciones de escritura de ETW.

`EtwWriteEx` fue una de las funciones más utilizadas por los proveedores para escribir eventos internos dentro de las sesiones. Esa fue la razón por la que el análisis se centró en ella.

El proveedor y el evento real que quería ser almacenado eran algunos de los parámetros que recibía `EtwWriteEx`. Una parte interesante de su código fuente se puede representar fácilmente como pseudocódigo, con la siguiente lista:

```

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: Fragmento de pseudocódigo de una parte interesante de la función `EtwWriteEx`, obtenido analizando las salidas del desensamblador y decompilador de IDA.

Al principio, ambas partes (líneas 1-7 y 8-14) pueden parecer iguales, pero tienen dos diferencias clave:

1. El desplazamiento usado para `v14`: La primera parte usa `0x64`, mientras que la segunda usa `0x65`.
2. El desplazamiento usado para `v15`: La primera parte usa `0x20`, mientras que la segunda usa `0x28`.

Volviendo a la figura 5.17, es posible entender que la primera parte usa los atributos `GuidEntry` y `EnableMask` mientras que la segunda usa `GroupGuidEntry` y `GroupEnableMask`. Esta fue la clave para comprender que los proveedores no siempre usan la misma entrada de GUID, sino que también pueden usar un “GUID de grupo”.

Además, se encontró que la función `EtwpLevelKeywordEnabled` recibía tres parámetros:

- *ProviderEnableInfo* de tipo `_TRACE_ENABLE_INFO`
- Event Level
- Event keyword

El principal objetivo de *EtwLevelKeywordEnabled* era verificar si el proveedor debía registrar el evento en la sesión de acuerdo con las reglas de filtrado⁸ definido cuando el proveedor se registró por primera vez.

En otras palabras, el listing 5.20 podría traducirse a “Si *EnableMask* es diferente de 0x0, verifique si el evento debe registrarse usando *GuidEntry*. Si es así, llame a *EtwEventWriteFull*. Si *GroupEnableMask* es diferente de 0x0, verifique si el evento debe registrarse usando *GroupEntry*. Si es así, llame a *EtwEventWriteFull*”.

El primer parámetro recibido en *EtwLevelKeywordEnabled* era de tipo `_TRACE_ENABLE_INFO`. Analizando su estructura:

```
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: Ilustración de la estructura `_TRACE_ENABLE_INFO` obtenida usando el comando `dt` de WinDBG.

Esta estructura parecía ser prometedora ya que tenía información que podía relacionarse con una sesión. Además, esta estructura se puede encontrar dentro de `_ETW_GUID_ENTRY` en el desplazamiento `0x50` y `0x70` (mientras que el primero se refiere al GUID del proveedor, el último se refiere al GUID del grupo de proveedores). Sin embargo, la lógica que llenó o usó esta estructura no está presente dentro de *EtwWriteEx*. Como se muestra en 5.20 después de algunas comprobaciones, se estaba llamando a la función *EtwEventWriteFull*. Para continuar con este análisis, fue necesario analizar esta última función.

5.2.2.3 Identificando la sesión destino

El análisis de *EtwEventWriteFull* fue el más difícil pero el más interesante. Aunque recibía 17 parámetros, tenía más de 1000 líneas de código y ayudaría a comprender varias piezas del rompecabezas, aquí solo se analizarán e ilustrarán unas pocas líneas.

Resultó, como se creía, que la lógica para entender en qué sesión iba a escribir la ejecución estaba dentro de esta función. El objetivo de este apartado es explicar en profundidad cómo se lleva a cabo este proceso de elección de sesión.

Para comenzar con el análisis, comencemos analizando este pseudocódigo simplificado que representa la parte clave del proceso:

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

```

1  [...]
2  while ( 1 ){
3      bsf_found = !_BitScanForward((unsigned int *)&enable_info_bucket_index,
                                   enable_mask);
4      guid_ptr_shifted_by_enable_info_bucket_index = (__int64)&ptr_guid_entry[4
                                   * (unsigned int)enable_info_bucket_index]
5
6      if ( _bittest64(&ptr_local_addr, *(unsigned __int8
                                   *)(&guid_ptr_shifted_by_enable_info_bucket_index + 0x76)) )
7          continue;
8
9      [...]

```

Fig. 5.22: Fragmento de pseudocódigo de una parte interesante de la función *EtwpEventWriteFull*, obtenido analizando las salidas del desensamblador y decompilador de IDA. Esta parte ilustra cómo la función selecciona qué sesión ETW escribir.

La línea 1 representa definiciones y declaraciones anteriores que no son importantes para el análisis actual.

La línea 2 representa el ciclo que se ejecutará. En cada iteración, elegirá una de las sesiones en las que está registrado el proveedor y verificará si debe escribir este evento en esa sesión. Para realizar tal tarea, se encontró que el atributo *enableMask* es una máscara de 8 bits que representa el uso de cada uno de los 8 buckets del atributo *enableInfo* (estructura *GuidEntry*). Si el primer bit (posición 0) de *enableMask* era 1, significa que se debe considerar la información dentro de *enableInfo*[0]. Si su valor fuera 0, la información de *enableInfo*[0] debería descartarse. Esto era exactamente lo que sucedía en la línea 3. Se realizaba una búsqueda⁹ sobre la variable **enable_mask** que contenía el contenido real del atributo *enableMask* de *GuidRegEntry*, y se escribía dentro de *enable_info_bucket_index* el índice del primer bit establecido en 1.

La línea 4 desplazó un puntero temporal a *GuidEntry* por $0x20 * \text{enable_info_bucket_index}$. En este caso, $0x20$ era el tamaño de la estructura *_TRACE_ENABLE_INFO* (figure 5.21), que era el tipo de cada bucket *enableInfo*. En otras palabras, estaba moviendo el puntero dependiendo de qué bucket de *enableInfo* debería considerarse en esta iteración del ciclo. La línea 6th fue la que dio sentido a todos los pasos antes mencionados. Se agregaba $0x76$ al puntero desplazado y verificaba¹⁰ si el bit en la posición especificada era 1 o 0. Básicamente, se trataba de comprobar si el *loggerId* del *enableInfo* correspondiente tenía algún valor. Hagamos un ejemplo para entender mejor esto:

Si el resultado de *enable_info_bucket_index* era igual a 0, significaba que el bit menos significativo de *enableMask* tenía 1. Como se explicó antes, esto significa que se debía considerar *enableInfo*[0]. Por lo tanto, el siguiente paso sería desplazar el puntero a *GuidEntry* por $0x20 * 0$, en otras palabras, no desplazarlo. Finalmente, se verifica si la posición $0x76$ tiene algo diferente de 0. $0x76 = 0x70 + 0x6$, $0x70$ era el desplazamiento del primer bucket del atributo *enableInfo* de *GuidEntry*, mientras que $0x6$ fue el desplazamiento de *loggerId* dentro de la estructura *_TRACE_ENABLE_INFO*. Ahora, si *enable_info_bucket_index* fuera 1 en lugar de 0, la única diferencia con el ejemplo anterior habría sido que el puntero a *guidEntry* se desplazaría $0x20 * 1 = 0x20$. Esto significaba que la posición para verificar

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>

con *_bittest64* habría sido $0x76 + 0x20$. Debido a que $0x20$ era del tamaño de la estructura *_TRACE_ENABLE_INFO*, habría accedido al siguiente *loggerId* del bucket de *enableInfo* (*enableInfo[1]*)

Si se cumplía la condición de **6th**, significaba que *loggerId* era 0 y, por lo tanto, no era interesante. Por lo tanto, habría saltado directamente a la siguiente iteración del ciclo. Si la condición no se cumpliera, habría continuado con el resto de las sentencias.

La línea **9** representa todas las operaciones que se ejecutaron una vez que se encontró una sesión válida del proveedor (es decir, verificar si este evento debe escribirse en esa sesión, la escritura real en la sesión, etc.).

El *loggerId* finalmente se convirtió en la forma de relacionar una ejecución de escritura con la sesión de destino. Recordando la estructura que representaba una sesión ETW (*_WMI_LOGGER_CONTEXT*, figura 5.19) el primero de sus atributos era el *loggerId*, una identificación para la sesión. Si bien, hasta este punto se desconocía el momento exacto en que se escribió el evento, esta lógica fue suficiente para entender cómo se eligió la sesión.

5.2.3 Detectando qué se escribe

Si bien el último análisis fue clave para comprender cómo se eligía la sesión de destino, fue necesario encontrar el momento exacto en que *EtwpAllocGuidEntry* estaba escribiendo el evento en esa sesión. Después de todo, ese era el objetivo principal.

Como se muestra en el listing 5.22, la línea **9** implicaba mucho más procesamiento una vez que se encontraba una sesión candidata. Entre todo ese procesamiento, fue posible encontrar el siguiente código relacionado con la forma en que se estaban escribiendo los eventos:

```

1  [...]
2  logger_id = *(unsigned __int16
3      *) (guid_ptr_shifted_by_enable_info_bucket_index+0x76);
4  [...]
5  if ( (unsigned int)logger_id >= 0x40 ){
6      ptr_wmi_trace_of_session_cpy = 1i64;
7      ptr_wmi_trace_of_session = 1i64;
8  }
9  else {
10     ptr_wmi_trace_of_session = *((_QWORD *) (ptr_silo_state + 8 * logger_id +
11         0x390));
12     ptr_wmi_trace_of_session_cpy = ptr_wmi_trace_of_session;
13 }
14 [...]
15 ptr_to_buffer = EtwpReserveTraceBuffer(
16     (unsigned int *)ptr_wmi_trace_of_session_cpy,
17     event_size,
18     (__int64)&ptr_to_buffer_offset,
19     &return_value_of_function_to_get_cpu_clock,
20     0
21 );
22 if ( ptr_to_buffer ){
23     *(_OWORD *) (ptr_to_buffer + 0x18) = *(_OWORD *) (ptr_guid_entry_cpy + 3);
24     *(_OWORD *) (ptr_to_buffer + 0x28) = *(_OWORD *) ptr_event_descriptor_cpy;
25     [...]
26 }
27 [...]

```

Fig. 5.23: Fragmento de pseudocódigo de una parte interesante de la función *EtwpEventWriteFull*, obtenido analizando las salidas del desensamblador y decompilador de IDA. Esta parte ilustra cómo la función realmente escribe un evento en la sesión ETW seleccionada.

La línea 1 representaba definiciones y declaraciones anteriores (como la lista completa 5.22) que no eran importantes para el análisis actual.

La línea 2 asignaría el valor de *loggerId* de la sesión seleccionada a una variable. En aras de la claridad, esta variable se llamó *logger_id*.

La línea 3 representaba definiciones y declaraciones que no eran importantes para este análisis.

La línea 4 preguntaba si *logger_id* $\neq 0x40$. En caso de que lo fuera, asignaría valores extraños a un supuesto puntero. La razón para hacerlo era que el número máximo de sesiones ETW que podían existir era 64 (0x40). En otras palabras, se aseguraba de que *logger_id* tuviera un valor válido.

En caso de que *logger_id* fuera válido, se ejecutarían las líneas 9 y 10. En la primera, se asignaría el valor de *ptr_silo_state* (variable definida antes pero cuyo contenido se podría inferir de su nombre) $+ 8 * \text{logger_id} + 0x390$. ¿Qué significaban todos estos valores? En el desplazamiento 0x390 de la estructura *_ETW_SILODRIVERSTATE* (figura 5.8) había un atributo llamado *WmipLoggerContext*. Este atributo era una array de punteros (*Ptr64*) a estructuras de tipo *_WMI_LOGGER_CONTEXT* (figura 5.19). Esta matriz tenía 64 buckets (nuevamente, debido a la cantidad máxima de sesiones posibles) y su índice en esta matriz coincidía con su valor de *loggerId*. Definitivamente, esto no fue casualidad. Al

agregar una nueva sesión de ETW, se usa el bucket libre de este *WmipLoggerContext* y, por lo tanto, su índice dentro de este array se usa para cumplir con el valor de *loggerId*. Finalmente, 8 era la longitud de una estructura *Ptr64*.

Como conclusión, el valor de *ptr_wmi_trace_of_session* como sugiere el nombre, será el puntero al *_WMI_LOGGER_CONTEXT* correspondiente de la sesión asociada con el *logger_id* encontrado.

La línea **12** representaba definiciones y declaraciones que no eran importantes para este análisis.

La línea **13** representó una llamada a la función más interesante de este análisis:

EtwpReserveTraceBuffer. Esta función recibía parámetros como el puntero al *_WMI_LOGGER_CONTEXT* de la sesión asociada y el tamaño del evento a escribir. Esta función devuelve un puntero al lugar (dentro de los búferes de la sesión ETW) donde los datos del evento (y los metadatos) deben escribirse después (análisis completo en la siguiente sección).

La línea **20** solo verificaría si el puntero era diferente de nulo (en caso de que el sistema operativo se quedara sin memoria).

Las líneas **21**, **22** y **23** representan el momento exacto en que todos los datos se escriben dentro del bloque de memoria devuelto por *EtwpReserveTraceBuffer*.

25 representaba declaraciones que se habrían ejecutado en caso de que no se cumpliera la línea de condición **20**.

Como resumen, es posible deducir que la línea **21** representó el primer momento en que los datos (y metadatos) del evento se estaban escribiendo en los búferes de la sesión.

5.2.4 Asegurando la correctitud de los eventos logeados

A pesar de que el análisis anterior parecía ser correcto, era necesario confirmar que todos los datos registrados en los búferes de ETW se escribieron como se esperaba. Era posible forzar al sistema a registrar un evento en la sesión de DiagTrack creando scripts específicos (más sobre esto en la sección 5.5.1). Por lo tanto, se siguió la siguiente estrategia para confirmar que todos los hallazgos fueran correctos:

1. Obtener la identificación del registrador de la sesión de DiagTrack.
2. Establecer un punto de interrupción algunas instrucciones antes de la escritura real, teniendo en cuenta el logger id.
3. Una vez alcanzado el breakpoint, imprimir el descriptor del evento y la cantidad de eventos en el búfer de DiagTrack.
4. Establecer un nuevo punto de interrupción pocas instrucciones después de la escritura.
5. Imprimir el descriptor del evento nuevamente y compararlo con el valor anterior.
6. Imprimir el contenido del evento para asegurarse de que todo funciona como se esperaba

Se estableció un punto de interrupción algunas instrucciones antes de llamar a la función *EtwpReserveTraceBuffer*, lo que significaba que el evento aún no se escribiría. Una vez que el debugger llegó a ese punto, se imprimió el descriptor del evento para tener algo con lo que comparar después:

```

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

```

Fig. 5.24: La combinación de comandos de WinDBG utilizada para realizar los pasos 1, 2 y 3.

En ese momento, `0x22` era el logger id de la sesión de DiagTrack. Después de imprimir el descriptor del evento, se ejecutó la segunda parte:

```

1  bp nt!EtwpEventWriteFull+0x3c1
2  g
3  dt nt!_EVENT_DESCRIPTOR @r13
4  !wmitrace.eventlogdump 0x22

```

Fig. 5.25: La combinación de comandos de WinDBG utilizada para realizar los pasos 4, 5 y 6.

Se alcanzó el nuevo punto de interrupción (pocas instrucciones después de escribir). El descriptor de eventos era exactamente igual que antes y el contenido del registro tenía la información esperada. Después de estas pruebas, fue posible concluir que el análisis realizado anteriormente era correcto.

Aunque creíamos que en este punto nos habíamos asegurado de que los eventos se registraran como se esperaba, queríamos ir un paso más allá. La idea era comparar los eventos extraídos con nuestra estrategia con los que Telemetry realmente envía a los servidores back-end de Microsoft. Para realizar dicha prueba se utilizaron dos herramientas diferentes:

- `xperf`¹¹
- Message Analyzer¹²

Con `xperf` es posible crear una sesión ETW. Por lo tanto, es posible crear una nueva sesión con exactamente los mismos proveedores que para Telemetría. El siguiente comando se utilizó para cumplir con dicho requisito:

```

1  xperf -start UserTrace -on
   Microsoft-Windows-Diagtrack+43ac453b-97cd-4b51-4376-db7c9bb963ac+da995380
2  -18dc-5612-a0db-161c5db2c2c1+6489b27f-7c43-5886-1d00-0a61bb2a375b -f
   C:\tmp\diaglog%d.etl -FileMode Newfile -MaxFile 50

```

Fig. 5.26: Comando de shell emitido para ejecutar el programa `xperf` para registrar todos los eventos de DiagTrack en un archivo. El objetivo de este comando es poder comparar más tarde el contenido de este archivo con la información extraída a través de nuestros scripts WinDBG.

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>

Además, este comando creará un archivo **.etl** que contiene todos los registros almacenados por los proveedores de ETW adjuntos a esa sesión. Para poder leer esos registros, se utilizó **Message Analyzer**. Solo una simple comparación entre los registros dentro del archivo **.etl** y los registros capturados con nuestra estrategia, aunque con algunas diferencias menores, hicieron que se pudiera concluir que estábamos haciendo las cosas bien. La siguiente figura muestra un ejemplo de cómo se presentaría el mismo evento en nuestra estrategia frente a cómo lo hizo Message Analyzer:

```

1  Con nuestra estrategia
2  =====
3  {"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}
4
5
6  Message Analyzer
7  =====
8  {"AppId": "W:0000f519feec486de87ed73cb92d3cac802400000000!000000667a0f0c0d5e9da69
9  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: Comparación entre el contenido del evento extraído a través de nuestra estrategia de establecer puntos de interrupción en lugares particulares dentro de *EtwpReserveTrace-Buffer* contra el contenido registrado por Message Analyzer

5.3 Analizando servicios

En este punto de la investigación ya habíamos entendido cómo saber qué información se iba a escribir y cómo. Sin embargo, nos faltaba una parte muy importante: el programa que en realidad estaba escribiendo el evento. A primera vista, esto puede parecer algo muy trivial de extraer. La estructura **process** contiene la información sobre el contexto de ejecución y, por lo tanto, se conoce el nombre del proceso que se está ejecutando. Esto podría extraerse fácilmente haciendo uso de la extensión WinDBG **process**:

```
kd> !process -l 0
PROCESS ffffff84963ad780
  SessionId: 0 Cid: 0374 Peb: 87e022f000 ParentCid: 022c
  DirBase: 212cd000 ObjectTable: fffffb50e23d8cf40 HandleCount: <Data Not Accessible>
  Image: svchost.exe
```

Fig. 5.28: Comando de WinDBG que muestra una ilustración de la información generada usando la extensión del proceso. En este caso en particular, está mostrando información relacionada con el proceso que está escribiendo actualmente.

Los servicios dentro de Windows generalmente se implementan dentro de los librerías vinculadas dinámicamente. Sin embargo, las DLL (por sus siglas en inglés) no pueden ejecutarse dentro de un proceso como archivos binarios ejecutables (.exe). Como parte de las características de Windows 10, se introdujo **svchosts.exe**. En pocas palabras, **svchosts.exe** permite que varios servicios de Windows se ejecuten de forma anidada, lo que significa que todos se ejecutarán dentro del mismo proceso. Desde la perspectiva de nuestra estrategia, esto representaba un problema ya que si queríamos tener una forma precisa de determinar qué proceso/servicio era el que realmente estaba escribiendo un evento, teníamos que encontrar una manera de lidiar con servicios anidados.

Nuestra primera idea fue forzar a que todos los servicios del sistema operativo, se ejecutaran de forma aislada (no anidada), a través del uso del siguiente comando:

```
1 $> sc config <service_name> type=own
```

Fig. 5.29: Comando de shell emitido para obligar a un servicio a ejecutarse de forma aislada.

Sin embargo, esta opción no funcionó ya que solo se aplicaba a los servicios que se estaban ejecutando en ese momento.

La segunda idea fue configurar de cierta forma la clave del registry llamada **HKLM:/SYSTEM/CurrentServices**, la cual define todas las configuraciones y capacidades de los servicios. Modificando la clave **Type** se podría especificar el tipo de servicio:

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

Desafortunadamente, esta opción no funcionó, ya que obligar a que cada servicio se ejecute de forma aislada requiere mucha RAM, lo que hace que esta opción sea inviable.

La siguiente (y definitiva) forma de identificar los servicios surgió después de semanas de investigación:

Alex Ionescu escribió un blog¹³ describiendo una herramienta desarrollada por él mismo llamada **ScTagQuery**. Dicha herramienta era capaz de devolver el servicio relacionado con un TID (thread id). En otras palabras, esta herramienta tiene la lógica para vincular un hilo con un servicio. Según su blog, lleva a cabo esta acción utilizando un valor dentro del TEB (Thread Environment Block) llamado **SubProcessTag**. Según su blog, este valor es un índice dentro de una base de datos de servicios. Sin embargo, nunca explica cómo funciona realmente este índice. Por suerte menciona que toda esta funcionalidad está implementada en el SCM (Service Control Manager).

```

1      kd> dt nt!_TEB
2          +0x000 NtTib                : _NT_TIB
3          +0x038 EnvironmentPointer    : Ptr64 Void
4          +0x040 ClientId              : _CLIENT_ID
5          +0x050 ActiveRpcHandle       : Ptr64 Void
6          ....
7          +0x1710 ActivityId           : _GUID
8          +0x1720 SubProcessTag        : Ptr64 Void
9          +0x1728 PerflibData         : Ptr64 Void
10         ...

```

Fig. 5.30: Ilustración de la estructura “_TEB” (Thread Environment Block) resaltando *SubProcessTag* en el offset 0x1720.

Reverseando la herramienta de Alex, se encontró una función interesante llamada **I_ScQueryServiceTagInfo**. Esta función fue la que realmente obtuvo el nombre del servicio al proporcionar una etiqueta de servicio. Para hacerlo, internamente llama a otra función llamada **OpenSCManagerW** que está documentada por Microsoft¹⁴:

```

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

```

Fig. 5.31: La documentación de OpenSCManagerW recopilada de la documentación oficial de Microsoft.

Los parámetros proporcionados a esta función fueron fijos (0, 0 y 4 respectivamente):

- **lpMachineName:** Si el puntero es NULL o apunta a una string vacío, la función se conecta al administrador de control de servicios en la computadora local.
- **lpDatabaseName:** Si es NULL, la base de datos SERVICES_ACTIVE_DATABASE se abre por defecto.
- **dwDesiredAccess:** Especifica los permisos de acceso al administrador de control de servicios. En nuestro caso, fue 0x4, lo que significa que estaba usando SC_MANAGER_ENUMERATE_SERVICE, el cual es requerido para llamar a la

¹³ <http://www.alex-ionescu.com/?p=52>

¹⁴ <https://docs.microsoft.com/en-us/windows/win32/api/winsvc/nf-winsvc-openscmangerw>

función `EnumServicesStatus` o `EnumServicesStatusEx` con el fin de enumerar los servicios que están en la base de datos.

Una vez que se devuelve el controlador a la base de datos SCM, se llama a otra función **`I_ScQueryServiceTagInfo`**. Sin embargo, esta última función estaba inicializando una comunicación RPC y, por lo tanto, dificultaba aún más el análisis. En lugar de continuar por este camino, decidimos echar un vistazo al binario que estaba implementando el SCM: **`services.exe`**.

Dentro de **`services.exe`** se encontraron varias funciones prometedoras como **`ScGenerateServiceDB`** o **`ScGetServiceNameTagMapping`**. Aunque esta última parecía ser la respuesta, después de algunas pruebas nos dimos cuenta de que dicha función se llamaba solo para ciertos servicios en particular. Al profundizar en el análisis, se encontró una tercera función: **`ScGenerateServiceTag`**.

Esta función se ejecuta cada vez que se registra un nuevo servicio en el sistema operativo. En palabras simples, **`ScGenerateServiceTag`** recibe `_SERVICE_RECORD` como el único parámetro y busca el último valor de las etiquetas asignadas. El nombre del servicio viene en el desplazamiento `0x10` dentro de la estructura `_SERVICE_RECORD`, y debido a que el primer parámetro viene en `rcx`, `@rcx+10` tendrá el valor real nombre del servicio actual.



```

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

var_18= dword ptr -18h

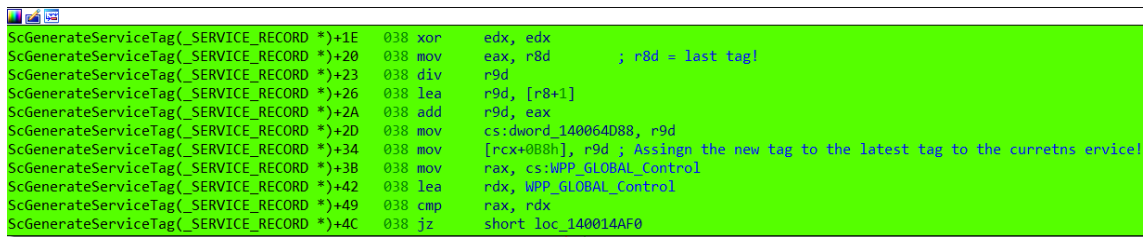
; FUNCTION CHUNK AT 0000000014003927A SIZE 00000024 BYTES

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: Primer basic block de la función `ScGenerateServiceTag`, obtenido usando el desensamblador de IDA, mostrado para ilustrar los offsets mencionados.

La última etiqueta utilizada se almacena en `cs:dword_140064D88` una cuestión de incrementar ese valor. Es por eso que el valor de la última etiqueta se almacena dentro de `r8d` y luego se incrementa. Finalmente, este nuevo valor se guarda dentro del registro de servicio provisto en el desplazamiento `0x0b8` (desplazamiento de la etiqueta de servicio).



```

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 ; Assignn the new tag to the latest tag to the curretns service!
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: Basic block de la función *ScGenerateServiceTag*, obtenido usando el desensamblador de IDA, para resaltar el proceso de lectura de la última etiqueta de servicio y asignación y almacenamiento de la nueva.

Aunque parecía prometedor, esta idea surgió de reversear el binario. Para confirmarlo, hicimos la siguiente prueba:

1. Desarrollar un .exe simple, que creara un nuevo archivo (código fuente en el Apéndice)
2. Poner un breakpoint en **ScGenerateServiceTag**.
3. Correr el siguiente comando:

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

Fig. 5.34: Comando de shell ejecutado para crear un servicio arbitrario a partir de un binario.

Después de ejecutar el último paso, el breakpoint se alcanzó con éxito!

```

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

```

Fig. 5.35: Ilustración de WinDBG del momento en que se alcanzó el punto de interrupción discutido anteriormente en la función *ScGenerateServiceTag* justo después de ejecutar el servicio falso.

Como resumen, cada vez que se agrega un nuevo servicio, se ejecutará **ScGenerateServiceTag** generando una etiqueta para ese servicio. Aunque no se encontró una forma de dumppear toda la base de datos, al establecer un punto de interrupción en esta función podríamos mantener una "copia" de la base de datos. Tener esta base de datos significa que cada vez que escribe un proceso **svchost.exe**, se podrá guardar su etiqueta de servicio al inspeccionar la TEB y compararlo con nuestra base de datos para saber cuál es el servicio y su información.

5.4 Automatizando

Aunque gran parte de todo el análisis anterior se llevó a cabo mediante análisis estático, el análisis dinámico también jugó un papel central. Además, uno de los objetivos de la investigación era tener un marco automático capaz de monitorear cada registro de la sesión de DiagTrack en tiempo real.

Es importante resaltar que la salida de estos scripts funcionan como fuente de información para su posterior procesamiento. La mayor parte de la información impresa por los scripts de WinDBG no se puede aprovechar de inmediato, sino que requiere un procesamiento completo una vez que finaliza la ejecución.

5.4.1 Automatizando la búsqueda del logger id

Una de las primeras cosas que fue necesario automatizar, fue la búsqueda del logger id de la sesión de DiagTrack. Este valor fue clave para todo el proceso debido a que era el nexo que se usaba para asegurar que la escritura estuviera relacionada con la sesión de DiagTrack.

El primer paso fue tratar de encontrar en qué momento se definía este valor para cada sesión de ETW. Al inspeccionar los símbolos de `ntoskrnl.exe`, hallamos la función *EtwpLookupLoggerIdByName* y, por su nombre, fue la primera candidata. Para confirmar que era una buena candidata, comenzamos con las tareas de ingeniería inversa y debugging de la misma.

```

1
2 __int64 __fastcall EtwpLookupLoggerIdByName(__int64 ptr_silo_globals, const
    UNICODE_STRING *logger_name, unsigned int *addr_of_logger_id)
3 {
4     unsigned int *logger_id_to_ret; // r14
5     const UNICODE_STRING *logger_name_cpy; // r15
6     __int64 ptr_silo_globals_cpy; // rbp
7     unsigned int v6; // esi
8     unsigned int logger_id_iterator; // ebx
9     unsigned int *logger_context; // rax
10    unsigned int *logger_context_cpy; // rdi
11
12    logger_id_to_ret = addr_of_logger_id;
13    logger_name_cpy = logger_name;
14    v6 = 0xC0000296
15    ptr_silo_globals_cpy = ptr_silo_globals;
16    logger_id_iterator = 0;
17    while ( 1 ) // Iterating all loggers
18    {
19        logger_context = EtwpAcquireLoggerContextByLoggerId(ptr_silo_globals_cpy,
            logger_id_iterator, 0);
20        if ( logger_context )
21            break;
22    LABEL_3:
23        if ( ++logger_id_iterator >= 0x40 ) // break conditions (there can not be
            more that 64 sessions)
24            return v6;
25    }
26    [...]
27
28    EtwpReleaseLoggerContext(logger_context_cpy, 0);
29    v6 = 0;
30    *logger_id_to_ret = logger_id_iterator;
31    return v6;

```

Fig. 5.36: Fragmento de pseudocódigo de *EtwpLookupLoggerIdByName* obtenido usando una combinación del desensamblador y decompilador de IDA.

De su código fue posible entender que:

1. El primer parámetro (*rcx*) es el puntero a la estructura **_ETW_SILODRIVERSTATE** (figura 5.8), el segundo (*rdx*) un puntero a un string unicode (probablemente el nombre del logger) y el tercero parece ser solo un buffer donde se colocará el el logger id.
2. Hay un iterador que representa el logger id, que irá desde 0 hasta 0x40 (64). Este iterador no podrá ir más allá de 0x40 porque es la cantidad máxima de sesiones que se pueden utilizar en simultáneo dentro de ETW.
3. Como resumen, esta función parece ser la que realmente mapea un nombre logger name con un logger id y registra esta relación dentro de la estructura **_ETW_SILODRIVERSTATE**.

Para ayudar a validar esta información, se procedió a realizar un análisis dinámico. Lo más probable es que la llamada a esta función se realizara durante (o inmediatamente después) del arranque del sistema operativo. Para confirmar esta idea, se estableció un breakpoint en la función *EtwpLookupLoggerIdByName* y se encendió la VM debugge:

```
1 bp nt!EtwpLookupLoggerIdByName
```

Después de un par de segundos, se alcanzó el punto de interrupción. Para confirmar las declaraciones antes mencionadas, se emitieron dos comandos:

```
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: Ejecución de dos comandos Windbg *kc* (para mostrar la pila de llamadas) y *ds* (para mostrar una cadena) después de alcanzar el punto de interrupción de *EtwpLookupLoggerIdByName*.

El primero fue **kc**. Esto imprime el stack de llamadas de la función. Como se puede apreciar, las funciones *nt!EtwpStartTrace* y *nt!EtwpStartLogger* fueron las llamadas. Como sugieren los nombres, estas funciones creaban y registraban por primera vez los logger id dentro del marco ETW, lo que confirma lo dicho anteriormente.

El segundo fue **ds @rdx**, que básicamente imprime un string Unicode. En este caso particular, se imprimió la cadena “UserNotPresentTraceSession”. Este nombre no es otro que el nombre del logger de una de las varias sesiones ETW que se inician de forma predeterminada en Windows.

Después de este análisis, fue posible concluir que al establecer breakpoint dentro de *EtwpLookupLoggerIdByName* y comparar el nombre del logger con el que se usa para la sesión de DiagTrack (**Diagtrack-Listener**) fue posible detectar cuándo se estaba creando el logger del Telemetry. Sin embargo, el logger id no se completará cuando se alcance el breakpoint, sino cuando finalice esta función.

El siguiente paso fue encontrar un lugar (función) que se ejecutará una vez que finalice la función *EtwpLookupLoggerIdByName* (para que el logger id de la sesión de Diagtrack ya esté configurado). Al analizar el stack de llamadas que se muestra en la figura 5.37, se puede apreciar que *nt!EtwpStartTrace* sería un buen candidato. Una vez que finaliza la

instrucción que realiza la llamada a *nt!EtwpStartLogger*, significa que el logger id ya fue creado y vinculado con la sesión correspondiente.

```

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: Código assembler de la función *nt!EtwpStartTrace* obtenido usando el desensamblador de IDA.

Al inspeccionar el código assembler de *nt!EtwpStartTrace*, fue posible confirmar que *nt!EtwpStartTrace+45* sería un muy buen lugar para establecer el segundo punto de interrupción, ya que es la instrucción inmediata después de la ejecución de las funciones antes mencionadas.

En resumen, para obtener el logger id de la sesión de Diagtrack, se deben realizar los siguientes pasos:

1. Configurar un breakpoint en *EtwpLookupLoggerIdByName*.
2. Continuar con la ejecución hasta que el logger name (que está guardado en *rdx*) sea **“Diagtrack-Listener”**.
3. Una vez que matchea, crear un one-time breakpoint en *nt!EtwpStartTrace+45* y continuar con la ejecución.
4. Cuando este último breakpoint sea alcanzado, extraer la información sobre el logger id de la estructura *_ESERVERSILO_GLOBALS*.

La versión final del script para realizar esta tarea se dividió en dos partes. La primera parte ilustra los pasos 1, 2 y 3. La segunda parte implementa el cuarto paso.

```
1 $$ RCX --> Pointer to ETW_SILODRIVERSTATE
2 $$ RDX --> Pointer to the logger name (unicode) (+0x8)
3
4 $$ Save the pointer to the logger name
5 r $t0 = poi(@rdx+0x8);
6
7 $$ Compare the string using alias
8 as /mu ${/v:LOGG_NAME} $t0;
9 .block{r $t10 = 0x0;r $t10 = $spat(@"${LOGG_NAME}", "Diagtrack-Listener")}
10 ad LOGG_NAME
11
12 $$ If it was the Diagtrack-Listener
13 .if($t10 == 1){
14     $$ Save the pointer to ETW_SILODRIVERSTATE
15     r $t1 = rcx;
16
17     $$ The array is in offset 0x1b0 of ETW_SILODRIVERSTATE
18     r $t2 = $t1+0x1b0;
19
20     $$ Lets put a breakpoint just after the logger was created
21     bp /1 nt!EtwpStartTrace+0x45 "$$><\\"path_to_second_part_script"
22 };
23 gc
```

Fig. 5.39: Primera mitad del script de WinDBG a cargo de obtener automáticamente (y almacenar dentro de un registro particular) el logger id. Específicamente, establece un punto de interrupción dinámico después de encontrar la estructura correcta relacionada con el servicio Diagtrack.

```

1 .for(r $t19 = 0;@$t19 < 0x40;r $t19 = @$t19+1){
2   $$ First, check if it is empty
3   .if ( poi(poi(@$t2)+@$t19*0x8) == 1){
4     .continue
5   }
6   .else{
7     $$ Save the pointer to the WMI_LOGGER_CONTEXT in t4
8     r $t4 = poi(@$t2) + (@$t19*0x8);
9
10    $$ Save the STRING UNICODE object in t5
11    r $t5 = poi(@$t4)+0x98;
12
13    $$ Save the UNICODE buffer
14    r $t6 = poi(@$t5+0x8);
15
16    $$ Create the alias of Unicode String
17    as /mu ${/v:LOGG_NAME_NEW} $t6;
18    .block{r $t10 = 0x0;r $t10 = $spat(@"${LOGG_NAME_NEW}",
19      "Diagtrack-Listener")}}
20    ad LOGG_NAME_NEW
21
22    .if($t10 == 1){
23      .printf "Logger id of Diagtrack-Listener found!!: %N\n", @$t19
24      $$ <PLACE FOR FUTURE CODE >
25      .break
26    };
27  };
28 gc

```

Fig. 5.40: Segunda mitad del script de WinDBG a cargo de obtener automáticamente (y almacenar dentro de un registro particular) el logger id. Específicamente, itera todas las sesiones de ETW disponibles hasta que finalmente encuentra la de DiagTrack. Después de encontrar el correcto, almacena el logger id de DiagTrack dentro del registro *t19* y deja el lugar para el código futuro.

5.4.2 Automatizando la extracción de información relacionada a los servicios

Después de darnos cuenta de que el aislamiento de los servicios era fundamental para detectar con precisión la entidad real que estaba realizando la operación de escritura, fue necesario crear una forma de automatizar este proceso.

Como se ilustra en la sección 5.3, la función principal a considerar fue *ScGenerateServiceTag*. Esta función se llama cada vez que se agrega un nuevo servicio a la base de datos de servicios. Además, *ScGenerateServiceTag* tiene toda la información necesaria: el nombre del binario y el índice del servicio.

Solo era cuestión de establecer un punto de interrupción en el lugar correcto, para imprimir ambos valores cada vez que se agregaba un nuevo servicio. Como se puede ver en la figura 5.33, el desplazamiento *0x3b* parece ser un buen candidato ya que es la siguiente instrucción después de almacenar el índice en la estructura *_SERVICE_RECORD*. Para obtener el nombre del servicio, necesitamos imprimir el valor almacenado en el despla-

miento *0x10* de la estructura *_SERVICE_RECORD* proporcionada por el parámetro (que está apuntado por *rcx*). Además, en el desplazamiento *0x3b* de *ScGenerateServiceTag*, el valor del nuevo índice se almacena dentro de *r9d*. Por lo tanto, una secuencia de comandos que podría usarse una vez que se alcance el punto de interrupción establecido en *ScGenerateServiceTag+0x3b* podría ser:

```

1    $$ Script that prints out the new service name and the corresponding tag
2    $$ Print out the placeholder
3    .printf "NEW_SERVICE_ADDED_START\n";
4    $$ Print out the name
5    .printf "%mu\n", poi(@rcx+0x10)
6    $$ Print out the tag
7    .printf "%N\n", r9d
8    $$ Print out the placeholder
9    .printf "NEW_SERVICE_ADDED_END\n";
10   $$ Continue
11   gc

```

Fig. 5.41: Script principal de WinDBG encargado de imprimir información de interés cada vez que se registra un nuevo servicio.

5.4.3 Automatizando la extracción de eventos en funciones de escritura

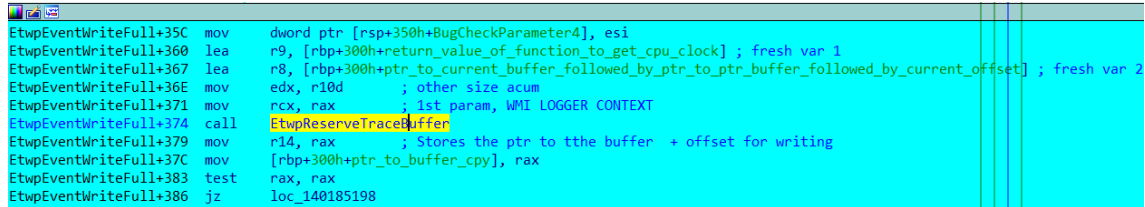
Uno de los análisis más importantes, si no el más importante, realizado fue el relacionado con las funciones de la escritura. Los proveedores de ETW utilizaron diferentes funciones para realizar sus acciones de escritura; y gracias al análisis realizado en la sección 5.2.1 fue posible concluir que había dos funciones importantes: *nt!EtwEventWriteFull* y *nt!EtwWriteUserEvent*. Por motivos de simplicidad y debido a su parecido, nos centraremos únicamente en los detalles del script creado para detectar escrituras de *nt!EtwEventWriteFull*.

El objetivo de este script era imprimir toda la información valiosa cada vez que se detectaba un evento DiagTrack:

- Timestmap
- GUID del proveedor
- Stack de llamadas
- Información sobre el proceso que estaba escribiendo (incluido el nombre del proceso/servicio)
- El PEB (Process Environment Block)
- La estructura del descriptor del evento
- El contenido del evento
- El tag del servicio, en el caso de que la escritura la haya realizado el proceso svchost.exe

Para asegurarnos de que el evento se escribiría en la sesión de DiagTrack, necesitábamos validar que el logger id de la escritura actual coincidiera con la de DiagTrack. Afortunadamente, ya sabíamos cómo realizar tal acción (revisar la figura 5.24).

Era crucial detectar todas las escrituras provenientes de *nt!EtwpEventWriteFull*. Sin embargo, elegir el lugar correcto para establecer el punto de interrupción fue un desafío. Después de un análisis en profundidad de diferentes candidatos, se concluyó que la mejor compensación sería justo antes de llamar a *EtwpReserveTraceBuffer*. La razón principal para elegir este desplazamiento fue que era la última instrucción de *nt!EtwpEventWriteFull* donde los búferes aún estaban vacíos y, al mismo tiempo, la información como el logger id aún era fácil de obtener.



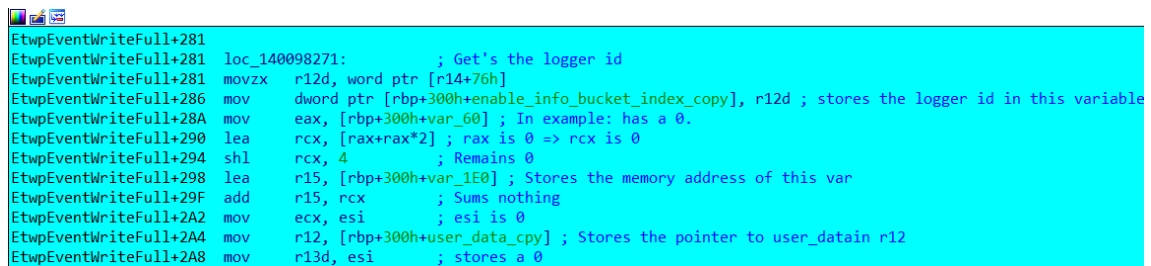
```

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     ecx, 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 de *EtwpEventWriteFull*, extraído con el desensamblador de IDA, que muestra el offset de la última instrucción ejecutada justo antes de llamar a *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 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_data in r12
EtwpEventWriteFull+2A8 mov     r13d, esi ; stores a 0

```

Fig. 5.43: Basic block de *EtwpEventWriteFull*, extraído con el desensamblador de IDA, que muestra cuándo y cómo se almacena el logger id en *r12d*.

Teniendo en cuenta que el script actual se ejecutaría después del presentado en la sección anterior (5.4.1) y por lo tanto *\$t19* contendría el id del registrador de DiagTrack, la tarea de determinar si el destino de la escritura actual era la sesión de DiagTrack podría resumirse en la chequear la siguiente condición :

```

1  $$ Compare the current LOGGER ID with the DiagTrack one.
2  .if (r12d == @$t19){
3      \DO something
4  }

```

Fig. 5.44: Primera versión del script de WinDBG encargado de extraer automáticamente información de interés desde el momento en que un proveedor está escribiendo un evento llamando a *EtwpEventWriteFull*.

Ahora que sabemos cómo estar seguro de si una llamada de escritura es interesante o no en nuestro contexto, era hora de extraer la información de ella. En este punto, hay al menos dos cosas fuera de la lista de información valiosa que se pueden extraer:

1. El GUID_ENTRY
2. El descriptor del evento

El GUID_ENTRY se almacenó en *rbp+300h-280h* y EVENT_DESCRIPTOR en *rbp+300h-2C0h*. Por lo tanto, podríamos agregar esta lógica a la versión actual del script:

```
1    $$ Compare the current LOGGER ID with the DiagTrack one.
2    .if (r12d == @$t19){
3        $$ In this function, the guid entry is situated in [rbp+300h-280h]
4        r $t1 = poi(rbp+300h-280h);
5
6        $$ In this function, the event_descriptor is situated in [rbp+300h-2C0h]
7        r $t2 = poi(rbp+300h-2C0h);
8
9        // Do more things
10   }
```

Fig. 5.45: Segunda versión del script de WinDBG encargado de extraer automáticamente información de interés desde el momento en que un proveedor está escribiendo un evento llamando a *EtwpEventWriteFull*.

Finalmente, dado que una parte de la información que se necesitaba extraer de esta función era el contenido real del evento que se estaba registrando, era necesario continuar con la ejecución de *EtwpEventWriteFull*, al menos, hasta que *EtwpReserveTraceBuffer* haya acabado. Para asegurarse de que todo ya se había ejecutado, se decidió establecer dinámicamente un nuevo punto de interrupción en el último basic block de *EtwpEventWriteFull*. Establecer este punto de interrupción en ese punto garantizará que la función finalice y que la mayor parte de la información esté lista para ser leída. Por eso se decidió establecer el punto de interrupción en *EtwpEventWriteFull+0x600*:

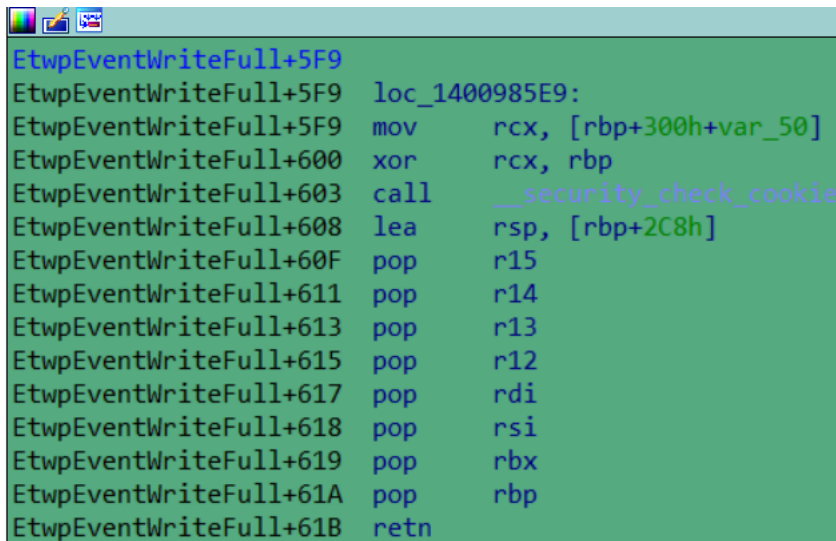


Fig. 5.46: Último basic block de la función *EtwpEventWriteFull*, obtenido usando el desensamblador de IDA, que se muestra para resaltar el mejor offset donde establecer el punto de interrupción para extraer información.

En aras de la simplicidad, se decidió dividir el guión en dos. El script “main” que contendría toda la lógica ya explicada y el “internal” que contendría la lógica de extracción del resto de información.

```

1    $$ Compare the current LOGGER ID with the DiagTrack one.
2    .if (r12d == @$t19){
3        $$ In this function, the guid entry is situated in [rbp+300h-280h]
4        r $t1 = poi(rbp+300h-280h);
5
6        $$ In this function, the event_descriptor is situated in [rbp+300h-2C0h]
7        r $t2 = poi(rbp+300h-2C0h);
8
9        $$ So, once we know that the provider is going to ask for buffers related
10       $$ with the telemetry, we jump to a part of the same function where the
11       $$ event should be already written to the buffers, and therefore we can
12       $$ print out the valuable information.
13       bp /1 nt!EtwpEventWriteFull+0x600 "$$><\"${arg1}\\"
14       gc;
15   }
16   .else{
17       gc;
18   }

```

Fig. 5.47: Versión final del script de WinDBG encargado de extraer automáticamente información de interés desde el momento en que un proveedor está escribiendo un evento mediante la llamada a *EtwpEventWriteFull*.

El script “main” solo extrajo dos de los siete elementos que queríamos extraer de la función de escritura. Por lo tanto, aún necesitaba extraer:

- Timestamp

- GUID del proveedor
- Stack de llamadas
- Información sobre el proceso que estaba escribiendo (incluido el nombre del proceso/servicio)
- El PEB (Process Environment Block)
- La estructura del descriptor del evento
- El contenido del evento
- El tag del servicio, en el caso de que la escritura la haya realizado el proceso svchost.exe

Algunos fueron fáciles:

- Timestamp: Únicamente fue necesario ejecutar: **.echo time**
- Stack de llamadas: Únicamente fue necesario ejecutar: **kc**
- El PEB (Process Environment Block): Únicamente fue necesario ejecutar: **!peb**
- Información sobre el proceso: Únicamente fue necesario ejecutar: **!process -1 0**

Pero... ¿y el resto?

5.4.3.1 Extracción del GUID de los proveedores

La extracción del GUID del proveedor no fue un desafío. Ya teníamos el puntero al `GUID_ENTRY` almacenado en el registro `t1` (ver figura 5.45). Al recordar el diseño completo de la estructura (figura 5.13), se puede ver fácilmente que el GUID del proveedor estaba en `t1+0x18` y `t1+0x18+0x8` (debido a que se dos words de largo).

En pocas palabras, la siguiente línea podría usarse para extraer el GUID completo del proveedor:

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

Fig. 5.48: Comando WingDBG que imprime el GUID del proveedor a partir de una estructura `_GUID_ENTRY` almacenada en el registro `t1`.

5.4.3.2 Extracción del contenido de los eventos

Haciendo uso de la extensión de Windbg **wmitrace** y debido a que el contenido ya estaba en los buffers dado que `EtwpReserveTraceBuffer` ya había sido ejecutada, fue posible leer la información registrada al leer los búferes ETW. Por motivos de simplicidad y corrección, se utilizaron algunos alias útiles:

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

Fig. 5.49: Comando WingDBG que imprime el contenido del evento usando el plugin `wmitrace`.

5.4.3.3 Extracción del tag del servicio

Uno de los mayores desafíos fue la presencia del proceso **svchosts.exe**. Como pueden contener varios servicios en su interior, era necesario implementar una forma de detectar cuál de todos los servicios posibles estaba detrás de la función de escritura. Con la técnica de aislamiento de servicios explicada en la sección 5.3, fue posible hacer coincidir un índice con el nombre de servicio correspondiente. Sin embargo, necesitábamos sacar la etiqueta de servicio de la información del proceso. Recordando lo explicado en la sección 5.3, esta información se almacenaba dentro del TEB (Thread Environment Block).

En simples pasos:

1. Extraer el nombre del proceso de la información del proceso.
2. Determinar (por comparación) si el nombre del proceso era “svchost.exe”.
3. Si era, leer el TEB del proceso y extraer la etiqueta.
4. Imprimir la etiqueta de servicio.

```

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: Fragmento WingDBG encargado de imprimir el nombre del proceso. En caso de que el proceso sea “svchost.exe”, imprimirá la etiqueta de servicio del PEB.

- Línea 1 resuelve el primer paso.
- Línea 2 resuelve el segundo paso. Las líneas
- 3 y 4 resuelven el tercer paso (consulte la figura 5.30).
- Las líneas 5, 6 y 7 resuelven el cuarto paso.

Era hora de completar el rompecabezas. Juntando todos los fragmentos desarrollados para extraer cada parte, el script final sería:

```

1      r $t11 = poi(rbp+300h-280h);
2      r $t12 = poi(rbp+300h-2C0h);
3
4      $$ Just to ensure that we are still in the same write
5      .if ((@$t11 == @$t1) & (@$t12 == @$t2)){
6          .reload
7          .printf "WRITE_INFO_START\n";
8
9          $$ If the process is in svchost, we want to have the service related to
10             the actual thread.
11          $$ In order to do that, we should print the tag of the service related to
12             the tread and
13          $$ afterwards compare it again to the dump of the services db.
14
15          .foreach /pS 13 (proc_name {!process -1 0 }) { aS ${/v:ProcessName}
16              "${proc_name}" };
17          .block{r $t10 = 0x0;r $t10 = $spat("${ProcessName}", "*svchost.exe*")}
18          .if ($t10 == 1){
19              .foreach /pS 5 (teb_addr {!thread @thread }) { r $t12 = teb_addr
20                  ;.break }
21              .printf "SERVICE_TAG_START\n"
22              .foreach /pS 1 (tag { dd @$t12+0x1720 L1 }) { .echo tag}
23              .printf "SERVICE_TAG_END\n"
24          }
25
26          .printf "DATE_TIMESTAMP_START\n";
27          .echotime;
28          .printf "DATE_TIMESTAMP_END\n";
29          .printf "PROVIDER_GUID_START\n"
30          .printf "%N %N\n", poi(@$t1+0x18), poi(@$t1+0x18+0x8);
31          .printf "PROVIDER_GUID_END\n"
32          .printf "CALL_STACK_START\n";
33          kc;
34          .printf "CALL_STACK_END\n";
35          .printf "PROCESS_INFO_START\n";
36          !process -1 0;
37          .printf "PROCESS_INFO_END\n";
38          .printf "PEB_INFO_START\n";
39          !peb;
40          .printf "PEB_INFO_END\n";
41          .printf "EVENT_DESCRIPTOR_START\n";
42          dt nt!_EVENT_DESCRIPTOR @$t2;
43          .printf "EVENT_DESCRIPTOR_END\n";
44          .printf "EVENT_JSON_FORMAT_START\n";
45          as /x ${/v:LOGGER_ID} @$t19;
46          .block{!wmitrace.logdump ${LOGGER_ID} -t 1}
47          ad LOGGER_ID
48          .printf "EVENT_JSON_FORMAT_END\n";
49          .printf "WRITE_INFO_END\n";
50          gc;
51      }
52      .else{
53          gc;
54      }

```

Fig. 5.51: Versión final del script interno de WinDBG encargado de imprimir toda la información de interés de una escritura realizada mediante la llamada a *EtwpEventWriteFull*. Es básicamente una combinación de todos los fragmentos antes mencionados.

Esta secuencia de comandos interna se “llamará” desde el script main ilustrado en la figura 5.47. Sin embargo, este main script también debe llamarse desde algún lugar. El último paso que fue necesario realizar para encadenar todos los scripts explicados en esta sección, fue agregar el script de extracción principal *EtwpeventwriteFull* dentro del script de búsqueda de logger id de DiagTrack representado en la figura 5.40. En particular, el “\$\$ El marcador de posición <PLACE FOR FUTURE CODE>” debe reemplazarse por:

```

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

```

Fig. 5.52: Ilustración sobre cómo debe reemplazarse la parte final del script de WinDBG representado por la Figura 5.40. Básicamente, esto encadena la ejecución del primer script (encargado de buscar la identificación del registrador) con los scripts que imprimen información interesante a partir de una llamada de escritura.

Dos avisos importantes:

1. El desplazamiento *0x371* se obtiene de la explicación de 5.42.
2. Aunque no se muestra aquí, el mismo análisis realizado con *EtwpeventwriteFull* se llevó a cabo con la función *EtwpWriteUserEvent*. .

Los scripts principales e internos para extraer información de *EtwpWriteUserEvent* se muestran a continuación:

```
1  $$
2  $$ ARG1: Path to the internal script for this function
3  $$
4  $$ Compare the current LOGGER ID with the DiagTrack one.
5  .if (ebx == @$t19){
6      $$ In this function, the guid entry is situated in [rbp+360h-340h]
7      r $t1 = poi(rbp+360h-340h);
8
9      $$ In this function, the event_descriptor is situated in [rbp+360h-330h]
10     + 0x28
11     r $t2 = poi(rbp+360h-330h)+0x28;
12
13     $$ So, once we know that the provider is going to ask for buffers related
14     $$ with the telemetry, we jump to a part of the same function where the
15     $$ event should be already written to the buffers, and therefore we can
16     $$ print out the valuable information.
17     bp /1 nt!EtwpWriteUserEvent+0xd47 "$$<\"${$arg1}\\""
18     gc;
19 }
20 .else{
21     gc;
22 }
```

Fig. 5.53: Script principal de WinDBG encargado de establecer el punto de interrupción dentro del *EtwpWriteUserEvent* para luego extraer la información de interés de la llamada de escritura.

```

1      r $t11 = poi(rbp+360h-320h);
2      r $t12 = poi(rbp+360h-328h)+0x28;
3
4      $$ Just to ensure that we are still in the same write
5      .if ((@$t11 == @$t1) & (@$t12 == @$t2)){
6          .reload
7          .printf "WRITE_INFO_START\n";
8
9          $$ If the process is in svchost, we want to have the service related to
10             the actual thread.
11          $$ In order to do that, we should print the tag of the service related to
12             the tread and
13          $$ afterwards compare it against the dump of the "services db".
14
15          .foreach /pS 13 (proc_name {!process -1 0 }) { aS ${/v:ProcessName}
16              "${proc_name}" };
17          .block{r $t10 = 0x0;r $t10 = $spat("${ProcessName}", "*svchost.exe*")}
18          .if ($t10 == 1){
19              .foreach /pS 5 (teb_addr {!thread @thread }) { r $t12 = teb_addr
20                  ;.break }
21              .printf "SERVICE_TAG_START\n"
22              .foreach /pS 1 (tag { dd @$t12+0x1720 L1 }) { .echo tag}
23              .printf "SERVICE_TAG_END\n"
24          }
25
26          .printf "DATE_TIMESTAMP_START\n";
27          .echotime;
28          .printf "DATE_TIMESTAMP_END\n";
29          .printf "PROVIDER_GUID_START\n"
30          .printf "%N %N\n", poi(@$t1+0x18), poi(@$t1+0x18+0x8);
31          .printf "PROVIDER_GUID_END\n"
32          .printf "CALL_STACK_START\n";
33          kc;
34          .printf "CALL_STACK_END\n";
35          .printf "PROCESS_INFO_START\n";
36          !process -1 0;
37          .printf "PROCESS_INFO_END\n";
38          .printf "PEB_INFO_START\n";
39          !peb;
40          .printf "PEB_INFO_END\n";
41          .printf "EVENT_DESCRIPTOR_START\n";
42          dt nt!_EVENT_DESCRIPTOR @$t2;
43          .printf "EVENT_DESCRIPTOR_END\n";
44          .printf "EVENT_JSON_FORMAT_START\n";
45          as /x ${/v:LOGGER_ID} @$t19;
46          .block{!wmitrace.logdump ${LOGGER_ID} -t 1}
47          ad LOGGER_ID
48          .printf "EVENT_JSON_FORMAT_END\n";
49          .printf "WRITE_INFO_END\n";
50          gc;
51      }
52      .else{
53          gc;
54      }

```

Fig. 5.54: Script interno de WinDBG encargado de imprimir información interesante a partir de una llamada de escritura realizada a través de *EtwpWriteUserEvent*. Este script es análogo al representado en la figura 5.51.

5.5 Triggers

5.5.1 Buscando nuevos triggers

En secciones anteriores, nos enfocamos mucho en explicar cómo los diferentes proveedores de ETW escribían eventos en las sesiones de ETW. Sin embargo, se pasó por alto un detalle muy importante. A pesar del hecho de que saber dónde va a escribir el proveedor es fundamental, si estableciésemos un breakpoint justo antes de la ejecución de la función de escritura, tendríamos que esperar hasta que el proveedor realmente tuviera que escribir. Un análisis sin poder controlar cuando ocurre la acción que se quiere inspeccionar, es realmente difícil de realizar.

Esa es la razón por la que se decidió construir lo que llamamos “Triggers”. Desarrollados con diferentes lenguajes, una vez que se ejecutan estos binarios, obligan a proveedores específicos a registrar un evento en la sesión de ETW de DiagTrack. Haber desarrollado estos binarios nos permitió hacer un análisis mejor y más preciso en menos tiempo.

Al final de esta investigación, habíamos desarrollado alrededor de cinco o seis disparadores diferentes. A pesar de tener sus propias particularidades, algunos de estos triggers se encontraron utilizando las mismas estrategias. En las siguientes secciones se describirá en detalle los Triggers más representativos encontrados durante el proyecto.

5.5.2 Notepad

Este Trigger fue uno de los primeros encontrados. Para encontrarlo, se estableció un breakpoint al comienzo de *EtwpEventWriteFull*. Después de varios minutos usando la computadora debugee como un usuario normal, se alcanzó el breakpoint.

Una vez dentro de *EtwpEventWriteFull* y con la ayuda de WinDBG (sección 2.2.2) fue posible recuperar el stack de llamadas completa de esa escritura y entender cuál era el programa/clase/objeto inicial que originó la llamada.

Después de analizar este comportamiento varias veces, se pudo concluir que solo con abrir y cerrar un proceso de Notepad, algunos proveedores de ETW escribirán un evento en la sesión de ETW de DiagTrack. Sin embargo, esto no siempre fue así. Según el nivel de Telemetría configurado, 0, 1 o 2 proveedores escribirán en la sesión de ETW. La siguiente tabla ilustra este comportamiento con información adicional:

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

La siguiente lista es una extracción de la información recopilada durante la captura de una escritura usando este Trigger:

```

1 PROVIDER_GUID: 46D31B9D487D6E37 538ADF8BCE54FDA8
   nt!EtwEventWriteFull+0x371:
2
3 PROCESS fffffd868344c480
4 SessionId: 1 Cid: 0774 Peb: fffffd868344c480; !peb
   821eee7000">821eee7000 ParentCid: 0e68">0e68
5 DirBase: 53cc5000 ObjectTable: fffffc7004a4e8b00 HandleCount: <Data Not
   Accessible>
6 Image: notepad.exe
7
8 +0x000 Id           : 0x8bd
9 +0x002 Version      : 0 ''
10 +0x003 Channel      : 0xb ''
11 +0x004 Level        : 0x5 ''
12 +0x005 Opcode       : 0 ''
13 +0x006 Task         : 0
14 +0x008 Keyword      : 0x00008000'00000800
15 fffff802'6ceab361 488bc8      mov     rcx,rcx
16
17 kd> !wmitrace.logdump 0x22 -t 1
18 [0]1310.0FF8:: 131717129297892613
   [Win32kTraceLogging/AppInteractivitySummary/]{\"UTCReplace_AppId\":
   \"{00001310-0001-00f[redacted]}

```

Fig. 5.55: Salida de un script WinDBG particular que muestra el GUID del proveedor, la información del proceso, el descriptor de eventos y el contenido del evento recopilados después de ejecutar el trigger del Bloc de notas.

Para ayudarnos con la futura automatización de estas pruebas, se desarrolló un script C# súper simple que imita esta misma acción.

```

1      using System;
2      using System.Diagnostics;
3      namespace TriggerNotepad
4      {
5          class Program
6          {
7              static void Main(string[] args)
8              {
9                  Process.Start("notepad.exe");
10                 String cmd = "/C taskkill /im notepad.exe";
11                 System.Threading.Thread.Sleep(2000);
12
13
14                 Process process = new Process();
15                 ProcessStartInfo startInfo = new ProcessStartInfo();
16                 startInfo.WindowStyle = ProcessWindowStyle.Hidden;
17                 startInfo.FileName = "cmd.exe";
18                 startInfo.Arguments = cmd;
19                 process.StartInfo = startInfo;
20                 process.Start();
21             }
22         }
23     }

```

Fig. 5.56: C# código fuente del trigger del Bloc de notas

5.5.3 Census

Mientras se buscaba información sobre posibles eventos que se registrarían en la sesión ETW de DiagTrack en Internet, se encontró un recurso web muy interesante de Microsoft[8]. Dentro de esta documentación había una sección dedicada a los eventos de Census. Este tipo de eventos estaban relacionados con aplicaciones instaladas dentro del sistema, información de hardware, información de memoria y más.

Después de analizar esta información, se encontró que con solo ejecutar el binario DeviceCensus.exe que viene por defecto dentro del directorio system32 de Windows, se estaba escribiendo un evento en la sesión ETW de DiagTrack. Como se explicó en el Trigger anterior, esto también dependería del nivel de Telemetría configurado. La siguiente tabla explica para cada nivel, qué proveedores de ETW escribirán un nuevo evento una vez que se ejecute este disparador:

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

La siguiente lista es una extracción de la información recopilada durante la captura de una escritura usando este Trigger:

```

1 PROVIDER_GUID: 46CF5C84262CDE7A 69EF913796942094
2
3 PROCESS ffff8e0d8a6e9080
4     SessionId: 1 Cid: 0f58 Peb: f0276b9000 ParentCid: 04e4
5     DirBase: 67a88000 ObjectTable: fffffc1882c3fa580 HandleCount: <Data Not
        Accessible>
6     Image: DeviceCensus.exe
7
8 START_EVENT_DESCRIPTOR
9     +0x000 Id           : 0x1fe
10    +0x002 Version      : 0 ''
11    +0x003 Channel      : 0xb ''
12    +0x004 Level        : 0x5 ''
13    +0x005 Opcode       : 0 ''
14    +0x006 Task         : 0
15    +0x008 Keyword      : 0x00008000'00000000
16 END_EVENT_DESCRIPTOR
17 START_EVENT_JSON_FORMAT
18 (WmiTrace) LogDump for Logger Id 0x21
19 Found Buffers: 4 Messages: 18, sorting entries
20 [0]0F58.133C:: 131728645762917499 [Census/App/]{ "__TlgCV__":
        "X0BX596o1kyBj176.0", "IEVersion": "11.0.14393.0", "CensusVersion":
        100143930000}
21 Total of 18 Messages from 4 Buffers

```

Fig. 5.57: Salida de una secuencia de comandos WinDBG particular que muestra el GUID del proveedor, la información del proceso, el descriptor de eventos y el contenido del evento recopilados después de ejecutar el trigger del Census.

La siguiente lista contiene el código fuente del binario desarrollado para ser utilizado como Trigger de Census:

```

1      using System;
2      using System.Diagnostics;
3      using System.Runtime.InteropServices;
4
5      namespace triggerCensus
6      {
7          class Program
8          {
9              static void Main(string[] args)
10             {
11                 Process process = new Process();
12                 ProcessStartInfo startInfo = new ProcessStartInfo();
13                 startInfo.WindowStyle = ProcessWindowStyle.Hidden;
14                 startInfo.FileName = "cmd.exe";
15                 startInfo.Arguments = "/C
16                     C:\\Windows\\System32\\DeviceCensus.exe";
17                 process.StartInfo = startInfo;
18                 process.Start();
19                 process.WaitForExit();
20             }
21         }
22     }

```

Fig. 5.58: C# código fuente del trigger de Census.

5.5.4 One Drive

Otro Trigger interesante que fue posible desarrollar gracias a la Documentación de Microsoft[8] fue el de OneDrive. Este Trigger, aunque un poco inestable, se desarrolló porque tenía una capacidad muy interesante: Incluso cuando el nivel de seguridad de la Telemetría está configurado, se loguean eventos. Sin embargo, después de analizar e invertir bastante tiempo en comprender el comportamiento de este Trigger, no fue posible desarrollar una versión estable. En otras palabras, no pudimos entender qué condiciones necesita para funcionar.

Para hacer uso de este Trigger, se utilizó el binario OneDriveStandaloneUpdater.exe. La siguiente lista es una extracción de la información recopilada durante la captura de una escritura usando este 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

La siguiente lista contiene el código fuente del binario desarrollado para ser utilizado como Trigger de Census:

```

1 PROVIDER_GUID: 4C59584AD34D654D BD7D81B2A46938B2 nt!EtwWriteUserEvent+0x48d:
2
3 PROCESS ffff8002598ac080
4     SessionId: 1 Cid: 1240 Peb: 0040d000 ParentCid: 0370
5     DirBase: 4e24e000 ObjectTable: ffff9301e7f9d300 HandleCount: <Data Not
        Accessible>
6     Image: OneDriveStandaloneUpdater.exe
7
8     +0x000 Id           : 0xa7
9     +0x002 Version      : 0 ''
10    +0x003 Channel       : 0xb ''
11    +0x004 Level         : 0x5 ''
12    +0x005 Opcode        : 0 ''
13    +0x006 Task          : 0
14    +0x008 Keyword       : 0x00008000'00000000
15    fffff801'b080295d 418bd7      mov     edx,r15d

```

Fig. 5.59: Salida de un script WinDBG particular que muestra el GUID del proveedor, la información del proceso, el descriptor de eventos y el contenido del evento recopilados después de ejecutar el trigger de OneDrive.

El siguiente código fuente hace referencia al programa desarrollado para ayudar con la automatización del Trigger de OneDrive:

```
1      using System;
2      using System.Diagnostics;
3      using System.Runtime.InteropServices;
4
5      namespace onedrivestandalonetrigger
6      {
7          class Program
8          {
9              static void Main(string[] args)
10             {
11                 Process process = new Process();
12                 ProcessStartInfo startInfo = new ProcessStartInfo();
13                 startInfo.WindowStyle = ProcessWindowStyle.Hidden;
14                 startInfo.FileName = "cmd.exe";
15                 string userNameAndHost =
16                     System.Security.Principal.WindowsIdentity.GetCurrent().Name;
17                 int indexOfSlash = userNameAndHost.IndexOf("\\");
18                 string userName = userNameAndHost.Substring(indexOfSlash+1);
19                 startInfo.Arguments = "/C C:\\Users\\"+ userName +
20                     "\\AppData\\Local\\Microsoft\\OneDrive\\OneDriveStandaloneUpdater.exe";
21                 process.StartInfo = startInfo;
22                 process.Start();
23                 process.WaitForExit();
24             }
25         }
26     }
```

Fig. 5.60: C# código fuente del trigger de OneDrive

6. CONCLUSIONES

En este proyecto analizamos cómo funciona Telemetría de Windows desde una perspectiva interna. Debido a que formaba parte del kernel de Windows y su código fuente estaba cerrado, se tuvo que utilizar una combinación de varias técnicas para comprender cómo funcionaba este componente. La ingeniería inversa y el debugging de kernel fueron las dos técnicas principales que nos permitieron profundizar en el corazón de Telemetría y nos llevaron a comprender cuáles y cómo se llevaban a cabo sus procesos y tareas internas. Además, al entender estructuras complejas del Kernel y desarrollar varios scripts para automatizar la recopilación de información, establecimos una base que simplificará las investigaciones y, por lo tanto, se aprovechará para trabajos futuros, ya sea para los mismos o diferentes componentes.

La ingeniería inversa es una herramienta poderosa. A pesar de que el código fuente del kernel de Windows está cerrado, fue posible comprender la funcionalidad interna de varios componentes al poder leer su código Assembler. Esto significa que, por lo general, con el tiempo adecuado, sin importar qué proceso, servicio, aplicación o ejecutable esté tratando de analizar, podrá comprender cómo funciona y funciona incluso sin su código fuente. En algunos casos, el análisis podría volverse más difícil o incluso imposible cuando se implementan técnicas de ofuscación; sin embargo, este no es el caso del kernel de Windows. Finalmente, la combinación de ingeniería inversa con análisis dinámico (poder hacer debugging) facilita aún más las cosas, ya que es posible detenerse en algunos puntos específicos e inspeccionar el estado actual de la máquina con el objetivo de comprender mejor la situación actual.

A lo largo del viaje, surgieron varios desafíos. Uno de ellos es el uso de estructuras Kernel que no estaban documentadas en absoluto. Como se mencionó en la introducción, leer y analizar código Assembler es más difícil que leer código de alto nivel. Por lo tanto, el proceso de aprender cómo se usan algunas estructuras cuando se desconocen se vuelve aún más difícil. Comprender su definición, sus offsets o incluso la semántica de cada puntero almacenado en ellos fue una de las partes más difíciles pero más gratificantes de toda la investigación. Creemos que haber develado todas estas estructuras podría ayudar a otros a evitar pasar por el mismo proceso.

Otro gran desafío enfrentado y superado fue el relacionado con la detección del proceso que estaba escribiendo un evento. La capacidad de **svchost.exe** para “ocultar” varios servicios dentro de él, nos impedía detectar con precisión la entidad detrás del envío de un nuevo evento. Como se explica en la sección 5.3, después de probar varias opciones, fue posible encontrar una manera de detectar qué servicio se estaba ejecutando en algún punto en particular (incluso en el caso de una escritura proveniente de **svchost.exe**) al inspeccionar dos cosas: el Thread Environment Block del proceso (TEB) y la base de datos de tags de los servicios. Basado en la investigación realizada contra el ejecutable **services.exe** y su implementación de asignación de tags, entendimos cómo se asignaban tags a los diferentes servicios una vez que se registraban.

Aprovechando el poder de Windbg y su lenguaje de script, fue posible crear una herramienta automática para el análisis de eventos. Al combinar todos los scripts de Windbg presentados, creamos una herramienta automatizada capaz de capturar y analizar los eventos registrados por los proveedores de ETW, justo antes de que se escriban en los buffers

de las sesiones. A pesar de que para poder usar esta herramienta, el debugging del kernel debe estar activado, poder controlar la ejecución justo antes de que se registre el evento brinda mucho poder y flexibilidad.

Finalmente, el desarrollo de Triggers fue clave durante todo el proyecto. Al poder realizar un análisis dinámico y, por lo tanto, leer el stack de llamadas cuando se produjo una escritura de ETW, fue posible comprender qué requisitos deben cumplirse para obligar a proveedores de ETW particulares a escribir un evento. Desarrollar estos “triggers” nos permitió realizar innumerables análisis y pruebas de forma controlada al saber quién, cuándo y qué registraban.

7. TRABAJO FUTURO

Como se indicó en la sección 6, uno de los objetivos de este proyecto fue sentar las bases, contribuir y ayudar a futuras investigaciones.

Habiendo explicado cómo combinar la ingeniería inversa y el debugging de kernel con el objetivo de realizar un análisis del kernel de un sistema operativo, esperamos alentar y permitir que otros investigadores o entusiastas repliquen estas técnicas para comprender mejor otros componentes, ya sea en el mismo sistema operativo o en uno diferente. Además, todas las estructuras y funcionalidades del kernel invertidas en este proyecto podrían ayudar a los investigadores de seguridad a comprender mejor los detalles de Telemetría o ETW y, por lo tanto, ayudarlos con el proceso de evaluación de su seguridad general. Por ejemplo, podría ayudar a encontrar una corrupción de memoria o cualquier otra vulnerabilidad potencial que podría terminar en una falla grave que afecte al sistema operativo Windows.

Creemos que el desarrollo de los scripts de Windbg podría inspirar a otras personas a crear proyectos nuevos e innovadores simplemente modificando o ampliando las funciones actuales. A pesar de que todos los scripts de Windbg desarrollados durante este trabajo se dedicaron a analizar la sesión de ETW **DiagTrack-Listener**, también se pensaron con la idea de bajo acoplamiento en mente. Como ejemplo, un simple cambio en el nombre del Logger que se compara en 5.39 (línea 9), permite realizar el mismo análisis con cualquier otra sesión ETW arbitraria. En términos más generales, este proyecto se dedicó a comprender cómo funcionaba la telemetría desde una perspectiva interna. Sin embargo, existen otros tipos de análisis que se podrían realizar frente a la Telemetría, donde nuestro trabajo podría ayudar a realizarlos. Por ejemplo, el análisis de privacidad de los datos contenidos dentro de los eventos que realmente se envían a los sistemas backend. Aunque varios trabajos anteriores ya se centraron en esa parte, al aprovechar la capacidad de saber cómo los almacena Telemetría, permitiría inspeccionarlos automáticamente incluso antes de enviarlos. Como consecuencia, se podrían desarrollar varias herramientas, como por ejemplo una herramienta de “Bloqueador de eventos”. Otro ejemplo interesante podría ser: Realizar un análisis de seguridad contra la forma en que se envían los eventos a los servidores backend. Tener el poder de modificar el contenido de un evento justo antes de ser enviado podría proporcionar, al menos, dos características diferentes. Primero, podría usarse para anonimizar u ofuscar alguna información que por alguna razón alguien quiera ocultar. Segundo, podría proporcionar una forma de enviar mensajes arbitrarios, con contenido arbitrario y, por lo tanto, comprobar si hay problemas de seguridad en la forma en que el servidor de backend de Microsoft analizan los eventos.

Finalmente, los “Triggers” desarrollados, aunque no parezcan demasiado cruciales, podrían ser clave para futuras investigaciones. En primer lugar, proporcionan una forma de realizar pruebas fácilmente al obligar a que se escriban nuevos eventos en la sesión ETW correspondiente. Eso, podrían facilitar el trabajo de un investigador que está tratando de realizar un análisis de los eventos. Además, proporcionan diferentes ejemplos de procesos/binarios (proveedores de ETW) que realmente están escribiendo en la sesión de ETW. Por lo tanto, podrían usarse como base para sacar patrones de ellos con el objetivo de construir un modelo de un proveedor ETW (relacionado con Telemetría o no).

8. APÉNDICES

Simple .exe that creates a new file

Desarrollado en C++.

```
1      #include "pch.h"
2      #include <iostream>
3      #include <fstream>
4
5      int main()
6      {
7          std::ofstream outfile("C:\\Users\\<user>\\Desktop\\test.txt");
8          outfile << "Hello world!\n";
9          outfile.close();
10     }
```

Fig. 8.1: C# código fuente de un simple programa para ser ejecutado como un servicio.

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. Müller, 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.