UCAM UNIVERSIDAD

Diseño e implementación de un EDR básico para Windows

Una aproximación práctica al desarrollo de sensores y agentes de seguridad

Noel Carrasco

ENIIT Innova IT Business School
Campus Internacional de Ciberseguridad
Máster en Análisis de Malware, Reversing y Bug Hunting

Barcelona, agosto 2025

Página intencionalmente dejada en blanco.

Diseño e implementación de un EDR básico para Windows

Una aproximación práctica al desarrollo de sensores y agentes de seguridad

Noel Carrasco

Supervisor: Sergio De Los Santos
Director Técnico, Master en Reversing

ENIIT Innova IT Business School
Campus Internacional de Ciberseguridad
Máster en Análisis de Malware, Reversing y Bug Hunting

Proyecto de Final de Máster

Barcelona, agosto 2025

Página intencionalmente dejada en blanco.

Diseño e implementación de un EDR básico para Windows

Página intencionalmente dejada en blanco.

Agradecimientos

Deseo expresar mi más profundo agradecimiento a mi tutor, cuya guía, paciencia y apoyo han sido esenciales para llevar a buen término la redacción de este trabajo. Su orientación constante me ha permitido avanzar con seguridad y confianza en cada etapa del proceso.

A mi familia, por su interés genuino y por estar siempre presentes, brindándome palabras de ánimo y acompañándome en este camino académico.

Y, de manera muy especial, a mi pareja, cuya comprensión, motivación y apoyo incondicional han sido la mayor fuente de fortaleza y el impulso más valioso para culminar este proyecto.

Página intencionalmente dejada en blanco.

Resumen

Hoy en día, muchas amenazas no dejan rastro en disco ni usan ejecutables evidentes. En su lugar, se apoyan en técnicas como la ejecución en memoria, el uso de herramientas legítimas del sistema o el "fileless malware". Frente a este tipo de ataques, los antivirus tradicionales son insuficientes, y se requiere un enfoque más dinámico. Aquí es donde entran los sistemas **EDR**, capaces de observar lo que ocurre en el sistema en tiempo real y reaccionar cuando se detectan comportamientos fuera de lo normal o habitual.

Este proyecto se centra en el desarrollo de un **EDR propio y funcional**, construido desde cero para sistemas Windows. La idea no es replicar una solución comercial, sino entender y controlar cada parte del proceso: desde cómo se recoge la telemetría en modo kernel, hasta cómo se procesan los eventos y se aplican reglas de detección en el agente.

El sistema recoge eventos relacionados con procesos, archivos, registro y red. A partir de esa información, se aplican **detecciones locales basadas en reglas**, y si algo encaja con un patrón definido, se pueden lanzar respuestas básicas como terminar un proceso o bloquear una operación. Todo funciona sin depender de la nube, pensado para entornos donde se necesita control total.

Además de ser una herramienta práctica, este EDR sirve como campo de pruebas para explorar cómo detectar actividad maliciosa desde la propia máquina. También se puede utilizar como laboratorio para aprender cómo construir sensores eficientes y saber qué limitaciones surgen al desarrollar este tipo de sistemas desde los cimientos del kernel.

Palabras-Clave: EDR (Endpoint Detection and Response), Detección de malware, Windows.

Página intencionalmente dejada en blanco.

Abstract

Nowadays, many threats leave no trace on disk and don't rely on obvious executables. Instead, they use techniques like in-memory execution, abuse of legitimate system tools, or fileless malware. Against this type of attack, traditional antivirus solutions fall short, and a more dynamic approach is required. This is where **EDR systems** come into play — capable of observing system activity in real time and reacting when abnormal or suspicious behavior is detected.

This project focuses on developing a **custom, functional EDR**, built entirely from scratch for Windows systems. The goal is not to replicate a commercial solution, but to understand and control every part of the process: from how telemetry is collected in kernel mode to how events are processed and detection rules are applied by the user-mode agent.

The system collects events related to processes, file operations, registry modifications, and network activity. Based on this information, it applies **rule-based local detections**, and if something matches a known pattern, basic response actions can be triggered — such as terminating a process or blocking an operation. Everything runs locally, without relying on the cloud, making it suitable for environments that require full control.

Beyond its practical use, this EDR also acts as a testbed for exploring how to detect malicious activity from within the endpoint itself. It also serves as a hands-on environment to learn how to build efficient sensors and to understand the challenges of developing this type of system from the kernel up.

Keywords: EDR (Endpoint Detection and Response), Malware detection, Windows.

Página intencionalmente dejada en blanco.

Índice general

Apéndices

Página intencionalmente dejada en blanco.

Índice de figuras

Página intencionalmente dejada en blanco.

Índice de cuadros

Página intencionalmente dejada en blanco.

Glosario

Detour Función que reemplaza temporalmente una API original del sistema. Permite interceptar llamadas, aplicar lógica personalizada y reenviar la llamada original. (*p. 12, 43–47, 50, 52*)

DeviceIoControl Función de la API de Windows que permite realizar operaciones personalizadas entre espacio de usuario y drivers, mediante IOCTLs definidos por el desarrollador. (*p. 43, 49, 52, 53*)

Driver del Kernel Módulo en modo kernel encargado de capturar telemetría a bajo nivel. Implementa sensores que observan procesos, archivos, red y memoria, comunicándose con el agente en espacio de usuario. (*p. 16, 17*)

Espacio de kernel Modo privilegiado de ejecución en el que operan el núcleo del sistema operativo y los drivers, con acceso total a recursos. (*p. 15, 16*)

Espacio de usuario Área de ejecución del sistema operativo donde corren las aplicaciones con privilegios restringidos, sin acceso directo al hardware. (*p. 15–17, 30, 35, 49, 53*)

Fast Fail Mecanismo de seguridad de Windows que termina inmediatamente un proceso cuando detecta condiciones que podrían comprometer la estabilidad del sistema, como corrupción del heap. (*p. 53*)

Handle Valor opaco devuelto por funciones del sistema operativo que representa una referencia a un objeto del kernel, como una sección de memoria, archivo o proceso. Usado para realizar operaciones posteriores sobre dicho objeto. (*p. 49*)

Hooking Técnica utilizada para interceptar llamadas a funciones o modificar su comportamiento, común en desarrollo de software e ingeniería inversa. (*p. 7, 12, 13, 18, 43, 44, 49, 50*)

Minifilter Controladores en modo kernel que se cargan en la pila del sistema de archivos de Windows y permiten interceptar y modificar operaciones de entrada/salida, útiles para monitorizar accesos a archivos. (*p. 9, 10*)

Modularidad Principio de diseño en el que un sistema se divide en componentes independientes que pueden desarrollarse, probarse y desplegarse por separado. (*p. 16*)

Prost Librería en Rust que permite compilar archivos .proto y generar estructuras con tipado fuerte compatibles con Protocol Buffers. (*p. 49*)

Protocol Buffers Formato de serialización binaria desarrollado por Google, usado para estructurar datos de forma compacta y eficiente en comunicaciones. (*p. 5, 52*)

PsCallbacks Mecanismo del kernel de Windows que permite registrar funciones de callback para ser notificadas cuando se crean o terminan procesos. Se usa a través de funciones como PsSetCreateProcessNotifyRoutine. (*p. 7, 12, 13*)

Sensor Módulo especializado en capturar eventos de un dominio específico del sistema (como red, procesos o archivos) y enviarlos para análisis o almacenamiento. (*p. 15*)

Telemetría Recopilación automática de datos sobre el comportamiento y estado de un sistema informático para su análisis posterior. (*p. 4*)

Userland Agent Componente principal del sistema EDR que opera en el espacio de usuario. Se encarga de recibir eventos de los sensores, aplicar lógica de detección, almacenar datos y exponer APIs de comunicación. (*p. 16–18*)

Página intencionalmente dejada en blanco.

Siglas

EDR Endpoint Detection and Response. (*p. xv, 3–5, 7–13, 15, 17, 18*)
ETW Event Tracing for Windows. (*p. 4, 7–9, 12, 13*)

IOCTL Input/Output Control. (*p. 16, 18, 49, 52*)

SPSC Single Producer / Single Consumer. (*p. 27*)

WAL Write-Ahead Logging. (*p. 5, 16, 18, 19*)
WFP Windows Filtering Platform. (*p. xviii, 7, 11, 12*)

Página intencionalmente dejada en blanco.

1 Introducción

Autor: Noel Carrasco
Versión actual: 1.0.0
Repositorio oficial: Repositorio en GitHub

1.1. Estado del Arte

La forma en que detectamos y respondemos al malware ha evolucionado drásticamente en la última década. Las soluciones tradicionales, como los antivirus basados en firmas, aún tienen utilidad frente a amenazas conocidas, pero se quedan cortas ante técnicas modernas diseñadas para evadir este tipo de defensas. Actualmente, muchos ataques no requieren descargar archivos ni utilizar ejecutables identificables: se apoyan en técnicas fileless, abuso de herramientas legítimas del sistema (lo que se conoce como living-off-the-land) y ejecución de código directamente en memoria, evitando el disco por completo.

Este cambio en las tácticas ofensivas ha impulsado el desarrollo de una nueva generación de soluciones: los sistemas EDR (Endpoint Detection and Response). A diferencia de los antivirus tradicionales, un EDR no depende únicamente de firmas, sino que monitoriza el comportamiento del sistema en tiempo real, recolectando telemetría detallada para identificar actividades sospechosas incluso sin una firma conocida.

Hoy en día, existen varias soluciones EDR comerciales consolidadas, como CrowdStrike, SentinelOne, Defender for Endpoint o Carbon Black. Todas siguen una lógica similar: recopilan grandes volúmenes de eventos del sistema —creación de procesos, accesos a archivos, conexiones de red, cambios en el registro, entre otros— y los analizan, a menudo desde plataformas en la nube. También hay alternativas open source como Wazuh, Osquery o Velociraptor, que, aunque menos potentes, ofrecen una base sólida para aprendizaje, investigación o despliegues personalizados. El blog técnico de 0xflux, *FluxSec* 0xflux, 2023a, también me aportó ideas prácticas sobre la evolución de los EDR y sobre los retos que enfrentan este tipo de sistemas en escenarios reales.

En cuanto a las técnicas de detección, el enfoque suele ser híbrido. Por un lado, aún se utilizan firmas, útiles contra amenazas conocidas. Por otro, se recurre a heurísticas que identifican

patrones anómalos, y cada vez con más frecuencia, a modelos de comportamiento y machine learning, especialmente en entornos con grandes volúmenes de datos donde el análisis manual es inviable. Sin embargo, estos métodos avanzados conllevan riesgos: si no están bien calibrados, pueden generar falsos positivos o pasar por alto amenazas reales.

Todo este ecosistema gira en torno a un elemento fundamental: la recolección de datos. Son los sensores y agentes desplegados en los endpoints los encargados de observar y registrar lo que ocurre en el sistema. En Windows, herramientas como Sysmon o tecnologías como ETW (Event Tracing for Windows) permiten interceptar eventos relevantes sin necesidad de tocar el núcleo. En Linux, soluciones como AuditD o eBPF ofrecen visibilidad a bajo nivel con un impacto controlado. A partir de estos datos, los agentes pueden preprocesar la información e incluso actuar localmente: aislar una máquina, finalizar un proceso, o notificar al sistema central si se detecta una amenaza.

A pesar de sus capacidades, los EDR actuales no están exentos de limitaciones. Muchas soluciones comerciales tienen un consumo elevado de recursos, sobre todo al habilitar módulos avanzados. Algunas dependen exclusivamente de la nube, lo que las vuelve inutilizables en entornos aislados. Además, al ser plataformas cerradas, su personalización y adaptación a escenarios específicos puede ser muy limitada. Y más allá de la tecnología, detectar malware real, especialmente el diseñado a medida, sigue siendo un reto complejo.

Este proyecto surge precisamente con el propósito de explorar ese terreno. La idea es diseñar y construir un EDR propio, comenzando desde lo esencial: sensores, agentes y una lógica de detección que pueda evolucionar. No se trata de imitar soluciones comerciales, sino de tener control total sobre qué eventos observar, cómo analizarlos y qué acciones tomar. Este entorno también servirá como campo de pruebas para experimentar con nuevas estrategias de detección, evaluar su eficacia frente a malware real y, sobre todo, entender en profundidad lo que implica defender un sistema desde dentro.

1.2. Motivación

El origen de este proyecto no responde tanto a una necesidad concreta, sino a una motivación personal y técnica: entender a fondo cómo funcionan los sistemas EDR. Esta idea se me ocurrió tras leer el libro *Evading EDR* de Matt Hand Hand, 2023, donde se analizan en detalle los mecanismos internos de un EDR. Ese libro fue la base sobre la que empecé a construir este proyecto. Durante años se ha hablado de los EDR como soluciones avanzadas y eficaces, pero también complejas y exigentes en cuanto a recursos. Ese discurso me generó curiosidad, pero fue a raíz de empezar a investigar, construir y diseccionar piezas de malware cuando surgió la verdadera pregunta: ¿cómo detectan las empresas software tan sofisticado y evasivo?

Esa inquietud me llevó a explorar el corazón de estas soluciones: quería saber qué componentes las integran, cómo identifican comportamientos maliciosos, por qué consumen tantos recursos y, sobre todo, qué tan difícil es construir mecanismos de detección robustos. Quería pasar del análisis superficial al entendimiento real, desmontando pieza por pieza lo que hay detrás de un sistema de defensa moderno.

Además, este proyecto me dio una excusa perfecta para adentrarme en una capa del sistema

que siempre me había interesado pero que también imponía respeto: el kernel. Mi interés en profundizar en las capacidades del kernel de Windows se reforzó gracias al libro de Pavel Yosifovich Yosifovich, 2023, que me sirvió de guía para entender mejor sus estructuras internas y protecciones. Una parte importante del reto ha sido precisamente esa: enfrentar las restricciones del entorno, descubrir hasta dónde se puede llegar sin comprometer la estabilidad del sistema, y adaptar el diseño de sensores o agentes a las particularidades de Windows. No se trata solo de que el código funcione; se trata de que sea eficiente, seguro y respetuoso con el entorno en el que se ejecuta.

En resumen, este trabajo es un desafío personal más que un producto terminado. Es una forma de llevar mis conocimientos en ciberseguridad y programación un paso más allá, creando un espacio de aprendizaje práctico. Cada decisión de diseño, cada obstáculo técnico, representa una oportunidad para aprender cómo se defiende un sistema desde dentro. Y cada encuentro con una muestra real de malware deja de ser un ejercicio teórico para convertirse en una experiencia tangible: un enemigo concreto que hay que saber reconocer, comprender y contener.

1.3. Objetivos del Proyecto

El objetivo general de este proyecto es **diseñar e implementar un sistema EDR funcional** capaz de recolectar telemetría en tiempo real, detectar actividad maliciosa y ejecutar respuestas automatizadas dentro de un entorno Windows.

La idea es empezar desde el núcleo del sistema. Por un lado, se van a desarrollar sensores en modo kernel capaces de capturar eventos como: operaciones sobre archivos, creación de procesos, cambios en el registro y actividad de red. Estos sensores alimentarán un agente en modo usuario que se encargará de procesar esa información, aplicar reglas de detección —como hashes sospechosos— y tomar decisiones simples, como registrar eventos o terminar un proceso. También se va a prestar atención a la comunicación entre kernel y userland, que en Windows no es trivial.

Por último, se va a poner a prueba el sistema en un entorno controlado. No basta con que el código compile o que los logs aparezcan: hay que **ver cómo reacciona frente a situaciones reales**, aunque sean simuladas. Esto incluye ataques básicos, uso de herramientas legítimas para comportamientos maliciosos y otras técnicas comunes. La meta es comprobar hasta dónde se puede llegar con reglas simples y qué limitaciones aparecen cuando el entorno se vuelve más complejo.

1.3.1. Alcance del Proyecto

Este trabajo está enfocado en construir un EDR básico pensado para sistemas Windows. La intención es cubrir el ciclo completo: captura de eventos, procesamiento, detección y respuesta.

Qué se va a construir

Dentro del alcance del proyecto se incluyen:

1. Sensores en modo kernel capaces de observar operaciones relevantes:
    - Acceso y modificación de archivos

   - Creación y terminación de procesos
   - Cambios en el registro de Windows
   - Establecimiento de conexiones de red
2. Un agente en modo usuario que:
   - Reciba la telemetría del kernel
   - Aplique detecciones básicas (por ejemplo, hashes conocidos o reglas YARA)
   - Ejecute acciones simples como matar un proceso o registrar un evento
3. Mecanismos de comunicación IPC entre los sensores y el agente, diseñados para ser fiables sin comprometer la estabilidad del sistema.
4. Un entorno de pruebas con simulaciones de ataque para evaluar cómo responde el sistema ante actividades sospechosas reales o artificiales.

Qué queda fuera

Por razones prácticas y de enfoque, el proyecto deja fuera varios elementos que suelen estar presentes en soluciones EDR avanzadas:

- No habrá **procesamiento en la nube** ni correlación de eventos entre múltiples endpoints
- No se utilizarán **modelos de machine learning** ni análisis de comportamiento avanzados
- No se pretende un despliegue masivo ni soportar entornos productivos reales
- No se contempla compatibilidad con sistemas que no sean Windows
- No se incorporan técnicas avanzadas de **protección contra evasión** o resistencia a manipulación

Este trabajo se lleva a cabo en entornos virtualizados, no en máquinas reales en producción. Los drivers se firman con certificados de prueba, y la lógica de detección se basa en reglas estáticas, sin sandbox ni análisis dinámico. Todo esto se asume desde el inicio, porque el foco no está en la robustez final, sino en la comprensión técnica del sistema.

1.4. Resumen de la Metodología

Este proyecto adopta un enfoque **bottom-up** centrado en construir un sistema EDR desde sus cimientos, empezando por el desarrollo de componentes en modo usuario y avanzando hacia la lógica en modo kernel. En lugar de utilizar frameworks preexistentes, se opta por desarrollar cada parte desde cero, lo que permite un control total sobre la implementación y el flujo de datos.

El primer paso consiste en estudiar los componentes de Windows que permiten observar el sistema. Entre estos se encuentran los *minifilters* para interceptar operaciones de archivos, los *callbacks* para procesos y registro, y Event Tracing for Windows (ETW) para el tráfico de red. Con esa base, se crean **drivers** que actúan como sensores y reportan la Telemetría que tanto nos interesa.

Sobre esta telemetría se construye un agente en modo usuario, responsable de recibir los eventos, aplicar reglas de detección -como coincidencias con hashes o reglas YARA- y ejecutar respuestas simples como finalizar procesos o bloquear accesos.

Finalmente, el sistema se pone a prueba en un entorno virtual controlado con **ataques simulados**, para validar tanto la eficacia de las detecciones como la estabilidad general del conjunto. Las herramientas empleadas incluyen Visual Studio, WinDbg y entornos virtualizados de Windows, entre otros.

1.5. Entorno de desarrollo y herramientas

Todo el desarrollo se ha realizado sobre una máquina virtual con **Windows 10**, utilizando **VMware Workstation Pro**. La versión objetivo del sistema es **Windows 11** con SDK `10.0.26100` (también virtualizada), que es para la que se compilan tanto los binarios en modo usuario como el driver. El hecho de que este virtualizada ha ayudado a recuperar el sistema en caso de corromperlo.

El lenguaje principal del proyecto es **Rust**, aunque en algunos puntos concretos se ha recurrido a **C** para facilitar ciertas integraciones con APIs del sistema operativo que aún no están bien resueltas en el ecosistema Rust. En el caso del driver, se está utilizando la librería `windows-drivers-rs`, mantenida por Microsoft. Esta librería todavía está en desarrollo y tiene algunas limitaciones, pero permite empezar a escribir controladores en Rust sin necesidad de construir todo el soporte desde cero.

La comunicación entre componentes y la serialización de los eventos se resuelve con **Protocol Buffers** de Google. Se ha escogido este protocolo para mantener una estructura de datos compacta y consistente tanto en kernel como en modo usuario y poder desacoplar un poco ambos componentes.

Para el almacenamiento de eventos se ha optado por **SQLite** en modo **Write-Ahead Logging** (**WAL**), principalmente por su simplicidad, buen rendimiento en escrituras concurrentes y facilidad de integración sin depender de servicios externos.

En cuanto a las herramientas de desarrollo, además de **Visual Studio** y **RustRover** como IDE, se ha utilizado principalmente **WinDbg** para la depuración en kernel, lo cual es especialmente útil durante el desarrollo de drivers. También se han empleado varias herramientas de **Sysinternals**, como `WinObj`, `ProcExp` y `ProcDump`, que ofrecen interfaces de usuario más intuitivas y amigables en comparación con WinDbg para tareas concretas.

1.6. Estructura del Documento

A continuación se describe brevemente el contenido de cada capítulo:

- **Capítulo 2 – Fundamentos y Trabajos Relacionados:** Presenta los conceptos necesarios para entender cómo funciona un sistema EDR. Se repasan los mecanismos internos del sistema operativo Windows, los distintos sensores que utiliza un EDR así como algunas de las técnicas utilizadas por el software malicioso.
- **Capítulo 3 – Arquitectura y Diseño del Sistema:** Describe la arquitectura general del EDR desarrollado en este proyecto, detallando cómo se organizan los sensores, el agente, las reglas de detección y los mecanismos de respuesta.

- **Capítulo 4 – Implementación:** Explica paso a paso cómo se han desarrollado los componentes principales del sistema. Incluye detalles técnicos sobre la escritura de los drivers, la lógica del agente en modo usuario y la comunicación entre ambos.
- **Capítulo 5 – Evaluación y Resultados:** Muestra cómo se ha probado el sistema y qué resultados se han obtenido. Se analizan tanto su eficacia frente a ataques simulados como su impacto en rendimiento y estabilidad.
- **Capítulo 6 – Conclusiones y Trabajo Futuro:** Resume las principales aportaciones del proyecto, reflexiona sobre los retos encontrados y propone posibles mejoras y líneas futuras de trabajo para seguir desarrollando el sistema.

Además, en los apéndices se incluyen materiales de soporte como **registros de prueba**, **reglas de detección** y **fragmentos de código** relevantes, que complementan un poco el contenido de los capítulos más técnicos.

2
Fundamentos Técnicos

Antes de entrar en la arquitectura y desarrollo del sistema, es importante tener una base clara sobre los **componentes internos de Windows** más relevantes para un sistema EDR. Este capítulo no pretende explicar el sistema operativo de manera exhaustiva, sino centrarse en aquellos componentes que permiten recolectar telemetría para detectar actividad sospechosa.

En este capítulo se explican mecanismos como las **PsCallbacks**, el sistema **ETW**, los **minifilters del sistema de archivos**, la **Windows Filtering Platform** (**WFP**) para el tráfico de red, o técnicas como el **Hooking de funciones mediante DLLs**. La idea es entender qué hace cada uno, qué información proporcionan y por qué son útiles para construir sensores dentro de un EDR.

Más adelante, se abordarán también las técnicas de evasión y contramedidas. Por ahora, el foco está en conocer los puntos de observación que ofrece Windows para capturar lo que realmente pasa en el sistema.

2.1. Componentes Nativos de Windows

El corazón de cualquier EDR está en su capacidad para observar en tiempo real lo que ocurre en el sistema. Para ello, se apoya en diversas interfaces y subsistemas que Windows expone a los desarrolladores para actuar cuando se detectan cambios en procesos, archivos, red o registros. A continuación se presenta una introducción a los más importantes, haciendo foco en la utilidad desde la perspectiva de un sistema de seguridad.

2.1.1. Notificaciones de Procesos

Las notificaciones de procesos, expuestas a través de la API `PsSetCreateProcessNotifyRoutine` (y su versión extendida `PsSetCreateProcessNotifyRoutineEx`), permiten a los desarrolladores del kernel registrar funciones que serán llamadas automáticamente cada vez que se cree o termine un proceso. Estas funciones son conocidas como callbacks, de ahí el nombre.

Su propósito principal no es la seguridad, sino proporcionar a los desarrolladores un mecanismo que permita tomar acciones cuando se crea, se destruye o se modifica un proceso en el sistema. Por ejemplo, una solución de control parental podría usar estas notificaciones para bloquear automáticamente cualquier intento de lanzar un navegador web fuera del horario permi-

tido, o un software de auditoría podría registrar cada ejecución de `cmd.exe` o `powershell.exe`.

Desde el punto de vista de un sistema EDR, este mecanismo se convierte en una de las **primeras líneas de observación**. Permite detectar en tiempo real la aparición de nuevos procesos o la terminación de los existentes, lo que es especialmente útil para construir un **árbol de procesos en memoria** y trazar la relación entre ellos. Estas notificaciones proporcionan datos muy interesantes como:

- El PID del proceso involucrado
- El PID del proceso padre (PPID)
- Un indicador de si se trata de creación o terminación
- Una cadena de texto con el comando lanzado
- La información de contexto de seguridad (token de acceso)

Esta información es especialmente útil cuando se busca detectar comportamientos anómalos, como la ejecución de `rundll32.exe` o `powershell.exe` desde un proceso que normalmente no debería lanzarlos, como por ejemplo, `winword.exe`, `notepad.exe` o `explorer.exe`. Este tipo de relación inusual entre procesos podría ser una señal de actividad maliciosa como la explotación de una macro o la ejecución remota de código.

Aunque este mecanismo proporciona información útil como la ruta del binario ejecutado y los argumentos de línea de comandos, no ofrece visibilidad sobre procesos ya en ejecución al momento de inicializar el driver. Además, al tratarse de una interfaz conocida y documentada, suele convertirse en un **objetivo frecuente para técnicas de evasión**. Aunque es una fuente de información muy útil, debe complementarse con otros mecanismos para ofrecer una cobertura realista y robusta.

2.1.2. Event Tracing for Windows

Event Tracing for Windows (ETW) es una infraestructura integrada en Windows pensada para ofrecer a los desarrolladores y administradores una forma eficiente de recolectar grandes volúmenes de eventos sin afectar al rendimiento del sistema, incluso bajo alta carga. Esto es especialmente útil para realizar análisis de rendimiento, depuración o registrar grandes cantidades de eventos con detalle.

Los componentes que generan eventos en ETW se conocen como **proveedor** (*provider*), que puede ser el propio sistema operativo, un controlador o incluso una aplicación hecha por ti. Estos proveedores emiten eventos cuando ocurre cierta actividad interna —como la creación de un proceso, la carga de una DLL o una conexión de red—. Los eventos que producen pueden ser recogidos por **consumidores** (*consumers*), que son las aplicaciones que procesan esta información. Herramientas como el Monitor de Recursos, Windows Performance Recorder o incluso PowerShell pueden actuar como consumidores, al igual que lo hacen las soluciones EDR.

Desde el punto de vista de un EDR, ETW es una **fuente de telemetría extremadamente rica y versátil**. Ofrece visibilidad sobre aspectos clave del sistema, incluyendo:

- Creación y terminación de procesos e hilos
- Carga de módulos en memoria (por ejemplo, DLLs)
- Actividad de red, incluyendo conexiones establecidas

- Operaciones de archivo (lectura, escritura, eliminación)
- Eventos de seguridad como inicios de sesión o cambios en permisos

Una de sus principales ventajas es que funciona de forma **asíncrona y con bajo impacto**, utilizando buffers en memoria para almacenar los eventos antes de procesarlos o volcarlos. Esto permite capturar grandes volúmenes de telemetría sin ralentizar el sistema, algo que es primordial para tareas en segundo plano como puede ser une EDR. Además, es adaptable, es decir, tú decides a que proveedores te suscribes y a cuales no, lo que permite ajustar el nivel de detalle que quieres recolectar en función del caso de uso o de los recursos disponibles.

Un ejemplo concreto es el proveedor `Microsoft-Windows-Kernel-Process`, que emite eventos relacionados con la creación y finalización de procesos a nivel del núcleo del sistema. Al suscribirse a este provider, un EDR puede obtener datos como el nombre del ejecutable, el identificador del proceso (PID), su proceso padre (PPID), la ruta de imagen completa y el tiempo de creación. Esta información puede servir de complemento o via alternativa a los PsCallback discutidos en sección anterior.

Sin embargo, ETW no está libre de limitaciones. Aunque es difícil de deshabilitar completamente sin romper funcionalidades básicas del sistema, **algunos proveedores específicos sí pueden ser desactivados o filtrados**. También es posible que un atacante intente saturar los buffers o interferir con el flujo de eventos para dificultar su análisis. Por ello, aunque ETW es una pieza clave dentro de un EDR, debe considerarse como parte de un conjunto más amplio de sensores y no como único punto de observación.

2.1.3. Drivers minifilter del Sistema de Archivos

Los **minifilters** son una clase especial de drivers en modo kernel que permiten interceptar y modificar las operaciones que se realizan sobre el sistema de archivos en Windows. Están diseñados principalmente como una herramienta para aquellos desarrolladores que necesitan observar o intervenir en el acceso a archivos y directorios sin tener que implementar directamente un controlador de sistema de archivos completo.

Su funcionamiento se apoya en el *Filter Manager*, un componente del kernel que actúa como intermediario entre los filtros instalados y el sistema de archivos. Esta arquitectura permite que múltiples minifilters puedan coexistir en el sistema, cada uno con su propia lógica: herramientas de backup, cifrado, compresión, o soluciones de seguridad como los EDR.

Cada Minifilter se registra con una determinada **altitud**, que representa su posición relativa dentro de la cadena de filtros. Las operaciones del sistema de archivos se procesan siguiendo ese orden de altitudes, pasando de un filtro a otro según su nivel (top to bottom). Esto permite que, por ejemplo, un Minifilter de cifrado actúe antes que uno de backup, o que un EDR reciba la operación después de que otros filtros ya la hayan inspeccionado o modificado. La correcta asignación de altitudes es fundamental para evitar conflictos y garantizar que cada filtro actúe en el momento adecuado dentro del flujo de ejecución. En el cuadro 2.1 pueden verse los distintos grupos de altitud disponibles asi como su rango.

Un driver Minifilter puede registrar **funciones de callback** para recibir notificaciones antes o después de que se ejecute una operación de entrada/salida (I/O). Estas operaciones incluyen,

Cuadro 2.1: *Grupos de carga de Minifiltros en Microsoft Windows*

Rango de Altitud	Nombre del Grupo	Rol del Minifiltro
420000–429999	Filter	Drivers heredados
400000–409999	FSFilter Top	Deben ir por encima de todos
360000–389999	FSFilter Activity Monitor	Monitoreo de I/O de archivos
340000–349999	FSFilter Undelete	Recuperación de archivos eliminados
320000–329999	FSFilter Anti-Virus	Antimalware
300000–309999	FSFilter Replication	Copia de datos a sistemas remotos
280000–289999	FSFilter Continuous Backup	Copia continua a medios de respaldo
260000–269999	FSFilter Content Screener	Prevención de creación de ciertos archivos
240000–249999	FSFilter Quota Management	Cuotas avanzadas de almacenamiento
220000–229999	FSFilter System Recovery	Integridad del sistema
200000–209999	FSFilter Cluster File System	Metadatos distribuidos en red
180000–189999	FSFilter HSM	Manejo jerárquico de almacenamiento
160000–169999	FSFilter Compression	Compresión de datos de archivos
140000–149999	FSFilter Encryption	Cifrado y descifrado de archivos
120000–129999	FSFilter Physical Quota Management	Cuotas físicas por bloques
100000–109999	FSFilter Open File	Snapshots de archivos abiertos
80000–89999	FSFilter Security Enhancer	Lockdowns y control de acceso mejorado
60000–69999	FSFilter Copy Protection	Verificación de datos fuera de banda
40000–49999	FSFilter Bottom	Se cargan debajo de todos
20000–29999	FSFilter System	Reservado
<20000	FSFilter Infrastructure	Uso del sistema, más cercano al filesystem

entre otras:

- Creación o apertura de archivos y carpetas
- Lectura y escritura de datos
- Renombrado o eliminación de archivos
- Cambios en atributos o permisos

Desde el punto de vista de un EDR, esta capacidad permite obtener **visibilidad directa y a bajo nivel** sobre las acciones que afectan al sistema de archivos. Por ejemplo, se puede detectar cuándo un ejecutable es creado en una carpeta sospechosa, como el directorio temporal de un usuario, o cuándo un proceso intenta eliminar archivos de log en, por ejemplo, `C:\Windows\System32\LogFiles`. Este tipo de actividad puede ser una señal de intento de evasión o limpieza de huellas.

Además, los minifilters permiten aplicar lógica condicional a estas operaciones. Un EDR puede bloquear ciertos accesos, registrar eventos para análisis posterior o marcar archivos como sospechosos si coinciden con ciertos criterios (como nombres, extensiones o ubicación en disco).

Al operar en un nivel tan cercano al núcleo del sistema, los minifilters ofrecen una visión detallada y son fácilmente programables gracias a la interfaz que ofrece Windows. No obstante, igual que con los anterores apartados, también presentan algunos retos. Por un lado, requieren de especial atención para evitar lo que se conocen como *race conditions* o conflictos con otros filtros instalados. Por otro, tampoco se libra de que **malware con privilegios elevados pueda intentar deshabilitar o insertar su propio Minifilter** a una altitud superior para interferir con los datos que le llegan a este. Por eso, siempre es mejor recibir la información de varias fuentes.

2.1.4. Windows Filtering Platform (WPF)

La **WFP** es una arquitectura nativa de Windows diseñada para ofrecer un punto de control detallado sobre el tráfico de red, tanto a nivel de kernel como en modo usuario. Su objetivo principal es proporcionar a los desarrolladores una forma estructurada de inspeccionar, modificar o bloquear paquetes de red en distintos puntos de la pila de red. Esta plataforma es la base de muchos componentes de red de Windows, incluidos el Firewall de Windows y las políticas de IPSec.

WFP se organiza en una serie de capas de filtrado —conocidas como *filtering layers*— que representan diferentes momentos del procesamiento de un paquete: desde que entra o sale por una interfaz, hasta que se entrega a una aplicación o se asocia a un socket. Los desarrolladores pueden registrar filtros en una o varias de estas capas, permitiendo aplicar reglas específicas o incluso funciones personalizadas llamadas **callouts**, que son invocadas automáticamente cuando se detecta tráfico relevante. El cuadro 2.2 ofrece una vision de las distintas capas que ofrece WPF.

Cuadro 2.2: *Capas de Filtro en Windows Filtering Platform (WFP)*

Nombre de la Capa (Layer)	Dirección del Tráfico	Rol del Filtro
FWPM_LAYER_INBOUND_IPPACKET_V4	Entrante	Inspección de paquetes IPv4 antes del reensamblado
FWPM_LAYER_OUTBOUND_IPPACKET_V4	Saliente	Modificación de paquetes IPv4 justo antes de salir
FWPM_LAYER_INBOUND_TRANSPORT_V4	Entrante	Filtros aplicados después del enrutamiento, antes de la entrega a la aplicación
FWPM_LAYER_OUTBOUND_TRANSPORT_V4	Saliente	Filtros antes de aplicar reglas de enrutamiento y enviar
FWPM_LAYER_ALE_AUTH_CONNECT_V4	Saliente	Autorización de conexiones salientes a nivel de aplicación (ALE)
FWPM_LAYER_ALE_AUTH_RECV_ACCEPT_V4	Entrante	Autorización de conexiones entrantes antes de llegar a la app
FWPM_LAYER_STREAM_V4	Bidireccional	Inspección de tráfico TCP en el flujo de datos (stream)
FWPM_LAYER_DATAGRAM_RECEIVE_V4	Entrante	Procesamiento de datagramas UDP recibidos
FWPM_LAYER_ALE_FLOW_ESTABLISHED_V4	Bidireccional	Gestión de flujo una vez establecida la conexión (tracking)
FWPM_LAYER_RESOURCE_ASSIGNMENT_V4	Saliente	Asignación de recursos de red, como interfaces o rutas

Desde el punto de vista de un EDR, WFP es una herramienta muy potente para obtener **visibilidad en tiempo real del comportamiento en red de los procesos**. Algunas de sus aplicaciones prácticas incluyen:

- Detectar la apertura de nuevas conexiones TCP o UDP desde un proceso determinado.
- Asociar el tráfico con el proceso que lo genera, incluyendo PID, nombre del ejecutable y ruta de imagen.
- Identificar patrones de comunicación anómalos, como intentos de conectar con dominios

poco comunes o direcciones IP internas no autorizadas.
- Implementar respuestas automáticas, como bloquear conexiones salientes desde un proceso sospechoso o generar alertas si se detecta exfiltración de datos.

Por ejemplo, un EDR podría registrarse a la capa `FWPM_LAYER_ALE_FLOW_ESTABLISHED_V4` para recibir notificaciones cada vez que se establece una conexión. Esto permitiría detectar si un proceso como `winword.exe`, habitualmente no asociado a actividad de red, intenta establecer una conexión saliente tras abrir un documento, lo que podría indicar actividad maliciosa relacionada con macros o explotación de vulnerabilidades.

Gracias a su integración nativa con la pila de red, WFP permite aplicar políticas de control de manera sencilla y precisa. Sin embargo, incorporar malos filtros puede causar interferencias con otras aplicaciones, y su mal uso puede afectar al rendimiento de red o generar falsas alarmas. Además, como otros mecanismos de seguridad en modo kernel, **puede ser un objetivo para técnicas de evasión**, especialmente por malware que intente inyectarse en procesos nativos de Windows para aprovecharse de reglas permisivas o *whitelists*.

Aun con estas limitaciones, WFP sigue siendo uno de los mecanismos más potentes y flexibles para incorporar capacidades de inspección de red dentro de un sistema EDR en Windows.

2.2. Otros Mecanismos Utilizados por los EDR

Además de los mecanismos nativos ofrecidos por Windows —como ETW, Callbacks o WFP—, existen otras técnicas que permiten recolectar telemetría que con los mecanismos oficiales no seríamos capaces.

Aunque utilizarlos tiene sus riesgos como posibles conflictos con el software ya instalado o el propio sistema operativo, estas soluciones son especialmente útiles para investigar procesos sospechosos en más detalle. Uno de los enfoques más comunes es el **Hooking de funciones en APIs del sistema**, pero existen otras estrategias que también han demostrado ser efectivas para detectar comportamientos anómalos.

2.2.1. Hooking de Funciones mediante DLLs

El **Hooking de funciones** es una técnica que permite interceptar llamadas a funciones del sistema operativo (también llamada Windows API) y desviar temporalmente su ejecución hacia una rutina personalizada (callback). Esta rutina (el *hook* o *Detour*) puede registrar información, modificar parámetros o incluso bloquear la operación antes de devolver el control a la función original. Aunque no es una técnica diseñada específicamente para propósitos de seguridad, se ha convertido en una herramienta habitual en soluciones EDR por su capacidad para observar, desde dentro del proceso, operaciones sensibles del sistema.

A diferencia de otros mecanismos como ETW o PsCallbacks, el Hooking permite acceder e interventir directamente en el momento exacto en el que una función es invocada, así como a los parámetros con los que fue llamada. Esto lo hace especialmente útil para monitorizar (y bloquear) funciones críticas de la API de Windows que están fuertemente asociadas a técnicas de malware, como la manipulación remota de memoria o la creación de procesos en contextos

inusuales. Sin Hooking, acceder a esa información en tiempo real sería extremadamente complejo o directamente inviable.

Una forma habitual de implementar esta técnica es mediante la **inyección de DLLs** en el espacio de direcciones del proceso objetivo. En el caso de los EDR, esta inyección puede realizarse desde el kernel tras recibir una notificación de creación de proceso mediante `PsCallbacks`. Utilizando un mecanismo conocido como **Kernel Asynchronous Procedure Calls (KAPCs)**, el sistema puede inyectar código sobre la memoria de nuevo proceso antes de que comience su ejecución. Una vez cargada la DLL, esta puede instalar los hooks necesarios sobre funciones del sistema que se quieran interceptar.

El Hooking no se limita a observar el flujo de ejecución; permite a las herramientas de seguridad **insertarse activamente en la lógica del proceso**. Con ello, pueden:

- Registrar cada vez que una función crítica es utilizada, junto con sus argumentos.
- Detectar patrones de uso anómalos que sugieran actividad maliciosa.
- Impedir directamente que ciertas operaciones se lleven a cabo, actuando como una capa de control en tiempo de ejecución.

Gracias a esta capacidad de inspección tan cercana al comportamiento real del proceso, el Hooking se ha convertido en un recurso indispensable para muchos EDR. Sin embargo, su uso también introduce complejidad y riesgos, especialmente frente a técnicas de evasión diseñadas para eliminar o evitar estos hooks durante la ejecución.

Al ser una modificación directa del flujo de ejecución, el Hooking puede ser detectado y eliminado por malware avanzado. Técnicas de **unhooking**, como la restauración de funciones desde copias limpias de ntdll.dll o la reescritura de la tabla de importación (IAT), permiten deshacer los hooks y evadir la detección. Por eso, muchos EDR modernos lo usan como complemento, apoyándose también en fuentes de telemetría más difíciles de manipular, como ETW o el monitoreo a nivel de kernel.

2.2.2. Análisis de Procesos mediante Sandboxing

Algunos EDR modernos integran mecanismos de **sandboxing de procesos** como parte de sus técnicas avanzadas de análisis. El objetivo no es solo monitorizar en tiempo real lo que hace un proceso, sino aislarlo y observar su comportamiento en un entorno controlado antes de permitir su ejecución completa en el sistema real. Esta técnica es útil para estudiar procesos sospechosos pero que aún no han ejecutado acciones maliciosas claras.

El enfoque habitual consiste en interceptar la creación de un nuevo proceso y, en lugar de dejarlo correr directamente en el entorno del usuario, **redirigir su ejecución hacia una instancia aislada o monitorizada intensivamente**. Esto puede lograrse utilizando máquinas virtuales internas, contenedores, o mediante técnicas de virtualización ligera que replican parte del entorno del proceso (por ejemplo, sistema de archivos, registros, claves de configuración, etc.) sin exponer recursos reales. Durante este análisis, el EDR puede observar aspectos como:

- Qué archivos intenta modificar o leer el proceso.
- A qué procesos o servicios intenta conectarse.
- Qué módulos carga en memoria y qué funciones ejecuta.

- Si intenta modificar el registro o persistirse en el sistema.

Este tipo de sandboxing no es el mismo que el que ofrecen productos específicos de análisis dinámico como Cuckoo o Joe Sandbox, sino que está integrado directamente dentro del flujo normal del sistema de protección. La ventaja es que permite detectar comportamientos maliciosos basados en ejecución real, no solo en indicadores estáticos o reglas heurísticas.

Incorporar esta capacidad implica una **complejidad considerable**. Requiere replicar parcialmente el entorno de ejecución, mantener compatibilidad con procesos legítimos y asegurar que la *sandbox* no altere el comportamiento del binario. Además, supone una penalización importante en tiempo y recursos, por lo que suele con mucha moderación.

Por estas razones, este proyecto no incluye mecanismos de sandboxing. El enfoque aquí es construir un sistema básico de detección basado en telemetría en tiempo real, sin replicar entornos ni ejecutar procesos en aislamiento. Aunque el análisis dinámico ofrece capacidades muy potentes, su implementación requiere una infraestructura y una lógica mucho más compleja, que escapa al alcance técnico de esta fase del proyecto.

3

Arquitectura y Diseño del Sistema

Este capítulo describe la arquitectura general del sistema EDR desarrollado en el proyecto. Se detalla cómo se estructuran los distintos componentes, qué responsabilidades asume cada uno y cómo se comunican entre sí.

El objetivo es mostrar y justificar las decisiones tomadas durante el diseño: por qué se ha elegido esta arquitectura en concreto, cómo se gestionan los flujos de datos y qué criterios se han seguido para construir un sistema eficiente, extensible y lo más estable posible dentro de las limitaciones del entorno.

El capítulo comienza presentando los **principios de diseño** que han guiado la construcción del sistema, seguidos de una **visión general de su arquitectura** y una descripción de los principales **componentes funcionales**, tanto en el Espacio de usuario como en el kernel. A continuación, se detalla el **modelo de comunicaciones**, que explica cómo fluye la información entre módulos y qué canales se utilizan en cada caso.

Posteriormente, se describe el **modelo de datos**, donde se analiza cómo se representan, procesan y almacenan los eventos generados por cada Sensor. Por último, se introduce el **modelo de métricas**, encargado de monitorizar el rendimiento y el consumo del sistema.

3.1. Principios de Diseño

El diseño de este sistema EDR parte de la idea de tener una arquitectura sencilla, flexible y completamente modificable, que permita entender en detalle cómo se comportan los distintos componentes en distintas situaciones. El objetivo es crear una base experimental sólida sobre la que se puedan probar ideas, extender funcionalidades y observar el impacto real de cada decisión.

Una de las claves del diseño ha sido mantener una **separación clara entre el Espacio de usuario y el Espacio de kernel**. Todo lo que puede hacerse de forma segura y eficiente en modo usuario se deja en esa capa, mientras que el kernel se reserva para operaciones que requieren mayor

visibilidad o privilegios. Esto no solo simplifica el desarrollo y la depuración, sino que también reduce el riesgo de errores fatales en el sistema.

El proyecto se ha desarrollado íntegramente en Rust, aprovechando su ecosistema de librerías para agilizar el desarrollo de funcionalidades en modo usuario y proporcionar una capa de seguridad extra en el kernel, donde prima la estabilidad y el control total sobre cada detalle. Ademas, el hecho de programar todos los sensores desde cero permite ajustar los sensores al detalle, medir su impacto en tiempo real y estudiar cómo afectan tanto a la detección como al rendimiento general del sistema.

Otro principio importante ha sido la **Modularidad**. Cada componente del sistema parte de una interfaz de programación común que luego se va complementando con las particularidades de cada uno. Por ejemplo, todos los eventos deben tener un identificador y una prioridad, pero no todos tendrán una IP o harán referencia a un archivo. Esta estructura facilita la incorporación de nuevas fuentes de telemetría o experimentos sin romper el resto del entorno. El mismo principio se aplica a las reglas de detección y a los mecanismos de respuesta, que se pueden añadir, ajustar o desactivar de forma independiente.

También se ha hecho especial hincapié en la **robustez y corrección del código**, especialmente en el desarrollo de componentes en modo kernel. Se siguen buenas prácticas de programación, se validan cuidadosamente los datos que cruzan la frontera kernel-user y se prioriza siempre la estabilidad del sistema operativo por encima de la cantidad de eventos capturados.

Por último, se ha optado por un modelo de comunicación híbrido tipo *push-pull*, donde los sensores generan eventos y los colocan en cola, pero es el agente quien decide cuándo consumirlos, permitiendo aplicar control de flujo y evitar sobrecargas. En lo que respecta al almacenamiento, los eventos se insertan en lotes en una base de datos SQLite en modo *WAL*, optimizada para escritura rápida y con índices orientados a las consultas más frecuentes.

En conjunto, estos principios buscan construir una arquitectura que no solo funcione, sino que sea una herramienta activa para aprender, experimentar y mejorar continuamente la lógica de detección de amenazas.

3.2. Visión General del Sistema

El sistema se compone de dos grandes bloques funcionales: el agente en Espacio de usuario, el driver en Espacio de kernel. Cada uno de estos componentes cumple un rol bien definido, asi como las interacciones entre ellos.

El núcleo del sistema es el **Userland Agent**, un servicio de Windows escrito en Rust que se encarga de coordinar la recepción, procesamiento y almacenamiento de eventos provenientes de distintos sensores. También es el encargado de escanear el sistema de archivos y configurar el resto de sensores. En paralelo, el **Driver del Kernel** implementa la lógica de captura de telemetría a bajo nivel utilizando mecanismos como callbacks e interfaces Input/Output Control (IOCTL).

La figura 3.1 muestra la arquitectura general del sistema y las interacciones entre sus compo-

nentes principales.

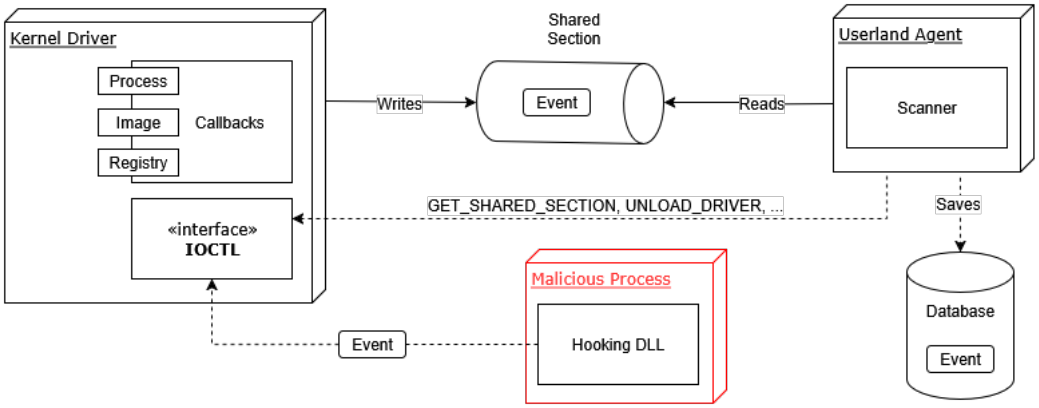

Figura 3.1: *Arquitectura del EDR*

En el siguiente apartado, se describen brevemente los componentes clave del sistema, sin entrar aún en detalles de implementación.

3.2.1. Userland Agent

El **Userland Agent** es el componente central en Espacio de usuario. Su función principal es actuar como puente entre los sensores del kernel y los módulos encargados de procesar, almacenar y presentar la información al usuario. Toda la telemetría capturada se canaliza a través de este servicio para su análisis y guardado.

Durante la fase de diseño también me apoyé en proyectos de referencia como *Sanctum*, desarrollado por 0xflux 0xflux, 2023b, que me sirvió como inspiración para estructurar un EDR de forma modular y extensible. El desarrollo del agente en Rust se apoyó constantemente en la documentación oficial del lenguaje The Rust Project Developers, 2025,

Internamente, el agente se estructura en varios módulos independientes:

- Un módulo de ingesta que recopila eventos desde el kernel.
- Un escáner periódico del sistema de archivos, basado en reglas YARA.
- Un módulo para almacenar telemetría en una base de datos SQLite.

El diseño modular del agente permite activar, desactivar o configurar fácilmente cualquiera de estos módulos sin tener que volver a cargar o instalar todos ellos. Algunos de ellos son indispensables para su funcionamiento como la ingesta de eventos a través de la sección compartida o su guardado en la base de datos.

3.2.2. Driver del Kernel

El **Driver del Kernel** es responsable de observar la actividad del sistema operativo desde el nivel más bajo posible. Aunque se entrega como un único binario `.sys`, internamente agrupa varios sensores independiente. La integración de Rust en el desarrollo de drivers se apoyó en

las librerías oficiales de Microsoft, *windows-drivers-rs* **microsoft2025windowsdriversrs**, que facilitan el trabajo con estructuras internas del kernel desde un entorno más seguro.

Entre sus funciones principales destacan:

- Observar operaciones del sistema de archivos mediante los *callbacks* instalados.
- Cargar eventos en la sección compartida con el agente.
- Recibir y procesar eventos generados por la DLL, encargada de recoger telemetria a través de *Hooking* de las API de Windows.

Para entender mejor la complejidad de escribir drivers en Windows consulté la guía de OSR Online, 2020, que explica de forma clara los fundamentos y las consideraciones de estabilidad en este entorno. Los sensores están diseñados para trabajar de forma asíncrona y con el menor impacto posible en el rendimiento del sistema.

En la siguiente sección se introducirán las diferentes interfaces de comunicación que tiene cada componente, explicando en que componente se encuentran y para que sive cada una de ellas.

3.3. Modelo de Comunicaciones

El EDR utiliza una arquitectura modular donde los distintos componentes intercambian información a través de canales definidos según su función. Estos canales se han elegido en función del tipo de dato transmitido y la relación entre los módulos, buscando siempre un equilibrio entre rendimiento, eficiencia y simplicidad.

Cuadro 3.1: *Comunicaciones entre componentes*

Origen	Destino	Canal	Propósito
Kernel Driver	Userland Agent	Sección compartida	Eventos de *callbacks* y *hooking*
Userland Agent	Kernel Driver	IOCTL	Gestión del driver
Hooking DLL	Kernel Driver	IOCTL	Eventos del proceso
Userland Agent	SQLite	`SQL + WAL`	Inserciones recurrentes

El driver en modo kernel comunica los eventos capturados al agente en espacio de usuario utilizando una sección de memoria compartida. Este canal permite transmitir de forma eficiente información relacionada con callbacks y hooks sin necesidad de llamadas síncronas costosas.

Para operaciones de control y gestión, el agente y otros módulos como la DLL de hooking se comunican con el driver mediante llamadas IOCTL. Esto permite realizar tareas administrativas o reportar eventos desde componentes en modo usuario hacia el núcleo.

Por otro lado, la persistencia de eventos se lleva a cabo en una base de datos SQLite configurada en modo `Write-Ahead Logging (WAL)`, lo que permite realizar inserciones frecuentes sin bloquear las consultas de lectura, incluso bajo carga.

En un futuro se pueden aprovechar las interficies o canales ya creados o se pueden generar otros nuevos.

3.4. Modelo de Datos y Almacenamiento

Los eventos generados por el sistema siguen una estructura común: todos incluyen campos básicos como la marca de tiempo, y algo que los identifica, como el PID o la ruta del ejecutable. A partir de ahí, se añaden más datos según el tipo de evento. Este formato uniforme permite que el agente los procese de forma sencilla, aplique reglas y los almacene sin complicaciones.

Cuando llegan al agente del espacio de usuario, los eventos pueden enriquecerse con más información y luego se guardan en una base de datos SQLite. Ésta está configurada en modo `Write-Ahead Logging (WAL)`, lo que permite hacer muchas inserciones sin bloquear las lecturas. Así se asegura un buen rendimiento incluso cuando hay mucho tráfico de eventos. Además, este modo permite que se sigan insertando eventos incluso mientras se están accediendo a los datos.

En el esquema actual (ver figura 3.2) cada tipo de evento tiene su propia tabla, con índices que facilitan búsquedas por campos como el timestamp, PID o ruta. En este esquema se incluyen tablas para eventos de hooking (hook_event), procesos (process_event), carga de imágenes (image_load_event), modificaciones del registro (registry_event) y escaneo de archivos (file_scanner). La tabla hook_event actúa como base para registrar llamadas específicas del sistema, como NtCreateThreadEx, NtMapViewOfSection, NtProtectVirtualMemory o NtSetValueKey, cada una almacenada en su propia subtabla con los campos necesarios. Esta estructura facilita una separación lógica entre eventos genéricos y detalles específicos, manteniendo el diseño modular y fácil de consultar.

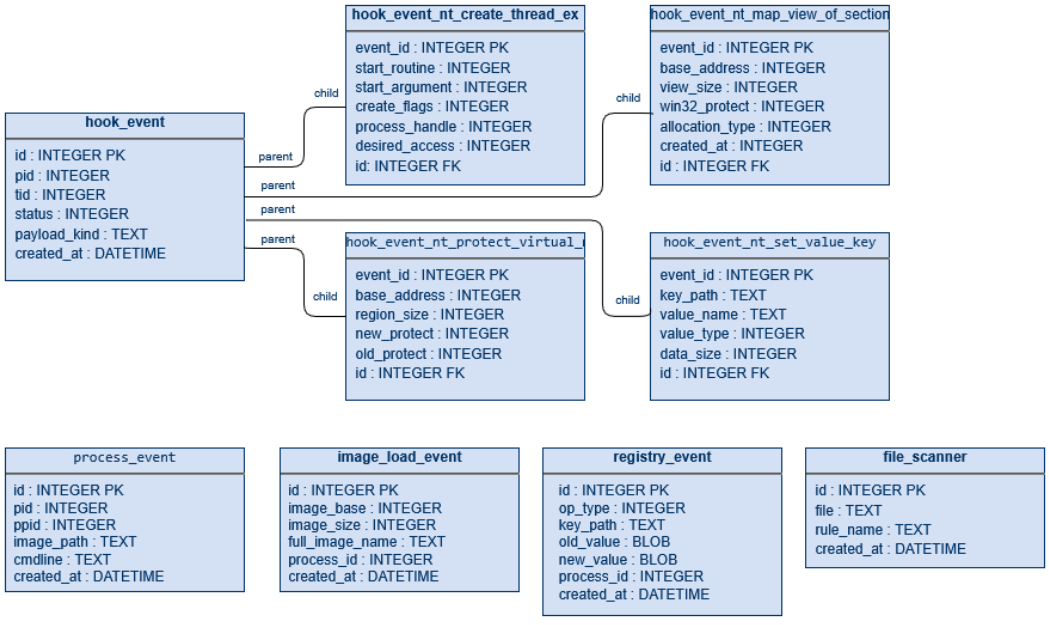

Figura 3.2: *Esquema de Base de Datos*

Con todo esto, ya tenemos una visión clara de cómo está construido el sistema por dentro: desde los principios que guiaron su diseño hasta la forma en que los componentes se comunican, almacenan y representan la información. A partir de aquí, toca ver cómo se lleva todo esto a la práctica. En el siguiente capítulo se entra ya en la implementación real del sistema: cómo se ha

construido cada módulo, cómo se integran entre sí y qué decisiones se han tomado para que todo esto funcione como se espera.

4
Implementación del Sistema

Después de haber explorado la arquitectura y el diseño general del sistema, en este capítulo entramos ya en la parte más práctica del proyecto: la implementación real. Aquí se detalla cómo se ha construido cada uno de los componentes, cómo se organizan internamente y qué herramientas y técnicas se han utilizado para dar forma a todo el sistema.

Este bloque concentra el grueso del trabajo. Es aquí donde las ideas planteadas anteriormente se traducen en código funcional. El objetivo no es solo mostrar el resultado, sino explicar cómo se ha llegado hasta ahí, qué decisiones técnicas se han tomado por el camino y cómo se ha buscado siempre mantener un buen equilibrio entre claridad, eficiencia y robustez.

El capítulo comienza describiendo el modelo de eventos y el sistema de serialización, que permite representar y transportar la información de forma coherente entre módulos. Después, se analiza cómo está organizado el código del proyecto, explicando la estructura de los paquetes y la lógica de cada uno. A partir de ahí, se detallan distintas partes clave de la implementación: el uso de memoria compartida, la integración con la base de datos, el escáner de archivos y el sistema de métricas encargado de medir el comportamiento general del sistema. Ahora si, vamos con la primera sección.

4.1. Modelo de eventos y sistema de serialización

Una de las partes más importantes del sistema es su capacidad para recopilar información desde múltiples fuentes: hooks en funciones del sistema, escaneos periódicos o eventos generados directamente desde el kernel. Cada una de estas fuentes produce eventos distintos, con su propia estructura y nivel de detalle. Para poder procesarlos todos de forma coherente y unificada, se ha diseñado un formato común que permita representarlos bajo un mismo esquema.

Con esta idea en mente, se ha creado un modelo de eventos unificado que encapsula cualquier tipo de suceso relevante en una estructura común. Esto facilita mucho su análisis y almacenamiento, y además permite que todos los módulos del sistema trabajen sobre una base compartida, sin preocuparse por los detalles internos de cada evento.

Para implementarlo, se ha optado por usar Protocol Buffers (Protobuf) como sistema de seriali-

zación. Cada tipo de evento —por ejemplo, la creación de un proceso o una llamada interceptada— tiene su propia estructura, pero todos terminan encapsulados en un mensaje principal llamado Event, que actúa como contenedor genérico. Gracias a este diseño, los eventos se pueden transmitir y almacenar de forma compacta, sin perder compatibilidad entre versiones y con la flexibilidad suficiente para seguir ampliando el sistema en el futuro.

En esta sección se explica cómo está organizado este modelo: cómo se estructura el código, cómo se definen los distintos eventos en Protobuf y cómo se gestionan los procesos de serialización y deserialización. También se verá cómo añadir nuevos tipos de eventos sin romper la compatibilidad ni modificar el resto del sistema.

4.1.1. Arquitectura y estructura del código fuente

El sistema de eventos se ha organizado dentro de la carpeta `shared`, concretamente en el módulo `events`, que centraliza todo lo relacionado con la representación y manipulación de eventos. Aquí se definen tanto las estructuras de datos como las utilidades necesarias para construir, convertir y serializar eventos de forma coherente en todo el proyecto.

El archivo principal, `mod.rs`, actúa como punto de entrada del módulo. Desde ahí se exponen los tipos más importantes —como `Event` y `EventKind`— y se han implementado funciones auxiliares que permiten generar eventos a partir de estructuras específicas sin preocuparse por los detalles internos. Para facilitar aún más esta tarea, se han añadido conversiones automáticas mediante la implementación de traits (`From<T>for Event`), lo que permite a cada sensor crear eventos sin necesidad de lógica adicional.

El módulo se ha dividido en varios subarchivos, cada uno agrupando los tipos de evento según su origen o función:

- `scanner.rs`: detecciones del escáner YARA.
- `hook.rs`: llamadas interceptadas en espacio de usuario.
- `callbacks.rs`: eventos generados desde el kernel (procesos, imágenes, registro).
- `event.rs`: definición general de `Event` y su variante `EventKind`, generadas a partir del esquema Protobuf.

Cada uno de estos submódulos está asociado a un archivo `.proto` específico, y durante la compilación se ha utilizado `prost-build` para generar automáticamente el código Rust correspondiente. Esto garantiza que todos los eventos compartan el mismo formato binario y puedan ser procesados de forma consistente.

4.1.2. Estructura del evento y definición del esquema

La pieza central del modelo es el mensaje `Event`, definido en el archivo `event.proto`. Este mensaje actúa como contenedor común para todos los eventos del sistema, independientemente de su origen. Su campo principal, `kind`, utiliza una construcción `oneof` que permite incluir un único tipo de evento entre varias opciones posibles.

Cada tipo concreto de evento (como `HookEvent`, `FileScannerEvent`, `ProcessEvent`, etc.) se ha definido por separado en su propio archivo `.proto`, y encapsula únicamente la información

Listado 1: *Esquema Protobuf simplificado del mensaje* `Event`

```
message Event {
  oneof kind {
    hook.HookEvent            hook          = 1;
    scanner.FileScannerEvent  scanner       = 2;
    callbacks.ProcessEvent    process_event = 3;
    callbacks.ImageLoadEvent  image_load    = 4;
    callbacks.RegistryEvent   registry      = 5;
  }
}
```

relevante para ese caso. Por ejemplo, un `HookEvent` puede contener los argumentos de una llamada a `NtCreateThreadEx`, mientras que un `ProcessEvent` recoge detalles como el PID, PPID, ruta del ejecutable y línea de comandos.

Este diseño permite extender el sistema fácilmente: para añadir un nuevo tipo de evento, basta con definir su mensaje en Protobuf y añadirlo como una nueva variante en el campo `oneof`.

Durante el proceso de compilación, todos los esquemas `.proto` se procesan mediante `prost-build`, generando automáticamente el código Rust correspondiente. El tipo resultante, también llamado `Event`, incluye los campos del esquema y una variante interna `event::Kind` que permite acceder directamente al tipo concreto de evento en tiempo de ejecución. El evento compilado se puede ver en el listado .

4.1.3. Subtipos de eventos y estructura interna

Cada tipo de evento incluido en el campo `kind` del mensaje `Event` representa una categoría concreta de actividad dentro del sistema. Estos subtipos se han definido por separado en sus propios archivos `.proto`, agrupados por su origen funcional: hooks de API, escáner YARA, callbacks del sistema, etc.

Uno de los subtipos más representativos es `HookEvent`, que describe llamadas realizadas a funciones sensibles de la API de Windows, interceptadas en tiempo de ejecución mediante técnicas de *hooking*. Estas funciones suelen estar relacionadas con operaciones potencialmente maliciosas, como la creación remota de hilos, la modificación de permisos de memoria o la escritura en el registro. Entre las funciones cubiertas se encuentran `NtCreateThreadEx`, `NtMapViewOfSection` o `NtSetValueKey`.

Este tipo de evento incluye información básica sobre el proceso que realizó la llamada (PID, TID, código de retorno), junto con un payload interno definido como un `oneof` que identifica la API concreta que ha sido invocada. Esto se puede ver en el listado 3.

Cada una de las variantes de `HookPayload` contiene campos específicos para su contexto: direcciones de memoria, tamaños, flags de protección o argumentos concretos de la API interceptada.

Otros subtipos del sistema son mucho más simples. Por ejemplo, `FileScannerEvent` contiene únicamente la ruta del archivo escaneado y el nombre de la regla YARA que generó una coin-

Listado 2: *Estructura del mensaje* `Event`

```rust
// This file is @generated by prost-build.
/// Unified event wrapper for all event types.
#[derive(Clone, PartialEq, ::prost::Message)]
pub struct Event {
    #[prost(oneof = "event::Kind", tags = "1, 2, 3, 4, 5")]
    pub kind: ::core::option::Option<event::Kind>,
}
/// Nested message and enum types in `Event`.
pub mod event {
    #[derive(Clone, PartialEq, ::prost::Oneof)]
    pub enum Kind {
        /// Events from hook module
        #[prost(message, tag = "1")]
        Hook(super::super::hook::HookEvent),
        /// Events from scanner module
        #[prost(message, tag = "2")]
        Scanner(super::super::scanner::FileScannerEvent),
        /// Events from callbacks: process creation
        #[prost(message, tag = "3")]
        ProcessEvent(super::super::callbacks::ProcessEvent),
        /// Events from callbacks: image load
        #[prost(message, tag = "4")]
        ImageLoad(super::super::callbacks::ImageLoadEvent),
        /// Events from callbacks: registry modifications
        #[prost(message, tag = "5")]
        Registry(super::super::callbacks::RegistryEvent),
    }
}
```

Listado 3: *Estructura del mensaje* `HookEvent` *con payload variable*

```
message HookEvent {
  uint32 pid = 1;
  uint32 tid = 2;
  int32 status = 3;
  HookPayload payload = 4;
}

message HookPayload {
  oneof payload {
    NtCreateThreadExEvent       nt_create_thread_ex       = 1;
    NtMapViewOfSectionEvent     nt_map_view_of_section    = 2;
    NtProtectVirtualMemoryEvent nt_protect_virtual_memory = 3;
    NtSetValueKeyEvent          nt_set_value_key          = 4;
  }
}
```

cidencia. Este evento se ha generado desde el módulo de escaneo periódico del agente.

También se han definido eventos derivados de notificaciones del kernel, como `ProcessEvent`, `ImageLoadEvent` y `RegistryEvent`. Estos recogen información sobre la creación de procesos, carga de módulos o modificaciones en el registro, respectivamente. Por ejemplo, `ProcessEvent` incluye el PID, el PPID, la ruta al ejecutable y la línea de comandos. En el caso de `RegistryEvent`, se informa sobre la clave afectada, el tipo de operación (crear, borrar, modificar) y los valores antiguo y nuevo.

Todos estos subtipos han sido diseñados para ser autosuficientes: incluyen toda la información necesaria para entender el evento sin depender de contexto externo. Además, al encapsularlos dentro del tipo común `Event`, el sistema puede procesarlos de forma uniforme, sin necesidad de distinguir su origen a priori.

4.1.4. Serialización y deserialización de eventos

Una vez generado un evento por cualquiera de los sensores del agente, el siguiente paso ha sido transportarlo a través del sistema para su análisis, registro o envío externo. Para ello, todos los eventos definidos mediante archivos `.proto` se han serializado utilizando **Protocol Buffers** (**Protobuf**), un formato binario compacto y eficiente ampliamente utilizado para la comunicación entre procesos. La serialización de eventos se implementó con Protocol Buffers Google, 2025, que permitió definir un formato compacto y eficiente para intercambiar datos entre kernel y userland.

En este proyecto, la compilación de los esquemas Protobuf se ha realizado mediante la librería externa `prost`, que genera automáticamente las estructuras correspondientes en Rust con soporte completo para serialización y deserialización. Todos los eventos implementan el trait `prost::Message`, lo que permite convertir cualquier instancia en una secuencia de bytes con el método `encode()`, y reconstruirla posteriormente usando `decode()`.

Un ejemplo práctico de deserialización puede verse en el listado 4. En este caso, se ha recibido un buffer de bytes desde una cola de eventos o una memoria compartida, y se desea reconstruir el objeto original.

Este mecanismo permite tratar todos los eventos como entidades autocontenidas: una vez serializado, el evento puede transmitirse o almacenarse sin necesidad de contexto adicional. Al llegar al consumidor, basta con deserializarlo para recuperar todos sus campos originales
.
Una vez definido el modelo de eventos, el siguiente paso ha sido encontrar una forma eficiente de moverlos por el sistema. En la mayoría de los casos, eso implica pasar datos desde el kernel hasta el proceso en espacio de usuario que los recoge y los procesa. Cómo se ha resuelto exactamente ese paso —y qué implicaciones tiene— se detalla en la siguiente sección.

4.2. Comunicación kernel–user mediante sección compartida

Una vez serializados los eventos, es necesario transportarlos desde el contexto del kernel —donde se generan muchos de ellos— hacia el proceso en espacio de usuario que se encarga de

Listado 4: *Ejemplo de deserialización de un evento usando* `prost`

```rust
let raw_data: &[u8] = get_next_ring_buffer_message();

match Event::decode(raw_data) {
    Ok(evt) => {
        if let Some(kind) = evt.kind {
            match kind {
                EventKind::Scanner(s) => {
                    println!("Archivo sospechoso: {}", s.file);
                }
                EventKind::Hook(h) => {
                    println!("Hook detectado en PID {}", h.pid);
                }
                _ => println!("Otro tipo de evento recibido"),
            }
        }
    }
    Err(e) => {
        eprintln!("Error al deserializar evento: {e}");
    }
}
```

consumirlos, procesarlos o reenviarlos. Para este propósito se ha diseñado un mecanismo ligero y eficiente basado en una **sección de memoria compartida**, expuesta como un búfer circular de un único productor y un único consumidor (modelo SPSC).

Este canal se implementa utilizando una `SECTION` del kernel de Windows, que actúa como zona de memoria compartida entre el driver y el proceso del espacio de usuario. El driver es responsable de crear esta sección y mapearla en espacio del sistema, mientras que el consumidor en espacio de usuario la solicita a través de un `IOCTL` y la mapea localmente para lectura.

En los siguientes apartados se detalla cómo se ha estructurado esta comunicación: qué módulos y archivos están implicados en cada lado, cómo se ha implementado el buffer circular en memoria compartida, y qué mecanismos se han utilizado para garantizar un acceso seguro y sin bloqueos. También se explica cómo se configura la seguridad de la sección, cómo se expone al proceso en el modo usuario y cómo se realiza la lectura de eventos desde el agente.

4.2.1. Arquitectura y archivos implicados

El sistema de intercambio de eventos mediante memoria compartida se organiza de forma modular, con una separación clara entre el código que vive en el driver en modo kernel y el que corre en el agente en modo usuario. Esta separación permite desarrollar, probar y mantener cada parte de forma independiente, al tiempo que garantiza que ambas comparten la misma definición de estructura y formato de datos.

Los archivos más relevantes son los siguientes:

- `kernel-driver/communications/memory_ring.rs`: Implementa el lado productor del anillo. Aquí se define la estructura interna del buffer, se crea y mapea la sección de memoria compartida, y se exponen las funciones necesarias para escribir eventos desde el kernel.

Esta parte se encarga de gestionar el espacio disponible, detectar cuando no hay sitio suficiente, y avanzar el puntero de escritura (`head`) de forma segura.

- `user-agent/communications /memory_ring.rs`: Este archivo contiene el consumidor del ring en modo usuario. Una vez mapeada la sección compartida, el agente lee los eventos uno a uno, interpretando la longitud y el contenido de cada uno. Se encarga también de avanzar el puntero `tail`, permitiendo que el productor libere espacio para nuevos eventos.
- `shared/constants.rs`: Aquí se define el nombre simbólico global de la sección (`Global\Gladix`) que ambos extremos usan para identificar y acceder a la misma región de memoria. También se define la variable `IOCTL_GLADIX_GET_SECTION_HANDLE` para establecer un contrato entre ambas partes.
- `user-agent/comms/ioctl.rs`: Encapsula la lógica para enviar el `IOCTL` al driver y obtener el `HANDLE` que luego se usará para mapear la sección. Esta capa permite abstraer la complejidad del intercambio bajo una interfaz simple.

Esta organización permite que la comunicación kernel–user funcione de forma eficiente, con un mínimo de acoplamiento y una interfaz clara para ambos lados. En los siguientes apartados se explicará con más detalle cómo se crea la sección, cómo se estructura el buffer, y cómo se mapea y utiliza desde userland.

4.2.2. Implementación del ring buffer en memoria compartida

El canal de comunicación entre el driver y el proceso en modo usuario se basa en un **ring buffer** en memoria compartida, optimizado para funcionar con un único productor (el driver) y un único consumidor (el agente). Esta configuración —conocida como *single-producer single-consumer* o SPSC— permite evitar completamente el uso de locks, minimizando la latencia y el impacto en el rendimiento.

El buffer se implementa sobre una `SECTION` del kernel, una región de memoria compartida que puede mapearse tanto desde el kernel como desde userland. Esta sección actúa como almacén circular de eventos: el productor los escribe al final del buffer, y el consumidor los lee por orden desde el principio. Cuando el final de la región se alcanza, ambos extremos vuelven al inicio de forma natural —de ahí el nombre de "buffer circular"—.

El diseño garantiza que nunca se sobrescriben eventos no leídos. Si el productor detecta que no hay espacio suficiente para almacenar un nuevo mensaje completo, lo descarta y avanza sin bloquearse. Este comportamiento está pensado para priorizar la estabilidad del sistema frente a la fiabilidad total: si el consumidor se retrasa, es preferible perder un evento antes que bloquear al kernel.

Toda la información de control del anillo —punteros de lectura y escritura, contador de descartes y capacidad total— se mantiene en una cabecera común, accesible desde ambos extremos. El formato y la estructura de esta cabecera se describen en la siguiente sección.

4.2.3. Estructura e implementación del ring buffer

La sección compartida se organiza en dos partes: una cabecera fija y una región de datos que actúa como anillo circular. Ambas residen en una única región de memoria contigua, creada

por el driver y mapeada también en el proceso en modo usuario.

La cabecera ocupa los primeros 16 bytes del mapeo y contiene la información mínima necesaria para coordinar la escritura y lectura de eventos. Cada campo tiene una función bien definida:

Campo	Descripción
`head` (`AtomicU32`)	Offset del próximo byte libre para el productor
`tail` (`AtomicU32`)	Offset del siguiente byte a leer por el consumidor
`dropped` (`AtomicU32`)	Contador de eventos descartados por falta de espacio
`size` (`u32`)	Tamaño total del área de datos (en bytes)

Justo después de esta cabecera comienza el área de datos, que actúa como un buffer circular. Cada evento que se escribe allí sigue el formato:

`[u32 len_le][payload]`

Es decir, primero se escribe un entero de 32 bits en formato little-endian que indica el tamaño del mensaje, seguido del contenido real. Esta convención permite al consumidor saber cuánto mide exactamente cada evento, incluso si se encuentra fragmentado entre el final y el inicio del buffer.

Tanto el productor como el consumidor acceden a esta región utilizando `head` y `tail` como punteros lógicos. Para mantener el comportamiento circular, se emplea aritmética modular: una vez alcanzado el final del área de datos, los punteros vuelven automáticamente al inicio del buffer. Esto permite aprovechar toda la memoria disponible sin necesidad de mover datos.

Antes de escribir un nuevo evento, el productor necesita calcular cuántos bytes libres quedan en el anillo. Para ello, primero calcula el espacio usado:

$$\texttt{used} = \begin{cases} \texttt{head} - \texttt{tail}, & \text{si } \texttt{head} \geq \texttt{tail} \\ \texttt{size} - (\texttt{tail} - \texttt{head}), & \text{en otro caso} \end{cases} \qquad \texttt{free} = \texttt{size} - \texttt{used}$$

El primer caso refleja la situación más común: el buffer no ha hecho wrap-around y los datos ocupan un bloque lineal. En el segundo, el puntero de escritura ha dado la vuelta y está detrás del de lectura: los datos ocupan dos bloques, uno al final y otro al principio del buffer. En ambos escenarios, la fórmula garantiza que nunca se sobrescriba información no leída.

Una vez calculado el espacio disponible, si el mensaje completo (incluyendo los 4 bytes de longitud) cabe, el evento se escribe; si no, se descarta y se incrementa el contador `dropped`. En ningún caso se bloquea la ejecución del kernel ni se usa sincronización pesada.

La escritura se realiza en dos pasos: primero se escribe la longitud del mensaje, y luego el contenido serializado. Si el mensaje queda partido entre el final y el principio del buffer, se realiza una copia circular en dos tramos. Finalmente, el puntero `head` se actualiza mediante la operación atómica `Release`, asegurando que el consumidor no vea datos incompletos.

El listado 5 muestra esta lógica aplicada en la función `push_bytes` del driver.

Una vez definido el mecanismo de escritura en la sección compartida, el siguiente paso ha sido garantizar que el proceso en espacio de usuario pueda acceder a esa región de memoria

Listado 5: *memory_ring.rs push_bytes(): escritura de un evento en el anillo*

```
pub fn push_bytes(&self, buf: &[u8]) {
    if buf.len() + 4 > size {
        hdr.dropped.fetch_add(1, Ordering::Relaxed);
        return;
    }

    let head = hdr.head.load(Ordering::Relaxed) as usize;
    let tail = hdr.tail.load(Ordering::Acquire) as usize;
    let used = if head >= tail { head - tail } else { size - (tail - head) };
    let free = size - used;

    if free < buf.len() + 4 {
        hdr.dropped.fetch_add(1, Ordering::Relaxed);
        return;
    }

    let mut off = head;

    let len_le = (buf.len() as u32).to_le_bytes();
    unsafe { ptr::copy_nonoverlapping(len_le.as_ptr(), self.data_ptr(off), 4) };
    off = (off + 4) % size;

    let first = core::cmp::min(buf.len(), size - off);
    unsafe {
        ptr::copy_nonoverlapping(buf.as_ptr(), self.data_ptr(off), first);
        if first < buf.len() {
            ptr::copy_nonoverlapping(
                buf.as_ptr().add(first),
                self.data_ptr(0),
                buf.len() - first,
            );
        }
    }
    off = (off + buf.len()) % size;

    hdr.head.store(off as u32, Ordering::Release);
}
```

sin restricciones indebidas. Dado que Windows impone controles estrictos sobre quién puede mapear una `SECTION`, es necesario configurar explícitamente los permisos de acceso. En la siguiente sección se detalla cómo se construye y aplica el descriptor de seguridad adecuado para que el agente pueda leer y escribir en la sección sin requerir privilegios elevados.

4.2.4. Creación del descriptor de seguridad (SD)

Para que un proceso en Espacio de usuario pueda mapear la sección compartida creada por el driver, es necesario configurar explícitamente los permisos de acceso mediante un **descriptor de seguridad** (`SECURITY_DESCRIPTOR`). Este objeto encapsula tanto una DACL (Discretionary ACL) como una SACL (System ACL), que juntas definen quién puede acceder a la sección y bajo qué condiciones.

Aunque una DACL con acceso total para todos los usuarios podría parecer suficiente, en Windows esto no garantiza que un proceso con integridad media (Medium IL) pueda realmente escribir en la sección. Esto se debe a que el sistema de *Mandatory Integrity Control* (MIC) impone reglas adicionales según el nivel de integridad del proceso y del objeto.

Para resolver esto, se ha generado un **descriptor dual**, con las siguientes propiedades:

- **DACL:** se otorgan los permisos mínimos necesarios (`SECTION_MAP_READ` y `SECTION_MAP_WRITE`) al SID `S-1-1-0` (.Everyone").
- **SACL:** se etiqueta la sección con el SID de integridad media `S-1-16-8192`, aplicando la marca `NO_WRITE_UP` para que procesos con IL más bajo no puedan escribir, pero los de IL igual o superior sí.

Esto garantiza que el agente, incluso si corre como proceso interactivo sin privilegios, pueda mapear y escribir en la sección sin restricciones adicionales impuestas por MIC, mientras se mantiene el control sobre quién puede acceder.

El listado 6 muestra cómo se construye este descriptor dentro del driver, mediante llamadas a primitivas internas del kernel.

Este descriptor se incluye en la estructura `OBJECT_ATTRIBUTES` al crear la sección con `ZwCreateSection`. Gracias a esta configuración, el proceso en userland no necesita permisos elevados ni *"hacks"* para obtener acceso de lectura/escritura a la memoria compartida.

4.2.5. Consumo de eventos desde userland

Para que el agente en espacio de usuario pueda acceder a los eventos generados por el kernel, es necesario mapear la sección compartida que ambos comparten. Aunque Windows permite acceder a estas secciones mediante su nombre simbólico (por ejemplo, con `OpenFileMappingW`), este enfoque puede fallar fácilmente en escenarios reales: distintas sesiones de usuario, niveles de integridad dispares o restricciones de acceso pueden impedir que el agente obtenga un `HANDLE` válido.

Para evitar estos problemas, el sistema implementa un mecanismo de traspaso explícito de handles: el agente solicita al driver un `HANDLE` válido a través de un `IOCTL` especial llamado

Listado 6: *Creación del descriptor de seguridad con DACL y SACL*

```
pub unsafe fn for_everyone() -> Result<SecurityDescriptor, NTSTATUS> {
    // SID de "Everyone" y Medium IL
    static EVERYONE_SID: [u8; 12] = [ ... ];
    static MEDIUM_IL_SID: [u8; 12] = [ ... ];

    // Calcular tamaños de SD, DACL y SACL
    let sd_size = size_of::<SECURITY_DESCRIPTOR>();
    let dacl_len = ...; // ACE + SID
    let sacl_len = ...; // Mandatory ACE + SID

    let total = sd_size + dacl_len + sacl_len;
    let raw = ExAllocatePool2(POOL_FLAG_PAGED, total as SIZE_T, POOL_TAG)?;

    let sd_ptr = raw as PSECURITY_DESCRIPTOR;
    let dacl_ptr = (raw as *mut u8).add(sd_size) as *mut ACL;
    let sacl_ptr = (raw as *mut u8).add(sd_size + dacl_len) as *mut ACL;

    // Inicializar SD y DACL
    RtlCreateSecurityDescriptor(sd_ptr, ...);
    RtlCreateAcl(dacl_ptr, ...);
    RtlAddAccessAllowedAce(dacl_ptr, ACL_REVISION, SECTION_RW_MASK, world_sid);
    RtlSetDaclSecurityDescriptor(sd_ptr, TRUE, dacl_ptr, FALSE);

    // Inicializar SACL con etiqueta Medium-IL y marca NO_WRITE_UP
    RtlCreateAcl(sacl_ptr, ...);
    RtlAddMandatoryAce(sacl_ptr, ACL_REVISION, 0, il_sid, ACE_TYPE, NO_WRITE_UP);
    RtlSetSaclSecurityDescriptor(sd_ptr, TRUE, sacl_ptr, FALSE);

    Ok(SecurityDescriptor { sd_ptr })
}
```

`IOCTL_GLADIX_GET_SECTION_HANDLE`. Esta operación se ejecuta en el contexto del proceso solicitante, lo que garantiza que el descriptor devuelto tenga los permisos adecuados para ser utilizado desde userland.

Una vez recibido el handle, el agente lo utiliza para mapear la sección completa en su espacio de direcciones mediante `MapViewOfFile`, con permisos de lectura y escritura. La escritura es necesaria para poder actualizar el campo `tail` de la cabecera, que indica hasta qué punto se han procesado los eventos. El siguiente listado 7 muestra cómo se realiza esta operación desde el lado del agente.

Listado 7: *Mapping de la sección compartida desde userland*

```
fn open_via_ioctl() -> io::Result<Mapping> {
    let handle = request_section_handle()?; // comunica con el driver

    let view_addr = unsafe {
        MapViewOfFile(handle, FILE_MAP_READ | FILE_MAP_WRITE, 0, 0, 0)
    };

    if view_addr.Value.is_null() {
        let err = io::Error::last_os_error();
        unsafe { CloseHandle(handle) };
        return Err(err);
    }

    let view = unsafe { NonNull::new_unchecked(view_addr.Value as *mut u8) };
    Ok(Mapping { handle, view })
}
```

Con la sección mapeada, el agente accede directamente a la cabecera y al área de datos. El campo `tail` se inicializa con su valor actual, y el sistema queda listo para comenzar el consumo de eventos.

Para leer un nuevo evento, el agente compara su puntero `tail` con el puntero `head` actualizado por el productor. Si ambos coinciden, no hay eventos disponibles y se retorna `None`. En caso contrario, se leen los primeros 4 bytes a partir de `tail` para determinar la longitud del evento.

Si la longitud es válida y el evento completo ya ha sido escrito por el kernel, el agente copia los datos —teniendo en cuenta que podrían estar divididos entre el final y el inicio del buffer—, los devuelve como `Vec<u8>` y actualiza su puntero local. Finalmente, almacena el nuevo valor de `tail` en la cabecera compartida usando semántica `Release`, lo que permite al productor liberar espacio sin necesidad de sincronización adicional.

El listado 8 muestra esta lógica de lectura.

En caso de datos corruptos o longitud inválida, el agente puede recuperar el sincronismo saltando directamente a `head`. Al evitar bloqueos y acoplamiento entre productor y consumidor, se logra un sistema robusto y adecuado para entornos en tiempo real.

Una vez que el agente ha consumido un evento desde el anillo, el siguiente paso consiste en procesarlo y almacenarlo de forma persistente para su posterior análisis. La sección siguiente

Listado 8: *Lectura de un evento desde userland*

```rust
pub fn next(&mut self) -> Option<Vec<u8>> {
    let head = self.hdr.head.load(Ordering::Acquire);
    if self.tail == head {
        return None;
    }

    // Leer longitud del mensaje (4 bytes LE)
    let len = {
        let mut tmp = [0u8; 4];
        self.copy_circular(self.tail, &mut tmp);
        u32::from_le_bytes(tmp)
    };

    if len == 0 || len > self.size - 4 {
        self.tail = head;
        self.hdr.tail.store(self.tail, Ordering::Release);
        return None;
    }

    // Verificar si el mensaje completo está disponible
    let avail = if self.tail <= head {
        head - self.tail
    } else {
        self.size - self.tail + head
    };
    if avail < len + 4 {
        return None;
    }

    // Copiar mensaje
    let mut buf = vec![0u8; len as usize];
    self.copy_circular(self.tail + 4, &mut buf);

    self.tail = (self.tail + 4 + len) % self.size;
    self.hdr.tail.store(self.tail, Ordering::Release);

    Some(buf)
}
```

detalla el sistema de persistencia basado en base de datos embebida.

4.3. Persistencia de eventos en base de datos

Una vez que los eventos llegan al proceso en espacio de usuario y han sido leídos correctamente desde la sección compartida, el siguiente paso es guardarlos de forma duradera para su análisis posterior. Aunque ya están disponibles en memoria, contar con una copia persistente permite revisar el histórico, generar reglas más complejas o simplemente no perder información en caso de cierre inesperado del proceso.

Para eso se ha optado por una base de datos embebida `SQLite`, configurada específicamente para soportar una alta tasa de escrituras sin penalizar el rendimiento. El almacenamiento se hace en segundo plano: el hilo principal recibe y decodifica los eventos, y se los pasa a un hilo

de persistencia a través de un canal interno. Así se evita que los picos de carga o latencias de disco bloqueen la ingesta de datos.

En esta sección veremos cómo está organizada esa base de datos, cómo se inicializa al arrancar el sistema, y cómo se realiza la persistencia de cada tipo de evento. También se explica el mecanismo basado en traits que permite extender el sistema fácilmente: cada evento sabe cómo debe almacenarse, sin que el hilo de persistencia tenga que conocer su estructura interna.

4.3.1. Estructura de la base de datos

El sistema de persistencia utiliza SQLite como motor de base de datos embebido. Este motor ha sido configurado para ofrecer un buen equilibrio entre rendimiento y durabilidad, activando el modo `WAL` (Write-Ahead Logging), una política de sincronización `NORMAL`, y limitando el tamaño del journal a 50 MB mediante pragmas en tiempo de arranque. El modo *Write-Ahead Logging* permite un escritor y múltiples lectores concurrentes, mejora la durabilidad ante caídas y simplifica el consumo de telemetría. Como límite operativo, el agente asume un único proceso escribiendo en la base de datos.

El esquema se divide en dos grandes bloques: eventos provenientes de **hooks en el kernel** (como llamadas a `NtCreateThreadEx` o `NtMapViewOfSection`) y eventos generados a partir de **callbacks** (como creación de procesos, carga de imágenes o modificaciones del registro).

Eventos hook. Todos los eventos capturados mediante mecanismos de hook comparten una tabla cabecera común: `hook_event`. Esta tabla contiene campos genéricos como el `pid`, `tid`, tipo de payload (`payload_kind`) y una marca de tiempo. Cada tipo de evento concreto se almacena en una tabla separada relacionada mediante clave foránea sobre el campo `event_id`. Por ejemplo:

- `hook_event_nt_create_thread_ex`: almacena la dirección de entrada, argumentos y flags de una creación de hilo remota.
- `hook_event_nt_map_view_of_section`: recoge detalles del mapeo de una región de memoria (dirección base, tamaño, protección).
- `hook_event_nt_protect_virtual_memory`: refleja cambios en protecciones de memoria.
- `hook_event_nt_set_value_key`: representa modificaciones del registro a nivel clave-valor.

Este modelo evita una tabla monolítica y facilita consultas específicas por tipo, sin perder la trazabilidad global del evento.

Eventos de callback. Los eventos generados a partir de mecanismos como `PsSetCreateProcessNotifyRoutineEx` o `CmRegisterCallbackEx` se almacenan en tablas individuales:

- `process_event`: contiene información de procesos creados, incluyendo PID, PPID, imagen y línea de comandos.
- `image_load_event`: almacena los binarios cargados en cada proceso, con su base, tamaño y ruta absoluta.
- `registry_event`: registra operaciones sobre el registro, con los valores antes y después, la clave modificada y el proceso responsable.

Cada una de estas tablas tiene índices en campos relevantes para acelerar consultas frecuentes, como búsquedas por PID, por nombre de módulo o por clave de registro. La normalización del esquema, junto con una segmentación lógica por tipo, permite mantener la base de datos eficiente incluso en escenarios con volúmenes altos de eventos.

4.3.2. Inicialización de la base de datos

La base de datos SQLite se inicializa al arrancar el agente en Espacio de usuario. Este proceso se encarga de crear el archivo en disco si no existe, aplicar las opciones de configuración adecuadas y ejecutar el esquema definido en un archivo externo (`schema.sql`).

Esta lógica se encuentra encapsulada en la función `init_database`, ubicada en el archivo `db/connection.rs`. El comportamiento es el siguiente:

1. Se construye la ruta completa del archivo de base de datos a partir del directorio de ejecución y del nombre configurado por el usuario.
2. Si el archivo no existe (primera ejecución), se marca como primera inicialización.
3. Se abre la conexión SQLite y se aplican pragmas para optimizar el rendimiento:
   - `journal_mode = WAL` activa el modo *Write-Ahead Logging*, permitiendo escrituras concurrentes sin bloquear lectores.
   - `synchronous = NORMAL` ofrece un equilibrio entre durabilidad y velocidad.
   - `journal_size_limit = 52428800` establece un límite de 50 MB para el tamaño del journal.
4. Si es la primera ejecución, se ejecuta todo el contenido del archivo `schema.sql` en un único batch.

Este diseño permite que la base de datos se cree automáticamente sin intervención manual, mientras mantiene la posibilidad de modificar su configuración desde un fichero externo. El listado 9 muestra esta lógica.

Gracias a este enfoque, el sistema puede inicializar y configurar automáticamente su almacenamiento sin necesidad de migraciones manuales ni dependencias externas.

4.3.3. Canal de eventos y persistencia en segundo plano

Una vez inicializada la base de datos, el agente lanza un hilo dedicado exclusivamente a persistir los eventos recibidos desde el kernel. Para comunicar los eventos entre el hilo principal y este consumidor, se utiliza un canal (`std::sync::mpsc::channel`) que transmite instancias del tipo `Event`.

Este diseño permite desacoplar completamente la lógica de captura y decodificación de eventos de su almacenamiento en disco. El canal actúa como búfer intermedio: el hilo principal puede enviar eventos rápidamente sin bloquearse, mientras el hilo de persistencia los procesa a su ritmo.

El flujo funciona de la siguiente forma:

1. El hilo principal decodifica cada evento recibido desde el ring buffer compartido.
2. Una vez reconstruido, lo envía por el canal al hilo de persistencia.

Listado 9: *Inicialización de la base de datos*

```rust
pub fn init_database(
    db_path: &PathBuf,
    cfg: &DatabaseConfig,
) -> Result<Connection, rusqlite::Error> {
    let path = db_path.join(&cfg.path);
    let first_run = !path.exists();

    let conn = Connection::open(&path)?;
    conn.busy_timeout(Duration::from_millis(1_000))?;
    conn.pragma_update(None, "journal_mode", &"WAL")?;
    conn.pragma_update(None, "synchronous", &cfg.synchronous.as_str())?;
    conn.pragma_update(None, "journal_size_limit", &(cfg.journal_size_limit as i64))?;

    if first_run {
        let schema = include_str!("schema.sql");
        conn.execute_batch(schema)?;
    }

    Ok(conn)
}
```

3. El hilo de persistencia lee del canal en bucle y llama al método `send_to_db` del evento.

El listado 10 muestra cómo se lanza este hilo de escucha:

Listado 10: *Lanzamiento del hilo de persistencia*

```rust
fn spawn_db_listener(db_conn: rusqlite::Connection, db_rx: EventRx) {
    let builder = thread::Builder::new().name("event_listener".into());
    if let Err(e) = builder.spawn(move || {
        for ev in db_rx.iter() {
            ev.send_to_db(&db_conn);
        }
        panic!("All event channel senders dropped!");
    }) {
        error!("Failed to spawn DB listener thread: {e}");
    }
}
```

Este hilo se mantiene activo indefinidamente mientras existan emisores vivos. Si todos los productores se cierran por error o finalización del programa, se detecta como condición anómala y se aborta con un `panic` explícito, lo que ayuda a identificar fallos lógicos en tiempo de desarrollo.

Este enfoque, el almacenamiento se realiza de forma asíncrona, sin bloquear la recepción de eventos, y permite manejar ráfagas de alta frecuencia sin perder datos mientras haya capacidad de procesamiento.

Aside: Cambios en caliente Los *workers* consultan el gestor de configuración en cada iteración y aplican variaciones del intervalo sin reinicio del servicio. Este patrón permite *hot-reload* acotado (parámetros no estructurales) manteniendo el bucle de escaneo estable.

4.3.4. Persistencia basada en traits y despacho dinámico

Para almacenar los eventos en la base de datos de forma modular y extensible, se utiliza un mecanismo basado en traits. En concreto, cada tipo de evento implementa el trait `DbLoggable`, que define el método `send_to_db`. Este método recibe una conexión SQLite y ejecuta la inserción correspondiente a su tipo, sin necesidad de que el hilo de persistencia conozca los detalles de cada uno.

La estructura `Event`, que representa un mensaje genérico, también implementa este trait. Su implementación actúa como un punto de entrada común: analiza qué tipo concreto de evento contiene (por ejemplo, `Hook`, `Scanner`, `ProcessEvent`, etc.) y delega la llamada en la variante correspondiente. Si el evento está vacío —algo que no debería ocurrir en condiciones normales—, se deja constancia en el log.

Esto permite que el hilo de persistencia pueda tratar todos los eventos por igual, sin importar su tipo real. Cada uno se encarga de su propia lógica de almacenamiento, lo que facilita el mantenimiento y la extensión del sistema: para añadir un nuevo tipo de evento, basta con implementar el trait y registrar su variante.

Por ejemplo, un evento de tipo `FileScannerEvent` simplemente inserta el nombre del fichero y la regla asociada. La inserción se realiza usando parámetros enlazados, y en caso de error se registra el fallo sin interrumpir el flujo general. Esto se puede apreciar en el listado 11.

Listado 11: *Persistencia de eventos de escaneo en base de datos*

```
impl DbLoggable for FileScannerEvent {
    fn send_to_db(&self, conn: &Connection) {
        conn.execute(
            "INSERT INTO file_scanner (file, rule_name) VALUES (?1, ?2)",
            params![&self.file, &self.rule_name],
        )
        .inspect_err(|e| {
            error!(
                "DB error inserting file_scanner (file={}, rule_name={}): {}",
                self.file, self.rule_name, e
            )
        })
        .ok();
    }
}
```

Este enfoque proporciona una interfaz clara, desacoplada y fácil de extender. Cada evento sabe cómo guardarse a sí mismo, y el hilo de persistencia simplemente actúa como un despachador que recibe y entrega los datos a su destino final.

4.4. Escaneo de archivos con reglas YARA

Además de observar el sistema en tiempo real, una parte importante de este proyecto es ser capaz de buscar archivos maliciosos de forma activa. Para eso se ha incorporado un módulo de escaneo periódico basado en reglas **YARA**, una herramienta muy conocida en análisis de malware por su capacidad para identificar patrones dentro de archivos binarios. Estas reglas

pueden detectar desde familias concretas de malware hasta comportamientos sospechosos más genéricos, según cómo estén definidas.

La idea detrás de este componente es sencilla: cada cierto tiempo, el agente revisa una serie de directorios configurados y analiza los archivos que encuentra usando las reglas YARA cargadas al inicio. Este análisis no es continuo, sino que se realiza de forma programada según el nivel de riesgo que se haya asignado a cada ruta. Así, se pueden escanear con más frecuencia ciertas carpetas sensibles y dejar otras zonas con menor prioridad.

Este módulo se integra dentro del agente en modo usuario y funciona de manera independiente al resto de sensores. Está pensado para ser lo más autónomo y configurable posible. Con ese fin se puede ajustar el intervalo de escaneo, las extensiones permitidas, el tamaño máximo de archivo, e incluso definir las reglas YARA desde un archivo individual o desde un paquete comprimido. En esta sección se explica cómo se ha diseñado todo esto, cómo está estructurado el código y cómo funciona el escaneo desde dentro.

4.4.1. Arquitectura y estructura del código fuente

La funcionalidad de escaneo con YARA se encuentra agrupada dentro del módulo `scanner`, ubicado en el agente en modo usuario. Este módulo contiene todos los componentes necesarios para gestionar la carga de firmas, organizar los hilos de trabajo y controlar el proceso de escaneo de archivos.

El archivo principal es `service.rs`, donde se define el servicio central encargado de lanzar y coordinar los hilos de escaneo. Cada hilo representa un grupo de riesgo distinto, como `high`, `medium` o `low`, según lo definido en el fichero de configuración.

Las reglas YARA se gestionan desde `signatures.rs`, que encapsula la lógica necesaria para cargar y compilar las firmas desde diferentes orígenes. Este archivo abstrae los detalles de si las reglas provienen de un único archivo `.yar` o de un archivo comprimido con múltiples firmas.

El archivo `worker.rs` contiene la definición del trabajador que ejecuta el escaneo como tal. Este componente se encarga de recorrer los directorios asignados y aplicar las reglas a los archivos que correspondan. Aunque cada worker funciona de manera independiente, todos comparten la misma lógica base definida en este módulo.

Para optimizar el rendimiento del sistema, se incluye también `cache.rs`, que implementa una caché en memoria ligera. Su única función es recordar qué archivos ya han sido escaneados recientemente para evitar volver a analizarlos si no han cambiado.

Por último, la integración con el resto del sistema se realiza desde el punto de entrada (main), a través de una función `init_scanner()` que inicializa todo el módulo de escaneo y se asegura de que esté listo antes de que el agente entre en funcionamiento.

4.4.2. Carga y compilación de reglas YARA

Antes de poder escanear archivos, es necesario cargar y compilar las reglas YARA que se van a utilizar. Para eso, el sistema cuenta con un componente dedicado dentro del módulo `scanner`,

ubicado en el archivo `signatures.rs`. Este módulo abstrae todos los detalles relacionados con la carga de firmas y expone una única función pública: `Signatures::load()`, que devuelve una estructura `Rules` lista para usar.

La carga admite dos modos: se puede pasar una ruta directa a un archivo `.yar`, o bien un archivo ZIP que contenga múltiples ficheros de reglas. Internamente, el código utiliza una enumeración `SignatureSource` para representar estas dos variantes:

```
pub enum SignatureSource {
    File(String),
    Zip(String),
}
```

Según el tipo, se ejecuta una de las dos funciones privadas: `parse_file()` para cargar un único archivo, o `parse_zip()` si se trata de un archivo comprimido. Ambas funciones utilizan un compilador de YARA proporcionado por la librería `yara-x`:

```
let mut compiler = Compiler::new();

match source {
    SignatureSource::File(path) => Self::parse_file(&mut compiler, path)?,
    SignatureSource::Zip(path)  => Self::parse_zip(&mut compiler, path)?,
};

let rules = compiler.build();
```

En el caso del ZIP, se recorren todos los ficheros del archivo comprimido y se compilan aquellos que tengan extensión `.yar` o `.yara`. Esto permite mantener las firmas organizadas por familias o categorías sin tener que juntarlas manualmente.

La compilación es estricta: si alguna regla es inválida, el proceso se detiene y se devuelve un error. Todos los errores posibles —ya sea de lectura, descompresión o compilación— se agrupan en una enumeración común llamada `SignaturesError`, lo que facilita su propagación y registro desde el exterior.

Este diseño permite al agente inicializar el escáner de forma segura. En caso de que la carga de reglas falle (por ejemplo, porque el archivo no existe o contiene errores), el módulo se salta silenciosamente sin afectar al resto del sistema, y se registra el error en el log para su análisis posterior.

Aside: Coste operativo de reglas Con un conjunto de aproximadamente `3506 reglas`, el tiempo típico de compilación es del orden de `60 segundos` y el consumo de memoria ronda los `100MB`. Se recomienda compilar una vez y reutilizar el objeto de reglas en memoria, invalidándolo solo tras cambios de configuración.

4.4.3. Lanzamiento y funcionamiento de los workers

Una vez cargadas las reglas YARA, el siguiente paso es distribuir el trabajo de escaneo entre varios hilos de ejecución. Esto se hace mediante un sistema de **workers** independientes, uno por cada grupo de riesgo configurado en el fichero `config.toml`. Cada worker se encarga de escanear periódicamente las rutas asociadas a su grupo, aplicando las reglas a los archivos que cumplan los criterios definidos.

El lanzamiento de los workers lo realiza el componente `ScannerService`, ubicado en `service.rs`. Esta estructura recibe como parámetros la configuración global, las reglas ya compiladas y el canal de eventos del sistema. Durante su inicialización, recorre los grupos definidos en la configuración y lanza un hilo de trabajo para cada uno. En el listado 12 se muestra cómo se lleva a cabo este proceso:

Listado 12: *Lanzamiento de workers desde `ScannerService`*

```
for (risk, _group_cfg) in risk_groups.iter() {
    let cache = ScanCache::default();
    let rules = Arc::clone(&self.rules);
    let cfg = Arc::clone(&self.cfg_mgr);

    spawn_worker(rules, cfg, risk.clone(), cache, self.evt_tx.clone());
}
```

Cada hilo se inicializa con su propio conjunto de reglas, acceso a la configuración, una caché de archivos y un canal de salida para los eventos. Estos hilos no se sincronizan entre sí, ya que trabajan sobre directorios distintos y no comparten estado.

El cuerpo del worker se encuentra en `worker.rs`. Allí se define una estructura `Worker` que encapsula la lógica de escaneo, incluyendo la lectura del intervalo de ejecución y la iteración sobre los directorios definidos. Cada worker ejecuta su bucle principal de forma indefinida: escanea, espera y repite. Además, comprueba en cada ciclo si ha cambiado el intervalo de escaneo, lo que permite aplicar cambios en caliente desde el archivo de configuración. La lógica de este ciclo puede verse en el listado 13.

Listado 13: *Bucle principal de ejecución de un worker*

```
loop {
    self.scan_directories(cache, evt_tx.clone());

    if let Some(new_int) = self.current_interval() {
        if new_int != interval {
            interval = new_int;
        }
    }

    thread::sleep(Duration::from_secs(interval));
}
```

Este diseño permite distribuir la carga de trabajo de forma flexible. Por ejemplo, las rutas marcadas como `high risk` se pueden escanear cada 60 segundos, mientras que las de menor prioridad pueden hacerlo con menos frecuencia. También se puede activar o desactivar un grupo

completo simplemente modificando la configuración.

4.4.4. Uso de caché para evitar escaneos redundantes

Escanear archivos binarios con YARA puede ser una operación costosa, sobre todo si se trata de ficheros grandes o de un volumen elevado. Para evitar escanear repetidamente archivos que no han cambiado entre ciclos, cada worker mantiene una pequeña caché en memoria que registra metadatos básicos sobre los archivos ya procesados.

Esta caché está implementada en el archivo `cache.rs`, bajo la estructura `ScanCache`. Es una estructura de tipo LRU (Least Recently Used), limitada a un número fijo de entradas, que asocia cada archivo con una tupla formada por su tamaño y la marca de tiempo de última modificación. Si ambas propiedades coinciden con una versión anterior, se asume que el contenido no ha cambiado y se salta el escaneo.

El método principal para esta comprobación es `is_unchanged()`, que recibe la ruta de un archivo y sus metadatos, y devuelve un booleano. El listado 14 muestra cómo se realiza esta verificación.

Listado 14: *Comprobación de cambio en archivo mediante caché*

```
pub fn is_unchanged(&self, path: &Path, meta: &Metadata) -> bool {
    let stamp = Self::stamp(meta);
    self.map.get(path).copied() == Some(stamp)
}
```

Cada vez que un archivo es escaneado con éxito, su entrada en la caché se actualiza mediante el método `update()`. Internamente, la caché utiliza un `FxHashMap` para las búsquedas rápidas y una cola `VecDeque` para controlar el orden de inserción y poder eliminar los elementos más antiguos cuando se supera el límite de capacidad.

Este enfoque no es infalible, pero ofrece un buen equilibrio entre precisión y rendimiento. Usar un hash criptográfico para detectar cambios sería más robusto, pero implicaría una penalización de rendimiento considerable. En cambio, comparar `mtime` y tamaño es lo bastante fiable para el tipo de escenarios en los que este agente va a operar, y permite mantener tiempos de escaneo bajos incluso en directorios con muchos binarios.

4.4.5. Generación y envío de eventos

Cuando una regla YARA coincide con el contenido de un archivo, el sistema genera un evento para reportar esa detección. Esta lógica está implementada directamente dentro de los workers, concretamente en la función `scan_one()`, que se encarga de aplicar las reglas sobre cada archivo individual. Si se detectan coincidencias, se construyen objetos de tipo `FileScannerEvent`, que encapsulan información como el nombre de la regla y la ruta del archivo afectado.

Cada evento generado se empaqueta dentro de un objeto `Event`, junto con su tipo, y se envía a través del canal de eventos general del sistema (`EventTx`). De este modo, el módulo de escaneo queda desacoplado del componente que consume los eventos, permitiendo que otras partes del

agente —como el sistema de almacenamiento o la capa de respuesta— actúen sobre las detecciones si es necesario.

El proceso puede verse en el listado 15, donde se muestra cómo se recorren las coincidencias encontradas y se envían los eventos al canal correspondiente.

Listado 15: *Generación y envío de eventos tras una detección YARA*

```
for m in matches.matching_rules() {
    let event = FileScannerEvent {
        file: path.to_string_lossy().into_owned(),
        rule_name: m.identifier().to_string(),
    };
    let evt = Event {
        kind: Some(EventKind::Scanner(event)),
    };

    if let Err(e) = event_tx.send(evt) {
        error!("event channel closed: {e}. Skipping event.");
        break;
    }
}
```

El tipo de evento utilizado aquí es `EventKind::Scanner`, que está reservado dentro del esquema de eventos para notificar resultados del módulo YARA. Esto facilita su identificación en los logs o en el almacenamiento posterior. Además, si el canal se encuentra cerrado —por ejemplo, si el sistema está apagándose— se captura el error y se evita que el hilo entre en pánico.

En conjunto, este mecanismo permite integrar de forma limpia las detecciones realizadas por el escáner con el flujo de eventos general del agente, sin necesidad de una lógica específica para el tipo de amenaza o el origen del archivo. Esto hace que el sistema sea fácil de extender o adaptar a otros módulos en el futuro.

Gracias a esta arquitectura modular, el escaneo con reglas YARA puede ejecutarse de forma periódica, configurable y aislada, sin interferir con el resto de componentes del sistema. Cada detección se reporta como un evento más dentro del pipeline del agente, lo que permite tratar estos hallazgos de la misma forma que cualquier otra fuente de telemetría. A partir de aquí, el foco pasa de escaneos programados a sensores que actúan en tiempo real, observando el comportamiento del sistema conforme ocurren los eventos. La siguiente sección explora precisamente ese tipo de mecanismos, empezando por la monitorización directa de procesos desde el kernel.

4.5. Userland Hooking mediante inyección de DLL

Tal y como se ha comentado en el apartado Subsección 2.2.1, este proyecto implementa una DLL hookeable para interceptar funciones del subsistema nativo de Windows (las llamadas `Nt` exportadas por `ntdll.dll`). La idea principal detrás de esta parte es capturar ciertas llamadas sensibles desde cualquier hilo del proceso y reenviar esa información a nuestro driver y poder tomar acciones rápidas o incluso denegar la llamada en un futuro.

Uno de los objetivos de esta parte era implementar la funcionalidad totalmente desde 0, buscando que el *Hooking* sea limpio, reversible y multiplataforma (x86 y x64). En los próximos apartados veremos como esta compuesto el código y como esta implementada esta funcionalidad para conseguir este mismo objetivo.

Si quieres ampliar información y contexto sobre este modulo puedes encontrar el código completo en la carpeta **Hooking-lib** en la raíz del repositorio del proyecto.

4.5.1. Arquitectura general

Tal y como hemos comentado previamente, la DLL está dividida en varios módulos, cada uno con su propia función. A continuación un resumen rápido:

- `lib.rs`: Punto de entrada. Se encarga de lanzar el proceso de instalación de hooks cuando la DLL se carga.
- `hooks.rs`: Encargado de hacer el parcheo de las funciones. Aquí se calculan offsets, se sobrescriben instrucciones, etc. Es donde se encuentra la parte mas técnica de este modulo.
- `call_guard.rs`: Implementa una especie de guardia de re-entrada por cada hilo (thread). Si el hook se dispara recursivamente, lo detecta y evita loops infinitos.
- `detours.rs`: Aquí se definen las funciones que queremos que se ejecuten cuando se llama a una función hookeada. Se definen llas funciones que queremos interceptar y la logica quq queremos que sigan.
- `manager.rs`: Administra la lista de hooks activos. Se encarga de instalarlos, desinstalarlos y mantener su estado.
- `comms.rs`: Implementa la comunicación con el exterior usando `DeviceIoControl` con nuestro driver.

4.5.2. Detour patching en `ntdll.dll`

A la hora de implementar Hooking como tal existen numerosas técnicas, como el *inline Hooking*, *IAT/EAT patching*, uso de vectores de excepciones o trampas de depuración mediante registros de hardware (también conocidos como *hardware breakpoints*). En este proyecto se ha elegido una técnica de *inline Detour Hooking*, también conocida como sobrescritura de prólogo, por ser una de las más directas y eficaces para este tipo de interceptación en tiempo de ejecución.

La idea consiste en modificar los primeros bytes de la función objetivo, insertando un salto incondicional hacia una rutina propia —el Detour— que actúa como intermediario. Esta rutina puede inspeccionar los argumentos, decidir si continuar con la ejecución, alterar el comportamiento, o incluso bloquear la llamada. Después de ejecutar la lógica deseada, el Detour transfiere el control de vuelta a la función original mediante una puerta de enlace o gateway, que contiene los bytes originales de la función antes del parcheo y un salto incondicional al resto de la función.

Aside; Qué modifica Detours La librería sobrescribe el prólogo de funciones objetivo con un salto a un *trampolín* controlado por la DLL, preservando instrucciones desplazadas. Tras ejecutar el *pre-hook*, el control puede volver a la implementación original o simular una respuesta según la política del hook.

Este método requiere especial atención en cuanto al número de bytes que se sobrescriben, ya que debe ser suficiente para colocar el salto sin romper instrucciones importantes. En arquitecturas de 64 bits, típicamente se emplean 13 bytes para insertar un `mov rax, addr` seguido de `jmp rax`, lo cual garantiza un salto absoluto fiable. En sistemas de 32 bits, el mismo efecto se logra en 7 bytes mediante `mov eax, addr` y `jmp eax`.

En el listado 19 se muestra un fragmento simplificado del código responsable de construir el patch que se aplicará sobre la función original al momento de instalar el hook:

Listado 16: *hooks.rs new(): patch code creation*

```
pub unsafe fn new(dll: &str, func: &str, detour: *const u8) -> Result<Self, String> { unsafe {
...
let mut patch = [0u8; PATCH_LEN];
#[cfg(target_pointer_width = "64")]
{
    // mov r10,<detour>
    patch[0] = 0x49;
    patch[1] = 0xBA;
    patch[2..10].copy_from_slice(&(detour as u64).to_le_bytes());
    // jmp r10
    patch[10] = 0x41;
    patch[11] = 0xFF;
    patch[12] = 0xE2;
}
#[cfg(target_pointer_width = "32")]
{
    // mov eax,<detour>
    patch[0] = 0xB8;
    patch[1..5].copy_from_slice(&(detour as u32).to_le_bytes());
    // jmp eax
    patch[5] = 0xFF;
    patch[6] = 0xE0;
}
...
}

```

Una parte crítica del proceso de *Hooking* es preservar el comportamiento original de la función que estamos monitorizando. Para ello, es necesario guardar los primeros bytes de la función antes de aplicar el *patch*, ya que estos serán sobrescritos por el salto al *Detour*. Estos bytes se utilizarán posteriormente para construir nuestro *gateway*. Este nos permite ejecutar la función original de forma transparente tras pasar por el *hook*.

Este paso garantiza que las instrucciones de la función original se mantienen intactas, incluyendo detalles sensibles como el identificador del *syscall*. En el listado 17 se puede ver cómo se realiza esta copia previa de instrucciones, que luego se integrará en el gateway dinámico:

4.5.3. Protección contra la reentrada

El objeto **CallGuard** actúa como una barrera de reentrada a nivel de hilo, utilizando un *slot* de almacenamiento local (TLS por sus siglas en inglés) para marcar si un hilo está ejecutando código dentro del hook. Esto es esencial para evitar recursividad o bucles infinitos, que podrían

Listado 17: *hooks.rs new(): copy original function before patching*

```
pub unsafe fn new(dll: &str, func: &str, detour: *const u8) -> Result<Self, String> { unsafe {
...
// Save the first PATCH_LEN bytes so we can restore them later
let mut saved = [0u8; PATCH_LEN];
ptr::copy_nonoverlapping(target, saved.as_mut_ptr(), PATCH_LEN);
...
}
```

producirse si, por ejemplo, una llamada a `NtReadVirtualMemory` dentro del *Detour* termina generando de nuevo una llamada a la misma función.

La técnica implementada garantiza que cada hilo solo puede entrar una vez en la lógica del Detour. Si se detecta una reentrada, el control se transfiere directamente a la función original, evitando así bucles infinitos o desbordamientos de pila. Esta solución es especialmente útil en contextos donde las propias funciones de Windows o librerías externas pueden invocar de forma indirecta la misma syscall interceptada. Mas adelante veremos un problema relacionado con esto mismo.

La gestión de esta lógica se hace mediante RAII (inicialización y liberación automática con Drop) lo que ayuda a prevenir errores y facilita su reutilización. En el listado 18 podemos ver el código que hace referencia a esta funcionalidad.

4.5.4. Administración de hooks: `HookManager`

Para organizar y simplificar la instalación de múltiples hooks, se define una estructura llamada `HookEntry`, que encapsula toda la información necesaria para describir un hook. Esta estructura necesita los siguientes datos para ser inicializada: la DLL objetivo, el nombre exacto de la función a interceptar, la dirección de la rutina de desvío (*Detour*) y un puntero donde se almacenará la dirección del gateway original para llamadas posteriores.

Estas entradas se gestionan a través de una estructura auxiliar llamada `HookManager`, que permite registrar múltiples hooks de forma ordenada antes de realizar la instalación efectiva. Esto facilita una instalación masiva coherente y controlada, además de ofrecer una forma sencilla de desinstalar todos los hooks cuando ya no se necesiten (por ejemplo, al descargar la DLL).

En el listado 19 se muestra un fragmento con las definiciones básicas de ambas estructuras.
El uso de la funcion `install_all` permite aplicar todos los hooks registrados en bloque. Esta operación es atómica a nivel lógico: si algún hook falla durante la instalación, se cancela el proceso para evitar un estado inconsistente. De igual forma, `uninstall_all` ofrece una forma segura y limpia de revertir todas las modificaciones realizadas, útil en contextos donde la DLL debe descargarse o se desea restaurar el comportamiento original del proceso.

4.5.5. Lógica de los detours

Los *detours* son funciones que actúan como reemplazos temporales de llamadas al sistema. Su función principal es interceptar una *syscall* real, ejecutar lógica personalizada (por ejemplo, registro o análisis), y luego continuar con la ejecución normal del sistema, asegurando que el

Listado 18: *call_guard.rs CallGuard implementation*

```rust
/// ...
/// RAII object returned by [`CallGuard::enter`]
pub struct CallGuard {
    _private: PhantomData<*const ()>,
}

impl CallGuard {
    /// Try to enter the protected section.
    /// Returns `Some(CallGuard)` on first entry (continue with detour code)
    /// or `None` if this thread is already executing inside the hook.
    #[inline(always)]
    pub fn enter() -> Option<Self> {
        unsafe {
            let idx = *TLS_SLOT;

            // If the slot already holds a non-NULL value, we're re-entering.
            if !TlsGetValue(idx).is_null() {
                return None;
            }

            // First time on this thread - mark as "inside".
            // `TlsSetValue` uses `Option<*const c_void>` (None = NULL).
            if TlsSetValue(idx, Some(SENTINEL)).is_ok() {
                Some(Self {
                    _private: PhantomData,
                })
            } else {
                // Extremely unlikely, but if the slot can't be set just bail
                // out and pretend we are re-entered to avoid recursion loops.
                None
            }
        }
    }
}

impl Drop for CallGuard {
    /// Clears the TLS flag so the thread can re-enter later.
    fn drop(&mut self) {
        unsafe {
            // Ignore failure: worst case the flag remains set and we'll
            // continue short-circuiting to the real API.
            let _ = TlsSetValue(*TLS_SLOT, None);
        }
    }
}
```

proceso original no perciba ningún cambio en su comportamiento.

Desde el punto de vista técnico, cada Detour debe tener exactamente la misma firma que la función que está interceptando: mismos tipos de argumentos, convención de llamada y tipo de retorno. Durante la fase de instalación del hook, se sobrescriben los primeros bytes de la función original con un salto al Detour. A partir de ahí, el control de ejecución fluye a través de nuestro código, donde se realiza la lógica de análisis o bloqueo.

Listado 19: *manager.rs HookEntry and HookManager structures*

```rust
pub struct HookEntry {
    /// Name of the DLL containing the target function.
    pub dll: &'static str,
    /// Name of the function to hook (must exactly match the exported name).
    pub func: &'static str,
    /// Pointer to your detour function (cast to `*const u8`).
    pub detour: *const u8,
    /// Address of a `static mut *const c_void` where we'll store the gateway pointer.
    pub orig_ptr: *mut *const c_void,
}

pub struct HookManager {
    /// A list of all hook specifications you've added but not yet installed.
    spec: Vec<HookEntry>,
    /// Once installed, each `Hook` is stored here so we can later remove it.
    live: Vec<Hook>,
}

pub fn install_all(&mut self) {
    for e in &self.spec {
        unsafe {
            let h = Hook::new(e.dll, e.func, e.detour).unwrap();
            *e.orig_ptr = h.gateway();
            self.live.push(h);
        }
    }
}

pub fn uninstall_all(&mut self) {
    for h in &self.live {
        unsafe {
            h.remove();
        }
    }
    self.live.clear();
}

```

Cada Detour implementado sigue una estructura común y bien definida:

1. Verifica si el hilo actual ya está ejecutando código del hook (usando `CallGuard::enter()`). Si se detecta reentrada, se llama directamente a la función original.
2. Si el acceso es válido, se procede a ejecutar la syscall real, pero encapsulada dentro de una función llamada `call_and_report`. Esta función permite realizar la llamada original y, tras obtener el resultado, generar y enviar un evento al driver con los datos recolectados de la llamada.

El código del listado 20 representa un ejemplo de Detour para `NtReadVirtualMemory`, donde se intercepta la lectura de memoria virtual de otro proceso. Puede verse cómo se usa el guardia de reentrada y cómo se encapsula el reporte del evento.
La función auxiliar `call_and_report` encapsula la ejecución de la syscall real y la generación del evento asociado. Esta separación permite mantener el código limpio y reutilizable para distintos detours. Su logica puede verse en el listado 21.

Listado 20: *detour.rs: Estructura de un detour*

```rust
//
// NtReadVirtualMemory
//
#[unsafe(no_mangle)]
pub unsafe extern "system" fn MyNtReadVirtualMemoryHook(
    process: HANDLE,
    base_addr: *const c_void,
    buffer: *mut c_void,
    size: usize,
    bytes_read: *mut usize,
) -> NTSTATUS {
    let _guard = match CallGuard::enter() {
        Some(g) => g,
        None => {
            let real: NtReadVirtualMemoryFn =
                core::mem::transmute(ORIG_NT_READ_VIRTUAL_MEMORY);
            return real(process, base_addr, buffer, size, bytes_read);
        }
    };

    call_and_report(
        || {
            let real: NtReadVirtualMemoryFn = mem::transmute(ORIG_NT_READ_VIRTUAL_MEMORY);
            real(process, base_addr, buffer, size, bytes_read)
        },
        || {
            Payload::NtReadVirtualMemory(NtReadVirtualMemoryEvt {
                base_address:           base_addr as usize as u64,
                buffer_size:            size as u64,
                bytes_read: if !bytes_read.is_null() { *bytes_read as u64 } else { 0 },
            })
        },
    )
}
```

Listado 21: *detour.rs: Preparacion de eventos*

```rust
unsafe fn call_and_report<R, F, P>(call_real: F, payload_fn: P) -> R
where
    F: FnOnce() -> R,
    P: FnOnce() -> Payload,
{
    let ret = call_real();

    let event = HookEvent {
        pid: GetCurrentProcessId(),
        tid: GetCurrentThreadId(),
        payload: Some(payload_fn()),
    };

    let _ = encode_and_send(&event);
    ret
}
```

4.5.6. Envío de eventos

Para enviar eventos al exterior desde los detours, se utiliza `DeviceIoControl` apuntando a un dispositivo simbólico creado por nuestro driver. Esta llamada permite transferir datos arbitrarios desde Espacio de usuario al kernel de forma síncrona, aprovechando los mecanismos estándar de I/O de Windows.

Los datos que se envían están definidos mediante Protocol Buffers y se serializan usando la librería `Prost`, que ofrece una implementación eficiente en Rust. Este formato permite que el esquema de eventos sea compacto, extensible y fácil de versionar sin romper compatibilidad.

Se mantiene en un `Handle` global al driver, protegido mediante primitivas de sincronización, lo que evita *race conditions* y asegura que sólo se intente abrir el dispositivo una vez. En caso de que el driver no esté cargado o el dispositivo no exista, los eventos simplemente se descartan: no se interrumpe la ejecución del proceso hookeado, lo cual es fundamental para la estabilidad del sistema.

El envío se realiza en una única función, `send_to_driver`, que se encarga de serializar el evento, llamar al `IOCTL`, y manejar posibles errores. Puedes encontrar esta función en el fichero `comms.rs` dentro de la carpeta `Hooking-lib`, o directamente en el listado 22.

Listado 22: *comms.rs: Envio de eventos*

```rust
/// Send `buffer` doum to the driver via IOCTL_SEND_HOOK_EVENT,
/// logging *all* errors along the way.
pub fn send_to_driver(buffer: &[u8]) {

    let device = match get_device_handle() {
        Ok(h) => h,
        Err(e) => return
    };

    let mut bytes_returned = 0u32;
    let ok = unsafe {
        DeviceIoControl(
            device,
            IOCTL_SEND_HOOK_EVENT,
            Some(buffer.as_ptr() as *const c_void),
            buffer.len() as u32,
            None,
            0,
            Some(&mut bytes_returned),
            None,
        )
    };
}
```

El código del listado 22 espera recibir estructuras serializadas con `Prost`, conforme al esquema de mensajes definido en nuestro archivo `events.proto` dentro de la carpeta *shared* en la raiz del proyecto. Un ejemplo de dicho esquema puede verse en el listado 23.

Listado 23: *events.proto: Eventos de Hooking*

```
// Use the oneof payload to record exactly one hook instance per message.
message HookEvent {
  // PID in which the hooked API was called.
  uint32  pid                 = 1;
  // Thread ID that issued the hooked API.
  uint32  tid                 = 2;
  oneof payload {
    // Each sub-message holds parameters logged for its matching NT API.
    NtCreateFileEvt      nt_create_file      = 10;
    NtWriteFileEvt       nt_write_file       = 11;
    NtDeleteFileEvt      nt_delete_file      = 12;
    NtCreateSectionEvt   nt_create_section   = 13;
    NtMapViewSectionEvt  nt_map_view         = 14;
    NtCreateProcessEvt   nt_create_process   = 15;
    NtCreateThreadEvt    nt_create_thread    = 16;
    NtQueueApcEvt        nt_queue_apc        = 17;
    LdrLoadDllEvt        ldr_load_dll        = 18;
    //...
  }
  message NtCreateFileEvt {
      string  path              = 1;
      uint32  desired_access    = 2;   // e.g., FILE_READ_DATA | FILE_WRITE_DATA
      uint32  share_mode        = 3;   // e.g., FILE_SHARE_READ
      uint32  create_disposition= 4;   // FILE_OPEN, FILE_OVERWRITE, FILE_CREATE, etc.
      uint64  allocation_size   = 5;   // Requested file size (0 if not specified)
  }
  //...
}
```

4.5.7. Cómo añadir un nuevo hook

Añadir soporte para interceptar una nueva syscall es un proceso directo y bien definido. El sistema está diseñado para ser extensible, y seguir una estructura uniforme facilita la incorporación de nuevos hooks sin duplicar lógica ni comprometer la estabilidad. Los pasos básicos son los siguientes:

1. Declarar una variable global `static mut ORIG_X` para almacenar la dirección original de la función que se va a interceptar.
2. Añadir al esquema `.proto` un nuevo tipo de evento que refleje los argumentos relevantes de esa syscall.
3. Implementar un Detour con la misma firma que la función original, aplicando el patrón habitual: proteger contra reentradas, capturar argumentos, construir el evento y ejecutar la función real a través del gateway.
4. Registrar la nueva entrada en el hook manager dentro de la función `install_all_hooks()`.

Este patrón se repite para cualquier función del subsistema nativo que se desee interceptar, haciendo que la lógica sea predecible, facil de mantener y segura. Un ejemplo de como implementar un nuevo hook se puede ver en el listado 24, donde se muestra un ejemplo completo de cómo implementar y registrar un nuevo hook genérico llamado `NtXYZ`.

Listado 24: *detours.rs: Registrar un nuevo hook*

```rust
pub static mut ORIG_XYZ:    *const c_void = ptr::null();

type NtXYZFn = extern "system" fn(
    HANDLE,
    *const c_void,
    *mut c_void,
    usize,
    *mut usize,
) -> NTSTATUS;

#[unsafe(no_mangle)]
pub unsafe extern "system" fn NtXYZFn(
) -> NTSTATUS {
    let _guard = match CallGuard::enter() {
        Some(g) => g,
        None => {
            let real: NtXYZFn =
                core::mem::transmute(ORIG_XYZ);
            return real(...);
        }
    };

    call_and_report(
        || {
            let real: NtXYZFn =
                core::mem::transmute(ORIG_XYZ);
            return real(...);
        },
        || {
            Payload::NtXYZFn(NtXYZEvt {
                ...
            })
        },
    )
}

pub fn install_all_hooks() -> Result<(), String> {
    let mut mgr = HookManager::new();
    mgr.add(HookEntry {
        dll:      "ntdll.dll",
        func:     "NtXYZ",
        detour:   NtXYZFn as *const u8,
        orig_ptr: unsafe { &raw mut ORIG_XYZ },
    });
    //...

    mgr.install_all();
    HOOK_MANAGER
        .set(Mutex::new(mgr))
        .map_err(|_| "hooks already installed".to_owned())?;

    Ok(())
}
```

4.5.8. Resumen del sistema de hooking

Para aclarar un poco todo lo que hemos visto, dejo un resumen de todo el proceso:

1. Cuando el sistema carga la DLL, se ejecuta `DllMain` y se lanza un nuevo hilo para evitar permanecer dentro del `loader lock`.
2. Ese hilo llama a `install_all()` del `HookManager`, que registra e instala todos los hooks especificados sobre las funciones `Nt` registradas.
3. Cada función hookeada es sobrescrita con una instrucción de salto a un Detour específico. Ese Detour es una función que tiene la misma firma que la syscall original.
4. Al ejecutarse una syscall interceptada, el flujo de ejecución entra en el Detour, donde se activa un `CallGuard` que ejerce de zona critica, la cual protege contra reentradas.
5. Dentro del Detour se recogen los argumentos relevantes y se empaquetan en un mensaje Protocol Buffers serializado.
6. Ese mensaje se envía mediante `DeviceIoControl` al driver utilizando un IOCTL personalizado.
7. Finalmente, el Detour llama a la syscall original usando un bloque `gateway` que replica los bytes sobrescritos y realiza la instrucción de syscall correspondiente.
8. Cuando se descarga la DLL, se llama a `uninstall_all()` que revierte los hooks y libera los recursos.

Esto no parece fácil, y de hecho, no lo es. Durante la implementación me he encontrado con varios inconvenientes y he hecho y rehecho varias versiones de lo mismo para acabar de depurar los problemas. En al siguiente sección explicaré el mayor problema que me he encontrado a la hora de implementar esta funcionalidad.

4.5.9. Problemas derivados del Heap Manager de Windows

Durante la fase de implementación, me encontré con un fallo fatal al inyectar la DLL en un proceso de Windows (en este caso, `notepad.exe`): una excepción del tipo `0xC0000409`, también conocida como `FAST_FAIL_UNEXPECTED_HEAP_EXCEPTION`. Este error es especialmente grave porque no permite recuperación: el proceso se termina inmediatamente.

El objetivo de la instrumentación era interceptar llamadas sensibles como `NtAllocateVirtualMemoryEx`, registrar sus parámetros y enviarlos a un driver en modo kernel (usando `DeviceIoControl`) para su análisis.

Técnicamente, todo parecía funcionar: el hook se instalaba correctamente, capturaba los argumentos de la syscall y enviaba los informes. Sin embargo, el proceso se cerraba abruptamente tras algunas llamadas. Para entender qué ocurría, utilicé `procdump.exe` de Sysinternals y el depurador WinDbg.

Dónde se rompe realmente Windows

El fallo ocurre al interceptar `NtAllocateVirtualMemoryEx` en momentos donde el sistema ya está utilizando esta función internamente, particularmente dentro del *heap manager* (por ejemplo, en funciones como `RtlpHp*` o `RtlAllocateHeap`). En estos casos, el heap puede no estar completamente consistente: puede haber locks activos, estructuras parcialmente inicializadas o buffers en transición.

Aside: Heap manager y llamadas sensibles. En fases tempranas de inicialización el *Segment Heap*/LFH pueden estar mutando estructuras internas. Interceptar o forzar reservas (`NtAllocateVirtualMemory(Ex` dentro de esas ventanas puede amplificar condiciones de carrera o disparar aserciones del gestor de heap. La DLL evita trabajo pesado en ese punto y traslada la lógica al hilo auxiliar.

Interceptar esta syscall en ese momento e introducir lógica adicional —como enviar datos al driver, formatear logs o acceder a más memoria— puede causar recursión o accesos prematuros al heap. El heap manager detecta esto como una posible corrupción y activa el mecanismo de *Fast Fail*, una medida de protección diseñada para impedir explotación.

Un ejemplo de traza en WinDbg al interceptar esta syscall muestra lo que se puede ver en el listado 25.

Listado 25: *call stack del crash de notepad.exe en WinDbg*

```
ntdll!RtlpHeapFatalExceptionFilter+0x11
ntdll!RtlAllocateHeap$filt$0+0x16
ntdll!_C_specific_handler+0x93
ntdll!RtlpExecuteHandlerForException+0xf
ntdll!RtlDispatchException+0x2c8
ntdll!KiUserExceptionDispatch+0x2e
hooking_lib!MyNtAllocateVirtualMemoryExHook+0x73
ntdll!RtlpHpEnvAllocVA+0xe6
ntdll!RtlpHpSegMgrCommit+0x235
ntdll!RtlpHpSegPageRangeCommit+0x245
ntdll!RtlpHpLfhSubsegmentCreate+0x25e
```

Como se puede observar en el listado 25, nuestra función hook se ejecuta en medio de una operación interna crítica del heap. Cualquier acción extra, por mínima que sea, resulta en un uso inseguro del heap y provoca el fallo.

Tras identificar esto, intenté minimizar el impacto eliminando el uso de asignaciones dinámicas en el hook: evitar estructuras como `Vec`, `String`, y cualquier contenedor que implique reserva de memoria. Aun así, no fue posible garantizar la estabilidad, ya que incluso funciones aparentemente simples como `DeviceIoControl` pueden invocar código del CRT o NT runtime que dependen del heap.

Decisión tomada

Como no es posible garantizar un entorno seguro al interceptar `NtAllocateVirtualMemoryEx`, decidimos excluir esta syscall (y otras similares) del sistema de instrumentación. Una alternativa sería rediseñar la lógica para ejecutar el análisis de forma asíncrona (por ejemplo, en otro hilo o proceso), pero eso implicaría perder contexto y sincronización.

En su lugar, opté por centrarme en otras syscalls del Espacio de usuario como `NtWriteFile`, `NtCreateFile`, `NtProtectVirtualMemory` y `NtCreateThreadEx`, cuyo uso es más predecible y menos propenso a ocurrir en contextos críticos del sistema.

4.5.10. Limitaciones y trabajo futuro

Como resultado de las decisiones de diseño y los problemas encontrados, esta funcionalidad tiene actualmente las siguientes limitaciones:

- No hay soporte para procesos WOW64 (ejecución de binarios x86 en sistemas x64).
- La comunicación con el driver no está cifrada.
- Si el driver no está presente, los eventos se descartan sin aviso.
- Algunas syscalls no pueden interceptarse de forma segura porque se usan en contextos internos críticos (como el heap manager o el loader).

En un futuro, se podría implementar un *ring-buffer* para solventar algunos de estos problemas. Esta solución no se ha desarrollado aún, ya que el objetivo del proyecto era experimentar con nuevos métodos; recordemos que ya se implementó una sección de memoria compartida entre el agente y el driver.

4.6. Callbacks del núcleo y eventos de sistema

Además de los eventos generados por técnicas activas como hooks o escaneos, el agente también recopila información de bajo nivel proporcionada directamente por el sistema operativo mediante callbacks del núcleo. Estos mecanismos permiten interceptar eventos relevantes como la creación de procesos, la carga de módulos o cambios persistentes en el registro sin necesidad de instrumentar manualmente cada función implicada.

El agente registra hasta tres callbacks distintos durante su inicialización: uno para la creación de procesos (`PsSetCreateProcessNotifyRoutineEx`), otro para la carga de imágenes (`PsSetLoadImageNotifyRoutine`) y un tercero para modificaciones del registro (`CmRegisterCallbackEx`). Cada uno de ellos está encapsulado en un módulo independiente y filtrado mediante políticas específicas que reducen el ruido sin perder eventos de interés para la detección.

Esta sección describe cómo se implementa cada uno de estos callbacks, qué información recoge y cómo se integra en el sistema de eventos del agente. También se aborda un problema crítico encontrado durante su desarrollo: la imposibilidad de descargar el driver mientras existían callbacks activos, y cómo se resolvió añadiendo una fase explícita de desregistro controlado mediante IOCTL.

4.6.1. Implementación de callbacks para eventos del sistema

Durante la fase de arranque, el driver registra de forma centralizada los tres callbacks principales del sistema operativo: uno para la creación de procesos, otro para la carga de imágenes, y un tercero —opcional y actualmente deshabilitado— para operaciones sobre el registro. La función `register_all()` del módulo `callbacks` orquesta este proceso de forma transaccional: si algún registro falla, los ya activados se desregistran y se limpia el puntero global de estado, evitando así estados inconsistentes.

Cada callback se implementa en un módulo dedicado que encapsula la lógica de conversión, filtrado y publicación del evento. En el caso del proceso de creación, se usa la API `PsSetCreateProcessNotifyRoutine` para interceptar la fase de inicialización de un nuevo proceso. Se extrae información como el PID, el PPID, la ruta de imagen y la línea de comandos. Por su parte, el callback de carga de

imágenes emplea `PsSetLoadImageNotifyRoutine` y recoge eventos cuando se mapean binarios (DLLs o ejecutables) en el espacio de direcciones de un proceso.

El tercer callback, basado en `CmRegisterCallbackEx`, permite interceptar operaciones como `RegSetValue`. Dado el gran volumen de eventos generados en rutas del registro poco relevantes, se aplica una política de filtrado que conserva únicamente los cambios significativos para la persistencia, configuración de servicios o técnicas comunes de ataque. No obstante, debido a problemas de estabilidad y carga —como se describirá más adelante— este callback ha sido temporalmente desactivado en la configuración por defecto del agente.

4.6.2. Tipos de eventos gestionados

El sistema registra tres callbacks principales que permiten capturar eventos críticos generados por el propio sistema operativo. Cada uno de ellos se implementa como un módulo autónomo dentro del controlador y está diseñado para ejecutar una mínima lógica de filtrado y construcción del evento, antes de publicarlo en el canal compartido. Esta separación permite un diseño modular, seguro y fácil de extender. A continuación se detalla el funcionamiento de cada uno de estos callbacks.

Carga de imágenes (`PsSetLoadImageNotifyRoutine`)

Este callback permite interceptar cada carga de imagen en el sistema —ya sea un ejecutable o una biblioteca dinámica (DLL)— e identificar el proceso en el que se ha producido, su ruta absoluta, la dirección base y el tamaño mapeado en memoria. Se trata de un mecanismo pasivo pero muy útil para detectar side-loading, módulos inyectados o bibliotecas residentes fuera de rutas estándar.

El controlador registra esta rutina mediante una llamada a `PsSetLoadImageNotifyRoutine` durante la inicialización. La función `load_image_notify` es invocada por el sistema con los parámetros relevantes, y tras extraerlos, el evento es encapsulado en una estructura `ImageLoadEvent` y serializado para su envío al espacio de usuario.

Dado que este callback se dispara con mucha frecuencia —especialmente durante el arranque del sistema o en procesos intensivos—, se aplican filtros específicos para evitar eventos irrelevantes. Se descartan, por ejemplo, cargas en el proceso PID 4 (System), imágenes menores de 4 KiB, y módulos ubicados en rutas como `WinSxS` o `DriverStore`. Estas heurísticas permiten conservar alta señal sin saturar el canal compartido ni la base de datos de eventos.
Este tipo de eventos complementa de forma directa a otros mecanismos de monitorización como los hooks o las reglas YARA.

En la siguiente subsubsección se describe el segundo tipo de callback implementado: la creación de procesos mediante `PsSetCreateProcessNotifyRoutineEx`, esencial para rastrear la actividad y el árbol de ejecución del sistema.

Creación de procesos (`PsSetCreateProcessNotifyRoutineEx`)

El segundo callback que se registra es el de creación de procesos, usando la API `PsSetCreateProcessNotifyRoutine`
Su propósito es interceptar cada nueva instancia de proceso en el sistema y extraer información clave como el PID, el PPID, la ruta del ejecutable y la línea de comandos usada. Este tipo de

Listado 26: *Callback de carga de imágenes*

```
unsafe extern "C" fn load_image_notify(
    full_image_name: *mut UNICODE_STRING,
    process_id: HANDLE,
    image_info: *mut IMAGE_INFO,
) {
    if full_image_name.is_null() || image_info.is_null() {
        return;
    }

    let path = uni_to_string(&*full_image_name);
    let info = &*image_info;
    let pid = process_id as u32;

    if !should_emit_image_load(pid, &path, info.ImageSize) {
        return;
    }

    let evt = ImageLoadEvent {
        image_base: info.ImageBase as u64,
        image_size: info.ImageSize as u32,
        full_image_name: path,
        process_id: pid,
    };
    push_event(evt.into());
}
```

eventos es especialmente útil para detectar ejecución de binarios sospechosos, hijacking de procesos legítimos o análisis del árbol de procesos durante una intrusión.

La implementación del callback es sencilla. El kernel invoca la función `process_notify` con una estructura `PS_CREATE_NOTIFY_INFO` cuando un proceso es creado. Si el puntero es nulo, se trata de una notificación de salida, que el sistema ignora por diseño para centrarse solo en los eventos de creación.

Al igual que en el caso anterior, se aplican filtros para evitar ruido excesivo: se descartan procesos con PID muy bajo (normalmente servicios del sistema) y aquellos sin ruta de imagen. El resto se encapsula en un `ProcessEvent` y se envía al canal compartido.
Este tipo de eventos aporta un contexto temporal muy valioso: saber qué procesos se están ejecutando, en qué orden y bajo qué padres permite detectar técnicas como el proceso hollowing, el uso de binarios LOLBin o patrones típicos de ejecución lateral.

En la siguiente subsubsección se aborda el tercer callback soportado por el sistema: la monitorización de cambios en el registro, una fuente común de persistencia y manipulación de configuraciones.

Modificaciones en el registro (`CmRegisterCallbackEx`)

El tercer y último callback que implementa el sistema es el de cambios en el registro de Windows, usando la función `CmRegisterCallbackEx`. Esta API permite interceptar operaciones críticas como la modificación de claves o valores dentro del registro, lo cual es especialmente

Listado 27: *Callback de creación de procesos*

```rust
unsafe extern "C" fn process_notify(
    process: PEPROCESS,
    _pid: HANDLE,
    info_ptr: *mut PS_CREATE_NOTIFY_INFO,
) {
    if info_ptr.is_null() {
        return; // Evento de salida: no se registra
    }

    let info = &*info_ptr;
    let pid = PsGetProcessId(process) as u32;
    let ppid = info.ParentProcessId as u32;
    let image_path = uni_to_string(info.ImageFileName);
    let cmdline = uni_to_string(info.CommandLine);

    if !should_emit_process_create(pid, &image_path) {
        return;
    }

    let evt = ProcessEvent {
        pid,
        ppid,
        image_path,
        cmdline,
    };
    push_event(evt.into());
}
```

relevante para detectar técnicas de persistencia, cambios en configuraciones de seguridad o manipulación de opciones de ejecución automática.

En nuestro caso, el sistema solo registra un tipo específico de evento: `RegNtPreSetValueKey`, es decir, operaciones que modifican valores justo antes de que el cambio se refleje en el registro. Se ignoran otros tipos de eventos para reducir el volumen de datos y centrarse en aquellos más representativos desde el punto de vista de la detección.

El siguiente fragmento muestra la lógica principal del callback. Primero se determina el tipo de evento recibido; si no es un `PreSetValue`, se descarta. Luego se resuelve el nombre de la clave modificada a partir del handle de objeto (ya que la estructura no lo incluye directamente) y se recupera el nombre del valor y los datos nuevos. Finalmente, se aplica una política de filtrado similar a la de otros callbacks antes de emitir el evento.

Gracias a este mecanismo es posible detectar manipulaciones en rutas sensibles como las claves de autorun, configuraciones de servicios, o ediciones de `IFEO` y `LSA`. No obstante, también se observó que este callback generaba una cantidad considerable de eventos incluso en sistemas inactivos, por lo que se optó por deshabilitarlo temporalmente en la versión actual. Este punto se retoma en detalle en las siguientes secciones, donde se analiza su impacto y cómo se resolvieron los problemas asociados a su descarga.

Listado 28: *Callback de escritura en el registro*

```
unsafe extern "C" fn registry_callback(
    _context: PVOID,
    notify_ptr: *mut c_void,
    argument: *mut c_void,
) -> NTSTATUS {
    let notify_class = *(notify_ptr as *const REG_NOTIFY_CLASS);
    if notify_class != REG_NOTIFY_CLASS::RegNtPreSetValueKey {
        return STATUS_SUCCESS;
    }

    let info = &*(argument as *const REG_SET_VALUE_KEY_INFORMATION);
    let key_path = query_object_name(info.Object)
        .unwrap_or_else(|_| "<unknown>".into());
    let value_name = uni_to_string(info.ValueName);
    let pid = PsGetCurrentProcessId() as u32;

    if !should_emit_registry_setvalue(&key_path,
        if value_name.is_empty() { None } else { Some(&value_name) }) {
        return STATUS_SUCCESS;
    }

    let new_buf = if info.DataSize > 0 && !info.Data.is_null() {
        slice::from_raw_parts(info.Data as *const u8, info.DataSize as usize).to_vec()
    } else {
        Vec::new()
    };

    let evt = RegistryEvent {
        key_path,
        op_type: OperationType::Modify as i32,
        old_value: Vec::new(),
        new_value: new_buf,
        process_id: pid,
    };
    push_event(evt.into());

    STATUS_SUCCESS
}
```

4.6.3. Gestión de callbacks y problemas durante la descarga del driver

Durante las pruebas con el subsistema de callbacks se detectó un problema crítico que afectaba a la estabilidad y capacidad de mantenimiento del sistema: una vez registrados ciertos callbacks del kernel, el driver ya no podía descargarse de forma limpia. A pesar de que las funciones de desregistro estaban implementadas correctamente, el sistema operativo mantenía referencias internas al código del driver que impedían su retirada.

Este comportamiento no solo resultaba molesto durante el ciclo de desarrollo —al impedir recargar el driver sin reiniciar—, sino que también comprometía el despliegue en entornos productivos, donde era necesario tener un mecanismo seguro para detener o actualizar el agente sin requerir un reinicio del sistema.

A continuación, se detalla la causa técnica de este comportamiento y la solución adoptada para

garantizar una descarga segura y controlada.

Persistencia de referencias internas a callbacks registrados

El origen del problema reside en el comportamiento interno del sistema operativo cuando se registran ciertos tipos de callbacks desde un driver en modo kernel. En particular, funciones como `PsSetCreateProcessNotifyRoutineEx`, `PsSetLoadImageNotifyRoutine` o `CmRegisterCallbackEx` instalan punteros a funciones dentro del espacio del driver, y el propio kernel mantiene referencias a estos punteros mientras los callbacks están activos.

Esto significa que, aunque el driver intente descargarse —por ejemplo, a través de `ZwUnloadDriver`—, Windows detecta que aún existen referencias activas a direcciones dentro de su código, y rechaza la operación con errores como `STATUS_IMAGE_STILL_LOADED`. Incluso si se invocan correctamente las funciones de desregistro durante la rutina de descarga, las referencias internas no se liberan de inmediato, especialmente si existen eventos pendientes en curso o si no se ha completado un rundown"de todos los callbacks.

Este comportamiento no es un error del sistema operativo, sino una medida de protección diseñada para evitar el uso de direcciones inválidas tras la descarga de un módulo. Sin embargo, en escenarios como el desarrollo iterativo del driver, donde se desea recargar versiones sucesivas sin reiniciar el sistema, esta protección se convierte en un obstáculo práctico.

Para resolver este bloqueo, se implementó un mecanismo de parada controlada de los callbacks antes de permitir la descarga, tal como se describe en la siguiente sección.

Solución implementada: parada controlada de callbacks

Una vez identificado el problema, se implementaron dos mecanismos complementarios para permitir la descarga segura del driver: un sistema de protección en tiempo de ejecución y una interfaz de control mediante IOCTL. Ambos trabajan juntos para garantizar que no se produzcan carreras ni referencias colgantes durante el proceso de descarga.
La primera pieza clave fue el sistema de **protección en tiempo de ejecución** (*runtime protection*), que garantiza que ningún callback se esté ejecutando en paralelo cuando se intentan desregistrar. Para ello, cada rutina de callback entra en una sección crítica protegida mediante contadores de uso (`rundown protection`) antes de comenzar su ejecución. Esto asegura que, al iniciar la secuencia de parada, el sistema pueda esperar de forma segura a que terminen todos los callbacks activos antes de continuar con la descarga del módulo.
Además de esto, se introdujo un **IOCTL específico** que permite al espacio de usuario solicitar la parada explícita de los callbacks antes de invocar `ZwUnloadDriver`. Esta parada llama internamente a la función `unregister_all()`, encargada de desregistrar todas las familias de callbacks activas y limpiar el puntero global a la estructura de anillo compartido.

El flujo correcto para descargar el driver desde espacio de usuario queda así:

1. Se llama a `IOCTL_StopCallbacks` para detener todos los callbacks activos y bloquear la entrada de nuevos eventos.
2. Se espera a que finalicen todos los callbacks en ejecución gracias a la protección en tiempo de ejecución.
3. Se invoca `ZwUnloadDriver()` para descargar el módulo sin bloqueos ni referencias activas.

Con esta solución, se elimina el riesgo de que el subsistema de eventos mantenga referencias vivas al código del driver, lo que previamente impedía su descarga o incluso provocaba que el sistema no pudiese reiniciarse correctamente. Además, el uso combinado de IOCTL + runtime protection permite extender fácilmente este modelo a otros subsistemas críticos del agente.

5
Evaluación del Sistema

En este capítulo se evalúa el funcionamiento del sistema EDR desarrollado, centrándose en tres aspectos clave: la capacidad para detectar eventos relevantes, el impacto en el rendimiento del sistema y las principales limitaciones encontradas durante las pruebas.

El objetivo de esta evaluación no es comparar el sistema con soluciones comerciales, sino validar que los distintos sensores funcionan como se espera, que los eventos se capturan y procesan correctamente, y que la arquitectura general es estable incluso bajo condiciones moderadas de carga.

Para ello, se han diseñado varios escenarios simulados que reproducen técnicas de ataque comunes, incluyendo el uso de binarios legítimos con fines maliciosos, la ejecución de código desde PowerShell, o la manipulación de archivos sospechosos. Asimismo, se ha monitorizado el consumo de recursos del agente y se han documentado los principales retos y limitaciones técnicas observadas.

5.1. Entorno de pruebas

Las pruebas se realizaron en un entorno controlado para garantizar aislamiento y reproducibilidad. Se utilizó una máquina virtual con Windows 11, configurada con X núcleos y Y GB de RAM, ejecutándose sobre [VMware/VirtualBox]. La red estuvo desactivada durante toda la evaluación para evitar fugas de información y mantener el comportamiento del malware contenido.

Para cada muestra, se ejecutó el sistema completo: el driver cargado en el kernel, el agente en userland, y la base de datos SQLite como backend de persistencia. En paralelo, se utilizó `ProcMon` para capturar eventos del sistema a bajo nivel y servir como punto de comparación. Las pruebas del sistema se realizaron apoyándome en herramientas como WinDbg Microsoft, 2025 y la suite Sysinternals **sysinternals2025**, que facilitaron el análisis de la telemetría y la validación de los eventos capturados.

Las muestras utilizadas incluían binarios reales (cuando fue legalmente posible) o simulaciones construidas para replicar técnicas maliciosas comunes como:

- Inyección de código mediante `NtCreateThreadEx`.
- Mapeo de regiones sospechosas con `NtMapViewOfSection`.
- Escritura en claves del registro para persistencia.
- Modificación de archivos o ejecución desde rutas temporales.

Durante cada ejecución, se recogió toda la telemetría generada por el agente, que fue almacenada en la base de datos y posteriormente analizada. Los logs de ProcMon fueron exportados para su revisión manual y utilizados como referencia para contrastar los eventos capturados por el sistema.

Los eventos capturados se almacenaron en una base de datos SQLite y fueron inspeccionados mediante la interfaz de SQLiteBrowser.

5.2. Pruebas funcionales

En esta sección se pone a prueba la cobertura de eventos de nuestro sistema frente a dos escenarios: un ejecutable de prueba diseñado ad-hoc y una muestra de malware real. En ambos casos, comparamos la telemetría que almacena nuestro agente con la que captura ProcMon, para verificar su fidelidad y alcance.

5.2.1. Proceso de prueba instrumentado

Se ha compilado un pequeño programa en Rust que realiza, paso a paso, llamadas clave de Windows: reserva memoria con `VirtualAlloc`, cambia permisos con `VirtualProtect`, crea o abre un fichero y escribe datos mediante `WriteFile`. Al ejecutarlo, pulsando ENTER en cada etapa, comprobamos que el agente interceptaba correctamente las llamadas a `NtCreateThreadEx`, `NtMapViewOfSection`, `NtProtectVirtualMemory` y `NtSetValueKey`. El código entero de este programa ad-hoc lo podeis encontrar en el listado 30, en el apendice.

Lo que se hizo con este ejecutable fue forzar la inyección manual de la DLL de hooking (cuyo nombre es `hooking_lib.dll`) usando herramientas como Process Explorer. De esta forma pudimos comprobar que tanto la carga de la librería como la inicialización de los hooks quedan reflejadas correctamente en la base de datos.La Figura 5.1 muestra el momento en que, desde Process Explorer, se inyecta manualmente `hooking_lib.dll` en `test_process.exe`.

A continuación, la Figura 5.2 presenta el registro correspondiente en la tabla `image_load_event`, donde se captura la ruta completa de la DLL y el PID del proceso receptor.

Además, se consultó la tabla `hook_event` y se comprobó que todas las funciones interceptadas por la DLL —como `NtCreateThreadEx` o `NtMapViewOfSection`— aparecieran con su marca de tiempo y PID correctos. Esto confirma que el sistema no solo detecta cargas de librerías, sino que arranca los hooks asociados inmediatamente tras la carga.

En la base de datos SQLite, cada una de esas llamadas quedó registrada en `hook_event`, con su PID, TID, tipo de payload y marca de tiempo. La Figura 5.3 muestra un fragmento de la tabla tras varias invocaciones a `VirtualProtect`, confirmando que no se perdió ninguna llamada.

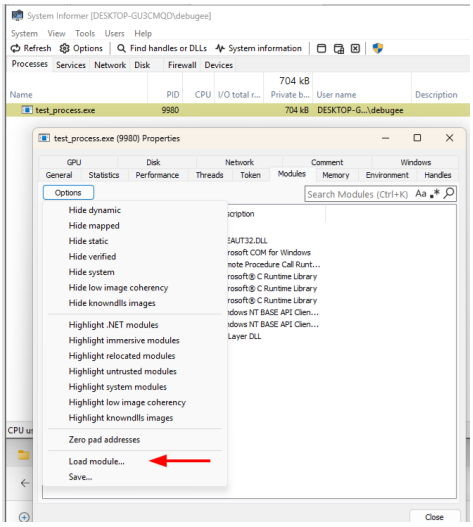

Figura 5.1: *Inyección manual de* `hooking_lib.dll` *en* `test_process.exe` *con Process Explorer.*

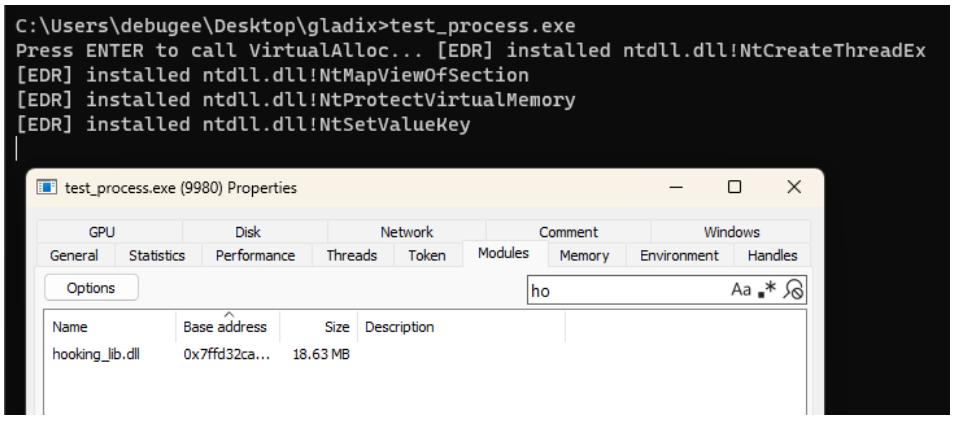

Figura 5.2: *Evento de carga de imagen registrado en* `image_load_event` *tras la inyección manual.*

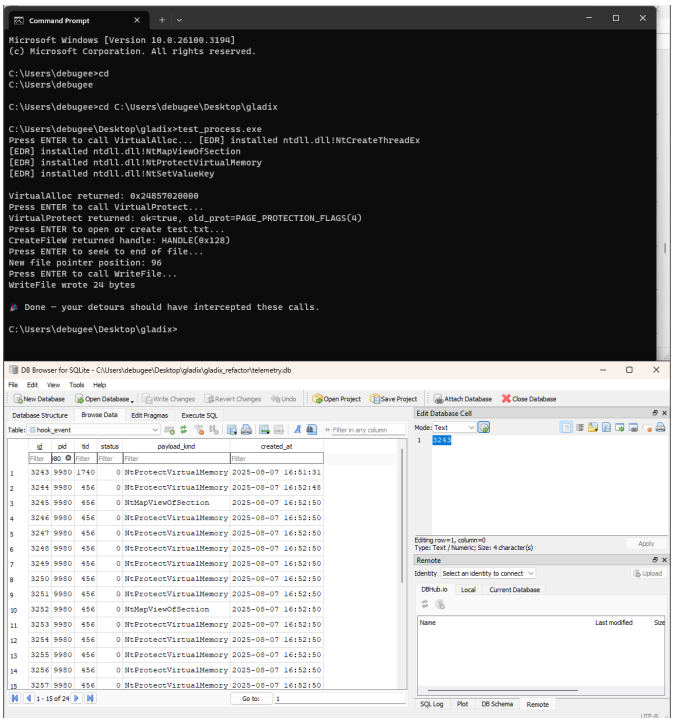

Figura 5.3: *Registros de `NtProtectVirtualMemory` en `hook_event`.*

5.2.2. Ejecución de muestra sospechosa

A continuación se ha ejecutado un binario malicioso con hash: `6dda005fa9d3f826124458af97d2e918a475d83447b2e057a3b0057441c3d6a7`.

Durante su arranque, se cargaron decenas de librerías de sistema y posibles payloads. Tanto ProcMon como nuestro agente registraron los eventos de carga de imagen (`Load Image`) y creación de proceso (`Process Start`).

La Figura 5.4 compara la lista de "Load Image" de ProcMon (arriba) con la tabla `image_load_event` en SQLite (abajo): se observa que el agente recoge las mismas rutas completas de DLLs, incluyendo `ntdll.dll`, `kernel32.dll` y `hooking_lib.dll` cuando se inyectó manualmente.

Por último, la creación del proceso principal quedó reflejada en `process_event` con la ruta completa del ejecutable malicioso y su PPID. La Figura 5.5 muestra ese registro, validando que no se omitió ningún evento de inicio.

En conjunto, estas pruebas muestran que el agente captura de forma completa:

- Todos los eventos de reservación y protección de memoria generados por el proceso de prueba.
- La creación de archivos y llamadas a `NtWriteFile` mediante hooks en `WriteFile`.
- La carga de imágenes tanto en el malware real como en la DLL inyectada manualmente.
- La creación de procesos con toda la metadata asociada (PID, PPID, ruta).

Comparado con ProcMon, el sistema recupera la misma telemetría esencial de "Process Start" y "Load Image" y añade la detección de hooks específicos de API. No se observaron omisio-

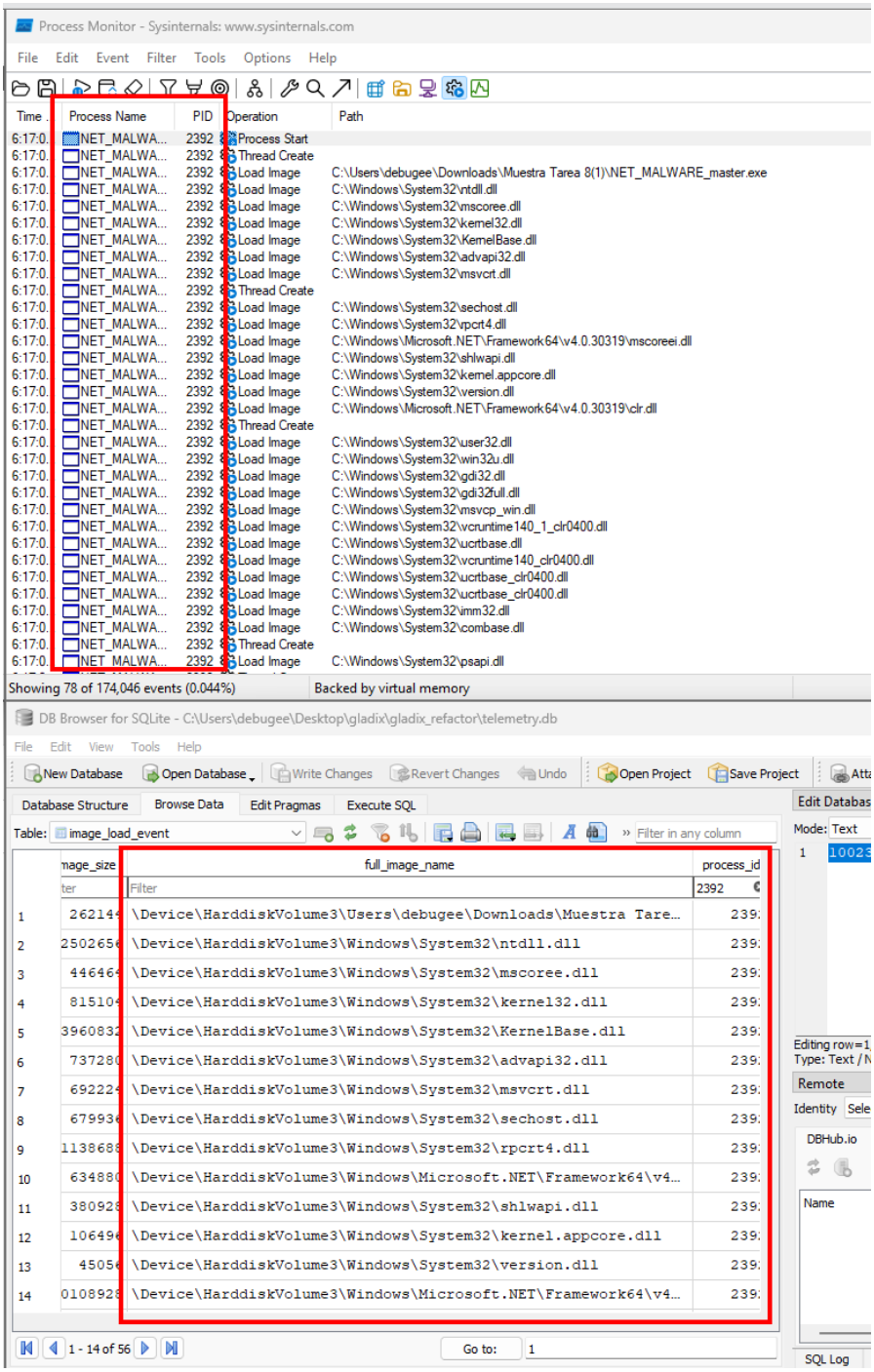

Figura 5.4: *Carga de imágenes: ProcMon (arriba) vs. base de datos (abajo).*

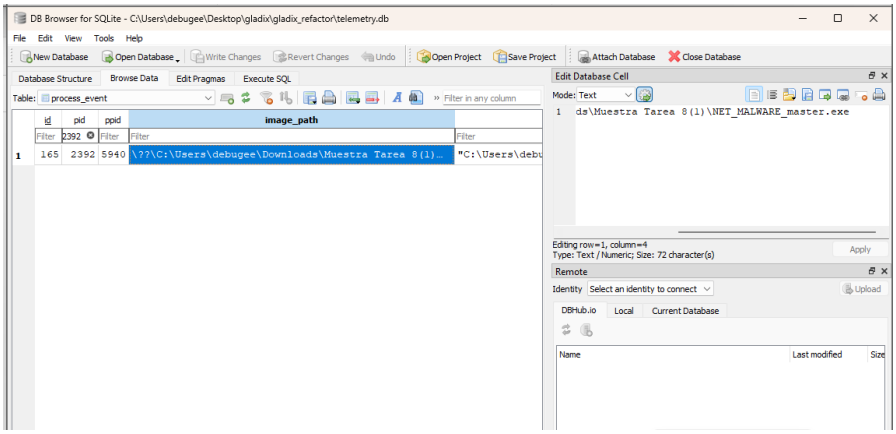

Figura 5.5: *Evento de creación de proceso registrado en* `process_event`.

nes en ninguno de los escenarios, aunque sí se detectó que eventos de red y acceso a archivos protegidos quedan pendientes de implementación por falta de sensores dedicados.

5.3. Consumo de recursos

Durante las pruebas se monitorizó el consumo de recursos del agente en modo usuario (`user-agent.exe`) y del driver en kernel. El objetivo era confirmar que el sistema puede operar de forma continua sin degradar el rendimiento del sistema o consumir recursos excesivos, incluso bajo carga moderada.

En condiciones normales de espera, el agente mantuvo un uso de CPU prácticamente nulo (menor al 0.1 %) y un consumo de memoria estable en torno a los 100 MB. El uso de recursos aumentó temporalmente durante fases intensivas como el escaneo de archivos y la compilación de reglas YARA. A continuación se resumen las métricas más relevantes obtenidas:

- **CPU media:** 0.02 % (con picos puntuales de hasta 49 % durante análisis).
- **Memoria privada:** 106 MB en reposo; pico observado de 329 MB durante escaneo.
- **Fallos de página:** ~255,000, con ~2,000 fallos duros (nivel razonable).
- **I/O:** 4,781 escrituras, totalizando unos 9.5 MB, correspondientes al volcado de eventos.
- **Handles abiertos:** 63, sin fugas ni crecimiento irregular.

La Figura 5.6 muestra una captura temporal del consumo durante un escaneo completo. Se distinguen varias fases claramente:

1. En la parte izquierda del gráfico se observa el inicio del análisis, donde el uso de CPU se eleva rápidamente a un 49 % sostenido, correspondiente a la lectura de archivos y evaluación de reglas YARA.
2. El pico de memoria (318 MB) se alcanza poco después, coincidiendo con la compilación de reglas y carga de módulos auxiliares.
3. Finalmente, la actividad de E/S aumenta (hasta 5 MB), indicando la escritura masiva de eventos detectados a la base de datos.

En paralelo, el driver en modo kernel se comportó de forma estable, sin impacto observable en uso de CPU o memoria. Al no realizar tareas complejas (más allá de capturar eventos y escribir

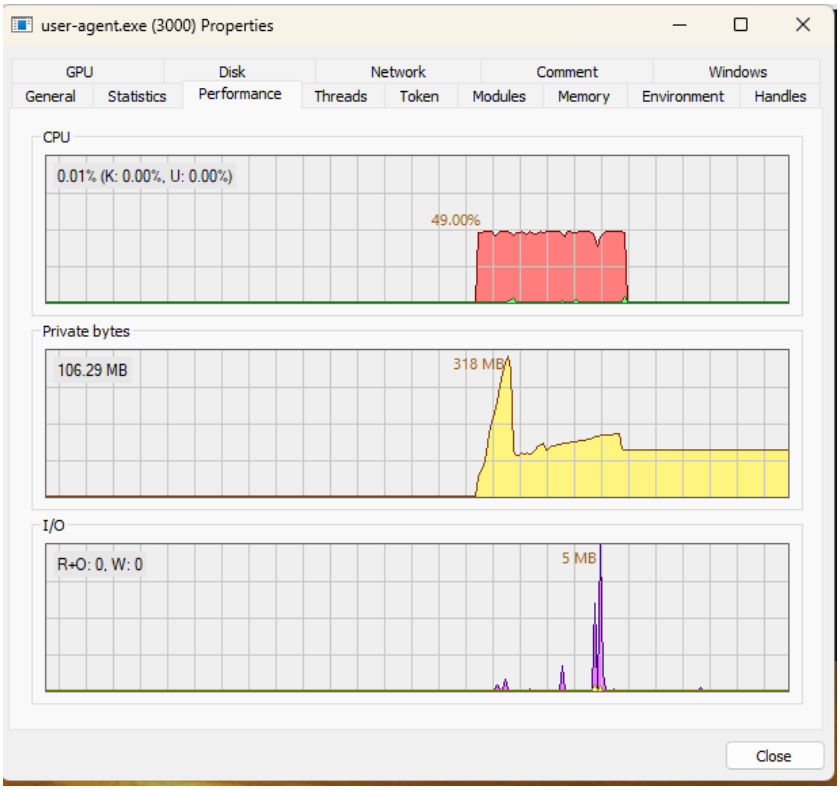

Figura 5.6: *Evolución del consumo de recursos durante un escaneo completo.*

en el buffer compartido), su huella en el sistema fue mínima.

Estos resultados demuestran que el sistema es capaz de mantener una operación continua con una huella de recursos aceptable, incluso bajo fases intensivas de análisis o escritura.

5.4. Limitaciones observadas

Durante las pruebas realizadas, el sistema funcionó correctamente y capturó los eventos esperados, pero también se identificaron varias limitaciones prácticas que conviene destacar. Estas observaciones no invalidan el funcionamiento general del EDR, pero sí evidencian áreas donde hay margen de mejora:

- **Paso intermedio por el kernel:** Los eventos interceptados por la DLL en userland no se envían directamente al agente, sino que pasan primero por el driver mediante una llamada `IOCTL`. Este paso adicional introduce cierto overhead innecesario, especialmente cuando se genera una gran cantidad de eventos en poco tiempo.
- **Inicio lento debido a reglas YARA:** En el arranque del agente, todas las reglas YARA se compilan en memoria. Al tener un número elevado de reglas o expresiones complejas, este proceso puede tardar varios segundos, retrasando la activación completa del sistema.
- **Escaneos sin control de frecuencia:** Actualmente, cada vez que se detecta una escritura a disco, se lanza un escaneo con YARA. No existe ningún tipo de *backoff* o mecanismo para agrupar eventos similares, lo que puede provocar un uso intensivo de CPU si el sistema está muy activo a nivel de I/O.
- **Cobertura parcial del registro de Windows:** Aunque se interceptan llamadas a `NtSetValueKey`, se observó que no todos los accesos al registro pasaban por el hook. Es probable que al-

gunas funciones usen rutas alternativas o que ciertas llamadas queden fuera del alcance de la instrumentación actual.

- **Consumo puntual de recursos:** Durante los escaneos intensivos, el agente alcanzó picos de uso de CPU cercanos al 50 % y más de 300 MB de memoria privada, como se ve en la Figura 5.6. Aunque estos valores se estabilizan después, pueden ser problemáticos en entornos con recursos limitados o con múltiples procesos siendo monitorizados al mismo tiempo.

Todas estas limitaciones están relacionadas con decisiones de diseño propias de un prototipo en fase experimental. Aun así, sirven como guía clara para futuras optimizaciones en rendimiento, cobertura y escalabilidad.

6
Conclusiones y Trabajo Futuro

Con este capítulo cerramos el recorrido del proyecto. A lo largo de todo el trabajo hemos ido construyendo paso a paso un prototipo de EDR para Windows, desde los primeros conceptos hasta tener un sistema que ya es capaz de observar y reaccionar ante ciertos eventos en el equipo. Más allá del código, ha sido un ejercicio de exploración: probar, equivocarse, ajustar y volver a intentarlo hasta dar con una solución estable.

El objetivo aquí no es entrar en los detalles técnicos, sino hacer balance. Primero se repasarán las aportaciones más importantes, es decir, qué partes han funcionado bien y qué se ha conseguido poner en marcha. Después se comentarán las posibles líneas de mejora y las ideas de futuro que podrían dar más solidez y amplitud al sistema. Por último, se cerrará con una conclusión general, a modo de reflexión sobre lo aprendido y el valor que deja el trabajo realizado.

En resumen, este capítulo quiere poner en contexto todo el esfuerzo previo: qué se ha logrado, qué falta por hacer y cómo este proyecto puede servir de base para seguir aprendiendo y mejorando en el campo de la detección y respuesta en *endpoints*.

6.1. Principales aportaciones

Antes de hablar de las posibles mejoras, merece la pena detenerse en lo que sí se ha conseguido. El objetivo inicial era construir un EDR básico para Windows que pudiera capturar telemetría de diferentes fuentes del sistema y centralizarla en un agente en espacio de usuario. En ese sentido, el proyecto ha cumplido con lo previsto: hoy se dispone de un prototipo capaz de observar lo que ocurre en el sistema, registrar esa información y almacenarla de forma organizada para su posterior análisis. Esto supone un paso importante, porque demuestra que es posible levantar una plataforma propia de detección desde cero, entendiendo cada engranaje y cada decisión técnica que hay detrás.

Entre las aportaciones más relevantes se pueden destacar las siguientes:

- El desarrollo de un **driver en modo kernel** capaz de capturar eventos clave como la creación de procesos, modificaciones en el registro o accesos a ficheros.
- La construcción de un **agente en espacio de usuario**, que recibe la telemetría del kernel y la organiza en una base de datos SQLite optimizada para inserciones concurrentes.

- El diseño de un **modelo de comunicación estable** entre kernel y userland, basado en memoria compartida con ring buffer e IOCTLs, que ha permitido transmitir los eventos sin afectar a la estabilidad del sistema durante las pruebas.
- La definición de un **esquema de datos modular**, con tablas diferenciadas para cada tipo de evento y un formato uniforme que simplifica tanto la ingesta como el análisis posterior.
- La realización de **pruebas en un entorno controlado**, que han confirmado que el prototipo es capaz de registrar y persistir correctamente la actividad generada por procesos instrumentados y muestras de prueba.
- La implementación de un **sistema de hooking mediante inyección de DLL**, que permite interceptar llamadas críticas a la API de Windows y registrar su uso, aportando un nivel de detalle adicional que no se consigue únicamente con callbacks en el kernel.

Cada parte del proyecto me enseñó algo diferente. Con el driver, entendí mejor el funcionamiento del kernel de Windows y la importancia de no comprometer la estabilidad del sistema. Desarrollando el agente en modo usuario, reforcé mis habilidades en Rust y comprobé su utilidad para escribir código seguro, especialmente al manejar datos sensibles. Aprendí la importancia de diseñar modelos de comunicación sencillos y fiables entre kernel y userland, así como a organizar telemetría útil para el análisis mediante una base de datos y un esquema modular. Las pruebas en entornos controlados verificaron el funcionamiento del sistema y revelaron limitaciones no tan evidentes teóricamente. Finalmente, el sistema de hooking con DLL demostró el valor de interceptar funciones críticas y las dificultades de mantener la estabilidad de los procesos al modificar su ejecución.

En conjunto, estas experiencias me permitieron ver de cerca lo que implica construir un EDR desde cero. Más allá del resultado técnico, lo valioso fue el proceso de aprendizaje: entender cómo encajan todas las piezas, descubrir sus limitaciones y comprobar que, paso a paso, es posible dar forma a un sistema real de detección en Windows.

6.2. Líneas de mejora y trabajo futuro

Aunque el prototipo cumple con los objetivos planteados, todavía queda mucho por hacer. Desde el principio asumí que este sería un proyecto de aprendizaje y no un producto listo para producción, así que es natural que existan muchos caminos abiertos para seguir mejorando y ampliando sus capacidades.

Algunas de las ideas más claras para el futuro son las siguientes:

- Ampliar las **fuentes de telemetría**, incorporando sensores adicionales como minifilters más completos o un mayor aprovechamiento de ETW para cubrir eventos que ahora mismo no están recogidos o redundar los que ya existen.
- Reforzar la **resistencia frente a evasión**, explorando técnicas anti-unhooking y validaciones de integridad que permitan detectar cuando un atacante intenta manipular el sistema.
- Mejorar la **capa de detección**, pasando de reglas simples a heurísticas más elaboradas o incluso explorando el uso de machine learning en fases posteriores del proyecto.
- Extender las **capacidades de respuesta**, más allá de la mera recolección de telemetría, permitiendo acciones como bloquear conexiones, aislar procesos o aplicar contramedidas básicas de forma automática.

- Experimentar con **despliegues multi-endpoint**, integrando el agente con un servidor central para correlacionar información y gestionar varios sistemas al mismo tiempo.
- Optimizar el **rendimiento del prototipo**, midiendo el impacto de los sensores en entornos más exigentes y afinando tanto la comunicación kernel-user como la persistencia de datos.

Todas estas líneas de mejora me muestran que el proyecto no termina aquí. Más bien, lo que he construido es una base sobre la que se pueden probar nuevas ideas y experimentar con distintas técnicas de detección y respuesta. Cada paso que se dé a partir de este prototipo puede acercar el sistema a algo más robusto y completo.

6.3. Conclusión final

Al terminar este proyecto siento que he alcanzado el objetivo principal: construir un prototipo de EDR que recoge telemetría en Windows y comprobar de primera mano cómo se integran sus piezas básicas. No es un sistema completo ni mucho menos, pero me ha permitido aprender en profundidad cómo funciona el kernel, cómo comunicarlo con userland y cómo organizar la información de manera que tenga sentido para la detección.

Más allá del código, lo que me llevo es la experiencia de haber trabajado en un entorno realista y haber superado los retos que implica moverse en capas tan sensibles del sistema operativo. Este trabajo me sirvió como punto de partida para entender mejor los EDR y me deja con la motivación de seguir explorando y mejorando en este campo.

Con todo ello, este capítulo pone fin al proyecto, pero no al camino iniciado. Lo construido es solo una primera base que abre la puerta a seguir explorando, ampliando y perfeccionando el sistema en el futuro. Más que un punto final, estas conclusiones marcan el comienzo de nuevas oportunidades de aprendizaje y desarrollo en el campo de la detección y respuesta en endpoints.

Página intencionalmente dejada en blanco.

Bibliography

0xflux (2023a). *Flux Security Blog*. Blog técnico de ciberseguridad. URL: https://fluxsec.red/.

0xflux (2023b). *Sanctum: Experimental EDR framework*. Repositorio en GitHub. URL: https://github.com/0xflux/sanctum.

Google (2025). *Protocol Buffers Documentation*. Documentación oficial. URL: https://protobuf.dev/.

Hand, Matt (sep. de 2023). *Evading EDR: The Definitive Guide to Defeating Endpoint Detection Systems*. San Francisco: No Starch Press, pág. 312. ISBN: 9781718503342.

Microsoft (2025). *WinDbg Preview Documentation*. Documentación oficial. URL: https://learn.microsoft.com/windows-hardware/drivers/debugger/.

Online, OSR (2020). *Getting Started Writing Windows Drivers*. Artículo introductorio. URL: https://www.osr.com/getting-started-writing-windows-drivers.

The Rust Project Developers (2025). *Rust Programming Language Documentation*. Documentación oficial. URL: https://doc.rust-lang.org/.

Yosifovich, Pavel (feb. de 2023). *Windows Kernel Programming, 2nd Edition*. Amazon Digital Services LLC - KDP, pág. 626.

Página intencionalmente dejada en blanco.

Apéndices

Página intencionalmente dejada en blanco.

A

Archivos adicionales

A.1. Fichero de configuración

Listado 29: *Fichero de configuración*

```
[database]
path               = "telemetry.db"
purge_on_restart   = true
synchronous        = "NORMAL"
journal_size_limit = 20000000
checkpoint_seconds = 30                # WAL commit time trigger
ttl_seconds        = 3600              # DB event delete time trigger
batch_size         = 1000              # In-memory buffer size before commit to WAL

[scanner]
max_file_size_mb = 50
extensions = ["exe", "dll", "sys", "ocx"]
rules = "C:\\Users\\MALDEV01\\RustroverProjects\\gladix-refactor\\user-agent\\src\\scanner\\yara-
↪  rules-core.yar"

[scanner.risk_groups.high]
scheduled_interval = 62
directories = ["C:\\Users\\MALDEV01\\Documents\\tools"]

[scanner.risk_groups.medium]
scheduled_interval = 100
directories = ["C:\\Users\\MALDEV01\\Documents\\GitHub"]

[scanner.risk_groups.low]
directories = ["C:\\Users\\MALDEV01\\Documents"]

[scanner.risk_groups.special]
directories = ["C:\\Users\\MALDEV01\\Documents"]

```

A.2. Código del proceso de pruebas

Listado 30: *Código de* `test_process.exe`

```rust
fn pause(step: &str) {
    print!("Press ENTER to {}... ", step);
    let _ = io::stdout().flush();
    let mut buf = String::new();
    let _ = io::stdin().read_line(&mut buf);
}

fn main() -> windows::core::Result<()> {
    // Reserve & commit 4 KB RW
    pause("call VirtualAlloc");
    let mem = unsafe {
        VirtualAlloc(
            None,
            4096,
            MEM_COMMIT | MEM_RESERVE,
            PAGE_READWRITE,
        )
    };
    println!("VirtualAlloc returned: {:?}", mem);

    pause("call VirtualProtect");
    ...

    pause("open or create test.txt");
    let name = to_pcwstr("test.txt");
    let file: HANDLE = unsafe {
        CreateFileW(
            PCWSTR(name.as_ptr()),
            FILE_GENERIC_WRITE.0,
            FILE_SHARE_READ,
            None,
            OPEN_ALWAYS,
            FILE_ATTRIBUTE_NORMAL,
            Option::from(HANDLE(null_mut())),
        )?
    };
    println!("CreateFileW returned handle: {:?}", file);

    pause("seek to end of file");
    ...

    pause("call WriteFile");
    let data = b"Hook test via WriteFile\n";
    let mut bytes_written: u32 = 0;
    unsafe {
        WriteFile(
            file,
            Some(data.as_ref()),
            Some(&mut bytes_written as *mut u32),
            None,
        )?
    };

    println!("\n Done - your detours should have intercepted these calls.");
}
```

Página intencionalmente dejada en blanco.