Titulo

Nombre

30 de noviembre de 2025

Índice general

Capítulo 1

Introducción

1.1. Alcance

El objetivo de este trabajo es desarrollar un escenario didáctico que permita a los analistas de ciberseguridad mejorar sus capacidades de detección e investigación mediante la comprensión del proceso completo de creación de un ciberataque. El resultado final del proyecto será un conjunto de artefactos forenses de los sistemas infectados, que servirá como base para ejercicios de análisis forense (blue team).

Para generar dicho recurso, el núcleo del trabajo se centrará en la concepción, diseño y desarrollo de un ciberataque realista, su ejecución controlada en un sistema víctima y la documentación íntegra del proceso ofensivo llevado a cabo. De este modo, quien reciba los artefactos forenses podrá analizar la infección desde el punto de vista del analista forense, pero también interpretar las acciones del atacante, entendiendo mejor el cómo y el porqué de cada artefacto encontrado.

En definitiva, este trabajo busca aproximar al analista a la mentalidad del atacante, con el fin de reforzar sus habilidades de detección, interpretación y respuesta ante amenazas reales.

1.2. Justificación

Este trabajo se sitúa dentro del ámbito de la *simulación de adversarios* (*Adversary Simulation*), una metodología que permite evaluar de forma controlada tanto la eficacia de los sistemas de defensa como las capacidades del personal ante incidentes de seguridad.

Aunque esta técnica puede aplicarse para probar sistemas, en este caso concreto el enfoque principal es la formación del personal. Se utiliza un es-

cenario de ataque realista para que los analistas practiquen la detección, el análisis y la interpretación de un incidente completo, mejorando sus habilidades y comprensión del comportamiento de un atacante.

1.3. Enunciado

Este recurso está diseñado para servir como material práctico de entrenamiento para analistas de seguridad. A continuación se presenta una breve descripción de la infraestructura de la organización víctima, junto con un enlace al conjunto completo de artefactos recopilados durante el incidente.

El objetivo del analista es reconstruir la historia completa del ataque a partir de dichos artefactos. Para facilitar este proceso, el trabajo se organiza en dos capítulos principales: en el capítulo *El Incidente* 3 se detalla de forma minuciosa cada acción realizada por el atacante, sus objetivos, las técnicas MITRE ATT&CK asociadas y los artefactos donde pueden encontrarse evidencias; por otra parte, el capítulo *Red Team Notes* 4 ofrece una visión desde la perspectiva del atacante, explicando los procedimientos empleados para ejecutar el ataque.

Infraestructura de la víctima

Como base del entorno corporativo se emplea **GOAD-Light**, una versión reducida del laboratorio GOAD orientada a entornos de Active Directory vulnerables. Este entorno proporciona un dominio Windows preconfigurado, servicios esenciales y múltiples debilidades que permiten simular técnicas reales de compromiso y movimiento lateral.

Las características completas del entorno GOAD-Light están documentadas por sus autores [1].

Dentro del dominio se integra una **workstation Windows 11**, actualizada y con todas las protecciones nativas habilitadas. Esta máquina actúa como punto de entrada inicial del ataque y como elemento de interacción principal del usuario víctima.

Para maximizar la visibilidad durante el análisis forense y aumentar el volumen de artefactos disponibles, se incorporan dos componentes adicionales:

- **Suricata**: desplegada para la monitorización y captura del tráfico de red, permitiendo correlacionar actividad sospechosa, conexiones salientes y patrones asociados al *command and control*.

- **Sysmon**: configurado en la workstation para generar telemetría enriquecida de eventos de Windows (creación de procesos, carga de DLLs, conexiones de red, modificaciones en el registro, entre otros), facilitando la reconstrucción temporal y contextual del ataque.

1.3.1. Recopilación de evidencias

En el siguiente enlace se encuentran todas las evidencias recopiladas del incidente:

Carpeta de evidencias del incidente

Cada carpeta corresponde a un equipo de la organización y lleva su nombre. Dentro de cada una se incluyen todos los artefactos adquiridos.

Los artefactos consisten en un volcado de la memoria RAM del equipo y otros elementos relevantes obtenidos durante la adquisición.

La recopilación se ha realizado con la herramienta **KAPE**, utilizando archivos `.tkape`. Estos archivos definen qué artefactos se recolectan, desde qué rutas y con qué reglas. Una explicación detallada sobre qué es un Target de KAPE puede consultarse en [2].

En este caso, se han utilizado los siguientes Targets:

- `!SANS_Triage` [3]
- `EventLogs` [4]

1.3.2. Cómo utilizar este recurso

Este recurso está pensado para servir como guía de apoyo durante el análisis del incidente y como material de referencia técnica para comprender en detalle cómo se desarrolló el ataque.

El contenido se organiza en dos bloques complementarios:

- **Capítulo 2: Marco Teórico.** Vocabulario técnico e instituciones sobre las que se apoya el contenido del trabajo.
- **Capítulo 3: Reconstrucción del ciberataque.** Presenta una visión estructurada del ataque, mapeando cada acción del adversario con las técnicas MITRE ATT&CK aplicadas y con los artefactos del sistema víctima donde podría encontrarse evidencia relevante. Este bloque está orientado a facilitar el trabajo del analista DFIR durante el proceso de correlación y reconstrucción de los hechos.

- **Capítulo 4: Red Team Notes.** Describe el ataque desde el punto de vista del adversario, explicando el razonamiento, las decisiones técnicas y los procedimientos utilizados en cada fase. Ofrece una visión interna del flujo del ataque y del funcionamiento de cada componente ofensivo.

El proyecto está diseñado para que el analista pueda avanzar de forma autónoma utilizando únicamente las evidencias del caso. Si en algún momento necesita orientación sobre dónde buscar rastros de una técnica, puede consultar el Capítulo 3. Si, en cambio, desea entender en profundidad por qué el atacante actuó de una determinada manera o cómo funciona una técnica concreta, puede acudir al Capítulo 4, donde se detalla su implementación desde la perspectiva del red team.

Si deseas analizar el caso sin spoilers, no continúes leyendo.

Capítulo 2

Marco Teórico

2.1. MITRE ATT&CK

MITRE ATT&CK es el marco de referencia fundamental para describir y clasificar el comportamiento de actores maliciosos en ejercicios de *Adversary Emulation*. Su estructura proporciona un vocabulario común basado en tres niveles de abstracción:

- **Tácticas (el "para qué")**: Representan los objetivos tácticos del adversario en cada fase del ataque. Describen la meta que se pretende alcanzar, como obtener acceso inicial, elevar privilegios o exfiltrar información.
- **Técnicas (el "qué")**: Definen el método concreto que utiliza el adversario para cumplir una táctica. Representan la acción observable destinada a lograr el objetivo, como el uso de *phishing*, ejecución de comandos o inyección de procesos.
- **Procedimientos (el "cómo")**: Son la implementación específica de una técnica. Detallan los pasos exactos, herramientas, parámetros y artefactos utilizados por un atacante. Un procedimiento debe ser lo suficientemente preciso como para que el *Blue Team* pueda generar reglas de detección y el *Red Team* pueda reproducirlo fielmente.

Este modelo estructurado permite mapear comportamientos reales, estandarizar evaluaciones y alinear las actividades defensivas y ofensivas dentro de un lenguaje común.

2.2. Cyber Kill Chain (CKC)

La *Cyber Kill Chain*, desarrollada por Lockheed Martin, describe las fases operativas de un ciberataque con el objetivo de estructurar y contextualizar las acciones del adversario. No se trata de un marco rígido, sino de un modelo conceptual que divide el ataque en etapas bien definidas para facilitar su análisis, detección e interrupción. El modelo identifica las tareas que un adversario debe completar para alcanzar su objetivo final, permitiendo anticipar comportamientos y reforzar las defensas en cada fase.

Los siete pasos de la *Cyber Kill Chain* mejoran la visibilidad del analista sobre el ataque y enriquecen su comprensión de las tácticas, técnicas y procedimientos (TTPs) empleados por el adversario.

1. **Reconocimiento**: Investigación del objetivo y recolección de información.
2. **Armamento**: Preparación o creación del payload y capacidades ofensivas asociadas.
3. **Entrega**: Transmisión del payload mediante el vector seleccionado.
4. **Explotación**: Activación del payload aprovechando una vulnerabilidad o interacción del usuario.
5. **Instalación**: Establecimiento del implante y preparación del entorno persistente.
6. **Mando y Control (C2)**: Creación del canal de comunicación entre el host comprometido y el adversario.
7. **Acciones sobre Objetivos**: Ejecución de los objetivos finales, como movimiento lateral, exfiltración o impacto.

2.3. Threat Intelligence

La Inteligencia de Amenazas es esencial para la simulación de adversarios (*Adversary Simulation*), ya que transforma datos sobre actores maliciosos, campañas y vulnerabilidades en información procesable. Se nutre de análisis técnicos, alertas de organismos de ciberseguridad, investigaciones de la industria y colecciones especializadas de TTPs mantenidas por la comunidad, como la *Adversary Emulation Library* [5], desarrollada por el *MITRE Center*

for Threat-Informed Defense. Este recurso proporciona perfiles de adversarios basados en comportamientos observados en grupos APT, aportando una base verificable para la creación de escenarios.

Esta información, estructurada mediante MITRE ATT&CK y contextualizada a través de la *Cyber Kill Chain*, permite diseñar simulaciones realistas, evaluar defensas, identificar brechas y entrenar al personal frente a amenazas alineadas con el panorama actual.

Capítulo 3

El Incidente

3.1. Descripción del Atacante

En este trabajo no se utiliza un adversario existente ni se replica el comportamiento de un APT conocido. En su lugar, se crea un atacante propio con el fin de construir un escenario didáctico totalmente adaptado a los objetivos del ejercicio. Esto permite ajustar, durante la elaboración del ataque, sus tácticas, técnicas y decisiones operativas para que encajen exactamente con las necesidades formativas del proyecto.

Aunque el adversario no está definido por completo al inicio, el ataque presentado sí constituye un escenario cerrado: la intrusión ya ha ocurrido y todas las evidencias están disponibles para su análisis.

3.2. Descripción del Ataque (Resumen Ejecutivo)

El escenario planteado reproduce un ataque estructurado en dos fases, diseñado para simular el comportamiento de un adversario realista. En la fase inicial, el atacante obtiene acceso al equipo víctima mediante un dispositivo físico que le permite ejecutar código de forma inmediata. Con este punto de apoyo establece una conexión remota hacia su infraestructura externa, desde la cual realiza un reconocimiento básico del sistema y prepara la siguiente fase.

En la segunda fase, el adversario despliega su componente principal, diseñado para operar de forma discreta y mantenerse en el sistema sin levantar alertas. Este componente establece persistencia, amplía la capacidad de control del atacante y se integra en procesos legítimos del sistema para dificultar

su detección. A partir de este momento, el adversario puede llevar a cabo acciones más avanzadas, como analizar el entorno, escalar privilegios, desplazarse por la red o extraer información.

3.3. Mapeo del Ataque a la Cyber Kill Chain y MITRE ATT&CK

En esta sección se presenta una referencia técnica que describe el ataque desde el punto de vista del adversario. Se detalla qué acciones llevó a cabo el atacante, cuáles eran sus objetivos en cada fase y qué técnicas MITRE ATT&CK aplicó. Cada acción se mapea con los artefactos del sistema víctima donde podría encontrarse información relevante sobre dicha actividad, proporcionando una visión clara y útil para el análisis.

La *Cyber Kill Chain* se utiliza como marco para organizar la exposición. El orden elegido es cronológico inverso —del estado final del ataque hacia sus fases iniciales—, ya que este enfoque facilita que el analista comprenda la secuencia real de los hechos: en un escenario DFIR, la investigación siempre comienza desde el impacto final y progresa hacia atrás.

El objetivo de esta sección es ofrecer una guía técnica que relacione de forma directa:

- Las acciones ofensivas del atacante,
- Las técnicas MITRE ATT&CK asociadas,
- Los artefactos del sistema donde podría hallarse evidencia de cada acción.

No se trata de una investigación completa, sino de una referencia estructurada que ayuda a entender cómo se manifiestan las técnicas del atacante en el sistema comprometido.

3.3.1. Acciones sobre el objetivo

Ransomware En este punto, el atacante ya ha permanecido un periodo considerable dentro de la organización, llevando a cabo diversas actividades de post-explotación. En la fase final de su operación ejecuta el cifrado de información crítica en el sistema comprometido.

Para una descripción detallada del procedimiento llevado a cabo por el adversario, consúltese el Capítulo 4, específicamente la Sección 4.8, donde se analiza el proceso completo con mayor profundidad.

Táctica: Impact (TA0040).

Técnicas: Data Encrypted for Impact (T1486).

Procedimiento: Se desplegó un *.NET assembly* dentro del proceso donde residía el implante. Este componente cifró el fichero `documento_importante.txt` utilizando criptografía simétrica. La clave simétrica fue posteriormente cifrada con una clave pública de un esquema asimétrico controlado por el atacante. Finalmente, se dejó una nota de rescate con un correo electrónico de contacto.

Artefactos relevantes

- ...

3.3.2. Mando y Control (C2)

Sliver C2 En este punto, el atacante ya ha superado las fases de explotación e instalación y dispone de un mecanismo persistente que le permite interactuar de forma continua con el sistema víctima mediante un implante de *Sliver*.

Para una descripción detallada del proceso mediante el cual el implante de Sliver se carga en memoria en cada ejecución, evitando la detección por parte del EDR, consúltese el Capítulo 4, Sección 4.4.3.

Asimismo, para obtener información sobre la infraestructura de servidores utilizada por el atacante, véase la Sección 4.7.

Táctica: Defense Evasion (TA0005).

Técnica: Reflective Code Loading (T1620).

Procedimiento: La DLL maliciosa utiliza carga reflexiva para ejecutarse dentro del proceso comprometido. El módulo `calibre-launcher.dll` establece una conexión con el servidor C2, descarga el implante de Sliver y lo carga en memoria mediante un mecanismo de reflexión, sin utilizar las APIs estándar de carga de librerías ni generar artefactos adicionales en disco.

Artefactos relevantes

- ...

Táctica: Command and Control (TA0011).

Técnicas:
- Application Layer Protocol: Web Protocols (T1071.001)
- Encrypted Channel (T1573)

Procedimiento: El implante residente en memoria mantiene un canal de comunicación persistente con la infraestructura de mando y control mediante un mecanismo de *beaconing*. Para reducir la probabilidad de detección, el tráfico se encapsula dentro de peticiones HTTPS que imitan patrones legítimos de navegación web. Además, todas las comunicaciones viajan cifradas, lo que impide la inspección de su contenido y permite al atacante intercambiar órdenes y resultados sin generar artefactos visibles o anomalías en el tráfico.

Artefactos relevantes

- ...

3.3.3. Instalación

Stage 2 En esta fase, el atacante ya ha superado la etapa de explotación inicial y ha logrado ejecutar código en el sistema víctima, obteniendo una *reverse shell*. A partir de este acceso, procede a preparar el *stage 2* del ataque y a desplegar su *payload* final, cuyo objetivo es establecer un mecanismo de acceso persistente al sistema comprometido.

Para lograrlo, el adversario emplea una combinación de técnicas basadas en *DLL Hijacking*, *Side Loading* y *DLL Proxying*. Un análisis detallado de este procedimiento puede consultarse en el Capítulo 4, Sección 4.4.

Táctica: Defense Evasion (TA0005).

Técnica: Modify Registry or File Permissions: File and Directory Permissions Modification (T1222).

Procedimiento: El atacante copia el ejecutable legítimo de Calibre desde su ubicación original a un directorio donde dispone de permisos de escritura, concretamente `%LOCALAPPDATA%\Microsoft\WindowsApps`. Esta acción le permite controlar el entorno de carga de librerías del

binario y establecer las condiciones necesarias para llevar a cabo el secuestro de DLL.

Artefactos relevantes

- ...

Táctica: Execution (TA0002).

Técnica: Ingress Tool Transfer (T1105).

Procedimiento: El atacante descarga en el directorio del binario legítimo una DLL maliciosa denominada `calibre-launcher.dll`, diseñada específicamente para reemplazar a la librería legítima. Esta DLL ya contiene la carga maliciosa integrada, lista para ejecutar el implante en cuanto la aplicación intente cargarla de forma normal.

Artefactos relevantes

- ...

Táctica: Defense Evasion / Privilege Escalation (TA0005 / TA0004).

Técnica:
- Hijack Execution Flow: DLL Search Order Hijacking (T1574.001).
- Masquerading: Match Legitimate Name or Location (T1036.005).

Procedimiento: El atacante renombra la DLL legítima a `calibre-launcher-old.dll` para conservar su funcionalidad original. A continuación, coloca en el directorio una DLL maliciosa con el nombre legítimo esperado por la aplicación (`calibre-launcher.dll`). Debido al orden de búsqueda de DLLs en Windows, la aplicación carga primero la DLL maliciosa, que actúa como un proxy reenviando las llamadas a la DLL legítima renombrada mientras ejecuta su código malicioso.

Artefactos relevantes

- ...

Persistencia El atacante establece un mecanismo de persistencia mediante la creación de una tarea programada, garantizando así que su *payload* se ejecute automáticamente tras cada reinicio del sistema y mantenga el acceso al entorno comprometido.

Para una descripción detallada del procedimiento, consúltese el Capítulo 4, Sección 4.6.

Táctica: Persistence (TA0003).

Técnica: Scheduled Task/Job (T1053).

Procedimiento: El atacante crea una tarea programada denominada `calibre-update`, configurada para ejecutarse en cada reinicio del sistema. Esta tarea invoca `calibre-debug.exe`, un ejecutable legítimo presente en la instalación original y que puede ejecutarse sin interfaz gráfica. Al lanzarse automáticamente, el binario vuelve a cargar la DLL maliciosa, permitiendo que el implante de Sliver recupere la persistencia tras cada reinicio.

Artefactos relevantes

- ...

3.3.4. Explotación

Stage 1 En este punto, el atacante obtiene la primera ejecución de código en el sistema víctima y despliega el *payload* inicial que le permite establecer una reverseshell con el equipo comprometido.

Para una descripción detallada del desarrollo y funcionamiento de este *payload*, véase el Capítulo 4, Sección 4.2.

Táctica: Execution (TA0002) / Command and Control (TA0011).

Técnica: Command and Scripting Interpreter: PowerShell (T1059.001).
Ingress Tool Transfer (T1105).
User Execution: Malicious File (T1204.002).
Remote Services / Reverse Shell (T1021).

Procedimiento: El atacante ejecuta un script de PowerShell que establece conexión con la infraestructura C2 y descarga un archivo temporal denominado `2fd19d11-d4b7-46d1-94db-7edd16980d15.tmp` en el directorio temporal del usuario (`$env:TEMP`). A continuación, renombra el archivo con extensión `.exe` y lo ejecuta de forma oculta mediante `-WindowStyle Hidden`. El binario lanzado establece una reverse shell hacia el servidor controlado por el atacante, brindándole acceso remoto al sistema comprometido.

Artefactos relevantes

- ...

3.3.5. Entrega

BadUSB En este punto, el atacante interactúa por primera vez con la organización víctima, marcando el inicio del incidente de seguridad. El objetivo de esta acción es introducir el primer *payload* dentro del entorno de la víctima mediante un dispositivo *BadUSB*.

Para un análisis detallado de las decisiones tomadas durante esta fase, véase el Capítulo 4, Sección 4.1.

Táctica: Initial Access (TA0001) / Execution (TA0002).

Técnica: Replication Through Removable Media (T1091)

Input Injection (T1674)

Procedimiento: El atacante conecta un dispositivo *BadUSB* que emula un teclado (HID). Nada más inicializarse, el dispositivo inyecta pulsaciones de teclado que abren el cuadro de ejecución mediante `Win + R` y lanzan una consola de PowerShell. Desde ella, se ejecutan los comandos necesarios para descargar y ejecutar el script malicioso sin interacción del usuario.

Artefactos relevantes.

- ...

3.3.6. Armamento

En esta fase, el atacante prepara los *payloads*, herramientas y la infraestructura que utilizará durante el ataque. Al igual que ocurre con la fase de reconocimiento, esta etapa se sitúa más cerca del análisis preventivo que del análisis forense de un incidente.

Las acciones llevadas a cabo por el atacante en esta fase se describen en el capítulo *Red Team Notes* (ver Sección 4), donde se detallan, desde la perspectiva ofensiva, el desarrollo de los *payloads*, las decisiones técnicas tomadas y la infraestructura empleada para ejecutar el ataque.

Desde el punto de vista del analista, esta fase puede apoyarse en servicios externos que realizan escaneos continuos de Internet en busca de infraestructuras maliciosas, como servidores de *Cobalt Strike* u otros *C2* con configuraciones mínimas o pobre *fingerprinting*. Estos servicios permiten a la organización detectar tempranamente posibles activos maliciosos relacionados con campañas activas y, en caso necesario, aplicar medidas preventivas como el bloqueo automático de conexiones hacia o desde dichas infraestructuras.

3.3.7. Reconocimiento

En esta fase, el atacante intenta recopilar información relevante sobre la organización, sus sistemas y posibles vectores de entrada. Desde el punto de vista del analista, estas actividades suelen considerarse parte de un análisis preventivo más que de un análisis de incidente propiamente dicho.

Dentro de esta categoría se incluyen tareas como el monitoreo de actividades de reconocimiento dirigidas a los endpoints expuestos de la organización, la revisión de bases de datos comercializadas por ciberdelincuentes que contengan información relacionada con la empresa y, en general, la supervisión de cualquier actividad externa que pueda indicar la recopilación de información previa a un ataque.

Capítulo 4

Red Team Notes

En esta sección se presentan todos los detalles relacionados con el desarrollo técnico del ataque, incluyendo la justificación de las decisiones adoptadas y las referencias a los repositorios necesarios para comprender y reproducir las técnicas empleadas.

4.1. Acceso Inicial: Justificación del Uso de BadUSB

El vector de *Initial Access* seleccionado para este proyecto se basa en el uso de un dispositivo *BadUSB*. Aunque no se trata de un método habitual en operaciones reales —debido a la necesidad de acceso físico, la existencia de controles como el bloqueo de puertos USB o la restricción de combinaciones como `Win + R`— su utilización resulta especialmente adecuada en un contexto didáctico.

La elección de BadUSB responde a su capacidad para ofrecer un escenario de ataque claro, reproducible y fácilmente comprensible. Permite demostrar de forma directa cómo una simple emulación de teclado puede derivar en la ejecución de código arbitrario en un sistema protegido, mostrando así el impacto que puede tener un fallo en los controles físicos o en las políticas de endurecimiento del puesto de trabajo.

Desde una perspectiva de ciberseguridad, este vector no difiere conceptualmente de otros métodos de acceso inicial ampliamente empleados por actores maliciosos. Tanto la explotación de vulnerabilidades (que suele culminar en la ejecución de un *payload* en memoria) como las campañas de *phishing* que despliegan instaladores maliciosos (`.msi`, `.exe`) o capturan credenciales válidas, comparten la misma finalidad: obtener la primera ejecución de código en el sistema de la víctima.

Para este ejercicio se empleó un payload BadUSB de desarrollo propio, adaptado al escenario del laboratorio [6].

4.1.1. Bypass de AMSI

Cuando el acceso inicial se realiza mediante un dispositivo BadUSB, la ejecución del código suele producirse a través de PowerShell. Esto implica que el primer mecanismo defensivo que debe evadirse habitualmente es AMSI (Antimalware Scan Interface), ya que intercepta y analiza el contenido antes de que PowerShell lo interprete.

Uno de los bypasses más utilizados es la técnica conocida como *AMSI Bypass – Memory Patching*. Durante la ejecución de código, PowerShell —o cualquier motor de scripting— invoca la función `AmsiScanBuffer()` presente en `amsi.dll`. Mediante la modificación en tiempo de ejecución de los bytes asociados al valor de retorno de esta función, es posible forzar que devuelva siempre un resultado "limpio" (por ejemplo, `0x80070057`, correspondiente a `E_INVALIDARG`). En muchas versiones de Windows, AMSI interpreta este estado como un fallo no malicioso y permite la ejecución del contenido sin aplicar medidas de bloqueo. *AMSI Bypass – Memory Patching: [7]*

En determinados escenarios, AMSI puede detectar intentos de manipulación directa de `AmsiScanBuffer()` y finalizar el proceso de manera inmediata, lo que limita la eficacia del método anterior. Para abordar esta situación se emplea una técnica alternativa ampliamente documentada.

El segundo método utilizado fue el *AMSI Bypass – Memory Patching via Reflection*. Durante la inicialización de PowerShell, la clase privada `System.Management.Automation.AmsiUtils` crea una variable estática denominada `amsiInitFailed`. Si este valor se establece en `true`, AMSI considera que su inicialización ha fallado y desactiva de forma global todos los análisis para el resto de la sesión, independientemente de que `amsi.dll` continúe cargada y operativa. Este enfoque resulta especialmente útil en entornos donde el parcheo directo del buffer es detectado y bloqueado. *AMSI Bypass - Memory Patching via Reflection: [8]*

4.2. Programar shellcodes

Un shellcode es una secuencia de instrucciones ensambladas en formato position-independent, capaz de ejecutarse en cualquier ubicación de memoria sin asumir un punto de entrada fijo. Esto permite que, independientemente de dónde sea cargado el bloque de código, su ejecución produzca un comportamiento controlado y determinista, característica esencial en escenarios de

explotación y ejecución arbitraria.

4.2.1. Position-Independent Code (PIC)

Una característica fundamental del *shellcode* es que debe estar implementado como *Position-Independent Code* (PIC). Esto implica que el código puede ejecutarse desde cualquier dirección de memoria sin asumir un *entry point* fijo. A diferencia de los programas compilados tradicionalmente —por ejemplo, un binario en C— que comienzan su ejecución en una dirección conocida y pueden referenciar offsets internos predecibles, un *shellcode* se ejecuta desde ubicaciones arbitrarias, desconocidas para sí mismo.

Para funcionar correctamente en este entorno dinámico, el *shellcode* debe ser capaz de determinar en tiempo de ejecución su posición en memoria. En arquitecturas como x86, una técnica clásica para lograrlo es el patrón `call-pop`, que permite recuperar el valor del contador de programa (`EIP`):

```
_start:
    call geteip

geteip:
    pop edx
    lea edx, [edx - 5]
```

La instrucción `call` empuja en la pila la dirección de retorno, que corresponde a la siguiente instrucción ejecutable. Al hacer un `pop` sobre ese valor, el *shellcode* recupera la dirección efectiva donde se encuentra en memoria. Restando el tamaño de la instrucción previa, se obtiene así el punto de inicio del propio bloque de código.

Esta técnica permite que el *shellcode* utilice rutas relativas para acceder a datos incrustados, cadenas, estructuras o código adicional, manteniendo su independencia respecto a la dirección donde haya sido inyectado.

4.2.2. Runtime Linking

Una vez resuelto el problema del *Position Independent Code* (PIC), el siguiente desafío fundamental en el desarrollo de shellcode es cómo invocar código de librerías dinámicas del sistema operativo. Cualquier operación relevante en Windows —como gestionar archivos, crear procesos o interactuar con memoria— requiere acceder a las *system calls* expuestas a través de módulos como `kernel32.dll`. Sin embargo, en un shellcode no disponemos del trabajo que normalmente realizan el *linker* en tiempo de compilación y

el *loader* en tiempo de carga, por lo que debemos reproducir este proceso manualmente durante la ejecución.

El objetivo principal consiste en localizar módulos cargados en el proceso y resolver las direcciones de sus funciones exportadas en tiempo de ejecución. Para ello recurrimos al *Process Environment Block* (PEB), cuya dirección es mantenida por la CPU en registros dedicados (`FS:[0x30]` en x86 y `GS:[0x60]` en x64). Desde el PEB es posible iterar sus estructuras internas, en particular la lista doblemente enlazada *InMemoryOrderModuleList*, que contiene información sobre todos los módulos cargados, incluyendo sus *base addresses*.

Una vez obtenida la base de un módulo concreto, el siguiente paso es analizar sus encabezados PE hasta localizar la *Export Table*. Esta tabla contiene los nombres de las funciones exportadas, sus *ordinals* y, lo más importante, los *Relative Virtual Addresses* (RVAs) que permiten calcular la dirección real de cada función mediante:

$$\text{func\_addr} = \text{module\_base} + \text{RVA}$$

Iterando esta tabla hasta encontrar la función deseada, el shellcode puede reconstruir la dirección exacta y, a partir de ese momento, invocar la API igual que lo haría un binario convencional.

Este mecanismo resuelve completamente la problemática del *runtime linking* dentro de un entorno sin información previa como es el shellcode. Para este proyecto se han utilizado implementaciones propias en ensamblador tanto para la obtención del *module base address* a través del PEB, como para la resolución dinámica de funciones exportadas. Dichas implementaciones pueden consultarse en los siguientes repositorios:

- Implementación de `GetModuleHandle`-like:
  - x64: [9]
  - x86: [10]
- Implementación de `GetProcAddress`-like:
  - x64: [11]
  - x86: [12]

Hashing de nombres de funciones

Como medida de evasión y anti-análisis, es conveniente evitar incluir cadenas de texto en claro dentro del shellcode. Si los nombres de las funciones

o módulos aparecen directamente en el binario, un analista puede identificarlos de manera inmediata aplicando herramientas básicas de extracción de cadenas. Para mitigar esto, una técnica habitual consiste en emplear algoritmos de *hashing* para representar los nombres de las funciones que queremos resolver en tiempo de ejecución.

En este proyecto se ha implementado un sistema de hashing sencillo basado en rotaciones y operaciones XOR. Con este mecanismo, el shellcode almacena únicamente los valores hash, lo que dificulta la identificación directa de las APIs utilizadas. Durante la ejecución, las funciones exportadas del módulo correspondiente se recorren una a una, calculando su hash y comparándolo con el valor objetivo.

Las implementaciones en ensamblador utilizadas son las siguientes:

- Hashing x64: [13]
- Hashing x86: [14]

4.2.3. ABI en x86 y x64

Finalmente, al trabajar con shellcode es imprescindible respetar las reglas de la *Application Binary Interface* (ABI), especialmente en entornos x64. En esta arquitectura, la pila debe mantenerse alineada a 16 bytes en el momento de realizar una llamada a una función (`RSP mod 16 = 0`). Además, la convención de llamadas de Windows x64 exige reservar un *shadow space* de 32 bytes antes de invocar cualquier función de una DLL del sistema; si estas condiciones no se cumplen, las llamadas pueden fallar de forma silenciosa o provocar un comportamiento indefinido.

También es necesario considerar la convención de llamadas utilizada para interactuar con las APIs del sistema. En x86, los argumentos se pasan por pila siguiendo un orden *right-to-left*. En x64, los primeros cuatro parámetros se pasan mediante registros (`RCX`, `RDX`, `R8` y `R9`), mientras que el resto se colocan en la pila respetando la alineación mencionada. Asimismo, la gestión de *stack frames*, la preservación de registros no volátiles y otras tareas que habitualmente realiza el compilador deben ser manejadas explícitamente dentro del shellcode.

Cumplir estrictamente estas normas resulta esencial para garantizar que las funciones del sistema operativo se ejecuten correctamente y que el shellcode mantenga un comportamiento estable y predecible.

4.2.4. Payloads finales

Una vez establecidos los tres pilares fundamentales para el desarrollo de shellcode —*Position Independent Code* (PIC), *runtime linking* y respeto de la ABI— es posible construir payloads completamente personalizados desde cero.

Como recomendación práctica, resulta útil desarrollar primero el programa equivalente en C y compilarlo sin optimizaciones. Esto permite tener una referencia clara de cómo debería verse el comportamiento final y observar directamente cómo el compilador gestiona aspectos como las llamadas a funciones, las variables locales o el uso del stack.

De forma adicional, en algunas circunstancias puede ser útil cargar un binario compilado en C en una herramienta de análisis estático como IDA, para comprobar el tamaño que el compilador reserva para ciertas variables o estructuras. Esto es especialmente práctico cuando ciertos valores no son triviales de deducir únicamente a partir de la documentación.

A continuación se muestran ejemplos de payloads desarrollados para este proyecto:

MessageBox payload Un payload sencillo cuyo objetivo es mostrar un *message box*. Se ha utilizado para validar las rutinas de inyección de código y comprobar la correcta resolución de funciones mediante *runtime linking*.

- x64: [15]
- x86: [16]

Reverse Shell payload Este es el **payload utilizado en la versión final del ataque** descrito en este trabajo. Se trata de la reverse shell empleada durante toda la ejecución práctica del ejercicio Red Team.

Por motivos didácticos, el shellcode de la reverse shell se ha incluido dentro de un fichero PE mínimo para que quede rastro en disco y su análisis resulte más sencillo posteriormente. Para generar este PE se ha utilizado la herramienta `sclauncher` [17], que crea únicamente las cabeceras básicas del ejecutable, define una sección `.text` y coloca en ella el bloque de shellcode. De este modo, el loader de Windows lo mapea correctamente y transfiere la ejecución a la shellcode sin necesidad de estructuras adicionales.

- **Reverse shell x86 (payload final utilizado en el ataque):** [18]

4.3. Inyección de código

La inyección de código consiste en conseguir que un proceso ejecute instrucciones distintas a las previstas originalmente. Existen múltiples técnicas para lograrlo, pero todas se basan en un principio común: modificar el flujo de ejecución o incorporar código arbitrario dentro del espacio de memoria de un proceso legítimo.

Los métodos clásicos incluyen la reserva de memoria en el proceso objetivo para copiar y ejecutar shellcode, así como la carga forzada de librerías que el binario no tenía previstas. Esta última categoría da lugar a un conjunto amplio de técnicas orientadas a manipular la forma en que Windows resuelve y carga DLLs.

Dentro de estas técnicas, una de las más relevantes y estudiadas es el DLL Hijacking.

4.3.1. DLL Hijacking

El *DLL Hijacking* consiste en forzar que el *loader* de Windows cargue una biblioteca dinámica controlada por el atacante, aprovechando el orden de búsqueda de DLLs o configuraciones del sistema.

Objetivo principal Lograr que un ejecutable legítimo cargue una DLL maliciosa con el mismo nombre que una DLL esperada, pero ubicada en una ruta que el atacante controla.

Orden de carga de DLLs en Windows

El *loader* resuelve las dependencias siguiendo el siguiente orden:

1. Directorio desde el cual se cargó la aplicación.
2. `C:\Windows\System32`
3. `C:\Windows\System`
4. `C:\Windows`
5. Directorio de trabajo actual (CWD).
6. Directorios definidos en la variable de entorno `PATH` del sistema.
7. Directorios definidos en la variable de entorno `PATH` del usuario.

KnownDLLs — Restricción crítica

Las DLL listadas en:

```
HKEY_LOCAL_MACHINE\SYSTEM\CurrentControlSet\Control\
Session Manager\KnownDLLs
```

solo pueden cargarse desde `System32`, lo que imposibilita su secuestro mediante *DLL Hijacking* tradicional.

4.4. Weaponización de DLL Hijacking

Para transformar este mecanismo en una técnica ofensiva se emplean tres enfoques complementarios: *Side Loading*, *DLL Proxying* y *Reflective Loading*.

En el ataque desarrollado en este trabajo se explota la aplicación legítima *Calibre* y, en concreto, su librería `calibre-launcher.dll`. La vulnerabilidad asociada a esta DLL está documentada en la plataforma *HijackLibs* [19], que mantiene un catálogo actualizado de DLLs susceptibles de ser secuestradas.

El objetivo es conseguir que un ejecutable legítimo, ya presente en el sistema, cargue una DLL bajo control del atacante. En este escenario, la DLL maliciosa actúa como stage 2 del ataque. Su cometido es establecer una conexión con el C2 y descargar la carga final: una DLL que contiene un implante de Sliver. Esta DLL descargada no se escribe en disco en ningún momento; la propia DLL maliciosa la carga directamente en memoria mediante reflective loading y transfiere la ejecución al implante. De este modo, un proceso legítimo termina ejecutando código malicioso sin dejar artefactos en el sistema de archivos.

De este modo, se consigue que un proceso legítimo —en este caso, *Calibre*— ejecute dentro de su propio espacio de direcciones un *implant* de *Sliver*, manteniendo la operación completamente *fileless*.

4.4.1. Side Loading

El primer objetivo es conseguir que un ejecutable legítimo cargue nuestra DLL maliciosa. La técnica de *Side Loading* se basa en colocar una DLL bajo control del atacante en el mismo directorio que el ejecutable objetivo, utilizando exactamente el mismo nombre que la DLL legítima que el programa espera cargar. Para ello, es necesario identificar previamente aplicaciones vulnerables a este tipo de *DLL Hijacking*.

Una limitación habitual es que los directorios donde residen los binarios legítimos no permiten la escritura por parte de usuarios sin privilegios elevados, lo que impide introducir la DLL maliciosa directamente. No obstante,

estos directorios sí permiten lectura, de modo que el atacante puede copiar el ejecutable legítimo a una ubicación donde sí disponga de permisos de escritura y, posteriormente, insertar allí la DLL manipulada.

Una vez trasladado el ejecutable a un directorio controlado, basta con colocar la DLL maliciosa con el mismo nombre que la DLL legítima requerida por el programa. De este modo, al ejecutarse, el binario cargará la versión maliciosa en lugar de la original.

Por ejemplo, es posible copiar los archivos de la aplicación a un directorio menos restrictivo mediante:

```
robocopy "C:\Program Files\Calibre2" ^
         "%LOCALAPPDATA%\Microsoft\WindowsApps" ^
         /E /COPY:DAT /R:3 /W:5
```

4.4.2. DLL Proxying

Con la técnica de *Side Loading* hemos logrado que un ejecutable legítimo cargue una DLL maliciosa. Sin embargo, esto introduce un nuevo problema: al sustituir la DLL original, el programa deja de disponer de las funciones que necesita para funcionar correctamente. Si la DLL maliciosa no reproduce esas exportaciones, la aplicación fallará o directamente no arrancará.

Para resolver este problema se utiliza *DLL Proxying*. Esta técnica consiste en compilar una DLL maliciosa que:

- Ejecuta las operaciones ofensivas necesarias (en nuestro caso, cargar en memoria el implante final).
- Mantiene toda la funcionalidad original de la DLL legítima, reenviando sus exportaciones reales.

Ficheros `.def`

Para que una DLL maliciosa actúe como sustituta de otra es necesario replicar exactamente sus exportaciones. Esto se consigue mediante un fichero `.def` (Module Definition File), que permite definir explícitamente qué funciones expone la DLL y, opcionalmente, redirigir cada una a otra DLL.

La estructura básica de un fichero `.def` es:

```
EXPORTS
FuncB = real-old.FuncB @2
FuncC = real-old.FuncC @3
```

Cada línea indica:

- el nombre que exporta la DLL maliciosa,
- la función real a la que se redirige,
- y el ordinal asociado.

El ordinal debe conservarse porque algunos programas importan funciones por número en lugar de por nombre.

Durante el proyecto se utilizó un script en Python para generar automáticamente estos ficheros `.def`. El script está disponible en [20]. Asimismo, puede encontrarse una demostración completa de *Side Loading + DLL Proxying* en [21].

En esa prueba de concepto se crea un ejecutable que importa tres funciones (`FuncA`, `FuncB`, `FuncC`) y una DLL legítima que las implementa. La DLL maliciosa se compila de forma que:

- exporta su propia versión de `FuncA` (donde podría incluirse cualquier lógica ofensiva),
- redirige `FuncB` y `FuncC` a la DLL real mediante el fichero `proxy.def`.

Una compilación típica sería:

```
x86_64-w64-mingw32-gcc -shared -o bin/malicious.dll
dll-malicious.c proxy.def -s
```

Para finalizar la demostración, basta con renombrar `malicious.dll` a `dll-real.dll` y colocarlo junto al ejecutable vulnerable: en ese momento, *Side Loading* y *DLL Proxying* quedan operativos.

El resultado es similar a un *man-in-the-middle* dentro del propio mecanismo de carga de DLLs: el programa cree estar usando la DLL legítima, pero en realidad está ejecutando código controlado por el atacante mientras conserva su funcionalidad original.

4.4.3. Reflective Loading

Una vez logramos que un proceso legítimo cargue una DLL maliciosa sin alterar su flujo normal, el siguiente paso es introducir la carga ofensiva real. En este escenario, el objetivo es cargar una DLL directamente en el espacio de direcciones del proceso, sin que exista físicamente en disco.

Windows impone una limitación importante: las DLL solo pueden cargarse mediante las APIs estándar (`LoadLibrary` y derivados), las cuales requieren que el archivo exista en el sistema de ficheros. La técnica de *Reflective Loading* evita esta restricción cargando la DLL íntegramente desde memoria. De este modo, es posible ejecutar código arbitrario sin generar artefactos en disco, reduciendo la exposición a firmas estáticas y detecciones basadas en *file scanning*.

Como referencia, se ha desarrollado una demostración en C que implementa la carga manual de una DLL sin utilizar las funciones estándar del sistema [22]. El código puede consultarse en el repositorio mencionado.

Proceso de carga reflectiva

Para implementar *Reflective Loading*, el programa debe llevar a cabo las siguientes etapas:

1. **Preparar la DLL en memoria.** El archivo PE de la DLL debe estar ya cargado en memoria, normalmente como un bloque continuo de bytes.

2. **Procesar las cabeceras.** Se interpretan las estructuras PE iniciales para obtener información crítica, incluyendo el tamaño total que ocupará la DLL en memoria (`IMAGE_OPTIONAL_HEADER64.SizeOfImage`).

3. **Reservar memoria para la DLL.** Se solicita al proceso un bloque de memoria con el tamaño indicado por `SizeOfImage`, donde se cargará la DLL reconstruida.

4. **Copiar las cabeceras.** Las cabeceras del PE se copian al nuevo bloque de memoria, ya que algunas estructuras deberán ser modificadas durante la carga.

5. **Mapear las secciones.** Utilizando `IMAGE_FIRST_SECTION`, se obtiene la dirección del primer `IMAGE_SECTION_HEADER`. Para cada sección:

   - Se leen los offsets y tamaños desde el PE en memoria.
   - Se copian los datos a la ubicación adecuada dentro del bloque reservado, respetando el RVA definido en las cabeceras.

6. **Resolver la tabla de importaciones.** La estructura `IMAGE_IMPORT_DESCRIPTOR` define los módulos y funciones requeridos por la DLL. Para cada módulo importado:

- Se comprueba si ya está cargado en el proceso; si no, se carga.
- Se obtiene su *base address*.
- Se recorren las tablas `PIMAGE_THUNK_DATA` para resolver cada función:
  - El nombre de la función se obtiene de la estructura original.
  - Se localiza su dirección en el módulo cargado.
  - Se escribe esa dirección en la IAT reconstruida.

Esto garantiza que, cuando la DLL cargada reflectivamente invoque funciones externas, las llamadas apunten a direcciones válidas.

7. **(Opcional) Aplicar relocaciones.** Si la DLL no se carga en su `ImageBase` preferido, deben procesarse las entradas de la tabla de relocaciones. En esta explicación se omite por simplicidad, aunque sería necesario en una implementación completa.

8. **Invocar `DllMain`.** Con la DLL completamente mapeada en memoria y las dependencias resueltas, solo queda ejecutar: `DllMain(baseAddress, DLL_PROCESS_ATTACH, NULL);`

En este punto, la DLL queda cargada y ejecutándose íntegramente desde memoria, sin intervención de las APIs estándar de Windows y sin necesidad de que exista como archivo en disco.

Esta sección no pretende ofrecer una explicación completamente sistemática de la técnica, sino proporcionar una visión operativa que permita comprender sus fundamentos. Para profundizar en el funcionamiento interno del proceso, se recomienda revisar en el anexo la sección dedicada a los ficheros PE, así como examinar detenidamente el código fuente del repositorio [22]. En dicha implementación se emplean las estructuras oficiales definidas en `winnt.h`, por lo que resulta especialmente útil comparar estas estructuras con los offsets y campos especificados en la documentación del formato PE. Para ello, puede consultarse la infografía técnica incluida en [23], lo que facilita relacionar las estructuras utilizadas en la implementación con sus equivalentes formales dentro del estándar PE definido por Microsoft.

4.5. Syscalls y evasión de EDR

En este punto del ataque, el implante ya está completamente desplegado: se ejecuta en el sistema comprometido, mantiene comunicación estable

con el C2 y está preparado para iniciar técnicas de post-explotación. Antes de avanzar, es imprescindible comprender los mecanismos modernos de detección utilizados por soluciones EDR, especialmente aquellos basados en *hooking* de APIs.

Los EDR actuales instrumentan funciones críticas de `kernel32.dll`, `kernelbase.dll` y `ntdll.dll` mediante **API hooking**. De este modo, cada vez que el implante invoca funciones como `CreateProcessW`, `VirtualAllocEx`, `NtWriteVirtualMemory` o similares, el EDR intercepta la llamada, analiza los parámetros y decide si permitir, alertar o bloquear la operación.

Para minimizar la superficie de detección, los operadores utilizan diferentes grados de interacción directa con el sistema operativo. Estos niveles pueden organizarse en cuatro categorías progresivas, desde las API de mayor visibilidad hasta técnicas de evasión más avanzadas.

4.5.1. Nivel 1 – Win32 API

- Conjunto de funciones de alto nivel diseñadas para desarrollo estándar.
- Ejemplos típicos: `CreateFileW`, `VirtualAllocEx`, `WriteProcessMemory`.
- Resueltas estática o dinámicamente desde `kernel32.dll` o `kernelbase.dll`.
- Constituyen el nivel más monitorizado: cualquier EDR comercial detecta su uso de forma prácticamente completa.

[24] – Demo de uso Win32 API.

4.5.2. Nivel 2 – Native API

- Funciones de bajo nivel exportadas por `ntdll.dll`.
- Siguen el prefijo `Nt` o `Zw`, como `NtCreateFile` o `NtAllocateVirtualMemory`.
- Representan los *wrappers* oficiales en espacio de usuario que preparan los registros y ejecutan la instrucción `syscall`.
- Aunque ofrecen mayor control, muchos EDR siguen aplicando *hooks* sobre estas funciones críticas.

[25] – Implementación de Native API para evitar hooks.

4.5.3. Nivel 3 – Direct Syscalls

- Eliminan por completo la dependencia de `ntdll.dll`.
- El operador carga manualmente el **System Service Number** (SSN) en `RAX` (x64) o `EAX` (x86) y ejecuta `syscall`.
- El SSN varía entre versiones y compilaciones de Windows, lo cual requiere enumeración dinámica o bases de datos específicas por build.
- Ejemplo habitual para `NtAllocateVirtualMemory`:

```
NtAllocateVirtualMemory PROC
mov r10, rcx
mov eax, 18h
syscall
ret
NtAllocateVirtualMemory ENDP
```

- Los EDR modernos pueden detectar esta técnica comparando el origen de la instrucción `syscall` mediante ETW/ETW-TI, ya que no proviene de `ntdll.dll`.

[26] – Ejemplo de Direct Syscalls en Windows.

4.5.4. Nivel 4 – Indirect Syscalls

- Variante avanzada de las direct syscalls que reintroduce un retorno controlado hacia `ntdll.dll`.
- El flujo salta a la instrucción original de `ntdll.dll` inmediatamente después de su propia `syscall`, preservando la legitimidad aparente del contexto.
- Ejemplo simplificado:

```
EXTERN sysAddrNtAllocateVirtualMemory:QWORD

NtAllocateVirtualMemory PROC
mov r10, rcx
mov eax, 18h
jmp QWORD PTR [sysAddrNtAllocateVirtualMemory]
NtAllocateVirtualMemory ENDP
```

- Desde la perspectiva del EDR (y de ETW), la instrucción `syscall` parece ejecutarse dentro de `ntdll.dll`, lo que la vuelve extremadamente difícil de detectar sin medidas basadas en telemetría avanzada o heurísticas de integridad de memoria.

[27] – Ejemplo de Indirect Syscalls en Windows.

Librerías como HellGate, HaloGate, Freshycalls y SysWhispers3 automatizan la ejecución de direct e indirect syscalls para evadir hooks de EDR en ntdll.dll. Resuelven los SSN y direcciones en tiempo de compilación o carga, generan stubs limpios o reutilizan gadgets legítimos terminados en syscall; ret dentro de secciones no hookeadas de ntdll.dll (técnicas Hell's Gate, Halo's Gate y Tartarus' Gate), permitiendo llamar syscalls sin tocar las funciones exportadas monitorizadas y sin escribir ensamblador manualmente.

Aunque en esta primera versión del ataque no se han empleado *syscalls* directas ni técnicas de *syscall bypass*, entender cómo funcionan los *hooks* en `ntdll.dll` y las formas de evitarlos es esencial. Estas capacidades son clave para reducir la visibilidad frente a EDR y serán necesarias en futuras versiones del ataque.

4.6. Persistencia

En un entorno Windows existen múltiples métodos para establecer persistencia tras comprometer un sistema. Entre las técnicas más comunes destacan:

- **Claves de registro** para ejecutar binarios maliciosos al inicio del sistema o sesión.
- **Servicios** configurados para iniciarse automáticamente.
- **Tareas programadas** creadas mediante `schtasks`.
- **Elementos del menú *Start-up*** o carpetas de inicio.
- **DLL Hijacking**, aprovechando ejecutables que cargan librerías sin rutas absolutas; si el binario vulnerable se ejecuta al iniciar el sistema.

Aunque el ecosistema de técnicas de persistencia es amplio, en este ataque se optó por un método sencillo, estable y difícil de detectar a simple vista: la creación de una tarea programada que simula una actualización legítima del software instalado en el equipo. Para ello, se configuró una tarea diaria

denominada `CalibreUpdater`, cuyo propósito aparente es ejecutar un componente de actualización del software Calibre, pero que en realidad invoca el *payload* malicioso.

La tarea fue creada mediante el siguiente comando:

```
schtasks /create ^
  /tn "CalibreUpdater" ^
  /tr "%LOCALAPPDATA%\Microsoft\WindowsApps\Calibre2\calibre-debug.exe \
       -c \"import time; time.sleep(86400)\"" ^
  /sc daily ^
  /st 09:00
```

Este enfoque permite que la persistencia pase desapercibida al camuflarse como una tarea rutinaria de mantenimiento del sistema, lo que reduce la probabilidad de detección durante una inspección superficial.

4.7. Infraestructura C2

La infraestructura del atacante se despliega en **AWS** para mantener aislamiento y control sobre todo el flujo operacional. Se divide en varios servicios independientes con el fin de separar funciones, reducir riesgos y evitar exponer directamente el servidor C2 principal. Sliver, además, incorpora técnicas de *anti-fingerprinting* que dificultan identificar el servidor como infraestructura maliciosa.

Exponer una *reverse shell* a Internet implica riesgos evidentes —interceptación, reutilización de la sesión o identificación del origen—, pero en este caso el sistema que la recibe es desechable, por lo que no se requiere una protección compleja. Aun así, existen alternativas comunes como túneles efímeros o proxys temporales.

La arquitectura se organiza en tres componentes diferenciados:

- **Servidor de entrega de payloads (CDN):** Aloja los binarios del *stage 2* y actúa como punto inicial de descarga. Su objetivo es evitar que el implante o los artefactos iniciales apunten directamente al C2 real.
- **Servidor para la reverse shell:** Máquina temporal usada exclusivamente para recibir la sesión interactiva del *stage 1*. Permite al atacante ejecutar comandos básicos y preparar la instalación del implante principal.

- **Servidor C2 del implante (Sliver):** Nodo dedicado a gestionar la comunicación segura del *stage 2*. Controla tareas, módulos, telemetría y persistencia del implante. Al mantenerse separado del flujo inicial, reduce significativamente la superficie de exposición.

4.8. Acciones finales

Una vez que el implante obtiene ejecución en el proceso comprometido, el atacante puede llevar a cabo distintas acciones finales para ejecutar código adicional en memoria. Estas técnicas permiten cargar nuevo payload sin necesidad de escribir artefactos en disco, manteniendo un perfil sigiloso. Entre las opciones más comunes se encuentran:

4.8.1. BOF y COFF

Los *Beacon Object Files* (BOF) y los *Common Object File Format* (COFF) son archivos `.obj` generados por el compilador de C pero no vinculados (*unlinked*). Estos objetos pueden ser cargados directamente en memoria por el implante, permitiendo ejecutar código nativo sin necesidad de crear un ejecutable completo. Su formato compacto y la ausencia de dependencias externas los hace ideales para operaciones sigilosas en memoria.

4.8.2. Shared Library Side Loading

El implante puede cargar bibliotecas dinámicas de forma reflexiva, sin utilizar las funciones estándar del sistema operativo. Este método, similar al empleado en el *stage 2* del ataque, permite mapear la DLL directamente en memoria, resolver sus símbolos manualmente y ejecutar su código sin registrarla en el cargador de Windows. El *side loading* facilita también la suplantación de DLLs legítimas para redirigir el flujo de ejecución hacia código malicioso.

4.8.3. Assemblies .NET

En el contexto de nuestro proyecto, los *assemblies* de `.NET` constituyen el mecanismo elegido para cargar y ejecutar código adicional dentro del propio implante. Windows inicializa una serie de procesos y servicios críticos para gestionar el sistema operativo, entre ellos el *Common Language Runtime (CLR)* en entornos donde se utilizan aplicaciones `.NET`. Aprovechar este com-

ponente nos permite inyectar y ejecutar código gestionado de forma sigilosa y flexible.

El implante está limitado al uso de `.NET Framework`, debido a las restricciones propias de Sliver. Es importante distinguirlo de `.NET Core / .NET 5+`, ya que sus características difieren notablemente:

- **.NET Framework (v4.x):** Orientado exclusivamente a Windows. No permite compilación cruzada, por lo que los assemblies deben generarse específicamente para esta plataforma. Aunque Windows incluye el CLR, en Sliver el runtime debe ser cargado manualmente por el implante mediante técnicas de *runtime hosting*, permitiendo ejecutar assemblies incluso en procesos que no habían inicializado previamente un entorno gestionado.
- **.NET Core / .NET 5+:** Multiplataforma y modular. A diferencia del Framework clásico, el payload puede **transportar su propio runtime**, permitiendo la ejecución sin depender de componentes instalados en el sistema objetivo. Está diseñado para entornos modernos y despliegues autocontenidos.

Un *assembly* `.NET` es esencialmente un contenedor que agrupa código compilado en **Intermediate Language (IL)**, un bytecode similar conceptualmente al de Java. Este IL no se ejecuta directamente, sino que es interpretado y gestionado por el *Common Language Runtime*. Esto implica que cuando compilamos un proyecto en C# no obtenemos código máquina, sino un lenguaje intermedio que será ejecutado por el CLR del `.NET Framework`.

En términos operativos, el implante es capaz de:

- Ejecutar un assembly **in-process**, cargando el CLR dentro del proceso actual e invocando directamente el código gestionado.
- Ejecutarlo en un **proceso sacrificado**, inicializando el runtime en un proceso externo para reducir la huella y aislar la ejecución.

Esto nos permite escribir código directamente en C# que será ejecutado en memoria, de forma muy similar al comportamiento de un shellcode, pero aprovechando la potencia y abstracción del ecosistema `.NET`. La técnica destaca por su flexibilidad, capacidad de integración con APIs de alto nivel y mínima fricción durante la carga del payload.

No obstante, esta aproximación no está exenta de riesgos de detección. Un sistema que monitorice la aparición del runtime de `.NET` en procesos donde normalmente no debería estar presente puede identificar esta anomalía, revelando la actividad del implante. Por ello, resulta relevante considerar medidas de evasión complementarias cuando se emplea este enfoque.

Capítulo 5

Anexo A: Windows Internals

En este anexo se ofrece una visión sintetizada de varios componentes fundamentales de Windows Internals que resultan esenciales para comprender el comportamiento del malware y de las técnicas de evasión modernas. Se introducen los conceptos de memoria virtual, el formato ejecutable PE y la arquitectura interna de los procesos en Windows, proporcionando el contexto técnico necesario para interpretar adecuadamente las operaciones realizadas por el adversario durante el ataque. Junto a cada explicación teórica, se incluyen pequeños fragmentos de código en C que muestran cómo navegar e interactuar con estas estructuras en tiempo de ejecución, facilitando una comprensión práctica de su funcionamiento real dentro del sistema operativo.

5.1. Memoria Virtual

La memoria virtual es uno de los pilares del diseño moderno de Windows y constituye un concepto clave para entender cómo se ejecutan y aíslan los procesos en el sistema operativo. Este mecanismo permite que cada proceso disponga de su propio espacio de direcciones lógico, independiente del resto, lo que facilita la protección, la gestión eficiente de recursos y la estabilidad del sistema.

Las infografías presentadas en esta sección se han extraído de [28].

5.1.1. Problemas Memoria Fisica

- Escasez de memoria
- Fragmentación
- Acceso a memoria (Seguridad)

5.1.2. Acceso indirecto a memoria

Cada proceso tiene su propio espacio de direcciones virtual, Para el proceso es como tener un espacio de momoria fisica contiguo reservado, Pero por detras el SO esta mapeando esas direcciones virtuales a distintas regiones de memoria fisica.

5.1.3. Paginación

Mapear cada byte de la memoria virtual directamente a un byte de la memoria física sería un proceso extremadamente ineficiente. Para optimizar este mecanismo, Windows —como la mayoría de sistemas operativos modernos— utiliza un esquema de paginación. En este modelo, la memoria virtual se divide en bloques llamados *pages*, mientras que la memoria física se organiza en bloques del mismo tamaño denominados *frames*. El tamaño habitual de una página es de 4 KB (4096 bytes), aunque pueden existir tamaños mayores en configuraciones específicas.

Cuando un proceso se ejecuta, sus páginas virtuales se asocian a *frames* disponibles en la memoria física. Este sistema permite una gestión más flexible, eficiente y segura de la memoria, facilitando además mecanismos como la protección de páginas, la paginación a disco y el aislamiento entre procesos.

5.1.4. Tabla de páginas

Cada proceso mantiene su propia tabla de páginas, una estructura residente en memoria física dentro del espacio del kernel y, por tanto, no accesible desde modo usuario. Esta tabla define cómo se traducen las direcciones virtuales a direcciones físicas y es utilizada por la *Memory Management Unit* (MMU), el componente de hardware encargado de realizar la traducción y aplicar las correspondientes protecciones de memoria. Para cada página se almacena información como el número de *frame*, los permisos asociados y distintos bits de control necesarios para garantizar la seguridad y el aislamiento entre procesos.

Tabla de páginas Multinivel

Las tablas de páginas lineales, donde cada entrada mapea directamente una página virtual a un *frame* físico, presentan importantes limitaciones en sistemas con grandes espacios de direcciones. Por ejemplo, en un sistema de 32 bits con páginas de 4 KB, se necesitarían 2^{20} entradas (aproximadamente 4 millones) para mapear 4 GB de memoria, lo que implicaría un consumo enorme de memoria para la propia tabla de páginas.

Para resolver este problema, los sistemas operativos modernos utilizan tablas de páginas multinivel (*multilevel page tables*). En este esquema, la tabla de páginas se organiza jerárquicamente: cada tabla superior apunta a varias tablas de nivel inferior, que a su vez contienen las entradas finales que mapean las páginas a los *frames* físicos.

Este enfoque permite:

- Reducir el consumo de memoria, ya que solo se crean las tablas de nivel inferior necesarias para las regiones de memoria utilizadas por el proceso.
- Mantener la escalabilidad en espacios de direcciones muy grandes (como 64 bits), evitando que una sola tabla lineal crezca hasta tamaños imprácticos.
- Facilitar la implementación de protección y aislamiento por regiones, ya que cada nivel puede contener información de permisos y control.

En resumen, las tablas de páginas multinivel combinan eficiencia y flexibilidad, siendo esenciales para la gestión de memoria en sistemas modernos.

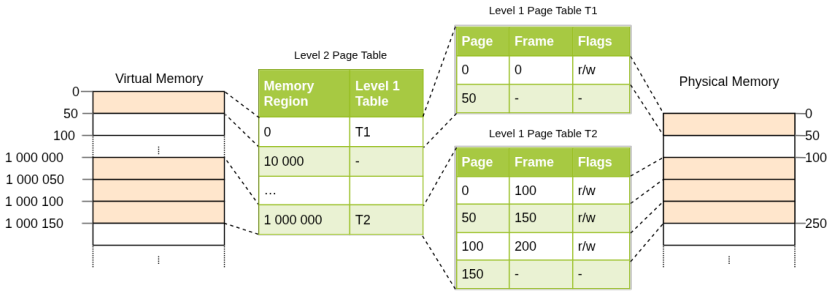


Figura 5.1: Esquema de tablas de páginas multinivel mostrando cómo una dirección virtual se traduce a una dirección física.

5.1.5. Memory Management Unit (MMU)

Sin mecanismos adicionales, la CPU tendría que acceder a la RAM dos veces por cada operación de lectura o escritura: primero para consultar la tabla de páginas y después para acceder a la dirección física resultante. Para evitar este coste, la *Memory Management Unit* (MMU), un componente de hardware integrado en la CPU, se encarga de realizar la traducción de direcciones virtuales a físicas. La MMU mantiene una caché de entradas recientes de la tabla de páginas llamada *Translation Lookaside Buffer* (TLB),

que reduce drásticamente el número de accesos a memoria. Además, la MMU aplica los permisos definidos para cada página, garantizando que un proceso no pueda acceder a regiones de memoria que no le pertenecen.

5.1.6. Acceso a memoria

Cuando un proceso accede a una dirección virtual, la CPU delega en la MMU la traducción de esa dirección a su ubicación física. Para ello, la MMU divide la dirección virtual en dos partes: el *page number*, formado por los bits más significativos, y el *page offset*, compuesto por los bits menos significativos. El *page number* identifica qué página virtual se está usando, mientras que el *offset* indica la posición exacta dentro de esa página.

Por ejemplo, en un sistema con páginas de 4 KB (4096 bytes), el *offset* ocupa los 12 bits menos significativos de la dirección virtual, ya que $2^{12} = 4096$. Los bits restantes corresponden al *page number*, que la MMU utilizará para localizar la página en la tabla y determinar el *frame* físico asociado.

Paginación en x86_64

La arquitectura x86_64 utiliza una **tabla de páginas de 4 niveles** y un tamaño de página de 4 KiB. Cada tabla de páginas, independientemente del nivel, tiene un tamaño fijo de 512 entradas, y cada entrada ocupa 8 bytes, por lo que cada tabla tiene un tamaño total de:

$$512 \text{ entradas} \times 8 \text{ B} = 4 \text{ KiB}$$

Esto permite que cada tabla encaje exactamente en una página de memoria.

Cada índice de tabla utiliza 9 bits, ya que $2^9 = 512$ entradas por tabla. Los 12 bits menos significativos de la dirección virtual corresponden al *offset* dentro de la página ($4 \text{ KiB} = 2^{12}$ bytes).

Los bits del 48 al 63 de la dirección virtual se descartan, lo que implica que, aunque la arquitectura es de 64 bits, en la práctica solo se utilizan **48 bits de dirección**. Sin embargo, estos bits no pueden tener valores arbitrarios: todos los bits en este rango deben ser copias del bit 47 (*sign extension*) para garantizar que las direcciones sean únicas y permitir futuras extensiones, como la introducción de tablas de páginas de 5 niveles.

En un esquema de 4 niveles, el índice de cada nivel se obtiene directamente de la dirección virtual, como se muestra en la Figura 5.2.

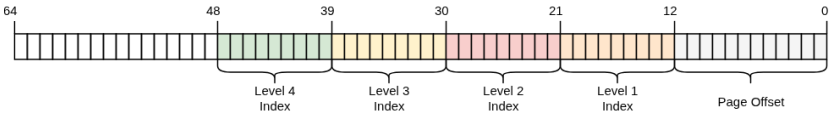


Figura 5.2: Desglose de la dirección virtual en índices de las tablas de páginas de x86_64.

Ejemplo de acceso a memoria: recuperación de instrucciones iniciales

Supongamos que un proceso recién lanzado va a ejecutar sus primeras instrucciones, situadas en la sección de código de su imagen PE.

1. **Dirección virtual del primer instruction pointer (IP/EIP/-RIP):** La CPU obtiene la dirección virtual inicial del registro de instrucción (`EIP/RIP` en x86/x64). Por ejemplo:

$$\text{Dirección virtual inicial} = 0x00400000$$

2. **División de la dirección virtual:** La MMU divide la dirección virtual en varios campos según el esquema de paginación:

   - *Page number*: los bits más significativos, utilizados como índices en la tabla de páginas multinivel (PML4, PDPT, PD, PT).
   - *Page offset*: los 12 bits menos significativos, que indican la posición exacta dentro de la página de 4 KiB.
   - Bits 48-63: copias del bit 47 (*canonical addresses*) para garantizar unicidad y permitir futuras extensiones de la paginación.

   Por ejemplo, en un sistema de 4 KiB:

$$\text{page offset} = 0x000, \quad \text{page number} = 0x00400$$

3. **Consulta de la MMU y TLB:** La MMU primero consulta el *Translation Lookaside Buffer (TLB)*, que almacena traducciones recientes de páginas virtuales a físicas.

   - En un *TLB hit*, se obtiene el *frame* físico directamente.
   - En un *TLB miss*, la MMU recorre la tabla de páginas multinivel en memoria kernel (PML4 $\rightarrow$ PDPT $\rightarrow$ PD $\rightarrow$ PT) para localizar el *frame* físico.

Supongamos que la tabla indica:

$$\text{page number 0x00400} \rightarrow \text{frame físico 0x1A3F0}$$

4. **Cálculo de la dirección física:** La dirección física se obtiene combinando el *frame* físico con el *offset*:

$$\text{Dirección física} = \text{frame} \times 4096 + \text{offset} = 0x1A3F0 \times 4096 + 0x000$$

5. **Acceso a memoria y caché:** La caché, parte del chip de la CPU, es una memoria intermedia rápida que mantiene copias de partes de la RAM (líneas de caché o palabras) que la CPU necesita.

   Cada acceso a memoria que no está en caché (*cache miss*) trae una línea completa desde RAM hacia la caché, aunque la CPU solo necesite unos pocos bytes. La CPU solicita a la RAM la línea de caché que contiene la instrucción. Por ejemplo, si la línea son 64 bytes y la instrucción comienza en la dirección física `0x1A3F012`, se cargan los bytes de `0x1A3F000` a `0x1A3F03F`. Una vez cargada la línea de caché, la CPU selecciona los bytes exactos correspondientes a la instrucción o dato solicitado.

6. **Ejecución:** Las instrucciones recuperadas se cargan en el *instruction decoder* de la CPU, que las decodifica y ejecuta secuencialmente.

5.2. Portable Executable (PE)

El formato *Portable Executable* (PE) es el estándar de Windows para almacenar ejecutables, bibliotecas dinámicas (*DLLs*) y otros tipos de archivos que pueden ser cargados y ejecutados por el sistema operativo. Es el equivalente funcional del formato ELF en Linux y proporciona toda la información necesaria para que el cargador de Windows gestione la ejecución de un programa: estructura de secciones, tablas de importación y exportación, encabezados, y metadatos de la aplicación. Gracias a esta organización, el sistema puede ubicar código y datos en memoria, resolver dependencias dinámicas y garantizar la correcta ejecución de procesos de manera segura y eficiente.

Los PE file es un fichero con valores hexadecimales dispuestos en un layout especifico para poder ser recorrido

Un archivo PE (*Portable Executable*) es un fichero binario cuyos valores están organizados en un diseño específico que permite recorrerlo y analizarlo de manera estructurada. La disposición de los encabezados, secciones y demás campos, incluyendo los offsets estándar de cada componente, se ilustra detalladamente en la infografía disponible en [29].

5.2.1. Headers

En el offset `0x00` se encuentran los encabezados del PE, que contienen información general sobre el ejecutable y permiten al sistema operativo interpretar correctamente el archivo. Entre los campos más relevantes se incluyen:

- **AddressOfEntryPoint:** indica la dirección virtual donde comienza la ejecución del programa dentro de la sección de código principal.
- **Machine:** especifica la arquitectura objetivo del ejecutable (por ejemplo, x86 o x86_64).
- **NumberOfSections:** indica cuántas secciones contiene el PE.
- **DataDirectory:** tabla de referencias a otras estructuras del PE, como la tabla de importaciones, exportaciones, recursos y la tabla de relocaciones.

Estos campos permiten al sistema operativo localizar las distintas secciones del PE y gestionar correctamente la memoria virtual del proceso al cargar el ejecutable. La estructura completa y los offsets de cada campo se pueden consultar en la infografía de referencia [29].

Implementación en C para iterar y recuperar los campos de los encabezados de un archivo PE [30].

5.2.2. Headers Tables

Las cabeceras de un archivo PE contienen varias tablas que son fundamentales para la correcta creación y carga de un proceso a partir de un ejecutable. Estas tablas, que incluyen información sobre exportaciones, importaciones y relocaciones, se encuentran referenciadas en el arreglo *Data Directory* del encabezado opcional del PE.

Para ver cómo recorrer un PE y acceder a cada una de estas tablas, se incluyen ejemplos de código en C en los repositorios de GitHub correspondientes, los cuales se adjuntan en cada sección específica.

Exports

Si el archivo PE corresponde a una biblioteca compartida (*DLL*) y exporta funcionalidad, esta información se almacena en la tabla de exportaciones. A diferencia de Linux, donde las funciones suelen estar disponibles por defecto, en Windows cada símbolo debe exportarse explícitamente para poder ser utilizado por otros módulos.

La tabla de exportaciones es esencial durante la carga de un proceso, ya que el *Windows Loader* la utiliza para resolver las direcciones de las funciones exportadas por cada DLL, garantizando que puedan ser correctamente mapeadas y enlazadas en el espacio de direcciones del proceso que las consume.

Ejemplos de código para analizar y recorrer la tabla de exportaciones de un fichero PE, identificando las funciones exportadas y sus direcciones virtuales, pueden encontrarse en [31].

Además, se incluye una implementación propia de `GetProcAddress` en ensamblador x64, donde se recorre manualmente la tabla de exportaciones del módulo cargado en memoria para localizar la dirección real de una función exportada [32], así como su versión equivalente para arquitecturas x86 [33].

Imports

La tabla de importaciones es fundamental para el *Windows Loader*, ya que contiene información sobre todas las dependencias externas que un ejecutable requiere. Durante la carga del proceso, el loader consulta esta tabla para determinar qué DLLs deben cargarse en memoria y, una vez cargadas, registra las direcciones de memoria de las funciones importadas.

De esta manera, cada llamada a una función importada en tiempo de ejecución puede resolverse inmediatamente usando la dirección ya conocida, evitando la necesidad de búsquedas adicionales. Esto permite que el código realice llamadas directas a las funciones de las bibliotecas externas de forma eficiente y segura.

Se proporciona una implementación en C para recorrer la tabla de importaciones de un PE y listar las librerías y funciones utilizadas [34].

Además, se incluye una implementación propia de un *Windows Loader* simplificado para cargar una DLL, en la que puede observarse explícitamente el proceso de parcheo de la *Import Address Table* (IAT) durante la resolución de dependencias [22].

Relocations

La tabla de *relocations* es fundamental cuando está habilitado el `ASLR` (Address Space Layout Randomization). Si el ejecutable no puede cargarse en su *ImageBase* preferido, el *Windows Loader* debe reubicarlo en una dirección distinta. Para ello, aplica un *delta* de reubicación calculado como:

$$\Delta = \text{ImageBaseReal} - \text{ImageBasePreferida}$$

Este delta debe sumarse a todas las direcciones absolutas del ejecutable, y la lista exacta de ubicaciones que requieren dicho parcheo se encuentra en la sección `.reloc` del PE.

La tabla de relocaciones no es una tabla plana, sino una lista de *bloques*. Cada bloque describe una página de memoria de 4 KB que contiene una o más direcciones que deben ser ajustadas.

Cada bloque inicia con esta cabecera de 8 bytes:

- **VirtualAddress (DWORD)**: dirección base de la página afectada.
- **SizeOfBlock (DWORD)**: tamaño total del bloque; el último bloque puede identificarse porque su valor es 0.

Tras el encabezado de cada bloque comienza un array de valores `WORD` (16 bits), donde cada entrada describe un parche. Cada palabra está empaquetada de la siguiente forma:

- **4 bits superiores (Type)**: indican el tipo de relocalización. Ejemplos:
  - `IMAGE_REL_BASED_DIR64`: punteros absolutos de 64 bits.
  - `IMAGE_REL_BASED_HIGHLOW`: punteros absolutos de 32 bits.

- **12 bits inferiores (Offset)**: desplazamiento dentro de la página.

El uso de un offset de 12 bits se debe a que cada bloque representa exactamente una página de 4 KB. Como una página tiene 4096 bytes y:

$$2^{12} = 4096,$$

los 12 bits inferiores son suficientes para direccionar cualquier byte dentro de dicha página. La dirección final a parchear se obtiene sumando este offset al `VirtualAddress` del bloque.

En conjunto, la tabla de relocaciones actúa como un "mapa de parches" generado por el compilador, que permite al *Windows Loader* ajustar todas las direcciones absolutas del binario cuando se carga en una posición distinta a la prevista originalmente.

Ejemplo de código para iterar la tabla de relocaciones de un PE y analizar los desplazamientos aplicables a las direcciones del ejecutable [35].

5.2.3. Sections

En términos de funcionalidad, las secciones del PE son equivalentes a los segmentos de un archivo ELF en Linux.

Las secciones se encuentran inmediatamente después de los encabezados del PE y están representadas como un conjunto de estructuras, una por cada sección. Cada estructura proporciona información como el offset y el tamaño de la sección, tanto en disco como en memoria. Esta información es utilizada por el *Windows Loader* para definir las distintas regiones de memoria con sus protecciones específicas (lectura, escritura, ejecución).

Las secciones de un proceso en memoria, como `.text`, `.data`, `.reloc`, `.idata` o `.edata`, ya están definidas en el archivo PE. Durante la carga, el loader recorre estas secciones, las mapea a las direcciones que tendrán en memoria y aplica los parcheos necesarios, como completar la IAT con las direcciones reales de las funciones importadas y ajustar cualquier relocación requerida.

En el siguiente código se muestra cómo recorrer las secciones de un archivo PE [30].

En la implementación propia de un *Windows Loader* simplificado [22], también se muestra cómo, desde C, se pueden iterar las secciones del PE, reservar la memoria correspondiente, mapearlas adecuadamente y aplicar los parcheos necesarios.

5.2.4. Ejemplo práctico: parcheo de un PE y modificación del Entry Point

Como cierre a esta sección, se presenta un ejemplo práctico que ilustra cómo modificar un archivo PE para alterar su flujo de ejecución durante la carga. En este proyecto se incluye un script en Python que automatiza el proceso de parcheo de un binario arbitrario, incorporando un *shellcode* desarrollado específicamente para esta prueba. Dicho *shellcode* muestra una ventana mediante la función `MessageBoxA` antes de que el programa continúe ejecutándose con normalidad.

La técnica empleada consiste en insertar el *shellcode* dentro del propio archivo PE, ubicándolo en una zona del fichero que no contenga datos relevantes, y modificar el *Entry Point* para que apunte a este nuevo código en lugar del original. Para mantener el comportamiento legítimo del ejecutable, el *shellcode* incluye un salto de retorno hacia la dirección del *Entry Point* original, de manera que, tras finalizar la ejecución de la rutina inyectada, el programa retoma su flujo normal sin alteraciones perceptibles.

Este ejemplo demuestra de forma práctica cómo las estructuras internas del formato PE pueden manipularse para modificar el inicio de ejecución sin comprometer la funcionalidad del binario, proporcionando un entorno ideal para experimentar con técnicas de parcheo, carga manual y redirección de control.

El script encargado del parcheo puede encontrarse en [36], y el *shellcode* desarrollado específicamente para esta prueba en [37].

5.3. Procesos Windows

Un proceso en Windows representa una entidad de ejecución que actúa como contenedor de los recursos necesarios para que uno o varios hilos puedan ejecutarse. Desde el punto de vista del sistema operativo, un proceso no ejecuta código por sí mismo, sino que proporciona el contexto y los recursos que los hilos utilizan. A continuación se describen sus componentes principales:

- **EPROCESS:** Estructura interna del kernel que representa a un proceso. Contiene información crítica como el identificador del proceso (PID), la lista de hilos asociados, permisos, límites de memoria, una referencia al *Process Environment Block* (PEB), así como metadatos utilizados por el planificador y el gestor de seguridad. Constituye la representación fundamental del proceso en modo kernel.
- **Hilos:** Conjunto de hilos pertenecientes al proceso. Cada hilo es administrado en modo kernel mediante la estructura `ETHREAD`, mientras que en modo usuario dispone de su propio *Thread Environment Block* (TEB).
- **Handles:** Los *handles* son referencias a objetos administrados por el kernel, tales como archivos, sockets, secciones de memoria, procesos o hilos. Funcionan de manera equivalente a los *file descriptors* en Linux, pero con un modelo mucho más generalista. El proceso mantiene una *handle table* que almacena y gestiona estas referencias.
- **Memoria:** Comprende todo el espacio de direcciones virtuales asignado al proceso, incluyendo secciones de código, datos, pilas, *heaps* y regiones mapeadas dinámicamente.
- **Módulos (DLLs):** Lista de módulos y bibliotecas cargadas en el proceso, mantenida en el *PEB Ldr*.

5.3.1. Creación de un proceso (Kernel)

La creación de un proceso en Windows sigue una secuencia estricta de pasos a nivel del kernel.

(1) Inicialización del espacio de direcciones

- **Modelo en Linux vs. Windows:** En Linux, los procesos se crean mediante `fork()`. En Windows el kernel crea un proceso completa-

mente nuevo, partiendo de un espacio de direcciones vacío que debe ser configurado desde el inicio.

- **Mapeo de `KUSER_SHARED_DATA`:** Una de las primeras acciones del kernel es mapear en el nuevo espacio de direcciones una página especial de 4 KB llamada `KUSER_SHARED_DATA`. Esta página está presente simultáneamente en *user mode* y *kernel mode* y permite compartir información sin necesidad de realizar una llamada al sistema. Contiene datos como:
  - El reloj del sistema,
  - Información de versión y arquitectura,
  - Rutas internas del sistema operativo,
  - Indicadores de configuración global.
- **Mapeo inicial del ejecutable:** Antes de cargar `ntdll.dll`, el kernel reserva el espacio de direcciones del proceso y mapea únicamente la sección de código principal (`.text`) del ejecutable en memoria.
- **Mapeo de `ntdll.dll`:** El siguiente módulo en cargarse es `ntdll.dll`. Este componente es esencial, ya que:
  - Contiene las *syscall stubs* que permiten la transición a *kernel mode*;
  - Implementa funciones internas utilizadas por el loader de *user mode*;
  - Cumple un rol análogo a `ld.so` en sistemas Linux.
- **Asignación del PEB (Process Environment Block):** Una vez inicializado el espacio de direcciones, el kernel crea y mapea el `PEB`, una estructura de 1–2 páginas ubicada en *user mode* que describe el estado general del proceso. El `PEB` contiene, entre otros:
  - Variables de entorno,
  - El directorio de trabajo,
  - La lista de módulos cargados (`PEB.Ldr`),
  - Punteros al *heap* y datos relacionados a la pila,
  - La dirección base de la imagen del ejecutable.

(2) Creación del hilo inicial

- **Mapeo del stack:** El kernel reserva y mapea la pila (*stack*) del hilo principal con sus límites superior e inferior, además de aplicar las protecciones apropiadas.
- **Mapeo del TEB (Thread Environment Block):** Para cada hilo se asigna un `TEB`, una estructura de aproximadamente 2 páginas que almacena información específica del hilo.
- **Inicialización del punto de entrada del hilo:** Finalmente, el contador de programa (IP) del nuevo hilo se inicializa en la función `ntdll!LdrInitializeThunk`. Esta rutina es responsable de completar la carga dinámica del proceso en modo usuario, incluyendo:
  - Resolución de importaciones,
  - Ejecución de inicializadores (`TLS callbacks`),
  - La transferencia final al *entry point* del ejecutable.

5.3.2. Process Environment Block (PEB)

El **Process Environment Block (PEB)** es una estructura interna de Windows que almacena información esencial sobre un proceso en ejecución. Aunque Microsoft no publica una definición oficial completa de esta estructura, su diseño es conocido gracias al análisis inverso y puede variar ligeramente entre versiones del sistema operativo. El PEB reside en *user mode* y es único para cada proceso.

Entre los elementos más relevantes del PEB se encuentran:

- **ImageBaseAddress:** Dirección base donde se ha mapeado la imagen principal del proceso.
- **BeingDebugged:** Indicador de si el proceso está siendo depurado.
- **Ldr (PEB_LDR_DATA):** Estructura que mantiene la lista de módulos cargados (DLLs). Recorrer esta lista es un mecanismo ampliamente utilizado por *shellcode* para recuperar la base de `kernel32.dll` y resolver funciones de la WinAPI.

Una explicación detallada y ejemplos de cómo iterar el PEB, tanto desde C como desde WinDbg, pueden encontrarse en [38].

Iterar el PEB para recuperar la base de un módulo

El procedimiento estándar para recorrer el PEB y obtener la base de una DLL concreta es el siguiente:

1. Desde la estructura `_PEB`, acceder al miembro `Ldr`, de tipo `_PEB_LDR_DATA`.
2. Desde `_PEB_LDR_DATA`, acceder a la lista doblemente enlazada `InLoadOrderModuleList`, de tipo `_LIST_ENTRY`.
3. Cada entrada de esta lista corresponde a una estructura `_LDR_DATA_TABLE_ENTRY`.
4. Desde `_LDR_DATA_TABLE_ENTRY`, acceder al campo `BaseDllName`, de tipo `_UNICODE_STRING`, para obtener el nombre del módulo.
5. Si el nombre coincide con el módulo deseado, recuperar el campo `DllBase` desde `_LDR_DATA_TABLE_ENTRY`, que contiene la dirección base cargada en memoria.
6. Si no coincide, avanzar al siguiente elemento de la lista (offset `0x00` de `_LDR_DATA_TABLE_ENTRY`).

Implementaciones prácticas de esta técnica en ensamblador, desarrolladas para este proyecto, están disponibles tanto para **x64** como para **x86**:

- Versión x64: [39]
- Versión x86: [40]

5.3.3. Secciones de un proceso

El espacio de direcciones de un proceso en memoria se organiza en varias secciones, que se enumeran de las direcciones más altas a las más bajas:

Páginas del Kernel Contienen el kernel y ocupan las mismas direcciones en todos los procesos. Como todas apuntan a los mismos *frames* de memoria física, el kernel no se carga repetidamente en cada proceso. Solo se accede a ellas mediante llamadas al sistema.

Stack (RW) Área de almacenamiento *LIFO* que crece hacia abajo, desde el final del espacio de direcciones hacia la BSS. Cada función tiene su propio *stack frame* con variables locales y argumentos. Su tamaño suele estar limitado entre 1 y 8 MB, y su acceso mediante el puntero de pila es muy rápido.

Heap (RW) Espacio para memoria dinámica, gestionada con llamadas como `malloc`. Crece hacia arriba, desde el final de la BSS hasta el inicio del stack. Su administración es más lenta que la del stack.

BSS y Data (RW) Contienen variables globales y estáticas. La sección **BSS** guarda variables no inicializadas, mientras que **Data** almacena variables inicializadas.

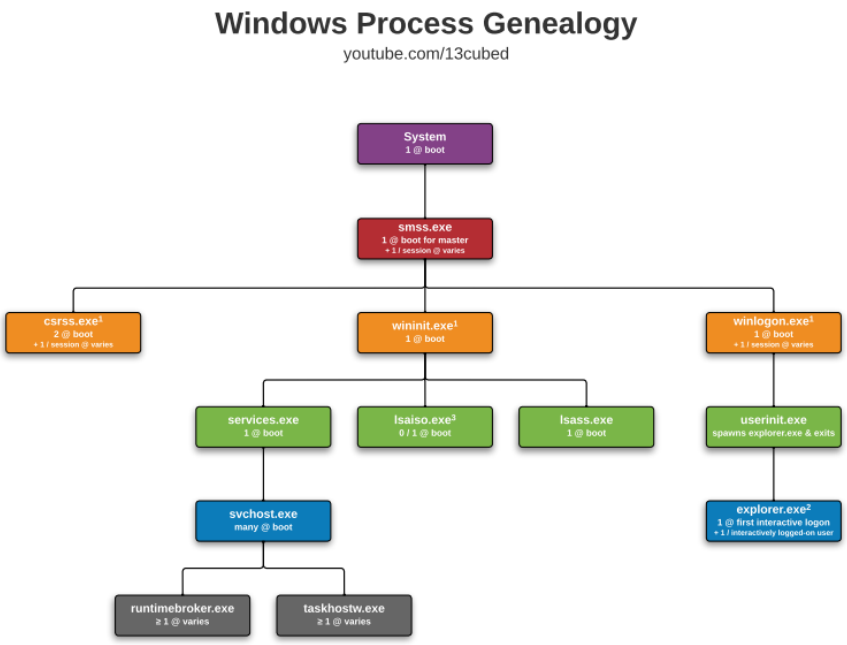
Text (RX) Contiene el código ejecutable del programa, cargado desde el binario en disco. Generalmente es de solo lectura y ejecución.

5.4. Genealogía de Procesos en Windows

Al iniciar, Windows lanza una serie de procesos fundamentales para gestionar todo el sistema operativo, desde la administración de recursos y servicios hasta la interacción con el usuario. Estos procesos forman una jerarquía, donde cada nuevo proceso puede generar otros procesos hijos según las necesidades del sistema o del usuario.

La genealogía de procesos muestra esta estructura desde el arranque del sistema hasta el inicio de sesión del usuario. Comprenderla es clave para análisis forense, respuesta a incidentes y detección de malware, ya que permite identificar comportamientos anómalos en los procesos del sistema.

Una genealogía típica de procesos en Windows se representa a continuación:



Fuente: `youtube.com/13cubed`

Nivel 0: Proceso raíz

- **System**: Primer proceso de espacio de usuario iniciado por el kernel. Tiene un PID fijo (normalmente 4) y no tiene padre.

Nivel 1: Session Manager

- **smss.exe (Session Manager Subsystem)**: Primer proceso real del espacio de usuario. Inicia las sesiones del sistema y lanza procesos críticos como `csrss.exe`, `wininit.exe` y `winlogon.exe`.

Nivel 2: Procesos críticos del sistema

- **csrss.exe**: Gestiona la consola y la creación de procesos. Hay una instancia por sesión.
- **wininit.exe**: Lanza procesos esenciales como `services.exe`, `lsaiso.exe` y `lsass.exe`.
- **winlogon.exe**: Gestiona la autenticación del usuario y permanece activo durante la sesión interactiva.

Nivel 3: Procesos del sistema

- **services.exe**: Service Control Manager. Inicia y gestiona los servicios del sistema, incluidos los alojados por `svchost.exe`.
- **lsaiso.exe**: Proceso de seguridad aislado que implementa funciones de cifrado y autenticación (opcional en versiones modernas de Windows).
- **lsass.exe**: Local Security Authority Subsystem Service. Encargado de políticas de seguridad, autenticación y gestión de credenciales.

Nivel 4: Servicios alojados y utilidades del sistema

- **svchost.exe**: Proceso contenedor que aloja múltiples servicios del sistema. Existen varias instancias según los grupos de servicios.
- **runtimebroker.exe / taskhostw.exe**: Procesos auxiliares para la ejecución de aplicaciones y tareas programadas.

Procesos del usuario interactivo

- **userinit.exe**: Iniciado por `winlogon.exe` tras la autenticación. Lanza el shell principal del usuario y termina.
- **explorer.exe**: Shell gráfico de Windows, representa el escritorio, la barra de tareas y el menú de inicio. Es el proceso raíz del entorno del usuario.

5.5. DLL (Dynamic-Link Library)

La mayoría de los ejecutables actuales dependen de librerías externas, algunas proporcionadas por el sistema operativo, como las funciones de entrada/salida de C (por ejemplo, `printf`, que se encuentra en `msvcrt.dll`), y otras que pueden ser librerías de terceros.

Estas dependencias pueden manejarse de dos formas:

- **Dependencias externas:** cargadas como DLL (*Dynamic-Link Library*) en tiempo de ejecución.
- **Dependencias estáticas:** incluidas directamente en el binario final durante la compilación.

Las DLL son archivos con formato PE, muy similares a los ejecutables. Cada módulo contiene sus propias cabeceras y secciones, tal como se explicó en el apartado de PE files, y se mapean en memoria de manera similar a un ejecutable.

5.5.1. Carga de DLLs

Las DLL se cargan en el espacio de direcciones del proceso de dos formas:

- **Carga estática:** Si el ejecutable indica durante la compilación que depende de una DLL, el *loader* de Windows la cargará al iniciar el programa y resolverá la *Import Address Table (IAT)*.
- **Carga dinámica:** Si la DLL se carga en tiempo de ejecución mediante funciones como `LoadLibraryA`, el *loader* mapea la librería en el espacio de direcciones y resuelve la IAT para el módulo que la cargó.

Una vez cargada, la DLL queda registrada en el *Process Environment Block (PEB)* del proceso. Si, en algún momento posterior, otro módulo intenta cargar la misma DLL, el sistema verifica el PEB y, en lugar de cargarla de nuevo, reutiliza la instancia ya existente en memoria.

Orden de búsqueda de DLLs en Windows

El sistema operativo busca las DLL en el siguiente orden:

1. Directorio desde el cual se cargó la aplicación
2. `C:\Windows\System32`

3. `C:\Windows\System`

4. `C:\Windows`

5. Directorio actual de trabajo (Current Working Directory)

6. Directorios listados en la variable de entorno `PATH` del sistema

7. Directorios listados en la variable de entorno `PATH` del usuario

Este mecanismo permite que un proceso utilice código externo sin incorporarlo de forma estática en su binario, favoreciendo la modularidad y la eficiencia en el uso de memoria.

Para el desarrollo del trabajo se realizó una implementación en C de una versión simplificada del loader de Windows. En la referencia [22] se describen de forma detallada todos los pasos que ejecuta un programa en C para convertir el archivo PE de una DLL en un módulo cargado en memoria y listo para ser utilizado.

5.6. Threads

Los *threads* (hilos) constituyen la unidad básica de ejecución dentro de un proceso. Mientras que un proceso actúa como un contenedor que agrupa recursos (*memory space*, manejadores de objetos, permisos, etc.), un hilo representa el flujo de instrucciones que realmente es ejecutado por la CPU. Un proceso puede contener uno o múltiples hilos, cada uno con su propio estado independiente (contador de programa, conjunto de registros, pila), aunque todos comparten el mismo espacio de direcciones del proceso.

Desde la perspectiva de la ciberseguridad y el desarrollo de malware, el control y manipulación de hilos es especialmente relevante. Muchas *Tactics, Techniques and Procedures* (TTPs) se basan en crear nuevos hilos, modificar los existentes o abusar de ellos para ejecutar *payloads* de forma encubierta, minimizando la generación de artefactos monitorizables por soluciones EDR.

5.6.1. Thread Environment Block (TEB)

El *Thread Environment Block* (TEB) es una estructura interna del sistema operativo Windows que almacena información específica del hilo en ejecución. Está situada en el espacio de usuario y es accesible directamente desde el propio hilo, lo que permite interactuar con ella sin necesidad de realizar llamadas a la API Win32. Entre los elementos más relevantes del TEB se incluyen:

- Un puntero al *Process Environment Block* (PEB).
- Información relacionada con la pila del hilo (dirección base y límite).
- La cadena de manejadores de excepciones estructuradas (SEH).
- El identificador del hilo (*Thread ID*, TID).
- Información sobre la última función Win32 invocada.
- Datos asociados al *Thread Local Storage* (TLS).

El acceso directo al TEB, sin pasar por funciones exportadas, es una técnica frecuente en *shellcodes* y cargas maliciosas debido a que permite evitar *hooks* a nivel de API y reducir la superficie detectable por mecanismos de seguridad.

5.6.2. Walking the TEB

FALTA DESARROLLO

5.7. User Mode y Kernel Mode

El espacio de ejecución en Windows se divide en dos niveles principales: **User Mode** y **Kernel Mode**. Cada uno tiene responsabilidades y privilegios distintos.

En este ejemplo se muestran todas las llamadas que realiza una aplicación para que una función pase de User Mode a Kernel Mode, es decir, el flujo completo desde las librerías de alto nivel hasta la ejecución de operaciones críticas en el núcleo del sistema.

1. **User Mode (U) – Aplicaciones y librerías de alto nivel**
   - `notepad++.exe`: operaciones internas de la aplicación, como: `SetLibraryProperty`.
   - `user32.dll`: funciones de interacción con la interfaz de Windows, por ejemplo `CallWindowProcW` o `IsWindowUnicode`.
   - `KernelBase.dll`: funciones de usuario de alto nivel que abstraen operaciones del sistema, como `WriteFile + 0x8d`.
   - `ntdll.dll`: proporciona la interfaz directa hacia las syscalls, por ejemplo `ZwWriteFile`.
2. **Kernel Mode (K) – Operaciones de bajo nivel y acceso a hardware**
   - `FLTMGR.SYS`: manejo de filtros de sistema de archivos, con funciones como `FltPerformSynchronousIo` y `FltSetCancelCompletion`.
   - `ntoskrnl.exe`: operaciones sobre drivers y dispositivos, incluyendo `IoCallDriver`, `NtDeviceIoControlFile` y `NtWriteFile`.

El flujo típico de ejecución comienza en User Mode, con la aplicación y librerías de alto nivel, y termina en Kernel Mode, donde se realizan operaciones críticas sobre el sistema y el hardware, como la escritura en disco con privilegios elevados. Este ejemplo ilustra cómo cada llamada permite que una función pase de User Mode a Kernel Mode, siguiendo la cadena completa de llamadas hasta el núcleo del sistema.

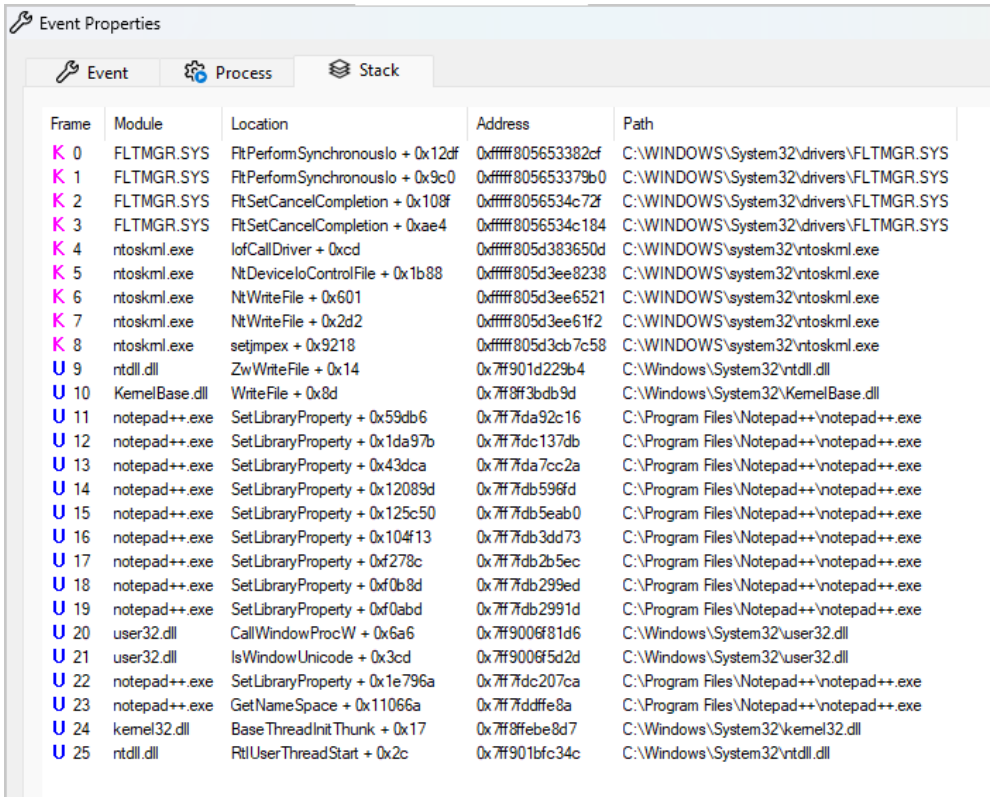


Figura: Captura de la herramienta *Process Monitor (ProMon)* de Sysinternals mostrando el flujo de llamadas de User Mode a Kernel Mode.

Bibliografía

[1] Orange Cyberdefense, "Goad-light laboratory documentation," https://orange-cyberdefense.github.io/GOAD/labs/GOAD-Light/, 2023, accedido el 25/11/2025.

[2] "Kape targets documentation," https://ericzimmerman.github.io/KapeDocs/#!Pages/2.1-Targets.md, accedido el 2025-01-01.

[3] "Kape target: !sans_triage," https://github.com/EricZimmerman/KapeFiles/blob/master/Targets/Compound/!SANS_Triage.tkape, accedido el 26 de noviembre de 2025.

[4] "Kape target: Eventlogs," https://github.com/EricZimmerman/KapeFiles/blob/master/Targets/Windows/EventLogs.tkape, accedido el 26 de noviembre de 2025.

[5] MITRE Center for Threat-Informed Defense, "Adversary emulation library," https://github.com/center-for-threat-informed-defense/adversary_emulation_library, 2025, repositorio oficial del MITRE Center for Threat-Informed Defense.

[6] nyascla, "Reverse shell badusb payload," https://github.com/nyascla/mal-dev-lab/tree/main/red-team-ops/flipper-zero/bad-usb/reverse-shell-disk, 2024, accedido el 25/11/2025.

[7] Nyascla, "Amsi bypass via memory patching," https://github.com/nyascla/mal-dev-lab/blob/main/EDR-bypass/AMSI-bypass/amsi-patching.ps1, 2024, accedido: 2025-11-25.

[8] ——, "Amsi bypass via reflection," https://github.com/nyascla/mal-dev-lab/blob/main/EDR-bypass/AMSI-bypass/amsi-patching-reflection.ps1, 2024, accedido: 2025-11-25.

[9] nyascla, "x64 get module handle (runtime linking)," https://github.com/nyascla/mal-dev-lab/blob/main/101-code/ASM-shellcodes/runtime_linking/x64/x64_get_module_handle.asm, accessed: 2025-11-25.

[10] ——, "x86 get module handle (runtime linking)," https://github.com/nyascla/mal-dev-lab/blob/main/101-code/ASM-shellcodes/runtime_linking/x86/get_module_handle.asm, accessed: 2025-11-25.

[11] ——, "x64 get proc address (runtime linking)," https://github.com/nyascla/mal-dev-lab/blob/main/101-code/ASM-shellcodes/runtime_linking/x64/x64_get_proc_address.asm, accessed: 2025-11-25.

[12] ——, "x86 get proc address (runtime linking)," https://github.com/nyascla/mal-dev-lab/blob/main/101-code/ASM-shellcodes/runtime_linking/x86/get_proc_address.asm, accessed: 2025-11-25.

[13] ——, "x64 rol/xor hashing routine," https://github.com/nyascla/mal-dev-lab/blob/main/101-code/ASM-shellcodes/hashing/x64/x64_rol_xor.asm, accessed: 2025-11-25.

[14] ——, "x86 rol/xor hashing routine," https://github.com/nyascla/mal-dev-lab/blob/main/101-code/ASM-shellcodes/hashing/x86/rol_xor.asm, accessed: 2025-11-25.

[15] ——, "Messagebox shellcode (x64)," https://github.com/nyascla/mal-dev-lab/tree/main/101-code/ASM-shellcodes/messagebox/x64/messagebox.asm, 2024.

[16] ——, "Messagebox shellcode (x86)," https://github.com/nyascla/mal-dev-lab/tree/main/101-code/ASM-shellcodes/messagebox/x64/messagebox.asm, 2024.

[17] jstrosch, "sclauncher: Minimal pe generator," https://github.com/jstrosch/sclauncher, 2021.

[18] nyascla, "Reverse shell shellcode (x86)," https://github.com/nyascla/mal-dev-lab/blob/main/101-code/ASM-shellcodes/reverse_shell/x86/reverse_shell.asm, 2024.

[19] H. Project. (2024) calibre-launcher.dll - hijacklibs. Accedido el 26 de noviembre de 2025. [Online]. Available: https://hijacklibs.net/entries/3rd_party/calibre/calibre-launcher.html

[20] Nyascla, "Script para generación automática de ficheros .def," https://github.com/nyascla/mal-dev-lab/blob/main/code-injection/Dll-hijacking/dll-proxying/def-file/exports.py, accedido: 2025-11-26.

[21] ——, "Demo de *Side Loading + DLL Proxying*," https://github.com/nyascla/mal-dev-lab/tree/main/code-injection/Dll-hijacking/dll-proxying, accedido: 2025-11-26.

[22] ——, "Manual dll loader (no aslr) — proof of concept," https://github.com/nyascla/mal-dev-lab/blob/main/101-windows/walking_load_dll/main-loader-NOASLR.c, 2024, accedido: 2025-02-27.

[23] ——, "Pe format layout infographic," https://github.com/nyascla/mal-dev-lab/blob/main/101-windows/walking_PE/utils/PE%20Format%20Layout.pdf, accessed: 2025-02-26.

[24] ——, "Win32 api syscall research," https://github.com/nyascla/mal-dev-lab/tree/main/syscalls/wwin32-api, 2024.

[25] ——, "Native api syscall research," https://github.com/nyascla/mal-dev-lab/tree/main/syscalls/native-api, 2024.

[26] ——, "Direct syscalls techniques," https://github.com/nyascla/mal-dev-lab/tree/main/syscalls/direct-syscalls, 2024.

[27] ——, "Indirect syscalls techniques," https://github.com/nyascla/mal-dev-lab/tree/main/syscalls/indirect-syscalls, 2024.

[28] P. Opp, "Paging introduction," https://os.phil-opp.com/paging-introduction/, consultado el 30 de noviembre de 2025.

[29] OpenRCE, "Pe format layout: Portable executable (pe) standard offsets," PDF file, 2025, infografía de los offsets del estándar PE, disponible públicamente en GitHub. [Online]. Available: https://github.com/nyascla/mal-dev-lab/blob/main/101-windows/walking_PE/utils/PE%20Format%20Layout.pdf

[30] nyascla, "walking_pe — cabezas de pe: recorrido de headers en c," Repositorio GitHub, accedido el 30 de noviembre de 2025. [Online]. Available: https://github.com/nyascla/mal-dev-lab/tree/main/101-windows/walking_PE

[31] ——, "walking_pe_tables — ejemplo de recorrido de exports en pe," Repositorio GitHub, accedido el 30 de noviembre de 2025. [Online]. Available: https://github.com/nyascla/mal-dev-lab/tree/main/101-windows/walking_PE_tables/exports

[32] Nyascla, "Custom getprocaddress implementation in x64 assembly," https://github.com/nyascla/mal-dev-lab/blob/main/101-code/ASM-shellcodes/runtime_linking/x64/x64_get_proc_address.asm, 2025, accessed: 2025-02-28.

[33] ——, "Custom getprocaddress implementation in x86 assembly," https://github.com/nyascla/mal-dev-lab/blob/main/101-code/ASM-shellcodes/runtime_linking/x86/get_proc_address.asm, 2025, accessed: 2025-02-28.

[34] nyascla, "walking_pe_tables — ejemplo de recorrido de imports en pe," Repositorio GitHub, accedido el 30 de noviembre de 2025. [Online]. Available: https://github.com/nyascla/mal-dev-lab/tree/main/101-windows/walking_PE_tables/import

[35] ——, "walking_pe_tables — ejemplo de recorrido de relocaciones en pe," Repositorio GitHub, accedido el 30 de noviembre de 2025. [Online]. Available: https://github.com/nyascla/mal-dev-lab/tree/main/101-windows/walking_PE_tables/relocs

[36] Nyascla, "Binary patching: Entry point modification," https://github.com/nyascla/mal-dev-lab/tree/main/binary-patching/entrypoint, 2025, accedido en 2025.

[37] ——, "Custom x86 messageboxa shellcode for pe patching," https://github.com/nyascla/mal-dev-lab/blob/main/101-code/ASM-shellcodes/messagebox/x86/messagebox_patch.asm, 2025, accedido en 2025.

[38] ——, "Walking the peb in windows," https://github.com/nyascla/mal-dev-lab/tree/main/101-windows/walking_PEB, 2024, accessed: 2025-02-01.

[39] ——, "Runtime linking in x64: Getmodulehandle implementation," https://github.com/nyascla/mal-dev-lab/blob/main/101-code/ASM-shellcodes/runtime_linking/x64/x64_get_module_handle.asm, 2024, accessed: 2025-02-01.

[40] ——, "Runtime linking in x86: Getmodulehandle implementation," https://github.com/nyascla/mal-dev-lab/blob/main/101-code/ASM-shellcodes/runtime_linking/x86/get_module_handle.asm, 2024, accessed: 2025-02-01.